Software Development Kit for Multicore Acceleration
Version 3.1

Data Communication and Synchronization Library Programmer's Guide and API Reference

SC33-8408-01

Software Development Kit for Multicore Acceleration
Version 3.1

Data Communication and Synchronization Library Programmer's Guide and API Reference

SC33-8408-01

Note

Before using this information and the product it supports, read the information in "Notices" on page 209.

Edition notice

This edition applies to version 4.0.0 of the *Data Communication and Synchronization Library* for the Software Development Kit for Multicore Acceleration Version 3.1 (program number 5724-S84) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this publication

This programmer's guide provides detailed information regarding the use of the Data Communication and Synchronization for Cell/B.E. and Hybrid library APIs. It contains an overview of the Data Communication and Synchronization library, detailed reference information about the APIs, and usage information for programming with the APIs.

For information about the accessibility features of this product, see Appendix H, "Accessibility features," on page 207.

Who should use this book

This book is intended for use by accelerated library developers.

What's new in this release

Features and improvements

- New DaCS support for the 32-bit environment
- New DaCS support for Fortran bindings
- New DaCS support for PDT trace hooks
- New DaCS example code to demonstrate the use of DaCS interfaces in both C and Fortran
- DaCS performance on Cell/B.E. is improved and the SPU library size is reduced
- DaCS for Hybrid support is upgraded from prototype to beta level
- Improved debug tracing and new returned error codes
- Added support for management of the local memory regions

New DaCS APIs

The following APIs have been added to the DaCS library:

API	**Functionality**
dacs_mem_create	Designates a region in the memory space of the current process for use by DMA services.
dacs_mem_share	Passes a memory handle from the current process to a remote process.
dacs_mem_accept	Accepts a memory handle from a remote process.
dacs_mem_release	Releases a previously-accepted memory handle.
dacs_mem_register	Registers this memory region to be used as a local memory handle on DMA operations.
dacs_mem_deregister	Deregisters memory access for a local region.
dacs_mem_destroy	Invalidates access to the specified memory region.
dacs_mem_put	Initiates a data copy from a local memory region into a remote memory region.
dacs_mem_get	Initiates a data copy from a remote memory region into a local memory region.

API	Functionality
dacs_mem_get_list	Initiates a scatter or gather data transfer from a remote memory region into a local memory region.
dacs_mem_put_list	Initiates a scatter or gather data transfer from a local memory region into a remote memory region.
dacs_mem_limits_query	Queries the limits on memory regions for communications with a specific DaCS Element process identifier.
dacs_mem_query	Queries the attributes of a memory region.
dacs_de_kill (prototype only)	Requests the termination of the specified AE process, identified by its DE and PID.
dacsf_makeptr	**Fortran only**: converts a `dacs_pvoid_t` handle to an address or Fortran pointer.
dacsf_makevoid	**Fortran only**: converts a Fortran pointer or address to a `dacs_pvoid_t` handle.

Deprecated DaCS APIs

The following APIs have been deprecated:

API	Replacement
dacs_runtime_init	dacs_init may be used instead.
dacs_runtime_exit	dacs_exit may be used instead.

Consolidated DaCS documentation

The new *Data Communication and Synchronization Library Programmer's Guide and API Reference* combines two guides from the previous SDK 3.0 Release These guides were:

- *Data Communication and Synchronization for Cell/B.E. Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Hybrid-x86. Programmer's Guide and API Reference*

The new manual provides a standalone guide to implementing the DaCS library on both Cell/B.E. and Hybrid platforms.

Added Fortran support

The DaCS library now supports Fortran. Usage details have been added to each API reference. In addition, there are two new Fortran-specific APIs as listed in the table below:

API	Functionality
dacsf_makeptr	Converts a `dacs_pvoid_t` handle to an address or Fortran pointer.
dacsf_makevoid	Converts a Fortran pointer or address to a `dacs_pvoid_t` handle.

Conventions

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
Bold	Lowercase commands, library functions.	**void sscal_spu (float *sx, float sa, int n)**
Italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	The following example shows how a test program, *test_name* can be run
`Monospace`	Examples of program code or command strings.	`int main()`

Document conventions

Throughout the DaCS API documentation, there are references to specific DaCS implementations and programming languages. The following table describes these references.

Table 2. Document conventions

Convention	Indicates	Example
Hybrid	The information in the section that follows applies to DaCS for Hybrid only.	**Hybrid**: the mailbox depth is limited to 32 incoming and outgoing mailboxes for each PID.
Cell/B.E.	The information in the section that follows applies to DaCS for Cell/B.E. only.	**Cell/B.E.**: the mailbox depth is limited to 4 incoming and 4 outgoing mailboxes for each SPU.
C	The information in the section that follows applies to the C programming language only.	**C**: a pointer to the message received.
Fortran	The information in the section that follows applies to the Fortran programming language only.	**Fortran**: the message received.
Fortran only	The preceding parameter is only available to the Fortran programming language.	**Fortran only**: the number of elements in the `envv` array.

Related information

See "Related documentation" on page 213.

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this publication, send your comments using Resource Link™ at http://www.ibm.com/servers/resourcelink. Click **Feedback** on the navigation pane. Be sure to include the name of the book, the form number of the book, and the specific location of the text you are commenting on (for example, a page number or table number).

Part 1. Overview of DaCS

The Data Communication and Synchronization (DaCS) library provides a set of services which ease the development of applications and application frameworks in a heterogeneous multi-tiered system, for example a 64 bit x86 system (x86_64) and one or more Cell/B.E. systems. The DaCS services are implemented as a set of APIs providing an architecturally neutral layer for application developers on a variety of multi-core systems. One of the key features that further differentiates DaCS from other programming frameworks is a hierarchical topology of processing elements, each referred to as a *DaCS Element* (DE). Within the hierarchy each DE can serve one or both of the following roles:

- A general purpose processing element, acting as a supervisor, control or master processor. This type of element usually runs a full operating system and manages jobs running on other DEs. This is referred to as a *Host Element* (HE).
- A general or special purpose processing element running tasks assigned by an HE. This is referred to as an *Accelerator Element* (AE).

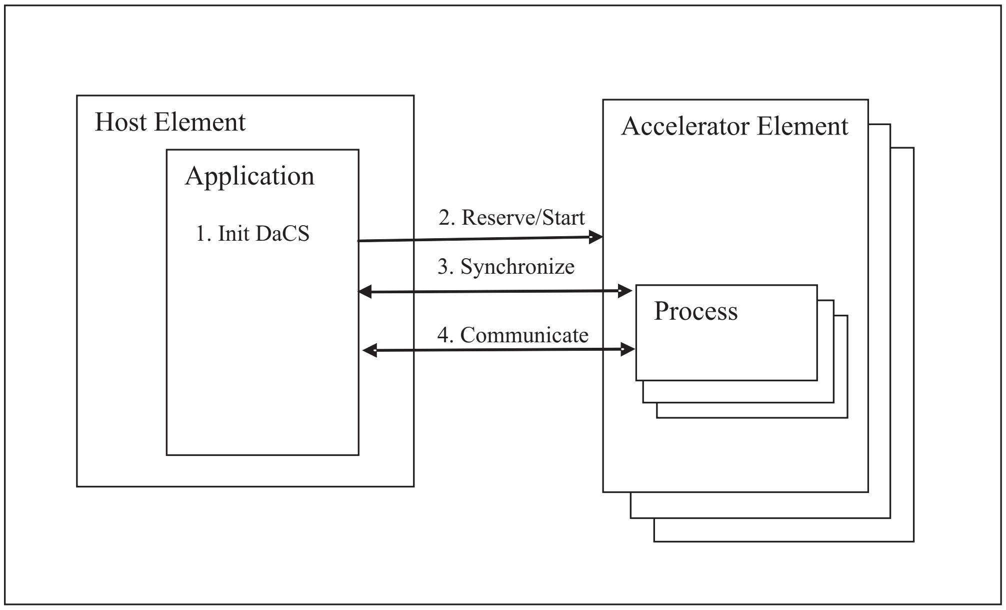


Figure 1. Example interaction between Host and Accelerator elements

The above figure demonstrates the relationship between the HE and AE in DaCS. An application is first started on the HE which uses the AEs to offload an accelerated workload.

1. Prior to using DaCS, the HE application must first initialize DaCS. See Chapter 7, “Initialization and termination,” on page 33 for further information.
2. Once initialized, the desired number of AEs may be reserved and programs started on them. See Chapter 8, “Reservation services,” on page 41 and Chapter 9, “Process management,” on page 49 for further information.
3. Processes may be controlled through provided synchronization primitives. See “Process synchronization” on page 65 for further information.
4. Now that the AE is up and running, it may communicate with the HE. Chapter 10, “Data communication,” on page 89 for further information.

Chapter 1. Services

The DaCS services can be divided into the following categories:

Resource Management

DaCS provides services for managing the hardware compute resources available to DaCS.

Resource reservation services are included for allocation, de-allocation and querying of system wide compute resources. These resource reservation services allow an HE to reserve AE resources below itself in the hierarchy. The APIs abstract the specifics of the reservation system (O/S, middleware, etc.) to allocate resources for an HE. Once reserved, the AEs can be used by the HE to execute tasks for accelerated applications.

Process Management

DaCS provides services for managing and controlling DaCS processes.

In conjunction with the reserved resources, process management services are also provided as a means for HE initiation and management of accelerated applications on AEs. These services include, but are not limited to, remote process launch and remote error notification.

Data communication

DaCS provides several mechanisms for managing, synchronizing, and sharing data through remote DMA, message passing, and mailbox services.

The DMA services provided offer a means to create, share, and transfer data to or from a local or remote memory region. Transfers are performed using a one-sided put/get remote direct memory access remote DMA model. In addition, the ability to perform scatter/gather list operations is also available and provides optional enforcement of ordering for the data transfers.

Message passing services provide the means for passing asynchronous messages using a two-sided send/receive model. Messages are passed point-to-point from one process to another.

Mailbox services provide a simple interface for synchronous transfer of small (32-bit) messages from one process to another.

Wait identifier data synchronization services are also provided to compliment the DMA and message passing services.

Error Handling

The error handling services enable the user to register error handlers and gather error information.

Chapter 2. DaCS implementations

This section discusses the different implementations of DaCS.

Two implementations of the DaCS API are currently supported. The first implementation, called DaCS for Cell/B.E., is support for the DaCS architecture on the Cell Broadband Engine Architecture (CBEA). The other implementation, called DaCS for Hybrid, is support of DaCS on a hybrid x86_64 / PowerPC Linux system. The abovementioned implementations can operate independently of one another or they can coexist. For example, with the current implementations the PPE can act as both an HE and an AE. When working with the SPEs (via DaCS for Cell) it is an HE, and when working with the x86_64 system (via DaCS for Hybrid) it is a AE. This is discussed in "Coexistence" on page 8.

DaCS for Cell/B.E.

DaCS for Cell/B.E. is an implementation of the DaCS API specification for the CBEA. DaCS for Cell/B.E. uses the PowerPC Processor Elements (PPE) as the HE and the Synergistic Processor Elements (SPE) as AEs. Within a Cell/B.E. blade server, there are two Cell/B.E.s, each containing a single PPE and eight SPEs. This is shown in the Figure below.

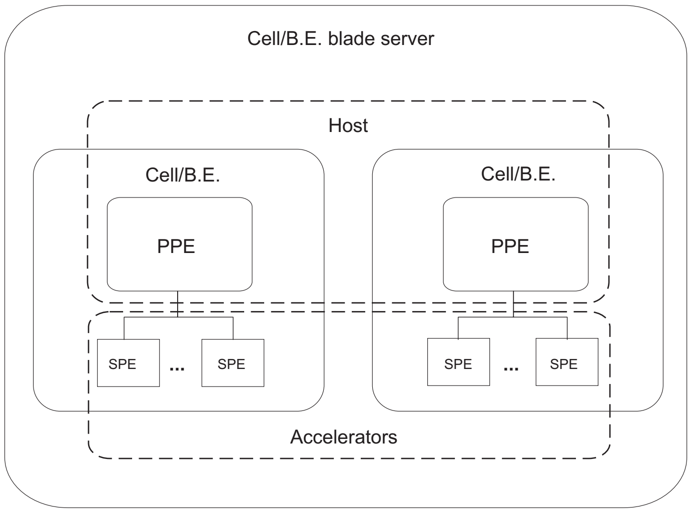


Figure 2. Example PPE Host with SPE Accelerators

DaCS for Hybrid

DaCS for Hybrid is an implementation of the DaCS API specification which supports the connection of an x86_64 system to one or more PPEs. In this environment, the x86_64 processor acts as the HE, whereas the PPEs act as AEs.

In DaCS 4.0, DaCS for Hybrid supports connecting the HE and AEs with sockets or PCIe over Axon. DaCS for Hybrid provides the x86_64 system access to the PPE. It allows a program to be started and stopped and data transfer between the HE and AEs. Direct access to the Synergistic Processor Elements (SPEs), from the x86_64 system is not provided. Instead, the SPEs can be accessed by the program running on the PPE. This is shown in the figure below.

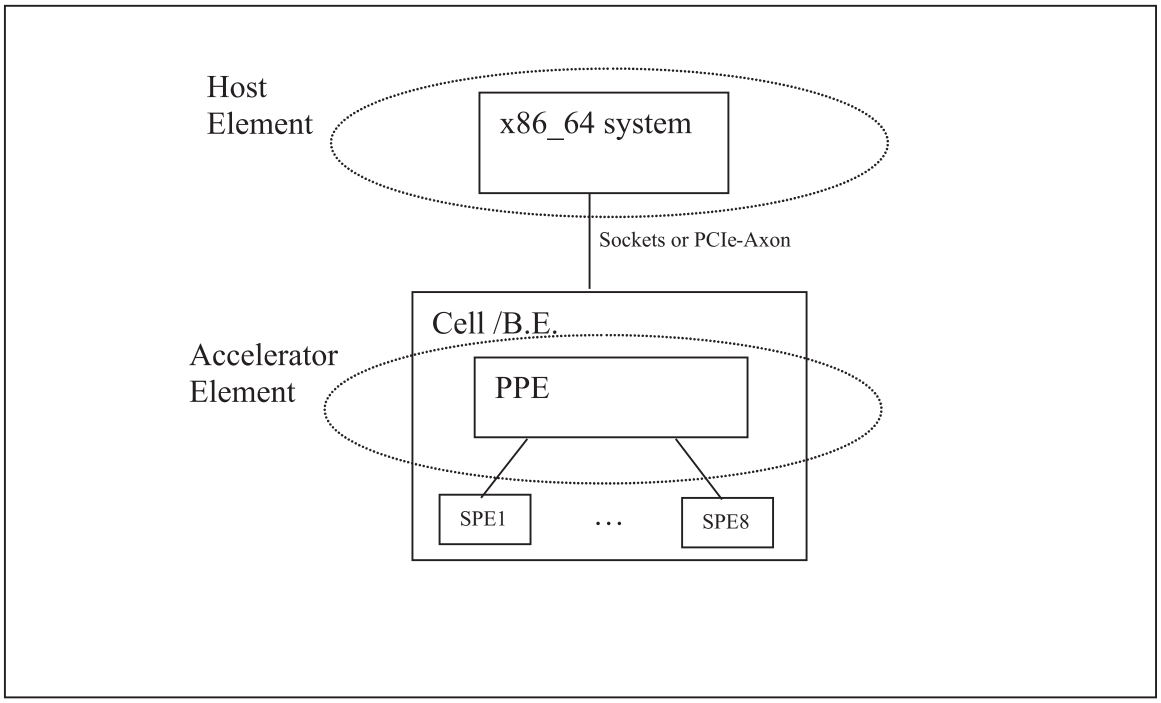


Figure 3. Example x86_64 Host with PPE Accelerators

In order to manage the interactions between both the HE and the AEs DaCS for Hybrid starts a service on each of them. On the system where the HE will run the service is the Host DaCS daemon (hdacsd) and on the AE the service is the Accelerator DaCS daemon (adacsd). These services are shared between all DaCS for Hybrid processes for an operating system image. For example, if the x86_64 system has multiple cores that each run as an HE, only a single instance of the `hdacsd` service is needed to manage the interactions of each of the host applications with their AEs. Similarly, on the AE, if the Cell/B.E. is on a Cell/B.E. blade server (which has two Cell/B.E.s), a single instance of the `adacsd` service is needed to manage both of the Cell/B.E.s acting as AEs, even if they are used by different HEs.

When a host application starts using DaCS for Hybrid it connects to the `hdacsd` service. This service manages the system topology from a DaCS perspective (managing reservations) and starts the accelerator application on the AE. Only process management requests will use the `hdacsd` and `adacsd` services. All other interactions between the host and accelerator application will flow via a direct connection. The following diagram provides a summary of the DaCS for Hybrid components and their relationships:

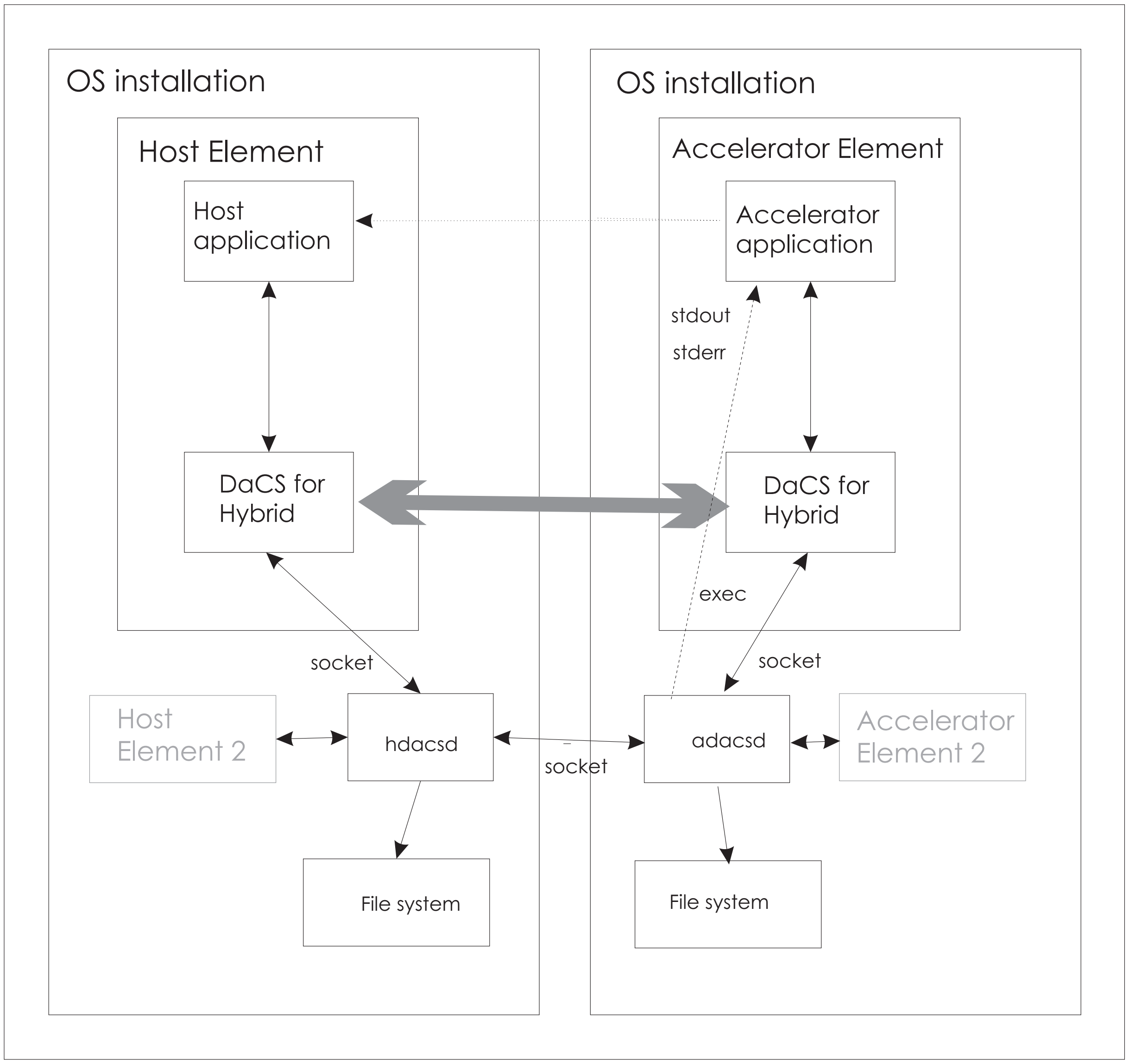


Figure 4. Summary of DaCS for Hybrid components

The figure above summarizes the parts of DaCS for Hybrid and their relationship to each other. The Host Application, which is running on the HE, is started outside the scope of DaCS. It is the application that is using DaCS (as an HE) so it can use one or more AEs. The line from the Host Application to the DaCS for Hybrid box represents the use of the DaCS for Hybrid library. The DaCS for Hybrid library uses the `hdacsd` for process management and uses a direct connection between the applications (via the DaCS for Hybrid library on each side) for all other communications.

The process management communications between the `hdacsd` and `adacsd` is represented by the arrow connecting them. This link is used for starting Accelerator Applications on the AE and communicating status and connection setup information. The `hdacsd` and `adacsd` use the file system for configuration information (discussed in Part 2, “Installing and configuring DaCS,” on page 11) and to optionally retrieve the executable for the Accelerator Application. Starting applications on the AE is discussed in “dacs_de_start” on page 54.

When the Accelerator Application is started, `stdout` and `stderr` are redirected back to the host application, which consolidates the output on the host application. Figure 4 also shows the OS Installation because this is the scope of the daemon. It manages all of the HEs or AEs that are within that scope. This is represented by the Host Element 2 and Accelerator Element 2 boxes.

Coexistence

DaCS for Hybrid and DaCS for Cell/B.E. can work together. To use them both you must install both. See the *SDK 3.1 Installation Guide* for further information.

When using both implementations of DaCS, the PPE acts as both an AE to the x86_64 host and an HE to the SPE accelerators. The DaCS library automatically directs incoming PPE requests to the appropriate HE or AE based on the supplied target DE and PID. For example, when a PPE application calls `dacs_send()`, the supplied target DE and PID are used to determine whether the message should be sent to the x86_64 (via DaCS for Hybrid) or to an SPE (via DaCS for Cell).

In a combined system, shared resources may be created on the PPE and shared with the x86_64 HE and SPE AEs. This provides a unified mechanism for data sharing and synchronization.

For example, mutexes and memory regions created on the PPE can be shared across all three DE types (x86_64, PPE and SPE) in the combined system. This means that data may be shared across the entire system with synchronized access.

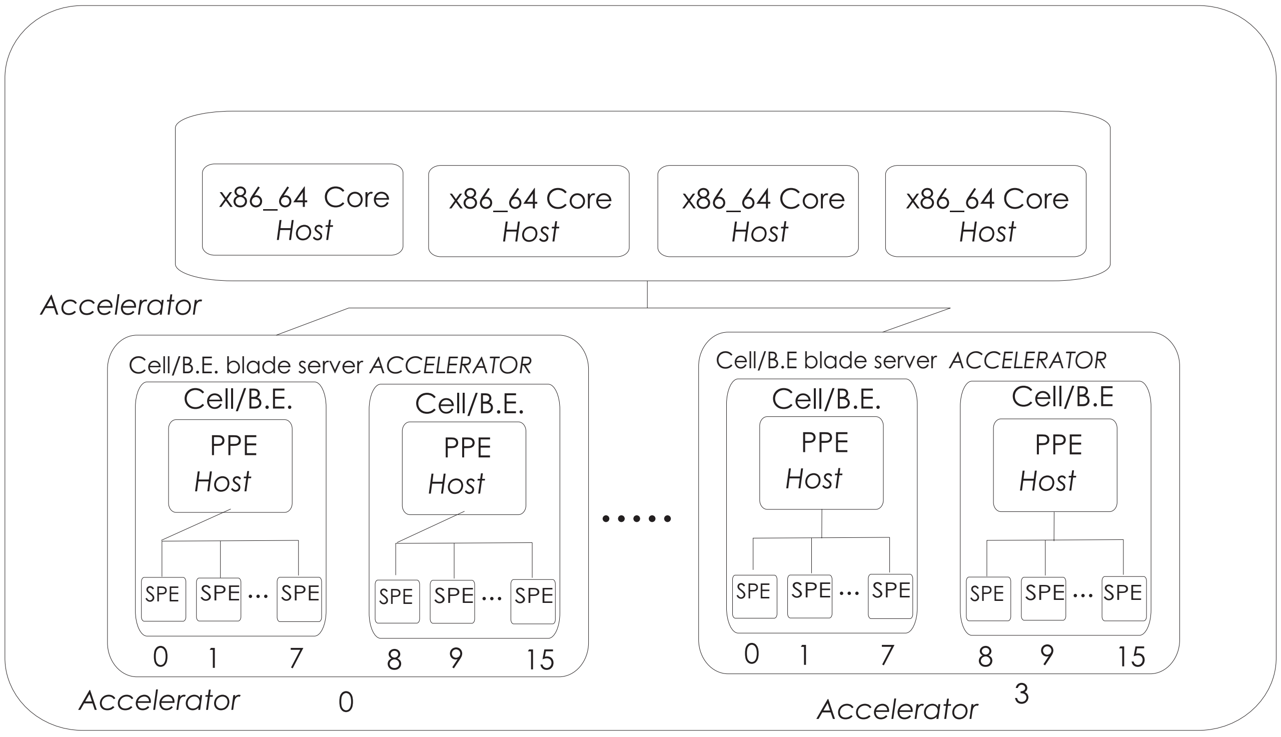


Figure 5. Example node with AMD Host and Cell/B.E. accelerators

Chapter 3. Specified operating environments

DaCS is supported on multiple platforms.

Support for the DaCS API library depends upon your implementation. The following two implementations of the DaCS API are currently supported:

- DaCS for Cell/B.E.
- DaCS for Hybrid

Note: The x86_64 portion of DaCS for Hybrid is only supported on AMD Opteron Processors.

DaCS for Cell/B.E. is provided with the SDK Product package. DaCS for Hybrid is provided with the SDK Extras package. For more information on these SDK packages, see Cell Broadband Engine Resource Center - downloads (http://www-128.ibm.com/developerworks/power/cell/downloads.html) on the IBM DeveloperWorks Web site.

For more information on these implementations, see Chapter 2, “DaCS implementations,” on page 5.

For more information on the hardware and software that supports the SDK, see Specified operating environments.

Supported DaCS implementations

The libraries for DaCS 4.0 have various levels of support, as documented in this section.

The *SDK for Multicore Acceleration Version 3.1* is available in three different package types: Developer, Product, and Extras. Full details of what these packages are and what support is available may be found at the IBM developerWorks Web site at: http://www-128.ibm.com/developerworks/power/cell/.

DaCS for Cell/B.E. support information
This is a component of the SDK Product package. The DaCS for Cell/B.E. APIs are production components and therefore are fully supported and warranted. These components are stable, with upgrade paths or backwards compatibility maintained between releases.

DaCS for Hybrid support information
This is a component of the SDK Extras package. This package is not officially supported by IBM, but limited assistance is available on the Cell/B.E. forum on developerWorks.

Prototype interfaces and components
These are components of the SDK Extras package. Limited assistance is available on the Cell/B.E. forum on developerWorks. These interfaces may not be supported by all implementations. For example, the `dacs_de_kill()` function is currently only supported by DaCS for Hybrid.

Supported compilers for the DaCS Fortran bindings

For the DaCS Fortran Bindings, the following table identifies which Fortran compilers have been tested by IBM. Not applicable indicates that a compiler does not support the architecture of the DaCS implementation specified.

	Hybrid	**PPU**	**SPU**
GNU Fortran	Yes	Yes	Yes
XLF	No	Yes	Yes
Path Scale	Yes	Not applicable	Not applicable

Note: The Fortran Bindings may work with other Fortran compilers such as PGI Fortran and INTEL Fortran in the Cell Hybrid implementation but they have not been tested.

Part 2. Installing and configuring DaCS

The DaCS libraries are part of the Cell Broadband Engine Software Development Kit (SDK) for Multicore Acceleration. The DaCS for Cell/B.E. components are installed in the default group of the SDK, while the DaCS for Hybrid components are an optional install. After installation of the DaCS for Hybrid components, some configuration is needed and is discussed in this section.

- You may need to set up library path information (by using `LD_LIBRARY_PATH` or `ldconfig`) or you may instead use RPATH information when linking DaCS applications.
- You will need to configure daemons on the host and start them on the host and accelerators.

Chapter 4. Installing DaCS

Several packages are available that provide the means to develop, deploy and debug DaCS applications on your x86_64 and Cell/B.E. and Hybrid system. The 32–bit and 64–bit packages can either be installed separately or together depending on the applications that the user will be running.

In the tables that follow:
- **ppc** refers to 32 bit PPU binaries
- **ppc64** refers to 64 bit PPU binaries
- **x86_64** refers to 64 bit binaries
- **i686** refers to 32 bit binaries

The DaCS runtime packages need to be installed on the target platform in order be able to run DaCS applications. The development packages are needed if DaCS development is going to be done. If the desired packages are not installed by default, refer to the *SDK Installation Guide* for instructions on how to install the other packages.

The following table shows the package names with a short description:

Table 3. SDK packages for DaCS in a Cell/B.E. environment.

Package	Description
dacs-4.0.0-#.ppc.rpm	DaCS Runtime - Contains the optimized PPU shared library.
dacs-4.0.0-#.ppc64.rpm	
dacs-devel-4.0.0-#.ppc.rpm	DaCS Development - Contains the header files, optimized static PPU and SPU libraries, and debug libraries (static PPU and SPU; shared PPU).
dacs-devel-4.0.0-#.ppc64.rpm	
dacs-trace-4.0.0-#.ppc.rpm	DaCS Trace Enabled Runtime - Contains the trace enabled PPU shared library.
dacs-trace-4.0.0-#.ppc64.rpm	
dacs-trace-devel-4.0.0-#.ppc.rpm	DaCS Trace Enabled Development - Contains the trace-enabled PPU and SPU static libraries.
dacs-trace-devel-4.0.0-#.ppc64.rpm	
dacs-cross-devel-4.0.0-#.noarch.rpm	DaCS Cross Development - Contains the header files and libraries needed for cross-architecture development.
dacs-tools-4.0.0-#.ppc64.rpm	DaCS tools. Contains tools for the diagnosis of problems.
dacs-compat-4.0.0-19.ppc64.rpm	Contains the optimized 64 bit PPU shared library supporting the DaCS 3.0 API
dacs-examples-source-4.0.0-#.noarch.rpm	DaCS Examples - Contains samples that demonstrate how to use the DaCS APIs.

Table 4. SDK packages for DaCS in a hybrid environment

Package	Description
dacs-hybrid-4.0.0-#.i686.rpm	DaCS for Hybrid Runtime - Contains the optimized shared library for the target platform.
dacs-hybrid-4.0.0-#.x86_64.rpm	
dacs-hybrid-4.0.0-#.ppc.rpm	
dacs-hybrid-4.0.0-#.ppc64.rpm	
dacs-hybrid-devel-4.0.0-#.i686.rpm	DaCS for Hybrid Development - Contains the header files, optimized static libraries, and debug libraries (static and shared) for the target platform.
dacs-hybrid-devel-4.0.0#-#.x86_64.rpm	
dacs-hybrid-devel-4.0.0-#.ppc.rpm	
dacs-hybrid-devel-4.0.0-#.ppc64.rpm	
dacs-hybrid-trace-4.0.0-#.i686.rpm	DaCS for Hybrid Trace Enabled Runtime - Contains the trace enabled shared library for the target platform.
dacs-hybrid-trace-4.0.0-#.x86_64.rpm	
dacs-hybrid-trace-4.0.0-#.ppc.rpm	
dacs-hybrid-trace-4.0.0-#.ppc64.rpm	
dacs-hybrid-trace-devel-4.0.0-#.i686.rpm	DaCS for Hybrid Trace Development - Contains the trace-enabled static library for the target platform.
dacs-hybrid-trace-devel-4.0.0-#.x86_64.rpm	
dacs-hybrid-trace-devel-4.0.0-#.ppc.rpm	
dacs-hybrid-trace-devel-4.0.0-#.ppc64.rpm	
dacs-hybrid-cross-devel-4.0.0-#.noarch.rpm	DaCS for Hybrid Cross Development - Contains the header files and libraries needed for cross-architecture development.
dacs-hybrid-tools-4.0.0-#.x86_64.rpm	DaCS for Hybrid-tools - Contains tools for the diagnosis of problems.
dacs-hybrid-daemon-4.0.0-#.i686.rpm	DaCS for Hybrid host daemon and configuration files.
dacs-hybrid-daemon-4.0.0-#.ppc.rpm	DaCS for Hybrid accelerator daemon and configuration files.
dacs-hybrid-examples-source-4.0.0-#.noarch.rpm	Contains examples that demonstrate how to use the DaCS APIs in a Hybrid environment.
datamover-pcie-axon-1.0.0–#.i686.rpm	Additional library necessary for DaCS for Hybrid to work with PCIe over Axon device driver. If PCIe over Axon is not being used, these libraries do not need to be installed.
datamover-pcie-axon-1.0.0–#.86_64.rpm	
datamover-pcie-axon-1.0.0–#.ppc.rpm	
datamover-pcie-axon-1.0.0–#.ppc64.rpm	

Chapter 5. Configuring DaCS for Hybrid

When DaCS for Hybrid is installed, some configuration is necessary as stated in this section. No configuration is necessary for standalone DaCS for Cell/B.E..

Configuring the topology

The `dacs_topology.config` file contains information about the physical layout of the system using DaCS for Hybrid and information on the HE and which AEs they can use. The topology configuration file must be modified to match the physical topology of the system on which you want to use DaCS.

There are two different communication networks that are defined in the topology file. The first is the process management network used by the daemons (hdacsd and adacsd) and the second is the data movement network which is used for data movement between the HE and the AE applications. Currently the process management network must be over sockets and the data movement network can be over either sockets or PCIe-Axon.

The topology file is specified using XML.

Table 5. Structure of the configuration file

Section name	Purpose
DaCS Topology (<DaCS_Topology>)	Outermost container
Hardware (<hardware>)	Contains information mapping physical hardware assets to DaCS elements. This includes the nesting relationships between blade servers and individual processing units.
Topology (<topology>)	Represents the ability to reserve DaCS elements using the information encoded in the hardware section.
Process Management (<processManagement>)	Represents information used by the DaCS daemons.
Communications (<communications>)	Defines communications connections between DaCS elements as listed in the Hardware section.

The elements and their associated attributes used in the topology configuration file are as follows:

DaCS_Topology> element

Format: <DaCS_Topology version=xs:string>

Example: <DaCS_Topology version='2.0'>

Description The <DaCS_Topology> element is the outermost element in the DaCS topology file. It contains the hardware, topology, processManagement and communications sections.

