Data Communication and Synchronization for Cell
Programmer’s Guide and API Reference
Version 1.0
DRAFT
Data Communication and Synchronization for Cell
Programmer’s Guide and API Reference
Version 1.0
DRAFT
Note
Before using this information and the product it supports, read the information in “Notices” on page 93.
<table>
<thead>
<tr>
<th>Appendix D. DaCS trace events</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>DaCS API hooks.</td>
<td>85</td>
</tr>
<tr>
<td>DaCS performance hooks.</td>
<td>86</td>
</tr>
<tr>
<td>Appendix E. Error codes</td>
<td>89</td>
</tr>
<tr>
<td>Appendix F. Accessibility features</td>
<td>91</td>
</tr>
<tr>
<td>Notices</td>
<td>93</td>
</tr>
<tr>
<td>Trademarks</td>
<td>95</td>
</tr>
<tr>
<td>Terms and conditions</td>
<td>96</td>
</tr>
<tr>
<td>Related documentation</td>
<td>97</td>
</tr>
<tr>
<td>Glossary</td>
<td>99</td>
</tr>
<tr>
<td>Index</td>
<td>101</td>
</tr>
</tbody>
</table>
About this publication

This programmer’s guide provides detailed information regarding the use of the Data Communication and Synchronization 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 E, “Accessibility features,” on page 91.

Who should use this book

This book is intended for use by accelerated library developers and compute kernel developers.

Related information

See “Related documentation” on page 97.

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](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).
Chapter 1. Overview

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 hierarchical memory system). 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 abstractions 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 otherDEs. 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).

The DaCS services provide the functionality needed to perform these roles and can be divided into the following categories:

Resource reservation
The resource reservation services allow an HE to reserve AEs 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
The process management services provide the means for an HE to execute and manage accelerated applications on AEs, including, but not limited to, remote process launch, and remote error notification.

Group management
The group management services provide the means to designate dynamic groups of processes for participation in collective operations. In SDK 3.0 this is limited to process execution synchronization (barrier).

Remote memory
The remote memory services provide the means to create, share, transfer data to, and transfer data from a remote memory segment. The data transfers are performed using a one-sided put/get remote direct memory access (rDMA) model. These services also provide the ability to scatter/gather lists of data, and provide optional enforcement of ordering for the data transfers.

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

Mailboxes
The mailbox services provide a simple interface for synchronous transfer of small (32-bit) messages from one process to another.
**Process Synchronization**

The process synchronization services provide the means to coordinate or synchronize process execution. In SDK 3.0 this is limited to the *barrier* synchronization primitive.

**Data Synchronization**

The data synchronization services provide the means to synchronize and serialize data access. These include management of *wait identifiers* for synchronizing data transfers, as well as *mutex* primitives for data serialization.

**Error Handling**

The error handling services enable the user to register error handlers and gather error information.
Chapter 2. Installing and configuring DaCS

The DaCS library should be installed as a component of the Cell BE Software Development Kit. Once it is installed no further configuration is necessary.

Installation

Several packages are available that provide the means to develop, deploy and debug DaCS applications on your Cell BE system. The following table shows the package names with a short description:

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dacs-3.0.#-#.ppc64.rpm</td>
<td>DaCS Runtime - Contains the optimized PPU shared library.</td>
</tr>
<tr>
<td>dacs-3.0.#-#.src.rpm</td>
<td>DaCS Source - Source code for the DaCS libraries.</td>
</tr>
<tr>
<td>dacs-devel-3.0.#-#.ppc64.rpm</td>
<td>DaCS Development - Contains the header files, optimized static PPU and SPU libraries, and debug libraries (PPU and SPU, static and shared).</td>
</tr>
<tr>
<td>dacs-cross-devel-3.0.#-#.noarch.rpm</td>
<td>DaCS Cross Development - Contains the header files and libraries needed for cross-architecture development.</td>
</tr>
<tr>
<td>dacs-debuginfo-3.0.#-#.ppc64.rpm</td>
<td>DaCS Debug Symbols - Contains the debugging symbols for the DaCS runtime library.</td>
</tr>
<tr>
<td>dacs-trace-3.0.#-#.ppc64.rpm</td>
<td>DaCS Trace Enabled Runtime - Contains the trace enabled PPU shared library.</td>
</tr>
<tr>
<td>dacs-trace-debuginfo-3.0.#-#.ppc64.rpm</td>
<td>DaCS Trace Debug Symbols - Contains the debugging symbols for the DaCS trace-enabled runtime library.</td>
</tr>
<tr>
<td>dacs-trace-devel-3.0.#-#.ppc64.rpm</td>
<td>DaCS Trace Enabled Development - Contains the header files and trace-enabled PPU and SPU static libraries.</td>
</tr>
<tr>
<td>dacsman-3.#-#.noarch.rpm</td>
<td>DaCS man pages.</td>
</tr>
</tbody>
</table>

See the SDK 3.0 Installation Guide for detailed installation and configuration information.
Chapter 3. Programming with DaCS

How to compile and build applications which use DaCS

Process Management Model

When working with the host and accelerators 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 id) and its Process Id (Pid). The DE Id is retrieved when the accelerator is reserved (using dacs_reserve_children()) and the Pid when the process is started (using dacs_de_start()). Since the parent is not reserved and a process is not started on it two constants are provided to identify the parent DACS_DE_PARENT and DACS_PID_PARENT. Similarly, to identifying the calling process itself, the constants DACS_DE_SELF and DACS_PID_SELF are provided.

Resource Sharing Model

The APIs supporting the locking primitives, remote memory access, 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 linked 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 actioned with.
- Usage: This may require closure (such as in the case of groups) or it 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 not longer being used and cleans up any local storage. The creator does a destroy (in this case dacs_remote_mem_destroy()) which will wait for all of the DEs it has shared with to release the item, and then destroy the shared item.

DaCS API functions

The DaCS library API services are provided as functions in the C language. The protocols and constants required are made available to the compiler by including the DaCS header file dacs.h as:

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

In general the return value from these functions is an error code (see Appendix E, "Error codes," on page 89). Data is returned within parameters passed to the functions.
**Note:** Implementations may provide options, restrictions and error codes that are not specified here.

**Note:** When more than one error condition is present it is not guaranteed which one will be reported.

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_runtime_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_runtime_exit()` to free all resources used by the environment.

### Building a DaCS application

Three versions of the DaCS libraries are provided with the DaCS packages: optimized, debug and traced. The optimized libraries have minimal error checking 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 C, “Performance and debug trace,”](#) on page 83 for more information on configuring and using traced libraries.

Both static and shared libraries are provided for the PPU but only static libraries are provided for the SPU. The desired library is selected by linking to the chosen library in the appropriate path. The static library is named `libdacs.a`, and the shared library is `libdacs.so`. The locations of these are:

<table>
<thead>
<tr>
<th>Description</th>
<th>PPU Library Path</th>
<th>SPU Library Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized</td>
<td>/usr/lib64</td>
<td>/usr/spu/lib</td>
</tr>
<tr>
<td>Debug</td>
<td>/usr/lib64/dacs/debug</td>
<td>/usr/spu/lib/dacs/debug</td>
</tr>
<tr>
<td>Traced</td>
<td>/usr/lib64/dacs/trace</td>
<td>/usr/spu/lib/dacs/trace</td>
</tr>
</tbody>
</table>
Chapter 4. Initializing and closing down the DaCS library

The `dacs_runtime_init` and `dacs_runtime_exit` services initialize and close down access to the DaCS library.

