Software Development Kit for Multicore Acceleration
Version 3.1

Accelerated Library Framework Programmer's Guide and API Reference

SC33-8333-03

Software Development Kit for Multicore Acceleration
Version 3.1

Accelerated Library Framework Programmer's Guide and API Reference

SC33-8333-03

Note
Before using this information and the product it supports, read the information in "Notices" on page 197.

Edition notice

This edition applies to version 3, release 1, modification 0, of the IBM Software Development Kit for Multicore Acceleration (Product number 5724-S84) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC33-8333-02.

Contents

About this publication

This programmer's guide provides detailed information regarding the use of the Accelerated Library Framework (ALF) APIs. It contains an overview of the ALF, detailed reference information about the APIs, and usage information for programming with the APIs.

This book addresses the ALF implementation for the Cell Broadband Engine™ (Cell/B.E.) architecture and the ALF implementation for Hybrid architecture.

For information about the accessibility features of this product, see Appendix H, "Accessibility features," on page 195.

Who should use this book

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

What's new in this release

ALF includes the following new functionality and changes for the Software Development Kit for Multicore Acceleration Version 3.1 (SDK):

Note: These changes are backward compatible with SDK 3.0.

- Two types of ALF runtime instances are supported:
  - **isolated** ALF runtime instance
  - **shared** ALF runtime instance

  The isolated ALF runtime instance exclusively refers to an ALF instance that does not share resources with other ALF runtime handles (exactly the same as it did in SDK 3.0). The shared ALF runtime instance you can use the shared ALF initialization API **alf_init_shared** to share one live ALF instance within one application process.
- A new task type, the lightweight task, is supported and some APIs have been added or updated to support this functionality.
- There are new APIs for the **alf_wb_sync** and **alf_wb_sync_wait** to allow you to synchronize work block execution.
- There is a prototype ALF for Cell/B.E. extension, which provides a host API for data strided access as a separately installable RPM.
- The **ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_UPDATE** is deprecated and the new API **ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE** is recommended.
- The `library_path` now allows multiple entries separated by colons (':'). An empty-string library path loads the accelerator library from the default operating system library path.
- A NULL accelerator library name searches the currently loaded shared object libraries for the executable image.
- A host debug library is provided to output error messages to aid in debugging.
- The accelerator type **ALF_ACCEL_TYPE_EDP** is added for the eDP Synergistic Processor Unit (SPU) for a IBM PowerXCell™ 8i processor.

- The new task descriptor **ALF_TASK_DESC_NUM_DTL** field type allows you to specify the number of data transfer lists.
- You can now register an error handler on ALF for Hybrid.
- ALF verifies the SPU shared object library version against the PowerPC® Processor Unit (PPU) version.
- There are performance improvements for both speed and space.

Related information

See Appendix G, “Related documentation,” on page 193.

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 IBM Resource Link™ at http://www.ibm.com/servers/resourcelink. Click **Feedback** the navigation pane. Be sure to include the name of the book, the form number of the book, and the specific location of the text you are commenting on (for example, a page number or table number).

Part 1. ALF overview

This topic describes what ALF is and what you can use ALF for.

It covers the following topics:

- A description of what is ALF is, see:
  - Chapter 1, “What is ALF?,” on page 3
  - Chapter 2, “Overview of ALF external components,” on page 5
  - Chapter 4, “Basic structure of an ALF application,” on page 9
- What you can use ALF for, see Chapter 3, “When to use ALF,” on page 7
- ALF-specific concepts, see Chapter 5, “Concepts,” on page 13

Chapter 1. What is ALF?

The Accelerated Library Framework (ALF) provides a programming environment for data and task parallel libraries and applications.

The ALF API provides you with a set of interfaces to simplify library development on heterogenous multi-core systems. You can use the provided framework to offload the computationally intensive work to the accelerators. More complex applications can be developed by combining the several function offload libraries. You can also choose to implement applications directly to the ALF interface.

ALF supports the multiple-program-multiple-data (MPMD) programming model where multiple programs can be scheduled to run on multiple accelerator elements at the same time.

The ALF functionality includes:

- Data transfer management
- Parallel task management
- Double buffering
- Dynamic load balancing for data parallel tasks

With the provided API, you can also create descriptions for multiple compute tasks and define their execution order by defining task dependency. Task parallelism is accomplished by having tasks without direct or indirect dependencies between them. The ALF runtime provides a parallel scheduling scheme for the tasks based on given dependencies.

ALF workload division

From the library or application programmer's point of view, ALF consists of the following two runtime components:

- A host runtime library
- An accelerator runtime library

The host runtime library provides the host APIs to the application. The accelerator runtime library provides the framework and APIs to invoke and support the application's accelerator code, usually the computational kernel and helper routines. This division of labor enables programmers to specialize in different parts of a given parallel workload.

Using a layered approach to develop applications in ALF

The ALF design enables you to break down your application development into layers. From an ALF perspective, there are three distinct layers within a given application:

Application
: You develop programs only at the host level. You can use the provided accelerated libraries without direct knowledge of the inner workings of the underlying system.

Accelerated library
: You use the ALF APIs to implement the internal logic of libraries. Because

ALF is an accelerator-oriented programming model, the common approach you take is to partition the problem into smaller tasks that run on the accelerators. The code that runs on the accelerator is generally called the computing kernel. Based on the type of problem you are trying to solve, you can choose the most appropriate task type or types that ALF supports. You then combine these different tasks to solve the more complex problems the library is required to accomplish, typically through the task dependency descriptions. For a work block task, you then need to partition the input and output into work blocks, which ALF can schedule to run on different accelerators.

Computational kernel
You write optimized accelerator code at the accelerator level. The ALF API provides a common programming framework for the compute task to be invoked automatically. For a work block task, the computational kernel is usually stateless, while for a lightweight task, the computational kernel is actually a "run to end" stateful thread.

ALF runtime framework

The runtime framework handles the underlying task management, data movement, and error handling, which means that the focus is on the kernel and the data partitioning, and not on the management of the task and work queue, or the direct memory access (DMA) list creation or execution

The ALF APIs are platform-independent and their design is based on the fact that many applications targeted for Cell/B.E. or multi-core computing follow the general usage pattern of dividing a set of data into self-contained blocks, creating a list of data blocks to be computed on the synergistic processing element (SPE), and then managing the distribution of that data to the various SPE processes. This type of control and compute process usage scenario, along with the corresponding work queue definition, are the fundamental abstractions in ALF.

The lightweight task type addresses applications that do not fall in to the usage patterns mentioned in the previous paragraph. The lightweight task is more like a multi-threading model for the accelerators.

Chapter 2. Overview of ALF external components

The following topics describe the different types of ALF task.

ALF work block task

Within the ALF work block task, a computational kernel is defined as an accelerator routine that takes a given set of input data, and computes the resulting output data based on the given input, see Figure 1. The corresponding input data and output data are divided into separate portions, called work blocks. For a single task, ALF allows these work blocks to be processed in parallel.

ALF lightweight task

The ALF lightweight task differs from the work block task. When a lightweight task is scheduled to run, ALF rumtime gives the control to the task instances running on the accelerators. The task instances then run until they finish processing and return the control to ALF runtime. The task ends when all task instance return to ALF.

Task dependency

With the provided ALF API, you can also create descriptions for multiple compute tasks, and define their execution order by defining their dependencies. Task parallelism is accomplished by having tasks without direct or indirect dependencies between them. The ALF runtime provides a parallel scheduling scheme for the provided tasks based on the given dependencies, which can schedule tasks based on their dependencies.

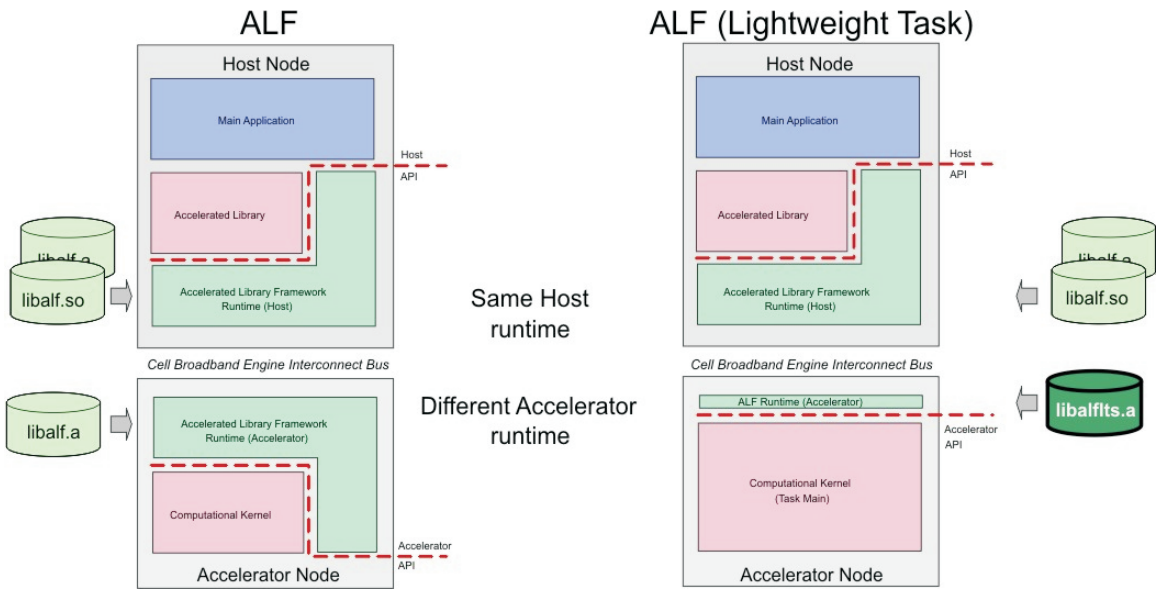


Figure 1. Overview of ALF

Chapter 3. When to use ALF

ALF is designed to help you to develop solutions to robust data parallel problems and task parallel problems.

The following problem types are well suited to work on ALF:

- **Computationally intensive data-parallel problems**: The ALF work block task API is designed to support data-parallel problems with the following characteristics:
  - Most of the parallel work focuses on performing operations on a large data set. The data set is typically organized into a common data structure, for example, an array of data elements.
  - A set of accelerators work collectively on the same data set, however, each accelerator works on a different partition of the data set. For ALF, the data set does not have to be regularly partitioned. Any accelerator can be set to work on any part of the data set.
  - The programs on the accelerators usually perform the same task on the data set.
- **Task-parallel problems**: The ALF API supports multiple tasks running on multiple accelerators at the same time. You can divide your application into subproblems and create one task for each subproblem. The ALF runtime determines how to schedule the multiple tasks on the available accelerators to obtain the highest degree of parallelism.

Certain problems can seem to be inherently serial at first; however, there might be alternative approaches to divide the problem into subproblems, and one or all of the subproblems can often be parallelized.

You need to be aware of the physical limitations on the supported platforms. For example, for the BladeCenter® QS21 and BladeCenter QS22 implementations, the SPE has the local memory size of 256 KB. If the data set of the problem cannot be divided into work blocks that fit into local storage, then ALF cannot be used to solve that problem.

Chapter 4. Basic structure of an ALF application

This topic describes what you need to do to run an ALF application.

The basic structure of an ALF application is shown in Figure 2 on page 10. The process on the host is as follows:

1. Initialize the ALF runtime.
2. Create a compute task.
3. If it is a work block task, after the task is created, you start to add work blocks to the work queue of the task.
4. Mark the task as finalized
5. Wait for the task to process all the work blocks, complete and shut down the ALF runtime to release the allocated resources.

For a work block task, the process on the accelerator is as follows:

1. After an instance of the task is spawned, it waits for work blocks to be added to the work queue.
2. The `alf_accel_comp_kernel` function is called for each work block.
3. If the task has been created with a task descriptor with `ALF_TASK_DESC_PARTITION_ON_ACCEL` set to 1, then the `alf_accel_input_dtl_prepare` function is called before the invocation of the compute kernel and the `alf_accel_output_dtl_prepare` function is called after the invocation of the compute kernel.

For a lightweight task, the process on the accelerator is as follows:

1. After an instance of the task is spawned, the accelerator side ALF runtime calls the `alf_accel_lts_main`.
2. The `alf_accel_lts_main` function runs until the task has completed processing. Then it returns to the caller (ALF runtime).

For examples of ALF applications including some source code samples, see Appendix C, “Examples,” on page 167.

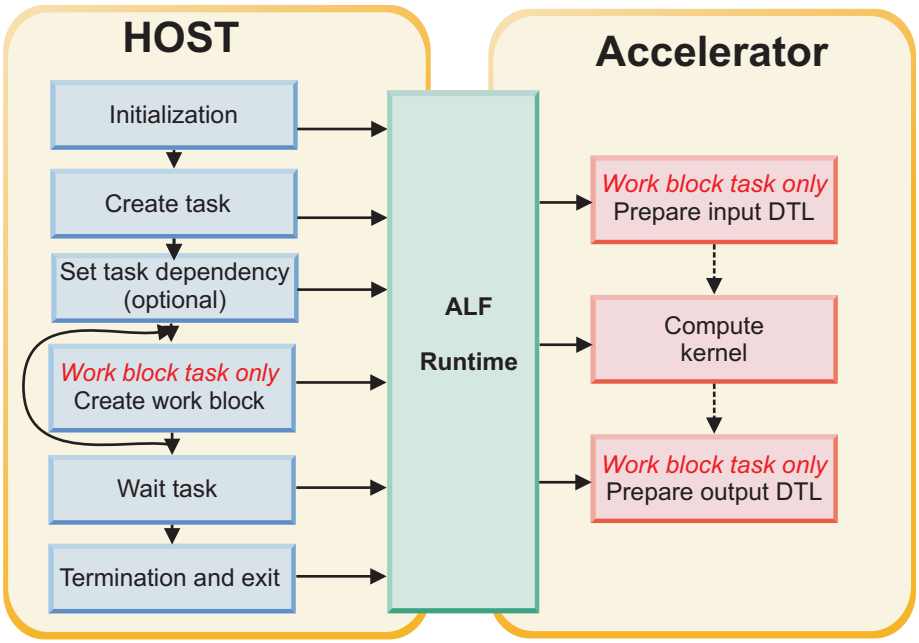


Figure 2. ALF application structure and process flow

Simple example

The following example shows a simple ALF application. The host application initializes the ALF runtime, creates a task descriptor and a task associated with that descriptor, then it waits for the task to complete, and finally exits the ALF runtime.

On the accelerator side, the computational kernel prints "Hello World" to stdout.

Source code for the host application

```
#include <stdlib.h>
#include <alf.h>

int main(int argc __attribute__ ((unused)), char *argv[] __attribute__ ((unused)))
{
alf_handle_t handle;
alf_task_desc_handle_t task_desc_handle;
alf_task_handle_t task_handle;
alf_wb_handle_t wb_handle;

alf_init(NULL, &handle);
alf_num_instances_set(handle, 1);
alf_task_desc_create(handle, 0, &task_desc_handle);
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REF_L,
(unsigned long long) "alf_hello_world_cell_spu.so");
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_IMAGE_REF_L,
(unsigned long long) "alf_hello_world_spu");
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_KERNEL_REF_L, (unsigned long long) "comp_kernel");
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_MAX_STACK_SIZE, 4096);
alf_task_create(task_desc_handle, NULL, 1, 0, 0, &task_handle);
alf_wb_create(task_handle, ALF_WB_SINGLE, 1, &wb_handle);
alf_wb_enqueue(wb_handle);
alf_task_finalize(task_handle);
alf_task_wait(task_handle, -1);
alf_exit(handle, ALF_EXIT_POLICY_FORCE, 0);

return 0;
}
```

Source code for the accelerator side

```
#include <stdio.h>
#include <alf_accel.h>

int comp_kernel(void *p_task_context __attribute__ ((unused)), void *p_parm_context __attribute__ ((unused)),
void *p_input_buffer __attribute__ ((unused)), void *p_output_buffer
__attribute__ ((unused)), void *p_inout_buffer __attribute__ ((unused)), unsigned int current_count
__attribute__ ((unused)), unsigned int total_count __attribute__ ((unused)))
{
printf("Hello World!\n");
return 0;
}

ALF_ACCEL_EXPORT_API_LIST_BEGIN;
ALF_ACCEL_EXPORT_API("", comp_kernel);
ALF_ACCEL_EXPORT_API_LIST_END;
```

Chapter 5. Concepts

The following sections explain the main concepts and terms used in ALF.

Computational kernel

A computational kernel is a user-defined accelerator routine that does the computation for a corresponding task.

For a work block task, the execution of the computational kernel takes a given set of input data, processes it, and returns the resulting output. For a lightweight task, the behavior of the computational kernel is user-defined, that is the kernel is responsible for reading its own input data from main storage, processing it into output data, and writing the output data back to the main storage.

You should implement the computational kernel according to the ALF computational kernel function prototype definitions with data in specific buffers (see Chapter 7, "Accelerator buffer management," on page 31. Computational kernels must be registered to the ALF runtime when the corresponding task descriptor is created.

For a work block task, the computational kernel is usually accompanied by four other auxiliary functions. The computational kernel and the four auxiliary functions form a 5-tuple for a task represented as:

```
{
alf_accel_comp_kernel,
alf_accel_input_dtl_prepare,
alf_accel_output_dtl_prepare,
alf_accel_task_context_setup,
alf_accel_task_context_merge
}
```

For a lightweight task, there is only one function for the computational kernel and no auxiliary functions:

```
{
alf_accel_lts_main
}
```

Note: The above accelerator function names are used as conventions for this document only. You can provide your own function name for each of these functions and register the function name through the task descriptor service.

Based on the different application requirements, some of the auxiliary functions for a work block task need not be defined, that is they can be NULL.

For more information about the APIs that define computational kernels, see "User-provided computational kernel APIs" on page 133.

Task descriptor

A task descriptor contains the descriptive information to create and manage a task.

The task descriptor also contains information about the names of the different user-defined accelerator functions. To maximize accelerator performance, ALF statically allocates memory on an accelerator for each task that is executing. This means that ALF requires you to provide information about buffers, stack usage, and the number of data transfer list entries ahead of time. Finally, it contains the data partition attribute setting.

For the lightweight task, You must manage the accelerator-side memory, ALF does not manage memory, such as buffers, or data transfer lists, except for stack usage and the task context buffer.

The following information is used to define a task descriptor:

- Accelerator executable image that contains the computational kernel:
  - For work block tasks:
    - The name of the accelerator computational kernel function
    - Optionally, the name of the accelerator input data transfer list prepare function
    - Optionally, the name of the accelerator output data transfer list prepare function
    - Optionally, the name of the accelerator task context setup function
    - Optionally, the name of the accelerator task context merge function
  - For lightweight tasks:
    - The name of the accelerator entry point function
- The following only applies to the work block task:
  - Work block parameter context buffer size
  - Work block input buffer size
  - Work block output buffer size
  - Work block overlapped buffer size
  - Work block number of data transfer list entries
  - Task data partition attribute:
    - Partition on accelerator
    - Partition on host
- Accelerator stack size
- Task context description
  - Task context buffer size
  - Task context entries: entry size, entry type

For more information about the compute task APIs, see "Compute task API" on page 88.

Task

A task is a program that is executed on a number of accelerators. For each accelerator that runs a task, a task instance is spawned.

A task is defined as a ready-to-be-scheduled instantiation of a task description. The number of instances is set by the `num_instances` parameter in the task creation function (`alf_task_create`) which explicitly requests a number of accelerators or by the ALF runtime which implicitly decides the necessary number of instances to create to run the compute task

For the work block task, after you have created the task, you can create work blocks and enqueue them on to the working queue associated with the task. For the work block task, the ALF framework employs an immediate runtime mode. This means that the task becomes ready to run before finalization if the following conditions are met:

- At least one work block has been enqueued
- The task has no unresolved dependency on other unfinished tasks
- Accelerator resources are available

For information about work blocks, see "Work blocks" on page 18.

For a lightweight task, the task does not become ready to run until it is finalized. An ALF library or application can use a combination of both work block tasks and lightweight tasks.

You can register an event handler to monitor different task events, see "Task events" on page 17.

Task finalize

For a work block task, after you have finished adding work blocks to the work queue, or for a lightweight task, after you have defined all the task attributes, you must call `alf_task_finalize` function to notify ALF that there are no more work blocks or information updates to this particular task. A work block task that is not "finalized" cannot be run to completion.

Task dependency and task scheduling

Task dependency in the ALF programming model makes sure multiple tasks are run in a specific order when that order is critical.