Attributes version: The verson number of the topology file. Currently only 2.0 is supported.

<hardware> element

Format: <hardware>

Example: <hardware>

Description: The <hardware> element is a section delimiter and container for <de> elements.

Attributes: None.

<de> element

Format: `<de name=xs:string type=xs:string [numProcess=xs:integer] [affinity=xs:string]>`

Example: `<de tag='OB1' type='DACS_DE_SYSTEMX' numProcess='1' />`

Description: The <de> element allows for the description and naming of hardware which will be used as HE and AE DaCS elements. <de> elements may be nested to show an association between physical pieces of hardware and logical grouping of DaCS Elements.

Attributes

name: A descriptive tag which represents this DE throughout the rest of the file.

type: A descriptive tag based on the DaCS Element enumerated type in the dacs.h header which represents the type of this DE.

numProcess: (OPTIONAL) A numeric tag which represents the maximum number of process which may be started on this DE when it is used as an AE. If this is not specified only one process will be allowed on the AE.

affinity: (OPTIONAL) The value for the processor affinity to use when starting a process on this de (when it is an AE). If this is not specified the process will be started without processor affinity. The value is modeled after the taskset mask where 0x01 is processor #0, 0x02 is processor #1, and so forth.

Note: When a blade is reserved the accelerator process is started without affinity. The connection entry used will be based on which CBE the operating system starts the process on.

<topology> element

Format: `<topology>`

Example: `<topology>`

Description The `<topology>` element is a section delimiter and container for `<canReserve>` elements.

Attributes None.

<canReserve> element

Format: `<canReserve he=xs:string ae=xs:string [only=xs:Boolean] />`

Example: `<canReserve he='OC1' ae='CBE22' only='false' />`

Description: The `<canReserve>` element defines the visibility and reservation relationships between DEs. Its direct translation is that the HE may reserve the AE, with the restrictions placed on it by the only attribute.

Attributes: These comprise the following:

he: A descriptive tag which represents a DE in topology file. It must match an entry in the `<hardware>` section.

ae: A descriptive tag which represents a DE in the topology file. It must match an entry in the `<hardware>` section.

only: (OPTIONAL) The only attribute can be specified when the AE attribute identifies a `DACS_DE_CELLBLADE`. When TRUE states that the given HE can only reserve the given `DACS_DE_CELLBLADE`, not the individual Cell/B.E. that it contains.

processManagement> element

Format: <processManagement>

Example: <processManagement>

Description: The <processManagement> element is a section delimiter and container for <mgtDe> elements.

Attributes: None.

<mgtDe> element

Format: `<mgtDe deName=xs:string fabric=xs:string connInfo=xs:string />`

Example: `<mgtDe deName='OB1'fabric='IP' connInfo='192.168.1.1' />`

Description The <mgtDe> element defines which DaCS Elements are serving as part of the process management framework, and the necessary communications information to allow connections between these DEs by the DaCS daemons.

Attributes:

deName: A descriptive tag which represents a DE in this file. It must match an entry in the <hardware> section.

fabric: A descriptive tag which represents information on the format of the connInfo information so that it may be processed. Currently only IP is supported.

connInfo: A representation of the information needed to setup communications to this DE by the process management daemons.

<communications> element

Format: `<communications>`

Example: `<communications>`

Description : The <communications> element is a section delimiter and container for fabric and connection information that allows for communication between DEs.

Attributes: None.

<fabrics> element

Format: <fabrics>

Example: <fabrics default=’IP’>

Description The <fabrics> element is a section delimiter and container for <fabric> elements.

Attributes

default: This identifies the value of the type attribute of the contained fabric element that represents the fabric that should be used by default. This can be overridden using the DACS_HYBRID_USE_FABRIC_TYPE environment variable.

<fabric> element

Format: `<fabric type=xs:string device=xs:string [connInfo=xs:string] />`

Example: `<fabric type='AxonD' device='DM_IBM_AXON_PCIE' connInfo='libdm_pcie_axon.' />`

Description: The `<fabric>` element describes a communications fabric, that is the underlying transport that is used for communication. It supplies a type name for a given fabric definition along with device and connection info.

Attributes:

type: A descriptive tag which represents the fabric throughout the rest of the file.

device: A descriptive tag which represents information on the format of the connInfo information so that it may be processed. Currently only IP and DM are supported.

connInfo: A representation of the information needed to setup the given fabric for use.

<connections> element

Format: `<connections>`

Example: `<connections>`

Description: The `<connections>` element is a section delimiter and container for `<connection>` elements.

Attributes: None.

<connection> element

Format: `<connection endPt1=xs:string endPt2=xs:string >`

Example: `<connection endPt1='OC1' endPt2='CBE22' >`

Description: The `<connection>` element is a container for information regarding how to communicate over a given fabric between **endPt1** and **endPt2**. It contains a series of `<forFabric>` elements, each of which describes specific communications information for a given fabric.

Attributes

endPt1: A descriptive tag which represents a DE in the topology file. It must match an entry in the `<hardware>` section.

endPt2: A descriptive tag which represents a DE in the topology file. It must match an entry in the `<hardware>` section.

<forFabric>

Format: `<forFabric type=xs:string>`

Example: `<forFabric type='IP'>`

Description The `<forFabric>` element exists inside of a `<connection>` element supplying a container for specific connection information between two DaCS elements for a given fabric.

Attributes:

type: A value representing the fabric. This must be one of the fabric types in the <fabrics> section.

<from1to2> element

Format: `<from1to2 connInfo=xs:string />`

Example: `<from1to2 connInfo='192.168.1.2' />`

Description The `<from1to2>` element describes a specific communication channel across a fabric, which is defined by the enclosing `<forFabric>` element, where the 1 and 2 in the name refer to the enclosing `<connection>` elements **endPt1** and **endPt2** attributes.

Attributes:

connInfo: A fabric specific address representation for the communication channel.

<from2to1> element

Format: `<from2to1 connInfo=xs:string />`

Example: `<from2to1 connInfo='192.168.1.2' />`

Description: The `<from2to1>` element describes a specific communication channel across a fabric, which is defined by the enclosing `<forfabric>` element, where the 1 and 2 in the name refer to the enclosing `<connection>` elements **endPt1** and **endPt2** attributes.

Attributes:

connInfo: A fabric specific address representation for the communication channel.

An XML schema for the topology file is provided as part of the dacs-hybrid-tools RPM and is installed in `/opt/cell/sdk/prototype/usr/src/dacs/hybrid/dacs-topology/dacs_topology.xsd`.

Setting affinity for DaCS for Hybrid

When creating the topology file, the use of affinity on the host element needs to be taken into consideration. In some cases, especially in the case of PCIe-Axon, associating a specific core with a specific Cell/B.E. provides the best performance. In other cases, such as sockets, it is less important. This is different from the specification of what affinity to use when starting the Accelerator process on an AE as part of its DE element. Instead, the HE affinity is set on the host element prior to running the host application (for example, by using taskset -p 0x00000001 $$).

The affinity on the host is used to determine which of the `de` elements should be used for the host element. For example, given the following snippet of the topology file:

```
<de name="OB1" type='DACS_DE_SYSTEMX' >
  <de name='OC1' type='DACS_DE_SYSTEMX_CORE' />
  <de name='OC2' type='DACS_DE_SYSTEMX_CORE' />
  <de name='OC3' type='DACS_DE_SYSTEMX_CORE' />
  <de name='OC4' type='DACS_DE_SYSTEMX_CORE' />
</de >
```

The first DACS_DE_SYSTEMX_CORE de element would be used for affinity 0x01, the second for 0x02, the third for 0x04, and the fourth for 0x08. Thus when the affinity is set to 0x02, the items in the topology file that reference OC3 will be used.

If affinity is not set, then the first entry OC1 will always be used. If affinity is set, then the associated entry for the core must be present in the topology file. For example, given the following snippet of the topology file:

```
    <de name="OB1" type='DACS_DE_SYSTEMX' >
      <de name='OC1' type='DACS_DE_SYSTEMX_CORE' />
    </de >
```

Not setting affinity or setting it to 0x01 will work. However, because entry is not found, any other affinity setting will cause the host application to be unable to reserve any AEs.

Affinity can be used to limit the number of AEs that an HE can use, for example, allowing each core to reserve only one AE:

```
  <topology >
        <canReserve he='OC1' ae='CBE22' />
        <canReserve he='OC2' ae='CBE12' />
        <canReserve he='OC3' ae='CBE21' />
        <canReserve he='OC4' ae='CBE11' />
  </topology >
```

Or to ensure that only the best performing configuration is used. In the example above, the hardware may perform best when OC1 communicates with CBE22, etc.

Verifying user ids on the accelerator

The accelerator application will be started using the user id of the host application. It is important to verify that these user ids are available on the accelerator prior to using DaCS for Hybrid.

Configuring when using sockets

The sample topology file using sockets is provided as part of the install: `/etc/dacs_topology.config.template_sockets`. This file can be used as the basis for a host connected to a Cell/B.E. blade server by getting the IP addresses and filling them in.

```
<DaCS_Topology version='2.0' >
  <!-- Sample Topology config for single-core HE and IP-connected Cell Blade      -->
  <!-- Modify the IP addresses to match the actual communication interfaces,      -->
  <!-- and copy to /etc/dacs_topology.config                                      -->

  <hardware >

    <de name="OB1" type='DACS_DE_SYSTEMX' >
      <de name='OC1' type='DACS_DE_SYSTEMX_CORE' />
    </de >

    <de name='CB1' type='DACS_DE_CELLBLADE' >
      <de name='CBE11' type='DACS_DE_CBE' />
      <de name='CBE12' type='DACS_DE_CBE' />
    </de >

  </hardware >

  <topology >

    <!-- Host application can reserve the Cell Blade   -->
    <!-- or any CBE on the Cell Blade                  -->
    <canReserve he='OC1' ae='CB1' />

  </topology >

  <!-- Configure the IP addresses for the DaCS daemons -->
  <!-- The IP addresses are for illustration only      -->
  <!-- Replace them with your actual IP addresses      -->
  <processManagement >
    <mgtDe deName='OB1' fabric='IP' connInfo='XXX.XXX.1.1' />
    <mgtDe deName='CB1' fabric='IP' connInfo='XXX.XXX.1.2' />
  </processManagement >

  <communications>

    <!-- This configuration file is only for TCP/IP -->
    <fabrics default='IP' >
      <fabric type='IP'       device='TCP' />
    </fabrics>

    <!-- Configure point-to-point connections over the communication fabrics -->
    <!-- Each connection identifies the HE and CBE endpoints and             -->
    <!-- the information needed to initiate a connection over each fabric    -->

    <connections >

      <!--  HE: OC1 to AE: CBE11 -->
      <connection endPt1='OC1' endPt2='CBE11' >
        <forFabric type='IP' >
          <from1to2 connInfo='XXX.XXX.1.2' />
          <from2to1 connInfo='XXX.XXX.1.1' />
        </forFabric >
      </connection >

      <!--  HE: OC1 to AE: CBE12 -->
      <connection endPt1='OC1' endPt2='CBE12' >
        <forFabric type='IP' >
          <from1to2 connInfo='XXX.XXX.1.2' />
          <from2to1 connInfo='XXX.XXX.1.1' />
        </forFabric >
      </connection >

    </connections >
  </communications>
</DaCS_Topology >
```

Note: The **canReserve** element in this example specifies the blade server (CB1) which allows the host (OC1) to reserve either (or both) of the Cell/B.E.s (CBE11 and CBE12).

Configuring when using PCIe-Axon

In order to use PCIe-Axon, the datamover-pcie-axon RPM and the PCIe-Axon device drivers must be installed on both the host and accelerators. Note that there are both 32 bit and 64 bit versions of the RPMs.

When installed, the permissions for the PCIe-Axon device driver will not allow DaCS for Hybrid to use it. With the DaCS hybrid program, the user program cannot run unless the user is running as superuser (root) or as a member of the root group. If an administrator does not choose to do this, the workaround is to give the appropriate read/write permissions to the user for the device driver.

The permissions to set on the axon devices are:

```
root@localConsole /]# ls -l /dev/axon*
crw-rw---- 1 root root 250, 0 2008-05-06 13:48 /dev/axon0
crw-rw---- 1 root root 250, 1 2008-05-06 13:48 /dev/axon1
crw-rw---- 1 root root 250, 2 2008-05-06 13:48 /dev/axon2
crw-rw---- 1 root root 250, 3 2008-05-06 13:48 /dev/axon3
```

Only a single process can be started on an AE when running over PCIe-Axon. Modifying the configuration file to support more than one process will result in unpredictable results.

A sample topology file using PCIe-axon (and sockets) is provided as part of the install: `/etc/dacs_topology.config.template_pcie`. This file contains documentation on how to modify it.

Configuring the DaCS for Hybrid daemon

The host daemon service is named `hdacsd` and the accelerator daemon service is named `adacsd`. Both daemons are configured by editing the `/etc/dacsd.conf` file on their respective systems.

Default versions of the `dacsd.conf` file are provided when installing the daemon RPMS. These default files will work in most cases without change. The `dacsd.conf` file contains detailed comments about the supported parameters and values and should be referred to for the most up-to-date information.

Sample file

```
# Configuration file for DaCS daemons, hdacsd and adacsd

# Configuration file version

dacsd_conf_version="1.0"

# Topology configuration file

dacs_topology_config=/etc/dacs_topology.config


# The number of seconds period between polling Cell Blades to keep
# track of their availability setting blade_monitor_interval to 0
# disables the Cell Blade monitor

blade_monitor_interval=60


# The default dacs_de_kill timeout, the number of seconds between
# sending SIGTERM and SIGKILL.  0 means that dacsd_he_terminate will
# send SIGKILL immediately, without sending SIGTERM

dacsd_kill_timeout=5


# AE Current Working Directory Prefix
#
# Each AE process started by dacs_de_start is given a temporary
# current working directory named
# <ae_cwd_prefix>/adacsd-tmp/<HE Process Info>/<AE Process Info>
# DaCS applications refer to this by $AE_CWD in dacs_de_start
# program parameters and environment variables.

ae_cwd_prefix=/adacsd


# AE Current Working Directory permissions
#
# Specifies the permissions given to the AE Current Working Directory
# The value is an octal number representing the bit pattern for the permissions,
# as supported by the chmod command. Note that these are directory permissions.

ae_cwd_permissions=0755
```

```
# Normally the AE Current Working Directories and their contents are
# deleted when the adacsd starts. (i.e., all files and directories
# below <ae_cwd_prefix>/adacsd-tmp). Set ae_cwd_keep=true if you
# want to prevent all AE Current Working Directories from being deleted
# when adacsd starts.
#
# Use ae_cwd_keep in conjunction with the HE process environment variable
# DACS_HYBRID_KEEP_CWD as follows.
#
# To keep an AE CWD indefinitely:
#   ae_cwd_keep must be set to true in the dacsd.conf file, and
#   the DACS_HYBRID_KEEP_CWD environment variable must be set to Y.
# To keep an AE CWD only until adacsd is restarted:
#   ae_cwd_keep must be set to false in the dacsd.conf file, and
#   the DACS_HYBRID_KEEP_CWD environment variable must be set to Y
# To delete an AE CWD when the HE process terminates:
#   DACS_HYBRID_KEEP_CWD must be unset, or set to something other than Y

ae_cwd_keep=false


# dacs_de_start transfers files via tar. The tar commands are
# configurable to help with debugging, but should not normally be
# changed. Configuring these commands incorrectly will cause
# dacsd_he_xfer to fail. For example:
# he_tar_command="/bin/tar cvvf -"
# ae_tar_command="/bin/tar xvvf -"

he_tar_command="/bin/tar cf -"
ae_tar_command="/bin/tar xf -"


# change adacsd_use_numa to false to disable numa support

adacsd_use_numa=true

# Set the size limit on core files for the AE process
# This cannot exceed the system-configured limit.
# if a core dump is larger than the size limit, the dump will not occur
# If child_rlimit_core=-1, the current core file size limit is NOT changed for AE process
# If child_rlimit_core=value>0, the current core file size limit will be changed
 to min(value, system_limit)
# If child_rlmit=-1, the core file size limit will be set to the system limit--which could be
infinite

child_rlimit_core=-1


# Set the size limit on core files for the hdacsd and adacsd daemons
# This cannot exceed the system-configured limit.
# if a core dump is larger than the size limit, the dump will not occur
# If DAEMON_COREFILE_LIMIT=0, daemon core files are disabled
# If DAEMON_COREFILE_LIMIT='unlimited', daemon core files are enabled

DAEMON_COREFILE_LIMIT='unlimited'


# Log size limit
#
# When the current log file size exceeds the specified limit,
# the current log file is renamed and a new log file is started.
# The log files are renamed to <log name>-<YYYY>-<MM>-<DD>-<HH>:<MM>:<SS>
# For example: /var/log/adacsd.log-2007-08-17-13:49:52
# A log size limit of 0 prevents this log rotation.
# The minimum non-zero value of log_size_limit is 4194304
```

```
log_size_limit=16777216


# Log file limit
#
# This value specifies the total number of log files to keep, including the current one.
# When the limit is exceeded, the oldest log files are deleted first.
# A log file limit of 0 prevents the deletion of old log files.
# The minimum non-zero value of log_file_limit is 1

log_file_limit=2


# Default startup options

ADACSD_ARGS="--log /var/log/adacsd.log --pidfile /var/run/adacsd.pid"
HDACSD_ARGS="--log /var/log/hdacsd.log --pidfile /var/run/hdacsd.pid"
```

Applying changes

Changes to the topology or daemon configuration do not take affect until the daemon is restarted. Start and stop the daemon using the service command in the `/sbin directory`. To stop the host daemon, type the following command as root:

```
# /sbin/service hdacsd stop
```

To start the host daemon type:

```
# /sbin/service hdacsd start
```

The daemon will also read changes to `dacsd.conf` when it receives a SIGHUP signal. To send a SIGHUP signal to the host daemon, send the following command as root:

```
# killall –SIGHUP hdacsd
```

See the service man page for more details about controlling daemons.

Part 3. Programming with DaCS

DaCS API functions

The DaCS library API services are provided as functions in the C and FORTRAN languages. The protocols and constants required are made available to the compiler by including the C or Fortran DaCS header files.

C: include `dacs.h` as:

```
#include <dacs.h>
```

Fortran: include `dacsf.h` and `dacsf_interface.h` as:

```
include 'dacsf.h'
include 'dacsf_interface.h'
```

See Appendix G, "DaCS Fortran bindings," on page 199 for more information.

Errors

In general the return value from these functions is an error code (see Appendix D, "Error codes," on page 193). Data is returned within parameters passed to the functions.

Implementations may provide options, restrictions and error codes that are not specified here.

When more than one error condition is present it is not guaranteed which one will be reported. The default (optimized) DaCS library does no error checking of the parameters of the DaCS API calls. These checks are only performed in the debug DaCS library. Therefore, it is recommended that application development be done using the debug library and the optimized library used for performance measurement and production runs.

Most DaCS APIs return a `DACS_ERR_T` value on completion. This error return type can be broken into three categories: errors, success, and status.

- Error values are less than 0 and imply a failure in the API call
- Status values are greater than 0, and are used for returning a non-failure state from certain APIs
- A return value of 0 (DACS SUCCESS) indicates a successful return from a DaCS operation

API environment

To make these services accessible to the runtime code each process must create a DaCS environment. This is done by calling the special initialization service `dacs_init()`. When this service returns the environment is set up so that all other DaCS function calls can be invoked.

When the DaCS environment is no longer required the process must call `dacs_exit()` to free all resources used by the environment.

Process management model

When working with the HE and AEs there has to be a way to uniquely identify the participants that are communicating. From an architectural perspective, each accelerator could have multiple processes simultaneously running, so it is not enough simply to identify the accelerator. Instead the unit of execution on the accelerator (the DaCS Process) must be identified using its DaCS Element Id (DE) and its Process Id (PID). The DE is received when the accelerator is reserved (using `dacs_reserve_children()`) and the PID is received when the process is started (using `dacs_de_start()`). Since the parent is not reserved, and no process is started on it, two constants are provided to identify the parent: `DACS_DE_PARENT` and `DACS_PID_PARENT`. When communicating with the parent, these constants must be used. Similarly, to identify the calling process itself, the constants `DACS_DE_SELF` and `DACS_PID_SELF` are provided.

Resource sharing model

The APIs supporting the locking primitives, memory regions and groups follow a consistent pattern of creation, sharing, usage and destruction:

- Creation: An object is created which will be shared with other DEs, for example with `dacs_remote_mem_create()`.
- Sharing: The object created is then shared by coordinated share and accept calls. The creator shares the item (for instance with `dacs_remote_mem_share()`), and the DE it is shared with accepts it (in this example with `dacs_remote_mem_accept()`). These calls must be paired. When one is invoked it waits for the other to occur. This is done for each DE the share is associated with.
- Usage: This may require closure (such as in the case of groups) or the object may immediately be available for use. For instance remote memory can immediately be used for put and get.
- Destruction: The DEs that have accepted an item can release the item when they are done with it (for example by calling `dacs_remote_mem_release()`). The release does not block, but notifies the creator that it is no longer being used and cleans up any local storage. The creator does a destroy (in this case `dacs_remote_mem_destroy()`) which blocks until all of the DEs it has shared the object with release the object. It then destroys the object.

Chapter 6. Building a DaCS application

Three versions of the DaCS libraries are provided with the DaCS packages: optimized, debug and trace. The optimized libraries do no error checking of the parameters of the DaCS API calls and are intended for production use. The debug libraries have much more error checking than the optimized libraries and are intended to be used during application development. The traced libraries are the optimized libraries with performance and debug trace hooks in them. These are intended to be used to debug functional and performance problems that might be encountered. The traced libraries use the interfaces provided by the Performance Debug Tool (PDT) and require that this tool be installed. See Appendix B, "Performance and debug trace," on page 187 for more information on configuring and using traced libraries.

Both static and shared libraries are provided for both implementations. For DaCS for Cell/B.E., only static libraries are provided for the SPE. The desired library is selected by linking to the chosen library in the appropriate path. Static libraries are named `libdacs.a` for DaCS Cell/B.E., `libdacs_hybrid.a` for DaCS Hybrid, and the shared libraries are `libdacs.so` for DaCS Cell/B.E. and `libdacs_hybrid.so` for DaCS Hybrid.

The C and Fortran APIs are both included in these libraries. C APIs start with the prefix `dacs_` and Fortran APIs start with the prefix `dacsf_`. See "Binding alias names" on page 204 for Fortran C interoperability information.

When using the DaCS Cell/B.E. and DaCS for Hybrid together the SPE code should link in the appropriate `libdacs.a` and the PPE code should link to the appropriate `libdacs_hybrid.so` or `libdacs_hybrid.a`.

If you use the `libdacs_hybrid.a` library, then other libraries are required. These are:

- -lstdc++ -ldl -lrt -pthread for the x86 side
- -lstdc++ -ldl -lrt -lspe2 -pthread for the ppu side
- -lstdc++ for the shared library on the PPU side

If you use the DaCS for Cell/BE static PPU library `libdacs.a`, then you will need to build with `'-lspe2'`.

The locations of the DaCS Cell static and shared libraries are:

Table 6.

Description	PPU 32 bit Library Path	PPU 64 bit Library Path	SPU Library Path
Optimized	/usr/lib	/usr/lib64	/usr/spu/lib
Debug	/usr/lib/dacs/debug	/usr/lib64/dacs/debug	/usr/spu/lib/dacs/debug
Traced	/usr/lib/dacs/trace	/usr/lib64/dacs/trace	/usr/spu/lib/dacs/trace

The locations of the DaCS Hybrid static and shared libraries are the same on both x86_64 and PPU:

Table 7.

Description	32 bit Library Path	64 bit Library Path
Optimized	/usr/lib	/usr/lib64
Debug	/usr/lib/dacs/debug	/usr/lib64/dacs/debug
Traced	/usr/lib/dacs/trace	/usr/lib64/dacs/trace

Affinity requirements for host applications

A DaCS for Hybrid application on the host (x86_64) may need to set affinity to start. If the topology configuration file has been setup to use affinity (see Chapter 5, "Configuring DaCS for Hybrid," on page 15 for further information) then processor affinity must be used. This can be done:

- on the command line,
- in `mpirun`, or
- through the `sched_setaffinity` function.

Here is a command line example to set affinity of the shell to the first processor:

```
taskset -p 0x00000001  $$
```

The bit mask, starting with 0 from right to left, is an index to the processor affinity setting. Bit 0 is on or off for CPU 0, bit 1 for CPU 1, and bit number *x* is CPU number *x*. `$$` means the current process gets the affinity setting.

```
taskset -p $$
```

will return the mask setting as an integer. Using the `-c` option makes the `taskset` more usable. For example,

```
taskset -pc 7  $$
```

will set the processor CPU affinity to CPU 7 for the current process. The `-pc` parameter sets by process and CPU number.

```
taskset -pc  $$
```

will return the current CPU setting for affinity for the current process. According to the man page for `taskset` a user must have `CAP_SYS_NICE` permission to change CPU affinity. See the man page for `taskset` for more details.

To launch a DaCS application use a `taskset` call, for example:

```
taskset 0x00000001  HelloDaCSApp Mike
```

or equivalently

```
taskset -c 0 HelloDaCSApp Mike
```

where the application program is `HelloDaCSApp` and is passed an argument of "`Mike`".

If specified in the topology configuration, on the accelerator system the `adacsd` launch of an AE application on a specific Cell/B.E. includes setting the affinity to the Cell/B.E. and its associated memory node. If the launch is on the Cell/B.E. blade server as an AE no affinity is set.

Blocking APIs

Some APIs are defined as blocking. This means that they do not return to the caller until the operation is finished. This does not reflect whether the underlying implementation spins, polls, or blocks the calling thread to a non- runnable state.

Note: For the current release, none of the blocking APIs block the calling thread to a non-runnable state. For cases where the application wishes to control the frequency at which the completion of an operation is checked, the associated non-blocking test API can be used instead of the blocking API.

Using the Hybrid library

The DaCS Hybrid version of the library integrates with the DaCS for Cell/B.E. implementation on a Cell/B.E. (PPU) system. Integrated API calls can be interpreted as Hybrid or Cell/B.E. library calls, depending on the set of parameters that are passed in. This can lead to confusing return codes in some situations, especially when debugging an application using the debug libraries. The memory init and cleanup APIs suffer from this problem in particular; the Hybrid versions are only called when `DACS_DE_PARENT` is used as the DE, any other value calls the Cell/B.E. version which returns `DACS_ERR_NOT_SUPPORTED_YET`, instead of `DACS_ERR_INVALID_DE`.

The following functions exhibit this behavior:

- `dacs_mem_accept`: The PPU host cannot accept memory from an SPU because an SPU cannot create memory.
- `dacs_mem_release`: The PPU host cannot release memory from an SPU because an SPU cannot create memory.
- `dacs_mem_register`: The PPU host cannot release memory from an SPU because an SPU cannot create memory.

Handling API Return codes

The API return code which is returned by the C signatures and returned as the rc out parameter on the Fortran signatures will always be positive when the service is successful and negative when it fails. It is a best practice to check for a negative return code when checking for a failure, rather than checking for a specific failing return code. Once a failing return code is discovered, the application can then process specific return codes and handle any unexpected ones. In this way, future API changes that add new failing return codes or change the failure returned when more than one is applicable will be handled.

Part 4. API reference

This topic describes the DaCS functions.

Chapter 7. Initialization and termination

The `dacs_init()` and `dacs_exit()` services initialize and close down access to the DaCS library.

Call `dacs_init()` before you use any other DaCS services, and do not use any DaCS services after you have called `dacs_exit()`.

Calling `dacs_exit()` on an AE causes the communications between the AE and HE to be stopped. On an AE, attempting to call `dacs_init()` after `dacs_exit()` will result in a hang or failure because `dacs_init()` must coordinate with an HE call to `dacs_de_start()`. This starts the AE process. An HE, however, may call `dacs_init()` after a successful call to `dacs_exit()` if it were not started through `dacs_de_start()`.

Prior to calling `dacs_exit()`, all accepted shared resources must be explicitly released. Not doing so can result in a hang on the destroy of the shared resource, as the owner waits on all participants to release.

Initialization and termination usage scenarios

The section below describes how to initialize and terminate DaCS for use.

Parent DE	**Child DE**
Initialize DaCS for use: `dacs_init(DACS_INIT_FLAGS_NONE)` Prior to using DaCS every DE must initialize it. In this case, the parent's execution is not related to any remote execution, so it simply initializes the state of DaCS without needing to communicate start-up state to an initiating parent.	

Start an accelerator DaCS application:
`dacs_de_start(`*`child_de`*`,`
      `child_app``,`
      `&argv``,`
      `&envv``,`
      `&child_pid``)`
 	
This initiates the execution of an accelerator application based on the application type denoted in *creation_flags*. The supplied application type indicates to DaCS how to interpret the *child_app* argument for starting the accelerator application. The *child_app* argument can be a filename, list, or handle to an embedded executable. Arguments to the child application are passed through the *argv* and *envv* argument pointers. On successful return, a pointer to the started process ID *child_pid* is returned.
	Initialize DaCS for use:
	`dacs_init(DACS_INIT_FLAGS_NONE)`
	
	Prior to using DaCS every DE must initialize it.
	
	In this case, the child DE's initialization of DaCS corresponds to the parents starting of the child application. The parent's call to de_start is blocked awaiting successful completion of the initialization and state feedback.

Parent DE	**Child DE**

Test child run state:
`dacs_de_test(`*`child_de`*`,`
      `child_pid``,`
      `&exit_status``)`
 	
At any time, the parent DE may request the state of its children processes. The state check is non-blocking, but will invalidate child processes that have terminated. The current state is returned via *exit_status*.
 	
In this case, the processes test reveals that the child has not exited and thus is still in the running state.	

	Terminate DaCS usage: `dacs_exit()` Once use of DaCS is complete, it needs to be shut-down. In this case, the accelerator application is terminating its use of DaCS with the intent of eventually exiting. Once DaCS has been closed-down, it cannot be restarted as that requires another start operation. Thus, the application must exit and be restarted in order to use DaCS again

Wait for child completion:
`dacs_de_wait(`*`child_de`*`,`
      `child_pid``,`
      `&exit_status``)`
 	
This allows a parent DE to wait on the completion of a child processes. The call blocks execution of the parent until the specified DE/PID has terminated. The status of the terminated process is returned in *exit_status*.
 	
In this case, the parent DE is waiting on the completion of *child_de* and *child_pid*. This child process has exited and thus it's status is returned in *exit_status*.	

Terminate DaCS usage: `dacs_exit()` Once use of DaCS is complete, it needs to be shut-down. In this case, the application is terminating its use of DaCS, but does not necessarily intend on exiting. Since the existence of this DE is not related to a parent's initiation, the application may call the init/exit pairing as many times as necessary.	

dacs_init

NAME

dacs_init - Initialize all runtime services for DaCS.

SYNOPSIS

C syntax

DACS_ERR_T dacs_init (uint32_t *config_flags***)**

Fortran syntax

dacsf_init (dacs_init_flags_t *config_flags***, DACS_ERR_T** *rc***)**

Call parameter

`config_flags`	Specifies various runtime settings.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_init` service initializes all runtime services for DaCS.

This service must be called for every process before any other DaCS services can be used. All other DaCS services will return `DACS_ERR_NOT_INITIALIZED` if called before this service.

A host process may call this service more than once, provided there is a call to `dacs_exit()` in between. An accelerator process may only call this service once, even if there is an intervening call to `dacs_exit()`. The host process is blocked in `dacs_de_start()` until `dacs_init()` is called on the accelerator.

Specifying that DaCS will be used in a single threaded mode means that DaCS will not perform locks to ensure that internal structures are not being updated simultaneously. This means that either the application using DaCS must only use DaCS from a single thread or must ensure that the DaCS calls are serialized (possibly via their own locking).

If single threaded mode is specified and the application does not adhere to it, internal data structures may be destroyed and unpredictable results occur.

The benefit of using single threaded mode is that for architectures where locks are expensive, such as the PPU, avoiding the locks can provide a performance improvement on every use of a DaCS service. Note that the threaded mode does not have to be the same for both the accelerator and the host. In this way single threaded mode can be used for the code running on the architecture with poor lock performance, while not using it on architectures that have fast locks.

The `config_flags` parameter can have the following settings:

- `DACS_INIT_FLAGS_NONE or 0`: default DaCS configuration
- `DACS_INIT_SINGLE_THREADED`: give a hint to DaCS that the application is running in a single-threaded mode. DaCS can use this value to optimize its services for this environment.

Cell/B.E. : the SPU is always single threaded and thus the config flag has no meaning.

RETURN VALUE

The `dacs_init` service returns the following codes:

- `DACS_SUCCESS`: DaCS environment was successfully initialized.
- `DACS_ERR_NO_RESOURCE`: Unable to allocate required resources.
- `DACS_ERR_INITIALIZED`: DaCS is already initialized.
- `DACS_ERR_DACSD_FAILURE`: Unable to communicate with DaCS daemon services.
- `DACS_ERR_INVALID_ATTR`: the flags parameter has invalid flag values specified.
- `DACS_ERR_VERSION_MISMATCH`: the accelerator does not match the host/parent version.
- `DACS_ERR_ARCH_MISMATCH`: Attempted to use a 64-bit accelerator application with a 32-bit host application or vice-versa.

SEE ALSO

dacs_exit(3)

dacs_exit

NAME

dacs_exit - Close down all runtime services for DaCS.

SYNOPSIS

C syntax

DACS_ERR_T dacs_exit ( )

Fortran syntax

dacsf_exit (DACS_ERR_T *rc*)

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_exit` service closes down and destroys all runtime services, processes, transfers, and memory used by DaCS. After calling this service, no other DaCS services can be used until another `dacs_init()` is performed. Calling `dacs_init()` after a `dacs_exit()` is only supported for a host process. For all accepted objects the application must call the appropriate release function and for all created objects the appropriate destroy function before `dacs_exit()` is called.

RETURN VALUE

The `dacs_exit` service returns the following error indicators:
- `DACS_SUCCESS`: DaCS environment was successfully shutdown.
- `DACS_ERR_NOT_INITIALIZED`: runtime environment was not initialized.
- `DACS_ERR_DACSD_FAILURE`: unable to communicate with DaCS daemon services.

SEE ALSO

dacs_init(3)

dacs_runtime_init (deprecated)

NAME

dacs_runtime_init - Initialize all runtime services for DaCS.