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

Calling `dacs_runtime_exit()` on an AE causes the communications between the AE and HE to be stopped. Calling `dacs_runtime_init()` after `dacs_runtime_exit()` will fail because once communications are stopped they can only be initiated by the HE calling `dacs_de_start()` to start a new process.
NAME
dacs_runtime_init - Initialize all runtime services for DaCS.

SYNOPSIS
DACS_ERR_T dacs_runtime_init ( void *, void * )

Call parameters
All parameters must be set to NULL for SDK 3.0. Passing in a value other than NULL will result in the error DACS_ERR_INVALID_ADDR.

DESCRIPTION
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.
- DACS_ERR_INITIALIZED: DaCS is already initialized.

SEE ALSO
dacs_runtime_exit(3)
dacs_runtime_exit

NAME

dacs_runtime_exit - Close down all runtime services for DaCS.

SYNOPSIS

DACS_ERR_T dacs_runtime_exit ( void )

Parameters
None

DESCRIPTION

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 support 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 DaCSD.

SEE ALSO

[dacs_runtime_init(3)]
Chapter 5. 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 SDK 3.0 only child-related APIs are defined and supported.

**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**

DACS_ERR_T dacs_get_num_avail_children (DACS_DE_TYPE_T type, uint32_t *num_children)

- **Call parameter**
  - type
    - The type of children to report. In SDK 3.0 this must be:
      - DACS_DE_SPE.

- **Return parameter**
  - *num_children
    - The number of available children. This may be zero if either no children of the requested type exist, or children exist but cannot be reserved.

**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.

**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.
- DACS_ERR_INVALID_ADDR: invalid pointer.
- DACS_ERR_INVALID_ATTR: invalid flag or enumerated constant.

**SEE ALSO**

dacs_reserve_children(3) dacs_release_delist(3)
**dacs_reserve_children**

**NAME**

dacs_reserve_children - Reserve children of a specified type.

**SYNOPSIS**

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

**Call parameters**

- type: The type of children to report. In SDK 3.0 this must be:
  - DACS_DE_SPE.
- *num_children: A pointer to the number of children requested.

**Return parameters**

- *num_children: A pointer to the number of children actually reserved. This may be less than or equal to the number requested.
- *de_list: 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.

**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.

**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.

**SEE ALSO**

dacs_get_num_avail_children(3) dacs_release_delist(3)
**NAME**

dacs_release_de_list - Release the reservations for a list of DEs.

**SYNOPSIS**

DACS_ERR_T dacs_release_de_list ( uint32_t num_des , de_id_t *de_list )

Call parameters
num_des The number of DEs in the list.
*de_list A pointer to the list of DEs to release.

**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.

**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.

**SEE ALSO**

dacs_get_num_avail_children(3) dacs_reserve_children(3)
Chapter 6. Process management

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

dacs_de_start

NAME

dacs_de_start - Start a process on a DE.

SYNOPSIS

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 )

Call parameters

de
   The target DE where the program will execute.
*prog
   A pointer to the program text to execute. What this points to is platform-dependent, and also dependent on the creation_flags parameter.
**argv
   A pointer to an array of pointers to argument strings (the argument list), terminated by a NULL pointer.
**envv
   A pointer to an array of pointers to environment variable strings (the environment list), terminated by a NULL pointer.
creation_flags
   An implementation-specific flag that specifies how the executable program is found. This can be any of:
   • DACS_PROC_LOCAL_FILE: a fully qualified pathname,
   • DACS_PROC_LOCAL_FILE_LIST: a list of fully qualified pathnames,
   • DACS_PROC_REMOTE_FILE: a fully qualified path on a remote system, or
   • DACS_PROC_EMBEDDED: the handle of an embedded executable image.

Return parameter

*pid
   A pointer to a location where the process id is stored on successful return.

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 and implementation dependent.

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_PROHIBITED**: the operation is prohibited by the implementation.
• **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_ARGV**: *argv* is too large or invalid.
• **DACS_ERR_INVALID_ENV**: *env* is too large or invalid.

**SEE ALSO**

`dacs_num_processes_supported(3)`  `dacs_num_processes_running(3)`
`dacs_de_wait(3)`  `dacs_de_test(3)`
dacs_num_processes_supported

NAME

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

SYNOPSIS

DACS_ERR_T dacs_num_processes_supported ( de_id_t de, uint32_t *num_processes )

Call parameter

de The DE to query.

Return parameter

*num_processes A pointer to a location where the maximum number of processes that can be started on this DE is stored.

Note: In SDK 3.0 only one 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.

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.

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

DACS_ERR_T dacs_num_processes_running ( de_id_t de, uint32_t *num_processes )

Call parameter

de  The DE to query.

Return parameter

*num_processes  A pointer to a location where the number of processes currently running on the target DE is stored.

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 had their exit status reaped (with dacs_de_test() or dacs_de_wait()). The target DE must have been reserved by the caller.

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.

SEE ALSO

dacs_de_start(3) dacs_num_processes_supported(3) dacs_de_wait(3)
dacs_de_test(3)
**NAME**

*dacs_de_wait* - Block the caller waiting for a process running on a DE to finish.

**SYNOPSIS**

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

**Call parameters**

- `de` The target DE.
- `pid` The target process.

**Return parameter**

- `*exit_status` A pointer to a location where the exit code is stored (if `DACS_STS_PROC_FINISHED` or `DACS_STS_PROC_FAILED`) or the signal number (if `DACS_STS_PROC_ABORTED`).

**DESCRIPTION**

The *dacs_de_wait* service returns the status of the target process, if it was successful, or an error code. 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.

When this service or *dacs_de_test()* detects a finished, failed, or aborted status, the status is reaped. Once the status of a process has been reaped, subsequent calls to query its status will fail with `DACS_ERR_INVALID_PID`.

**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. The platform-specific exception code is returned in `exit_status`. For Linux/UNIX this is the signal number which caused the termination.
- `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.

**SEE ALSO**

- `dacs_de_start(3)`
- `dacs_num_processes_supported(3)`
- `dacs_num_processes_running(3)`
- `dacs_de_test(3)`

Chapter 6. Process management 19
**NAME**

dacs_de_test - Test the status of a process.

**SYNOPSIS**

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

Call parameters

de The target DE.

pid The target process.

Return parameter

*exit_status A pointer to a location where the exit code is stored (if DACS_STS_PROC_FINISHED or DACS_STS_PROC_FAILED) or the signal number (if DACS_STS_PROC_ABORTED).

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

**DESCRIPTION**

The dacs_de_test service returns the status of the target process, if it was successful, or an error code.

When this service or dacs_de_wait() detects a finished, failed, or aborted status, the status is reaped. Once the status of a process has been reaped, subsequent calls to query its status will fail with DACS_ERR_INVALID_PID.

**RETURN VALUE**

The dacs_de_test service returns an error indicator defined as:

- DACS_STS_PROC_RUNNING: the process is still running,
- 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. The platform-specific exception code is returned in exit_status. For Linux/UNIX this is the signal number which caused the termination.
- 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.

**SEE ALSO**

[dacs_de_start(3)](dacs_de_start(3)) [dacs_num_processes_supported(3)](dacs_num_processes_supported(3))
[dacs_num_processes_running(3)](dacs_num_processes_running(3)) [dacs_de_wait(3)](dacs_de_wait(3))
Chapter 7. Group functions

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 related processes that share a common set of resources. A process is an identifiable unit of execution.

A group is referred to by a handle. The handle may be a pointer, offset, index or any other unique identifier of the group.

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

The members of a group work together to perform a common task, through the use of shared resources. Being a member of a group requires participation in group activities.

Group design

Membership
In DaCS membership is by invitation only. A process cannot declare itself a member of a group; it must be added by the group creator. This works on the basis that the creator of a group resource knows which processes will be participating in the use of the resource. The member being added to a group must also cooperate by accepting membership, and must make a request to leave a group.