Some common dependency scenarios include:

- Data dependency: where the output of one task is the input of another task
- Resource conflict: where the tasks share some common resources such as temporary buffers
- Execution sequencing: where the tasks have run in a predetermined order

After you have created a task, you can use the function `alf_task_depends_on` to specify the task's dependency relationship with another existing task. The ALF runtime uses a task's dependency and the number of requested accelerators when it schedules a task.

The ALF framework does not detect circular dependencies. For a task that depends on other tasks, you must define the dependencies before any work block is added to the task. If a circular dependency is present in the task definitions, the application hangs when it is invoked.

A task that depends on other tasks cannot be processed until all the dependent tasks finish. Tasks are created in immediate mode. After a task has been created and its dependencies are satisfied, the task is scheduled to run.

For an example of how to use task dependency, see "Task dependency example" on page 179.

Task instance

A task can be scheduled to run on multiple accelerators. Each task running on an accelerator is known as a task instance. If a task is created without the `ALF_TASK_ATTR_SCHED_FIXED` attribute for the work block task, the ALF runtime may load and unload an instance of a task to and from an accelerator at anytime. If a work block task is created without the attribute, the ALF runtime can select to adjust the number of task instances at runtime if one of the following conditions is met:

- The task is started for the first time
- During the task execution new accelerator resources become available
- The runtime decides to reduce the amount of accelerator resources that are allocated to the running task. This is only possible for the selected task instance if the following criteria are met:
  - Has completed the write-out of the previous work block's result
  - Not during a multiuse work block
  - Not during an incomplete group of bundled work blocks

When a new task instance is started, the runtime initializes its copy of task context with the default values defined by the host buffer and calls the `alf_accel_task_context_setup`. When a task instance is stopped, the runtime invokes `alf_accel_task_context_merge` to merge the current copy of task context with the context of a running instance.

For a lightweight task, the ALF runtime starts as many instances of the task as possible at the beginning of task execution and it does not adjust the number of instances unless the corresponding instances return from the computing kernel.

The ALF runtime posts an event after a task instance is started on an accelerator or unloaded from an accelerator. You can choose to register an event handler for these events, see "Task events" on page 17.

Fixed task mapping

When the application scenario requires, you can explicitly require the runtime to start a fixed number of task instances for a specific task. This is known as fixed task mapping. To do this, you need to :

1. Provide the number of task instances at task creation time through the `alf_task_create` interface
2. Set the `ALF_TASK_ATTR_SCHED_FIXED` task attribute

In this case, the ALF runtime ensures that the given number of task instances is started and does not change this number until the task ends.

For work block tasks, fixed mapping is usually used only with the cyclic work block distribution policy to make sure work blocks are assigned in a specified order. Other than in the case of cyclic work block distribution, it is not recommended to use fixed mapping for work block tasks as this can limit the runtime's capability to dynamically map the task to available resources.

For lightweight tasks, fixed mapping is needed when the application has been tuned to a specific number of task instances.

Task context

This topic describes when to use a task context.

Note: For more information, refer to "Buffer types" on page 31.

A task context is used to address the following usage scenarios:

Common data across work blocks and different task instances

A task context can be used to contain common for all work blocks in a work block task or as common initial values for all task instances of a lightweight task. This is useful for static input data, lookup tables, or any other input data that is common to all task instances. Because the ALF runtime loads the task context to accelerator memory before any work block is processed or lightweight task is executed, you are assured that the common data is always there for the kernel's use.

Reducing partial results across work blocks

For work block tasks, a task context can be used to incrementally aggregate and update the final result of a task based on each work block's computation. For these applications, the computational results of separate work blocks are the intermediate results. These intermediate results are stored in the task context. You can update the task context in the computational kernel as part of the work block computation. After all the work blocks have been processed, the ALF runtime applies a reduction step to merge the intermediate results of the task instances into a single final result using a user-provided `alf_accel_task_context_merge` function.

For an example about how to apply the concept of task context to find the maximum value or the minimum value of a large data set, see "Min-max finder example" on page 172.

Tip: ALF does not make a copy of the corresponding host-side buffer of the task context. This buffer is only accessed when the task is actually run, you need to make sure the host-side buffer is available for use.

Task events

The ALF framework provides notifications for task events.

These task events are:

- `ALF_TASK_EVENT_READY` - the task is ready to be scheduled
- `ALF_TASK_EVENT_FINALIZED` - all the work blocks for the task have been enqueued `alf_task_finalized` has been called
- `ALF_TASK_EVENT_INSTANCE_START` - a new instance of the task has started
- `ALF_TASK_EVENT_INSTANCE_END` - one instance of the task has ended
- `ALF_TASK_EVENT_FINISHED` - the task has finished running

- `ALF_TASK_EVENT_DESTROY` - the task is destroyed explicitly

For information about how to set event handling, see `alf_task_event_handler_register`.

Work blocks

This topic describes what a work block is and the different work block scheduling policies.

A work block represents an invocation of a computational kernel of a work block task with a specific set of related input data, output data, and parameters. The input and output data are described by corresponding data transfer lists. The parameters are provided through the ALF APIs. Depending on the application, the data transfer list can either be generated on the host (host data partition) or on the accelerators (accelerator data partition).

The ALF accelerator runtime processes a work block by retrieving from host memory its parameters and its input data, using its input data transfer list, into buffers within accelerator memory. It then calls the computational kernel to process the input data into the output data. Afterwards it puts the output data back into the host memory. The ALF accelerator runtime manages the buffers within the accelerator's memory to accommodate each work block's input and output data.

Single-use work block

A single-use work block is processed only once by the ALF accelerator runtime. A single-use work block gives you the option of generating input and output data transfer lists for the work block on either the host or the accelerator. A single-use work block invokes the computational kernel only once.

Multi-use work block

A multi-use work block is repeatedly processed up to the specified iteration count which invokes the computational kernel each time. Unlike using a single-use work block, a multi-use work block does not allow you to generate input and output data transfer lists from the host process with host partitioning. For multi-use work blocks, all input and output data transfer lists must be generated on the accelerators using accelerator partitioning for each iteration that a work block is processed by the ALF runtime. For each iteration of the multi-use work block, the ALF runtime passes the parameters, total number of iterations, and current iteration count to the accelerator data partition subroutines, and you can generate the corresponding data transfer lists for each iteration based on this information. See "Accelerator data partitioning" on page 29 for more information about single-use work blocks and multi-use work blocks.

Data transfer list

Data transfer lists describe a work block's input and output data.

You can choose to generate the data transfer lists for each task's work blocks' input and output data on either the host or the accelerator.

For many applications, the input data for a single compute kernel cannot be stored contiguously in the host memory. For example, in the case of a multi-dimensional matrix, the matrix is usually partitioned into smaller sub-matrices for the accelerators to process. For many data partitioning schemes, the data of the

sub-matrices is scattered to different host memory locations. Accelerator memory is usually limited, and the most efficient way to store the submatrix is contiguously. Data for each row or column of the submatrix is put together in a contiguous buffer. For input data, they are gathered to the local memory of the accelerator from scattered host memory locations. With output data, the above situation is reversed, and the contiguous data in the local memory of the accelerator is scattered to different locations in host memory.

The ALF API uses data transfer list to represent the scattered input and output data in the host memory. A data transfer list contains entries that consist of the data size and a pointer to the host memory location of the data. The data in the local accelerator memory is always packed and is organized in the order of the entries in the list. For input data, the data transfer list describes a data gathering operation. For output data, the data transfer list describes a scattering operation. See Figure 3 for a diagram of a data transfer list.

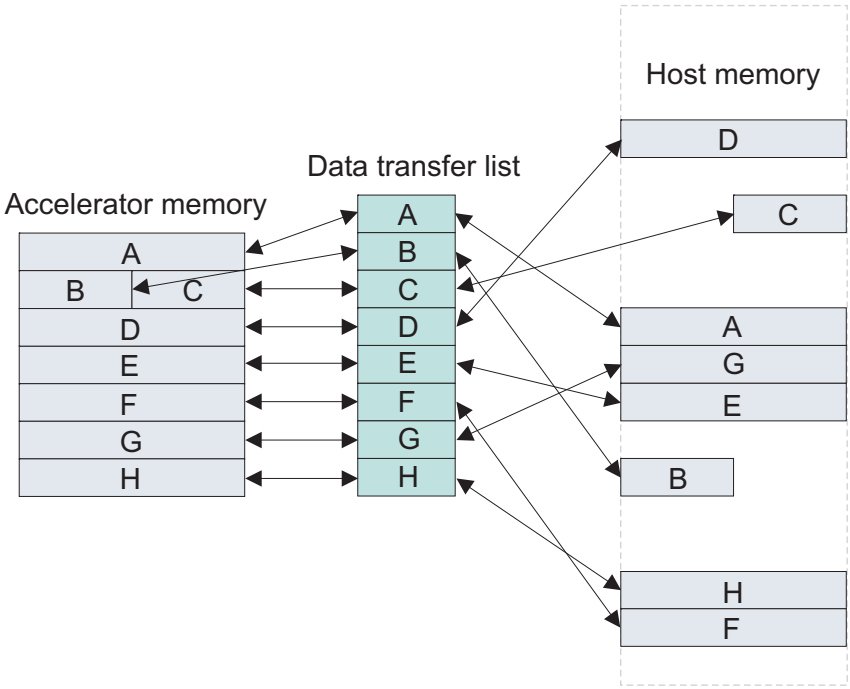


Figure 3. Data transfer list

To maximize accelerator performance, ALF employs a static memory allocation model per task execution on the accelerator. This means programmers need to explicitly specify the maximum number of data transfer lists, and the maximum number of entries in a data transfer list that a task can have. This can be set through the `alf_task_desc_set_int32` function with the `ALF_TASK_DESC_NUM_DTL_ENTRIES` function.

For information about data transfer list limitations for Cell/B.E. implementations, see "Data transfer list limitations" on page 60.

Work block scheduling

This section describes work block scheduling.

It describes the following types or work block scheduling policies:

- "Default work block scheduling policy" on page 20

- "Cyclic work block distribution policy"
- "Bundled work block distribution policy" on page 21

Default work block scheduling policy

The ALF API supports multiple ways of assigning work blocks to task instances. ALF starts a variable number of task instances up to the number provided in the `alf_task_create` function. By default, enqueued work blocks are assigned to any task instance in any order.

The ALF runtime tries to balance the load of the task instances to ensure that the task can complete in the shortest time. This means that task instances that start early or run faster may process more work blocks than those that start later or run slower. In other words, ALF schedules work blocks on a variable number of task instances in a non-cyclic manner.

Figure 4 shows an example of the default work block scheduling policy where task instances process work blocks at different rates. Task Instance 1 processes work blocks faster than Task Instance 2, which processes faster than Task Instance 3.

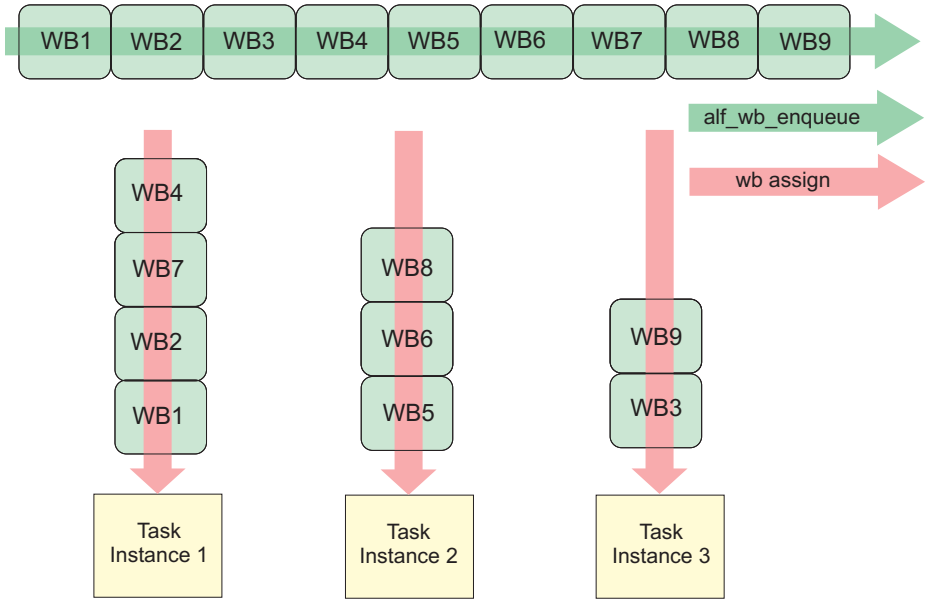


Figure 4. Default work block scheduling behavior

Cyclic work block distribution policy

This topic describes how the cyclic work block distribution policy works and what you need to do to enable it.

You can enable cyclic work block distribution by setting the attributes `ALF_TASK_ATTR_WB_CYCLIC` and `ALF_TASK_ATTR_SCHED_FIXED` when you create the task. These attributes enable the work blocks to be assigned in a round robin order to a fixed number of task instances.

This kind of work block distribution is usually used when you want to make sure the work blocks are assigned evenly and in order to task instances. Because work block *n* and work block *n* + (number of task instance) are guaranteed to be assigned to the same task instance in order, it is possible to allow the two work

blocks to have some shared data or states. For an alternative solution about how to assign work blocks with shared data or contexts to one task instance, refer to Figure 6 on page 22.

You must provide the number of task instances in the `alf_task_create` function. ALF starts this fixed number of task instances. The work blocks are assigned to the task instances in a cyclical manner in the order of the work blocks being enqueued through calling the function `alf_wb_enqueue`. Figure 5 shows an example of cyclic work block distribution. There are three task instances requested and started. One work block at a time is assigned. Work block 1 is assigned to Task Instance 1, work block 2 is a assigned to Task Instance 2, Work block 3 is assigned to Task Instance 3, and the cycle starts again with Task Instance 1.

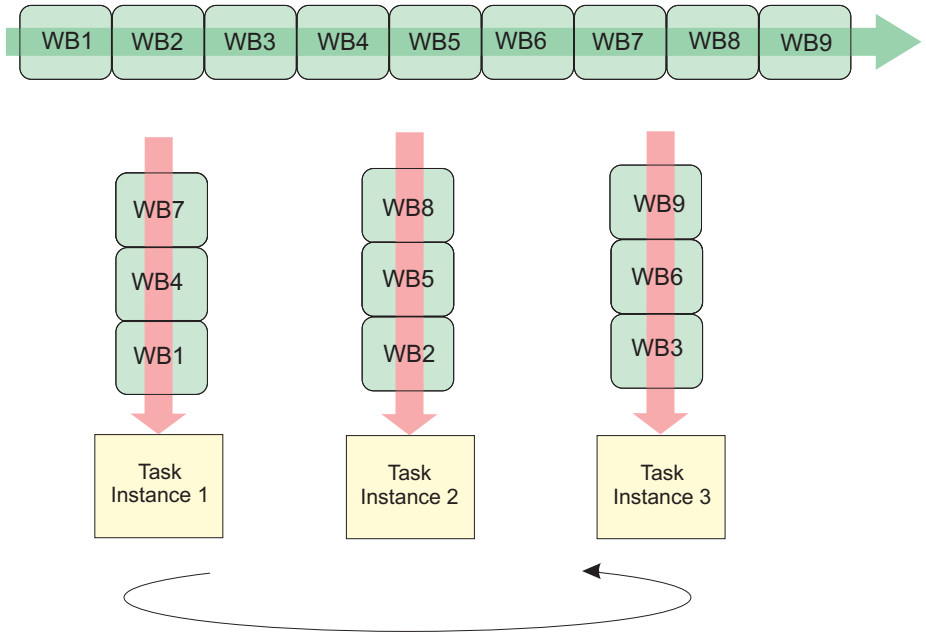


Figure 5. Cyclic work block distribution

Bundled work block distribution policy

This topic describes how the bundled work block distribution policy works and what you need to do to enable it.

The work blocks are assigned to the task instances in a group of bundle_size at a time. They are assigned in the order of the work blocks being enqueued through calling the function `alf_wb_enqueue`. All work blocks in a bundle are assigned to one task instance, and the order defined in `alf_wb_enqueue` is preserved.

Bundled work block distribution is useful if you want to make sure the group of work blocks can have shared context and be processed in order. One such example is when the data partition of one work block cannot cover a large block of related data because of local memory size limitations.

When you call `alf_task_create` to create the task you use the parameter `wb_dist_size` to specify the bundle size. Bundled distribution can also be used together with the cyclic distribution to further control the work block scheduling. Figure 6 on page 22 shows an example of the bundled distribution policy where task instances process two work blocks at a time at different rates, that is in a

non-cyclic manner.

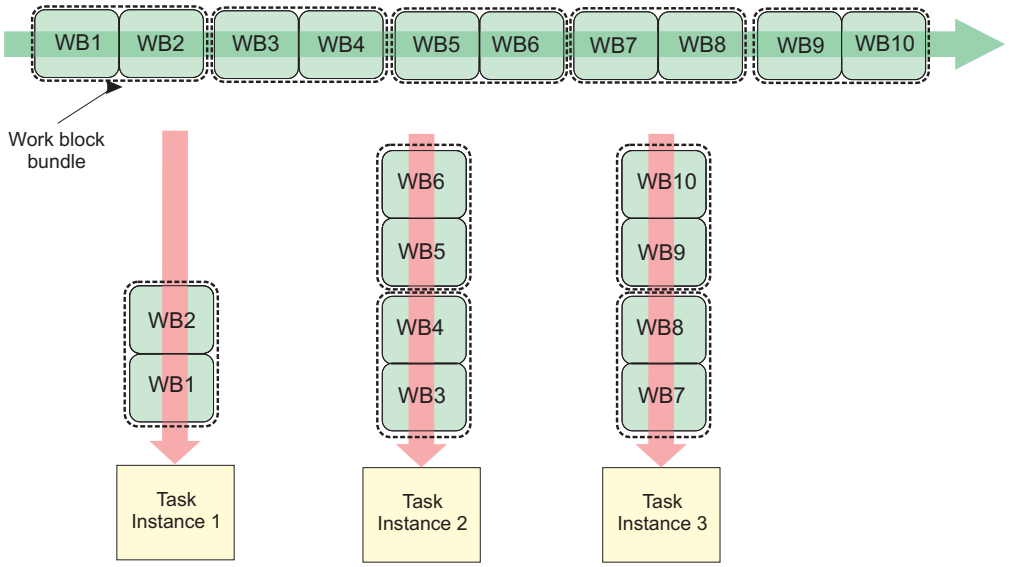


Figure 6. Bundled work block distribution

Data set

An ALF data set is a logical set of data buffers. A data set provides information to the ALF runtime about all the data specific to a task's work block. The ALF runtime uses this information to optimize how data is moved from the host's memory to the accelerator's memory and back.

Figure 7 shows how data sets are used on a hybrid system.

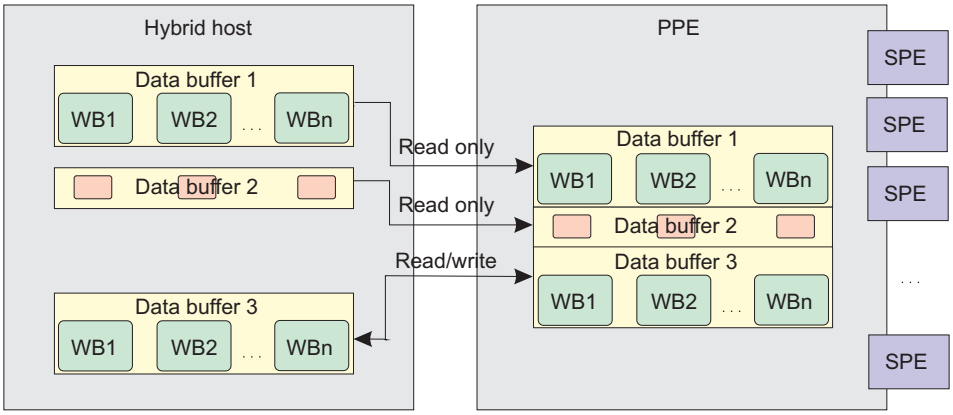


Figure 7. How data sets are used

The process for using data sets is as follows:

1. Set up a data set independently of tasks or work blocks. To do this, you use the `alf_dataset_create`, and `alf_dataset_buffer_add` functions.
2. Associate the data set to one or more tasks. To do this, use the `alf_task_dataset_associate` function.
3. As work blocks are enqueued, they are checked against the associated data set to ensure they reside within one of the buffers.