SYNOPSIS

C syntax

DACS_ERR_T dacs_runtime_init (void *, void *)

Fortran syntax

dacsf_runtime_init(DACS_ERR_T *rc*)

Call parameter
All parameters must be set to `NULL` for DaCS 4.0. Passing in a value other than `NULL` will result in the error `DACS_ERR_INVALID_ADDR`.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_runtime_init` function is deprecated. Use the `dacs_init()` function instead.

The `dacs_runtime_init` service initializes all runtime services for DaCS.

Note: This service must be called for every process before any other DaCS services can be used. All other DaCS services will return `DACS_ERR_NOT_INITIALIZED` if called before this service.
A host process may call this service more than once, provided there is a call to `dacs_runtime_exit()` in between. An accelerator process may only call this service once, even if there is an intervening call to `dacs_runtime_exit()`.

RETURN VALUE

The `dacs_runtime_init` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: invalid pointer.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.This can be caused by either running out of memory or a failure in the pthread library.
- `DACS_ERR_INITIALIZED`: DaCS is already initialized.
- `DACS_ERR_ARCH_MISMATCH`: attempted to use a 64-bit accelerator application with a 32-bit host application or vice-versa.
- `DACS_ERR_VERSION_MISMATCH`: version mismatch between library and DaCS.

SEE ALSO

dacs_init(3), dacs_exit(3), dacs_runtime_exit(3)

dacs_runtime_exit (deprecated)

NAME

dacs_runtime_exit - Close down all runtime services for DaCS.

SYNOPSIS

C syntax:

DACS_ERR_T dacs_runtime_exit (void)

Fortran syntax:

dacsf_runtime_exit(DACS_ERR_T *rc*)

Return parameter	
`rc`	**Fortran only**: see Return value

DESCRIPTION

The `dacs_runtime_exit` function is deprecated. Use the `dacs_exit()` function instead.

The `dacs_runtime_exit` service closes down and destroys all runtime services, processes, transfers, and memory used by DaCS. After calling this service, no other DaCS services can be used until another `dacs_runtime_init()` is performed. Calling `dacs_runtime_init()` after a `dacs_runtime_exit()` is only supported for a host process.

RETURN VALUE

The `dacs_runtime_exit` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_DACSD_FAILURE`: unable to communicate with the DaCS daemon services.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_runtime_init(3), dacs_init(3), dacs_exit(3)

Chapter 8. Reservation services

In the DaCS environment, hosts and accelerators have a hierarchical parent-child relationship. This hierarchy forms a logical topology of parents, children, and peers. In DaCS 4.0, only child-related APIs are defined and supported. DaCS only supports access and reservation of immediate children to the calling DE.

Reservation services usage scenarios

The section below describes how to check for the number of available children and then reserve the desired number of child accelerators.

Parent DE	Child DE

Check for the number of available children of the specified type:
`dacs_get_num_avail_children(DACS_DE_SPE,`
`            *num_avail )`
 	
Prior to reserving accelerator children, DaCS applications should query for the number of available children of the desired type. The returned count of available children is volatile and is not guaranteed after the call has completed.
 	
In this case, DaCS is being queried for the number of available SPE accelerator children.	

Reserve the desired number of child accelerators:
`dacs_reserve_children(DACS_DE_SPE,`
`          &num_reserved,`
`          &de_list)`
 	
Reserving accelerator DEs makes them exclusively available to the calling DaCS application. This call is a request to reserve and does not guarantee the specified number of DEs have been reserved.
 	
In this case, a request for *num_reserved* accelerators has been made. On return, *num_reserved* is overloaded to contain the actual number of accelerators reserved. The list of reserved DEs is returned in *de_list*.	

Release DE list

The section below describes how to release the child accelerators previously reserved.

Parent DE	**Child DE**

Release the desired number of child accelerators:
`dacs_release_de_list(`*`num_reserved`*`,`
`         `*`&de_list`*`)`
 	
Once a DaCS application has finished using the reserved accelerator DEs, it must release them so they are available for other DaCS applications to use. In this case, the DaCS application requests *num_reserved* DEs from *de_list* be released.	

dacs_get_num_avail_children

NAME

dacs_get_num_avail_children - Return the number of children of the specified type available to be reserved.

SYNOPSIS

C syntax

DACS_ERR_T dacs_get_num_avail_children (DACS_DE_TYPE_T *type*, **uint32_t** **num_children***)**

Fortran syntax

dacsf_get_num_avail_children (DACS_DE_TYPE_T *type*, **dacs_int32_t** *num_children*, **DACS_ERR_T** *rc*)

Call parameter	
`type`	The type of children to report.

Return parameters	
`num_children`	**C**: a pointer to the number of children available. **Fortran**: number of children available.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_get_num_avail_children` service returns the number of children of the caller of the specified type that are available for reservation.

`type` can be any of:

`DACS_DE_SYSTEMX`: the supervising host for a node.

`DACS_DE_CELL_BLADE`: an entire Cell/B.E. blade. This DE type encapsulates both `DACS_DE_CBE` types within it, making them unavailable for use. This DE type has 16 `DACS_DE_SPE` children.

`DACS_DE_CBE`: a single Cell/B.E. within a blade. Use of this type makes the associated blade unavailable for use. This DE type has 8 `DACS_DE_SPE` children.

`DACS_DE_SPE`: Cell/B.E. Synergistic Processing Element.

Note: This service returns the number of children that were available at the time of the call. The actual number can change any time after the call. The number of children is only returned upon success.

RETURN VALUE

The `dacs_get_num_avail_children` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return. Note that success does not guarantee available children.
- `DACS_ERR_INVALID_ADDR`: invalid pointer.

- `DACS_ERR_INVALID_ATTR`: invalid flag or enumerated constant.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_reserve_children(3), dacs_release_de_list(3)

dacs_reserve_children

NAME

dacs_reserve_children - Reserve children of a specified type.

SYNOPSIS

C syntax

DACS_ERR_T dacs_reserve_children (DACS_DE_TYPE_T *type*, **uint32_t** **num_children*, **de_id_t** **de_list***)**

Fortran syntax

dacsf_reserve_children (DACS_DE_TYPE_T *type*, **dacs_int32_t** *num_children*, **dacs_de_id_t** *de_list*, **DACS_ERR_T** *rc***)**

Call parameters

`type`	The type of children to report.
`num_children`	**C:** a pointer to the number of children requested. **Fortran**: the number of children requested.

Return parameters

`num_children`	**C**: a pointer to the number of children actually reserved. This may be less than or equal to the number requested. **Fortran**: the number of children actually reserved. This may be less than or equal to the number requested.
`de_list`	**C**: a pointer to a location where the list of reserved children is returned. The space for this list must be allocated by the caller and must have enough room for `num_children` entries. **Fortran**: the list of reserved children is returned. The space for this list must be allocated by the caller and must have enough room for `num_children` entries.
`rc`	**Fortran only**: see Return Value.

DESCRIPTION

The `dacs_reserve_children` service attempts to reserve the requested number of children of the specified type. The actual number reserved may be less than or equal to the number requested. The actual number and list of reserved children is returned to the caller.

Be sure to check both the return code and the value returned in `num_children`. A return code of `DACS_SUCCESS` and a value of 0 in `num_children` indicates no children were reserved.

`type` can be any of:

`DACS_DE_SYSTEMX`: the supervising host for a node.

DACS_DE_CELL_BLADE: an entire Cell/B.E. blade. This DE type encapsulates both DACS_DE_CBE types within it, making them unavailable for use. This DE type has 16 DACS_DE_SPE children.

DACS_DE_CBE: a single Cell/B.E. within a blade. Use of this type makes the associated blade unavailable for use. This DE type has 8 DACS_DE_SPE children.

DACS_DE_SPE: Cell/B.E. Synergistic Processing Element.

Cell/B.E.: accepts all valid types, but only returns a non-zero value for DACS_DE_SPE.

Hybrid: returns non-zero (if possible) for DACS_DE_CELL_BLADE and DACS_DE_CBE.

Cell/B.E.: this function is not supported on the SPUs.

RETURN VALUE

The dacs_reserve_children service returns an error indicator defined as:

- DACS_SUCCESS: normal return.
- DACS_ERR_INVALID_ATTR: invalid flag or enumerated constant.
- DACS_ERR_INVALID_ADDR: invalid pointer.
- DACS_ERR_INVALID_SIZE: number of children requested must be greater than zero.
- DACS_ERR_NOT_INITIALIZED: DaCS has not been initialized yet.
- DACS_ERR_NOT_SUPPORTED_YET: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_get_num_avail_children(3), dacs_release_de_list(3)

dacs_release_de_list

NAME

dacs_release_de_list - Release the reservations for a list of DEs.

SYNOPSIS

C syntax

DACS_ERR_T dacs_release_de_list (uint32_t *num_des*, **de_id_t** **de_list*)

Fortran syntax

dacsf_release_de_list (dacs_int32_t *num_des*, **dacs_de_id_t** *de_list*, **DACS_ERR_T** *rc*)

Call parameters

`num_des`	The number of DEs in the list. This must be greater than 0.
`de_list`	**C:** a pointer to the list of DEs to release. **Fortran:** the list of DEs to release.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_release_de_list` service releases the reservation for the specified list of DEs. On successful return all DEs in the list are released (made available). On failure none of the DEs in the list are released.

Cell/B.E.: this function is not supported on the SPUs.

RETURN VALUE

The `dacs_release_de_list` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: invalid pointer.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_RESOURCE_BUSY`: the resource is in use.
- `DACS_ERR_INVALID_SIZE`: invalid list size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_get_num_avail_children(3), dacs_reserve_children(3)

Chapter 9. Process management

This chapter describes the functions for starting, stopping and monitoring processes on DEs.

Current Working Directory in DaCS for Hybrid

A key element in process management is the current working directory on the accelerator file system. DaCS for Hybrid internally determines this for the `dacs_de_start()` call. For environment variables such as `PATH` or `LD_LIBRARY_PATH`, the underlying implementation will substitute the current working directory for `$AE_CWD`.

For example, if the current working directory on the accelerator is `/DACS-TMP/HOME/USER` then `PATH=$AE_CWD:/USR/BIN` points to `/DACS-TMP/HOME/USER:/USR/BIN` on the accelerator file system, and `LD_LIBRARY_PATH=$AE_CWD:/DACS_LIB` points to `/DACS-TMP/HOME/USER:/DACS_LIB`.

Note that when using the current working directory as part of the `DACS_START_ENV_LIST` environment variable it must be preceded by a backslash. For example: `DACS_START_ENV_LIST="LD_LIBRARY_PATH=\$AE_CWD"`.

All files transferred by `dacs_de_start()` are placed in the current working directory. This will be unique across all AE applications on an accelerator.

Note: the file that is stored in the current working directory is in a fully-qualified path in that working directory, so if you were passing the file `/home/joe/dataset` it would be put in `$AE_CWD/home/joe/dataset` directory.

When the launched accelerator process terminates, DaCS clears the working directory by default. A configuration option in `/etc/dacsd.conf` is available to allow retention of the current working directory. See "Configuring the DaCS for Hybrid daemon" on page 22 for further information.

Note: the programmer using `LD_LIBRARY_PATH` may need to incorporate the DaCS libraries, and any required `.so` files, into the environment variable for running accelerator DaCS applications.

Environment variables in DaCS for Hybrid

Hybrid DaCS has specific environment variables in the process issuing the `dacs_de_start()` call. This allows an external program such as a debugging or profiling tool to be started which in turn starts the accelerator process. These variables are described in "ENVIRONMENT" on page 56.

Process control

This topic describes process control.

dacs_num_processes_supported

NAME

dacs_num_processes_supported - Return the number of processes that can be started on a DE.

SYNOPSIS

C syntax

DACS_ERR_T dacs_num_processes_supported (de_id_t *de*, **uint32_t** **num_processes***)**

Fortran syntax

dacsf_num_processes_supported (dacs_de_id_t *de*, **dacs_int32_t** *num_processes*, **DACS_ERR_T** *rc***)**

Call parameter	
`de`	The DE to query.

Return parameters	
`num_processes`	**C**: a pointer to a location where the maximum number of processes that can be started on this DE is stored **Fortran**: the maximum number of processes that can be started on this DE.
`rc`	**Fortran**: see Return value.

Cell/B.E.: on DaCS 4.0 only 1 process is supported per DE.

DESCRIPTION

The `dacs_num_processes_supported` service returns the number of simultaneous processes that can be started on the specified DE. The target DE must have been reserved by the caller.

Cell/B.E.: this is not supported on the SPU.

RETURN VALUE

The `dacs_num_processes_supported` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_TARGET`: the operation is not allowed for the target de.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_de_start(3), dacs_num_processes_running(3), dacs_de_wait(3), dacs_de_test(3)

dacs_num_processes_running

NAME

dacs_num_processes_running - Return the number of processes currently running on a DE.

SYNOPSIS

C syntax

DACS_ERR_T dacs_num_processes_running (de_id_t *de*, **uint32_t** **num_processes***)**

Fortran syntax

dacsf_num_processes_running (dacs_de_id_t *de*, **dacs_int32_t** *num_processes*, **DACS_ERR_T** *rc***)**

Call parameter	
`de`	The DE to query.

Return parameters	
`num_processes`	**C**: a pointer to a location where the number of processes currently running on the target DE is stored. **Fortran**: the number of processes currently running on the target DE.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_num_processes_running` service returns the number of processes currently running on the specified DE. This includes all processes that have been started (with `dacs_de_start()`) and have not yet been reaped (with `dacs_de_test()` or `dacs_de_wait()`). The target DE must have been reserved by the caller.

Cell/B.E: this is not supported on the SPU.

RETURN VALUE

The `dacs_num_processes_running` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_TARGET`: the operation is not allowed for the target de.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_de_start(3), dacs_num_processes_supported(3), dacs_de_wait(3), dacs_de_test(3)

dacs_de_start

NAME

dacs_de_start - Start a process on a DE.

SYNOPSIS

C syntax

DACS_ERR_T dacs_de_start (de_id_t *de*, **void** **prog*, **char const** ***argv*, **char const** ***envv*, **DACS_PROC_CREATION_FLAG_T** *creation_flags*, **dacs_process_id_t** **pid* **)**

Fortran syntax

dacsf_de_start_std_file (dacs_de_id_t *de*, **character** *prog*, **character** *argv*, **integer** *argv_size*, **character** *envv,* **integer** *envv_size*, **DACS_PROC_CREATION_FLAG_T** *creation flags*, **dacs_process_id_t** *pid*, **DACS_ERR_T** *rc***)**

dacsf_de_start_std_embedded(dacs_de_id_t *de*, **external** *prog*, **character** *argv*, **integer** *argv_size*, **character** *envv,* **integer** *envv_size*, **dacs_process_id_t** *pid*, **DACS_ERR_T** *rc***)**

dacsf_de_start_ptr_file(dacs_de_id_t *de*, **character** *prog* , **dacs_pvoid_t** *argv*, **dacs_pvoid_t** *envv*, **DACS_PROC_CREATION_FLAG_T** *creation_flags*, **dacs_process_id_t** *pid*, **DACS_ERR_T** *rc***)**

dacsf_de_start_ptr_embedded (dacs_de_id_t *de*, **external** *prog*, **dacs_pvoid_t** *argv*, **dacs_pvoid_t** *envv*, **dacs_process_id_t** *pid*, **DACS_ERR_T** *rc***)**

Call parameters

`de`	The target DE where the program will execute.
`prog`	**C:** pointer to a filename, handle, or file list that identifies the executable to be initiated as indicated in the specified `creation_flags`. **Fortran:** a filename, handle, or file list that identifies the executable to be initiated as indicated in the specified `creation_flags`.
`argv`	**C:** a pointer to an array of pointers to argument strings (the argument list), terminated by a `NULL` pointer. **Fortran**: platform dependent. Either an array of strings or an address handle created by `dacsf_makevoid`, depending on the `creation_flags` parameter, platform, and Fortran subroutine.
`argv_size`	**Fortran only**: the number of elements in the `argv` array (`dacsf_de_start_std_*` subroutines).
`envv`	**C**: a pointer to an array of pointers to environment variable strings (the environment list), terminated by a `NULL` pointer. **Fortran**: platform dependent. Either an array of strings or an address handle created by `dacsf_makevoid`, depending on the `creation_flags` parameter, platform, and Fortran subroutine.

envv_size	**Fortran only**: the number of elements in the envv array (dacsf_de_start_std_* subroutines).
creation_flags	An implementation-specific flag that specifies how the executable program is found.

Return parameters

pid	**C**: a pointer to a location where the process id is stored on successful return. **Fortran**: the process id on successful return.
rc	**Fortran only**: see Return value.

DESCRIPTION

The dacs_de_start service starts a process on the specified DE. The service can be called several times to start one or more processes on the same DE. The number of processes that can be started on a particular DE is platform dependent and can be determined by calling dacs_num_processes_supported().

The program's main function signature must be compatible with

```
int main(int argc, char *argv[], char *envp[])
```

argv is an array of argument strings passed to the new program. argv[0] is a pointer to the program name, and the remaining argv arguments are initialized from the values passed on the dacs_de_start() argv parameter.

envv is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. In **C**, both argv and envp must be terminated by a null pointer. In **Fortran**, the number of elements in the array is specified in the argv_size and envv_size parameters. envp is initialized from the dacs_de_start() env parameter, from the DACS_START_ENV_LIST environment variable, and includes additional environment variables added by DaCS.

Cell/B.E: the PPU passes the argv and envv pointers directly to the SPUs spu_main() function, and does not copy argument strings into local store.

creation_flags can be any of:
- DACS_PROC_LOCAL_FILE **Hybrid only**: a fully qualified pathname,
- DACS_PROC_LOCAL_FILE_LIST: specifies the name of a file which contains a list of files to transfer to the PPE prior to launching the accelerator process. File names are fully-qualified POSIX-compliant pathname files.
- DACS_PROC_REMOTE_FILE **Hybrid only**: a fully qualified path on a remote system
- DACS_PROC_EMBEDDED **Cell/B.E. only**: the handle of an embedded executable image.

When a user uses DACS_PROC_LOCAL_FILE_LIST they pass a list of files to transfer to the AE. For example:

```
/tmp/helloworld
/tmp/libhello.o
```

If the user places a '!' character as the first character on a line then the file is not transferred but is assumed to be on the AE already. This is useful when the program to be executed already resides on the AE, but the SPU program and other files need to be transferred.

```
!/tmp/helloworld
/tmp/libhello.o
```

If the first line (the binary program to execute) is marked with an '!' then DACS does not assume it is in the current working directory, in other words it does not prepend the $AE_CWD string to the file name.

Cell/B.E.: this function is not supported on the SPUs.

Fortran: For further Fortran examples see: "dacsf_de_start examples" on page 205

RETURN VALUE

The `dacs_de_start` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: a pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: a flag or enumerated constant is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_TARGET`: the operation is not allowed for the target DE.
- `DACS_ERR_PROC_LIMIT`: the maximum number of processes supported has been reached.
- `DACS_ERR_INVALID_PROG`: the specified program could not be executed.
- `DACS_ERR_INVALID_CWD`: the `ae_cwd_prefix` specified in the `dacsd.conf` file is invalid.
- `DACS_ERR_NOT_FOUND`: program file not found.
- `DACS_ERR_TOO_LONG`: program pathname is too long.
- `DACS_ERR_VERSION_MISMATCH`: the host and accelerator applications have incompatible software versions.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NO_PERM`: no permission to the specified program or path.
- `DACS_ERR_RESOURCE_BUSY`: specified program is busy.
- `DACS_ERR_TERM`: program completed without calling `dacs_init()`.
- `DACS_ERR_ARCH_MISMATCH`: attempting to run a 64-bit AE program with a 32-bit HE application or vice-versa.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

ENVIRONMENT

Hybrid only: DaCS for Hybrid has specific environment variables in the process issuing the `dacs_de_start()` call. This allows an external program such as a debugging or profiling tool to be started which in turn starts the accelerator process.

These variables are:

DACS_START_PARENT
: specifies the command used to start an auxiliary program which starts the accelerator process. Within the command, %e , %a and %p are replaced respectively by the accelerator executable name, the accelerator arguments, and the parent's listening port value.

  For example, given:

  ```
  DACS_START_PARENT="/usr/bin/gdb --args %e %a"
  ```

  then:

  ```
  dacs_de_start (de, "myaccel", "myargs", 0, ppid)
  ```

  would launch the command:

  ```
  /usr/bin/gdb --args myaccel myargs
  ```

DACS_START_FILES
: specifies the name of a file which contains a list of files to transfer to the PPE prior to launching the accelerator process. File names are fully-qualified POSIX-compliant pathname files.

DACS_START_ENV_LIST
: specifies an additional list of environment variables for the initial program spawn on the accelerator. List items are separated by semicolons. An example of the format is:

  ```
  ENV1=VAL1;ENV2=VAL2;QSHELL_*;ENV3
  ```

  where:

  - ENV1 and ENV2 are the environment variables and *VAL1* and *VAL2* are their respective settings,
  - *QSHELL_** means pull all environment variables prefixed with *QSHELL_* from the present environment, and
  - ENV3 means pull the environment variable from the present environment and pass on.

  Delete functions, such as <name>= and <prefix>*= to drop environment variables by name or prefix, are not supported in DaCS 4.0.

DACS_PARENT_PORT
: specifies the value of %p to pass in the dacs_de_start() call. This value is post-incremented in the environment so that it is one more on the next dacs_de_start() call.

  Note: the port allocated by DACS_PARENT_PORT is solely the responsibility of the environment setter and is not guaranteed to be available on the accelerator OS.

In the execution environment, the environment variables in DACS_START_ENV_LIST will be a list appended to the environment variables in the list under parameter char const **envv.

The use of duplicate environment variables across the lists in the dacs_de_start() service and DACS_START_ENV_LIST is possible. However the value that will be used is implementation dependent, because accessing environment variables is implementation dependent.

dacs_de_start executes the program pointed to by prog. prog must be either a binary executable, or a script starting with a line of the form #! interpreter [arg].

In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, but which is invoked as `interpreter [arg] <filename>`.

SEE ALSO

dacs_num_processes_supported(3), dacs_num_processes_running(3), dacs_de_wait(3), dacs_de_test(3), dacsf_makevoid(3), dacs_de_kill(3)

dacs_de_test

NAME

dacs_de_test - Test the status of a DE process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_de_test (de_id_t *de*, **dacs_process_id_t** *pid*, **int32_t** **exit_status***)**

Fortran syntax

dacsf_de_test (dacs_de_id_t *de*, **dacs_process_id_t** *pid*, **dacs_int32_t** *exit_status*, **DACS_ERR_T** *rc***)**

Call parameters	
`de`	The target DE.
`pid`	The target process.

Return parameters	
`exit_status`	**C**: a pointer to the location where the exit code or signal number is stored depending on the status of the DE. **Fortran**: the exit code or the signal number, depending on the status of the DE.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_de_test()` service returns either the current state of the target DE process or an error if the call was unsuccessful. If successful, the returned state can be any one of the `DACS_STS_*` return values mentioned below. The remainder of return values are consider errors. This operation is only supported for child processes started by the caller.

Once this service or `dacs_de_wait()` returns a non-busy status, the status is considered to be consumed and will no longer be available. Subsequent calls to query the status will fail with `DACS_ERR_INVALID_PID`.

If this service or `dacs_de_wait()` are not called to reap the process, the process is still considered to be a running process (see "dacs_num_processes_running" on page 53). This could prevent future attempts to start new processes, on the DE, if the number of running processes matches the maximum number supported.

The exit_status return value varies depending on the status returned for the child DE. If the child DE terminated with a `DACS_STS_PROC_FAILED` status then the value returned through exit_status is the non-zero exit code returned from the child executable. If the child DE terminated with a `DACS_STS_PROC_ABORTED` status then the value returned through exit_status is a platform-specific exception code.

Hybrid: in a hybrid environment the standard Linux/UNIX signal number, which caused the termination, is returned through exit_status.

Cell/B.E.: in a Cell/B.E environment the exit_code, as defined by libspe2's stop_info, is returned through exit_status.

Cell/B.E: `dacs_de_test` is not supported on the SPU.

RETURN VALUE

The `dacs_de_test` service returns an error indicator defined as:

- `DACS_STS_PROC_RUNNING`: The process is still active. `exit_status` is unmodified.
- `DACS_STS_PROC_FINISHED`: The process completed successfully. `exit_status` is zero.
- `DACS_STS_PROC_FAILED`: The process exited with an error. `exit_status` contains the process exit code.
- `DACS_STS_PROC_KILLED`: process was killed using `dacs_de_kill` .
- `DACS_STS_PROC_ABORTED`: the process terminated abnormally. `exit_status` contains the terminating signal code.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to a valid process.
- `DACS_ERR_INVALID_TARGET`: the operation is not allowed for the target process.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_DACSD_FAILURE`: unable to communicate with DaCS daemon services.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

Note: If the return value is `DACS_STS_PROC_RUNNING` then the `exit_status` is not modified.

SEE ALSO

dacs_de_start(3), dacs_num_processes_supported(3), dacs_num_processes_running(3), dacs_de_wait(3)

dacs_de_wait

NAME

dacs_de_wait - Wait on the completion of a DE process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_de_wait (de_id_t *de*, **dacs_process_id_t** *pid*, **int32_t** **exit_status***)**

Fortran syntax

dacsf_de_wait (dacs_de_id_t *de*, **dacs_process_id_t** *pid*, **dacs_int32_t** *exit_status*, **DACS_ERR_T** rc**)**

Call parameters	
`de`	The target DE.
`pid`	The target process.

Return parameters	
`exit_status`	**C**: A pointer to the location where the exit code or signal number is stored depending on the status of the DE. **Fortran:** the exit code or signal number depending on the status of the DE.
`rc`	**Fortran only**: see Return value below.

DESCRIPTION

The `dacs_de_wait` service returns the status of the target process if it was successful, or an error code if not. If the process is running at the time of the call, the call blocks until it finishes execution. If the process has finished execution at the time of the call, the call does not block.

Once this service or `dacs_de_test()` returns a non-busy status, the status is considered to be consumed and will no longer be available. Subsequent calls to query the status will fail with `DACS_ERR_INVALID_PID`.

If this service or `dacs_de_test()` are not called to reap the process, the process is still considered to be a running process (see "dacs_num_processes_running" on page 53). This could prevent future attempts to start new processes, on the DE, if the number of running processes matches the maximum number supported.

The `exit_status` return value varies depending on the status returned for the child DE. If the child DE terminated with a `DACS_STS_PROC_FAILED` status then the value returned through `exit_status` is the non-zero exit code returned from the child executable. If the child DE terminated with a `DACS_STS_PROC_ABORTED` status then the value returned through `exit_status` is a platform-specific exception code.

Hybrid: in a hybrid environment the standard Linux/UNIX signal number, which caused the termination, is returned through `exit_status`.

Cell/B.E.: in a Cell/B.E environment the `exit_code`, as defined by libspe2's `stop_info`, is returned through `exit_status`.

Cell/B.E.: `dacs_de_wait` is not supported on the SPU.

RETURN VALUE

The `dacs_de_wait` service returns an error indicator defined as:

- `DACS_STS_PROC_FINISHED`: the process finished execution without error.
- `DACS_STS_PROC_FAILED`: the process exited with a failure.
- `DACS_STS_PROC_ABORTED`: the process terminated abnormally.
- `DACS_STS_PROC_KILLED`: process was killed via `dacs_de_kill`.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to a valid process.
- `DACS_ERR_INVALID_TARGET`: the operation is not allowed for the target process.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_DACSD_FAILURE`: unable to communicate with DaCS daemon services.
- `DACS_ERR_NOT_SUPPORTED_YET`: The DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_de_start(3), dacs_num_processes_supported(3), dacs_num_processes_running(3), dacs_de_test(3)

dacs_de_kill (prototype)

NAME

dacs_de_kill - Kill a process on a DE.

SYNOPSIS

C syntax

DACS_ERR_T dacs_de_kill (de_id_t *de*, **dacs_process_id_t** *pid*, **DACS_KILL_TYPE_T** *flag***)**

Fortran syntax

dacsf_de_kill (dacs_de_id_t *de*, **dacs_process_id_t** *pid*, **DACS_KILL_TYPE_T** *flag*, **DACS_ERR_T** *rc***)**

Call parameters	
`de`	The de_id of the AE to terminate.
`pid`	The process id of the process to kill.
`flag`	An implementation-specific flag: `DACS_KILL_TYPE_ASYNC` The `DACS_KILL_TYPE_ASYNC` flag protects internal DaCS data structures, but does not protect application-critical sections. An error handler registered with dacs_errhandler_reg is not called with this flag.

Return parameter	
`rc`	**Fortran only**: see Return value

DESCRIPTION

`dacs_de_kill` is a prototype and only supported on DaCS for Hybrid.

`dacs_de_kill` requests the termination of the specified AE process, identified by its DE and PID. Only AE processes started by the calling process can be killed. The call to `dacs_de_kill()` must be followed by a call to either `dacs_de_wait()` or `dacs_de_test()` to complete the termination.

Cell/B.E.: `dacs_de_kill()` is not a supported on a PPU HE.

RETURN VALUE

The `dacs_de_kill` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID is either invalid or not reserved.
- `DACS_ERR_INVALID_TARGET`: the operation is not allowed for the target DE.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_NOT_INITIALIZED`: The DaCS environment is not initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: The DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_num_processes_supported(3), dacs_num_processes_running(3), dacs_de_wait(3), dacs_de_test(3)

Process synchronization

Groups

Group functions allow you to organize processes into groups so that they can be treated as a single entity.

Definitions

Group

A group is a collection of processes which can be involved in a collective operation, such as a barrier.

Group Member

A group member is a process, uniquely identifiable by its DE and PID combination.

Group design

Membership

A process can only be added to a group by the group creator/owner. The group owner shares the group handle, generated when the group was created, with the process to be added. In turn, the process being added must accept membership to the group. Accepting membership means the process will participate in group operations, such as barriers. Prior to exiting, the process must request to leave.

Group Leader/Owner

Creating a group through `dacs_group_init()` implicitly makes the calling DE the owner. Group ownership does not imply membership, so the owner must add itself if it wishes to participate in group operations. The owner is responsible for adding members to the group and destroying the group when all members have left the group.

Note: In DaCS 4.0, groups can only be created on an HE and only its direct children may be added.

When DaCS for Hybrid is working with DaCS for Cell/B.E., creation of a group on a PPU is done in the PPU's role as an HE. This means that only it and the SPU AEs can be members of the group. Trying to share the group with the x86_64 HE (which is done as a PPU AE) will fail.

Barriers

Barriers provide synchronization among the participating members of a group. A barrier is an implied resource associated with being in a group; they are not allocated, initialized, shared or destroyed.

Group usage scenarios

Group operations can be classified into three stages: initialization, operation and termination. An example showing the services used in these stages follows.

Initialization

The following steps, in this order, would be used by the group owner and members to create and join a group.

Owner	Members
Create the group: `dacs_group_init( &group, flags );` This creates an opaque group handle. The handle will then used by all members during group operations.	
Add members (identified by DE and PID) to the group, one by one: `dacs_group_add_member( de, pid, group );`	
	Accept their addition, individually: `dacs_group_accept( de, pid, &group );`

(Optional) Add itself to the group:
`dacs_group_add_member( DACS_DE_SELF,`
`        DACS_PID_SELF, group );`
 	
(This does not require an accept response.)	

Close the initialization of the group: `dacs_group_close( group );`	

Operation

Group operations are controlled by *barriers*. These are used to synchronize the processing by different members of the group. If it is necessary to ensure that no member enters a new stage of processing before other members are ready then each member must make a wait call. Each member will then be blocked until all members have made this call. When the last member is accounted for, all members will be released.

Owner	Members
(Optional) Wait on barrier, individually: `dacs_barrier_wait( group )` If the owner added itself to the group, then it too must wait on the barrier.	Wait on barrier, individually: `dacs_barrier_wait( group );`

Termination

The following steps, in this order, would be used by the group owner and members to remove a group.

Owner	Members
Destroy the group: `dacs_group_destroy( &group );`	
	Leave the group, individually: `dacs_group_leave( &group );`

dacs_group_init

NAME

dacs_group_init - Initialize a DaCS group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_group_init (dacs_group_t **group*, **uint32_t** *flags***)**

Fortran syntax

dacsf_group_init (dacs_group_t *group*, **dacs_int32_t** *flags*, **DACS_ERR_T** *rc***)**

Call parameter

`flags`	Flags for group initialization.

Return parameters

`group`	**C**: a pointer to a group handle which is filled in upon successful return. **Fortran**: a group handle which is filled in upon successful return.
`rc`	**Fortran Only**: see Return Value.

Note: In DaCS 4.0 no flags will be supported and the flags value passed in must be zero.

DESCRIPTION

The `dacs_group_init` service initializes a DaCS group and returns a handle to the group. The calling process is the owner of the group. The owner process is not a member of the group by default, but may add itself as a member.

Note: In DaCS 4.0, only an HE can initiate a group.

RETURN VALUE

The `dacs_group_init` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_NO_RESOURCE`: could not allocate required resources.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_group_add_member(3), dacs_group_close(3), dacs_group_destroy(3), dacs_group_accept(3), dacs_group_leave(3), dacs_barrier_wait(3)

dacs_group_add_member

NAME

dacs_group_add_member - Add a member to a DaCS group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_group_add_member (de_id_t *de*, **dacs_process_id_t** *pid*, **dacs_group_t** *group***)**

Fortran syntax

dacsf_group_add_member (dacs_de_id_t *de*, **dacs_process_id_t** *pid*, **dacs_group_t** *group*, **DACS_ERR_T** *rc***)**

Call parameters

`de`	The DE of the member to add. The group owner must specify a value of `DACS_DE_SELF` to add itself.
`pid`	The process ID of the member to add. The group owner must specify a value of `DACS_PID_SELF` to add itself.
`group`	The handle of the group to which the new member is to be added.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_group_add_member` service adds the specified `DE/PID` as a member of the specified group. This service can only be called by the process which owns the group. If the owner process is adding itself the service returns immediately. If the member to be added is not the owner of the group this service blocks, waiting for an associated `dacs_group_accept()` call from the new member.

Note: This function is only supported on HE.

RETURN VALUE

The `dacs_group_add_member` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified `DE` is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified `PID` does not refer to an active process.
- `DACS_ERR_INVALID_HANDLE`: the group handle does not refer to a valid group.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_GROUP_CLOSED`: the group is closed.
- `DACS_ERR_GROUP_DUPLICATE`: the specified process is already a member of the specified group.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_OWNER`: the caller is not the owner of the group.

- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_group_init(3), dacs_group_close(3), dacs_group_destroy(3), dacs_group_accept(3), dacs_group_leave(3), dacs_barrier_wait(3)

dacs_group_close

NAME

dacs_group_close - Close a DaCS group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_group_close (dacs_group_t *group***)**

Fortran syntax

dacsf_group_close (dacs_group_t *group***, DACS_ERR_T** *rc***)**

Call parameter	
`group`	The handle of the group to close.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_group_close` service closes the specified group, so no new members may be added. The specified group must have been initialized with `dacs_group_init()`. Only the group owner may close the group. Group member operations will block until the group is closed.