Group Leader/Owner
Groups can only be created on an HE. The HE, by creating a group, implicitly becomes the group owner. This does not imply membership or participation in the group. The group owner is responsible for adding and removing members from the group. It is also responsible for allocating resources to be shared with the group. When the group is no longer needed, and all members have left, the owner then has the responsibility of destroying the group.

Barriers
Certain resources require a group in order to work properly. One such resource is a barrier. Without a known and fixed set of participants barriers cannot work properly. For this reason the concept of groups is necessary.

Barriers provide synchronization of a fixed number of participants. Groups are required in order for the barrier functionality to properly track barrier quora. Barriers are an implied resource associated with being in a group; they are not allocated, initialized, shared or destroyed.
Group usage scenario

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.

<table>
<thead>
<tr>
<th>Owner</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create the group: dacs_group_init(&amp;group, flags);</td>
<td></td>
</tr>
<tr>
<td>This creates an opaque group handle. The handle will then used by all members during group operations.</td>
<td></td>
</tr>
<tr>
<td>Add members (identified by DE and PID) to the group, one by one: dacs_group_add_member(de, pid, group);</td>
<td>Accept their addition, individually:</td>
</tr>
<tr>
<td>(Optional) Add itself to the group: dacs_group_add_member(DACS_DE_SELF, DACS_PID_SELF, group);</td>
<td>dacs_group_accept(de, pid, group);</td>
</tr>
<tr>
<td>(This does not require an accept response.)</td>
<td></td>
</tr>
<tr>
<td>Close the initialization of the group: dacs_group_close(group);</td>
<td></td>
</tr>
</tbody>
</table>

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 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.

<table>
<thead>
<tr>
<th>Owner</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait on barrier, individually: dacs_barrier_wait(group);</td>
<td></td>
</tr>
</tbody>
</table>

Termination

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

<table>
<thead>
<tr>
<th>Owner</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destroy the group: dacs_group_destroy(group);</td>
<td></td>
</tr>
</tbody>
</table>
Group owner functions

**dacs_group_init**

**NAME**

dacs_group_init - Initialize a DaCS group.

**SYNOPSIS**

DACS_ERR_T dacs_group_init ( dacs_group_t *group, uint32_t flags )

**Call parameter**

flags  
Flags for group initialization.

**Return parameter**

*group  
A pointer to a group handle which is filled in upon successful return.

**Note:** In SDK 3.0 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.

**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.

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

Call parameters
de The DE ID of the member to add. The group owner may specify a value of DACS_DE_SELF to add itself.
pid The process ID of the member to add. The group owner may 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.

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.

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.

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

DACS_ERR_T dacs_group_close ( dacs_group_t group )

Call parameter

| group | The handle of the group to close. |

DESCRIPTION

The dacs_group_close service closes the initialization of the specified group. Once closed, new members cannot be added to the group. The specified group must have been initialized with dacs_group_init(). This service may only be called by the owner of the group. Group collective operations will block until the group is closed.

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_OWNER: the caller is not the owner of the group.
- DACS_ERR_GROUP_CLOSED: the group is already closed.

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
DACS_ERR_T dacs_group_destroy ( dacs_group_t *group )
```

Call parameter

*group A pointer to the handle of the group to remove.

**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 other members have left the group. If the calling process is the owner of the group and also a member of the group it is implicitly removed from the group when it is destroyed.

**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.

**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)

**Group member functions**

Group member functions allow processes to join or leave groups created by other processes.
dacs_group_accept

NAME

dacs_group_accept - Accept membership to a DaCS group.

SYNOPSIS

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

Call parameters

   de  The DE ID of the group owner.
   pid The process ID of the group owner.

Return parameter

   *group A pointer to the handle of the group to be filled in.

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.

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.

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)

Chapter 7. Group functions  27
dacs_group_leave

NAME

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

SYNOPSIS

DACS_ERR_T dacs_group_leave ( dacs_group_t *group )

Call parameter
*group A pointer to the handle of the group to leave.

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.

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)

Process synchronization

The process synchronization services provide a means to coordinate or synchronize process execution.
dacs_barrier_wait

NAME

dacs_barrier_wait - Synchronize members of a group.

SYNOPSIS

DACS_ERR_T dacs_barrier_wait ( dacs_group_t group )

Call parameter

group The handle of the group with which to synchronize.

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.

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)
The data communication services transfer data from one process to another. To accommodate transfers across systems with different data representation formats (endian-ness), the services provide an option for byte swapping. The types of byte swapping supported are:

- **DACS_BYTE_SWAP_DISABLE**
  - no byte-swapping.
- **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).

Different platforms and implementations may also have different alignment restrictions for the data being transferred. The data communication services will return the **DACS_ERR_NOT_ALIGNED** error code when those alignment restrictions are not met.

**Note:** For DaCS on Cell BE the services support data transfers of any size other than zero. DaCS enforces no size limit, so the size of transfer that can actually be performed is only restricted by the inherent system limitations. Data transfers that are smaller than 16 bytes must be aligned according to the size of the transfer, rounded down to the nearest power of two at or below the size. Data transfers that are 16 bytes or larger must be 16 byte aligned. For DMA list operations, however, the transfer size for each element must be a multiple of 16.

The data communication services require that the caller specify either 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.

**Note:** In SDK 3.0 direct communication is only allowed between a parent and its children. Attempts to communicate to a process which is not the parent or child of the initiator will result in an error of **DACS_ERR_INVALID_TARGET**.

Three different data communication models are supported: remote direct memory access (rDMA), message passing, and mailboxes.

**Remote Direct Memory Access (rDMA)**

**Managing Remote Memory.**

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

**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.

**SYNOPSIS**

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

**Call parameters**

- ***addr**
  A pointer to the base address of the memory region to be shared.

- **size**
  The size of the memory region in bytes.

- **access_mode**
  The access mode to be given to the memory region. This may be:
  - DACS_READ_ONLY,
  - DACS_WRITE_ONLY, or
  - DACS_READ_WRITE.

**Return parameter**

- ***mem**
  A pointer to a remote memory handle to be filled in.

**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() and dacs_remote_mem_accept() services to share and gain access to remote shared memory.

**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.

**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)]
dacs_remote_mem_share

NAME

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

SYNOPSIS

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

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. |

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.

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.

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 a memory handle from a remote process.

SYNOPSIS

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

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 parameter
*mem
A pointer to the accepted memory handle.

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 filled in upon successful return.

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.

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 a previously accepted remote memory handle.

**SYNOPSIS**

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

Call parameter

*mem  A pointer to the remote memory handle.

**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.

**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.

**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 sharing of a specified memory region.

SYNOPSIS

DACS_ERR_T dacs_remote_mem_destroy ( dacs_remote_mem_t *mem )

Call parameter

*mem A pointer to a remote memory handle.

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.

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.

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
DACS_ERR_T dacs_remote_mem_query ( dacs_remote_mem_t mem,
DACS_REMOTE_MEM_ATTR_T attr, uint64_t *value )
```

Call parameters

- `mem` - The handle of the remote memory area to query.
- `attr` - The attribute to be queried. This may be one of:
  - `DACS_REMOTE_MEM_SIZE`,
  - `DACS_REMOTE_MEM_ADDR`, or
  - `DACS_REMOTE_MEM_ACCESS_MODE`.

Return parameter

- `*value` - A pointer to the location where the attribute value is to be returned. If the requested attribute is `DACS_REMOTE_MEM_ACCESS_MODE`, the output value will be one of:
  - `DACS_READ_ONLY`,
  - `DACS_WRITE_ONLY`, or
  - `DACS_READ_WRITE`.