4. After you have finished with the data set, you destroy it by using the `alf_dataset_destroy` function.

A data set can have a set of data buffers associated with it. A data buffer can be identified as read-only, write-only, or read-write. You can add as many data buffers to the data set as needed. Different ALF implementations may limit the number of data buffers in a specific data set. Refer to the implementation documentation for restriction information about the number of data buffers in a data set. However, after a data set has been associated with a task, you cannot add additional data buffers to the data set.

A task can optionally be associated with one and only one data set. Work blocks for this task refer to input, output and in-out data within the data set for read-only, write-only and read-write buffers respectively. Only references to work block input, output, and in-out data which is outside of the data set result in an error. The task context buffer and work block parameter buffer do not need to reside within the data set and are not checked against it.

Multiple tasks can share the same data set. It is your responsibility to make sure that the data in the data set is used correctly. If two tasks with no dependency on each other use the same data from the same data set, ALF cannot guarantee the consistency of the data. For tasks with a dependency on each other and which use the same data set, the data set gets updated in the order in which the tasks are run.

For host data partitioning you should not create and use data sets. For accelerator data partitioning you must create and use data sets.

For an example of how to use data sets, see "Data set example" on page 181.

Error handling

This topic describes the ALF error handling capabilities.

ALF supports the capability to handle runtime errors. Upon encountering an error, the ALF runtime frees up resources, then exits by default. To enable you to handle errors in a more graceful manner, you can register a callback error handler function with the ALF runtime. Depending on the type of error, your error handler function can direct the ALF runtime to retry the current operation, stop the current operation, or shut down. These are controlled by the return values of your error handler function.

When several errors happen in a short time or at the same time, the ALF runtime attempts to invoke the error handler in sequential order.

Possible runtime errors include the following:

- Compute task runtime errors such as bus error, undefined computing kernel function names, invalid task execution images, memory allocation issues, dead locks, and others
- Detectable internal data structure corruption errors, which might be caused by improper data transfers or access boundary issues
- Application detectable/catchable errors

Error codes are used for return values when an error occurs. For this implementation, the ALF header file, `errno.h`, is used. See Appendix F, "Error

codes and descriptions," on page 191 and also the API definitions in Chapter 19, "ALF API overview," on page 65 for a list of possible error codes.

Part 2. Installing and configuring ALF packages

The ALF library can be installed as a component of the SDK.

Refer to the *SDK Installation Guide* for more information about installation and configuration.

The following packages are provided for the ALF for Cell/B.E. library.

Table 1. ALF for Cell/B.E. packages

Package	Description
`alf-*.ppc.rpm`	32-bit ALF for Cell/B.E. runtime package - contains the optimized shared library for the host.
`alf-devel-*ppc.rpm`	32-bit ALF for Cell/B.E. development package - contains all the header files for the host, and the static host runtime library.
`alf-spu-devel-*.ppc.rpm`	32-bit ALF for Cell/B.E. development package - contains all the header files for the accelerator, the static accelerator runtime library, and the error check enabled static accelerator library. **Note:** This package is required by both the alf-devel-*.ppc.rpm and the alf-devel-*.ppc64.rpm files.
`alf-trace-*.ppc.rpm`	32-bit ALF for Cell/B.E. trace-enabled package - contains the optimized shared library with PDT debug and trace enabled.
`alf-trace-devel-*.ppc.rpm`	32-bit ALF for Cell/B.E. development package with PDT trace - contains the host static library with PDT debug and trace enabled.
`alf-spu-trace-devel-*.ppc.rpm`	32-bit ALF for Cell/B.E. development package with PDT trace - contains the accelerator static library with PDT debug and trace enabled. **Note:** This package is required by both the alf-trace-devel-*.ppc.rpm and the alf-trace-devel-*.ppc64.rpm files.
`alf-*.ppc64.rpm`	64-bit ALF for Cell/B.E. runtime package - contains the optimized host shared library.
`alf-devel-*.ppc64.rpm`	64-bit ALF for Cell/B.E. development package - contains all the header files for the host and host static library.
`alf-trace-*.ppc64.rpm`	64-bit ALF for Cell/B.E. trace-enabled package - contains the optimized host shared library with PDT debug and trace enabled.
`alf-trace-devel-*.ppc64.rpm`	64-bit ALF for Cell/B.E. development package with PDT trace - contains the host static library with PDT debug and trace enabled.
`alf-cross-devel-*.noarch.rpm`	ALF for Cell/B.E. cross development package - contains all the header files and libraries needed for cross-architecture development.
`alf-examples-source-*.noarch.rpm`	ALF for Cell/B.E. example sources.
`alf-manpages-*.noarch.rpm`	ALF for Cell/B.E. man pages.

Table 1. ALF for Cell/B.E. packages (continued)

Package	Description
`alf_compat-*.ppc.rpm`	32-bit ALF for Cell/B.E. compatibility package - contains runtime libraries needed for running the previous ALF release.
`alf_compat-*.ppc64.rpm`	64-bit ALF for Cell/B.E. compatibility package - contains runtime libraries needed for running the previous ALF release.

For the ALF for Cell/B.E. library, there are also debuginfo versions associated with most of the above packages except `alf-cross-devel`, `alf-examples-source`, and `alfman`.

The following packages are provided for the ALF for Hybrid library.

Table 2. ALF for Hybrid packages

Package	Description
`alf-hybrid-*.x86_64.rpm`	ALF for Hybrid runtime package for x86_64 - contains the optimized ALF host shared library.
`alf-hybrid-*.ppc64.rpm`	ALF for Hybrid runtime package for PPC64 - contains the ALF PPE daemon program.
`alf-hybrid-devel-*.x86_64.rpm`	ALF for Hybrid development package for x86_64 - contains the header files, the optimized static x86_64 host static library, and the x86_64 error-checking enabled library.
`alf-hybrid-spu-devel-*.ppc.rpm`	ALF for Hybrid development package for PPC - contains the accelerator static library, and the error-checking enabled accelerator static library.
`alf-hybrid-trace-*.ppc64.rpm`	ALF for Hybrid trace-enabled package for PPC64 - contains the trace-enabled PPE daemon program.
`alf-hybrid-trace-*.x86_64.rpm`	ALF for Hybrid- trace-enabled runtime for the x86_64 - contains the traced-enabled host shared library.
`alf-hybrid-trace-devel-*.x86_64.rpm`	ALF for Hybrid trace-enabled development package for x86_64 - contains the trace-enabled host shared and static library.
`alf-hybrid-spu-trace-devel-*.ppc.rpm`	ALF for Hybrid trace-enabled development package for PPC- contains the trace-enabled accelerator static library.
`alf-hybrid-cross-devel-*.noarch.rpm`	ALF for Hybrid Cross development package - contains the header files and libraries needed for cross-architecture development.
`alf-hybrid-examples-source-*.noarch.rpm`	ALF for Hybrid example sources.
`alf-manpages-*.noarch.rpm`	ALF for Hybrid man pages.

Part 3. Programming with ALF

This section describes how to program with ALF.

The following topics are described.

Note: The capabilities described in the following sections are provided by ALF and only apply to work block tasks. For lightweight tasks the capabilities are not provided by ALF, instead it is the your responsibility to provide these capabilities.

For configuration information including how to switch compilers, see the `alf/README_alf_samples` file.

Chapter 6. Data partitioning

An important part to solving data parallel problems using multiple accelerators is to figure out how to partition data across the accelerators. The ALF API does not automatically partition data, however, it does provide a framework so that you can systematically partition the data.

The ALF API provides the following different data partition methods:

- "Host data partitioning"
- "Accelerator data partitioning"

These methods are described in the following sections.

Host data partitioning

You can use the provided APIs on the host to partition your application's data.

To do this, you build a data transfer list for the work blocks thru the `alf_wb_dtl_begin`, `alf_wb_dtl_entry_add`, and `alf_wb_dtl_end` APIs.

This method is particularly useful when the data associated with the work blocks is simple, and the host has sufficient compute resources to efficiently generate the data partitioning information for all the accelerators.

Accelerator data partitioning

When the data partition scheme is complex and requires significant computing resources, it can be more efficient to generate the data transfer lists on the accelerators. This is especially useful if the host computing resources can be used for other work or if the host does not have enough computing resources to compute data transfer lists for all of its work blocks.

Accelerator data partition APIs

For accelerated partitioning, accelerated library developers must provide the `alf_accel_input_dtl_prepare` function and the `af_accel_output_dtl_prepare` function to do the input and output data partitioning and to generate the associated data transfer lists. The `alf_accel_input_dtl_prepare` is the input data partitioning function and the `alf_accel_output_dtl_prepare` is the output data function.

Host memory addresses

Because the host does not generate the data transfer lists when using accelerator data partitioning, the host addresses of input and output data buffers can be explicitly passed to the accelerator through the task context or the work block parameter.

For an example, see "Matrix add - accelerator data partitioning example" on page 170

Chapter 7. Accelerator buffer management

This topic describes when to use the different types of buffers.

On the accelerator, the ALF accelerator runtime manages the data of the work blocks and the task for the compute kernel. You only need to focus on the organization of data and the actual computational kernel. The ALF accelerator runtime handles buffer management and data movement. However, it is still important that you have an understanding of how each buffer is used and its relationship with the computational kernel.

To make the most efficient use of accelerator memory, the ALF runtime needs to know the memory usage requirements of the task. The ALF runtime requires that you specify all of the memory resources each task uses. The runtime can then allocate the requested memory for the task when it is executed.

Buffer types

The ALF accelerator runtime code provides buffers for each task instance.

The following buffers are available:

- "Task context buffer"
- "Work block parameter buffer" on page 32
- "Work block input data buffer" on page 32
- "Work block output data buffer" on page 33
- "Work block overlapped input and output data buffer" on page 33

Task context buffer

A task context buffer is used by applications that require common data that can be referenced and updated by all work blocks. It is also useful for merging operations or all-reduce operations. A task context is optionally associated with a task. You specify the size of the task context buffer through the task descriptor. If the size of the task context buffer is specified as zero (0) in the task descriptor, there is no task context buffer associated with the any of the tasks created with that task descriptor.

The lifecycle of the task context is shown in Figure 8 on page 32. To create the task, you call the task creation function `alf_task_create`. You provide the data for the initial task context by passing a data buffer with the initial values. After the compute task has been scheduled to be run on the accelerators, the ALF framework creates a local copy of the task context for each task instance that is running.

You can provide a function to initialize the local task context (`alf_accel_task_context_setup`) on the accelerator. The ALF runtime invokes this function when the running task instance is first loaded on an accelerator as shown in Figure 8 on page 32 (a).

All work blocks that are processed by one task instance share the same local copy of task context on that accelerator as shown in Figure 8 on page 32 (b).

When the ALF scheduler requests an accelerator to unload a task instance, you can provide a merge function (`alf_accel_task_context_merge`), which is called by the runtime, to merge that local task context with the local task context of another accelerator as shown in Figure 8 (c).

When a task is shut down and all instances of the task are destroyed, the runtime automatically calls the merge function on the task instances to merge all of the local task contexts into a single task context and write the final result to the task context on host memory provided when the task is created, as shown in Figure 8 (d).

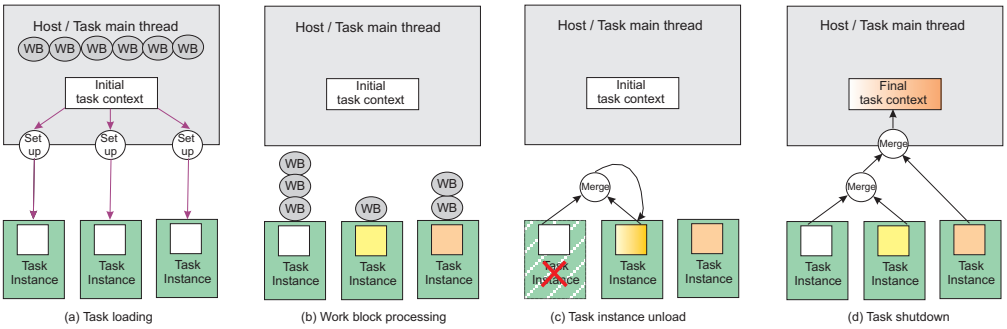


Figure 8. Task context buffer lifecycle

Work block parameter buffer

The work block parameter buffer serves two purposes:

- It passes work block-specific constants or reference-by-value parameters
- It reserves storage space for the computational kernel to save and communicate the data specific to one work block across the following APIs, which can be either a single-use work block or a multi-use work block

This buffer can be used by the following APIs:

- `alf_accel_comp_kernel`
- `alf_accel_input_dtl_prepare`
- `alf_accel_output_dtl_prepare`

The parameters are copied to an internal buffer associated with the work block data structure in host memory when the `alf_wb_add_parm` accelerator routine is invoked.

For more information, see Chapter 10, “Modifying the work block parameter buffer when using multi-use work blocks,” on page 39.

Work block input data buffer

The work block input data buffer contains the input data for each work block (or each iteration of a multi-use work block) for the compute kernel. For each iteration of the ALF computational kernel, there is a single contiguous input data buffer. However, the data for the input buffer can come from distinct sections of a large data set in host memory. These separate data segments are gathered into the input data buffer on the accelerators. The ALF framework minimizes performance overhead by not duplicating input data unnecessarily. When the content of the work block is constructed by `alf_wb_dtl_entry_add`, only the pointers to the input data are saved to the internal data structure of the work block. This data is

transferred to accelerator memory when the work block is processed. A pointer to the contiguous input buffer in accelerator memory is passed to the computational kernel.

For more information about data scattering and gathering, see "Data transfer list" on page 18.

Work block output data buffer

This buffer is used to save the output of the compute kernel. It is a single contiguous buffer in accelerator memory. Output data can be transferred to distinct memory segments within a large output buffer in host memory. After the compute kernel returns from processing one work block, the data in this buffer is copied to the host memory locations specified by the `alf_wb_dtl_entry_add` routine when the work block is constructed.

Work block overlapped input and output data buffer

The overlapped input and output buffer (overlapped I/O buffer) contains both input and output data. The input and output sections are dynamically designated for each work block.

This buffer is especially useful when you want to maximize the use of accelerator memory and the input buffer can be overwritten by the output data.

For more information about when to use this buffer, refer to Chapter 8, "When to use the overlapped I/O buffer," on page 35.

For an example of how to use the buffer, see "Overlapped I/O buffer example" on page 177.

Chapter 8. When to use the overlapped I/O buffer

An overlapped I/O buffer is designed to maximize the memory usage on accelerators.

This is particularly useful when there is limited accelerator memory for input and output data. For each task instance, the ALF runtime provides an optional overlapped I/O buffer. This buffer is accessible from the user-defined computational kernel as well as the `input_dtl_prepare` and `output_dtl_prepare` functions. For each overlapped I/O buffer, you can dynamically define three types of buffer areas for each work block:

- `ALF_BUF_OVL_IN`: Data in the host memory is copied to this section of the overlapped I/O buffer before the computational kernel is called
- `ALF_BUF_OVL_OUT`: Data in this buffer area of the overlapped I/O buffer is written back to the host memory after the computational kernel is called
- `ALF_BUF_OVL_INOUT`: Data in the host memory is copied to this buffer area before the computational kernel is called and is written back to the same host memory location after the computational kernel is called

For examples of how to use the overlapped I/O buffer, see "Overlapped I/O buffer example" on page 177.

Points to consider when using the overlapped I/O buffer

When you use overlapped I/O buffer, you need to make sure that the input data area defined by `ALF_BUF_OVL_IN` and `ALF_BUF_OVL_INOUT` do not overlap each other. The ALF runtime does not guarantee the order in which the input data is fetched into accelerator memory, so the input data can become corrupted if these two areas overlap. Figure 9 shows a corrupted overlapped I/O buffer. In a double buffering scenario the first overlapped in-out buffer extends into the second overlapped input buffer.

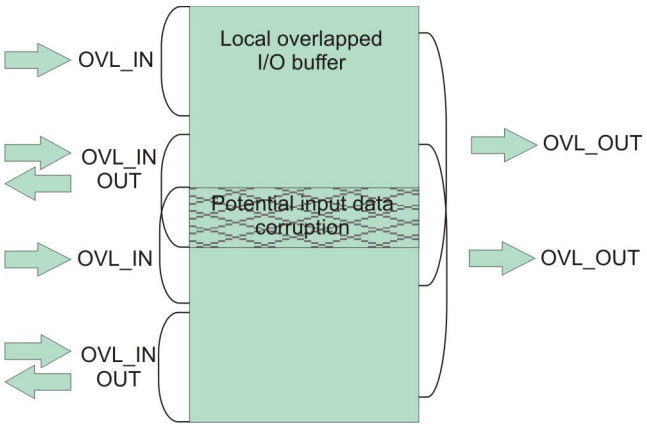


Figure 9. Corrupted overlapped I/O buffer

If you choose to partition data on the accelerator, you need to generate the data transfer lists for the input buffer, the overlapped input buffer, and the overlapped I/O buffer in the user-provided `alf_accel_input_dtl_prepare` function and generate the data transfer lists for both the output buffer and the overlapped

output buffer in the user-provided `alf_accel_output_dtl_prepare` function.

Chapter 9. Using work blocks and order of function calls per task instance on the accelerator

Based on the characteristics of an application, you can use single-use work blocks or multi-use work blocks to efficiently implement data partitioning on the accelerators.

For a given task that can be partitioned into *N* work blocks, the following describes how the different types of work blocks can be used, and also the order of function calls per task instance based on a single instance of a the task on a single accelerator:

1. Task instance initialization (this is done by the ALF runtime).
2. Conditionally execute: `alf_accel_task_context_setup`. It is only called if the task has a context. The runtime calls it when the initial task context data has been loaded to the accelerator and before any work blocks are processed.
3. For each work block WB(k) or pending context merge:
   a. Conditionally execute: `alf_accel_task_context_merge` and go to Step 3. This API is only called when the task context of another unloaded task instance is to be merged to current task instance.
   b. For a single-use work block WB(k) with i=0 and N=1, or for each iteration of a multi-use work block `i < N` (total number of iteration).
      1) `alf_accel_input_list_prepare(WB(k), i, N)`: It is only called when the task requires accelerator data partition.
      2) `alf_accel_comp_kernel(WB(k), i, N)`: The computational kernel is always called.
      3) `alf_accel_output_list_prepare(WB(k), i, N)`: It is only called when the task requires accelerator data partition.
4. Write out the task context.
5. Unload image awaiting scheduling of a new task instance.
   a. If a new task instance is created, go to Step 2.

For step 3, the calling order of the three function calls is defined by the following rules:

- For a specific single-use work block `WB(k)`, the following calling order is guaranteed:
  1. `alf_accel_input_list_prepare(WB(k))`
  2. `alf_accel_comp_kernel(WB(k))`
  3. `alf_accel_output_list_prepare(WB(k))`
- For two single-use work blocks that are assigned to the same task instance in the order of `WB(k)` and `WB(k+1)`, ALF only guarantees the following calling orders:
  - `alf_accel_input_list_prepare(WB(k))` is called before `alf_accel_input_list_prepare(WB(k+1))`
  - `alf_accel_comp_kernel(WB(k))` is called before `alf_accel_comp_kernel(WB(k+1))`
  - `alf_accel_output_list_prepare(WB(k))` is called before `alf_accel_output_list_prepare(WB(k+1))`

- For a multi-use work block `WB(k,N)`, it is considered as *N* single use work blocks assigned to the same task instance in the order of incremental iteration index `WB(k,0)`, `WB(k, 1)`, ..., `WB(k, N-1)`. The only difference is that all of these work blocks share the same work block parameter buffer. Other than that, the API calling order is still decided by the previous two rules. See Chapter 10, "Modifying the work block parameter buffer when using multi-use work blocks," on page 39.

Chapter 10. Modifying the work block parameter buffer when using multi-use work blocks

The work block parameter buffer of a multi-use work block is shared by multiple invocations of the `alf_accel_input_dtl_prepare` accelerator function and the `alf_accel_output_dtl_prepare` accelerator function. When you change the contents of this buffer, you must be careful because the ALF runtime does double buffering transparently. It is possible that the `current_count` arguments for succeeding calls to the `alf_accel_input_dtl_prepare` function, the `alf_accel_comp_kernel` function, and the `alf_accel_output_dtl_prepare` function are not strictly incremented when a multi-use work block is processed. Modifying the work block parameter buffer according to the `current_count` in one of the subroutines can cause unexpected effects to other subroutines when they are called with different `current_count` values at a later time.