Note: This function is only supported on HE.

RETURN VALUE

The `dacs_group_close` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_HANDLE`: the group handle does not refer to a valid group.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_OWNER`: the caller is not the owner of the group.
- `DACS_ERR_GROUP_CLOSED`: the group is already closed.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_group_init(3), dacs_group_add_member(3), dacs_group_destroy(3), dacs_group_accept(3), dacs_group_leave(3), dacs_barrier_wait(3)

dacs_group_destroy

NAME

dacs_group_destroy - Remove a DaCS group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_group_destroy (dacs_group_t **group***)**

Fortran syntax

dacsf_group_destroy (dacs_group_t *group***, DACS_ERR_T** *rc***)**

Call parameter	
`group`	**C**: a pointer to the handle of the group to remove.
	Fortran: the handle of the group to remove.

Return parameter	
`rc`	**Fortran**: see Return value.

DESCRIPTION

The `dacs_group_destroy` service removes the specified group and invalidates the handle. This service may only be called by the owner of the group, and blocks until all members have left the group. Group owners that were also added as members do not need to call `dacs_group_release()`, as an implicit release is performed on destroy.

Note: This function is only supported on HE.

RETURN VALUE

The `dacs_group_destroy` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_HANDLE`: the group handle does not refer to a valid group.
- `DACS_ERR_NOT_OWNER`: the caller is not the owner of the group.
- `DACS_ERR_GROUP_OPEN`: the group has not been closed.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_group_init(3), dacs_group_add_member(3), dacs_group_close(3), dacs_group_accept(3), dacs_group_leave(3), dacs_barrier_wait(3)

dacs_group_accept

NAME

dacs_group_accept - Accept membership to a DaCS group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_group_accept (de_id_t *de*, **dacs_process_id_t** *pid*, **dacs_group_t** **group***)**

Fortran syntax

dacsf_group_accept (dacs_de_id_t *de*, **dacs_process_id_t** *pid*, **dacs_group_t** *group*, **DACS_ERR_T** *rc* **)**

Call parameters

`de`	The DE of the group owner.
`pid`	The PID of the group owner.

Return parameters

`group`	**C**: a pointer to the handle of the group to be filled in. **Fortran**: the handle of the group to be filled in.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_group_accept` service accepts membership to a group and returns the group handle. For each `dacs_group_accept()` call there must be an associated `dacs_group_add_member()` call by the owner of the group. This service blocks until the caller has been added to the group by the group owner.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

RETURN VALUE

The `dacs_group_accept` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified `pid` does not refer to an active process.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_TARGET`: the operation not allowed for the target process.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_group_init(3), dacs_group_add_member(3), dacs_group_close(3), dacs_group_destroy(3), dacs_group_leave(3), dacs_barrier_wait(3)

dacs_group_leave

NAME

dacs_group_leave - Request from a member to leave a DaCS group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_group_leave (dacs_group_t **group***)**

Fortran syntax

dacsf_group_leave (dacs_group_t *group* **, DACS_ERR_T** *rc***)**

Call parameter	
`group`	**C**: a pointer to the handle of the group to leave. **Fortran**: the handle of the group to leave.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_group_leave` service removes the calling process from the specified group. All members other than the owner must leave the group before it can be destroyed. The specified group handle is invalidated upon successful return. This service does not block unless the group is not yet closed.

RETURN VALUE

The `dacs_group_leave` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_HANDLE`: the group handle does not refer to a valid DaCS group.
- `DACS_ERR_OWNER`: the owner of the group may not leave it.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_group_init(3), dacs_group_add_member(3), dacs_group_close(3), dacs_group_destroy(3), dacs_group_accept(3), dacs_barrier_wait(3)

dacs_barrier_wait

NAME

dacs_barrier_wait - Synchronize members of a group.

SYNOPSIS

C syntax

DACS_ERR_T dacs_barrier_wait (dacs_group_t *group***)**

Fortran syntax

dacsf_barrier_wait (dacs_group_t *group***, DACS_ERR_T** *rc***)**

Call parameter	
`group`	The handle of the group with which to synchronize.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_barrier_wait` service blocks the caller on a group barrier until all members in the group have reached the barrier. The caller must be a member of the specified group.

RETURN VALUE

The `dacs_barrier_wait` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_HANDLE`: the group handle does not refer to a valid DaCS group.
- `DACS_ERR_NO_PERM`: caller is not a member of the group.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_group_init(3), dacs_group_add_member(3), dacs_group_close(3), dacs_group_destroy(3), dacs_group_accept(3), dacs_group_leave(3)

Mutexes

Shared data accesses can be serialized with DaCS by using a **mu**tual **ex**clusion primitive (mutex) to protect critical sections. A mutex is hosted from the creating DEs memory space and is controlled, using the provided services, by both local and remote processes.

The significant features of a mutex are:

- the mutex is not recursive
- it is held by a particular DE/PID and that DE/PID is the only one that can unlock it
- the lock is not thread based within the DE/PID. Any thread in the locking DE/PID can unlock the mutex.

Sharing Mutexes It is possible to created a mutex so that it is accessible across the entire hierarchy (x86, PPU, and, SPU). In this case, the mutex should be created on the PPE and shared up to the HE (the x86) and also down to the AEs (SPUs). The mutex can then be used to synchronize processes across all three levels.

The services which process a mutex fall into two categories:

- Mutex management services, for managing the mutex shared resource, which include `dacs_mutex_init()`, `dacs_mutex_share()`, `dacs_mutex_accept()`, `dacs_mutex_release()` and `dacs_mutex_destroy()`
- Mutex locking services, for locking and unlocking a mutex, which include `dacs_mutex_lock()`, `dacs_mutex_unlock()` and `dacs_mutex_try_lock()`.

Mutex owner functions

Using mutex owner functions.

Mutex owner usage scenarios

The table below describes how to create, share and accept the mutex.

Creator DE	User DE
Create a mutex: `dacs_mutex_init(&mutex)` Create a mutex to be used for synchronizing process access to a critical resource. On return, a handle to the created mutex is returned.	.
Share the mutex with the synchronized DE/PIDs: `dacs_mutex_share(accepter_de, accepter_pid, mutex)` The mutex must be explicitly shared with each DE/PID that will use it for synchronization. The request to share a mutex will block until the corresponding accept is performed.	Accept the shared mutex: `dacs_mutex_accept(creator_de, creater_pid, &mutex)` Each DE/PID that needs to use the mutex for synchronization, must explicitly accepted it from the sharing DE/PID. A handle to the shared mutex will be returned.

Destroying a mutex: The table below shows how to destroy a mutex

Creator DE	User DE
Destroy the mutex: `dacs_mutex_destroy(&mutex)` The creator of a mutex must destroy it when it is no longer needed. Since the mutex had been shared for remote process use, the call blocks until the mutex has been released by all DEs it was shared with.	Release the mutex: `dacs_mutex_release(&mutex)` Every mutex that was accepted must be released when no longer needed. Every accepter of the mutex must release it once finished, so the creator knows when it is safe to destroy it.

Mutex synchronization

This section describes mutex synchronization.

Locking and unlocking a mutex: The table below describes how to lock and unlock mutexes.

Creator DE	User DE
Lock the shared mutex: `dacs_mutex_lock(mutex)` Locking the mutex blocks all other users of the mutex until it has been unlocked. This synchronizes access to critical sections of code. In this case, the mutex is available, so this DE/PID does not need to block, and returns with the mutex held.	
	Try to lock the shared mutex: `dacs_mutex_trylock(mutex)` To avoid blocking on a held mutex, simply try to lock it. If the lock is held, this call will return that the mutex is busy and another lock attempt may be performed later. If the mutex is available for locking, then the mutex is reserved. In this case, the mutex is already held by the creator DE/PID, thus the attempt will return as busy.
	Lock the shared mutex: `dacs_mutex_lock(mutex)` Locking the mutex blocks all other users of the mutex until it has been unlocked. This synchronizes access to critical sections of code. In this case the mutex is held by the creator DE/PID, so this DE/PID will block until the mutex is available

Creator DE	User DE
Unlock the shared mutex: `dacs_mutex_unlock(mutex)` Unlocking the mutex makes it available for use by potential waiters. Any waiters for the mutex can now retry obtaining the mutex once again. In this case, there is a user DE/PID blocked waiting on the release of the mutex. Once completely unlocked, the mutex is available for the blocked DE/PIDs to obtain.	
	Unlock the shared mutex: `dacs_mutex_unlock(mutex)` Unlocking the mutex makes it available for use by potential waiters. Any waiters for the mutex can now retry obtaining the mutex once again. In this case there are no DE/PIDS blocked waiting on the mutex, so it is simply released and available for the next locker.

dacs_mutex_init

NAME

dacs_mutex_init - Initialize a mutual exclusion variable.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_init (dacs_mutex_t **mutex***)**

Fortran syntax

dacsf_mutex_init (dacs_mutex_t *mutex***, DACS_ERR_T** *rc***)**

Return parameters

`mutex`	**C**: a pointer to a newly initialized mutex handle. **Fortran**: a newly initialized mutex handle.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_init` service initializes a mutual exclusion variable and returns a handle to it.

Cell/B.E.: this function is not supported on the SPUs.

RETURN VALUE

The `dacs_mutex_init` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources, such as memory.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mutex_share(3), dacs_mutex_accept(3), dacs_mutex_lock(3), dacs_mutex_try_lock(3), dacs_mutex_unlock(3), dacs_mutex_release(3), dacs_mutex_destroy(3)

dacs_mutex_share

NAME

dacs_mutex_share - Share a mutual exclusion variable with a remote process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_share (de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mutex_t** *mutex***)**

Fortran syntax

dacsf_mutex_share (dacs_de_id_t *dst_de*, **dacs_process_id_t** *dst_pid* **, dacs_mutex_t** *mutex*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_de`	The target DE for the share.
`dst_pid`	The target process for the share.
`mutex`	The handle of the mutex that is to be shared.

Return parameter	
`rc`	**Fortran only**: see Return value

DESCRIPTION

The `dacs_mutex_share` service shares the specified mutual exclusion variable between the current process and the remote process specified by `dst_de` and `dst_pid`. This service blocks the caller, waiting for the remote process to call `dacs_mutex_accept()` to accept the mutex.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: this function is not supported on the SPUs.

RETURN VALUE

The `dacs_mutex_share` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return; sharing succeeded.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_INVALID_HANDLE`: the specified mutex handle is not valid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources, such as memory.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_accept(3), dacs_mutex_lock(3), dacs_mutex_try_lock(3), dacs_mutex_unlock(3), dacs_mutex_release(3), dacs_mutex_destroy(3)

dacs_mutex_destroy

NAME

dacs_mutex_destroy - Destroy a mutual exclusion variable.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_destroy (dacs_mutex_t ***mutex*****)**

Fortran syntax

dacsf_mutex_destroy (dacs_mutex_t *mutex*, **DACS_ERR_T** *rc***)**

Call parameter	
mutex	**C**: a pointer to the handle of the mutex to destroy. **Fortran**: the handle of the mutex to destroy.

Return parameter	
rc	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_destroy` service destroys the specified mutex and invalidates the handle. This service blocks until all users of the mutex have released it. The mutex may only be destroyed by the process that initialized it (the owner).

The destroy will succeed whether or not the mutex is held by its owner.

Cell/B.E.: this function is not supported on the SPUs.

RETURN VALUE

The `dacs_mutex_destroy` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_HANDLE`: the specified mutex handle is invalid.
- `DACS_ERR_NOT_OWNER`: this operation is only valid for the owner of the resource.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_share(3), dacs_mutex_accept(3), dacs_mutex_lock(3), dacs_mutex_try_lock(3), dacs_mutex_unlock(3), dacs_mutex_release(3)

dacs_mutex_accept

NAME

dacs_mutex_accept - Accept access to a shared mutual exclusion variable from a remote process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_accept (de_id_t *remote_de*, **dacs_process_id_t** *remote_pid*, **dacs_mutex_t** **received_mutex***)**

Fortran syntax

dacsf_mutex_accept (dacs_de_id_t *remote_de*, **dacs_process_id_t** *remote_pid*, **dacs_mutex_t** *received_mutex*, **DACS_ERR_T** *rc***)**

Call parameters	
`remote_de`	The source DE which is sharing the mutex handle.
`remote_pid`	The source PID which is sharing the mutex handle.

Return parameters	
`received_mutex`	**C**: A pointer to the handle of the accepted mutex. **Fortran**: the handle of the accepted mutex.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_accept` service receives a mutual exclusion variable from a remote process. The service blocks until the remote process shares the mutex with a call to `dacs_mutex_share()`.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

RETURN VALUE

The `dacs_mutex_accept` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_share(3), dacs_mutex_lock(3), dacs_mutex_try_lock(3), dacs_mutex_unlock(3), dacs_mutex_release(3), dacs_mutex_destroy(3)

dacs_mutex_release

NAME

dacs_mutex_release - Release a mutual exclusion variable.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_release (dacs_mutex_t **mutex***)**

Fortran syntax

dacsf_mutex_release (dacs_mutex_t *mutex***, DACS_ERR_T** *rc***)**

Call parameter

`mutex`	**C**: a pointer to the handle of the mutex to release. **Fortran**: the handle of the mutex to release.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_release` service releases a previously accepted mutex object and invalidates the handle. When all accepters have released the mutex, it may be destroyed by its owner. This service does not block.

The release will succeed whether or not the mutex is held by the caller.

RETURN VALUE

The `dacs_mutex_release` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_HANDLE`: the specified mutex handle is invalid.
- `DACS_ERR_OWNER`: this operation is not allowed for the owner of the resource.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_share(3), dacs_mutex_accept(3), dacs_mutex_lock(3), dacs_mutex_try_lock(3), dacs_mutex_unlock(3), dacs_mutex_destroy(3)

dacs_mutex_lock

NAME

dacs_mutex_lock - Acquire a lock on a mutual exclusion variable.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_lock (dacs_mutex_t *mutex***)**

Fortran syntax

dacsf_mutex_lock (dacs_mutex_t *mutex***, DACS_ERR_T** *rc***)**

Call parameter	
`mutex`	The handle of the mutex to lock.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_lock` service acquires the specified mutex. The caller must either be the owner of the mutex, or have previously accepted the mutex with a call to `dacs_mutex_accept()`. This service blocks the caller until the mutex is acquired.

RETURN VALUE

The `dacs_mutex_lock` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return; lock succeeded.
- `DACS_ERR_INVALID_HANDLE`: the specified mutex handle is not valid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_share(3), dacs_mutex_accept(3), dacs_mutex_try_lock(3), dacs_mutex_unlock(3), dacs_mutex_release(3), dacs_mutex_destroy(3)

dacs_mutex_try_lock

NAME

dacs_mutex_try_lock - Attempt to acquire a lock on a mutual exclusion variable.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_try_lock (dacs_mutex_t *mutex***)**

Fortran syntax

dacsf_mutex_try_lock (dacs_mutex_t *mutex***, DACS_ERR_T** *rc***)**

Call parameter	
`mutex`	The handle of the mutex to lock.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_try_lock()` service attempts to acquire the specified mutex. The caller must be the owner of the mutex or previously accepted the mutex with a call to `dacs_mutex_accept()`. This service does not block. Instead, if the mutex is unavailable at the time of the call, it will immediately return a `DACS_ERR_MUTEX_BUSY` error to the caller. If the lock was successfully acquired, the call will return a success status with the mutex held.

RETURN VALUE

The `dacs_mutex_try_lock` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return; lock was acquired.
- `DACS_ERR_MUTEX_BUSY`: the mutex is not available.
- `DACS_ERR_INVALID_HANDLE`: the specified mutex handle is not valid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_share(3), dacs_mutex_accept(3), dacs_mutex_lock(3), dacs_mutex_unlock(3), dacs_mutex_release(3), dacs_mutex_destroy(3)

dacs_mutex_unlock

NAME

dacs_mutex_unlock - Unlock a mutual exclusion variable.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mutex_unlock (dacs_mutex_t *mutex***)**

Fortran syntax

dacsf_mutex_unlock (dacs_mutex_t *mutex***, DACS_ERR_T** *rc***)**

Call parameter	
`mutex`	The handle of the mutex to unlock.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mutex_unlock` service unlocks a mutex. The caller must either be the owner of the mutex, or have previously accepted the mutex with a call to `dacs_mutex_accept()`.

RETURN VALUE

The `dacs_mutex_unlock` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return; unlock succeeded.
- `DACS_ERR_INVALID_HANDLE`: the specified mutex handle is not valid.
- `DACS_ERR_NO_PERM`: the requester did not have the lock.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mutex_init(3), dacs_mutex_share(3), dacs_mutex_accept(3), dacs_mutex_lock(3), dacs_mutex_try_lock(3), dacs_mutex_release(3), dacs_mutex_destroy(3)

Chapter 10. Data communication

DaCS provides communication services for both one-sided (rDMA) and two-sided (mailboxes and message passing) data transfers. The general characteristics of these services are described here, with the details in the following reference sections.

Data transfers are performed between two process, identified by the source or destination DE and PID, as appropriate. As a convenience to the programmer, the special values `DACS_DE_PARENT` and `DACS_PID_PARENT` are defined, which can be used to refer to the parent DE and PID respectively. The special values `DACS_DE_SELF` and `DACS_PID_SELF` are also provided for those interfaces where the caller is the target of the operation.

DaCS does not impose any limitation on the size of a transfer, so the size of transfer that can actually be performed is anywhere from zero up to the maximum size imposed by the inherent system limitations.

To accommodate transfers across systems with different data representation formats (endian-ness), the services provide an option for byte swapping.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

When byte swapping is used, the data being swapped must be naturally aligned on the source system, to the alignment implied by the chosen flag. Failure to properly align the data addresses, offsets, or sizes in accordance with the specified swap flag could result in a failure.

Cell/B.E: Although DaCS on Cell supports transfers of unaligned data with no specific size limitations, best performance will be achieved when the native alignment rules are followed. This means that for transfers 16-bytes or greater, the size of transfer is a multiple of 16 bytes and the source and destination buffer are 16-byte aligned. For transfers that are less than 16 bytes, the size of the transfer is a power of 2, and the source and destination buffer have the same relative 16-byte offset. For DMA list operations however, the size of each element must be multiple of 16 bytes and it must be 16-byte aligned.

Note: In DaCS 4.0 direct communication is only allowed between a parent and its children. Attempts to communicate with yourself or with a process which is not the parent or child of the initiator will result in an error of `DACS_ERR_INVALID_TARGET`.

Remote Direct Memory Access

DaCS provides two sets of rDMA function sets in order to improve ease of use while also allowing maximum performance. The two sets of functions differ in whether the local memory, used in an rDMA transfer, needs to be registered with DaCS. Based on this difference, the two sets of functions will be referred to as registered and unregistered local region functions.

Note: Memory regions used for these two function sets are not compatible and cannot be interchanged.

In both sets of functions, at least one memory region must be created and shared with a process that is remote to the region. This is commonly referred to as the remote memory region. For example, if an HE wants to use put to move data into an AEs memory, the AE must first create a memory region and then share the memory region with the HE (which accepts it).

Another similarity in the two function sets is that certain preparation must be performed on the local memory to be used in conjunction with the shared remote memory region. This preparation may include pinning the memory so that it does not get paged out while the rDMA is being performed. The method used to prepare memory can impact performance, which differentiates the two sets of rDMA functionality.

In the case where the local memory is unregistered, the local memory preparation and platform specific limitations are handled inside of the DaCS library, unbeknownst to the user. These are referred to as the unregistered local region functions. This set of functionality provides a convenient more automated means for performing the DMAs at the expense of decreased performance.

In the case where the local memory is registered, the application is responsible for explicitly identifying the local memory regions to be used in rDMA transfers. Platform limitations, such as the number of available memory regions or the size of an rDMA request, must be considered when managing these regions. Although this approach involves a bit more application intervention, it presents a better opportunity for maximizing rDMA performance

Registered local region functions

The use of a registered local memory region exposes the limitations of the underlying architecture in an effort to maximize performance. Registering a local memory region permits DaCS to minimize both error checking as well as setup overhead on each use of that region.

Local memory region usage scenarios

The table below shows how to initialize and share a memory region.

DE A	DE B

A memory region is created to be used for DMA:
`dacs_mem_create(memA_addr,`
`        memA_size,`
`        remote_access,`
`        local_access,&memA)`
 	
This creates a memory region of *memA_size* bytes starting at local address *memA_addr*. The memory region will have *remote_access* permissions for remote consumers and *local_access* permissions locally. A handle to the created memory region *memA* is returned.	

Share the memory region for remote usage:
`dacs_mem_share(de_B,`
`        pid_B,`
`        memA)`
 	
The memory region must be explicitly shared with each DE/PID that will access the region remotely. The request to share a memory region will block until the corresponding accept is performed.
	Each shared memory region must be accepted:
	`dacs_mem_accept(de_A,`
	`        pid_A,`
	`        &memA)`
	
	Each remotely created memory region to be accessed must be accepted from the sharing DE/PID. A handle to the shared memory region *memA* is returned.

	The memory region can be queried for its attributes:
	`dacs_mem_query(memA,`
	`        DACS_REMOTE_MEM_SIZE,`
	`        &memQ)`
	
	The remotely created and shared memory region can be queried to obtain its region attributes. Attributes are individually queried. The above query requests the size of the remotely created region.

	A memory region is created to be used for DMA:
	`dacs_mem_create(memB_addr,`
	`        memA_size,`
	`        remote_access,`
	`        local_access,`
	`        &memB)`
	
	This creates a local memory region to be used in conjunction with the accepted memory region *memA*. This memory region is created based on the size of the remotely created region *memA_size* and starts at local address *memB_addr*. The memory region will have *remote_access* permissions for remote consumers and *local_access* permissions locally.

DE A	DE B

	The memory region is registered for local DMA use:
	`dacs_mem_register(de_A,`
	`        pid_A,`
	`        memB)`
	
	All unshared memory regions must be registered for access by the remote DE/PID, if they are to be used for DMA.

Getting data from a remote memory region: This section shows how to get from a remote memory region once it has been created:

DE A	DE B

	Get data from the remote memory region to the local memory region:
	`dacs_mem_get(memB,`
	`              memB_offset,`
	`              memA,`
	`              memA_offset,`
	`              size,`
	`              widB,`
	`              order,`
	`              swap )`
	
	In this case we are initiating a MEM_GET of *size* bytes from remote memory region *memA* at offset *memA_offset*. This data is stored into local memory region *memB* at offset *memB_offset*. The memory transfer is tracked through the wait identifier *widB* and is ordered according to the specified ordering type *order*. Intermediate byte swapping is performed based on the swap type *swap*.

DE A	DE B
	Wait on the MEM_GET to complete: `dacs_wait(widB)` In order to verify the completion of the DMA operation, the wait identifier must be successfully waited on or tested. Multiple messages or DMAs may be issued under a single wait identifier which only requires a single call to wait or test. The call to wait or test will not complete until all DMAs or messages associated with the wait identifier have completed. Wait is a blocking status check of all outstanding transactions for wait identifier *widB*. It will not return until all transfers associated with *widB* have either completed or encountered an error. In the case of a MEM_GET, the successful completion of waiting or testing guarantees that *size* data bytes have arrived in the associated local memory region *memB* at the specified offset *memB_offset*.

Putting data into a remote memory region: This section shows how to put data into a remote memory region:

DE A	DE B
	Put data from the local memory region to the remote memory region: `dacs_mem_put(memA, memA_offset, memB, memB_offset, size, widB, order, swap )` In this case we are initiating a PUT of *size* bytes from local memory region *memB* at offset *memB_offset*. This data is stored into remote memory region *memA* at offset *memA_offset*. The memory transfer is tracked through the wait identifier *widB* and is ordered according to the specified ordering type *order*. Intermediate byte swapping is performed based on the swap type.

DE A	DE B
	Test whether the MEM_PUT has completed: `dacs_test(`*`widB`*`)` In order to verify the completion of the DMA operation, the wait identifier must be successfully waited on or tested. Multiple messages or DMAs may be issued under a single wait identifier which only requires a single call to wait or test. The call to wait or test will not complete until all DMAs or messages associated with the wait identifier have completed. Test is a non-blocking status check of all outstanding transactions for wait identifier *widB*. If any transfers, associated with *widB*, have not completed, the call will return with a busy status. In the case of a PUT, the successful completion of waiting or testing only means that the local memory buffer is available for re-use. It does not guarantee that the data has successfully arrived at the remote memory region.

Deregistering memory regions: This section shows how to deregister a locally-created memory region.

DE A	DE B
	Deregister previously registered memory regions: `dacs_mem_deregister(`*`de_A`*`,`*`pid_A`*`,`*`memB`*`)` For each DE/PID a locally created memory region was registered a corresponding deregistration must be performed. Prior to destroying a locally created memory region that had been registered, it must first be deregistered. For every DE and PID a local memory region was registered for, a corresponding deregistration must be performed. This detaches the locally created memory region from being associated with the specified DE/PID.
	The creator of a memory region must destroy it when the region is no longer needed: `dacs_mem_destroy(`*`&memB`*`)` In this case the DE is destroying the memory region it created for local use. Normally this call would block until the memory object had been released by all DEs it was shared with, but in this case it was never shared.

DE A	**DE B**
The creator of a memory region must destroy it when the region is no longer needed: `dacs_mem_destroy(&memA)` In this case the DE is destroying the memory region it created for remote use. Since the memory region had been shared for remote use, the call must block until the memory region has been released by all DEs it was shared with.	Every remote memory region that was accepted must be released when no longer needed: `dacs_mem_release(&memA)` Each consumer of the remote memory must release the memory object once finished, so the creator knows when it is safe to destroy it.

dacs_mem_create

NAME

dacs_mem_create - Designate a region in the memory space of the current process for use by DMA services.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_create (void **addr*, **uint64_t** *size*, **DACS_MEM_ACCESS_MODE_T** *rmt_access_mode*, **DACS_MEM_ACCESS_MODE_T** *lcl_access_mode*, **dacs_mem_t** **mem***)**

Fortran syntax

dacsf_mem_create (dacs_pvoid_t *addr*, **dacs_int64_t** *size*, **DACS_MEM_ACCESS_MODE_T** *rmt_access_mode*, **DACS_MEM_ACCESS_MODE_T** *lcl_access_mode*, **dacs_mem_t** *mem*, **DACS_ERR_T** *rc* **)**

Call parameters	
`addr`	**C**: a pointer to the base address of the memory region to be shared. **Fortran**: handle to local address to create a memory region over. Use `dacsf_makevoid` to create the handle.
`size`	The size of the memory region in bytes.
`rmt_access_mode`	Permission granted for remote access of the memory region.
`lcl_access_mode`	Permission granted for local access of the memory region.

Return parameters	
`mem`	**C**: a pointer to a memory handle to be filled in. **Fortran**: a handle to the memory region.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mem_create` service creates and returns a handle associated with the given memory region. The returned handle can be used with the `dacs_mem_share()` service to use it as a remote memory region, as well as the `dacs_mem_register()` service to use it as a local memory region.

The following access modes are available:

- `DACS_MEM_READ_ONLY`: the memory region is only readable
- `DACS_MEM_WRITE_ONLY`: the memory region is only writeable
- `DACS_MEM_READ_WRITE`: the memory region is both readable and writeable
- `DACS_MEM_NONE`: the memory region is neither readable nor writeable

Cell/B.E.: This call is not supported on the SPU, and calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The `dacs_mem_create` service returns an error indicator defined as:

- `DACS_SUCCESS`: the memory region was successfully created.
- `DACS_ERR_INVALID_ADDR`: the address or memory handle address was invalid.
- `DACS_ERR_INVALID_SIZE`: the memory regions must be larger than 0.
- `DACS_ERR_INVALID_ATTR`: an access mode was invalid.
- `DACS_ERR_NO_RESOURCE`: there are no resources available to complete this request.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3),dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3), dacsf_makevoid(3)

dacs_mem_share

NAME

dacs_mem_share - Pass a memory handle from the current process to a remote process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_share (de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mem_t** *mem***)**

Fortran syntax

dacsf_mem_share (dacs_de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mem_t mem** *mem*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_de`	The target DE for the share.
`dst_pid`	The target process for the share.
`mem`	The handle of the memory to be shared.

Return parameter	
`rc`	**Fortran only**: See Return value.

DESCRIPTION

The `dacs_mem_share` service shares the specified memory handle from the current process to the remote process specified by `dst_de` and `dst_pid`. This service then blocks, waiting for a matching call to the `dacs_mem_accept()` service on the remote side.

Note: when communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: memory sharing is not supported from the PPU to the SPU. Attempting this will result in a `DACS_ERR_NOT_SUPPORTED_YET` error.

RETURN VALUE

The `dacs_mem_share` service returns an error indicator defined as:
- `DACS_SUCCESS`: the memory region was successfully shared.
- `DACS_ERR_INVALID_DE`: the DE ID was invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the program id is not valid.
- `DACS_ERR_INVALID_HANDLE`: the memory handle was invalid or is only initialized for local access.
- `DACS_ERR_NO_PERM`: handle has no remote permissions.
- `DACS_ERR_NOT_OWNER`: the handle cannot be shared because this DE/PID did not create it.
- `DACS_ERR_NO_HANDLE`: there are no local resources available for this memory handle.

- `DACS_ERR_INVALID_TARGET`: the handle is already shared with the target or has attempted to share with itself.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized yet.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.

SEE ALSO

dacs_mem_create(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_accept

NAME

dacs_mem_accept - Accept a memory handle from a remote process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_accept (de_id_t *src_de*, **dacs_process_id_t** *src_pid*, **dacs_mem_t** **mem***)**

Fortran syntax

dacsf_mem_accept (dacs_de_id_t *src_de*, **dacs_process_id_t** *src_pid*, **dacs_mem_t** *mem*, **DACS_ERR_T** *rc***)**

Call parameters	
`src_de`	The source DE which is sharing the memory handle.
`src_pid`	The source process which is sharing the memory handle.

Return parameters	
`mem`	**C**: a pointer to the accepted memory handle. **Fortran**: the accepted memory handle.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mem_accept` service blocks the caller until it receives a memory handle from an associated `dacs_mem_share()` call. The memory handle is returned on success.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: the PPU host cannot accept memory from an SPU because an SPU cannot create memory.

RETURN VALUE

The `dacs_mem_accept` service returns an error indicator defined as:

- `DACS_SUCCESS`: The memory region was successfully accepted.
- `DACS_ERR_INVALID_DE`: The DE id was invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the program id is not valid.
- `DACS_ERR_INVALID_TARGET`: attempt to accept with self.
- `DACS_ERR_INVALID_ADDR`: the address of the output handle is invalid.
- `DACS_ERR_NO_RESOURCE`: there are no local resources available for this memory handle.
- `DACS_ERR_NOT_INITIALIZED`: DaCs has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_release(3),dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_release

NAME

dacs_mem_release - Release a previously accepted memory handle.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_release (dacs_mem_t **mem***)**

Fortran syntax

dacsf_mem_release (dacs_mem_t *mem***, DACS_ERR_T** *rc***)**

Call parameter	
`mem`	**C**: a pointer to the memory handle. **Fortran**: a remote memory handle.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mem_release` service releases a previously accepted memory object and invalidates the handle. When all accepters have released the object it may be destroyed by its owner. This service does not block.

Cell/B.E.: this call is not supported on the PPU. Calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The `dacs_mem_release` service returns an error indicator defined as:
- `DACS_SUCCESS`: the memory region was successfully released.
- `DACS_ERR_INVALID_HANDLE`: the handle is not for a valid remote memory region.
- `DACS_ERR_INVALID_ADDR`: the address of the output handle is invalid.
- `DACS_ERR_OWNER`: attempt to release a created memory region.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_register

NAME

dacs_mem_register - Register this memory region to be used as a local memory handle on DMA operations.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_register (de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mem_t** *mem***)**

Fortran syntax

dacsf_mem_register (dacs_de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mem_t** *mem*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_de`	The target DE for the register.
`dst_pid`	The target process for the register.
`mem`	The handle of the memory to be registered.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mem_register` service registers the specified memory handle from the current process to the remote process specified by `dst_de` and `dst_pid`.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: this call is not supported on the PPU. Calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The `dacs_mem_register` service returns an error indicator defined as:

- `DACS_SUCCESS`: the memory region was successfully accepted.
- `DACS_ERR_INVALID_DE`: the DE id was invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the program id is not valid.
- `DACS_ERR_INVALID_HANDLE`: the memory handle was invalid or the region was only created was only created for remote access.
- `DACS_ERR_NOT_OWNER`: the handle cannot be registered. The current DE/PID is not the owner.
- `DACS_ERR_NO_PERM`: the memory region was created with no local permissions.
- `DACS_ERR_NO_RESOURCE`: there are no local resources available for this memory handle.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_deregister

NAME

dacs_mem_deregister - Deregister memory access for a local region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_deregister (de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mem_t** *mem***)**

Fortran syntax

dacsf_mem_deregister (dacs_de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_mem_t** *mem*, **DACS_ERR_T** *rc***)**

Call parameters

`dst_de`	The target DE for the deregister.
`dst_pid`	The target process for the deregister.
`mem`	**C**: a pointer to the memory handle. **Fortran**: the memory handle.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

Remove the capability to use a created and registered memory region as a local memory region in DMA operations with the specified DE/PID. The memory handle must have been previously registered for this method to successfully complete.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: this call is not supported on the PPU. The PPU host cannot accept memory from an SPU because an SPU cannot create memory. Calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The `dacs_mem_deregister` service returns an error indicator defined as:

- `DACS_SUCCESS`: The memory region was successfully deregistered.
- `DACS_ERR_INVALID_DE`: The DE id was invalid or not reserved.
- `DACS_ERR_INVALID_HANDLE`: memory region was never registered.
- `DACS_ERR_INVALID_PID`: the program id is not valid.
- `DACS_ERR_INVALID_TARGET`: the memory handle was not registered with this DE/PID.
- `DACS_ERR_NOT_OWNER`: the handle cannot be deregistered. The current DE/PID is not the owner.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

- DACS_ERR_NOT_SUPPORTED_YET: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_destroy

NAME

dacs_mem_destroy - Invalidate access to the specified memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_destroy (dacs_mem_t ***mem*****)**

Fortran syntax

dacsf_mem_destroy (dacs_mem_t *mem***, DACS_ERR_T** *rc***)**

Call parameter	
`mem`	**C**: pointer to a memory handle. **Fortran**: the memory handle.

Return parameter	
`rc`	**Fortran only**: see Return Value.