**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.

**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.

**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)`

**rDMA block transfers**

The DMA services provide a means to perform direct memory accesses to and from remote memory.
dacs_put

NAME

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

SYNOPSIS

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 )

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 A pointer to the source memory buffer.
size The size of the transfer.
wid The communications wait identifier.
order_attr An ordering attribute. Possible values 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.
swap The little-endian or big-endian byte-swapping flag. Possible values are:
  • DACS_BYTE_SWAP_DISABLE,
  • DACS_BYTE_SWAP_HALF_WORD,
  • DACS_BYTE_SWAP_WORD or
  • DACS_BYTE_SWAP_DOUBLE_WORD.

See Chapter 8, “Data communication,” on page 31 for details.

DESCRIPTION

The dacs_put service initiates data transfer from the caller memory, specified by src_addr, to the target memory, specified by dst_remote_mem and remote_mem_offset. This operation is non-blocking (the call initiates the transfer, but the transfer may continue after the call returns). To ensure that the transfer has completed on the DE, so that the local buffer can be reused or changed, you should issue a call to dacs_wait() or dacs_test() with the same wait identifier.

The target remote memory region must have been previously accepted by the caller with a call to dacs_remote_mem_accept().

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_put service returns an error indicator defined as:

- **DACS_SUCCESS**: normal return.
- **DACS_ERR_BUF_OVERFLOW**: the buffer has overflowed - the specified offset or size exceed the bounds of the target buffer.
- **DACS_ERR_NOTAligned**: 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**: the size was zero or not supported by the platform.
- **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.

SEE ALSO

[dacs_get(3) dacs_put_list(3) dacs_get_list(3) dacs_test(3) dacs_wait(3)]
dacs_get

NAME

dacs_get - Get data from remote memory to local memory.

SYNOPSIS

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 )

Call parameters

*dst_addr        A pointer to the base address of the destination memory
    buffer.
src_remote_mem   The remote memory handle of the source buffer.
src_remote_mem_offset The offset into the offset in remote buffer where the get is to
    start.
size             The size of the transfer.
wid              A communications wait identifier.
order_attr       An ordering attribute. Possible values 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.
swap             The little-endian or big-endian byte-swapping flag. Possible
    values are:
    • DACS_BYTE_SWAP_DISABLE,
    • DACS_BYTE_SWAP_HALF_WORD,
    • DACS_BYTE_SWAP_WORD or
    • DACS_BYTE_SWAP_DOUBLE_WORD.

See Chapter 8, “Data communication,” on page 31 for details.

DESCRIPTION

The dacs_get service returns data from the target memory, specified by
src_remote_mem and src_remote_mem_offset, to the caller memory, specified by
dst_addr. This operation is non-blocking (the call initiates the transfer, but the
transfer may continue after the call returns). To ensure that the transfer has
completed you should issue a call to dacs_wait() or dacs_test() with the same
wait identifier.

The target remote memory region must have been previously accepted by the
caller using a call to dacs_remote_mem_accept().

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_BUF_OVERFLOW**: the buffer has overflowed - the specified offset and size exceed the bounds of the target buffer.
- **DACS_ERR_NOT_ALIGNED**: the buffer is not aligned properly for its size.
- **DACS_ERR_INVALID_ADDR**: the pointer is invalid.
- **DACS_ERR_INVALID_ATTRIB**: the flag or enumerated constant is invalid.
- **DACS_ERR_INVALID_SIZE**: the size is zero or not supported by the platform.
- **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.

SEE ALSO

- [dacs_put(3)]
- [dacs_put_list(3)]
- [dacs_get_list(3)]
- [dacs_test(3)]
- [dacs_wait(3)]

rDMA list transfers

The DMA list services enable scatter/gather operations between non-contiguous regions of memory and one contiguous region of memory.

The list services operate on DMA list elements. A list element is defined as a tuple (offset, size), where offset is a 64-bit offset into the remote memory block, and size is the size of the block.

**Note**: The size of the transfer specified by each list element must be a multiple of 16 bytes and must be 16 byte aligned.
**dacs_put_list**

**NAME**

dacs_put_list - Push data from local memory blocks to a remote memory area.

**SYNOPSIS**

DACS_ERR_T dacs_put_list ( dacs_remote_mem_t dst_remote_mem, dacs_dma_list_t *dst_dma_list, uint32_t dst_list_size, void *src_addr, 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 )

**Call parameters**

dst_remote_mem The remote memory handle for the destination buffer.
*dst_dma_list A pointer to a list of entries describing the transfer locations in the destination buffer.
dst_list_size The number of elements in the destination DMA list.
src_addr The base address of the source memory buffer.
*src_dma_list A pointer to a list of entries describing the transfer locations in the source buffer.
src_list_size The number of elements in the source DMA list.
wid The communications wait identifier associated with this transfer.
order_attr Ordering attribute. Possible values 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 completed.
  • DACS_ORDER_ATTR_NONE: no ordering is enforced.
swap The little-endian or big-endian byte-swapping flag. Possible values are:
  • DACS_BYTE_SWAP_DISABLE,
  • DACS_BYTE_SWAP_HALF_WORD,
  • DACS_BYTE_SWAP_WORD or
  • DACS_BYTE_SWAP_DOUBLE_WORD.

See Chapter 8, “Data communication,” on page 31 for details.

**DESCRIPTION**

The dacs_put_list service pushes data from the memory blocks specified in the src_dma_list to the remote memory area referenced by the dst_remote_mem handle using the specified dst_dma_list. The interface supports the specification of two DMA lists, one of which must contain a single element. This applies to both Scatter (src_list_size=1) and Gather to Remote (dst_list_size=1) operations.

The source address for each DMA operation is an effective address formed by the sum of src_addr and the offset specified in each DMA list element. The assumption is that all of the source data is in a contiguous buffer starting at src_addr. For cases where the source data may not be in a contiguous buffer with a known base address, a source address of zero may be specified. In this case the actual address of the data 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 on this DE you must make a call to `dacs_wait()` or `dacs_test()` with the same wait identifier. This ensures that the local buffers and transfer list parameter can be changed or reused.

The target remote memory region must have been previously accepted by the caller with a call to `dacs_remote_mem_accept()`.

**dacs_put_list - Scatter (src_list_size = 1 dst_list_size = 4)**

```
Local Memory                     Remote Memory

src_list[0]

DataA
DataB
DataC
DataD

dst_list[0]
dst_list[1]
dst_list[2]
dst_list[3]

DataA
DataB
DataC
DataD
```

**Dacs_put_list - Gather to Remote (src_list_size = 4 dst_list_size = 1)**

```
Local Memory                     Remote Memory

DataA
DataB
DataC
DataD

src_list[0]
src_list[1]
src_list[2]
src_list[3]

dst_list[0]

DataA
DataB
DataC
DataD
```

Put list to a remote memory region with `src_list_size = 1`
RETURN VALUE

The dacs_put_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.
- **DACS_ERR_NOTAligned**: 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**: the size is zero or not supported by the platform.
- **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.

SEE ALSO

dacs_put(3) dacs_get(3) dacs_get_list(3) dacs_test(3) dacs_wait(3)
**NAME**

dacs_get_list - Get data from a remote memory area and place it in local buffers.

**SYNOPSIS**

```c
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 )
```

**Call parameters**

- `*dst_addr` - A pointer to the base address of the destination memory buffer.
- `*dst_dma_list` - A pointer to a list of entries describing transfer locations in the destination buffer.
- `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` - A pointer to a list of entries describing transfer locations in the source buffer.
- `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. Possible values 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.
- `swap` - The little-endian or big-endian byte-swapping flag. Possible values are:
  - `DACS_BYTE_SWAP_DISABLE`,
  - `DACS_BYTE_SWAP_HALF_WORD`,
  - `DACS_BYTE_SWAP_WORD`,
  - `DACS_BYTE_SWAP_DOUBLE_WORD`.