Chapter 11. Double buffering on ALF

When transferring data in parallel with the computation, double buffering can reduce the time lost to data transfer by overlapping it with the computation time.

The ALF runtime implementation on Cell/B.E. architecture supports three different kinds of double buffering schemes; four-buffer scheme, three-buffer scheme, and overlapped I/O buffer scheme.

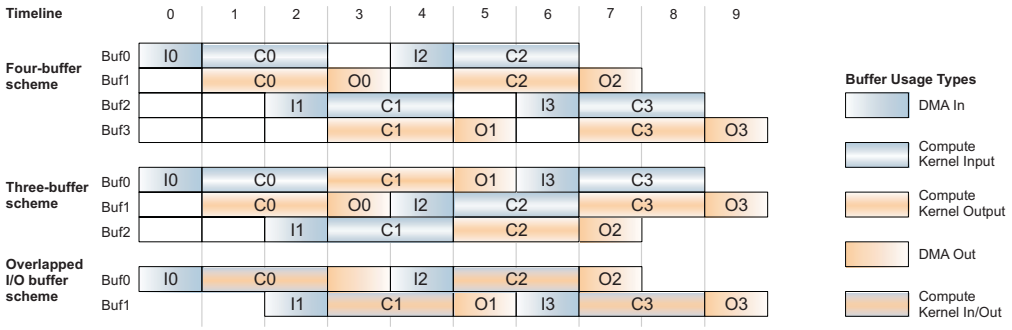


Figure 10. ALF double buffering

See Figure 10 for an illustration of how double buffering works inside ALF. The ALF runtime evaluates each work block and decides which buffering scheme is most efficient. At each decision point, if the conditions are met, that buffering scheme is used. The ALF runtime first checks if the work block uses the overlapped I/O buffer. If the overlapped I/O buffer is not used, the ALF runtime next checks the conditions for the four-buffer scheme. If the four-buffer scheme is not used, then it checks the conditions for the three-buffer scheme. If the conditions for none of these schemes are met, the ALF runtime uses single buffering, that is, it does not use double buffering.

The examples that follow use these assumptions:

1. All SPUs have 256 KB of local memory.
2. 16 KB of memory is used for code and runtime data including stack, the task context buffer, and the data transfer list. This leaves 240 KB of local storage for the work block buffers.
3. Transferring data in or out of accelerator memory takes one unit of time and each computation takes two units of time.
4. The input buffer size of the work block is represented as `in_size`, the output buffer size as `out_size`, and the overlapped I/O buffer size as `overlap_size`.
5. There are three computations to be done on three inputs, which produces three outputs.

Buffer schemes

The conditions and decision tree are further explained in the following examples:

- **Four-buffer scheme**: In the four-buffer scheme, two buffers are dedicated for input data and two buffers are dedicated for output data. The buffering used is shown in the Four-buffer scheme section of Figure 10.

  - **Conditions satisfied**: The ALF runtime chooses the four-buffer scheme if the work block does not use the overlapped I/O buffer and the buffer sizes satisfy the following condition: 2*(in_size + out_size) <= 240 KB.
  - **Conditions not satisfied**: If the buffer sizes do not satisfy the four-buffer scheme condition, the ALF runtime checks if the buffer sizes satisfy the conditions of the three-buffer scheme.
- **Three-buffer scheme**: In the three-buffer scheme, the buffer is divided into three equally sized buffers of the size max(in_size, out_size). The buffers in this scheme are used for both input and output as shown in the Three-buffer scheme section of Figure 10 on page 41. This scheme requires the output data movement of the previous result to be finished before the input data movement of the next work block starts, that is the DMA operations must be done in order. The advantage of this approach is that for a specific work block, if the input and output buffer are almost the same size, the total effective buffer size can be 2*240/3 = 160 KB.
  - **Conditions satisfied**: The ALF runtime chooses the three-buffer scheme if the work block does not use the overlapped I/O buffer and the buffer sizes satisfy the following condition: 3*max(in_size, out_size) <= 240 KB.
  - **Conditions not satisfied**: If the conditions are not satisfied, the single-buffer scheme is used.
- **Overlapped I/O buffer scheme**: In the overlapped I/O buffer scheme, two contiguous buffers are allocated as shown in the Overlapped I/O buffer scheme section of Figure 10 on page 41. The overlapped I/O buffer scheme requires the output data movement of the previous result to be finished before the input data movement of the next work block starts.
  - **Conditions satisfied**: The ALF runtime chooses the overlapped I/O buffer scheme if the work block uses the overlapped I/O buffer and the buffer sizes satisfy the following condition: 2*(in_size + overlap_size + out_size) <= 240 KB.
  - **Conditions not satisfied**: If the conditions are not satisfied, the single-buffer scheme is used.
- **Single-buffer scheme**: If none of the cases outlined above can be satisfied, double buffering is not used, and performance is degraded.

When creating buffers and data partitions, remember the conditions of these buffering schemes. If your buffer sizes can meet the conditions required for double buffering, it can result in a performance gain, but double buffering does not double the performances in all cases. When the unit of time required by data movements and computation are significantly different, the problem becomes either I/O-bound or computing-bound. In this case, enlarging the buffers to allow more data for a single computation might improve the performance even with a single buffer.

Chapter 12. Performance and debug trace

The Performance Debugging Tool (PDT) provides trace data necessary to debug functional and performance problems for applications using the ALF library.

Versions of the ALF libraries built with PDT trace hooks are delivered with SDK.

Installing the PDT

The libraries with the trace hooks, which are enabled, are packaged in separate "`-trace`" named packages. The trace enabled libraries are installed into a subdirectory named `alf/trace` in the library install directories. These packages and the PDT are included in the SDK package but may not be installed by default. For more information, refer to the *SDK Installation Guide*.

Refer to the *SDK Performance Guide* for instructions about how to install PDT, and how to set the correct environment variables to cause trace events to be generated.

Trace control

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

Environment variable

PDT supports an environment variable (`PDT_CONFIG_FILE`) that allows you to specify the relative or full path to a configuration file.

ALF ships an example configuration file that lists all of the ALF groups and events, and allows the user to turn selected ones off as desired. This is shipped as:

For ALF Cell/B.E.:

```
/usr/share/pdt/config/pdt_alf_config_cell.xml
```

For ALF for Hybrid x-86:

```
/usr/share/pdt/config/pdt_alf_config_hybrid.xml
```

An example of ALF with trace enabled is provided with the SDK 3.1 PDT package.

Part 4. Programming ALF on different platforms

This section describes information about installing and programming ALF for Cell/B.E. and Hybrid.

It describes the following:

For installation information, refer to the *SDK for Multicore Acceleration Version Installation Guide*.

Chapter 13. Implementation overview

This topic briefly describes the ALF for Cell/B.E. and the ALF for Hybrid implementations

ALF for Cell/B.E.

ALF for Cell/B.E. is an implementation of the ALF API specification for the Cell/B.E. processor. In this implementation, the PPEs serve as the hosts, the SPEs act as accelerators. For SDK 3.1, both 32-bit and 64-bit implementations are provided for this platform.

ALF for Hybrid

ALF for Hybrid is an implementation of the ALF API specification in a system configuration with an Opteron x86_64 system connected to one or more Cell/B.E. processors. In this implementation, the Opteron system serves as the host, the SPEs on the Cell/B.E. BladeCenters act as accelerators, and the PPEs on the Cell/B.E. processors act as facilitators only. From the ALF application programmer's perspective, the application interaction, as defined by the ALF API, is between the Hybrid host and the SPE accelerators.

This implementation of the ALF API uses the Data Communication and Synchronization (DaCS) library as the process management and data transport layer. Refer to the *DaCS for Hybrid Programmer's Guide and API Reference* for more information about how to set up DaCS in this environment.

To manage the interaction between the ALF host runtime on the Opteron system and the ALF accelerator runtime on the SPE, this implementation starts a PPE process (ALF PPE daemon) for each ALF runtime. The PPE program is provided as part of the standard ALF runtime library.

Chapter 14. Building and linking an application or an accelerated library

This topic describes how to create and link an application.

Three versions of the ALF libraries are provided with the SDK:

- Optimized: These libraries have minimal error checking on the host and accelerator and are intended for production use. For ALF for Cell/B.E. they are located in `/usr/lib` and `/usr/spu/lib`. For ALF for Hybrid they are located in `/opt/cell/sdk/prototype/usr/lib` and `/opt/cell/sdk/prototype/usr/spu/lib`.
- Error-check enabled: These libraries outputs debugging messages on the host, have error checking on the accelerators, and are intended to be used during application development. For ALF for Cell/B.E. they are located in `/usr/lib/alf/debug` and `/usr/spu/lib/alf/debug`. For ALF for Hybrid they are located in `/opt/cell/sdk/prototype/usr/lib/alf/debug` and `/opt/cell/sdk/prototype/usr/spu/lib/alf/debug`.
- Traced: These libraries are the optimized with performance and debug trace hooks in them. They are intended for debugging functional and performance problems associated with ALF. Refer to Chapter 12, "Performance and debug trace," on page 43 for more information about performance and debug options for ALF. For ALF for Cell/B.E. they are located in `/usr/lib/alf/trace` and `/usr/spu/lib/alf/trace`. For ALF for Hybrid they are located in `/opt/cell/sdk/prototype/usr/lib/alf/trace` and `/opt/cell/sdk/prototype/usr/spu/lib/alf/trace`.

Additionally, both shared object libraries and static libraries are provided for the ALF host libraries. The ALF SPE runtime library is only provided as static libraries.

An ALF application can be built as two separate files as follows:

- The first file is for the ALF host application, and you need to do the following:

  For Cell/B.E. applications:

  1. Compile the PPE host application. ALF host include file, `alf.h`, is located in the `/usr/include` directory.
  2. Link the PPE host application with the ALF PPE host shared object library, `libalf.so`, found in `/usr/lib` directory for 32–bit and `/usr/lib64` for 64–bit.

  For Hybrid applications:

  1. Compile the x86_64 host application with the `-D_ALF_PLATFORM_HYBRID_` define variable. The ALF host include file, `alf.h`, is located in the `/opt/cell/sdk/prototype/usr/include` directory.
  2. Link the x86_64 host application with the ALF x86_64 host debug shared object library, `libalf_hybrid.so`, found in the `/opt/cell/sdk/prototype/usr/lib64/alf/debug` directory and the DaCS x86_64 host runtime library, `libdacs_hybrid.so`. After it has been debugged use the optimized version in `/opt/cell/sdk/prototype/usr/lib64`.
- The second file is for the ALF SPE accelerator computational kernel, and you need to do the following:

  For Cell/B.E. applications:

  1. Compile the application's SPE code. The ALF accelerator header file, `alf_accel.h`, is located in `/usr/spu/include`.

2. For a work block task, link the application's SPE code with the ALF SPE accelerator debug static runtime library, `libalf.a`, found in `/usr/spu/lib/alf/debug`.For a lightweight task, link the application's SPE code with the ALF SPE accelerator static debug library, `libalflts.a`, found in `/usr/spu/lib/alf/debug`. After it has been debugged use the optimized library in `/usr/spu/lib`.
3. Use the `ppu-embedspu` utility to embed the SPU binary into a PPE ELF object file.
4. Link the resulting PPE ELF object into a PPE shared object library.

For Hybrid applications:

1. Compile the application's SPE code with the `-D_ALF_PLATFORM_HYBRID_` define variable. The ALF accelerator header file, `alf_accel.h` is located in the `/opt/cell/sysroot/usr/spu/include` and the `/opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/include` directories.

   Note: This assumes a cross-build environment where the host is the ALF for Hybrid host machine and the target is the Cell/B.E..
2. For a work block task, link the application's SPE code with the ALF SPE accelerator debug static library, `libalf_hybrid.a`, found in the `/opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/lib/alf/debug` directory. For a lightweight task, link the application's SPE code with ALF SPE accelerator debug static runtime library, `libalflts_hybrid.a`, which is located in `/opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/lib/alf/debug`. After the application has been debugged, use the optimized library in `/opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/lib`.
3. Use the `ppu-embedspu` utility to embed the SPU binary into a PPE ELF object file.
4. Link the resulting PPE ELF object needs to be linked as a PPE shared object library.

An ALF accelerated library for Cell/B.E. can be built as a single shared object library as follows:

The first step is to create an ALF SPE accelerator computational kernel as a PPE ELF object as follows:

1. Compile the application's SPE code. The ALF accelerator header file, `alf_accel.h`, is located in `/usr/spu/include`.
2. For a work block task, link the application's SPE code with the ALF SPE accelerator static runtime library, `libalf.a`, found in `/usr/spu/lib`. For a lightweight task, link the application's SPE code with ALF SPE accelerator static runtime library, `libalflts.a`, which is located in `/usr/spu/lib`.
3. Use the `ppu-embedspu` utility to embed the SPU binary into a PPE ELF object files.
4. Link the resulting PPE ELF object into the host accelerated shared object library.

The second step is to create an ALF host shared object library containing the ALF accelerator executable image as follows:

1. Compile the PPE host library. The ALF host include file, `alf.h`, is located in `/usr/include` directory.
2. Link the PPE host library with the ALF PPE host shared object library, `libalf.so`, found in `/usr/lib` directory for 32-bit and `/usr/lib64` for 64-bit, and the ALF accelerator PPE ELF object containing the embedded ALF SPE accelerator computational kernel which was created in the last step.

The third step is to link your application with this shared object library as follows:

1. Edit your PPE host application containing a NULL library path for `alf_init` and a NULL library name for `alf_task_desc_set_int64` and compile it.
2. Link your PPE host application with your accelerated library.

By setting both to NULL then when you run your program and it calls your PPE functions your accelerated library is loaded, and when your PPE functions call ALF it searches the currently loaded shared object libraries which now contains your accelerated library for your SPE image and functions.

For references, Makefiles are provided for all of the samples in the package:

For Cell/B.E.:

`alf-examples-source-*.noarch.rpm`

For Hybrid:

`alf-hybrid-examples-source-*.noarch.rpm`

Linking to the correct library

This topic describes to which library you should link for your ALF implementation.

- Make sure that ALF applications are linked with the correct library for the intended ALF implementation (either Cell/B.E. or Hybrid). That is, link the ALF for Cell/B.E. accelerator computational kernel with either the `libalf.a`, for work block support or `libalflts.a` for lightweight task support. Link the ALF for Hybrid accelerator computation kernel with either `libalf_hybrid.a` for work block support or `libalflts_hybrid.a` for lightweight task support.
- Linking against the wrong library produces a link error.
- An application's ALF SPE accelerator computational kernel must be relinked to be compatible with ALF for SDK 3.1.

Chapter 15. Embedding the SPU binaries into the PPU binary on Cell/B.E. systems

This topic describes what you need to consider when you embed SPU binaries.

Because the accelerator can run on a different system than that of the host, ALF usually assumes the accelerator kernel binary is standalone and is loaded at runtime. However, the Cell/B.E. Linux® ABI supports a special feature called CESOF, which defines how you can wrap an SPE executable binary into a PPE binary. The ALF implementation for Cell/B.E. also supports this feature. When the SPE binary is embedded directly into the PPE application itself or is within a pre-linked PPE shared object library, the task descriptor attribute `ALF_TASK_DESC_ACCEL_LIBRARY_REF_L` can be set to NULL. Depending on the build process, there are two different scenarios to consider:

- **The SPE binary is embedded within another PPE shared object library**

  The typical scenario is a PPE functional library that uses ALF. The library is shipped in shared object format and with the SPU binaries embedded. The build process is the same as other PPE libraries with embedded SPU binaries. It is not necessary to build a separate shared object binary for the SPE binary, as the required symbols are exported together with the PPE library.
- **The SPE binary is embedded within the application binary statically**

  In this case, the SPE binary is embedded statically into the application binary. Because only referenced external symbols are exported to the dynamic symbol table, the following build options are necessary for the ALF runtime to find the SPE kernel symbols:
  - If you use gcc to generate the final binary, you must specify the `-rdynamic` command line option
  - If you use GNU ld to generate the final binary, you must specify `-export-dynamic` command line option

Chapter 16. Running an application

ALF uses both a library path and a library name to create a fully-qualified accelerator library file name to be loaded.

The table below shows the resulting file name based upon various ways to specify the library path and library name. Use the ALF host debug library located in `/usr/lib[64]/alf/debug` for ALF for Cell/B.E. and in `/opt/cell/sdk/prototype/usr/lib[64]/alf/debug` for ALF for Hybrid to aid in debugging problems loading this library.

Table 3. Library paths and library names

Accelerator library path and library name					Accelerator library name	
					Library name NULL	Library name NOT NULL
Accelerator library path	alf_init config parm NOT NULL	cfg.library_path NOT NULL			NULL	`library_path/library_name`
		cfg.library_path NULL	ALF_LIBRARY_PATH Defined	ALF_LIBRARY_PATH NOT empty	NULL	`ALF_LIBRARY_PATH/library_name`
				ALF_LIBRARY_PATH empty	NULL	`(sys_libpath)/library_name`
			ALF_LIBRARY_PATH NOT defined		NULL	`(ALF_default)/library_name`
	alf_init config parm NULL	ALF_LIBRARY_PATH defined	ALF_LIBRARY_PATH NOT empty		NULL	`ALF_LIBRARY_PATH/library_name`
			ALF_LIBRARY_PATH empty		NULL	`(sys_libpath)/library_name`
		ALF_LIBRARY_PATH NOT defined			NULL	`(ALF_default)/library_name`

- **Library Name NULL** means `alf_task_desc_set_int64 (task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REL_L, NULL);` In this case, the accelerator execution image must be embedded in the binary of the application or the linked host library. For Cell/B.E., refer to description in Chapter 15, "Embedding the SPU binaries into the PPU binary on Cell/B.E. systems," on page 53.
- **Library Name NOT NULL** means `alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REL_L,"string");`
- **ALF_LIBRARY_PATH NOT Defined** means `unset ALF_LIBRARY_PATH`
- **ALF_LIBRARY_PATH Defined** and **ALF_LIBRARY_PATH NOT Empty** means `export ALF_LIBRARY_PATH=string`
- **ALF_LIBRARY_PATH Defined** and **ALF_LIBRARY_PATH Empty** means `export ALF_LIBRARY_PATH=`
- `(ALF_default)` is defined as the current shell path '.' currently.
- `(sys_libpath)` is defined as the system level dynamic library / shared object file search path.

Running an application on Cell/B.E. systems

The following steps describe how to run an ALF for Cell/B.E. application.

To run an application, do the following:

1. Build the ALF application as an executable, `my_appl`.
2. Build the ALF accelerator computational kernel as an SPE binary executable and embed it into a PPE shared object library, `my_appl.so`.

3. Copy the PPE shared object library `my_appl.so` containing the embedded SPE binaries to a selected directory, `/tmp/my_directory`.

   Note: If the PPE host application and the PPE shared object library are copied to the current working directory, then the ALF_LIBRARY_PATH does not need to be set. In other words, the default ALF_LIBRARY_PATH is the current working directory.
4. Set the environment variable ALF_LIBRARY_PATH to the selected directory on the Cell/B.E. For example:

   ```
   export ALF_LIBRARY_PATH=/tmp/my_directory
   ```
5. Invoke the PPE host application from the current working directory. For example:

   ```
   ./my_appl
   ```

Running an application on Hybrid systems

The following steps describe how to run an ALF for Hybrid application.

Note: You need to ensure that the shared object libraries `libalf_hybrid.so` and `libdacs_hybrid.so` are accessible. You can set this through `LD_LIBRARY_PATH`. For example:

```
export LD_LIBRARY_PATH=/opt/cell/sdk/prototype/usr/lib64
```

To run an application, do the following:

1. Build the ALF for Hybrid application,`my_appl`. Build the ALF for Hybrid accelerator computational kernel as an SPE binary executable and embed it into a PPE shared object library, `my_appl.so`.
2. Copy the PPE shared object library with the embedded SPE binaries from the host where it was built to a selected directory on the Cell/B.E. where it is to be executed. For example:

   ```
   scp my_appl.so <CBE>:/tmp/my_directory
   ```
3. On the ALF for Hybrid host machine, set the environment variable `ALF_LIBRARY_PATH` to the above selected directory on the Cell/B.E.. For example:

   ```
   export ALF_LIBRARY_PATH=/tmp/my_directory
   ```
4. Set the processor affinity on the Hybrid host. Refer to the *DaCS Programmer's Guide and API Reference*, section "Affinity requirements for host applications" for more information about how to do this. For example:

   ```
   taskset –p 0x00000001 $$
   ```
5. Run the x86_64 host application in the host environment. For example:

   ```
   ./my_appl
   ```

Chapter 17. Optimizing ALF applications

This section describes how to optimize your ALF applications.