DESCRIPTION

The `dacs_mem_destroy` service invalidates the sharing of a specific memory region that has been created by `dacs_mem_create()`. Only the creator of the memory region may destroy it. This service blocks until all users of the memory region have released it with either `dacs_mem_release()` or `dacs_mem_deregister()`. The handle will be invalidated on successful completion of this method.

RETURN VALUE

The `dacs_mem_destroy` service returns an error indicator defined as:
- `DACS_SUCCESS`: the memory region was successfully destroyed.
- `DACS_ERR_INVALID_HANDLE`: the handle is not for a valid local memory region.
- `DACS_ERR_INVALID_ADDR`: the address of the output handle is invalid.
- `DACS_ERR_NOT_OWNER`: attempt to destroy an accepted memory region.
- `DACS_ERR_RESOURCE_BUSY`: the memory region is still registered for use.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_get

NAME

dacs_mem_get - Get data from remote memory to local memory.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_get (dacs_mem_t *dst_local_mem*, **uint64_t** *dst_local_mem_offset*, **dacs_mem_t** *src_remote_mem*, **uint64_t** *src_remote_mem_offset*, **uint64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_mem_get (dacs_mem_t *dst_local_mem*, **dacs_int64_t** *dst_local_mem_offset*, **dacs_mem_t** *src_remote_mem*, **dacs_int64_t** *src_remote_mem_offset*, **dacs_int64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_local_mem`	The memory handle of the destination buffer.
`dst_local_mem_offset`	The offset into local buffer where the get is to start.
`src_remote_mem`	The memory handle of the source buffer.
`src_remote_mem_offset`	The offset into the remote buffer where the get is to start.
`size`	The size of transfer in bytes.
`wid`	A communications wait identifier.
`order_attr`	An ordering attribute.
`swap`	The little-endian or big-endian byte-swapping flag.

Return parameter	
`rc`	**Fortran only**: see Return value

DESCRIPTION

This is a non-blocking DMA operation that initiates a data copy from a remote memory region into a local memory region. The data source is specified as an offset, `src_remote_mem_offset`, into the remote memory region `src_remote_mem`. The remote memory region must have been previously accepted using `dacs_mem_accept()`. The data destination is specified as an offset, `dst_local_mem_offset`, into the local memory region `dst_local_mem`. The local memory region must have been previously registered using `dacs_mem_register()`.

This operation is associated with a specified wait identifier, `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same `wid`. Successfully waiting on the completion of this operation guarantees that all data is available locally.

Possible values of `order_attr` are:
- `DACS_ORDER_ATTR_FENCE`: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.

- `DACS_ORDER_ATTR_BARRIER`: execution of this command and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_NONE`: no ordering is enforced.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or share remote memory.

RETURN VALUE

The `dacs_mem_get` service returns an error indicator defined as:

- `DACS_SUCCESS`: the get was started successfully.
- `DACS_ERR_INVALID_HANDLE`: the remote or local handle was invalid or the local memory region may have not been registered.
- `DACS_ERR_BUF_OVERFLOW`: the specified size at the given offset exceeded the bounds of the destination memory region.
- `DACS_ERR_INVALID_SIZE`: the specified size at the given offset exceeded the bounds of the source memory region.
- `DACS_ERR_INVALID_WID`: the `wid` is not reserved.
- `DACS_ERR_INVALID_ATTR`: the order or byteswap attribute is invalid.
- `DACS_ERR_NO_RESOURCE` : there are no local resources available.
- `DACS_ERR_NO_PERM`: local memory regions have no write access or remote memory region has no read access.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_put

NAME

dacs_mem_put - Initiate a data transfer from local memory to remote memory.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_put (dacs_mem_t *dst_remote_mem*, **uint64_t** *dst_remote_mem_offset*, **dacs_mem_t** *src_local_mem*, **uint64_t** *src_local_mem_offset*, **uint64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_mem_put (dacs_mem_t *dst_remote_mem*, **dacs_int64_t** *dst_remote_mem_offset*, **dacs_mem_t** *src_local_mem*, **dacs_int64_t** *src_local_mem_offset*, **dacs_int64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_remote_mem`	The memory handle of the destination buffer.
`dst_remote_mem_offset`	The offset into the remote buffer where the put is to be performed.
`src_local_mem`	The memory handle of the source buffer.
`src_local_mem_offset`	The offset into the local buffer where the put is to be performed.
`size`	The amount of data to transfer in bytes.
`wid`	The communications wait identifer.
`order_attr`	An ordering attribute.
`swap`	The little-endian or big-endian byte-swapping flag.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

This is a non-blocking DMA operation that initiates a data copy from a local memory region into a remote memory region. The data source is specified as an offset, `src_local_mem_offset`, into the local memory region `src_local_mem`. The local memory region must have been previously registered using `dacs_mem_register()`. The data destination is specified as an offset, `dst_remote_mem_offset`, into the remote memory region `dst_remote_mem`. The remote memory region must have been previously accepted using `dacs_mem_accept()`.

This operation is associated with a specified wait identifier, `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same `wid`. Successfully waiting on the completion of this operation only guarantees that the data has been sent and that the source buffer can be safely reused.

Possible values of `order_attr` are:

- `DACS_ORDER_ATTR_FENCE`: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_BARRIER`: execution of this command and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_NONE`: no ordering is enforced.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or share remote memory.

RETURN VALUE

The `dacs_mem_put` service returns an error indicator defined as:

- `DACS_SUCCESS`: the put was started successfully.
- `DACS_ERR_INVALID_HANDLE`: the remote or local handle was invalid or the local memory region may have not been registered.
- `DACS_ERR_BUF_OVERFLOW`: the specified size at the given offset exceeded the bounds of the destination memory region.
- `DACS_ERR_INVALID_SIZE`: the specified size at the given offset exceeded the bounds of the source memory region.
- `DACS_ERR_INVALID_WID`: the `wid` is not reserved.
- `DACS_ERR_INVALID_ATTR`: the order or byteswap attribute is invalid.
- `DACS_ERR_NO_RESOURCE` : there are no local resources available.
- `DACS_ERR_NO_PERM`: the local memory region has no read access or remote memory region has no write access
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_get_list

NAME

dacs_mem_get_list - Initiate a scatter or gather data transfer from a remote memory region into a local memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_get_list, (dacs_mem_t *dst_lcl_handle*, **dacs_dma_list_t** **dst_list*, **uint32_t** *dst_count*, **dacs_mem_t** *src_rmt_handle*, **dacs_dma_list_t** **src_list*, **uint32_t** *src_count*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_mem_get_list (dacs_mem_t *dst_lcl_handle*, **dacs_dma_list_t** *dst_list* **,** **dacs_int32_t** *dst_count*, **dacs_mem_t** *src_rmt_handle*, **dacs_dma_list_t** *src_list*, **dacs_int32_t** *src_count*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_lcl_handle`	Locally created memory region.
`dst_list`	List of offset/sizes within the local memory region.
`dst_count`	The number of destination DMA list descriptors.
`src_rmt_handle`	Accepted remote memory region handle.
`src_list`	List of offset/sizes within the remote memory region.
`src_count`	The number of source DMA list descriptors.
`wid`	Reserved wait id.
`order_attr`	An ordering attribute.
`swap`	The little-endian or big-endian byte-swapping flag.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

This is a non-blocking DMA operation that initiates a scatter or gather data copy from a remote memory region into a local memory region. This method can either scatter contiguous remote data blocks across numerous, possibly disjunct, local memory locations or gather numerous, possibly disjunct, remote data blocks into a contiguous local data area. The data source is specified through the list, `src_dma_list`, which contains size and offset pairings that are applied relative to the specified remote memory region `src_remote_mem`. This remote memory region must have been previously accepted using dacs_mem_accept(). The data destination is specified through the list, `dst_dma_list`, which contains size and offset pairings that are applied relative to the specified local memory region `dst_local_mem`. This local memory region must have been previously registered using `dacs_mem_register()`.

This operation is associated with a specified wait identifier, `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same `wid`. Successfully waiting on the completion of this operation guarantees that all data is available locally.

Possible values of `order_attr` are:
- `DACS_ORDER_ATTR_FENCE`: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_BARRIER`: execution of this command and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_NONE`: no ordering is enforced.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or sharing remote memory.

RETURN VALUE

The `dacs_mem_get_list` service returns an error indicator defined as:
- `DACS_SUCCESS`: the get was started successfully.
- `DACS_ERR_INVALID_HANDLE`: the remote or local handle was invalid or the local memory region may have not been registered.
- `DACS_ERR_BUF_OVERFLOW`: this can be for one of the following reasons:
  1. Source list is too large for destination list.
  2. Destination list is too large for local (destination) region.
  3. Destination list is out-of-bounds of local (destination) region.
- `DACS_ERR_INVALID_SIZE`: this can be for one of the following reasons:
  1. Destination list is too large for source list.
  2. Source list too large for remote (source) region.
  3. Source list is out-of-bounds of remote (source) region.
  4. The destination or the source list size is zero.
  5. Either destination or the source list size is not equal to 1.
- `DACS_ERR_INVALID_WID`: the wid is not reserved.
- `DACS_ERR_INVALID_ATTR`: the order or byteswap attribute is invalid.
- `DACS_ERR_NO_RESOURCE`: there are no local resources available.
- `DACS_ERR_INVALID_ADDR`: the remote or local DMA list address is invalid.
- `DACS_ERR_NO_PERM`: Local memory regions has no write access or remote memory region has no read access.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the swap identifier is invalid or the result of calling the API on an unsupported platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_put_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_put_list

NAME

dacs_mem_put_list - Initiate a scatter or gather data transfer from a local memory region into a remote memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_put_list (dacs_mem_t *dst_rmt_handle*, **dacs_dma_list_t** **dst_list*, **uint32_t** *dst_count*, **dacs_mem_t** *src_lcl_handle*, **dacs_dma_list_t** **src_list*, **uint32_t** *src_count*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_mem_put_list (dacs_mem_t *dst_rmt_handle*, **dacs_dma_list_t** *dst_list*, **dacs_int32_t** *dst_count*, **dacs_mem_t** *src_lcl_handle*, **dacs_dma_list_t** *src_list*, **dacs_int32_t** *src_count*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_rmt_handle`	Accepted remote memory region handle.
`dst_list`	List of offset/sizes within the remote memory region.
`dst_count`	The number of destination DMA list descriptors.
`src_lcl_handle`	Locally created memory region.
`src_list`	List of offset/sizes within the local memory region.
`src_count`	The number of source DMA list descriptors.
`wid`	Reserved wait id.
`order_attr`	An ordering attribute.
`swap`	The little-endian or big-endian byte-swapping flag.

Return parameter	
`rc`	**Fortran only**: See Return values

DESCRIPTION

This is a non-blocking DMA operation that initiates a scatter or gather data copy from a local memory region into a remote memory region. This method can either scatter contiguous local data blocks across numerous, possibly disjunct, remote memory locations or gather numerous, possibly disjunct, local data blocks into a contiguous remote data area. The data source is specified through the list, `src_dma_list`, which contains size and offset pairings that are applied relative to the specified local memory region `src_local_mem`. This local memory region must have been previously registered using dacs_mem_register(). The data destination is specified through the list, `dst_dma_list`, which contains size and offset pairings that are applied relative to the specified remote memory region `dst_remote_mem`. This remote memory region must have been previously accepted using `dacs_mem_accept()`.

This operation is associated with a specified wait identifier, `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same `wid`. Successfully waiting on the completion of this operation only guarantees that the data has been sent and that the source buffer can be safely reused.

Possible values of `order_attr` are:
- `DACS_ORDER_ATTR_FENCE`: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_BARRIER`: execution of this command and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_NONE`: no ordering is enforced.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or share remote memory.

RETURN VALUE

The `dacs_mem_put_list` service returns an error indicator defined as:
- `DACS_SUCCESS`: the get was started successfully.
- `DACS_ERR_INVALID_HANDLE`: the remote or local handle was invalid or the local memory region may have not been registered.
- `DACS_ERR_BUF_OVERFLOW`: this can be for one of the following reasons:
  1. Source list is too large for destination list.
  2. Destination list is too large for local (destination) region.
  3. Destination list is out-of-bounds of local (destination) region.
- `DACS_ERR_INVALID_SIZE`: this can be for one of the following reasons:
  1. Destination list is too large for source list.
  2. Source list too large for remote (source) region.
  3. Source list is out-of-bounds of remote (source) region.
  4. The destination or the source list size is zero.
  5. neither the destination nor the source list size is equal to 1.
- `DACS_ERR_INVALID_WID`: the wid is not reserved.
- `DACS_ERR_INVALID_ATTR`: the order or byteswap attribute is invalid.
- `DACS_ERR_NO_RESOURCE`: there are no local resources available.
- `DACS_ERR_INVALID_ADDR`: the remote or local DMA list address is invalid.
- `DACS_ERR_NO_PERM`: Local memory regions has no write access or remote memory region has no read access.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the swap identifier is invalid or the result of calling the API on an unsupported platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_limits_query(3), dacs_mem_query(3)

dacs_mem_limits_query

NAME

dacs_mem_limits_query - Query the limits on memory regions for communications with a specific DE/PID.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_limits_query (DACS_MEM_LIMITS_T *attr*, **de_id_t** *tgt_de*, **dacs_process_id_t** *tgt_pid*, **uint64_t *** *value***)**

Fortran syntax

dacsf_mem_limits_query (DACS_MEM_LIMITS_T *attr*, **dacs_de_id_t** *tgt_de*, **dacs_process_id_t** *tgt_pid*, **dacs_int64_t** *limit*, **DACS_ERR_T** *rc***)**

Call parameters	
`attr`	The specific limit value to query.
`tgt_de`	The target DE for the query.
`tgt_pid`	The target process for the query.

Return parameters	
`value`	**C only:** a pointer to the location where the attribute value is to be returned. The value returned in this location depends on the `attr` parameter passed. See Description for further details. A value of `UINT64_MAX` indicates there is no limit for this attribute.
`limit`	**Fortran only**: the value returned for limit depends on the attr parameter passed. A value of -1 indicates there is no limit for the attribute.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

Query the limits on memory regions for communications with a specific DE/PID. Underlying devices may have restrictions on the size of memory regions. These query attributes are used to help the programmer manage their DMA resources. The numbers returned are guaranteed values at the time of the call. It may be possible to successfully ask for more or larger memory regions than what is returned here. These values may change after each `dacs_mem_share` or `dacs_mem_register` call made.

Valid query limit attributes are:

- `DACS_MEM_REGION_MAX_NUM`: the maximum number of memory regions available for communication with the target DE/PID.
- `DACS_MEM_REGION_MAX_SIZE`: the size, in bytes, of the largest single memory region that can be created and shared/registered with the target DE/PID. This size may change based on the current number of allocated memory regions and the amount of memory each allocated memory region covers. As the aggregate size of all allocated memory regions increases, this size may lower as it approaches a maximum total aggregate memory size for the underlying device.

- `DACS_MEM_REGION_AVAIL`: the total number of currently unused memory regions for communications with the target DE/PID.

Note: when communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

RETURN VALUE

The `dacs_mem_limits_query` service returns an error indicator defined as:

- `DACS_SUCCESS`: query was successful.
- `DACS_ERR_NOT_INITIALIZED`: DaCS not initialized.
- `DACS_ERR_INVALID_TARGET`: DE/PID cannot be self.
- `DACS_ERR_INVALID_ATTR`: attribute was invalid.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_ADDR`: the address passed in for the return value was invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3),dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_query(3)

dacs_mem_query

NAME

dacs_mem_query - Query the attributes of a memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mem_query (dacs_mem_t *mem*, **DACS_MEM_ATTR_T** *attr*, **uint64_t** **value***)**

Fortran syntax

dacsf_mem_query_lcl_perm (dacs_mem_t *mem*, **DACS_MEM_ACCESS_MODE_T** *lcl_mode*, **DACS_ERR_T** *rc***)**

dacsf_mem_query_rmt_perm (dacs_mem_t *mem*, **DACS_MEM_ACCESS_MODE_T** *rmt_mode*, **DACS_ERR_T** *rc***)**

dacsf_mem_query_rmt_size (dacs_mem_t *mem*, **dacs_int64_t** *mem_size*, **DACS_ERR_T** *rc***)**

dacsf_mem_query_addr (dacs_mem_t *mem*, **dacs_pvoid_t** *addr*, **DACS_ERR_T***rc***)**

Call parameters	
`mem`	The handle of the memory area to query.
`attr`	The attribute to be queried. See Description for more information.

Return parameters	
`value`	**C only**: a pointer to the location where the attribute value is to be returned. The value returned in this location depends on the `attr`. See Description for more information on the parameters that may be passed. For `DACS_RMT_MEM_PERM` and `DACS_LCL_MEM_PERM`. See Description for more information about the access mode permissions.
`lcl_mode`	**Fortran only**: the local access mode for the requested handle. See Description for more information about the access mode permissions.
`rmt_mode`	**Fortran only**: the remote access mode for the requested handle. See Description for more information about the access mode permissions.
`mem_size`	**Fortran only**: the size for the requested handle.
`addr`	**Fortran only**: the address for the requested handle.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mem_query` service queries the attributes of the specified memory region. The memory region being queried must have been created or accepted by the caller.

The value returned depends on the `attr` parameter passed as shown below:

- `DACS_MEM_SIZE`: size of the memory region.

- `DACS_MEM_ADDR`: address of the memory region.
- `DACS_RMT_MEM_PERM`: remote permissions on the memory region.
- `DACS_LCL_MEM_PERM`: local permissions on the memory region.

The following access mode permissions are available:
- `DACS_MEM_READ_ONLY`: the memory region is only readable
- `DACS_MEM_WRITE_ONLY`: the memory region is only writeable
- `DACS_MEM_READ_WRITE`: the memory region is both readable and writeable
- `DACS_MEM_NONE`: the memory region is neither readable nor writeable

RETURN VALUE

The `dacs_mem_query` service returns an error indicator defined as:
- `DACS_SUCCESS`: query was successful.
- `DACS_ERR_INVALID_HANDLE`: memory handle was invalid.
- `DACS_ERR_INVALID_ATTR`: attribute was invalid.
- `DACS_ERR_INVALID_ADDR`: value address was invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mem_create(3), dacs_mem_share(3), dacs_mem_accept(3), dacs_mem_release(3), dacs_mem_register(3), dacs_mem_deregister(3), dacs_mem_destroy(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_get_list(3), dacs_mem_put_list(3), dacs_mem_limits_query(3)

Unregistered local region functions

The unregistered local region functions provide the means for sharing memory regions with remote processes. A memory region is made available to remote consumers using a *share/accept* model whereby the owner of the memory creates and shares a remote memory handle which is then accepted and used by remote processes.

The unregistered local region functions, because they always work, make it easier to get an application working, but by hiding the limitation they may hide situations where an application change is required to achieve optimal performance.

Hybrid only: To help identify these situations, DaCS for Hybrid will create a log entry when it hides a limitation. These logs are not automatically created, but can be turned on as described in Appendix F, "DaCS for Hybrid debugging," on page 197.

Note: With the exception of `dacs_remote_mem_query()`, the remote memory handle can only be used on DaCS memory transfer services by the remote processes, and only after they have accepted a share. The owner of the shared memory cannot use these services.

Note: When DaCS for Hybrid is being used with DaCS for Cell/B.E, remote memory that is created on the PPU (using `dacs_remote_mem_create()`) can be shared with both the x86_64 (HE) and with the SPUs (AEs). If this is done then either can use the put or get services to get from the memory shared by the PPU.

Remote memory usage scenarios

This table shows how to initialize and share a remote memory region.

DE A	DE B

A memory region is created to be used for DMA:
`dacs_remote_mem_create(`*`memA_addr`*`,`
        `memA_size``,`
        `mem_access``,`
        `&memA``)`
 	
This creates a memory region of *`memA_size`* bytes starting at local address *`memA_addr`*. The memory region will have remote_access permissions for remote consumers and local_access permissions locally. A handle to the created memory region *`memA`* is returned.	

Share the memory region for remote usage:
`dacs_remote_mem_share(`*`de_B`*`,`
        `pid_B``,`
        `memA``)`
 	
The memory region must be explicitly shared with each DE/PID that will access the region remotely. The request to share a memory region will block until the corresponding accept is performed.
	Each shared memory region must be accepted:
	`dacs_remote_mem_accept(`*`de_A`*`,`
	        `pid_A``,`
	        `&memA``)`
	
	Each remotely created memory region to be accessed must be accepted from the sharing DE/PID. A handle to the shared memory region *`memA`* will be returned.

DE A	DE B

	The memory region can be queried for its attributes:
	`dacs_remote_mem_query(`*`memA`*`,`
	         `DACS_REMOTE_MEM_SIZE``,`
	         `&memQ``)`
	
	The remotely created and shared memory region can be queried to obtain its region attributes. Attributes are individually queried. The above query requests the size of the remotely created region.

Get from a remote memory region: Getting data from a remote memory region.

DE A	DE B

	Get data from the remote memory region:
	`dacs_get(`*`memB_addr`*`,`
	    `memA``,`
	    `memA_offset``,`
	    `size``,`
	    `widB``,`
	    `order``,`
	    `swap``)`
	
	In this case we are initiating a GET of *`size`* bytes from remote memory region *`memA`* at offset *`memA_offset`*. This data is stored directly into local memory starting at *`memB_addr`*. The memory transfer is tracked through the wait identifier *`widB`* and is ordered according to the specified ordering type *`order`*. Intermediate byte swapping is performed based on the swap type *`swap`*.

	Wait on the MEM_GET to complete: `dacs_wait(`*`widB`*`)` In order to verify the completion of the DMA operation, the wait identifier must be successfully waited on or tested. Multiple messages or DMAs may be issued under a single wait identifier which only requires a single call to wait or test. The call to wait or test will not complete until all DMAs or messages associated with the wait identifier have completed. Wait is a blocking status check of all outstanding transactions for wait identifier 'widB'. It will not return until all transfers, associated with *`widB`*, have either completed or encountered an error. In the case of a GET, the successful completion of waiting or testing guarantees that *`size`* data bytes have arrived in the associated local memory buffer pointed to *`memB_addr`*.

Put to a remote memory region: Putting data into a remote memory region.

DE A	DE B

	Put data into the remote memory region:
	`dacs_put(memA,`
	`    memA_offset,`
	`    memB_addr,`
	`    size,`
	`    widB,`
	`    order,`
	`    swap)`
	
	In this case we are initiating a PUT of `size` bytes from local memory region `memA` at offset `memA_offset`. This data is stored directly into local memory starting at `memB_addr`. The memory transfer is tracked through the wait identifier `widB` and is ordered according to the specified ordering type `order`. Intermediate byte swapping is performed based on the swap type `swap`.

	Test whether the PUT has completed: `dacs_test(widB)` In order to verify the completion of the DMA operation, the wait identifier must be successfully waited on or tested. Multiple messages or DMAs may be issued under a single wait identifier which only requires a single call to wait or test. The call to wait or test will not complete until all DMAs or messages associated with the wait identifier have completed. Test is a non-blocking status check of all outstanding transactions for wait identifier `widB`. If any transfers, associated with `widB`, have not completed, the call will return with a busy status. In the case of a PUT, the successful completion of waiting or testing only means that the local memory buffer is available for re-use. It does not guarantee that the data has successfully arrived at the remote memory region.

Destroying remote memory: Releasing and destroying a remote memory region.

DE A	DE B
The creator of a memory region must destroy it when the region is no longer needed: `dacs_remote_mem_destroy(&memA)` In this case the DE is destroying the memory region it created for remote use. Since the memory region had been shared for remote use, the call must block until the memory region has been released by all DEs it was shared with.	Every remote memory region that was accepted must be released when no longer needed: `dacs_remote_mem_release(&memA)` Each consumer of the remote memory must release the memory object once finished, so the creator knows when it is safe to destroy it.

Using the Hybrid library

The DaCS Hybrid version of the library integrates with the DaCS on Cell /B.E. implementation on a CBEA (PPU) system. Integrated API calls can be interpreted as Hybrid or Cell/B.E. library calls, depending on the set of parameters that are passed in. This can lead to confusing return codes in some situations, especially when debugging an application using the debug libraries. The memory init and cleanup APIs suffer from this problem in particular; the Hybrid version are only called when DACS_DE_PARENT is used as the DE, any other value calls the Cell/B.E. version which returns `DACS_ERR_NOT_SUPPORTED_YET`, instead of `DACS_ERR_INVALID_DE`.

The following functions exhibit this behavior:

- `dacs_remote_mem_accept`: The PPU host cannot accept memory from an SPU because an SPU cannot create memory.
- `dacs_remote_mem_release`: The PPU host cannot release memory from an SPU because an SPU cannot create memory.

dacs_remote_mem_create

NAME

dacs_remote_mem_create - Designate a region in the memory space of the current process for access by remote processes by DMA services.

SYNOPSIS

C syntax

DACS_ERR_T dacs_remote_mem_create (void **addr*, **uint64_t** *size*, **DACS_MEMORY_ACCESS_MODE_T** *access_mode*, **dacs_remote_mem_t** **mem***)**

Fortran syntax

dacsf_remote_mem_create (dacs_pvoid_t *addr*, **dacs_int64_t** *size*, **DACS_MEMORY_ACCESS_MODE_T** *access_mode*, **dacs_remote_mem_t** *mem*, **DACS_ERR_T** *rc***)**

Call parameters

`addr`	**C**: a pointer to the base address of the memory region to be shared. **Fortran**: the base address of the memory region to be shared. Use `dacsf_makevoid` to create the handle.
`size`	The size of the memory region in bytes.
`access_mode`	The access mode to be given to the memory region.

Return parameters

`mem`	**C**: a pointer to a remote memory handle to be filled in. **Fortran**: a remote memory handle to be filled in.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_remote_mem_create` service creates and returns a handle associated with the given memory region. The returned handle can be used with the `dacs_remote_mem_share()` service to share and gain access to remote shared memory.

The access mode may be one of:
- `DACS_READ_ONLY`
- `DACS_WRITE_ONLY`
- `DACS_READ_WRITE`

Cell/B.E.: this call is not supported on the SPU, and calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The service `dacs_remote_mem_create` returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.

- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.
- `DACS_ERR_INVALID_SIZE`: a size of zero was requested.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: this call is not currently supported.

SEE ALSO

dacs_remote_mem_share(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3), dacsf_makevoid(3)

dacs_remote_mem_share

NAME

dacs_remote_mem_share - Share memory region access with a remote process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_remote_mem_share (de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_remote_mem_t** *mem***)**

Fortran syntax

dacsf_remote_mem_share (dacs_de_id_t *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_remote_mem_t** *mem*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_de`	The target DE for the share.
`dst_pid`	The target process for the share.
`mem`	The handle of the remote memory to be shared.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_remote_mem_share` service shares the specified remote memory handle from the current process to the remote process specified by `dst_de` and `dst_pid`. This service then blocks, waiting for a matching call to the `dacs_remote_mem_accept` service on the remote side.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: this call is not supported on the SPU, and calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The `dacs_remote_mem_share` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_INVALID_HANDLE`: the remote memory handle is invalid.
- `DACS_ERR_NOT_OWNER`: this operation is only valid for the owner of the resource.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: this call is not currently supported.
- `DACS_ERR_NO_RESOURCE`: no resources are currently available.

SEE ALSO

dacs_remote_mem_create(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3)

dacs_remote_mem_accept

NAME

dacs_remote_mem_accept - Accept access to a remote memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_remote_mem_accept (de_id_t *src_de*, **dacs_process_id_t** *src_pid*, **dacs_remote_mem_t** **mem*)

Fortran syntax

dacsf_remote_mem_accept (dacs_de_id_t *src_de*, **dacs_process_id_t** *src_pid*, **dacs_remote_mem_t** *mem*, **DACS_ERR_T** *rc*)

Call parameters	
`src_de`	The source DE which is sharing the remote memory handle.
`src_pid`	The source process which is sharing the remote memory handle.

Return parameters	
`mem`	**C**: a pointer to the accepted memory handle. **Fortran**: the accepted memory handle.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_remote_mem_accept` service blocks the caller until it receives a remote memory handle from an associated `dacs_remote_mem_share()` call. The remote memory handle is returned on success.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: attempting to use this API on the PPU to accept a memory region from the SPU will return `DACS_ERR_NOT_SUPPORTED_YET`.

RETURN VALUE

The `dacs_remote_mem_accept` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate the required resources.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: this call is not currently supported by this platform.

SEE ALSO

dacs_remote_mem_create(3), dacs_remote_mem_share(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3)

dacs_remote_mem_release

NAME

dacs_remote_mem_release - Release access to a previously accepted remote memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_remote_mem_release (dacs_remote_mem_t **mem***)**

Fortran syntax

dacsf_remote_mem_release (dacs_remote_mem_t *mem***, DACS_ERR_T** *rc***)**

Call parameter

mem	**C**: a pointer to the remote memory handle. **Fortran**: the remote memory handle.

Return parameter

rc	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_remote_mem_release` service releases a previously accepted remote memory object and invalidates the handle. When all accepters have released the object it may be destroyed by its owner. This service does not block.

Cell/B.E.: attempting to use this API on the PPU to accept a memory region from the SPU will return `DACS_ERR_NOT_SUPPORTED_YET`.

RETURN VALUE

The `dacs_remote_mem_release` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_HANDLE`: the specified handle does not refer to a valid remote memory object.
- `DACS_ERR_OWNER`: this operation is not valid for the owner of the resource.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: this call is not currently supported.
- `DACS_ERR_RESOURCE_BUSY`: an operation that uses the handle is still active.

SEE ALSO

dacs_remote_mem_create(3), dacs_remote_mem_share(3), dacs_remote_mem_accept(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3)

dacs_remote_mem_destroy

NAME

dacs_remote_mem_destroy - Invalidate remote access to the specified memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_remote_mem_destroy (dacs_remote_mem_t **mem* **)**

Fortran syntax

dacsf_remote_mem_destroy (dacs_remote_mem_t *mem*, **DACS_ERR_T** *rc***)**

Call parameter

`mem`	**C**: a pointer to a remote memory handle. **Fortran**: the remote memory handle.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_remote_mem_destroy` service invalidates the sharing of a specific memory region which has been created by `dacs_remote_mem_create()`. Only the creator of the memory region may destroy it. This service blocks until all users of the memory region have released it. The handle is invalidated on successful completion of this method.

Cell/B.E.: this call is not supported on the SPU, and calling it will result in the returning of the `DACS_ERR_NOT_SUPPORTED_YET` error indicator.

RETURN VALUE

The `dacs_remote_mem_destroy` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_HANDLE`: the remote memory handle is invalid.
- `DACS_ERR_NOT_OWNER`: this operation is only valid for the owner of the resource.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.
- `DACS_ERR_NOT_SUPPORTED_YET`: this call is not currently supported.

SEE ALSO

dacs_remote_mem_create(3), dacs_remote_mem_share(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_query(3)

dacs_remote_mem_query

NAME

dacs_remote_mem_query - Query the attributes of a remote memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_remote_mem_query (dacs_remote_mem_t *mem***, DACS_REMOTE_MEM_ATTR_T** *attr***, uint64_t** **value***)**

Fortran syntax

dacsf_remote_mem_query_mode (dacs_remote_mem_t *mem***, DACS_REMOTE_MEM_ATTR_T** *mode***, DACS_ERR_T** *rc***)**

dacsf_remote_mem_query_size (dacs_remote_mem_t *mem***, dacs_int64_t** *size***, DACS_ERR_T** *rc***)**

dacsf_remote_mem_query_addr (dacs_remote_mem_t *mem***, dacs_pvoid_t** *addr***, DACS_ERR_T** *rc***)**

Call parameters

`mem`	The handle of the remote memory area to query.
`attr`	**C only**: the attribute to be queried.

Return parameters

`value`	**C only**: a pointer to the location where the attribute value is to be returned.
`mode`	**Fortran only**: the returned access mode from `dacs_remote_mem_query_mode`.
`size`	**Fortran only**: the returned memory region size from `dacs_remote_mem_query_size` .
`addr`	**Fortran only** : the returned memory region address from `dacs_remote_mem_query_addr`.
`rc`	**Fortran only**: See Return value.

DESCRIPTION

The `dacs_remote_mem_query` service queries the attributes of the specified remote memory region. The memory region being queried must have been created or accepted by the caller.

C only: the `attr` parameter can be any of:

`DACS_REMOTE_MEM_SIZE`:

`DACS_REMOTE_MEM_ADDR`

`DACS_REMOTE_MEM_ACCESS_MODE`

C only: the contents of `value` depends on the `attr` parameter passed as shown in the following table:

- `DACS_REMOTE_MEM_SIZE`: a `uint64_t` value indicating the size of the memory region in bytes.

- `DACS_REMOTE_MEM_ADDR`: a `uint64_t` value indicating the starting virtual address within the *creating* process of the memory region.
- `DACS_REMOTE_MEM_ACCESS_MODE`: this can be one of:

  `DACS_READ_ONLY`

  `DACS_WRITE_ONLY`

  `DACS_READ_WRITE`

RETURN VALUE

The `dacs_remote_mem_query` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_INVALID_HANDLE`: the specified handle is invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_remote_mem_create(3), dacs_remote_mem_share(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3)

dacs_put

NAME

dacs_put - Initiate a data transfer from local memory into a remote memory region.

SYNOPSIS

C syntax

DACS_ERR_T dacs_put (dacs_remote_mem_t *dst_remote_mem*, **uint64_t** *dst_remote_mem_offset*, **void** **src_addr*, **uint64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_put (dacs_remote_mem_t *dst_remote_mem*, **dacs_int64_t** *dst_remote_mem_offset*, **dacs_pvoid_t** *src_addr*, **dacs_int64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters	
`dst_remote_mem`	The remote memory handle of the destination buffer.
`dst_remote_mem_offset`	The offset into the remote buffer where the put is to be performed.
`src_addr`	**C**: a pointer to the source memory buffer. **Fortran**: the source memory buffer. Use `dacsf_makevoid` to create the handle.
`size`	The amount of data to transfer in bytes.
`wid`	The communications wait identifier.
`order_attr`	An ordering attribute. See Description for further information about possible values.
`swap`	The little-endian or big-endian byte-swapping flag. See Description for further information about possible values.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

This is a non-blocking DMA operation that initiates a data copy from local memory into a remote memory region. The data source is specified as a pointer, `src_addr`, into the caller's local address space. It is the responsibility of the caller to ensure that enough memory is available to accommodate the specified size starting from this address. The data destination is specified as an offset, `dst_remote_mem_offset`, into the remote memory region `dst_remote_mem`. The remote memory region must have been previously accepted using `dacs_remote_mem_accept()`.