See Chapter 8, “Data communication,” on page 31 for details.

**DESCRIPTION**

The `dacs_get_list` service gets data from the remote memory area referenced by the `dst_remote_mem` struct, using the specified `dma_list`, and places it in the buffers specified by `dst_dma_list`. The interface supports the specification of two DMA lists, one of which must contain a single element. This applies to both `Gather` (`dst_list_size=1`) and `Scatter to local` (`src_list_size=1`) operations.

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)**

![Diagram of dacs_get_list - Gather](image)

Get list from a remote memory region with source list size = 1

**dacs_get_list - Scatter to local (src_list_size = 1 dst_list_size = 4)**

![Diagram of dacs_get_list - Scatter to local](image)
**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.
- **DACS_ERR_NOT_ALIGNED**: 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**: the size is zero or not supported by the platform.
- **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.

**SEE ALSO**

[dacs_put(3) dacs_get(3) dacs_put_list(3) dacs_test(3) dacs_wait(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.
dacs_send

NAME

dacs_send - send a message to another process

SYNOPSIS

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 )

Call parameters

*src_data A pointer to the beginning of the source (send) message buffer.
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.
wid A wait identifier.
swap The little-endian or big-endian byte-swapping flag. Possible values are:
  • DACS_BYTE_SWAP_DISABLE,
  • DACS_BYTE_SWAP_HALF_WORD,
  • DACS_BYTE_SWAP_WORD or
  • DACS_BYTE_SWAP_DOUBLE_WORD.

See Chapter 8, “Data communication,” on page 31 for details.

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.

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.

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_NOTAligned: the requested data transfer does not have proper
    alignment for its size.
  • DACS_ERR_INVALID_SIZE: the size is zero or not supported by the platform.
  • DACS_ERR_INVALID_WID: the wait identifier is invalid.
  • DACS_ERR_INVALID_STREAM: the stream identifier is invalid.
SEE ALSO

dacs_recv(3), dacs_wait(3), dacs_test(3)

dacs_recv

NAME

dacs_recv - receive a message from another process

SYNOPSIS

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 )

Call parameters

*dst_data      A pointer to the beginning of the destination (receive) data buffer.
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            A wait identifier.
swap           The little-endian or big-endian byte-swapping flag. Possible values are:
                • DACS_BYTE_SWAP_DISABLE,
                • DACS_BYTE_SWAP_HALF_WORD,
                • DACS_BYTE_SWAP_WORD or
                • DACS_BYTE_SWAP_DOUBLE_WORD.

See Chapter 8, “Data communication,” on page 31 for details.

Return parameter

*dst_data      The pointer to the received data buffer.

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 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_BYTE_SWAP_MISMATCH, and
no data is transferred.
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_NOT_ALIGNED: the buffer is not aligned properly for the size of the transfer.
- DACS_ERR_INVALID_SIZE: the size is zero or not supported by the platform.
- DACS_ERR_INVALID_WID: the wait identifier is invalid.
- DACS_ERR_INVALID_STREAM: the stream identifier is invalid.

SEE ALSO

dacs_send(3)  dacs_wait(3)  dacs_test(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 are between a host (parent) and an accelerator process (DE id and Pid). Each mailbox has two sets of slots. One set is written to by the host and read by the accelerator process, and the other is written to by the accelerator process and read by the host. A host with a single accelerator running a single process will have one mailbox with 4 slots in each set. A host with two accelerators, each running a single process, will have two mailboxes each with 4 slots in each set. Thus the host will have 4 slots in one mailbox for mail coming from one accelerator process and 4 slots in another mailbox for mail coming from the other accelerator process.

Note: 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.

The mailbox depth is limited to 4 (host) and 4 (accelerator).
**NAME**

dacs_mailbox_write - Send a single variable to another process.

**SYNOPSIS**

DACS_ERR_T dacs_mailbox_write ( uint32_t *msg, de_id_t dst_de,
   dacs_process_id_t dst_pid)

Call parameters
*msg  A pointer to the message to write.
dst_de The message destination DE.
dst_pid The destination process id.

**DESCRIPTION**

The dacs_mailbox_write service writes a single 32-bit unsigned integer to the
specified target mailbox. There are an 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.

**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.

**SEE ALSO**

[\dacs_mailbox_read(3)\] [\dacs_mailbox_test(3)\]
dacs_mailbox_read

NAME

dacs_mailbox_read - Receive a single variable from another process.

SYNOPSIS

DACS_ERR_T dacs_mailbox_read ( uint32_t *msg, de_id_t src_de,
   dacs_process_id_t src_pid)

Call parameters
src_de The message source DE.
src_pid The message source process.

Return parameter
*msg A pointer to the message received.

DESCRIPTION

The dacs_mailbox_read service reads a single 32-bit unsigned integer from the
specified source mailbox. There are a number of mailbox slots for each process; this
number is defined by the implementation. If the source does not have any pending
mailbox messages this service call blocks until one arrives.

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.

SEE ALSO

\[ dacs_mailbox_write(3), dacs_mailbox_test(3) \]
**dacs_mailbox_test**

**NAME**

dacs_mailbox_test - Test if a mailbox access will succeed.

**SYNOPSIS**

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)

**Call parameters**

* rw_flag
  Flag to indicate which mailbox to test:
  - 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.

* de
  The DE owning the mailbox to test.

* pid
  The process owning the mailbox to test.

**Return parameter**

* *mbox_status
  A pointer to the location where the mailbox status is returned. The contents are:
  - set to zero if the mailbox will block, or
  - set to non-zero if the mailbox will not block.

**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().

**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.

**SEE ALSO**

[dacs_mailbox_read(3) dacs_mailbox_write(3)]
Chapter 9. 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**

DACS_ERR_T dacs_wid_reserve ( dacs_wid_t *wid )

Return parameter

*wid A pointer to the reserved wait identifier.

**DESCRIPTION**

The dacs_wid_reserve service reserves a wait identifier.

**RETURN VALUE**

The dacs_wid_release 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.

**SEE ALSO**

[dacs_wid_release(3)]
**dacs_wid_release**

**NAME**

dacs_wid_release - Release a reserved wait identifier.

**SYNOPSIS**

DACS_ERR_T dacs_wid_release ( dacs_wid_t *wid )

Call parameter

*wid A pointer to the wait identifier to be released.

**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.

**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)
Chapter 10. Transfer completion

A wait identifier is reserved and assigned to a data communication operation when it is started. 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 DE so the parameters can be changed or reused.

SYNOPSIS

DACS_ERR_T dacs_test ( dacs_wid_t wid )

Call parameter

wid A communication wait identifier

DESCRIPTION

The dacs_test service checks the data transfers for the given communication wait identifier and returns their status.

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_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.

SEE ALSO

dacs_wait(3) dacs_put(3) dacs_get(3) dacs_put_list(3) dacs_get_list(3)
NAME

dacs_wait - Wait for a communications operation to finish. Wait for communication operations to finish on this DE so the parameters can be changed or reused.

SYNOPSIS

DACS_ERR_T dacs_wait ( dacs_wid_t wid )

Call parameter

wid A communication wait identifier

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.

RETURN VALUE

The dacs_wait service returns an error indicator defined as:

- DACS_WID_READY: all data transfers have completed.
- DACS_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.