It covers the following topics:

- "Using accelerator data partitioning"
- "Using multi-use work blocks"
- "What to consider for data layout design"
- "Using data sets" on page 58

Using accelerator data partitioning

If the application operates in an environment where the host has many accelerators to manage and the data partition schemes are particularly complex, it is generally more efficient for the application to specify accelerator partitioning in the task descriptor and generate the data transfer lists on the accelerators instead on the host.

For more information about how to use this feature, refer to "Accelerator data partitioning" on page 29.

Using multi-use work blocks

If there are many instances of the task running on the accelerators and the amount of computation per work block is small, the ALF runtime can become overwhelmed with transferring work blocks and associated data in and out of accelerator memory. In this case, multi-use work blocks can be used in conjunction with accelerator data partitioning to further improve performance for an ALF application.

For an example of how to use multi-use work blocks, refer to "Implementation 2: Making use of multi-use work blocks together with task context or work block parameter/context buffers" on page 176.

What to consider for data layout design

Efficient data partitioning and data layout design is the key to a well-performing ALF application. Improper data partitioning and data layout design either prevents ALF from being applicable or results in degraded performance. Data partition and layout is closely coupled with compute kernel design and implementation, and they should be considered simultaneously. You should consider the following for your data layout and partition design:

- Use the correct size for the data partitioned for each work block. Often the local accelerator memory is limited. Performance can degrade if the partitioned data cannot fit into the available memory. For example, on Cell/B.E. architecture, if the input buffer of a work block is larger than 128 KB, it might not be possible to support double buffering on the SPE. This can result in up to 50% performance loss.

- Minimize the amount of data movement. A large amount of data movement can cause performance loss in applications. Improve performance by avoiding unnecessary data movements.
- Simplify data movement patterns. Although the data transfer list feature of ALF enables flexible data gathering and scattering patterns, it is better to keep the data movement patterns as simple as possible. Some good examples are sequential access and using single contiguous movements instead of many small discrete movements.
- Avoid data reorganization. Data reorganization requires extra work. It is better to organize data in a way that suits the usage pattern of the algorithm than to write extra code to reorganize the data when it is used.
- Be aware of the address alignment limitations on Cell/B.E..

Using data sets

The data set is the primary mechanism for optimizing an ALF application on a hybrid system. Using the data set improves ALF application performance on the hybrid environment significantly. For an overview about data sets, see "Data set" on page 22.

The ALF for Hybrid implementation uses the data set to speed up data read/write access time by migrating the data set closer to the accelerators. That is, when the task is ready and is scheduled for execution on a specific PowerPC Processing Element (PPE), the associated data set is transferred from the Hybrid host to the PPE. All read-only and read-write buffers are transferred at this time. This makes the data set available for the accelerator's data access. Similarly, when you issue the `alf_task_wait` function, and the task completes, the data set is transferred from the PPE to the Hybrid host. All read-only and read-write buffers are transferred at this time. This makes the data set available for the host's data access.

Refer to "Data set example" on page 181 for an example of how to use data sets.

Chapter 18. Platform-specific constraints for the ALF implementation

This section describes constraints that apply when you program ALF.

SPE accelerator memory constraints

This topic describes the memory limitations for both the ALF for Hybrid and ALF for Cell/B.E. implementations.

Note: Because the ALF for Hybrid implementation is based upon the ALF for Cell/B.E.. implementation, the following constraints are applicable to both the Cell/B.E. and Hybrid implementations. Although the explanations and examples used sometimes only refer to the Cell/B.E. implementation, they are applicable to both implementations.

The size of local memory on the SPE accelerator is 256 KB and is shared by code and data. Memory is not virtualized and is not protected. See Figure 11 on page 60 for a typical memory map of an SPU program. There is a runtime stack above the global data memory section. The stack grows from the higher address to the lower address until it reaches the global data section. Due to the limitation of programming languages and compiler and linker tools, you cannot predict the maximum stack usage when you develop the application and when the application is loaded. If the stack requires more memory than what was allocated you do not get a stack overflow exception (unless this was enabled by the compiler at build time) you get undefined results such as bus error or illegal instruction. When there is a stack overflow, the SPU application is stopped or fails and a message is sent to the PPE.

ALF allocates the work block buffers directly from the runtime stack, as shown in Figure 12 on page 60. This is implemented by adjusting the stack pointer (or equivalently by pushing a large amount of data into the stack). The larger the buffer is, the better the ALF runtime can optimize the performance of a task by using techniques like double buffering. It is better to let ALF allocate as much memory as possible from the runtime stack. If the stack size is too small at runtime, a stack overflow occurs and it causes unexpected exceptions such as incorrect results or a bus error.

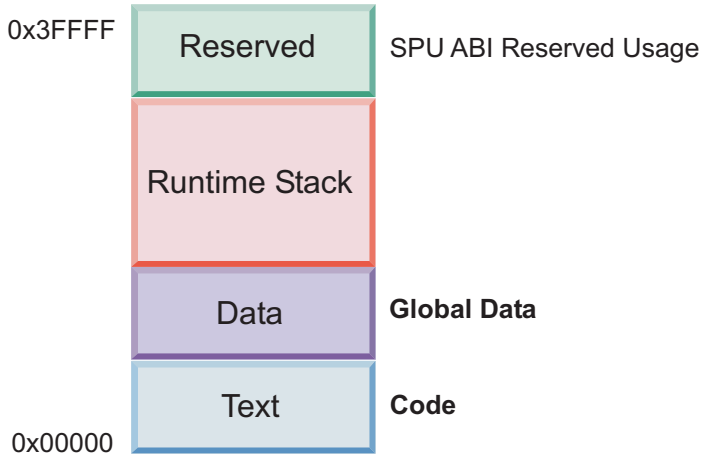


Figure 11. SPU local memory map of a common Cell/B.E. application

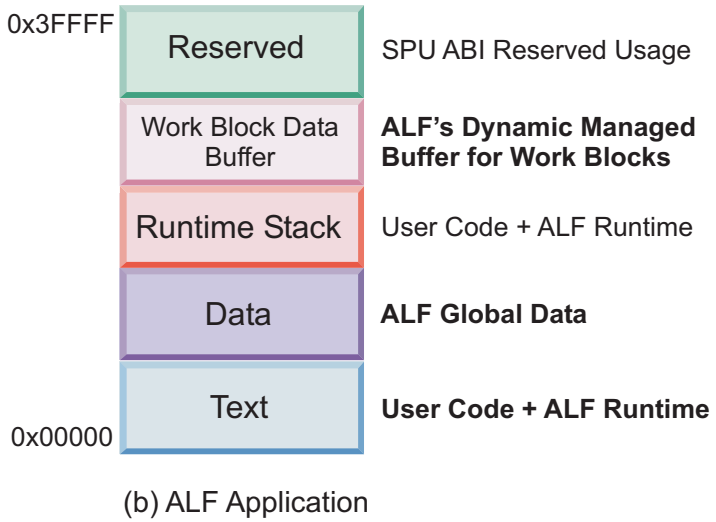


Figure 12. SPU local memory map of an ALF application

Data transfer list limitations

Data transfer information is used to describe the five types data movement operations for one work block as defined by `ALF_BUF_TYPE_T`.

Note: Because the ALF for Hybrid implementation is based upon the ALF for Cell/B.E. implementation, the following constraints are applicable to both the Cell/B.E. and Hybrid implementations. Although the explanations and examples used sometimes only refer to the Cell/B.E. implementation, they are applicable to both implementations.

The ALF implementation on Cell/B.E. has the following internal constraints:

1. Data transfer information for a single work block can consist of up to a specific number of data transfer lists as defined by the task descriptor attribute `ALF_TASK_DESC_NUM_DTL`. For programmers the limitation is that `alf_wb_dtl_begin` should not exceed this number for each work block. An

ALF_ERR_NOBUFS is returned if this limit is exceeded. Due to the limitation of items 2, 3 and 4 in this list, it is possible that the limitation can be reached without explicitly calling alf_wb_dtl_begin by so many times.

2. Each data transfer list may consist of up to 2048 data transfer entries. The alf_wb_dtl_entry_add call automatically creates a new data transfer list when this limitation is reached. The limitation of item 1 in this list still applies in this case.
3. Each entry can describe up to 16 KB of data transfer between a contiguous area in host memory and an equivalent area in accelerator memory. The alf_wb_dtl_entry_add call automatically breaks an entry larger than 16 KB to multiple entries. The limitation of items 1 and 2 in this list still apply in this case.
4. All of the entries within the same data transfer list share the same high 32 bits of the effective address. This means that when a data transfer entry goes across 4 GB address boundary, it must be broken up and put into two different data transfer lists. In addition, two succeeding entries use different high 32 bits of the effective address, they need to be put into two lists. The alf_wb_dtl_entry_add call automatically creates a new data transfer list in the above two situations. The limitation of items 1, 2 and 3 in this list still apply in this case.
5. The local store area described by each entry within the same data transfer list must be contiguous. You can use the local buffer offset parameter "offset_to_accel_buf" to address within the local buffer when alf_wb_dtl_begin is called to create a new list.
6. The transfer size and the low 32 bits of the effective address for each data transfer entry must be aligned on a 16–byte address boundary. The alf_wb_dtl_entry_add call does NOT automatically deal with alignment issues. An ALF_ERR_INVAL error is returned if there is an unaligned address. The same limitation also applies to the offset_to_accel_buf parameter of alf_wb_dtl_begin.

Transfer data with optimal alignment and granularity. The most important factor in maximizing the performance of the memory subsystem is to ensure that all transfers begin on a cache-line (128 byte) boundary and are sized as a multiple of 128 bytes. The C/C++ type attribute aligned can be used to ensure arrays, structures, and classes are properly aligned. If alignment cannot be guaranteed, then consider transferring extra data on the front and back to ensure cache-line alignment and size is achieved. This is especially important for writes to memory (DMA PUTs).

For the large memory configuration on the IBM BladeCenter QS22 (greater than 8 GB), partial cache-line writes can have significant application performance impact due to the resulting read-modify-write sequence that occurs when storing a partial cache-line. If the cache-line alignment can not be achieved, then the next best option is to ensure that the alignment of both the source and destination buffers for all transfers have the same alignment (that is, the same seven least significant address bits). This ensures that performance penalties associate with partial cache-line transfers only occur on the first and possibly the last transfer.

Data set constraints for ALF for Hybrid

The ALF for Hybrid data set implementation is limited to eight (8) buffers.

Providing a data set with no data buffers or buffers with a size of 0 (zero) returns an error. The API specification does not specify the behavior in this case.

Other known limitations for ALF for Hybrid

The following are the known limitations for the ALF for Hybrid implementation.

- Accelerator partitioning only works if there is a data set associated with the specified task.
- The ALF for Hybrid implementation currently does not support data coherency across multiple IBM BladeCenters QS21 and QS22. Using a data set with one Hybrid x-86 host and multiple IBM BladeCenters leads to data corruption. However, host data partitioning without data sets works across multiple IBM BladeCenters QS21 and QS22.
- The ALF API supports the concept of heterogeneous accelerators; however, in this hybrid implementation of ALF, only the SPE is supported as an accelerator. The `accel_type` parameter in the API function `alf_task_desc_create` is ignored.
- If the debug version of the ALF SPU runtime is used then all of the buffer sizes provided through `alf_task_desc_set_int32`need to be multiples of 16 bytes, otherwise the SPE returns a fatal error.

Part 5. API reference

This section covers the following topics:

Chapter 19. ALF API overview

This topic describes API conventions, data types, and data structures.

Conventions

ALF and *alf* are the prefixes for the namespace for ALF. For normal function prototypes and data structure declarations, use all lowercase characters with underscores (_) separating the words. For macro definitions, use all uppercase characters with underscores separating the words.

Data type assumptions

`int`	This data type is assumed to be signed by default on both the host and accelerator. The size of this data type is defined by the Application Binary Interface (ABI) of the architecture. However, the minimum size of this data type is 32 bits. The actual size of this data type might differ between the host and the accelerator architectures.
`unsigned int`	This data type is assumed to be the same size as that of int, but it is unsigned.
`char`	This data type is not assumed to be signed or unsigned. The size of this data structure, however, must be 8 bits.
`long`	This data type is not used in the API definitions because it might not be uniformly defined across platforms.
`void *`	The size of this data type is defined by the ABI of the corresponding architecture and compiler implementation. Note that the actual size of this data type might differ between the host and accelerator architectures.

Platform-dependent auxiliary APIs or data structures

The basic APIs and data structures of ALF are designed with cross-platform portability in mind. Platform-dependent implementation details are not exposed in the core APIs.

Common data structures

The enumeration type `ALF_DATA_TYPE_T` defines the data types for data movement operations between the hosts and the accelerators. The ALF runtime does byte swapping automatically if the endianness of the host and the accelerators are different. To disable endian swapping, you can use the data type `ALF_DATA_BYTE`.

`ALF_DATA_BYTE`	For data types that are independent of byte orders
`ALF_DATA_INT16`	For two bytes signed / unsigned integer types
`ALF_DATA_INT32`	For four bytes signed / unsigned integer types
`ALF_DATA_INT64`	For eight bytes signed / unsigned integer types
`ALF_DATA_FLOAT`	For four byte floating point types
`ALF_DATA_DOUBLE`	For eight byte floating point types
`ALF_DATA_ADDR32`	For 32-bit address types
`ALF_DATA_ADDR64`	For 64-bit address types

ALF also has its own definition of the following different data types for you to use.

`alf_data_byte_t`	For one byte data type
`alf_data_uint16_t`	For two bytes unsigned integer data type
`alf data_int16_t`	For two bytes signed integer data type
`alf_data_uint32_t`	For four bytes unsigned integer data type
`alf_data_int32_t`	For four bytes signed integer data type
`alf_data_uint64_t`	For eight bytes signed integer type
`alf_data_int64_t`	For eight bytes signed integer type
`alf_data_addr32_t`	For 32–bit address
`alf_data_addr64_t`	For 64–bit address

ALF_NULL_HANDLE

NAME

ALF_NULL_HANDLE - Used to indicate a non-initialized handle in the ALF runtime environment.

DESCRIPTION

The constant `ALF_NULL_HANDLE` is used to indicate a non-initialized handle in the ALF runtime environment. All handles should be initialized to this value to avoid ambiguity in the semantics of the code.

ALF_STRING_TOKEN_ MAX

NAME

ALF_STRING_TOKEN_ MAX - This constant defines the maximum allowed length of the string tokens in unit of bytes, excluding the trailing zero.

DESCRIPTION

These string tokens are used in ALF as identifiers of function names or other purposes. Currently, this value is defined to be 251 bytes.

ALF_DATASET_BUFFER_MAX_NUM

NAME

ALF_DATASET_BUFFER_MAX_NUM - This constant defines the maximum number of data buffers that are allowed to be added to a data set.

DESCRIPTION

The data buffers added to a data set should be no more than this number. Currently, the value is defined to be 8.

alf_strerror

NAME

alf_strerror - Returns a descriptive string for the specific error code value.

SYNOPSIS

const char* alf_strerror(int error_code)

Parameters
`error_code [IN]` Error code

DESCRIPTION

This function returns a descriptive string for the specific error code value.

RETURN VALUE

The following table lists the corresponding descriptive string for each error code. The returned string is not modifiable.

Error code	Descriptive string
0	OK
ALF_ERR_PERM	No permission
ALF_ERR_SRCH	No such task
ALF_ERR_2BIG	I/O buffer request exceeds limitations
ALF_ERR_NOEXEC	Cannot execute task
ALF_ERR_BADF	Bad handle
ALF_ERR_AGAIN	Try again
ALF_ERR_NOMEM	Out of memory
ALF_ERR_FAULT	Invalid address
ALF_ERR_BUSY	Resource busy
ALF_ERR_INVAL	Invalid argument
ALF_ERR_RANGE	Out of range
ALF_ERR_NOSYS	Function not implemented
ALF_ERR_BADR	Resource request cannot be fulfilled
ALF_ERR_NODATA	No more data available
ALF_ERR_TIME	Time out
ALF_ERR_COMM	Communications error
ALF_ERR_PROTO	Internal protocol error
ALF_ERR_BADMSG	Unrecognized message
ALF_ERR_OVERFLOW	Overflow
ALF_ERR_INCOMPAT	Accelerator incompatibility
ALF_ERR_NOBUFS	No buffer space available
ALF_ERR_ACCEL	Generic accelerator error
All other values	Unrecognized error code

Chapter 20. Host API

This topic describes the syntax for the host API.

- The host API includes the following:
- "Basic framework API" on page 72
- "Compute task API" on page 88
- "Work block API" on page 110
- "Data set API" on page 123

Basic framework API

This topic describes the definitions for the basic framework data types and functions.

The following APIs and data types are described:

ALF_LIBRARY_PATH environment variable

NAME

ALF_LIBRARY_PATH

DESCRIPTION

ALF_LIBRARY_PATH is an environment variable that defines a string which contains one or more library paths. Both ALF for Cell/B.E. and ALF for Hybrid support this environment variable. These library paths are used at runtime to find and load the application's accelerator shared object library. Each entry in the library path is prepended to the library name to form fully-qualified library file name to be loaded. The library name is defined in the ALF task descriptor using the alf_task_desc_set_int64 function with the ALF_TASK_DESC_ACCEL_LIBRARY_REF_L parameter. If the library path is an empty string then nothing is prepended to the library name, and the library is located and loaded using the default operating system shared object library search protocol. For example, for Linux the entries in the LD_LIBRARY_PATH environment variable are used. If the library name value is NULL then the library path is ignored, no library is loaded and the accelerator executable image is searched for in the currently loaded shared object libraries. The format of the path string is platform-dependent, including the format of each single path string and the separation token between multiple path strings. For the Linux environment, "/" separates file names in a path, and ":" separates unique paths. For example, /x/y/z:/a/b/c.

Overridden: If the application explicitly specifies a library_path string in the configuration data structure when it initializes the ALF runtime with alf_init or alf_init_shared, then this environmental variable is overridden. If the application specifies a NULL for the library path string, then this environment variable is used.

Default: If the ALF_LIBRARY_PATH environment variable is not set and the explicit configuration library_path string is not provided, the default search path of accelerator execution images is the current directory (./).

The following describes how the accelerator execution binary is located. For more information, refer to Chapter 16, "Running an application," on page 55.

Definitions:

- library_full_path: The full path to the library file in which the accelerator execution binary locates
- library_name: The argument of alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REL_L, library_name). The default is NULL if not set
- ALF_LIBRARY_PATH: The shell environmental variable ALF_LIBRARY_PATH
- alf_cfg: The alf configuration information for alf_init(p_sys_config_info, &alf_handle)
- alf_cfg.library_path: The library_path member of alf configuration information for alf_init
- ALF_DEFAULT_PATH: It is internally defined as the current shell path "."
- SYS_LIB_PATH: It is defined as the system default dynamic library or shared object file search path

Conditions:

```
if(library_name == NULL)
{
   library_full_path = NULL;
   /* In this case, the accelerator execution image must be already loaded into the current
   process's address space by either the system application/library loader or other library
   loading routines. For cell/B.E. system, refer to the descriptions in section
   "Embedding the SPU Images to PPU Binary on Cell/B.E. systems". */
}
else if(alf_cfg == NULL || alf_cfg.library_path == NULL)
/* the system configuration does not define the library path */
{
   alf_library_path_env = getenv(ALF_LIBRARY_PATH);
   if(alf_library_path_env == NULL) /* ALF_LIBRARY_PATH not defined */
   {
        alf_library_path_env = ALF_DEFAULT_PATH;
   }
   else if(strlen(alf_library_path_env) == 0) /* ALF_LIBRARY_PATH defined as empty string */
   {
        alf_library_path_env = SYS_LIB_PATH;
   }
   /* else ALF_LIBRARY_PATH defined with a path string */
   library_file_path = alf_library_path_env + "/" + library_name;
}
else
/* the system configuration defines the library path */
{
   library_file_path = alf_cfg.library_path + "/" + library_name;
}
```

SEE ALSO

alf_init(3); alf_init_shared(3)

alf_handle_t data type

NAME

alf_handle_t - This data type is a reference to an instance of the ALF runtime.