This operation is associated with a specified wait identifier, `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same `wid`. Successfully waiting on the completion of this operation only guarantees that the data has been sent and that the source buffer can be safely reused.

Possible values for `order_attr` are:
- `DACS_ORDER_ATTR_FENCE`: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_BARRIER`: execution of this command and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_NONE`: no ordering is enforced.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

The target remote memory region must have been previously accepted by the caller with a call to `dacs_remote_mem_accept()`.

Note: The caller of the `dacs_put()` and `dacs_get()` methods is the process that accepted the memory handle. The owner of the remote memory cannot use these functions.

Cell/B.E. this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU in to create or share remote memory.

RETURN VALUE

The `dacs_put` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_BUF_OVERFLOW`: the specified size at the given offset exceeded the bounds of the destination memory region.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_INVALID_WID`: the wait identifier is invalid.
- `DACS_ERR_INVALID_HANDLE`: the remote memory handle is invalid.
- `DACS_ERR_NO_PERM`: the resource attributes do not allow this operation.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_get(3), dacs_put_list(3), dacs_get_list(3), dacs_test(3), dacs_wait(3), dacsf_makevoid(3), dacs_remote_mem_create(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3), dacs_remote_mem_share(3)

dacs_get

NAME

dacs_get - Initiate a data transfer from a remote memory region into local memory.

SYNOPSIS

C syntax

DACS_ERR_T dacs_get (void **dst_addr*, **dacs_remote_mem_t** *src_remote_mem*, **uint64_t** *src_remote_mem_offset*, **uint64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_get (dacs_pvoid_t *dst_addr*, **dacs_remote_mem_t** *src_remote_mem*, **dacs_int64_t** *src_remote_mem_offset*, **dacs_int64_t** *size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters

`dst_addr`	**C**: a pointer to the base address of the destination memory buffer. **Fortran**: the base address of the destination memory buffer. Use `dacsf_makevoid` to create the handle.
`src_remote_mem`	The remote memory handle of the source buffer.
`src_remote_mem_offset`	The offset into the remote buffer where the get is to start.
`size`	The size of the transfer in bytes.
`wid`	A communications wait identifier.
`order_attr`	An ordering attribute. See Description for further information about possible values.
`swap`	The little-endian or big-endian byte-swapping flag. See Description for further information about possible values.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

This is a non-blocking DMA operation that initiates a data copy from a remote memory region into local memory. The data source is specified as an offset, `src_remote_mem_offset`, into the remote memory region `src_remote_mem`. The remote memory region must have been previously accepted using `dacs_remote_mem_accept()`. The data destination is specified as a pointer, `dst_addr`, into the caller's local address space. It is the responsibility of the caller to ensure that enough memory is available to accommodate the specified size starting from this address.

This operation is associated with a specified wait identifier, `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same `wid`. Successfully waiting on the completion of this operation guarantees that all data is available locally.

Possible values of `order_attr` are:
- `DACS_ORDER_ATTR_FENCE`: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_BARRIER`: execution of this command and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same `wid` have completed.
- `DACS_ORDER_ATTR_NONE`: no ordering is enforced.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or share remote memory.

Note: The user of the `dacs_put()` and `dacs_get()` methods is the process that accepted the memory handle. The owner of the remote memory cannot use these functions.

RETURN VALUE

The `dacs_get` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_INVALID_SIZE`: the specified size at the given offset exceeded the bounds of the source memory region.
- `DACS_ERR_INVALID_WID`: the wait identifier is invalid.
- `DACS_ERR_INVALID_HANDLE`: the remote memory handle is invalid.
- `DACS_ERR_NO_PERM`: the resource attributes do not allow this operation.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_put(3), dacs_put_list(3), dacs_get_list(3), dacs_test(3), dacs_wait(3), dacsf_makevoid(3), dacs_remote_mem_create(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3), dacs_remote_mem_share(3)

dacs_put_list

NAME

dacs_put_list - Initiate a scatter or gather data transfer into a remote memory region from a local memory.

SYNOPSIS

C syntax

DACS_ERR_T dacs_put_list, (dacs_remote_mem_t *dst_mem*, **dacs_dma_list_t** * *dst_list*, **uint32_t** *dst_count*, **void** **src_addr*, **dacs_dma_list_t** **src_list*, **uint32_t** *src_count*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_put_list (dacs_remote_mem_t *dst_mem*, **dacs_dma_list_t** *dst_list*, **dacs_int32_t** *dst_count*, **dacs_pvoid_t** *src_addr*, **dacs_dma_list_t** *src_list*, **dacs_int32_t** *src_count*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters

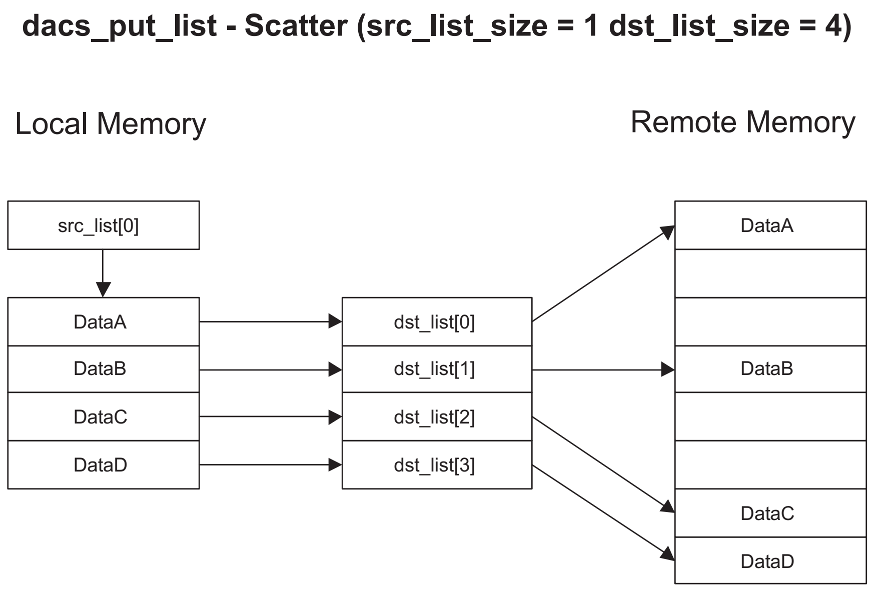
`dst_mem`	The remote memory handle for the destination buffer.
`dst_list`	**C**: a pointer to a list of entries describing transfer locations in the destination buffer. **Fortran**: a list of entries describing transfer locations in the destination buffer. Use `dacsf_makevoid` to create the handles.
`dst_count`	The number of elements in the destination DMA list.
`src_addr`	**C**: the base address of the source memory buffer. **Fortran**: The base address of the source memory buffer. Use `dacsf_makevoid` to create the handle.
`src_list`	**C**: a pointer to a list of entries describing transfer locations in the source buffer. **Fortran**: a list of entries describing transfer locations in the source buffer. Use `dacsf_makevoid` to create the handles.
`src_count`	The number of elements in the source DMA list.
`wid`	The communication wait identifier associated with this transfer.
`order_attr`	Ordering attribute. See Description for further information about possible values.
`swap`	The little-endian or big-endian byte-swapping flag. See Description for further information about possible values.

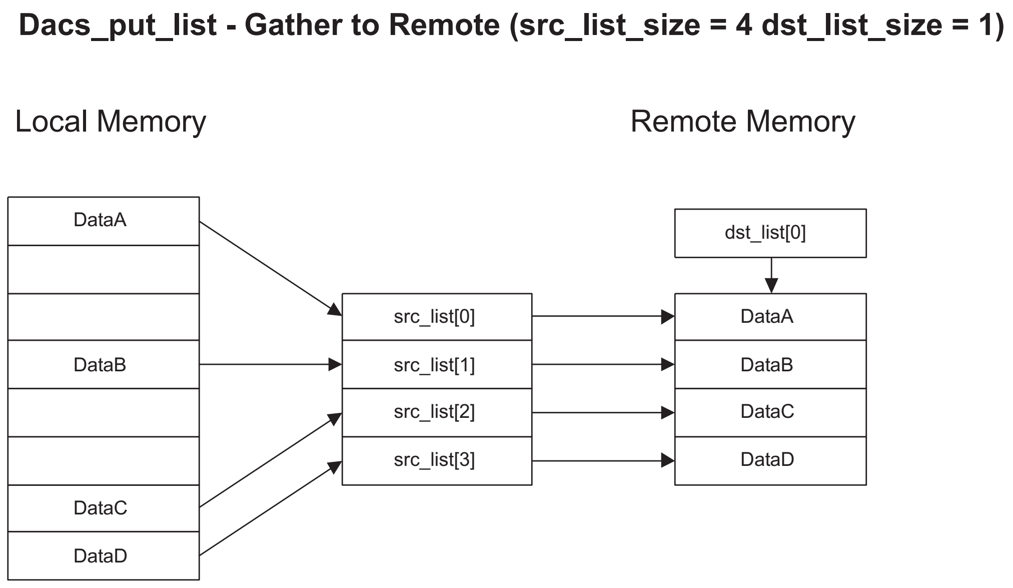
Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

This is a non-blocking DMA operation that initiates a scatter or gather data copy from local memory into a remote memory region. This method can either scatter contiguous local data blocks across numerous, possibly disjunct, remote memory locations or gather numerous, possibly disjunct, local data blocks into a contiguous remote data area. The data source is specified through the list, `src_dma_list`, which contains size and offset pairings that are applied relative to the specified local memory pointer `src_addr`. It is the responsibility of the caller to ensure that enough memory is available to accommodate all size and offset pairings, starting from this address. The data destination is specified through the list, `dst_dma_list`, which contains size and offset pairings that are applied relative to the specified remote memory region `dst_remote_mem`. This remote memory region must have been previously accepted using `dacs_remote_mem_accept()`.





Get list from a remote memory region with source list size = 4

This operation is associated with a specified wait identifier: `wid`. To ensure that the initiated data transfer has completed, either `dacs_wait()` or `dacs_test()` must be successfully called using the same wid. Successfully waiting on the completion of this operation only guarantees that the data has been sent and that the source buffer can be safely reused.

Possible values of order_attr are:
- DACS_ORDER_ATTR_FENCE: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same wid have completed.
- DACS_ORDER_ATTR_BARRIER: execution of this operation and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same wid have been completed.
- DACS_ORDER_ATTR_NONE: no ordering is enforced.

The services provide the following types of byte swapping:

DACS_BYTE_SWAP_DISABLE: no byte-swapping.

Hybrid only: the following values are also supported:

DACS_BYTE_SWAP_HALF_WORD: byte-swapping for halfwords (2 bytes).

DACS_BYTE_SWAP_WORD: byte-swapping for words (4 bytes).

DACS_BYTE_SWAP_DOUBLE_WORD: byte-swapping for double words (8 bytes)

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or share remote memory.

RETURN VALUE

The dacs_put_list service returns an error indicator defined as:
- DACS_SUCCESS: the put was started successfully.
- DACS_ERR_INVALID_HANDLE: the remote or local handle was invalid or the local memory region may have not been registered.
- DACS_ERR_BUF_OVERFLOW: this can be for one of the following reasons:
  1. Source list is too large for destination list.
  2. Destination list is too large for local (destination) region.
  3. Destination list is out-of-bounds of local (destination) region.
- DACS_ERR_INVALID_SIZE: this can be for one of the following reasons:
  1. Destination list is too large for source list.
  2. Source list too large for remote (source) region.
  3. Source list is out-of-bounds of remote (source) region.
  4. The destination or the source list size is zero.
  5. On Cell /B.E., neither the destination nor the source list size is equal to 1.
- DACS_ERR_INVALID_WID: the wid is not reserved.
- DACS_ERR_INVALID_ATTR: the order or byteswap attribute is invalid.
- DACS_ERR_NO_RESOURCE: there are no local resources available.
- DACS_ERR_INVALID_ADDR: the remote or local DMA list address is invalid.
- DACS_ERR_NO_PERM: Local memory regions has no write access or remote memory region has no read access.
- DACS_ERR_NOT_ALIGNED: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- DACS_ERR_NOT_SUPPORTED_YET: the DaCS function is currently unsupported by this platform.

- DACS_ERR_NOT_INITIALIZED: DaCS has not been initialized.

SEE ALSO

dacs_put(3), dacs_get(3), dacs_get_list(3), dacs_test(3), dacs_wait(3), dacs_makevoid(3), dacs_remote_mem_create(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3), dacs_remote_mem_share(3), dacs_makevoid(3)

dacs_get_list

NAME

dacs_get_list - Initiate a scatter or gather data transfer from a remote memory region into local memory.

SYNOPSIS

C syntax

DACS_ERR_T dacs_get_list (void **dst_addr*, **dacs_dma_list_t** **dst_dma_list*, **uint32_t** *dst_list_size*, **dacs_remote_mem_t** *src_remote_mem*, **dacs_dma_list_t** **src_dma_list*, **uint32_t** *src_list_size*, **dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_get_list (dacs_pvoid_t *dst_addr*, **dacs_dma_list_t** *dst_dma_list*, **dacs_int32_t** *dst_list_size*, **dacs_remote_mem_t** *src_remote_mem*, **dacs_dma_list_t** *src_dma_list*, **dacs_int32_t** *src_list_size* **, dacs_wid_t** *wid*, **DACS_ORDER_ATTR_T** *order_attr*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters

`dst_addr`	**C**: a pointer to the base address of the destination memory buffer. **Fortran**: the base address of the destination memory buffer. Use `dacsf_makevoid` to create the handle.
`dst_dma_list`	**C**: a pointer to a list of entries describing transfer locations in the destination buffer. **Fortran**: a list of entries describing transfer locations in the destination buffer. Use `dacsf_makevoid` to create the handles.
`dst_list_size`	The number of elements in the destination DMA list.
`src_remote_mem`	A handle for the remote source memory buffer.
`src_dma_list`	**C**: a pointer to a list of entries describing transfer locations in the source buffer. **Fortran**: a list of entries describing transfer locations in the source buffer. Use `dacsf_makevoid` to create the handles.
`src_list_size`	The number of elements in the source DMA list.
`wid`	The communication wait identifier associated with this transfer.
`order_attr`	Ordering attribute. See Description for further information about possible values.
`swap`	The little-endian or big-endian byte-swapping flag. See Description for further information about possible values.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

This is a non-blocking DMA operation that initiates a scatter or gather data copy from a remote memory region into local memory. This method can either scatter contiguous remote data blocks across numerous, possibly disjunct, local memory locations or gather numerous, possibly disjunct, remote data blocks into a contiguous local data area. The data source is specified through the `list`, src_dma_list, which contains size and offset pairings that are applied relative to the specified remote memory region src_remote_mem. This remote memory region must have been previously accepted using dacs_remote_mem_accept(). The data destination is specified through the list, dst_dma_list, which contains size and offset pairings that are applied relative to the specified local memory pointer dst_addr. It is the responsibility of the caller to ensure that enough memory is available to accommodate all size and offset pairings, starting from this address.

This operation is associated with a specified wait identifier wid. To ensure that the initiated data transfer has completed, either dacs_wait() or dacs_test() must be successfully called using the same wid. Successfully waiting on the completion of this operation guarantees that all data is available locally.

Possible values of order_attr are:
- DACS_ORDER_ATTR_FENCE: execution of this operation is delayed until all previously issued DMA operations to the same DE using the same wid have completed.
- DACS_ORDER_ATTR_BARRIER: execution of this operation and all subsequent DMA operations are delayed until all previously issued DMA operations to the same DE using the same wid have been completed.
- DACS_ORDER_ATTR_NONE: no ordering is enforced.

Cell/B.E.: this API is not supported by the PPU for accessing remote memory on the SPU. This is due to the inability of the SPU to create or share remote memory.

The services provide the following types of byte swapping:

DACS_BYTE_SWAP_DISABLE: no byte-swapping.

Hybrid only: the following values are also supported:

DACS_BYTE_SWAP_HALF_WORD: byte-swapping for halfwords (2 bytes).

DACS_BYTE_SWAP_WORD: byte-swapping for words (4 bytes).

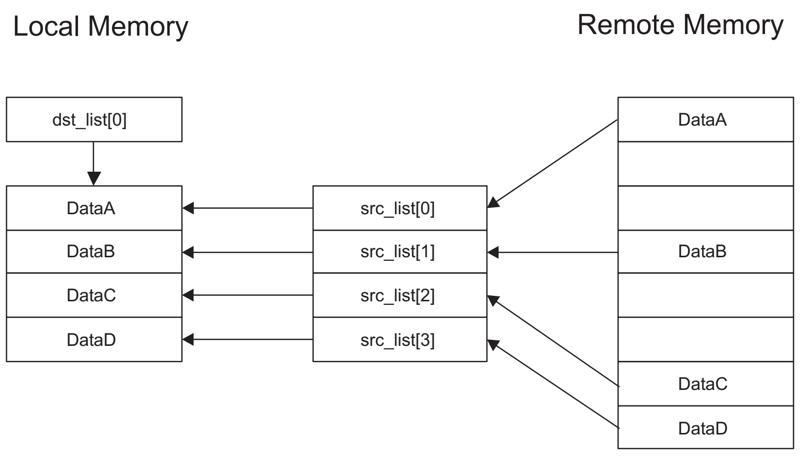
DACS_BYTE_SWAP_DOUBLE_WORD: byte-swapping for double words (8 bytes)

The destination address for each DMA operation is an effective address formed by the sum of dst_addr and the offset specified in each DMA list element. The assumption is that the destination buffers for the data are all within a contiguous buffer starting at dst_addr. For cases where the destination buffers may not be in a contiguous buffer with a known base address, a destination address of zero may be specified. In this case the actual address of the destination buffer can be used as the offset in the DMA list element.

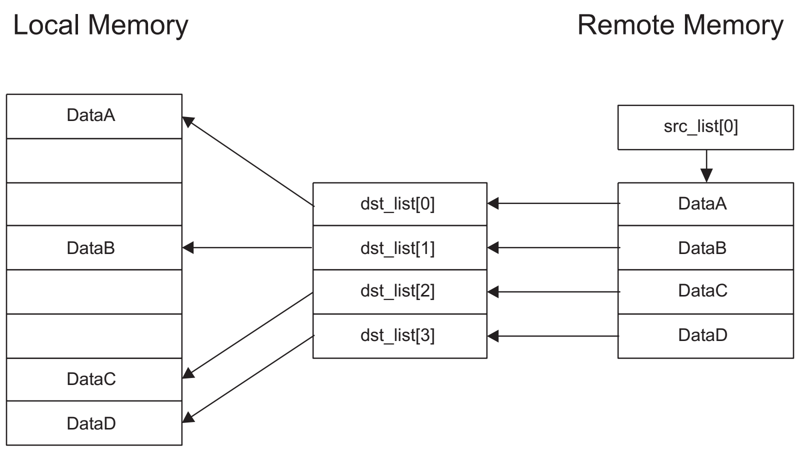
This is an asynchronous service in that the data transfers are only initiated (but not completed) when it returns. To ensure completion of the transfer you should make a call to dacs_wait() or dacs_test() passing the wait identifier.

The target remote memory region must have been previously accepted by the caller with a call to dacs_remote_mem_accept().

dacs_get_list - Gather (src_list_size = 4 dst_list_size = 1)



dacs_get_list - Scatter to local (src_list_size = 1 dst_list_size = 4)



Get list from a remote memory region with source list size = 1

RETURN VALUE

The `dacs_get_list` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_BUF_OVERFLOW`: the buffer has overflowed - the specified offset or size of one or more list elements exceed the bounds of the target buffer.
- : the buffer is not aligned correctly for the size of the transfer.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_INVALID_SIZE`: this error is returned if the offset+size is outside the src/local address range specified.

- `DACS_ERR_INVALID_WID`: the wait identifier is invalid.
- `DACS_ERR_INVALID_HANDLE`: the remote memory handle is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.
- `DACS_ERR_NO_PERM`: the resource attributes do not allow this operation.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_put(3), dacs_get(3), dacs_put_list(3), dacs_test(3), dacs_wait(3), dacs_makevoid(3) dacs_remote_mem_create(3), dacs_remote_mem_accept(3), dacs_remote_mem_release(3), dacs_remote_mem_destroy(3), dacs_remote_mem_query(3), dacs_remote_mem_share(3)

Message passing

The messaging passing services provide two way communications using the familiar send/recv model. These services are asynchronous, but can be synchronized using the `dacs_test()` and `dacs_wait()` services as needed.

Message passing usage scenarios

The table below describes the process of message passing.

Sender DE	Receiver DE

The sender initiates the send of its data:
`dacs_send(`*`msg_addr`*`,`
    `msg_size``,`
    `recv_de``,`
    `recv_pid``,`
    `msg_stream``,`
    `send_wid``,`*`swap`*`)`
 	
This DE posts a request to send a message of *msg_size* bytes, from its local memory starting at *msg_addr* to the recipient DE/PID. The message is sent via stream ID *msg_stream*. The memory transfer is tracked through the wait identifier *send_wid*. Intermediate byte swapping is performed based on the swap type *swap*.
	The recipient of the message initiates the receive:
	`dacs_recv(`*`local_addr`*`,`
	    `msg_size``,`
	    `send_de``,`
	    `send_pid``,`
	    `msg_stream``,`
	    `recv_wid``,`*`swap`*`)`
	
	This DE posts a request to receive a message of *msg_size* bytes, into local memory starting at *local_addr*, from the sender DE/PID. The message is expected to be sent via stream ID *msg_stream*. The memory transfer is tracked through the wait identifier *recv_wid*. Intermediate byte swapping is performed based on the swap type *swap*.

Test whether the SEND has completed: `dacs_test(`*`send_wid`*`)` In order to verify the completion of the message passing operation, the wait identifier must be successfully waited on or tested. Multiple messages or DMAs may be issued under a single wait identifier which only requires a single call to wait or test. The call to wait or test will not complete until all DMAs or messages associated with the wait identifier have completed. Test is a non-blocking status check of all outstanding transactions for wait identifier *send_wid*. If any transfers, associated with *send_wid*, have not completed, the call will return with a busy status. In the case of a SEND, the successful completion of waiting or testing only means that the local memory buffer at *msg_addr* is available for re-use. It does not guarantee that the message has successfully arrived in the receiver's buffer.	Wait on the RECEIVE to complete: `dacs_wait(`*`recv_wid`*`)` In order to verify the completion of the message passing operation, the wait identifier must be successfully waited on or tested. Multiple messages or DMAs may be issued under a single wait identifier which only requires a single call to wait or test. The call to wait or test will not complete until all DMAs or messages associated with the wait identifier have completed. Wait is a blocking status check of all outstanding transactions for wait identifier *recv_wid*. It will not return until all transfers, associated with *recv_wid*, have either completed or encountered an error. In the case of a RECEIVE, the successful completion of waiting or testing guarantees that a message of *msg_size* bytes has completely arrived in the associated local memory buffer at *local_addr*.

dacs_send

NAME

dacs_send - Send a message to another process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_send (void **src_data*, **uint32_t** *size*, **de_id_t** *dst_de*, **dacs_process_id_t** *dst_pid*, **uint32_t** *stream*, **dacs_wid_t** *wid*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_send (dacs_pvoid_t *src_data*, **dacs_int32_t** *size*, **dacs_de_id_t** *dst_de*, **dacs_process_id_t** *dst_pid*, **dacs_stream_t** *stream*, **dacs_wid_t** *wid*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters

`src_data`	**C**: a pointer to the beginning of the source (send) message buffer. **Fortran**: the beginning of the source (send) message buffer. Use dacsf_makevoid to create the handle
`size`	The size of the message buffer.
`dst_de`	The message destination DE.
`dst_pid`	The message destination process.
`stream`	The identifier of the stream on which the message is to be sent. The stream id must be between 0 and DACS_STREAM_UB inclusive.
`wid`	The wait identifier to synchronize on.
`swap`	The little-endian or big-endian byte-swapping flag. See Description for further information about possible values.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_send` service asynchronously sends a message to another process. Upon successful return a send operation is either pending or in progress. Use `dacs_test()` or `dacs_wait()` to test for completion on this DE, so that the local buffer can be reused or changed.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

`DACS_BYTE_SWAP_HALF_WORD`: byte-swapping for halfwords (2 bytes).

`DACS_BYTE_SWAP_WORD`: byte-swapping for words (4 bytes).

`DACS_BYTE_SWAP_DOUBLE_WORD`: byte-swapping for double words (8 bytes)

Note: The size of the buffer at the destination process must be greater than or equal to amount of data sent; otherwise the send operation fails silently. This error will later be reported by `dacs_test()` or `dacs_wait()` as `DACS_ERR_BUF_OVERFLOW`.

Cell/B.E.: up to 8 outstanding (unmatched) sends and receives are supported.

Note: When communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: can return `DACS_ERR_NOT_SUPPORTED_YET` if a swap flag other than `DACS_BYTE_SWAP_DISABLE` is used.

RETURN VALUE

The `dacs_send` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_INVALID_SIZE`: the size is not supported by the platform.
- `DACS_ERR_INVALID_WID`: the wait identifier is invalid.
- `DACS_ERR_INVALID_STREAM`: the stream identifier is invalid.
- `DACS_ERR_NOT_ALIGNED`: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- `DACS_ERR_NOT_SUPPORTED_YET`: the DaCS function is currently unsupported by this platform.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_recv(3), dacs_wait(3), dacs_test(3), dacsf_makevoid(3)

dacs_recv

NAME

dacs_recv - Receive a message from another process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_recv (void **dst_data*, **uint32_t** *size*, **de_id_t** *src_de*, **dacs_process_id_t** *src_pid*, **uint32_t** *stream*, **dacs_wid_t** *wid*, **DACS_BYTE_SWAP_T** *swap***)**

Fortran syntax

dacsf_recv (dacs_pvoid *dst_data*, **dacs_int32_t** *size*, **de_id_t** *src_de*, **dacs_process_id_t** *src_pid*, **dacs_stream_t** *stream*, **dacs_wid_t** *wid*, **DACS_BYTE_SWAP_T** *swap*, **DACS_ERR_T** *rc***)**

Call parameters

`dst_data`	**C**: a pointer to the beginning of the destination (receive) data buffer. **Fortran**: the beginning of the destination(receive) message buffer. Use `dacsf_makevoid` to create the handle.
`size`	The size of the message buffer.
`src_de`	The message source DE.
`src_pid`	The message source process.
`stream`	The stream on which to receive the message, or **DACS_STREAM_ALL**.
`wid`	The wait identifier to synchronize on.
`swap`	The little-endian or big-endian byte-swapping flag.See Description for further information about possible values.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_recv` service asynchronously receives a message from another process. Upon successful return a receive operation is either pending or in progress. You should use `dacs_test()` or `dacs_wait()` to test for completion.

The number of bytes sent by the source process must be less than or equal to the local buffer size, otherwise the receive operation fails.

Stream identifiers are used to select messages for reception. A message will be received if the stream identifier of the message matches the stream identifier specified to `dacs_recv()`, or if `DACS_STREAM_ALL` is specified. Stream identifier values must be between 0 and `DACS_STREAM_UB` inclusive.

The services provide the following types of byte swapping:

`DACS_BYTE_SWAP_DISABLE`: no byte-swapping.

Hybrid only: the following values are also supported:

DACS_BYTE_SWAP_HALF_WORD: byte-swapping for halfwords (2 bytes).

DACS_BYTE_SWAP_WORD: byte-swapping for words (4 bytes).

DACS_BYTE_SWAP_DOUBLE_WORD: byte-swapping for double words (8 bytes)

The swap flag must be the same at both ends of the transfer. If not the completion test (dacs_test() or dacs_wait()) will fail with DACS_ERR_BYTESWAP_MISMATCH, and no data is transferred.

Note: When communicating with the parent node, DACS_DE_PARENT and DACS_DE_PID must be used.

Cell/B.E.: up to 8 outstanding (unmatched) sends and receives are supported.

Cell/B.E.: can return DACS_ERR_NOT_SUPPORTED_YET if a swap flag other than DACS_BYTE_SWAP_DISABLE is used.

RETURN VALUE

The dacs_recv service returns an error indicator defined as:

- DACS_SUCCESS: normal return.
- DACS_ERR_INVALID_ADDR: the pointer is invalid.
- DACS_ERR_INVALID_ATTR: the flag or enumerated constant is invalid.
- DACS_ERR_NO_RESOURCE: unable to allocate required resources.
- DACS_ERR_INVALID_DE: the specified DE is either invalid or not reserved.
- DACS_ERR_INVALID_PID: the specified PID does not refer to an active process.
- DACS_ERR_INVALID_TARGET: this operation is not allowed for the target process.
- DACS_ERR_INVALID_SIZE: the DaCS function is currently unsupported by this platform.
- DACS_ERR_INVALID_WID: the wait identifier is invalid.
- DACS_ERR_INVALID_STREAM: the stream identifier is invalid.
- DACS_ERR_NOT_ALIGNED: an alignment conflict exists between the swap flag granularity and the address, offset, or size.
- DACS_ERR_NOT_SUPPORTED_YET: the DaCS function is currently unsupported by this platform.
- DACS_ERR_NOT_INITIALIZED: DaCS has not been initialized.

SEE ALSO

dacs_send(3), dacs_wait(3), dacs_test(3), dacsf_makevoid(3)

Mailboxes

The mailbox services provide a simple method of passing a single 32-bit unsigned word between processes. These services use a blocking read/write model. The mailbox is a FIFO queue with an implementation-specific depth.

Mailboxes provide communication between an accelerator and it's host. Each accelerator has an incoming and outgoing mailbox to the host (full duplex), each has a size of 4 when using Dacs for Cell/B.E. and a size of 32 when using DaCS for Hybrid. Mailboxes will block when the implementation specific size is reached.

Mailbox usage scenario

The table below shows how to use the mailbox services.

Writer DE	**Reader DE**

Test whether the reader has any available incoming mailbox slots:
`dacs_mailbox_test(DACS_TEST_MAILBOX_WRITE,`
`        reader_de,`
`        reader_pid,`
`        &avail_slots)`
 	
This test checks that the reader DE/PID has available mailbox slots for posting a new mailbox message. The test returns a non-zero value if there are mailbox slots for the target reader.	

	Test whether the reader has any available incoming mailbox slots:
	`dacs_mailbox_test(DACS_TEST_MAILBOX_READ,`
	`        writer_de,`
	`        writer_pid,`
	`        &avail_mail)`
	
	This test checks that the reader DE/PID has available mailbox slots for posting a new mailbox message. The test returns a non-zero value if there are mailbox slots for the target reader.

Post a mail message for the target DE/PID:
`dacs_mailbox_write(&message,`
`         reader_de,`
`         reader_pid )`
 	
The writer can post a mail message for the reader. If no mailbox slots are available then the call will block until a slot is available. Upon return, it is guaranteed that the mail message, pointed to by *message* is available in the reader's mailbox.
	Fetch a mail message from the source DE/PID:
	`dacs_mailbox_read(&message,`
	`        writer_de,`
	`        writer_pid )`
	
	The reader can check whether a mail message from the writer has been delivered. If no messages are available from the writer then the call will block until one is available. Upon return, the incoming mail message is removed from the mailbox and stored at the location pointed to by *message*.

dacs_mailbox_write

NAME

dacs_mailbox_write - Send a single 32-bit value to another process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mailbox_write (uint32_t **msg*, **de_id_t** *dst_de*, **dacs_process_id_t** *dst_pid***)**

Fortran syntax

dacsf_mailbox_write (dacs_int32_t *msg*, **dacs_de_id_t** *dst_de*, **dacs_process_id_t** *dst_pid*, **DACS_ERR_T** *rc***)**

Call parameters

`msg`	**C**: a pointer to the message to write. **Fortran**: message to write
`dst_de`	The message destination DE.
`dst_pid`	The destination process id.

Return parameter

`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mailbox_write` service writes a single 32-bit unsigned integer to the specified target mailbox. There are a number of mailbox slots for each process; this number is defined by the implementation. If the destination has an empty mailbox slot this service returns immediately. Otherwise this service blocks until a slot becomes available. Mailbox operations are ordered with respect to local and remote memory operations, as well as message passing operations.

Byte-swapping is done automatically if required. A DE cannot write to its own mailbox and can only read from its own mailbox. Any attempt to do otherwise returns an error.

Note: when communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

Cell/B.E.: The mailbox depth is limited to 4 incoming and 4 outgoing mailboxes for each SPU.

Hybrid: The mailbox depth is limited to 32 incoming and outgoing mailboxes for each PID.

RETURN VALUE

The `dacs_mailbox_write` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.

- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mailbox_read(3), dacs_mailbox_test(3)

dacs_mailbox_read

NAME

dacs_mailbox_read - Receive a single 32-bit value from another process.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mailbox_read (uint32_t **msg*, **de_id_t** *src_de*, **dacs_process_id_t** *src_pid***)**

Fortran syntax

dacsf_mailbox_read (dacs_int32_t *msg*, **dacs_de_id_t** *src_de*, **dacs_process_id_t** *src_pid*, **DACS_ERR_T** *rc***)**

Call parameters

`src_de`	The message source DE.
`src_pid`	The message source process.

Return parameters

`msg`	**C**: a pointer to the message received.
	Fortran: the message received.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mailbox_read` service reads a single 32-bit unsigned integer from the specified source mailbox. There are an implementation specific number of mailbox entries for each DE and PID. If the source does not have any pending mailbox messages this service call blocks until one arrives.

Mailbox operations are ordered with respect to local and remote memory operations, as well as message passing operations.

Byte-swapping is done automatically if required. A DE cannot write to its own mailbox and can only read from its own mailbox. Any attempt to do otherwise returns an error.

Note: when communicating with the parent node, DACS_DE_PARENT and DACS_DE_PID must be used.

Cell/B.E.: The mailbox depth is limited to 4 incoming and 4 outgoing mailboxes for each SPU.

Hybrid: The mailbox depth is limited to 32 incoming and outgoing mailboxes for each PID.

RETURN VALUE

The `dacs_mailbox_read` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: operation not allowed for the target process.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mailbox_write(3), dacs_mailbox_test(3)

dacs_mailbox_test

NAME

dacs_mailbox_test - Test if a mailbox access returns data or blocks.

SYNOPSIS

C syntax

DACS_ERR_T dacs_mailbox_test (DACS_TEST_MAILBOX_T *rw_flag*, **de_id_t** *de*, **dacs_process_id_t** *pid*, **int32_t** **mbox_status***)**

Fortran syntax

dacsf_mailbox_test (DACS_TEST_MAILBOX_T *rw_flag*, **dacs_de_id_t** *de*, **dacs_process_id_t** *pid*, **dacs_int32_t** *mbox_status*, **DACS_ERR_T** *rc***)**

Call parameters	
`rw_flag`	Flag to indicate which mailbox to test.
`de`	The DE owning the mailbox to test.
`pid`	The process owning the mailbox to test

Return parameters	
`mbox_status`	**C**: a pointer to the location where the mailbox status is returned. **Fortran**: mailbox status.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_mailbox_test` service allows the programmer to test if the mailbox will block before calling `dacs_mailbox_read()` or `dacs_mailbox_write()`.