SEE ALSO

dacs_test(3) dacs_put(3) dacs_get(3) dacs_put_list(3) dacs_get_list(3)
Chapter 11. Locking Primitives

Shared data accesses can be serialized with DaCS by using a mutual exclusion primitive (mutex) to protect critical sections. A mutex can be hosted on any DE memory space and can be local or remote relative to the process calling these services.

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, and
- the lock is not thread based within the DE/PID. Any thread in the locking DE/PID can unlock the mutex.

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(), and

mutex locking services, for locking and unlocking a mutex, which include dacs_mutex_lock(), dacs_mutex_unlock() and dacs_mutex_try_lock().
NAME
dacs_mutex_init - Initialize a mutual exclusion variable.

SYNOPSIS
DACS_ERR_T dacs_mutex_init ( dacs_mutex_t *mutex )

Return parameter
*mutex A pointer to a newly initialized mutex handle.

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

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.

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

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

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.

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.

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_PID: the specified PID does not refer to an active process.
- DACS_ERR_TARGET: this operation is not allowed for the target process.
- DACS_ERR_HANDLE: the specified mutex handle is not valid.

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_accept**

**NAME**

dacs_mutex_accept - Receive a share on a mutual exclusion variable from a remote process.

**SYNOPSIS**

```c
DACS_ERR_T dacs_mutex_accept ( de_id_t src_de, dacs_process_id_t src_pid,
    dacs_mutex_t *mutex )
```

Call parameters

- `src_de` The source DE which is sharing the mutex handle.
- `src_pid` The source PID which is sharing the mutex handle.

Return parameter

- `*mutex` A pointer to the handle of the accepted mutex.

**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()`.

**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.

**SEE ALSO**

- `dacs_mutex_init(3)`
- `dacs_mutex_share(3)`
- `dacs_mutex(3)`
- `dacs_mutex_lock(3)`
- `dacs_mutex_try_lock(3)`
- `dacs_mutex_unlock(3)`
- `dacs_mutex_release(3)`
- `dacs_mutex_destroy(3)`
**NAME**

dacs_mutex_lock - Acquire a lock on a mutual exclusion variable.

**SYNOPSIS**

DACS_ERR_T dacs_mutex_lock ( dacs_mutex_t mutex )

Call parameter

mutex The handle of the mutex to lock.

**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.

**SEE ALSO**

[Chapter 11. Locking Primitives](#)
dacs_mutex_try_lock

NAME

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

SYNOPSIS

DACS_ERR_T dacs_mutex_try_lock ( dacs_mutex_t mutex )

Call parameter
mutex The handle of the mutex to lock.

DESCRIPTION

The dacs_mutex_try_lock service attempts to acquire a lock on 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(). If the mutex is available for locking, this
service gets the lock and returns. If a lock on the mutex is not available, this
method returns with a DACS_ERR_MUTEX_BUSY status.

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.

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

DACS_ERR_T dacs_mutex_unlock ( dacs_mutex_t mutex )

Call parameter
mutex The handle of the mutex to unlock.

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.

SEE ALSO

[link to related documentation]
**NAME**

dacs_mutex_release - Release a mutual exclusion variable.

**SYNOPSIS**

DACS_ERR_T dacs_mutex_release ( dacs_mutex_t *mutex )

Call parameter
*mutex A pointer to the handle of the mutex to release.

**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.

**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)
NAME

dacs_mutex_destroy - Destroy a mutual exclusion variable.

SYNOPSIS

DACS_ERR_T dacs_mutex_destroy ( dacs_mutex_t *mutex )

Call parameter
mutex A pointer to the handle of the mutex to destroy.

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.

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.

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)
Chapter 12. 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 SDK 3.0 any synchronous 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 (host process) and termination requests from a parent (accelerator 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_runtime_init() for this purpose. Applications using the DaCS library should not create any application threads before calling dacs_runtime_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:

```c
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 */
return dacs_rc;
}
```

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:

```c
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 */
return dacs_rc;
}
```
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 3.0, 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:

```c
-- 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
```

## dacs_errhandler_reg

### NAME

dacs_errhandler_reg - Register an error handler to be called when an asynchronous or fatal error occurs.

### SYNOPSIS

```c
DACS_ERR_T dacs_errhandler_reg ( dacs_error_handler_t handler, uint32_t flags )
```

**Call parameters**

- **handler**
  
  A pointer to an error handling function. This function will be passed the error object containing the error information, and returns a boolean indicating whether termination is requested or not. For fatal internal errors the process will be terminated without consideration for the handler's return value.

- **flags**
  
  Flags for error handling options. 

  **Note:** In SDK 3.0 no flags are supported: the flags value passed in must be 0 (zero).

The prototype of the handler is:

```c
int (*dacs_error_handler_t)( dacs_error_t error )
```

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.

### 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 SDK 3.0 the return value from the user handler will be ignored in all cases.

**Note:** If the error handler is coded in the form

```c
int my_errhandler(dacs_error_t error)
```

then write the registration as

```c
dacs_rc=dacs_errhandler_reg((dacs_error_handler_t)&my_errhandler,0);
```

where `dacs_rc` has been declared as a variable of type `DACS_ERROR_T`. 

Chapter 12. Error handling 71
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.

**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
const char * dacs_strerror ( DACS_ERR_T errcode )
```

Call parameter

- `errcode` - An error code that was returned by a DaCS API.

**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, or NULL if no string was found.

**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)`
**NAME**

dacs_error_num - Return the error code for the specified error handle.

**SYNOPSIS**

DACS_ERR_T dacs_error_num ( dacs_error_t error )

Call parameter
error An error handle.

**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, 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) ]
**NAME**

dacs_error_code - Retrieve the extended error code from the specified error object.

**SYNOPSIS**

DACS_ERR_T dacs_error_code ( dacs_error_t error , uint32_t *code )

Call parameter
error            An error handle.

Return parameter
*code            A pointer to the error code.

**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.

**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) ]
NAME

dacs_error_str - Retrieve the error string for the specified error object.

SYNOPSIS

DACS_ERR_T dacs_error_str ( dacs_error_t error, const char **errstr )

Call parameter
error
An error handle.

Return parameter
**errstr
A pointer to the error string.

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().

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.

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**

DACS_ERR_T dacs_error_de ( dacs_error_t error, de_id_t *de )

Call parameter

error An error handle.

Return parameter

*de A pointer indicating the DE which was the source of the error.

**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.

**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)
**NAME**

dacs_error_pid - Retrieve the originating PID for the specified error object.

**SYNOPSIS**

DACS_ERR_T dacs_error_pid ( dacs_error_t error, dacs_process_id_t *pid )

Call parameter
error An error handle.

Return parameter
*pid A pointer indicating the PID which was the source of the error.

**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.

**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)]
Appendix A. Data types

Data type declarations in this document follow the C99 (ISO/IEC 9899:1999) convention. In addition, the following data types are defined in dacs.h:

**dacs_remote_mem_t**
A handle to a memory region specified for use by other remote processes

**dacs_dma_list_t**
This structure describes a dma_list_element in 64 bit addressing mode

- `uint64_t offset`: 64-bit offset into the dacs_remote_mem_t block
- `uint64_t size`: size of the buffer to be transferred, in bytes

```c
typedef struct dma_list {
  uint64_t_t offset;
  uint64_t_t size;
} dacs_dma_list_t;
```

**dacs_error_t**
This is an opaque handle which refers to an error object. A handle of this type is passed to the user-registered handler.