DESCRIPTION

This data type is a reference to an instance of the ALF runtime. The data type is initialized by the `alf_init` API and is released or destroyed by the `alf_exit` API.

alf_init function

NAME

alf_init - Initializes the isolated ALF instance.

SYNOPSIS

int alf_init(void* p_sys_config_info, alf_handle_t* p_alf_handle);

Parameters

`p_sys_config_info [IN]`

A platform-dependent configuration data structure that the ALF runtime uses for system initialization and configuration. If no configuration data structure is required then set this parameter to NULL.

ALF for Cell/B.E.: This parameter points to an `alf_sys_config_t_CBEA` data structure. This data structure is defined as follows:

```
typedef struct {
    char* library_path;
 } alf_sys_config_t_CBEA;
```

ALF for Hybrid: This parameter points to an `alf_sys_config_t_Hybrid` data structure. This data structure is defined as followed;

```
typedef struct {
  char* library_path;
  unsigned int num_of_ppes;
  char* ppe_image_path;
  char* ppe_image_name;
  char* ppe_image_mode;
 } alf_sys_config_t_Hybrid;
```

`p_alf_handle [OUT]`

A pointer to the handle of the ALF runtime. It is updated if the call is successful. Otherwise, it is not modified.

DESCRIPTION

This function initializes the ALF runtime and creates an isolated ALF instance. An isolated ALF instance is not shared with any other ALF instances. It allocates the necessary resources and global data as well as setting up any platform-specific configuration information.

RETURN VALUE

0

Successful

less than 0

Errors:

- `ALF_ERR_INVAL`: Invalid input parameter
- `ALF_ERR_NODATA`: Some system configuration data is not available
- `ALF_ERR_NOMEM`: Out of memory or some system resources have been used up

OPTIONS

Field values for ALF for Cell/B.E.

`library_path`	A pointer to a character string containing the path to all of the application's computational kernel shared object libraries. If the pointer is NULL, the `ALF_LIBRARY_PATH` environment variable is checked and if it is defined then it is used. If neither is set, the default "." (the current directory) is used.
Field values for ALF for Hybrid	
`library_path`	A pointer to a character string containing the path to all of the application's computational kernel shared object libraries. If the pointer is NULL, the `ALF_LIBRARY_PATH` environment variable is checked and if it is defined then it is used. If neither is set, the default "." is used.
`num_of_ppes`	The number of PPEs to use for application execution. If the value is 0, the `ALF_NUM_OF_PPES` environment variable is checked and if it is defined then it is used. If neither is set, the default 0 (use all available PPEs) is used.
`ppe_image_path`	Depending upon the `ppe_image_mode` (see below), it is the pointer to a character string containing either the path to all of the application's computational kernel shared object libraries the remote path to the application's PPE daemon or the local path to the application's file list which are to be transferred to the PPE and the first executed. If the pointer is NULL, the `ALF_PPE_IMAGE_PATH` environment variable is checked and if it is defined then it is used. If neither is set, the default `/opt/cell/sdk/prototype/usr/bin` is used.
`ppe_image_name`	Depending upon the `ppe_image_mode` (see below), it is the pointer to a character string containing either the name of the application's PPE daemon, or the name of the application's file list. If the pointer is NULL, the `ALF_PPE_IMAGE_NAME` environment variable is checked and if it is defined then it is used. If neither is set, the default `alf_hybrid_d_ppu64` is used.
`ppe_image_mode`	The pointer to the character string containing the mode to interpret the `ppe_image_path` and `ppe_image_name` (see above). It is either the case-insensitive string `"remote"` or `"filelist"`. If the pointer is NULL, the `ALF_PPE_IMAGE_MODE` environment variable is checked and if it is defined then it is used. If neither is set, the default `"remote"` is used.

alf_init_shared function

NAME

alf_init_shared - Initializes a shared ALF instance.

SYNOPSIS

int alf_init_shared(void* p_sys_config_info, alf_handle_t* p_alf_handle);

Parameters

`p_sys_config_info [IN]`	A platform-dependent configuration data structure that the ALF runtime uses for system initialization and configuration. If no configuration data structure is required then set this parameter to NULL. See “alf_init function” on page 76 for further information about this parameter.
`p_alf_handle [OUT]`	A pointer to the handle of the ALF runtime. It is updated if the call is successful. Otherwise, it is not modified.

DESCRIPTION

This function initializes the ALF runtime and creates a shared ALF instance. A shared ALF instance is shared by multiple ALF handles. If no previous shared ALF instance exists then it allocates the necessary resources and global data as well as setting up any platform specific configuration information. If a previous shared ALF instance exists then it returns a handle which references the existing shared ALF instance.

RETURN VALUE

0	Successful
less than 0	Errors: • `ALF_ERR_INVAL`: Invalid input parameter • `ALF_ERR_NODATA`: Some system configuration data is not available • `ALF_ERR_NOMEM`: Out of memory or some system resources have been used up

alf_query_system_info function

NAME

alf_query_system_info - Queries basic system information.

SYNOPSIS

int alf_query_system_info(alf_handle_t alf_handle, ALF_QUERY_SYS_INFO_T query_info, ALF_ACCEL_TYPE_T accel_type, unsigned int * p_query_result);

Parameters

`alf_handle [IN]` Handle to the ALF runtime.

`query_info [IN]` A query identification that indicates the item to be queried.

- `ALF_QUERY_NUM_ACCEL`: Returns the number of accelerators in the system. When the system does not have the specified accelerator type, the return value is zero and is a valid return value.
- `ALF_QUERY_HOST_MEM_SIZE`: Returns the memory size of host nodes up to 4T bytes, in units of kilobytes (2^10 bytes). When the size of memory is more than 4T bytes, the total reported memory size is (`ALF_QUERY_HOST_MEM_SIZE_EXT`*4T + `ALF_QUERY_HOST_MEM_SIZE`*1K) bytes. In case of systems where virtual memory is supported, this should be the maximum size of one contiguous memory block that a single user space application could allocate.
- `ALF_QUERY_HOST_MEM_SIZE_EXT`: Returns the memory size of host nodes, in units of 4T bytes (2^42 bytes).
- `ALF_QUERY_ACCEL_MEM_SIZE`: Returns the memory size of accelerator nodes up to 4T bytes, in units of kilo bytes (2^10 bytes) . When the size of memory is more than 4T bytes, the total reported memory size is (`ALF_QUERY_ACCEL_MEM_SIZE_EXT`*4T + `ALF_QUERY_ACCL_MEM_SIZE`*1K) bytes. For systems where virtual memory is supported, this should be the maximum size of one contiguous memory block that a single user space application could allocate.
- `ALF_QUERY_ACCEL_MEM_SIZE_EXT`: Returns the memory size of accelerator nodes, in units of 4T bytes (2^42 bytes).
- `ALF_QUERY_HOST_ADDR_ALIGN`: Returns the basic requirement of memory address alignment on host node side, in exponential of 2. A zero stands for byte aligned address. A 4 is to align by 16 byte boundaries.
- `ALF_QUERY_ACCEL_ADDR_ALIGN`: Returns the basic requirement of memory address alignment on accelerator node side, in exponential of 2. A zero stands for byte aligned address. An 8 is to align by 256 byte boundaries
- `ALF_QUERY_DTL_ADDR_ALIGN`: Returns the address alignment of data transfer list entries, in exponential of 2. A zero stands for byte aligned address. An 8 is to align by 256 byte boundaries.
- `ALF_QUERY_ACCEL_ENDIAN_ORDER`: The endianness or byte ordering of data within the accelerator node. The value is either:
  - `ALF_ENDIAN_ORDER_BIG`
  - `ALF_ENDIAN_ORDER_LITTLE`
- `ALF_QUERY_HOST_ENDIAN_ORDER`: The endianness or byte ordering of data within the host node. The value is either:
  - `ALF_ENDIAN_ORDER_BIG`
  - `ALF_ENDIAN_ORDER_LITTLE`

`accel_type [IN]` Accelerator type. When the system does not have the specified accelerator type, the query value is zero and the return value is zero.

- `ALF_ACCEL_TYPE_SPE`: The accelerator is SPU of a Cell/B.E. processor
- `ALF_ACCEL_TYPE_EDP`: The accelerator is eDP SPU of a IBM PowerXCell 8i processor

`p_query_result [OUT]` Pointer to the variable where the value of the query is returned. If a NULL pointer is provided, the value is not returned. If the query fails, the value is undefined.

DESCRIPTION

This function queries basic system information for the specific accelerator type from the ALF runtime.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Unsupported query • `ALF_BADF`: Invalid ALF handle • `ALF_ERR_GENERIC`: Generic internal errors

alf_num_instances_set function

NAME

alf_num_instances_set - Sets the maximum number of parallel task instances that the ALF instance can have at one time.

SYNOPSIS

int alf_num_instances_set(alf_handle_t alf_handle, unsigned int number_of_instances);

Parameters

`alf_handle [IN]`	A handle to the ALF instance.
`number_of_instances [IN]`	Specifies the maximum number of task instances. When this parameter is zero, the ALF instance sets no limit.

DESCRIPTION

This function sets the maximum number of parallel task instances that the ALF instance can have at one time. The number must be within the allowed resource ranges. If `number_of_instances` is zero, the ALF instance sets no limit and later it returns an error if it cannot accommodate a particular task creation request for a given number of task instances. For an isolated ALF runtime created by `alf_init`, this function should be called once. For a shared ALF instance created by `alf_init_shared`, this function can be called multiple times and the number of task instances is the maximum of the `number_of_instances` of all of the `alf_num_instances_set` calls.

RETURN VALUE

> 0	The actual number of parallel task instances provided by the ALF instance.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted at the current context

alf_num_instances_query function

NAME

alf_num_instances_query - Queries the maximum number of parallel task instances that the ALF instance can have at one time.

SYNOPSIS

int alf_num_instances_query(alf_handle_t alf_handle);

Parameters

`alf_handle [IN]`	A handle to the ALF instance.

DESCRIPTION

This function queries the maximum number of parallel task instances that the ALF instance can have at one time.

RETURN VALUE

>=0	The actual number of task instances from the ALF instance. If it is zero, the number has not been set.
less than 0	Errors occurred: • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted at the current ALF context

alf_exit function

NAME

alf_exit - Terminates an ALF instance.

SYNOPSIS

int alf_exit(alf_handle_t alf_handle, ALF_EXIT_POLICY_T policy, int timeout);

Parameters

`alf_handle [IN]`
The handle to the ALF instance.

`policy [IN]`
Specifies the shutdown behavior policy:

- `ALF_EXIT_POLICY_FORCE`: Performs a shutdown immediately and stops all unfinished tasks if there are any.
- `ALF_EXIT_POLICY_WAIT`: Waits for all tasks to be processed and then shut down.
- `ALF_EXIT_POLICY_TRY`: Returns with a `ALF_ERR_BUSY` if there are any unfinished tasks.

`time_out [IN]`
A timeout value that has the following values:

- > 0 : Wait at most the specified milliseconds before a timeout error happens or a forced shutdown
- = 0 : Shutdown or return without wait
- less than 0 : Waits forever, only valid with `ALF_EXIT_POLICY_WAIT`

DESCRIPTION

This function terminates the ALF instance. It kills all running or pending tasks for this ALF handle depending on the `policy` parameter. If the ALF handle refers to an isolated ALF instance, the ALF instance is also destroyed at the same time. For a shared ALF instance, it does not affect any tasks associated with other ALF handles that refer to the same ALF instance. The shared ALF instance is destroyed only if there are no more ALF handles that refer to this ALF instance.

RETURN VALUE

>= 0
The shutdown succeeded. The number of unfinished work blocks is returned.

less than 0
The shutdown failed:

- `ALF_ERR_INVAL`: Invalid input argument
- `ALF_ERR_BADF`: Invalid ALF handle
- `ALF_ERR_PERM`: The API call is not permitted at the current context
- `ALF_ERR_NOSYS`: The required policy is not supported
- `ALF_ERR_TIME`: Timeout
- `ALF_ERR_BUSY`: There are tasks still running

alf_error_handler_register function

NAME

alf_error_handler_register - Registers a error handler function to the ALF instance.

SYNOPSIS

int alf_error_handler_register(alf_handle_t alf_handle, alf_error_handler_t error_handler_function, void *p_context)

Parameters

`alf_handle [IN]`	A handle to the ALF instance.
`error_handler_function [IN]`	A pointer to the user-defined error handler function. A NULL value resets the error handler to the ALF default handler.
`p_context [IN]`	A pointer to the user-defined context data for the error handler function. This pointer is passed to the user-defined error handler function when it is invoked.

DESCRIPTION

This function registers an error handler function for the specified ALF instance. If no error handler function has ever been registered, then the ALF default handler is automatically registered. If an error handler has already been registered, the new one replaces the current one. If a NULL error handler is registered, the ALF default handler is registered. For a shared ALF instance, ALF tracks the source of errors and redirect errors to the registered error handler of a specific ALF handle. However, in the case of an error that is not specific, all registered callbacks are invoked and the return values is checked to see if they are the same. If any of the return values differs then the final effective value is `ALF_ERR_POLICY_ABORT`.

RETURN VALUE

0	Successful.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_FAULT`: Invalid buffer or error handler address (only when it is possible to detect the fault)

alf_error_handler_t function prototype

NAME

alf_error_handler_t - Function prototype defining a callback function which can be registered and invoked by the ALF runtime for customized error handling.

SYNOPSIS

alf_error_handler_t(*alf_error_handler_t)(void *p_context_data, int error_type, int error_code, char *error_string)

Parameters

`p_context_data [IN]` A pointer to context data that was passed to the ALF instance when the error handler is registered. The ALF instance passes it to the error handler when the error handler is invoked. The error handler can use it to keep private data.

`error_type [IN]` A system-wide definition of error type codes, including the following:

- `ALF_ERR_FATAL`: You cannot continue, ALF must shut down.
- `ALF_ERR_EXCEPTION`: You can choose to retry or skip the current operation.
- `ALF_ERR_WARNING`: You can choose to continue by ignoring the error.

`error_code [IN]` A type-specific error code.

`error_string [IN]` A C string that holds a printable text string that provides information about the error. See "USAGE" for more information.

DESCRIPTION

This function prototype defines a callback function that can be registered to the ALF instance for customized error handling.

RETURN VALUE

`ALF_ERR_POLICY_RETRY`	Indicates that the ALF instance should retry the operation that caused the error. If a severe error occurs and the ALF instance cannot retry this operation, it reports an error and shuts down.
`ALF_ERR_POLICY_SKIP`	Indicates that the ALF instance should skip the operation that caused the error and continue processing. If the error is severe and the ALF instance cannot continue, it reports an error and shuts down.
`ALF_ERR_POLICY_ABORT`	Indicates that the ALF instance must stop all operations and shut down.
`ALF_ERR_POLICY_IGNORE`	Indicates that the ALF instance ignores the error and continues. If the error is severe and the ALF instance cannot continue, it reports an error and shuts down.

USAGE

`error_string [IN]`

The string has the following format:

```
ALF <error_type>: <error_code> 'task_handle: <task_handle>,
instance_id: <instance_id>, number_of_instances: <number_of_instances>,
extra error code: <extra_error_code>, error reason: <error_reason>,
error sub reason: <error_sub_reason>'
```

where:

- <error_type> is either "runtime warning", "runtime exception" or "runtime error".
- <error_code> is defined in Appendix F, "Error codes and descriptions," on page 191.
- <task_handle> is the hexadecimal value of the task handle.
- <instance_id> is the decimal value of the identifier of the task instance.
- <number_of_instances> is the decimal value of the total number of task instances for this task.
- <extra_error_code> is the hexadecimal value.

  If the <error_code> is ALF_ERR_ACCEL, then it is zero (0)

  otherwise:

  if the <error_type> is runtime exception, then it is the return value of a work block task function, that is, task context setup, input data preparation, computational kernel, output data preparation, and task context merge, or it is the return value of the lightweight task **alf_accel_lts_main** function

  if the <error_type> is runtime error, then it is 0xNNNNLLLL where NNNN is an ALF SPU runtime file number and LLLL is the line number.
- <error_reason> is the hexadecimal value.

  if the <error_code> is ALF_ERR_ACCEL then it is (see *SPE Runtime Management Library Programmer's Guide and API Reference* for more details about the **spe_stop_info_t**'s stop_reason and result)

  0x00000003 - SPE_RUNTIME_ERROR

  <error_sub_reason> is:

  – 0x00000004 - SPE_SPU_HALT
  – 0x00000010 - SPE_SPU_SINGLE_STEP
  – 0x00000020 - SPE_SPU_INVALID_INSTR
  – 0x00000040 - SPE_SPU_INVALID_CHANNEL

  0x00000004 - SPE_RUNTIME_EXCEPTION

  <error_sub_reason> is:

  – 0x00000008 - SPE_DMA_ALIGNMENT
  – 0x00000020 - SPE_DMA_SEGMENTATION
  – 0x00000040 - SPE_DMA_STORAGE
  – 0x00000800 - SPE_INVALID_DMA

  0x00000005 - SPE_RUNTIME_FATAL. <error_sub_reason> contains the (implementation-dependent) errno as set by the underlying system call that failed.

  0x00000006 - SPE_CALLBACK_ERROR. <error_sub_reason> contains the return code from the failed library callback or it is set to -1 in the case of an unregistered library callback.

  0x00000007 - SPE_ISOLATION_ERROR. <error_sub_reason> contains the implementation-dependent error code fro the failed starting of an isolated SE program. otherwise it is zero (0).
- <error_sub_reason> is the hexadecimal value.

  if the <error_code> is ALF_ERR_ACCEL then see <error_sub_reason> under each <error_reason> above (see *SPE Runtime Management Library Programmer's Guide and API Reference* for more details about the **spe_stop_info_t**'s stop_reason and result) otherwise it is minus one (-1).

For example:

```
ALF runtime error: 2000 'task_handle: 0x2570310, instance_id: 1,
number_of_instances: 5, extra error code: 0x00000000,
error reason: 0x00000004, error sub reason: 0x00000040'
```

The is an ALF fatal runtime error with an error code of 2000 (this error code has an enumeration value of ALF_ERR_ACCEL and an alf_strerror value of ″Generic accelerator error″). This error occurred when executing task 0x2570310 in task instance 1 of 5. Because the error code is ALF_ERR_ACCEL the extra error code is ignored, the error reason indicates that a ″SPE Runtime Management Library″ stop reason of SPE_RUNTIME_EXCEPTION occurred. and the error sub-reason indicates that a ″SPE Runtime Management Library″ result of SPE_DMA_STORAGE which is caused by a invalid DMA storage address.

Compute task API

This topic describes the definitions are the compute task functions and data types.

The following functions and data types are described:

alf_task_handle_t data type

NAME

alf_task_handle_t - This data type is a handle to a specific compute task bound to one or more accelerators as task instances.

DESCRIPTION

This data type is created by calling the `alf_task_create` function. It is used to wait for the task to finish processing all queued work blocks by calling the `alf_task_wait` function. It is also used when indicating to the ALF instance that no new work blocks are to be added to the work queue of the corresponding task by calling the `alf_task_finalize` function. It is implicitly destroyed by calling the `alf_exit` function. It is explicitly destroyed by calling the `alf_task_destroy` function, although that is typically not done.

alf_task_desc_handle_t data type

NAME

alf_task_desc_handle_t - This data type is a handle to a task descriptor.

DESCRIPTION

This data type is used to specify task descriptor information. It is created by calling `alf_task_desc_create` function. It is used to updated the task descriptor by calling the `alf_task_desc_set_int32` and the `alf_task_desc_set_int64` functions. It is destroyed by calling `alf_task_desc_destroy`.

alf_task_desc_create function

NAME

alf_task_desc_create - Creates a task descriptor.

SYNOPSIS

int alf_task_desc_create (alf_handle_t alf_handle, ALF_ACCEL_TYPE_T accel_type, alf_task_desc_handle_t * p_desc_info_handle);

Parameters

`alf_handle`	A handle to an ALF instance.
`accel_type [IN]`	The accelerator type of this task descriptor. Tasks created from this task descriptor run on this accelerator type. The type can be either ALF_ACCEL_TYPE_SPE or ALF_ACCEL_TYPE_EDP.
`p_task_desc_handle [OUT]`	A pointer to the handle of the created task descriptor. If it fails, the contents of the pointer are not modified.