`rw_flag` can be any of:
- `DACS_TEST_MAILBOX_READ`: test the read mailbox to see if a call to `dacs_mailbox_read()` will block, or
- `DACS_TEST_MAILBOX_WRITE`: test the write mailbox to see if a call to `dacs_mailbox_write()` will block.

`mbox_status` can be:
- zero if the mailbox read or write will block,
- non-zero if the mailbox read or write will not block.

Note: when communicating with the parent node, `DACS_DE_PARENT` and `DACS_DE_PID` must be used.

RETURN VALUE

The `dacs_mailbox_test` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.

- `DACS_ERR_INVALID_DE`: the specified DE is either invalid or not reserved.
- `DACS_ERR_INVALID_PID`: the specified PID does not refer to an active process.
- `DACS_ERR_INVALID_TARGET`: this operation is not allowed for the target process.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_mailbox_read(3), dacs_mailbox_write(3)

Wait identifier management services

These services are intended to manage wait identifiers (WIDs), which are used to synchronize data communication. A WID is required for the data communication services, and is used to test for completion of asynchronous data transfers.

dacs_wid_reserve

NAME

dacs_wid_reserve - Reserve a wait identifier.

SYNOPSIS

C syntax

DACS_ERR_T dacs_wid_reserve (dacs_wid_t **wid***)**

Fortran syntax

dacsf_wid_reserve (dacs_wid_t *wid***, DACS_ERR_T** *rc***)**

Return parameters

`wid`	**C**: A pointer to the reserved wait identifier. **Fortran**: The reserved wait identifier.
`rc`	**Fortran only**: See Return value.

DESCRIPTION

The `dacs_wid_reserve` service reserves a wait identifier.

RETURN VALUE

The `dacs_wid_reserve` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NO_WIDS`: no wait identifiers are available.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_wid_release(3)

dacs_wid_release

NAME

dacs_wid_release - Release a reserved wait identifier.

SYNOPSIS

C syntax

DACS_ERR_T dacs_wid_release (dacs_wid_t ***wid*****)**

Fortran syntax

dacsf_wid_release (dacs_wid_t *wid***, DACS_ERR_T** *rc***)**

Call parameter	
`wid`	**C**: a pointer to the wait identifier to be released. **Fortran**: the wait identifier to be released.

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_wid_release` service releases the reserved wait identifier. If a data transfer using the wait identifier is still active, an error is returned and the wait identifier is not released. On successful return the wait identifier is invalidated.

RETURN VALUE

The `dacs_wid_release` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return; the wait identifier was invalidated.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_WID_ACTIVE`: a data transfer involving the wait identifier is still active.
- `DACS_ERR_INVALID_WID`: the specified wait identifier is not reserved.

SEE ALSO

dacs_wid_reserve(3)

Transfer completion

A wait identifier is reserved and assigned to an asynchronous data communication operation on initiation. These routines test the wait identifier to see if the communication operation has completed.

dacs_test

NAME

dacs_test - Test if communication operations have finished on this `wid` so the parameters can be changed or reused.

SYNOPSIS

C syntax

DACS_ERR_T dacs_test (dacs_wid _t *wid*)

Fortran syntax

dacsf_test (dacs_wid _t *wid*, DACS_ERR_T *rc*)

Call parameter	
`wid`	A communication wait identifier.

Return parameter	
`rc`	**Fortran only**: See Return value.

DESCRIPTION

The dacs_test service checks the data transfers for the given communication wait identifier and returns a status. This service can be called multiple times. When it returns `DACS_WID_READY` and no other operations are performed, subsequent calls will then return `DACS_ERR_WID_NOT_ACTIVE`.

RETURN VALUE

The `dacs_test` service returns an error indicator defined as:

- `DACS_WID_READY`: all data transfers have completed.
- `DACS_WID_BUSY`: one or more data transfers have not completed.
- `DACS_ERR_WID_NOT_ACTIVE`: there are no outstanding transfers to test.
- `DACS_ERR_INVALID_WID`: the specified wait identifier is invalid.
- `DACS_ERR_BYTESWAP_MISMATCH`: the Little-endian / Big-endian architectures at the ends of the transfer are incompatible.
- `DACS_ERR_BUF_OVERFLOW`: the data to be transferred is too large for the receive buffer.
- `DACS_ERR_INVALID_PID`: a target process of an operation has either been killed, died, or already exited while this process is waiting for the operations to complete.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_wait(3), dacs_put(3), dacs_get(3), dacs_put_list(3), dacs_get_list(3), dacs_put_list(3), dacs_wid_reserve(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_put_list(3), dacs_mem_get_list(3)

dacs_wait

NAME

dacs_wait - Wait for communication operations to finish on this `wid` so the parameters can be changed or reused.

SYNOPSIS

C syntax

DACS_ERR_T dacs_wait (dacs_wid_t *wid*)

Fortran syntax

dacsf_wait (dacs_wid_t *wid*, DACS_ERR_T *rc*)

Call parameter	
`wid`	A communication wait identifier.

Return parameter	
`rc`	**Fortran only**: See Return value.

DESCRIPTION

The `dacs_wait` service blocks the caller, waiting for outstanding data transfers for the given wait identifier to complete. It returns when all outstanding transfers are finished. If one or more of the transfers fails, the first failure encountered is reported.

Successfully completing a DaCS wait ensures that rDMA data is available at the remote location and indicates that the local buffer is available for re-use. Both past and future memory accesses are ordered with respect to this call.

`dacs_wait()` will not return until the rDMAs associated with the wid parameter have completed. For example the data associated with a `dacs_put()` will be available for use on the remote system when `dacs_wait()` returns.

Note: On `dacs_send()`, successful completion of wait does not guarantee that data is available at the remote location. It only guarantees that the local buffer may be safely re-used.

RETURN VALUE

The `dacs_wait` service returns an error indicator defined as:
- `DACS_WID_READY`: all data transfers have completed.
- `DACS_ERR_WID_NOT_ACTIVE`: there are no outstanding transfers to test.
- `DACS_ERR_INVALID_WID`: the specified wait identifier is invalid.
- `DACS_ERR_BYTESWAP_MISMATCH`: the Little-endian / Big-endian architectures at the ends of the transfer are incompatible.
- `DACS_ERR_BUF_OVERFLOW`: the data to be transferred is too large for the receive buffer.

- `DACS_ERR_INVALID_PID`: a target process of an operation has either been killed, died, or already exited while this process is waiting for the operations to complete.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_test(3), dacs_put(3), dacs_get(3), dacs_put_list(3), dacs_get_list(3), dacs_wid_reserve(3), dacs_mem_get(3), dacs_mem_put(3), dacs_mem_put_list(3), dacs_mem_get_list(3)

Chapter 11. Error handling

DaCS provides support for registration of user-created error handlers which are called under certain error conditions. The error handlers can be called for synchronous or asynchronous errors.

In DaCS any asynchronous error reported to the error handlers will cause the process to abort. This will happen when DaCS has detected a fatal error from which it cannot recover. Asynchronous errors include child failures (accelerator process) and termination requests from a parent (host process). Abnormal child termination will cause the parent to abort after calling all registered error handlers.

A normal child exit with a non-zero status will be reported asynchronously to the error handlers, but will not cause the process to abort. This allows the parent process to determine if the non-zero exit represents an error condition.

When it is called a user error handler is passed an error object describing the error, which can be inspected using services provided. The error object contains the DE and PID of the failing process. These can be used to call `dacs_de_test()` to reap its status and so allow another process to be started on that DE.

The DaCS library uses the `SIGTERM` signal for handling asynchronous errors and termination requests. A dedicated error handling thread is created in `dacs_init()` for this purpose. Applications using the DaCS library should not create any application threads before calling `dacs_init()`, and no application thread should unmask this signal.

User error handler example

User error handler registration

For this example we're going to create an user error handler called `my_errhandler`. Once this has been defined we can register the user error handler using the `dacs_errhandler_reg` API:

```
dacs_rc= dacs_errhandler_reg((dacs_error_handler_t)&my_errhandler,0);
```

Note: If the address of `my_errhandler` is not passed or the cast to `dacs_error_handler_t` is omitted the compiler will produce warnings.

User error handler code:

```
/*****************************************************************
Example of a user error handler
This includes invocations of additional functions of
the passed "dacs_error_t" error parameter
*****************************************************************/
int my_errhandler(dacs_error_t error){
        /*need local variables for passback of values */
        DACS_ERR_T dacs_rc=0;
        DACS_ERR_T dacs_error_rc;//hold code for error
        de_id_t de=0;
        dacs_process_id_t pid=0;
        uint32_t code = 0;
        const char * error_string;

        /* Get the DACS_ERR_T in the error to learn what happened */
```

```
    printf("\n\n--in my_dacs_errhandler\n");
         dacs_error_rc=dacs_rc=dacs_error_num(error);
         printf("  dacs_error_num indicates DACS_ERR_T=%d %s\n",
                dacs_rc,dacs_strerror(dacs_rc));

         /* Get the exit code in the error to learn what happened */
         dacs_rc=dacs_error_code(error,&code);
         if(dacs_rc){//if error invoking dacs_error_code
           printf("  dacs_error_code call had error DACS_ERR_T=%d %s\n",
                  dacs_rc,dacs_strerror(dacs_rc));
         }
         else {
           if (DACS_STS_PROC_ABORTED==dacs_error_rc){
              printf("  dacs_error_code signal signal=%d  ",code);
           }
           else if (DACS_STS_PROC_FAILED==dacs_error_rc){
              printf("  dacs_error_code exit code=%d\n",code);
           }
           else {//else reason is different than aborted or failed
              printf("  dacs_error_code exit/signal code=%d\n",code);
           }
         }

         /* Get the error string in the error to learn what happened */
         dacs_rc=dacs_error_str(error,&error_string);
         if(dacs_rc){//if error invoking dacs_error_str
           printf("  dacs_error_str call had error DACS_ERR_T=%d %s\n",
                  dacs_rc,dacs_strerror(dacs_rc));
         }
         else {
           printf("  dacs_error_str=%s\n",error_string);
         }

         /* what DE had this error ? */
         dacs_rc=dacs_error_de(error,&de);
         if(dacs_rc){//if error invoking dacs_error_de
           printf("  dacs_error_de call had error DACS_ERR_T=%d %s\n",
                  dacs_rc,dacs_strerror(dacs_rc));
         }
         else {
           printf("  dacs_error_de=%08x\n",de);
         }

         /* what was the dacs_process_id_t? */
         dacs_rc=dacs_error_pid(error,&pid);
         if(dacs_rc){//if error invoking dacs_error_pid
           printf("  dacs_error_pid call had error"
                  "DACS_ERR_T=%d %s\n",dacs_rc,dacs_strerror(dacs_rc));
         }
         else {
           printf("  dacs_error_pid=%ld\n",pid);
         }
         printf("exiting user error handler\n\n");
         return 0;//in SDK 3.1, return value is ignored
}
```

User error handler output

Example output if the accelerator program exits with a return code of 9:

```
--in my_dacs_errhandler
  dacs_error_num indicates DACS_ERR_T=4 DACS_STS_PROC_FAILED
  dacs_error_code exit code=9
  dacs_error_str=DACS_STS_PROC_FAILED
  dacs_error_de=01020200
  dacs_error_pid=5503
exiting user error handler
```

Example output if the accelerator program aborts:

```
--in my_dacs_errhandler
  dacs_error_num indicates DACS_ERR_T=5 DACS_STS_PROC_ABORTED
  dacs_error_code signal signal=6    dacs_error_str=DACS_STS_PROC_ABORTED
  dacs_error_de=01020200
  dacs_error_pid=5894
exiting user error handler
```

Abnormal child termination

In the case of abnormal child termination, the error code in the error object is a platform-specific code.

Hybrid: Error code is the signal number that caused the termination.

Cell/B.E.: Error code is the libspe2 exception code. See the *SPE Runtime Management Library* documentation for further information.

dacs_errhandler_reg

NAME

dacs_errhandler_reg - Register an error handler to be called when an asynchronous or fatal error occurs.

SYNOPSIS

C syntax

DACS_ERR_T dacs_errhandler_reg (dacs_error_handler_t *handler*, **uint32_t** *flags***)**

Fortran syntax

dacsf_errhandler_reg (external *handler*, **dacs_int32_t** *flags*, **DACS_ERR_T** *rc***)**

Call parameters	
`handler`	**C**: a pointer to an error handling function. **Fortran**: an error handling function reference.
`flags`	Flags for error handling options. **Note:** In DaCS 4.0 no flags are supported; the flags value passed in must be 0 (zero).

Return parameter	
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_errhandler_reg` service registers an error handler. This handler will then be called whenever an asynchronous DaCS process fails, or a synchronous DaCS process encounters a fatal error.

Note: In DaCS 4.0 the return value from the user handler will be ignored in all cases.

C only: if the error handler is coded in the form

```
uint32_t my_errhandler(dacs_error_t error)
```

then write the registration as

```
dacs_rc=dacs_errhandler_reg(&my_errhandler,0);
```

where `dacs_rc` has been declared as a variable of type `DACS_ERROR_T`.

Fortran only: Code the error handler in a module with the form:

```
FUNCTION errhandler(errdata)RESULT (rc)
```

Then code the error handler registration as:

```
call dacsf_errhandler_reg(errhandler,flags,rc)
```

A complete example of error handler registration in both C and Fortran is included in the DaCS example source code which can be found in the rpms:

dacs-examples-source*.rpm and the dacs-hybrid-examples-source*.rpm

The user-registered handler must accept a handle to an error object, and return 1 (one) or 0 (zero) to indicate whether the error is deemed fatal or not. For fatal internal errors, the process will be terminated without consideration for the handler's return value.

In **C**, the prototype of the handler is:

int (*dacs_error_handler_t)(dacs_error_t *error* **)**

In **Fortran**, the interface of the handler is:

```
function handler(errdata) RESULT(rc)
 include "dacsf.h"
 integer(kind=dacs_pvoid_t), intent(in) :: errdata
 integer(kind=dacs_err_t) :: rc
end function handler
```

RETURN VALUE

The `dacs_errhandler_reg` service returns an error indicator defined as:

- `DACS_SUCCESS`: normal return.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_INVALID_ATTR`: the flag or enumerated constant is invalid.
- `DACS_ERR_NO_RESOURCE`: unable to allocate required resources.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_strerror(3), dacs_error_num(3), dacs_error_code(3), dacs_error_str(3), dacs_error_de(3), dacs_error_pid(3)

dacs_strerror

NAME

dacs_strerror - Return a pointer to a string describing an error.

SYNOPSIS

C syntax

const char * dacs_strerror (DACS_ERR_T *errcode***)**

Fortran syntax

dacsf_strerror (DACS_ERR_T *errcode***, character** *errorstr*)

Call parameter	
`errcode`	An error code that was returned by a DaCS API.

Return parameter	
`errorstr`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_strerror` service returns a pointer to the error string for the given error code. The input error code can be any error returned by the DaCS API.

RETURN VALUE

The `dacs_strerror` service returns the error string for the given error code. In Fortran, the length of the string variable is defined by the constant `DACS_MAX_ERRSTR_LEN` in `dacsf.h`.

If no string was found, in **C** a NULL is returned, in **Fortran** an empty string is returned.

SEE ALSO

dacs_errhandler_reg(3), dacs_error_num(3), dacs_error_code(3), dacs_error_str(3), dacs_error_de(3), dacs_error_pid(3)

dacs_error_num

NAME

dacs_error_num - Return the error code for the specified error handle.

SYNOPSIS

C syntax

DACS_ERR_T dacs_error_num (dacs_error_t *error***)**

Fortran syntax

dacsf_error_num (dacs_int64_t *error***, DACS_ERR_T** *errorcode***)**

Call parameter	
`error`	An error handle.

Return parameter	
`errorcode`	**Fortran only**: see Return Value.

DESCRIPTION

The `dacs_error_num` service returns the error code associated with the specified error handle.

RETURN VALUE

The `dacs_error_num` service returns a DaCS error code, `DACS_ERR_NOT_INITIALIZED` or `DACS_ERR_INVALID_HANDLE` if the given handle does not refer to a valid error object.

SEE ALSO

dacs_errhandler_reg(3), dacs_strerror(3), dacs_error_code(3), dacs_error_str(3), dacs_error_de(3), dacs_error_pid(3)

dacs_error_code

NAME

dacs_error_code - Retrieve the extended error code from the specified error object.

SYNOPSIS

C syntax

DACS_ERR_T dacs_error_code (dacs_error_t *error*, **uint32_t** **code***)**

Fortran syntax

dacsf_error_code (dacs_int64_t *error*, **dacs_int32_t** *code*, **DACS_ERR_T** *rc*)

Call parameter

`error`	An error handle.

Return parameters

`code`	**C**: a pointer to the error code. **Fortran**: the error code.
`rc`	**Fortran** only: see Return value.

DESCRIPTION

The `dacs_error_code` service retrieves the platform-specific extended error code from the specified error object.

RETURN VALUE

The `dacs_error_code` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return; error code is returned in *code*.
- `DACS_ERR_INVALID_HANDLE`: the error handle is invalid.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_errhandler_reg(3), dacs_strerror(3), dacs_error_num(3), dacs_error_str(3), dacs_error_de(3), dacs_error_pid(3)

dacs_error_str

NAME

dacs_error_str - Retrieve the error string for the specified error object.

SYNOPSIS

C syntax

DACS_ERR_T dacs_error_str (dacs_error_t *error*, **const char** ***errstr***)**

Fortran syntax

dacsf_error_str (dacs_int64_t *error*, **character** *errstr*, **DACS_ERR_T** *rc*)

Call parameter	
`error`	An error handle.

Return parameters	
`errstr`	**C**: a pointer to the error string. **Fortran**: the error string.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_error_str` service returns the error string associated with the specified error. This is the string that is returned from `dacs_strerror()`.

In Fortran, the length of the string variable is defined by the constant `DACS_MAX_ERRSTR_LEN` in `dacsf.h`. If the string returned is larger than the *errstr* length, the string is truncated.

RETURN VALUE

The `dacs_error_str` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return: a pointer to the error string is passed back in *errstr*.
- `DACS_ERR_INVALID_HANDLE`: the specified error handle is invalid.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_errhandler_reg(3), dacs_strerror(3), dacs_error_num(3), dacs_error_code(3), dacs_error_de(3), dacs_error_pid(3)

dacs_error_de

NAME

dacs_error_de - Retrieve the originating DE for the specified error object.

SYNOPSIS

C syntax

DACS_ERR_T dacs_error_de (dacs_error_t *error*, **de_id_t** **de***)**

Fortran syntax

dacsf_error_de (dacs_int64_t *error*, **dacs_de_id_t** *de*, **DACS_ERR_T** *rc***)**

Call Parameter

`error`	An error handle.

Return parameters

`de`	**C**: a pointer indicating the DE which was the source of the error. **Fortran**: the DE which was the source of the error.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_error_de` service returns the originating DE for the specified error object.

RETURN VALUE

The `dacs_error_de` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return: the originating DE is passed back in *de*.
- `DACS_ERR_INVALID_HANDLE`: the specified error handle is invalid.
- `DACS_ERR_INVALID_ADDR`: the pointer is invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_errhandler_reg(3), dacs_strerror(3), dacs_error_num(3), dacs_error_code(3), dacs_error_str(3), dacs_error_pid(3)

dacs_error_pid

NAME

dacs_error_pid - Retrieve the originating PID for the specified error object.

SYNOPSIS

C syntax

DACS_ERR_T dacs_error_pid (dacs_error_t *error*, **dacs_process_id_t** **pid*)

Fortran syntax

dacsf_error_pid (dacs_int64_t *error*, **dacs_process_id_t** *pid*, **DACS_ERR_T** *rc*)

Call parameter

`error`	An error handle.

Return parameters

`pid`	**C**: a pointer indicating the PID which was the source of the error.
	Fortran: the PID which was the source of the error.
`rc`	**Fortran only**: see Return value.

DESCRIPTION

The `dacs_error_pid` service returns the originating PID for the specified error object.

RETURN VALUE

The `dacs_error_pid` service returns an error indicator defined as:
- `DACS_SUCCESS`: normal return; the originating PID is passed back in *pid*.
- `DACS_ERR_INVALID_HANDLE`: the specified error handle is invalid.
- `DACS_ERR_INVALID_ADDR`: the specified address is invalid.
- `DACS_ERR_NOT_INITIALIZED`: DaCS has not been initialized.

SEE ALSO

dacs_errhandler_reg(3), dacs_strerror(3), dacs_error_num(3), dacs_error_code(3), dacs_error_str(3), dacs_error_de(3)

Chapter 12. Fortran Address Handling

Some of the DaCS C APIs include a parameter type of `void *`. In Fortran two APIs are provided to convert addresses to and from a `dacs_pvoid_t` handle for use as parameters in the DaCS Fortran APIs. These are:

- dacsf_makeptr: convert a `dacs_pvoid_t` handle to an address or Fortran pointer.
- dacsf_makevoid: convert a Fortran pointer or address to a `dacs_pvoid_t` handle.

dacsf_makeptr

NAME

dacsf_makeptr - Convert a `dacs_pvoid_t` handle to an address or Fortran pointer.

SYNOPSIS

C syntax

Fortran only.

Fortran syntax

pointer dacsf_makeptr (dacs_pvoid_t ***holder*****)**

Call parameter

`holder`	Handle returned from a DaCS API such as `dacsf_mem_query_addr`.

DESCRIPTION

The `dacsf_makeptr` service converts a `dacs_pvoid_t` handle to an address. Both 32 and 64 bit addressing is supported. To convert a `dacs_pvoid_t` into a Fortran pointer the program must call the function `dacsf_makeptr` after calling an API which returns a `dacs_pvoid_t` such as `dacsf_mem_query` to obtain the address.

Note: The C utility function prototype for `dacsf_makeptr()` is `void * dacsf_makeptr(int64_t *holder)`

RETURN VALUE

The `dacsf_makeptr` service returns a Fortran pointer.

SEE ALSO

dacs_mem_query(3), dacs_remote_mem_query(3)

dacsf_makevoid

NAME

dacsf_makevoid - Convert a Fortran pointer or address to a `dacs_pvoid_t` handle.

SYNOPSIS

C syntax

Fortran only.

Fortran syntax

dacsf_makevoid (pointer *in*, **dacs_pvoid_t** *out***)**

Call parameter	
`in`	a Fortran pointer or data object where the handle will be used as a parameter in a DaCS API such as `dacsf_put`

Return parameter	
`out`	Handle of type `dacs_pvoid_t`.

DESCRIPTION

The `dacsf_makevoid` service converts the address of a Fortran data area, or the contents of a Fortran pointer to a `dacs_pvoid_t` handle. Both 32 and 64 bit addressing is supported.

Note: The C utility function prototype for `dacs_makevoid()` is `void dacsf_makevoid(void *in, int64_t *out)`.

RETURN VALUE

None.

SEE ALSO

dacs_put(3), dacs_get(3), dacs_put_list(3), dacs_get_list(3), dacs_send(3), dacs_recv(3), dacs_mem_create(3), dacs_remote_mem_create(3), dacs_de_start(3)

Part 5. Appendixes

Appendix A. DaCS DE types

The current DaCS Element (DE) types in the current supported DaCS topology are listed below.

DACS_DE_SYSTEMX
: The x86_64 supervisor host for a node.

DACS_DE_CELL_BLADE
: An entire Cell/B.E. blade server. If a program is run on this DE, it has 16 SPE children, and the `DACS_DE_CBE` elements are not allowed to execute any processes. Some applications may find this configuration useful.

DACS_DE_CBE
: Cell/B.E.processor. A Cell/B.E. blade server contains two of these. If used this way, a Cell/B.E. has 8 SPE children. As with the `DACS_DE_CELL_BLADE`, if processes are running on a `DACS_DE_CBE` element, no processes are allowed on the parent `DACS_DE_CELL_BLADE`. Running processes on a Cell/B.E. node allows finer control of memory and processor affinity and may increase performance.

DACS_DE_SPE
: Cell/B.E. Synergistic Processing Element.

Appendix B. Performance and debug trace

The Performance Debugging Tool (PDT) provides trace data necessary to debug functional and performance problems for applications using the DaCS library.

Versions of the DaCS libraries built with PDT trace hooks enabled are delivered with DaCS 4.0.

Installing and running the PDT

The libraries with the trace hooks enabled are packaged in separate `-trace` named packages. The trace-enabled libraries install to a subdirectory named `dacs/trace` in the library install directory. These packages and the PDT are included in the SDK 3.1 package but may not be installed by default. Please refer to the PDT user's guide for full instructions on how to install PDT, and how to set the correct environment variables to cause trace events to be generated. Included in the DaCS trace package is an example PDT configuration file which shows the available trace events that can be enabled or disabled.

Trace control

In the Hybrid environment, PDT functions the same as it does in the single-system environment: When a PDT-enabled application starts, PDT reads its configuration from a file. For a distributed DaCS application you can distribute the PDT configuration with each job by specifying it as one of the DACS_START_FILES (see "dacs_de_start" on page 54). The PDT configuration for DaCS is separate from the configuration for your job.

Environment variable

PDT supports an environment variable (`PDT_CONFIG_FILE`) which allows you to specify the relative or full path to a configuration file. DaCS will ship an example configuration file which lists all of the DaCS groups and events and allows you to turn selected items on or off as desired. This will be shipped as:

```
/usr/share/pdt/config/pdt_dacs_config_cell.xml
/usr/share/pdt/config/pdt_dacs_config_hybrid.xml
```

To see DaCS PPU or Hybrid events the application must use the trace-enabled DaCS PPU/Hybrid code. If the application is using the static library then it must be re-linked with the trace-enabled library code (`/usr/lib64/dacs/trace/libdacs.a` or `/usr/lib/dacs/trace/libdacs.a` as appropriate for Cell/B.E. on the PPU, and `/usr/lib64/dacs/trace/libdacs_hybrid.a` or `/usr/lib/dacs/trace/libdacs_hybrid.a` as appropriate for DaCS for Hybrid). If the application was built using the shared DaCS library then no re-linking is needed. In that case the library path must be changed to point to the trace-enabled code as well as the PDT trace library, by setting the environment as appropriate before running the application:

If the application was built using the shared PPU library then no re-linking is needed. In that case the library path must be changed to point to the trace-enabled PPU code as well as the PDT trace library, by setting the environment as appropriate before running the application:

```
LD_LIBRARY_PATH=/usr/lib64/dacs/trace:/usr/lib64/trace # for 64-bit applications
LD_LIBRARY_PATH=/usr/lib/dacs/trace:/usr/lib/trace # for 32-bit applications
```

Appendix C. DaCS trace events

Where inputs or outputs are pointers to scalar types, both the pointer and the contents will be traced. To avoid any extra overhead of checking for NULL pointers, the trace code will only trace contents for pointers that are either required to be non-NULL by the API spec. or already have appropriate checks in the library. The contents of aggregate types will not be traced unless the entire object is passed in as an argument.

In general, there will be two trace hooks per API. The first will trace the input parameters and the second will trace the output values as well as the time interval of the API call. The performance hooks will generally have entry and exit hooks so the post-processing tools can show the time deltas. Note that the performance hooks are also debug hooks and will be enabled when either category is enabled.

DaCS API hooks

Table 8. Trace hooks enabled by `LIBDACS` group (0x04) in the config file.

Hook identifier	Traced values
_DACS_BARRIER_WAIT_ENTRY	group
_DACS_BARRIER_WAIT_EXIT_INTERVAL	retcode
_DACS_DE_KILL_ENTRY	deid, pid
_DACS_DE_KILL_EXIT_INTERVAL	retcode
_DACS_DE_START_ENTRY	deid, text, argv, envv, creation_flags, p_pid
_DACS_DE_START_EXIT_INTERVAL	retcode, pid
_DACS_DE_TEST_ENTRY	deid, pid, p_exit_status
_DACS_DE_TEST_EXIT_INTERVAL	retcode, exit_status
_DACS_DE_WAIT_ENTRY	deid, pid, p_exit_status
_DACS_DE_WAIT_EXIT_INTERVAL	retcode, exit_status
_DACS_GENERIC_DEBUG	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_DACS_GET_ENTRY	dst_addr, src, src_offset, size, wid, order_attr, swap
_DACS_GET_EXIT_INTERVAL	retcode
_DACS_GET_LIST_ENTRY	dst_addr, dst_dma_list, dst_list_size, src_remote_mem, src_dma_list, src_list_size, wid, order_attr, swap
_DACS_GET_LIST_EXIT_INTERVAL	retcode
_DACS_MBOX_READ_ENTRY	msg, src_de, src_pid
_DACS_MBOX_READ_EXIT_INTERVAL	retcode
_DACS_MBOX_TEST_ENTRY	rw_flag, deid, pid, p_mbox_status
_DACS_MBOX_TEST_EXIT_INTERVAL	retcode, result
_DACS_MBOX_WRITE_ENTRY	msg, dst_de, dst_pid
_DACS_MBOX_WRITE_EXIT_INTERVAL	retcode
_DACS_MUTEX_ACCEPT_ENTRY	deid, pid, mutex
_DACS_MUTEX_ACCEPT_EXIT_INTERVAL	retcode

Table 8. Trace hooks enabled by `LIBDACS` group (0x04) in the config file. (continued)

Hook identifier	**Traced values**
_DACS_MUTEX_DESTROY_ENTRY	mutex
_DACS_MUTEX_DESTROY_EXIT_INTERVAL	retcode
_DACS_MUTEX_INIT_ENTRY	mutex
_DACS_MUTEX_INIT_EXIT_INTERVAL	retcode
_DACS_MUTEX_LOCK_ENTRY	mutex
_DACS_MUTEX_LOCK_EXIT_INTERVAL	retcode
_DACS_MUTEX_RELEASE_ENTRY	mutex
_DACS_MUTEX_RELEASE_EXIT_INTERVAL	retcode
_DACS_MUTEX_SHARE_ENTRY	deid, pid, mutex
_DACS_MUTEX_SHARE_EXIT_INTERVAL	retcode
_DACS_MUTEX_TRY_LOCK_ENTRY	mutex
_DACS_MUTEX_TRY_LOCK_EXIT_INTERVAL	retcode
_DACS_MUTEX_UNLOCK_ENTRY	mutex
_DACS_MUTEX_UNLOCK_EXIT_INTERVAL	retcode
_DACS_PUT_ENTRY	dst, dst_offset, src_addr, size, wid, order_attr, swap
_DACS_PUT_EXIT_INTERVAL	retcode
_DACS_PUT_LIST_ENTRY	dst, dst_dma_list, dma_list_size, src_addr, src_dma_list, src_list_size, wid, order_attr, swap
_DACS_PUT_LIST_EXIT_INTERVAL	retcode
_DACS_RMEM_ACCEPT_ENTRY	src_de, src_pid, remote_mem
_DACS_RMEM_ACCEPT_EXIT_INTERVAL	retcode
_DACS_RMEM_CREATE_ENTRY	addr, size, mode, local_mem
_DACS_RMEM_CREATE_EXIT_INTERVAL	retcode
_DACS_RMEM_DESTROY_ENTRY	remote_mem
_DACS_RMEM_DESTROY_EXIT_INTERVAL	retcode
_DACS_RMEM_RELEASE_ENTRY	remote_mem
_DACS_RMEM_RELEASE_EXIT_INTERVAL	retcode
_DACS_RMEM_SHARE_ENTRY	dst, dst_pid, local_mem
_DACS_RMEM_SHARE_EXIT_INTERVAL	retcode
_DACS_INIT_ENTRY	config_flags
_DACS_INIT_EXIT_INTERVAL	retcode
_DACS_EXIT_ENTRY	zero
_DACS_EXIT_EXIT_INTERVAL	retcode
_DACS_SEND_ENTRY	src_data, len, dst_de, dst_pid, stream, wid, swap
_DACS_SEND_EXIT_INTERVAL	retcode
_DACS_RECV_ENTRY	dst_data, len, src_de, src_pid, stream, wid, swap
_DACS_RECV_EXIT_INTERVAL	retcode

Table 9. Trace hooks enabled by the `LIBDACS2` *group (0x0A) in the config file:*

Hook identifier	**Traced values**
_DACS_GROUP_INIT	p_group, flags, group, retcode
_DACS_GROUP_ADD_MEMBER_ENTRY	deid, pid, group
_DACS_GROUP_ADD_MEMBER_EXIT_INTERVAL	retcode
_DACS_GROUP_CLOSE_ENTRY	group
_DACS_GROUP_CLOSE_EXIT_INTERVAL	retcode
_DACS_GROUP_DESTROY_ENTRY	p_group, group
_DACS_GROUP_DESTROY_EXIT_INTERVAL	retcode,
_DACS_GROUP_ACCEPT_ENTRY	deid, pid, p_group
_DACS_GROUP_ACCEPT_EXIT_INTERVAL	retcode, group
_DACS_GROUP_LEAVE_ENTRY	p_group, group
_DACS_GROUP_LEAVE_EXIT_INTERVAL	retcode, group
_DACS_WID_RESERVE_ENTRY	p_wid
_DACS_WID_RESERVE_EXIT_INTERVAL	retcode, wid
_DACS_WID_RELEASE_ENTRY	p_wid, wid
_DACS_WID_RELEASE_EXIT_INTERVAL	retcode, wid
_DACS_TEST_ENTRY	wid
_DACS_TEST_EXIT_INTERVAL	retcode
_DACS_WAIT_ENTRY	wid
_DACS_WAIT_EXIT_INTERVAL	retcode
_DACS_GET_NUM_AVAIL_CHILDREN	type, p_num_children, num_children, retcode
_DACS_RESERVE_CHILDREN_ENTRY	type, p_num_children, num_children, p_de_list
_DACS_RESERVE_CHILDREN_EXIT_INTERVAL	retcode, num_children, child[1..18]
_DACS_RELEASE_DE_LIST_ENTRY	num_des, p_num_de_list, child[1..16]
_DACS_RELEASE_DE_LIST_EXIT_INTERVAL	retcode, num_children, child[1..16]
_DACS_MEM_CREATE_ENTRY	addr, size, rmt_access_mode, lcl_acces_mode, p_mem
_DACS_MEM_CREATE_EXIT_INTERVAL	retcode, mem
_DACS_MEM_SHARE_ENTRY	dst_de, dst_pid, local_mem
_DACS_MEM_SHARE_EXIT_INTERVAL	retcode
_DACS_MEM_ACCEPT_ENTRY	src_de, src_pid, p_mem
_DACS_MEM_ACCEPT_EXIT_INTERVAL	retcode, mem
_DACS_MEM_DESTROY_ENTRY	remote_mem
_DACS_MEM_DESTROY_EXIT_INTERVAL	retcode
_DACS_MEM_RELEASE_ENTRY	remote_mem
_DACS_MEM_RELEASE_EXIT_INTERVAL	retcode
_DACS_MEM_PUT_ENTRY	dst, dst_offset, src, src_offset, size, wid, order_attr, swap
_DACS_MEM_PUT_EXIT_INTERVAL	retcode
_DACS_MEM_GET_ENTRY	dst, dst_offset, src, src_offset, size, wid, order_attr, swap
_DACS_MEM_GET_EXIT_INTERVAL	retcode
_DACS_MEM_PUT_LIST_ENTRY	dst, dst_dma_list, dst_list_size, src, src_dma_list, src_list_size, wid, order, attr, swap

Table 9. Trace hooks enabled by the `LIBDACS2` group (0x0A) in the config file: (continued)

Hook identifier	Traced values
_DACS_MEM_PUT_LIST_EXIT_INTERVAL	retcode
_DACS_MEM_GET_LIST_ENTRY	dst, dst_dma_list, dst_list_size, src, src_dma_list, src_list_size, wid, order, attr, swap
_DACS_MEM_GET_LIST_EXIT_INTERVAL	retcode
_DACS_MEM_REGISTER_ENTRY	dst_de, dst_pid, local_mem
_DACS_MEM_REGISTER_EXIT_INTERVAL	retcode
_DACS_MEM_DEREGISTER_ENTRY	dst_de, dst_pid, local_mem
_DACS_MEM_DEREGISTER_EXIT_INTERVAL	retcode

DaCS performance hooks

The `COUNTERS` and `TIMERS` hooks contain data that are accumulated during the DaCS calls. These data and trace events are reported by the `dacs_exit()` function.