```c
int (*dacs_error_handler_t)(dacs_error_t error)
```
The user-registered handler must accept a handle to an error object and return 1 or 0 to indicate whether the error is deemed fatal.
Appendix B. DaCS DE types

The current DaCS Element (DE) types in the current supported DaCS topology are listed below.

**DACS_DE_SYSTEMX**

The supervising host for a node.

**DACS_DE_CELL_BLADE**

An entire Cell BE blade. 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 BE Blade Engine. A Cell BE Blade contains two of these. If used this way, a Cell BE 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 BE node allows finer control of memory and processor affinity and may increase performance.

**DACS_DE_SPE**

Cell BE Synergistic Processing Element.
Appendix C. 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 SDK 3.0.

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 3.0 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

When a PDT-enabled application starts, PDT reads its configuration from a file. 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

In order to see the trace events the application must be built with the trace-enabled libraries. To see SPE events the application’s SPE code must be rebuilt with special compile settings (see the PDT User’s Guide for specifics) and needs to be linked with /usr/spu/lib/dacs/trace/libdacs.a, the trace-enabled DaCS SPU library code. To see PPE events the application must use the trace-enabled DaCS PPU code. If the application is using the static PPU library then it must be re-linked with /usr/lib64/dacs/trace/libdacs.a, the trace-enabled DaCS PPU library code. 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 before running the application:

LD_LIBRARY_PATH=/usr/lib64/dacs/trace:/usr/lib64/trace
Appendix D. 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 3. Trace hooks enabled by LIBDACS group (0x04) in the config file.

<table>
<thead>
<tr>
<th>Hook identifier</th>
<th>Traced values</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DACS_BARRIER_WAIT_ENTRY</td>
<td>group</td>
</tr>
<tr>
<td>_DACS_BARRIER_WAIT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_DE_KILL_ENTRY</td>
<td>deid, pid</td>
</tr>
<tr>
<td>_DACS_DE_KILL_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_DE_START_ENTRY</td>
<td>deid, text, argv, envv, creation_flags, p_pid</td>
</tr>
<tr>
<td>_DACS_DE_START_EXIT_INTERVAL</td>
<td>retcode, pid</td>
</tr>
<tr>
<td>_DACS_DE_TEST_ENTRY</td>
<td>deid, pid, p_exit_status</td>
</tr>
<tr>
<td>_DACS_DE_TEST_EXIT_INTERVAL</td>
<td>retcode, exit_status</td>
</tr>
<tr>
<td>_DACS_DE_WAIT_ENTRY</td>
<td>deid, pid, p_exit_status</td>
</tr>
<tr>
<td>_DACS_DE_WAIT_EXIT_INTERVAL</td>
<td>retcode, exit_status</td>
</tr>
<tr>
<td>_DACS_GENERIC_DEBUG</td>
<td>long1, long2, long3, long4, long5, long6, long7, long8, long9, long10</td>
</tr>
<tr>
<td>_DACS_GET_ENTRY</td>
<td>dst_addr, src, src_offset, size, wid, order_attr, swap</td>
</tr>
<tr>
<td>_DACS_GET_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_GET_LIST_ENTRY</td>
<td>dst_addr, dst_dma_list, dst_list_size, src_remote_mem, src_dma_list, src_list_size, wid, order_attr, swap</td>
</tr>
<tr>
<td>_DACS_GET_LIST_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MBOX_READ_ENTRY</td>
<td>msg, src_de, src_pid</td>
</tr>
<tr>
<td>_DACS_MBOX_READ_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MBOX_TEST_ENTRY</td>
<td>rw_flag, deid, pid, p_mbox_status</td>
</tr>
<tr>
<td>_DACS_MBOX_TEST_EXIT_INTERVAL</td>
<td>retcode, result</td>
</tr>
<tr>
<td>_DACS_MBOX_WRITE_ENTRY</td>
<td>msg, dst_de, dst_pid</td>
</tr>
<tr>
<td>_DACS_MBOX_WRITE_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_ACCEPT_ENTRY</td>
<td>deid, pid, mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_ACCEPT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
</tbody>
</table>
Table 3. Trace hooks enabled by LIBDACS group (0x04) in the config file. (continued)

<table>
<thead>
<tr>
<th>Hook identifier</th>
<th>Traced values</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DACS_MUTEX_DESTROY_ENTRY</td>
<td>mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_DESTROY_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_INIT_ENTRY</td>
<td>mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_INIT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_LOCK_ENTRY</td>
<td>mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_LOCK_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_RELEASE_ENTRY</td>
<td>mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_RELEASE_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_SHARE_ENTRY</td>
<td>deid, pid, mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_SHARE_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_TRY_LOCK_ENTRY</td>
<td>mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_TRY_LOCK_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_MUTEX_UNLOCK_ENTRY</td>
<td>mutex</td>
</tr>
<tr>
<td>_DACS_MUTEX_UNLOCK_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_PUT_ENTRY</td>
<td>dst, dst_offset, src_addr, size, wid, order_attr, swap</td>
</tr>
<tr>
<td>_DACS_PUT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_PUT_LIST_ENTRY</td>
<td>dst, dst_dma_list, dma_list_size, src_addr, src_dma_list, src_list_size, wid, order_attr, swap</td>
</tr>
<tr>
<td>_DACS_PUT_LIST_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RMEM_ACCEPT_ENTRY</td>
<td>src_de, src_pid, remote_mem</td>
</tr>
<tr>
<td>_DACS_RMEM_ACCEPT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RMEM_CREATE_ENTRY</td>
<td>addr, size, mode, local_mem</td>
</tr>
<tr>
<td>_DACS_RMEM_CREATE_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RMEM_DESTROY_ENTRY</td>
<td>remote_mem</td>
</tr>
<tr>
<td>_DACS_RMEM_DESTROY_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RMEM_RELEASE_ENTRY</td>
<td>remote_mem</td>
</tr>
<tr>
<td>_DACS_RMEM_RELEASE_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RMEM_SHARE_ENTRY</td>
<td>dst, dst_pid, local_mem</td>
</tr>
<tr>
<td>_DACS_RMEM_SHARE_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RUNTIME_EXIT_ENTRY</td>
<td>zero</td>
</tr>
<tr>
<td>_DACS_RUNTIME_EXIT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
<tr>
<td>_DACS_RUNTIME_INIT_ENTRY</td>
<td>argp, envp</td>
</tr>
<tr>
<td>_DACS_RUNTIME_INIT_EXIT_INTERVAL</td>
<td>retcode</td>
</tr>
</tbody>
</table>

**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_runtime_exit() function.
Table 4. Trace hooks enabled by LIBDACS_GROUP group (0x06) in the config file.

<table>
<thead>
<tr>
<th>Hook identifier</th>
<th>Traced values</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DACS_COUNTERS1</td>
<td>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</td>
</tr>
<tr>
<td>_DACS_COUNTERS2</td>
<td>dacs_mutex_try_success, dacs_mutex_try_failure, dacs_x1, dacs_x2</td>
</tr>
<tr>
<td>_DACS_HOST_MUTEX_INIT</td>
<td>lock</td>
</tr>
<tr>
<td>_DACS_HOST_MUTEX_LOCK</td>
<td>lock, miss</td>
</tr>
<tr>
<td>_DACS_HOST_MUTEX_TRYLOCK</td>
<td>lock, ret</td>
</tr>
<tr>
<td>_DACS_HOST_MUTEX_UNLOCK</td>
<td>lock</td>
</tr>
<tr>
<td>_DACS_PERF_GENERIC_DEBUG</td>
<td>long1, long2, long3, long4, long5, long6, long7, long8, long9, long10</td>
</tr>
<tr>
<td>_DACS_SPE_MUTEX_INIT</td>
<td>lock</td>
</tr>
<tr>
<td>_DACS_SPE_MUTEX_LOCK</td>
<td>lock, miss</td>
</tr>
<tr>
<td>_DACS_SPE_MUTEX_TRYLOCK</td>
<td>lock, ret</td>
</tr>
<tr>
<td>_DACS_SPE_MUTEX_UNLOCK</td>
<td>lock</td>
</tr>
<tr>
<td>_DACS_TIMERS</td>
<td>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</td>
</tr>
</tbody>
</table>
Appendix E. Error codes

This section describes the DaCS error codes.

All error codes which may be issued by DaCS APIs are listed here:

- **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_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_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_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_NOTAligned**: The buffer is incorrectly aligned for the size of the data.
- **DACS_ERR_NOT_INITIALIZED**: DaCS has not been initialized.
- **DACS_ERR_NOT_OWNER**: This operation is only permitted for the owner of the resource.
- **DACS_ERR_OWNER**: This operation is not permitted for the owner of the resource.
- **DACS_ERR_PROC_LIMIT**: The maximum number of processes supported has been reached.
- **DACS_ERR_PROHIBITED**: This operation is prohibited by the implementation.
- **DACS_ERR_RESOURCE_BUSY**: The specified resource is in use.