DESCRIPTION

This function creates a task descriptor and returns a handle to it. The task descriptor contains all the information needed to create a compute task.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_NOMEM`: Out of memory or system resource • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_GENERIC`: Generic internal errors

alf_task_desc_destroy function

NAME

alf_task_desc_destroy - Destroys a task descriptor.

SYNOPSIS

int alf_task_desc_destroy (alf_task_desc_handle_t task_desc_handle);

Parameters

`task_desc_handle [IN/OUT]` A handle to the task descriptor. The task descriptor referenced by this handle is destroyed.

DESCRIPTION

This function destroys a task descriptor referenced by the specified handle, and frees up its resources. If it is being used by a task then its task descriptor cannot be destroyed and an error results.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid task descriptor handle. • `ALF_ERR_BUSY`: This task descriptor is being used by one or more tasks. All tasks using this task descriptor must be destroyed before you can destroy this task descriptor. • `ALF_ERR_PERM`: The API call is not permitted at the current context. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_desc_ctx_entry_add function

NAME

alf_task_desc_ctx_entry_add - Adds a task context entry to the task descriptor.

SYNOPSIS

int alf_task_desc_ctx_entry_add (alf_task_desc_handle_t task_desc_handle, ALF_DATA_TYPE_T data_type, unsigned int size);

Parameters

`task_desc_handle [IN]`	A handle to a task descriptor
`data_type [IN]`	The data type of each element for this entry
`size [IN]`	The number of elements for this entry. Each element is of type data_type.

DESCRIPTION

This function adds a task context entry to the task descriptor. Each addition describes the type and size of one entry of the task context data. Multiple additions describe the structure of the task context data. The total size of the task context data must be specified by calling the `alf_task_desc_set_int32` function and setting the `ALF_TASK_DESC_TSK_CTX_SIZE` field. The address of the task context data is specified when calling the `alf_task_create` function.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid task descriptor handle • `ALF_ERR_NOSYS`: The `ALF_DATA_TYPE_T` provided is not supported. • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_NOBUFS`: The requested entry has exceeded the maximum buffer size • `ALF_ERR_GENERIC`: Generic internal errors

alf_task_desc_set_int32 function

NAME

alf_task_desc_set_int32 - Sets a 32-bit value into a task descriptor field.

SYNOPSIS

int alf_task_desc_set_int32 (alf_task_desc_handle_t task_desc_handle, ALF_TASK_DESC_FIELD_T field, unsigned int value);

Parameters

`task_desc_handle [IN/OUT]` A handle to a task descriptor

`field [IN]`

The field to be set. Possible inputs are

- `ALF_TASK_DESC_TASK_TYPE`: The type of the task, The following is defined for the *value* parameter:
  - `ALF_TASK_TYPE_WORKBLOCK`: Work block task (default if the task type is not specified)
  - `ALF_TASK_TYPE_LIGHTWEIGHT`: Lightweight task
  - The default value is `ALF_TASK_TYPE_WORKBLOCK`.

The following are used by both work block tasks and lightweight tasks:

- `ALF_TASK_DESC_TSK_CTX_SIZE`: Size of the task context buffer. The address of the buffer is passed to `alf_task_create` as the `p_task_context_data` parameter. The default value is zero (0).

The following are only for work block tasks. They are ignored for lightweight tasks without causing an error.

- `ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE`: size of the work block parameter buffer. The default value is zero (0).
- `ALF_TASK_DESC_WB_IN_BUF_SIZE`: size of the work block input buffer. The default value is zero (0).
- `ALF_TASK_DESC_WB_OUT_BUF_SIZE`: size of the work block output buffer. The default value is zero (0)
- `ALF_TASK_DESC_WB_INOUT_BUF_SIZE`: size of the work block overlapped input/output buffer. The default value is zero (0)
- `ALF_TASK_DESC_NUM_DTL`: maximum number of data transfer lists for a single work block in the task. Zero 0 is a valid value. The default value is 10. The maximum value is 128.
  Note: Internally, data transfer lists that cross a 4 GB boundary or exceed the maximum number of entries per list (2048) are broken up into separate lists. This value must account for that.
- `ALF_TASK_DESC_NUM_DTL_ENTRIES`: maximum number of entries for all of the data transfer lists of a single work block in the task. Zero (0) is a valid value. The default value is 128.
  Note: Internally, entries larger than 16 KB are broken up into separate entries. This value must account for that.
- `ALF_TASK_DESC_PARTITION_ON_ACCEL`: non-zero value specifies accelerator data partitioning. That is, the accelerator functions (`alf_accel_input_dtl_prepare` and `alf_accel_output_dtl_prepare`) are invoked to generate data transfer lists input and output data if a work block. A zero (0) value specifies host data partitioning. The default value is zero (0).
- `ALF_TASK_DESC_MAX_STACK_SIZE`: maximum size of the accelerator runtime stack. The default value is 1024.

`value [IN]`

New value of the specified field

DESCRIPTION

This function sets the value for a specific integer field of the task descriptor. The default value of an un-set field is zero (0).

RETURN VALUE

0	Successful
less than 0	Errors occurred: • ALF_ERR_INVAL: Invalid input argument • ALF_ERR_BADF: Invalid task descriptor handle • ALF_ERR_NOSYS: The ALF_TASK_DESC_FIELD provided is not supported. • ALF_ERR_PERM: The API call is not permitted at the current context • ALF_ERR_RANGE: The specified value is out of the allowed range • ALF_ERR_GENERIC: Generic internal errors

alf_task_desc_set_int64 function

NAME

alf_task_desc_set_int64 - Sets the value for a specific long integer field of the task descriptor structure.

SYNOPSIS

int alf_task_desc_set_int64(alf_task_desc_handle_t task_desc_handle, ALF_TASK_DESC_FIELD_T field, unsigned long long value);

Parameters

`task_desc_handle [IN/OUT]` Handle to the task descriptor structure

`field [IN]`

The field to be set. Possible inputs are:

The following are for both work block tasks and lightweight tasks:

- `ALF_TASK_DESC_ACCEL_LIBRARY_REF_L`: Name of the library that contains the accelerator executable image. The library name is appended to each library path entry to create a fully-qualified shared object library file name to be loaded. The library path is specified in the ALF runtime using the `alf_init` or `alf_init_shared` function with a configuration parameter or in the execution environment using the `ALF_LIBRARY_PATH` environment variable. If the library path is an empty string then nothing is prepended to the library name, and it is located and loaded using the default operating system shared object library search protocol. For example, for Linux the entries in the `LD_LIBRARY_PATH` environment variable are used. If the library name value is NULL then the library path is ignored, no library is loaded and the accelerator executable image is searched for in the currently loaded shared object libraries. The default value is zero (0) or NULL.
- `ALF_TASK_DESC_ACCEL_IMAGE_REF_L`: Name of the accelerator executable image. For Cell/B.E. this is the symbol name of the PPU object containing an embedded SPU executable image. The default value is zero (0) or NULL.

The following are for work block tasks:

- `ALF_TASK_DESC_ACCEL_KERNEL_REF_L`: Name of the computational kernel function, this usually is a string constant that the accelerator runtime could use to find the correspondent function. The default value is zero (0) or NULL.
- `ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L`: Name of the input list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function. The default value is zero (0) or NULL.
- `ALF_TASK_DESC_ACCEL_OUTUT_DTL_REF_L`: Name of the output list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function. The default value is zero (0) or NULL.
- `ALF_TASK_DESC_ACCEL_CTX_SETUP_REF_L`: Name of the context setup function, this usually is a string constant that the accelerator runtime could use to find the correspondent function. The default value is zero (0) or NULL.
- `ALF_TASK_DESC_ACCEL_CTX_MERGE_REF_L`: Name of the context merge function, this usually a string constant that the accelerator runtime could use to find the correspondent function. The default value is zero (0) or NULL.

The following are for lightweight tasks:

- `ALF_TASK_DESC_ACCEL_LTS_MAIN_REF_L`: Name of the main entry point function of the lightweight task. This usually is a string constant that the accelerator runtime uses to find the corresponding function pointer. The default value is zero (0) or NULL.

`value [IN]`

New value of the specified field

DESCRIPTION

This function sets the value for a specific long integer field of the task descriptor structure. All string constants must have a maximum number of `ALF_STRING_TOKEN_MAX` size.

RETURN VALUE

0	Successful
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid task descriptor handle • `ALF_ERR_NOSYS`: The `ALF_TASK_DESC_FIELD` provided is not supported. • `ALF_ERR_PERM`: The API call is not permitted at the current context • `ALF_ERR_RANGE`: The specified value is out of the allowed range • `ALF_ERR_GENERIC`: Generic internal errors

alf_task_create function

NAME

alf_task_create - Creates a task and allows you to add work blocks to the work queue of the task.

SYNOPSIS

int alf_task_create(alf_task_desc_handle_t task_desc_handle, void* p_task_context_data, unsigned int num_instances, unsigned int tsk_attr, unsigned int wb_dist_size, alf_task_handle_t *p_task_handle);

Parameters

`task_desc_handle [IN]`
Handle to a `task_desc` structure.

`p_task_context_data [IN]`
Pointer to the task context data for this task. The structure and size for the task context have been defined through `alf_task_desc_add_task_ctx_entry`. If there is no `task_context`, a NULL pointer can be provided. If the task context is defined in the task descriptor but a NULL value is provided, the content of the task context buffer on the accelerator is not initialized. For a work block task, it is possible to use the `alf_accel_context_setup` auxiliary function to do the initialization.

`num_instances [IN]`
The suggested maximum number of instances of the task. The valid range of `num_instances` is from 0 to the maximum number of instances. If the `num_instances` is zero then the maximum number of instances is used. The maximum number of instances is specified by either a previous call to `alf_num_instances_set` or if no previous call then the physical number of accelerators on the system. If `ALF_TASK_ATTR_SCHED_FIXED` is set the `num_instances` must be between 1 and the maximum number of instances.

`tsk_attr [IN]`
Attribute for a task. This value can be set to a bit-wise OR to one of the following:

- `ALF_TASK_ATTR_SCHED_FIXED`: The task is scheduled on a fixed number of accelerators as specified by `num_instances`. By default, a task can be scheduled on any number of accelerators less or equal to what is specified by `num_instance` and the actual number of accelerators can be adjusted any time during the execution of the task.
- `ALF_TASK_ATTR_WB_CYCLIC`: (this attribute is ignored if the task type is lightweight) the work blocks for this task are distributed to the accelerators in a cyclic order as specified by `num_instances`. By default, the work blocks distribution order is determined by the ALF runtime. This option must be used combined with `ALF_TASK_ATTR_SCHED_FIXED`.

`wb_dist_size [IN]`
The specified block distribution bundle size in number of work blocks per distribution unit. A 0 (zero) value is treated as 1 (one). Refer to “Work block scheduling” on page 19 for more details about work block distribution.

(this parameter is ignored if the task type is lightweight)

`p_task_handle [OUT]`
Returns a handle to the created task. The content of the pointer is not modified if the call returns failure.

DESCRIPTION

Work block task

This function creates a task and allows you to enqueue work blocks to the task. The task remains in a pending status until the following condition is met: All dependencies are satisfied and either at least one work block is added or the task is finalized by calling `alf_task_finalize`.

When the condition is met, the task becomes ready to run. However, when the task actually starts to run depends accelerator binary compatibility, the available accelerator resources and the scheduling of ALF runtime. Multiple independent tasks can also run concurrently if there are enough accelerator resources. When the task starts to run, it keeps running until at least one of the following two conditions is met:

- The task has been finalized by calling `alf_task_finalize` and all the enqueued work blocks are processed and the task context has been merged and written back;
- `alf_task_destroy` is called to explicitly destroy the task.

Note: A finalized task without any work block enqueued is never actually loaded and run. The runtime considers this task as completed immediately after the dependencies are satisfied.

Lightweight task

This function creates a task. The task remains in a pending status until the following condition is met: All dependencies are satisfied and the task is finalized by calling `alf_task_finalize`.

When the condition is met, the task becomes ready to run. However, when the task actually starts to run depends on the available accelerator resources and the scheduling of ALF runtime. Multiple independent tasks can also run concurrently if there are enough accelerator resources. When the task starts to run, it keeps running until at least one of the following two conditions is met:

- All instances of the task has exited
- `alf_task_destroy` is called to explicitly destroy the task

Note: The `alf_task_destroy` may not be able to destroy a running task properly with some implementations when the accelerator side programmer does not invoke `alf_accel_test_cancel`.

RETURN VALUE

0	Successful

less than 0	Errors occurred: • ALF_ERR_INVAL: Invalid input argument • ALF_ERR_BADF: Invalid ALF handle • ALF_ERR_NOMEM: Out of memory or system resource • ALF_ERR_PERM: The API call is not permitted at the current context • ALF_ERR_NOEXEC: Invalid task image format or description information, for example, wrong task type or different accelerator targets and so on • ALF_ERR_2BIG: Memory requirement for the task exceeds maximum range • ALF_ERR_NOSYS: The required task attribute is not supported • ALF_ERR_BADR: The requested number of accelerator resources is not available • ALF_ERR_GENERIC: Generic internal errors • ALF_ERR_INCOMPAT: The accelerator binary file is not compatible.

alf_task_finalize function

NAME

alf_task_finalize - Finalizes a task description. For a work block task it also closes the work block queue for the specified task.

SYNOPSIS

int alf_task_finalize (alf_task_handle_t task_handle)

Parameters

`task_handle [IN]` The task handle that is returned by the `alf_create_task` API

DESCRIPTION

This function finalizes the task. After the task has been finalized, future calls to `alf_wb_create` and `alf_task_depends_on` and `alf_task_event_handler_register` return errors.

Note: Task finalization is a compulsory condition for a task to run and complete normally.

RETURN VALUE

less than 0 — Errors occurred:

- `ALF_ERR_BADF`: Invalid task descriptor handle.
- `ALF_ERR_SRCH`: Already finalized task handle.
- `ALF_ERR_PERM`: The API call is not permitted at the current context. For example, some created work block handles are not enqueued.
- `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_wait function

NAME

alf_task_wait - Waits for the specified task to complete.

SYNOPSIS

int alf_task_wait(alf_task_handle_t task_handle, int time_out);

Parameters

`task_handle [IN]` A task handle that is returned by the `alf_create_task` API.

`time_out [IN]` A timeout input with the following options for values:

- > 0: Waits for up to the number of milliseconds specified before a timeout error occurs.
- less than 0: Waits until all of the accelerators finish processing.
- 0: Returns immediately.

DESCRIPTION

This function waits for the specified task to finish. For a work block task, it is to finish processing all work blocks on all the scheduled accelerators. For a lightweight task, the call to the computational kernel `alf_accel_lts_main` on each instance must be returned. For a work block task, it waits until all the work blocks on all the scheduled accelerators are complete. The task must be finalized (`alf_task_finalize` must be called) before this function is called. Otherwise, an `ALF_ERR_PERM` is returned. Data referenced by the task's work blocks can only be used safely after this function returns. If the host application updates the data buffers referenced by work blocks or the task context buffer while the task is running, the result can be undetermined. If you need to update the buffer contents, the only safe point is before the `ALF_TASK_EVENT_READY` task event is handled by the task event handler registered by `alf_task_event_handler_register`.

RETURN VALUE

0 All of the accelerators finished the job.

less than 0 Errors occurred:

- `ALF_ERR_INVAL`: Invalid input argument.
- `ALF_ERR_BADF`: Invalid task handle.
- `ALF_ERR_NODATA`: The task is (during wait) or was (before wait) destroyed explicitly.
- `ALF_ERR_TIME`: Timeout.
- `ALF_ERR_PERM`: The API is not permitted at the current context. For example, the task is not finalized.
- `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_query function

NAME

alf_task_query - Queries the current status of a task.

SYNOPSIS

int alf_task_query(alf_task_handle_t task_handle, unsigned int *p_unfinished_wbs, unsigned int *p_total_wbs);

Parameters

`task_handle [IN]`	The task handle to be checked.
`p_unfinished_wbs [OUT]`	A pointer to an integer buffer where the number of unfinished work blocks of this task is returned. When a NULL pointer is given, the return value is ignored. On error, a returned value is not defined. This parameter is ignored for lightweight tasks. A zero returns when the task has been completed before the query.
`p_total_wbs [OUT]`	A pointer to an integer buffer where the total number of submitted work blocks of this task is returned. When a NULL pointer is given, the return value is ignored. On error, a returned value is not defined. This parameter is ignored for lightweight tasks. A zero returns when the task has been completed before the query.

DESCRIPTION

This function queries the current status of a task.

RETURN VALUE

> 1	The task is pending or ready to run.
1	The task is currently running.
0	The task finished normally.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_NODATA`: The task was explicitly destroyed. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_destroy function

NAME

alf_task_destroy - Abnormally stops and destroys the specified task.

SYNOPSIS

int alf_task_destroy(alf_task_handle_t* p_task_handle)

Parameters

`task_handle [IN]`	The pointer to a task handle that is returned by the `alf_create_task` API.

DESCRIPTION

This function explicitly destroys the specified task if it is in pending or running state. For a work block task, if there are work blocks that are still not processed, this routine stops the execution of those work blocks. If a task is running when this API is invoked, the task is cancelled before the API returns. Resources associated with this task are recycled by the runtime either synchronously or asynchronously, depending on the runtime implementation. This API does nothing on an already completed task. If a task is destroyed explicitly, all tasks that depend on this task directly or indirectly are destroyed. Because ALF frees task resources automatically, it is not necessary to call this API to free up resources after a task has been run to complete normally. The API should only be used to explicitly end a task when you need to handle an abnormal situation. Do not use it to free up task resources when the task has run to complete.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_PERM`: The API call is not permitted at current context. • `ALF_ERR_BUSY`: Resource busy. • `ALF_ERR_SRCH`: Already destroyed task handle. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_depends_on function

NAME

alf_task_depends_on - Describes a relationship between two tasks.

SYNOPSIS

int alf_task_depends_on (alf_task_handle_t task_handle_dependent, alf_task_handle_t task_handle);

Parameters

`task_handle_dependent [IN]`	The handle to the dependent task
`task_handle [IN]`	The handle to a task

DESCRIPTION

This function describes a relationship between two tasks. The task specified by `task_handle_dependent` cannot be scheduled to run until the task specified by `task_handle` has run to finish normally. When this API is called, `task_handle` must not be an explicitly destroyed task. An error is reported if it is the case. If the task associated with `task_handle` is destroyed before normal completion, the `task_handle_dependent` is also destroyed because its dependency can no longer be satisfied.

If task A depends on task B, a call to `alf_task_wait` (A_handle) effectively enforces a wait on task B as well. A duplicate dependency is handled silently and not treated as an error.

Refer to “Task dependency and task scheduling” on page 15 for more information on task dependency and limitations on when the task dependencies can be set.

Note: This function can only be called before any work blocks are enqueued to the `task_handle_dependent` and before the `task_handle_dependent` is finalized. For the `task_handle`, these constraints are not applicable.

Whenever a situation occurs that is not permitted, the function returns `ALF_ERR_PERM`.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_PERM`: The API call is not permitted at the current context. For example, the dependency cannot be set because of the current state of the task. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_event_handler_register function

NAME

alf_task_event_handler_register - Allows you to register and unregister an event handler for a specific task.

SYNOPSIS

int alf_task_event_handler_register (alf_task_handle_t task_handle, int (*task_event_handler)(alf_task_handle_t task_handle, ALF_TASK_EVENT_TYPE_T event, void* p_data), void* p_data, unsigned int data_size, unsigned int event_mask);

Parameters

`task_handle [IN]` The handle to a task.

`task_event_handler [IN]` Pointer of the event handler function for the specified task. A NULL value indicates the current event handler is to be unregistered.

`p_context [IN]` A pointer to a context buffer that is copied to another buffer managed by the ALF runtime. The pointer to this buffer is passed to the event handler. The content of the context buffer is copied by value only. A NULL value indicates no context buffer.

`context_size [IN]` The size of the context buffer in bytes. Zero indicates no context buffer.