Table 10. Trace hooks enabled by `LIBDACS_GROUP` group (0x06) in the config file.

Hook identifier	Traced values
_DACS_COUNTERS1	dacs_de_starts, dacs_de_waits, dacs_put_count, dacs_get_count, dacs_put_bytes, dacs_get_bytes, dacs_send_count, dacs_recv_count, dacs_send_bytes, dacs_recv_bytes
_DACS_COUNTERS2	dacs_mutex_try_success, dacs_mutex_try_failure, dacs_x1, dacs_x2
_DACS_HOST_MUTEX_INIT	lock
_DACS_HOST_MUTEX_LOCK	lock, miss
_DACS_HOST_MUTEX_TRYLOCK	lock, ret
_DACS_HOST_MUTEX_UNLOCK	lock
_DACS_PERF_GENERIC_DEBUG	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_DACS_SPE_MUTEX_INIT	lock
_DACS_SPE_MUTEX_LOCK	lock, miss
_DACS_SPE_MUTEX_TRYLOCK	lock, ret
_DACS_SPE_MUTEX_UNLOCK	lock
_DACS_TIMERS	dacs_put, dacs_put_list, dacs_wait, dacs_send, dacs_recv, dacs_mutex_lock, dacs_barrier_wait, dacs_mbox_read, dacs_mbox_write, dacs_x
_DACS_PPE_UNALIGNED_DMA	op, ls, ea, len
_DACS_SPE_UNALIGNED_DMA	op, ls, ea, len
_DACS_INTERNAL_BARRIER_WAIT	
_DACS_INTERNAL_MESSAGE_WAIT	
_DACS_INTERNAL_MUTEX_WAIT	

Appendix D. Error codes

This section describes the DaCS error codes

All error codes which may be issued by DaCS APIs are listed here:

DACS_ERR_ARCH_MISMATCH: attempted to use a 64-bit accelerator application with a
    32-bit host application or vice-versa
DACS_ERR_BUF_OVERFLOW: buffer overflow
    - the specified offset or size exceed the bounds of the target buffer.
DACS_ERR_BYTESWAP_MISMATCH: the byte swap flags on the source and target
    do not match.
DACS_ERR_DACSD_FAILURE: unable to communicate with the DaCS daemon services.
DACS_ERR_DE_TERM: the de_started process has terminated before calling dacs_init().
DACS_ERR_GROUP_CLOSED: the group is closed.
DACS_ERR_GROUP_DUPLICATE: the specified process is already a member of the
    specified group.
DACS_ERR_GROUP_OPEN: the group has not been closed.
DACS_ERR_INITIALIZED: DaCS is already initialized.
DACS_ERR_INVALID_ARGV: the value of argv is too large or invalid.
DACS_ERR_INVALID_ADDR: the pointer is invalid.
DACS_ERR_INVALID_ATTR: the flag or enumerated constant is invalid.
DACS_ERR_INVALID_CWD: an error occurred accessing the current working directory on the accelerator

DACS_ERR_INVALID_DE: the specified DE is either invalid or not reserved.
DACS_ERR_INVALID_ENV: the value of env is too large or invalid.
DACS_ERR_INVALID_HANDLE: the handle is invalid.
DACS_ERR_INVALID_PID: the specified PID does not refer to a valid process.
DACS_ERR_INVALID_PROG: unable to execute the specified program.
DACS_ERR_INVALID_SIZE: the size is zero or is not supported by the platform.
DACS_ERR_INVALID_STREAM: the stream identifier is invalid.
DACS_ERR_INVALID_TARGET: this operation is not allowed for the target DE or process.
DACS_ERR_INVALID_USERNAME: current userid is not configured.
Typically this occurs if the userid is not setup on the accelerator but it is set up on the host
DACS_ERR_INVALID_WID: the wait identifier is invalid.
DACS_ERR_MUTEX_BUSY: the mutex is not available.
DACS_ERR_NO_PERM: the process does not have the appropriate privilege
    or the resource attributes do not allow the operation.
DACS_ERR_NO_RESOURCE: unable to allocate required resources.
DACS_ERR_NO_WIDS: no more wait identifiers are available to be reserved.
DACS_ERR_NOT_ALIGNED: an alignment conflict exists between the swap flag
granularity and the address, offset, or size.
DACS_ERR_NOT_INITIALIZED: DaCS has not been initialized.
DACS_ERR_NOT_FOUND: there is an error in the configuration file and some aspect
 of the hardware information is missing.
DACS_ERR_NOT_OWNER: this operation is only permitted for the owner of the resource.
DACS_ERR_NOT_SUPPORTED_YET: the DaCS function is currently unsupported by this
    platform.
DACS_ERR_OWNER: attempt to release a resource owned by the process. Resource must be
    destroyed.
DACS_ERR_PROC_LIMIT: the maximum number of processes supported has been reached.

DACS_ERR_RESOURCE_BUSY: the specified resource is in use.
DACS_ERR_SYSTEM: an error occurred interacting with the operating system
DACS_ERR_TOO_LONG: the executable name specified in de_start is too long
DACS_ERR_VERSION_MISMATCH: version mismatch between two parts of DaCS. This could be a mismatch
between the library and the DaCS daemon services or between the host and accelerator libraries.
DACS_ERR_WID_ACTIVE: data transfer involving the wait identifier is still active.
DACS_ERR_WID_NOT_ACTIVE: there are no outstanding transfers to test.
DACS_STS_PROC_ABORTED: the process terminated abnormally.
DACS_STS_PROC_FAILED: the process exited with a failure.
DACS_STS_PROC_FINISHED: the process finished execution without error.
DACS_STS_PROC_KILLED: the process was killed by a call to dacs_de_kill

DACS_STS_PROC_RUNNING: the process is still running.
DACS_SUCCESS: the API returned successfully.
DACS_WID_READY: all data transfers have completed.
DACS_WID_BUSY: one or more data transfers have not completed.

Appendix E. Environment variables

The environment variables supported by the DaCS APIs are summarized in the tables below:

Table 11. DaCS for Cell Environment Variables

Environment variable	Description
DACS_SPE_EXCEPTION_HALT	Halt SPU execution on exception. This variable is used for debugging SPU exceptions within a DaCS application. Typically, DaCS will handle SPU exceptions by collecting the pertinent error information along with reaping the SPU thread. There are times where the SPU application programmer may wish for the SPU application to halt at the exception point for easier diagnosis and debug. Setting this environment variable will cause the SPU application to halt, on exception, at the offending location for debugging purposes.

Table 12. DaCS for Hybrid Environment Variables

Environment variable	Description
DACS_PARENT_PORT	Specifies the port value to pass when starting the AE process. See "dacs_de_start" on page 54 for more information.
DACS_START_ENV_LIST	Specified an additional list of environment variables for the initial program spawn on the accelerator. See "dacs_de_start" on page 54 for more information.
DACS_START_FILES	Specifies the name of a file which contains a list of files to transfer to the AE prior to launching the AE process. See "dacs_de_start" on page 54 for more information.
DACS_START_PARENT	Specifies the command used to start an auxiliary program which starts the AE process. See "dacs_de_start" on page 54 for more information.
DACS_HYBRID_DEBUG	Sets the level of debug messages to generate for DaCS for Hybrid. When set: • The daemon sends additional details about the daemon operation to the daemon log • The optimized and debug libraries send additional details about library operations to the log • The trace libraries send details about the library operation to PDT For more details see Appendix F, "DaCS for Hybrid debugging," on page 197.
DACS_HYBRID_KEEP_CWD	When the `dacsd.conf` file specifies that the current working directory should not be saved (default), this environment variable can be set to `Y` to indicate that the behavior should be overridden temporarily. For more details see Chapter 5, "Configuring DaCS for Hybrid," on page 15.

Table 12. DaCS for Hybrid Environment Variables (continued)

Environment variable	Description
DACS_HYBRID_USE_FABRIC_TYPE	More than one fabric type can be specified in the `dacs_topology.config` file. The fabric that is used by default is set using the default attribute of the fabrics element. For debugging, this environment variable can be set to indicate a temporary value for the fabric, overriding the default in the configuration file. **Note:** This variable must be set to one of the values specified in the `dacs_topology.config` file as a type from one of the fabric elements.For more details see Chapter 5, "Configuring DaCS for Hybrid," on page 15.

Appendix F. DaCS for Hybrid debugging

DaCS for Hybrid provides message redirection, logging and tools to aid with debugging.

Message redirection

With the redirection of stdout and stderr from the AEs to their associated HE, the output becomes intermingled. To improve the ability to debug code, it is recommended that messages should either be unique across all parts of the application or should identify where they are coming from.

In addition, while output from the same source (in other words the same AE process) is in the order written, output from different sources may not be in the correct order relative to each other. For example if the HE prints out a message, intermingled AE messages occurring after it in the output may have actually occurred before it in time. To enable synchronization of output, it is recommended that output be produced on both the HE and AE at points where they synchronize (such as a share and accept).

Logging

In some cases DaCS for Hybrid will log information about the daemon or application. Log entries are created with different severities:

Emergency = 0

Alert = 1

Critical = 2

Error = 3

Warning = 4

Notice = 5

Information = 6

Debug = 7

If the events being logged are severe enough (Emergency, Alert, or Critical), they are also logged to the syslog (when the syslog daemon is running).

Daemon Logs

The daemon logs can contain information about the heartbeat between the daemons, messages flowing between processes, process start and end, and topology information.The level and amount of information logged can be controlled by using the `dacs-hdacsd-loglevel` tool (prototype). Calling this tool with the parameter LOG_DEBUG will cause all messages at all levels to be logged. Calling it with LOG_NOTICE will log only messages at the Notice level or higher. The environment variable `DACS_HYBRID_DEBUG` can also be used to control the level of messages. Setting it equal to Y does all messages at all levels, setting it to a number logs messages at that level (see above) or more severe.

The log is put into `/var/log/hdacsd.log` or `/var/log/adacsd.log` for the hdacsd or adacsd respectively.

The `dacs-hdacsd-diag` (prototype) tool can be used to retrieve information about the daemons, including the logs. Note this is only available on the HE, but it does, if requested, retrieve logs from the AEs as well.

Application Logs

Application Logs Critical or severe problems will be logged for the application. If a trace library is being used, these logs are redirected to PDT. Otherwise the logged information will go into `/tmp/dlog_<`*`pid`*`>.log`, where pid is the process id. The PID information is available as part of the daemon log from when the process was started.

In addition to critical messages, if the `dacs-hdacsd-loglevel` tool (prototype) or `DACS_HYBRID_DEBUG` environment variable are used (described above), then messages about areas where DaCS for Hybrid performance could be improved by application changes are logged. For example, if memory regions are exhausted and `dacs_put` is used, DaCS for Hybrid will not be able to get the temporary memory region needed to perform the put and may either have to delay the put or redirect it to a different data path with less performance.

DaCS diagnostic tools

A number of tools to aid in diagnosis of DaCS for Hybrid are provided. These tools are prototype tools and are not supported. The tools are installed in the `/opt/cell/sdk/prototype/usr/bin` directory.

The current DaCS Diagnostic Tools for Hybrid are listed below.

dacs-hdacsd-diag

this tool gathers information about the DaCS for Hybrid daemons (`hdacsd and adacsd`) that are running, their versions, configuration and current status. It also supports gathering of log files from both the hdacsd and adacsd. Results are returned as an xml file.

dacs-hdacsd-loglevel

this tool allows the logging level of the daemon to be changed between `LOG_NOTICE` (log only significant messages) and `LOG_DEBUG` (log all messages)

dacs-diag

this tool gathers diagnostic information and checks for common problems such as the library not being installed, the daemon not running, basic `dacs_topology.config` problems such as invalid ip addresses, etc.

dacs-hdacsd-check-availability

this tool queries the status of the daemons to ensure that they are operating and communicating.

dacs-query

this tool queries how many accelerator elements are available and of what type for the host element the tool is run on

Appendix G. DaCS Fortran bindings

The Fortran bindings allow Fortran programs to invoke the DaCS APIs. These bindings follow the DaCS APIs as closely as possible, however some differences exist. The bindings are written to the Fortran 90 standard.

Overview

The DaCS Fortran bindings consist of the following items:
- dacsf.h: Fortran include file which defines constants and data types
- dacsf_interface.h: Fortran include file which defines the Fortran interfaces for the DaCS APIs
- Entry points in each of the DaCS libraries for the Fortran bindings
- DaCS Fortran binding examples in the DaCS examples RPMs

The interfaces in Fortran to the DaCS APIs are subroutine calls. Arguments are passed in by reference, and can be modified by the API. This follows the DaCS API pattern of modifying parameters. The Fortran Interface definitions identify which parameters are modified with the Intent keyword. Many interfaces for DaCS are simple, and consist of integers as call and return parameters, and return an integer return code. Some exceptions to this exist however, in particular the handling of strings and pointers.

The DaCS APIs require that space be allocated by the caller for returned string and array parameter values from subroutine calls. The caller in Fortran must ensure that enough space is allocated. Constants are defined to determine the maximum space required.

For a complete listing of the compilers that have been tested with the Fortran binding, see "Supported compilers for the DaCS Fortran bindings" on page 9.

Programming with the Fortran bindings

The DaCS Fortran bindings are provided as subroutines and functions in the Fortran language. The example below illustrates the conventions:

```
!compile command:
!ppu-gfortran  -I /usr/include -o simple simple.f90 /usr/lib64/libdacs.a -lspe2
program simple
implicit none

include 'dacsf.h'
include 'dacsf_interface.h'

integer(kind=dacs_err_t) :: rc
integer(kind=dacs_int32_t) :: num_processes
character (len= DACS_MAX_ERRSTR_LEN) :: return_msg

call dacsf_init(DACS_INIT_FLAGS_NONE,rc)
call dacsf_strerror(rc,return_msg)
write (*,*) 'dacsf_init[',trim(return_msg),']'

call dacsf_num_processes_supported(DACS_DE_SELF, num_processes,rc)
call dacsf_strerror(rc,return_msg)
write (*,*) 'dacsf_num_processes_supported[',trim(return_msg),']'
write (*,*) 'num_processes[',num_processes,']'
```

```
call dacsf_exit(rc)
write (*,*) 'dacsf_exit[',trim(return_msg),']'
end program
```

Include files

Two include files are provided as part of the DaCS Fortran bindings. The file `dacsf.h` contains the DaCS constants and data types definitions used in the DaCS subroutines and functions. The file `dacsf_interface.h` contains the subroutine and function interface definitions. The file `dacsf_interface.h`, while not required to be included in the program, provides compile time validation of the subroutine calls in the application when it is included. These include files can be used as documentation for the subroutine calls, data types, and constants.

Subroutines and functions

The majority of the DaCS APIs are implemented as Fortran subroutines, due to the number of in/out parameters. The error handler procedure which can be registered with the dacsf_errhandler_reg API, and the utility routine dacsf_makeptr, are functions.

Return codes

All DaCS APIs return the return code as the last parameter except dacsf_strerror, and dacsf_error_num. The return codes are defined in dacsf.h and match the C return codes defined in dacs.h.

Differences in the Fortran Bindings From the DaCS C APIs

The sections below describe the differences between the Fortran implementation and the C implementation of some APIs.

dacs_de_start

The dacs_de_start API is implemented in Fortran as an Interface Block with the name dacs_de_start. To handle the parameter overloading in the C API, four Fortran subroutines are defined

- dacsf_de_start_std_file
- dacsf_de_start_std_embedded
- dacsf_de_start_ptr_file
- dacsf_de_start_ptr_embedded

dacsf_de_start_std_*
: Provides the standard main `argv` and `envv` parameters. These subroutines are used with DaCS for Hybrid when starting an accelerator process on the PPE from the x86_64 system host element. Only the `dacsf_de_start_std_file` subroutine is supported by DaCS for Hybrid.

dacsf_de_start_ptr_*
: Provides the SPU main `argv` and `envv` parameters. These subroutines are used with DaCS for Cell when starting an accelerator process on the SPE from the PPE system host element. Both the `dacsf_de_start_ptr_file`, `dacsf_de_start_ptr_embedded` subroutines are supported by DaCS for Cell. The SPU main C Function prototype is `extern int main (unsigned long long spuid, unsigned long long argp, unsigned long long envp)`.

dacsf_de_start_*_file
: Provide the filename or file list that identifies the executable to be initiated.

dacsf_de_start_*_embedded
: Provide an external reference to the program to start as it is embedded in the executable.

The table below summarizes the parameters used in the subroutine definitions for `dacsf_de_start`.

Table 13.

Parameter name	dacsf_de_start_std_file	dacsf_de_start_ptr_file	dacsf_de_start_ptr_embedded
de	dacs_de_id_t	dacs_de_id_t	dacs_de_id_t
prog	string	string	external
argv	array of strings (if empty, argv[0] is still passed to the AE)	dacs_pvoid_t	dacs_pvoid_t
argv_size	argv element count	na	na
envv	array of strings (if empty, the variables in `DACS_START_ENV_LIST` are still passed to the AE)	dacs_pvoid_t	dacs_pvoid_t
envv_size	envv element count	na	na
creation flags	dacs_proc_creation_flags_t	dacs_proc_creation_flags_t	na
pid	dacs_process_id_t	dacs_process_id_t	dacs_process_id_t
rc	dacs_err_t	dacs_err_t	dacs_err_t4

Note: na=not available.
The most commonly used subroutines are:
- PPU to SPU: dacsf_de_start_ptr_embedded
- Opteron to PPU: dacsf_de_start_std_file

dacs_remote_mem_query and dacs_mem_query

The C APIs `dacs_remote_mem_query` and `dacs_mem_query` have multiple Fortran subroutines to match the data types returned in the value parameter.

Table 14. dacs_remote_mem_query

parameter name	dacsf_remote_mem_query_mode	dacsf_remote_mem_query_addr	dacsf_remote_mem_query_size
mem	dacs_remote_mem_t	dacs_remote_mem_t	dacs_remote_mem_t
mode/mem_size/ addr	dacs_memory_access_mode_t	dacs_pvoid_t	dacs_int64_t
rc	dacs_err_t	dacs_err_t	dacs_err_t

Table 15. dacs_mem_query

parameter name	dacsf_mem_query_lcl_perm	dacsf_mem_query_rmt_perm	dacsf_mem_query_addr	dacsf_mem_query_size
mem	dacs_mem_t	dacs_mem_t	dacs_mem_t	dacs_mem_t
mode/mem_size/ addr	dacs_mem_access_mode_t	dacs_mem_access_mode_t	dacs_pvoid_t	dacs_int64_t
rc	dacs_err_t	dacs_err_t	dacs_err_t	dacs_err_t

dacsf_makevoid and dacsf_makeptr

Some of the DaCS C APIs include a parameter type of void *. In Fortran two utility procedures are provided to convert addresses to and from a `dacs_pvoid_t` handle for use as parameters in the DaCS Fortran bindings.

Subroutine `dacsf_makevoid`:

`dacsf_makevoid` is not defined in `dacsf_interface.h`. This function converts a void * reference to an `dacs_pvoid_t` handle. Both 32 and 64 bit addressing is supported. The closest corresponding data type to void * in Fortran is an `integer(kind=8)` or `dacs_int64_t`. To pass a C void * parameter in a DaCS subroutine the Fortran program must first call the `dacsf_makevoid` subroutine to obtain the 64 bit handle for the void * address. This handle is passed in the DaCS subroutine and converted back to a void *pointer in the binding implementation.

The C utility function prototype for `dacsf_makevoid` is `void dacsf_makevoid(void *in, int64_t *out)`

See "dacsf_makevoid" on page 181 for further information.

Function `dacsf_makeptr`:

`dacsf_makeptr` is not defined in `dacsf_interface.h`. This function converts a `dacs_pvoid_t` handle to an address. Both 32 and 64 bit addressing is supported. To convert a `dacs_pvoid_t` into a Fortran pointer the program must call the function `dacsf_makeptr` after calling an API which returns a `dacs_pvoid_t`, such as `dacs_mem_query_addr` to obtain the address.

The C utility function prototype for `dacsf_makeptr` is `void * dacsf_makeptr(int64_t *holder)`

See "dacsf_makeptr" on page 180 for further information.

The APIs listed in the following table include a `dacs_pvoid_t` parameter and require the use of `dacsf_makevoid` or `dacsf_makeptr`:

Table 16.

API	Parameter	Function	Parameter Description
dacsf_remote_mem_create	addr	dacsf_makevoid	memory region to be shared
dacsf_remote_mem_query_addr	addr	dacsf_makeptr	memory region address
dacsf_put	src_addr	dacsf_makevoid	source memory buffer
dacsf_get	dst_addr	dacsf_makevoid	destination memory buffer
dacsf_put_list	src_addr	dacsf_makevoid	source memory buffer
dacsf_get_list	dst_addr	dacsf_makevoid	destination memory buffer
dacsf_recv	dst_data	dacsf_makevoid	destination message buffer
dacsf_send	src_data	dacsf_makevoid	source message buffer
dacsf_mem_create	addr	dacsf_makevoid	address to create memory region over
dacsf_mem_query_addr	addr	dacsf_makeptr	address for the memory region
dacsf_de_start_ptr_file	argv,envv	dacsf_makevoid	long long
dacsf_de_start_ptr_embedded	argv,envv	dacsf_makevoid	long long

See the DaCS Fortran examples for uses of `dacsf_makevoid` and `dacsf_makeptr`.

Data types

Signed and Unsigned Integers

In Fortran all integers are signed. Where the C APIs specify unsigned integers these are converted to signed integers. In the DaCS Debug libraries limit checking is done for the following subroutines. If the value is less than 0 as a signed integer then `DACS_ERR_INVALID_SIZE` is returned.

API	Parameter	Intent	Value Checked	Return Code
dacsf_recv	len	in	<0	DACS_ERR_INVALID_SIZE
dacsf_send	len	in	<0	DACS_ERR_INVALID_SIZE
dacsf_num_processes_supported	num_processes	out	<0	DACS_ERR_INVALID_SIZE
dacsf_num_processes_running	num_processes	out	<0	DACS_ERR_INVALID_SIZE
dacsf_mem_create	mem_size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_mem_query_size	mem_size	out	<0	DACS_ERR_INVALID_SIZE
dacsf_mem_put	mem_size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_mem_get	mem_size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_mem_put_list	dst_list[].size src_list[].size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_mem_put_list	dst_list[].size src_list[].size	in	<0	DACS_ERR_INVALID_SIZE

dacsf_remote_mem_create	mem_size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_remote_mem_query_size	mem_size	out	<0	DACS_ERR_INVALID_SIZE
dacsf_put	mem_size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_get	mem_size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_put_list	dst_list[].size src_list[].size	in	<0	DACS_ERR_INVALID_SIZE
dacsf_put_list	dst_list[].size src_list[].size	in	<0	DACS_ERR_INVALID_SIZE

Strings and Arrays of strings

The Fortran interoperability calling convention for passing strings as arguments is to automatically append the length of the string area in the interface as an additional parameter. This length is the allocated length and will include any default blank padding.

The `dacsf_de_start` API is the only API which has strings as in parameters. It will accept strings in this format for the prog, argv, and envv parameters. The binding code removes trailing (but not embedded) blanks.

When using the `dacsf_de_start_std_*` subroutines the argv and envv parameters are arrays of strings. The argv_size and envv_size specify the number of entries in the array. A null element following the last string element is NOT required for argv and envv parameters. If no elements are to be passed use the constant `DACS_NULL_STRINGLIST` to specify a zero element array.

The `dacsf_strerror` and `dacsf_error_str` APIs contain an error message character strings as an out parameter. Use the constant `DACS_MAX_ERRSTR_LEN` to allocate enough space for the error string to be returned, for example

```
character (len= DACS_MAX_ERRSTR_LEN) :: return_msg
```

The string is truncated if the length in the subroutine call is less than the length of the string.

Arrays

The subroutines `dacsf_reserve_children` and `dacsf_release_de_list` include an array of `dacs_de_id_t` as a parameter. Be sure to allocate enough elements in the array for the values. Constants are provided in `dacsf.h` to define the maximum number of SPEs available.The parameters must be allocated as arrays even if they contain only a single element.

The subroutines `dacsf_put_list`, `dacsf_get_list`, `dacsf_mem_put_list` and `dacsf_mem_get_list` include arrays of the type `dacs_dma_list_t`. These parameters must be defined as arrays even if they contain only a single element.

Binding alias names

Fortran compilers provide options to generate names interoperable with C programs in multiple variations using options flags. Using aliases the DaCS Fortran bindings support binding names with no underscore, one underscore, two underscores, and an uppercase version of the name with no underscores. Applications converted to use the DaCS Fortran bindings will not be required to conform to a single interoperability convention.

For each compiler the table below shows the compiler flags required to generate each alias. NA indicates that the compiler will not generate the specified alias.

Table 17. Compiler flags

Compiler	One underscore	No underscores	Two underscores	Upper case (no underscore)
GNU gfortran	default	-fno-underscoring	-fsecond-underscore	NA
IBM® XLF	-qextname	-qnoextname (default)	NA	-qmixed -qnoextname
Pathscale	default	-fno-underscoring	-fsecond-underscore	NA
PGI	default	NA	-Msecond-underscore	NA
Intel®	default	-assume nounderscore	-assume 2underscores	-assume nounderscore

dacsf_de_start examples

These are code fragments which demonstrate how to code dacsf_de_start subroutines. These code fragments will not compile and execute by themselves. See the DaCS Fortran examples for sample code which will compile and run.

Variable Definitions:

```
integer(kind=dacs_de_id_t)              :: de
character(len=*), parameter             :: file = "/home/user1/fortrantest/myprogram"
external                                :: hello_world
character(len=10), dimension(1)         :: argv = (/"one"/)
character(len=10), dimension(2)         :: envv = (/"a", "b"/)
integer(kind=dacs_pvoid_t)              :: spu_argv
integer(kind=dacs_pvoid_t)              :: spu_envv
integer(kind=dacs_proc_creation_flag_t) :: creation_flags
integer(kind=dacs_process_id_t)         :: pid
integer(kind=dacs_err_t)                :: rc
```

Subroutine Calls:

This call demonstrates using the generic interface name to invoke the subroutine `data_de_start_std_file`. This call is made from the Opteron to PPU because it passes standard main argv and envv parameters and does not use the embedded creation flag. The creation flags are set to specify a local file name is passed. See the dacs_de_start API section for a discussion of the creation flags.

```
call dacsf_de_start(de,file,argv,size(argv),envv,size(envv),creation_flags,pid,rc)
```

This is the same call as above using the subroutine name:

```
call dacsf_de_start_std_file(de,file,argv,size(argv),envv,
size(envv),creation_flags,pid,rc)
```

This is the same call as above except no data is passed for argv and envv

```
call dacsf_de_start_std_file(de,file,DACS_NULL_STRINGLIST,0,DACS_NULL_STRINGLIST,0,
creation_flags,pid,rc)
```

This call demonstrates the embedded version DACS_PROC_EMBEDDED. The creation flag parameter is not required as it can be determined from the parameters. This call is made from the PPU to the SPU because it passes the SPU main arguments and it uses the embedded creation flag:

```
call dacsf_de_start_ptr_embedded(de,hello_world,spu_argv,spu_envv,pid,rc)
```

DaCS Fortran binding examples

The DaCS Fortran Binding Example code can be found in the dacs-examples-source*.rpm and the dacs-hybrid-examples-source*.rpm.

These examples include four separate programs which demonstrate the use of Fortran, DaCS and the Fortran bindings for the DaCS API in both the cell and hybrid architectures.

The first three programs are designed to run on DaCS Cell and are included in the DaCS Samples. The fourth is designed to run on DaCS Hybrid and is included with the DaCS Hybrid samples.

- spe2_c: Cell programs which use libspe2 C APIs between PPU and SPU
- dacs_c: Cell programs which use DaCS C APIs between PPU and SPU
- dacs_f: Cell programs which use DaCS Fortran APIs between PPU and SPU
- dacs_f Hybrid: Hybrid and Cell programs which use DaCS Fortran APIs between Opteron, PPU, and SPU

DaCS APIs not supported on the SPU

The following APIs are not supported on the SPU. Attempts to build SPU code with these APIs will result in a link error.

```
dacsf_de_kill
dacsf_de_start_ptr_embedded
dacsf_de_start_ptr_file
dacsf_de_start_std_embedded
dacsf_de_start_std_file
dacsf_de_test
dacsf_de_wait
dacsf_group_add_member
dacsf_group_close
dacsf_group_destroy
dacsf_group_init
dacsf_mem_share
dacsf_mutex_destroy
dacsf_mutex_init
dacsf_mutex_share
dacsf_num_processes_running
dacsf_num_processes_supported
dacsf_release_de_list
dacsf_remote_mem_create
dacsf_remote_mem_destroy
dacsf_remote_mem_share
```

Appendix H. Accessibility features

IBM and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able/ for more information about the commitment that IBM has to accessibility.

Notices

This information was developed for products and services offered in the U.S.A.

The manufacturer may not offer the products, services, or features discussed in this document in other countries. Consult the manufacturer's representative for information on the products and services currently available in your area. Any reference to the manufacturer's product, program, or service is not intended to state or imply that only that product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any intellectual property right of the manufacturer may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any product, program, or service.

The manufacturer may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the manufacturer.

For license inquiries regarding double-byte (DBCS) information, contact the Intellectual Property Department in your country or send inquiries, in writing, to the manufacturer.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: THIS INFORMATION IS PROVIDED "AS IS " WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. The manufacturer may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to Web sites not owned by the manufacturer are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this product and use of those Web sites is at your own risk.

The manufacturer may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the manufacturer.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

Terms and conditions

Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the IBM developerWorks® Web site located at:

`http://www.ibm.com/developerworks/power/cell/`

Click the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*

Programming

- *Cell Broadband Engine Programmer's Guide*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Programming Tutorial*

Library

- *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference*
- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Fast Fourier Transform Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*
- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Runtime Extensions Library Programmer's Guide and API Reference*

- *Three dimensional FFT Prototype Library Programmer's Guide and API Reference*

Installation

- *SDK for Multicore Acceleration Version 3.1 Installation Guide*

Tools

- *Getting Started - XL C/C++ for Multicore Acceleration for Linux*
- *Compiler Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Language Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Programming Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Installation Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Getting Started - XL Fortran for Multicore Acceleration for Linux*
- *Compiler Reference - XL Fortran for Multicore Acceleration for Linux*
- *Language Reference - XL Fortran for Multicore Acceleration for Linux*
- *Optimization and Programming Guide - XL Fortran for Multicore Acceleration for Linux*
- *Installation Guide - XL Fortran for Multicore Acceleration for Linux*
- *Performance Analysis with the IBM Full-System Simulator*
- *IBM Full-System Simulator User's Guide*
- *IBM Visual Performance Analyzer User's Guide*

IBM PowerPC® Base

- *IBM PowerPC Architecture™ Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *IBM PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

Glossary

Accelerator

General or special purpose processing element in a hybrid system. DaCS elements which have a parent (host) element are considered accelerators. An accelerator elements can be a host as well, if it has children (other accelerators) attached.

AE

See Accelerator.

DaCS Element

A general or special purpose processing element in a topology. This refers specifically to the physical unit in the topology. A DE can serve as a Host or an Accelerator.

DE

See DaCS element.

de_id

A unique number assigned by the DaCS application at runtime to a physical processing element in a topology.

group

A group construct specifies a collection of DaCS DEs and processes in a system. A group is also used to define the participants in a barrier operation.

handle

A handle is an abstraction of a data object; usually a pointer to a structure.

HE

See Host.

Host

A general purpose processing element, acting as a supervisor, control or master processor. This type of element usually runs a full operating system and manages jobs running on other DEs. This is referred to as a Host Element (HE).

Hybrid

A 64 bit x86 system using a Cell BE as an accelerator.

node

A node is a functional unit in the system topology, consisting of one host together with all the accelerators connected as children in the topology (this includes any children of accelerators).

parent

The parent of a DE is the DE that resides immediately above it in the topology tree. A parent DE is always a Host Element.

PPE

PowerPC Processor Element. The general-purpose processor in the Cell/B.E. processor.

process

A process is a standard UNIX-type process with an individual address space.

SIMD

Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SPE

Synergistic Processor Element. Extends the PowerPC 64 architecture by acting as cooperative offload processors (synergistic processors), with the direct memory access (DMA) and synchronization mechanisms to communicate with them (memory flow control), and with enhancements for real-time management. There are 8 SPEs on each Cell/B.E. processor.

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

System X

This is a project-neutral description of the supervising system for a node.

thread

A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously if each thread has its own architectural state (registers, program counter, flags, and other program-visible state). Each SPE can support only a single thread at any one time. Multiple SPEs can simultaneously support multiple threads. The PPE supports two threads at any one time, without the need for software to create the threads. It does this by duplicating the architectural state. A thread is typically created by the `pthreads` library.

topology

A topology is a configuration of DaCS elements in a system. The topology specifies how the different processing elements in a system are related to each other. DaCS assumes a tree topology: each DE has at most one parent.

Index

Printed in USA

SC33-8408-01