- **DACS_ERR_WID_ACTIVE**: A 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_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.
- **DACS_WID_NOT_ACTIVE**: There are no outstanding transfers to test.
Appendix F. Accessibility features

IBM® and accessibility

See the IBM Accessibility Center at [http://www.ibm.com/able/](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.
The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning products not produced by this manufacturer was obtained from the suppliers of those products, their published announcements or other publicly available sources. This manufacturer has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to products not produced by this manufacturer. Questions on the capabilities of products not produced by this manufacturer should be addressed to the suppliers of those products.

All statements regarding the manufacturer’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The manufacturer’s prices shown are the manufacturer’s suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to the manufacturer, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. The manufacturer, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

CODE LICENSE AND DISCLAIMER INFORMATION:

The manufacturer grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, THE MANUFACTURER, ITS PROGRAM DEVELOPERS AND SUPPLIERS, MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS THE MANUFACTURER, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

**Trademarks**

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM
developerWorks
PowerPC
PowerPC® Architecture
Resource Link

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Cell Broadband Engine™ and Cell/B.E.™ are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.
Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

**Personal Use:** You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of the manufacturer.

**Commercial Use:** You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of the manufacturer.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any data, software or other intellectual property contained therein.

The manufacturer reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by the manufacturer, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

THE MANUFACTURER MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THESE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the developerWorks® Web site located at:

http://www-128.ibm.com/developerworks/power/cell/

Click on 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
• SPU Assembly Language Specification
• SPU Application Binary Interface Specification
• SIMD Math Library Specification for Cell Broadband Engine Architecture
• Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification

Programming
• Cell Broadband Engine Programming Handbook
• Programming Tutorial
• SDK for Multicore Acceleration Version 3.0 Programmer’s Guide

Library
• SPE Runtime Management library
• SPE Runtime Management library Version 1.2 to Version 2.0 Migration Guide
• Accelerated Library Framework for Cell Programmer’s Guide and API Reference
• Accelerated Library Framework for Hybrid-x86 Programmer’s Guide and API Reference
• Data Communication and Synchronization for Cell Programmer’s Guide and API Reference
• Data Communication and Synchronization for Hybrid-x86 Programmer’s Guide and API Reference
• SIMD Math Library Specification
• Monte Carlo Library API Reference Manual (Prototype)

Installation
• SDK for Multicore Acceleration Version 3.0 Installation Guide
IBM XL C/C++ Compiler and IBM XL Fortran Compiler

Detail about documentation for the compilers is available on the developerWorks Web site.

Draft comment
Should we name the documentation here? What is it?

IBM Full-System Simulator and debugging documentation

Detail about documentation for the simulator and debugging tools is available on the developerWorks Web site.

Draft comment
Should we name the documentation here? What is it?

PowerPC Base

- PowerPC Architecture™ Book, Version 2.02
  - Book I: PowerPC User Instruction Set Architecture
  - Book II: PowerPC Virtual Environment Architecture
  - Book III: PowerPC Operating Environment Architecture
Glossary

**Accelerator**
General or special purpose processing element in a hybrid system. An accelerator might have a multi-level architecture with both host elements and accelerator elements. An accelerator, as defined here, is a hierarchy with potentially multiple layers of hosts and accelerators. An accelerator element is always associated with one host. Aside from its direct host, an accelerator cannot communicate with other processing elements in the system. The memory subsystem of the accelerator can be viewed as distinct and independent from a host. This is referred to as the subordinate in a cluster collective.

**All-reduce operation**
Output from multiple accelerators is reduced and combined into one output.

**cluster**
A collection of nodes.

**Compute kernel**
Part of the accelerator code that does stateless computation task on one piece of input data and generates corresponding output results.

**Compute task**
An accelerator execution image that consists of a compute kernel linked with the accelerated library framework accelerator runtime library.

**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 to the physical processing element in a topology. The de_id is usually assigned (or derived) when the node is powered up. It should not change until the node is powered down again.

**group**
A group construct specifies a collection of DaCS DEs and processes in a system.

**handle**
A handle is an abstraction of a data object; usually a pointer to a structure.

**Host**
A general purpose processing element in a hybrid system. A host can have multiple accelerators attached to it. This is often referred to as the master node in a cluster collective.

**Main thread**
The main thread of the application. In many cases, Cell/B.E. architecture programs are multi-threaded using multiple SPEs running concurrently. A typical scenario is that the application consists of a main thread that creates as many SPE threads as needed and the application organizes them.

**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.
PPE
PowerPC Processor Element. The general-purpose processor in the Cell/B.E. processor.

process
A process is a standard UNIX-type process with a separate 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.

SPMD
Single Program Multiple Data. A common style of parallel computing. All processes use the same program, but each has its own data.

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.

Work block
A basic unit of data to be managed by the framework. It consists of one piece of the partitioned data, the corresponding output buffer, and related parameters. A work block is associated with a task. A task can have as many work blocks as necessary.

Work queue
An internal data structure of the accelerated library framework that holds the lists of work blocks to be processed by the active instances of the compute task.
## Index

### D
- dacs_barrier_wait 29
- dacs_de_start 15
- dacs_de_test 20
- dacs_de_wait 19
- dacs_errhandler_reg 71
- dacs_error_code 74
- dacs_error_de 76
- dacs_error_num 73
- dacs_error_pid 77
- dacs_error_str 75
- dacs_get 40
- dacs_get_list 45
- dacs_get_num_avail_children 11
- dacs_group_accept 27
- dacs_group_add_member 24
- dacs_group_close 25
- dacs_group_destroy 26
- dacs_group_init 23
- dacs_group_leave 28
- dacs_mailbox_read 52
- dacs_mailbox_test 53
- dacs_mailbox_write 51
- dacs_mutex_accept 62
- dacs_mutex_destroy 67
- dacs_mutex_init 60
- dacs_mutex_lock 63
- dacs_mutex_release 66
- dacs_mutex_share 61
- dacs_mutex_try_lock 64
- dacs_mutex_unlock 65
- dacs_num_processes_running 18
- dacs_num_processes_supported 17
- dacs_put 38
- dacs_put_list 42
- dacs_recv 49
- dacs_release_de_list 13
- dacs_remote_mem_accept 34
- dacs_remote_mem_create 32
- dacs_remote_mem_destroy 36
- dacs_remote_mem_query 37
- dacs_remote_mem_release 35
- dacs_remote_mem_share 33
- dacs_reserve_children 12
- dacs_runtime_exit 9
- dacs_runtime_init 8
- dacs_send 48
- dacs_strerror 72
- dacs_test 57
- dacs_wait 58
- dacs_wid_release 56
- dacs_wid_reserve 55
- documentation 97

### S
- SDK documentation 97
- SIGTERM 69

---

E

- error handler 71