`event_mask [IN]` A bitwise OR of `ALF_TASK_EVENT_TYPE_T` values. `ALF_TASK_EVENT_TYPE_T` is defined as follows:

- `ALF_TASK_EVENT_FINALIZED`: This task has been finalized. No additional work block can be added to this task. The registered event handler is invoked right before `alf_task_finalize` returns.
- `ALF_TASK_EVENT_READY`: This task has been scheduled for execution. The registered event handler is invoked as soon as the ALF runtime determines that all dependencies have been satisfied for this specific task and can schedule this task for execution as soon as this event handler returns.
- `ALF_TASK_EVENT_FINISHED`: All work blocks in this task have been processed. The registered event handler is invoked as soon as the last work block has been processed and the task context and dataset (if associated) have been written back to host memory.
- `ALF_TASK_EVENT_INSTANCE_START`: One new instance of the task is started on an accelerator after the event handler returns.
- `ALF_TASK_EVENT_INSTANCE_END`: One existing instance of the task ends and the task context has been copied out to the original location or has been merged to another current instance of the same task. The event handler is called as soon as the task instance is ended and unloaded from the accelerator.
- `ALF_TASK_EVENT_DESTROY`: The task is destroyed explicitly.

DESCRIPTION

This function allows you to register an event handler for a specified task. This function can only be called before `alf_task_finalize` is invoked. An error is returned if a you try to register an event handler for a task that has been finalized.

If the `task_event_handler` function is NULL, this function unregisters the current event handler. If there is no current event handler, nothing happens.

Note: If the event handler is registered after the task begin to run, some of the events may not be seen.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input handle. • `ALF_ERR_BADF`: Invalid ALF task handle. • `ALF_ERR_PERM`: The API call is not permitted at the current context. • `ALF_ERR_NOMEM`: Out of memory. • `ALF_ERR_FAULT`: Invalid buffer or error handler address (only when it is possible to detect the fault). • `ALF_ERR_GENERIC`: Generic internal errors.

Work block API

This topic describes the definitions for the work block functions and data types.

The following functions and data types are described:

alf_wb_handle_t data type

NAME

alf_wb_handle_t - This data structure refers to the work block being constructed by the control node.

DESCRIPTION

This data structure refers to a work block being constructed by the control node. It is created by calling `alf_wb_create`.

alf_wb_sync_handle_t data type

NAME

alf_wb_sync_handle_t - This data structure refers to the synchronization point.

alf_wb_create function

NAME

alf_wb_create - Creates a new work block for the specified compute task.

SYNOPSIS

int alf_wb_create(alf_task_handle_t task_handle, ALF_WORK_BLOCK_TYPE_T work_block_type, unsigned int repeat_count, alf_wb_handle_t *p_wb_handle);

Parameters

`task_handle` `[IN]`	The handle to the compute task.
`work_block_type` `[IN]`	The type of work block to be created. Choose from the following types: • `ALF_WB_SINGLE`: Creates a single-use work block • `ALF_WB_MULTI`: Creates a multi-use work block. This work block type is only supported when the task is created with the `ALF_PARTITION_ON_ACCEL` attribute.
`repeat_count` `[IN]`	Specifies the number of iterations for a multi-use work block. This parameter is ignored when a single-use work block is created.
`p_wb_handle` `[OUT]`	The pointer to a buffer where the created handle is returned. The contents are not modified if this call fails.

DESCRIPTION

This function creates a new work block for the specified computing task. The work block is added to the work queue of the task and the runtime releases the allocated resources once the work block is processed. The caller can only update the contents of a work block before it is added to the work queue. After the work block is added to the work queue, the lifespan of the data structure is left to the ALF runtime. The ALF runtime is responsible for cleaning up any resource allocated for the work block. This API can only be called before `alf_task_finalize` is invoked. After the `alf_task_finalize` is called, further calls to this API return an error.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed in current context. For example, the task has already been finalized or the work block has been enqueued. • `ALF_ERR_BADF`: Invalid task handle. • `ALF_ERR_NOMEM`: Out of memory. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_wb_enqueue function

NAME

alf_wb_enqueue - Adds the work block to the work queue of the specified task handle.

SYNOPSIS

int alf_wb_enqueue(alf_wb_handle_t wb_handle)

Parameters

`wb_handle [IN]` The handle of the work block to be put into the work queue.

DESCRIPTION

This function adds the work block to the work queue of the specified task handle. The caller can only update the contents of a work block before it is added to the work queue. After it is added to the work queue, you cannot access the `wb_handle`.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid task handle or work block handle • `ALF_ERR_PERM`: Operation not allowed in current context • `ALF_ERR_BUSY`: An internal resource is occupied • `ALF_ERR_GENERIC`: Generic internal errors

alf_wb_parm_add function

NAME

alf_wb_parm_add - Adds the given parameter to the parameter and context buffer of the work block in the order that this function is called.

SYNOPSIS

int alf_wb_parm_add(alf_wb_handle_t wb_handle, void *pdata, unsigned int size_of_data, ALF_DATA_TYPE_T data_type, unsigned int address_alignment)

Parameters

`wb_handle [IN]`	The work block handle.
`pdata [IN]`	A pointer to the data to be copied.
`size_of_data [IN]`	The size of the data in units of the data type.
`data_type [IN]`	The type of data. This value is required if data endianess conversion is necessary when moving the data.
`address_alignment [IN]`	Power of 2 byte alignment of 2 $^{address\_alignment}$. The valid range is from 0 to 16. A zero indicates a byte-aligned address. An 8 indicates alignment on 256 byte boundaries.

DESCRIPTION

This function adds the given parameter to the **parameter and context buffer** of the work block in the order that this function is called. The starting address is from offset zero. The added data is copied to the internal parameter and context buffer immediately. The relative address of the data can be aligned as specified. For a specific work block, additional calls to this API return an error after the work block is put into the work queue by calling the `alf_wb_enqueue` function.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed in current context. • `ALF_ERR_BADF`: Invalid task handle or work block handle. • `ALF_ERR_NOBUFS`: Some internal resource is occupied. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_wb_dtl_begin function

NAME

alf_wb_dtl_begin - Marks the beginning of a data transfer list for the specified target `buffer_type`.

SYNOPSIS

int alf_wb_dtl_begin (alf_wb_handle_t wb_handle, ALF_BUF_TYPE_T buffer_type, unsigned int offset_to_accel_buf);

Parameters

`wb_handle [IN]`	The work block handle.
`buffer_type [IN]`	The type of the buffer. Possible values are: • `ALF_BUF_IN`: Input to the input only buffer • `ALF_BUF_OUT`: Output from the output only buffer • `ALF_BUF_OVL_IN`: Input to the overlapped buffer • `ALF_BUF_OVL_OUT`: Output from the overlapped buffer • `ALF_BUF_OVL_INOUT`: In/out to/from the overlapped buffer
`offset_to_accel_buf [IN]`	Offset of the target buffer on the accelerator.

DESCRIPTION

This function marks the beginning of a data transfer list for the specified target `buffer_type`. Further calls to function `alf_wb_dtl_entry_add` refers to the currently opened data transfer list. You can create multiple data transfer lists per buffer type, however, only one data transfer list is opened for entry at any time for a specific work block there can be no nesting of data transfer list.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed. • `ALF_ERR_BADF`: Invalid work block handle. • `ALF_ERR_2BIG`: The offset to the accelerator buffer is larger than the size of the buffer. • `ALF_ERR_NOSYS`: The specified I/O type feature is not supported. • `ALF_ERR_BADR`: The requested buffer is not defined in the task context. • `ALF_ERR_GENERIC`: Generic internal errors. • `ALF_ERR_NOBUFS`: The internal data buffer is used up.

alf_wb_dtl_entry_add function

NAME

alf_wb_dtl_entry_add - Adds an entry to the input or output data transfer lists of a single use work block.

SYNOPSIS

int alf_wb_dtl_entry_add (alf_wb_handle_t wb_handle, void* host_addr, unsigned int size, ALF_DATA_TYPE_T data_type);

Parameters

`wb_handle [IN]`	The work block handle
`host_address [IN]`	The pointer (EA) to the data in remote memory
`size [IN]`	The size of the data in units of the data type
`data_type [IN]`	The type of data, this value is required if data endianess conversion is necessary when doing the data movement

DESCRIPTION

This function adds an entry to the input or output data transfer lists of a single use work block. The entry describes a single piece of data transferred from and to the remote memory. For a specific work block, further calls to this API return errors after the work block is put to work queue by calling `alf_wb_enqueue`.

For a specific work block, further calls to this API return error after the work block is put to work queue by calling `alf_wb_enqueue`. If the work block's task is associated with a dataset, the specified buffer with **host_addr** and **size** must be contained within the dataset. Adding a dtl entry describing a buffer that is outside the associated dataset returns a `ALF_ERR_PERM` error.

This function can only be called if the task descriptor associated with the work block's task is created with the task descriptor attribute `ALF_TASK_DESC_PARTITION_ON_ACCEL` set to false.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: Operation not allowed. • `ALF_ERR_BADF`: Invalid work block handle. • `ALF_ERR_2BIG`: Trying to add too many lists. • `ALF_ERR_NOBUFS`: The amount of data to move exceeds the maximum buffer size. • `ALF_ERR_FAULT`: Invalid host address (if it can be detected). • `ALF_ERR_GENERIC`: Generic internal errors.

alf_wb_dtl_end function

NAME

alf_wb_dtl_end - This function marks the ending of a data transfer list.

SYNOPSIS

int alf_wb_dtl_end (alf_wb_handle_t wb_handle);

Parameters

`wb_handle [IN]`	The work block handle

DESCRIPTION

This function marks the ending of a data transfer list.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • `ALF_ERR_PERM`: Operation not allowed. • `ALF_ERR_BADF`: Invalid work block handle.

alf_wb_sync function

NAME

alf_wb_sync - Adds a synchronization point.

SYNOPSIS

int alf_wb_sync (alf_task_handle_t task_handle, ALF_SYNC_TYPE_T sync_type, int (*sync_callback_func)(alf_wb_sync_handle_t sync_handle, void* p_context), void* p_context, unsigned int context_size, alf_wb_sync_handle_t *p_sync_handle);

Parameters

`task_handle [IN]`	Task handle.
`sync_type [IN]`	This can be set to one of the following values: • `ALF_SYNC_BARRIER`: When the ALF runtime reaches this synchronization point, all work blocks enqueued before this point must be finished before any new work blocks added after the synchronization point can be processed on any of the accelerators. If a callback function is registered to this synchronization point, the work queue continues to run only when the callback function returns. • `ALF_SYNC_NOTIFY`: The ALF accelerator runtime sends a notification to the ALF host runtime and invoke the registered callback function when this synchronization point is reached. However, it does not ensure the order of work block completion.
`sync_callback_func [IN]`	The pointer to the call back function that is registered for this synchronization point. This parameter can be NULL if you do not want a call back function.
`p_context [IN]`	A pointer to a context buffer that is **copied** to an internal buffer. The pointer to the internal buffer is passed to the callback function if there is a callback function registered. The content of the context buffer is copied by value only. A NULL value indicates no context buffer.
`context_size [IN]`	The size of the context buffer in bytes. Zero indicates no context buffer.
`p_sync_handle [IN/OUT]`	Pointer to buffer where the handle to the created synchronization point is returned.

DESCRIPTION

This function adds a synchronization point to the current work queue for the specified task. The programmer can register a callback function for this. ALF runtime invokes the callback function when the synchronization condition is met. This API can only be called before `alf_task_finalize` is invoked. After `alf_task_finalize` is called, further calls to the function return an error.

Note: For a synchronization point without an associated callback function, its behavior is always non-blocked. In this case, use `alf_wb_sync_wait` to check the status of the synchronization point. If a callback function is associated with the synchronization point, its behavior is always blocked, and the ALF runtime does not assign new work blocks to the accelerators until the callback function has returned. In either case, `alf_wb_sync_wait` is always supported.

RETURN VALUE

0	Success.
less than 0	Errors occurred: • ALF_ERR_PERM: Operation not allowed. • ALF_ERR_BADF: Invalid work block handle. • ALF_ERR_INVAL: Invalid input argument. • ALF_ERR_NOMEM: Out of memory. • ALF_ERR_BADR: Generic internal errors.

sync_callback_func function

NAME

sync_callback_func - The call back function for the synchronization point.

SYNOPSIS

int (*sync_callback_func)(alf_wb_sync_handle_t sync_handle, void *p_context)

Parameters

`sync_handle [IN]`	The handle to the synchronization point.
`p_context [IN]`	A pointer to the buffer where the programmer supplied context values are duplicated. The contents of this buffer are not kept after the callback function is returned.

DESCRIPTION

The callback function may be invoked in a different thread context than the main application.

RETURN VALUE

0	No errors.
less than 0	Errors occurred during the callback. An internal error with type `ALF_ERR_EXCEPTION` is raised.

alf_wb_sync_wait function

NAME

alf_wb_sync_wait - Waits for the arrival of a synchronization point.

SYNOPSIS

int alf_wb_sync_wait (alf_wb_sync_handle_t sync_handle, int time_out)

Parameters

`sync_handle [IN]` Task handle. This is the value returned from `alf_wb_sync`.

`time_out` Timeout value in milliseconds.

- > 0: The function waits for up to `time_out` milliseconds before a time out error occurs.
- 0: The function checks the status of the synchronization point and return immediately.
- less than 0: The function waits until the synchronization point is reached.

DESCRIPTION

This function waits for the arrival of a synchronization point. The status of a synchronization point can be queried multiple times. When a synchronization point is already reached calls to this API always return success.

RETURN VALUE

0 The synchronization operation completed successfully.

less than 0 Errors occurred:

- `ALF_ERR_PERM`: Operation not allowed.
- `ALF_ERR_BADF`: Invalid work block handle
- `ALF_ERR_INVAL`: Invalid input arguments
- `ALF_ERR_TIME`: Timout.

Data set API

This topic describes the definitions for the data set functions and data types.

- The following functions and data types are described:
- "alf_dataset_handle_t data type" on page 124
- "alf_dataset_create function" on page 125
- "alf_dataset_buffer_add function" on page 126
- "alf_dataset_destroy function" on page 127
- "alf_task_dataset_associate function" on page 128

alf_dataset_handle_t data type

NAME

alf_dataset_handle_t - This data structure is a handle for the data set.

alf_dataset_create function

NAME

alf_dataset_create - Creates a dataset.

SYNOPSIS

int alf_dataset_create(alf_handle_t alf_handle, alf_dataset_handle_t * p_dataset_handle);

Parameters

`alf_handle[in]`	Handle to the ALF runtime
`p_dataset_handle[out]`	Handle to the dataset

DESCRIPTION

This function creates a dataset.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid ALF handle • `ALF_ERR_GENERIC`: Generic internal errors

alf_dataset_buffer_add function

NAME

alf_dataset_buffer_add - Adds a data buffer to the data set.

SYNOPSIS

int alf_dataset_buffer_add(alf_dataset_handle_t dataset, void *buffer, unsigned long long size, ALF_DATASET_ACCESS_MODE_T access_mode);

Parameters

`buffer` Address of the buffer to be added

`size` Size of the buffer

`access mode` Access mode for the buffer. A buffer can have either of the following access modes:

- `ALF_DATASET_READ_ONLY`: The data set buffer is read-only. Work blocks referencing the data in this buffer cannot update this buffer as an output buffer.
- `ALF_DATASET_WRITE_ONLY`: The data set buffer is write-only. Work blocks referencing the data in this buffer as input data result in indeterminate behavior. If the application does not write to this buffer during a task's execution, the content of the buffer is indeterminate.
- `ALF_DATASET_READ_WRITE`: The data set buffer allows both read and write access. Work blocks can use this buffer as input buffers and output buffers and/or in out buffers. If the application does not update the this buffer content through a task, its content is indeterminate.

DESCRIPTION

This function adds a data buffer to the data set.

RETURN VALUE

0 Success

less than 0 Errors occurred:

- `ALF_ERR_INVAL`: Invalid input argument
- `ALF_ERR_BADF`: Invalid dataset handle
- `ALF_ERR_PERM`: The API call is not permitted with the current calling context. The dataset has been associated with a task and thus closed from further buffer additions.
- `ALF_ERR_GENERIC`: Generic internal errors

alf_dataset_destroy function

NAME

alf_dataset_destroy - Destroys a given data set.

SYNOPSIS

int alf_dataset_destroy(alf_dataset_handle_t dataset_handle);

Parameters

`dataset_handle`	Handle to the dataset

DESCRIPTION

This function destroys a given dataset. Further references to the dataset result in indeterminate behaviors. Further references to the data within a dataset are still valid. You cannot destroy a dataset if there are still running tasks associated with a dataset.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument. • `ALF_ERR_PERM`: The API call is not permitted with the current calling context. A task is already using the dataset. • `ALF_ERR_GENERIC`: Generic internal errors.

alf_task_dataset_associate function

NAME

alf_task_dataset_associate - Associates a given task with a dataset.

SYNOPSIS

int alf_task_dataset_associate(alf_task_handle_t task, alf_dataset_handle_t dataset);

Parameters

`dataset_handle`	Handle to dataset
`task_handle`	Handle to the task

DESCRIPTION

This function associates a given task with a dataset. This function can only be called before any work block is enqueued for the task. After a task is associated with a dataset, all subsequent work blocks created and enqueued for this task cannot reference data outside the dataset.

After a task is associated with a dataset, further calls to `alf_data_buffer_add` results in error.

After a task is associated with a dataset, the host application program can only use the data after `alf_task_wait` is called and returned.

RETURN VALUE

0	Success
less than 0	Errors occurred: • `ALF_ERR_INVAL`: Invalid input argument • `ALF_ERR_BADF`: Invalid task handle • `ALF_ERR_PERM`: The API call is not permitted with the current calling context. T • `ALF_ERR_SRCH`: Invalid dataset handle • `ALF_ERR_GENERIC`: Generic internal errors

Chapter 21. Accelerator API

This topic describes the definitions for the accelerator APIs and data types.

Computational kernel function exporting macros

The ALF MPMD programming model supports multiple computational kernels in a single accelerator execution image. To allow the ALF runtime to differentiate between different functions for different kernels, you need to export these functions to the ALF runtime. Some macros are provided to make sure the function exporting can be performed in a platform-neutral way. In each accelerator side execution image, there must be at least one computational kernel API exporting definition section. However the maximum allowed number of sections is platform-dependent.

The following example shows how these macros are used.

```
/* API implementations for task "foo" */
int foo_comp_kernel(...) {...}
int foo_input_prepare(...) {...}
int foo_output_prepare(...) {...}
int foo_ctx_setup(...) {...}
int foo_ctx_merge(...) {...}

/* API implementations for task "bar" */
int bar_comp_kernel(...) {...}
int bar_input_prepare(...) {...}
int bar_output_prepare(...) {...}
int bar_ctx_setup(...) {...}
int bar_ctx_merge(...) {...}

/* API exporting definition section */
ALF_ACCEL_API_LIST_BEGIN

/* for task "foo" */
ALF_ACCEL_EXPORT_API ("foo_comp_kernel", foo_comp_kernel);
ALF_ACCEL_EXPORT_API ("foo_input_prepare", foo_input_prepare);
ALF_ACCEL_EXPORT_API ("foo_output_prepare", foo_output_prepare);
ALF_ACCEL_EXPORT_API ("foo_ctx_setup", foo_ctx_setup);
ALF_ACCEL_EXPORT_API ("foo_ctx_merge", foo_ctx_merge);

/* for tas "bar" */
ALF_ACCEL_EXPORT_API ("bar_comp_kernel", bar_comp_kernel);
ALF_ACCEL_EXPORT_API ("bar_input_prepare", bar_input_prepare);
ALF_ACCEL_EXPORT_API ("bar_output_prepare", bar_output_prepare);
ALF_ACCEL_EXPORT_API ("bar_ctx_setup", bar_ctx_setup);
ALF_ACCEL_EXPORT_API ("bar_ctx_merge", bar_ctx_merge);

ALF_ACCEL_EXPORT_API_LIST_END
```

ALF_ACCEL_EXPORT_API function

NAME

ALF_ACCEL_EXPORT_API - Declares one entry of the computing kernel API exporting definition section.

SYNOPSIS

ALF_ACCEL_EXPORT_API(const char *p_api_name, int (*p_api)())

Parameters

`p_api_name[IN]`	The string constant that uniquely identifies the exported API. It is recommended to be just the same as the correspondent function identifier.
`p_api[IN]`	The exported function entry pointer.

DESCRIPTION

This macro declares one entry of the computing kernel API exporting definition section. The ALF runtime locates the entry address of the user-implemented computing kernel functions based on information provided by the corresponding entries.

ALF_ACCEL_EXPORT_API_LIST_BEGIN function

NAME

ALF_ACCEL_EXPORT_API_LIST_BEGIN - This macro declares the beginning of computational kernel API exporting definition section.

DESCRIPTION

This macro must be the first statement of the definition section.

ALF_ACCEL_EXPORT_API_LIST_END function

NAME

ALF_ACCEL_EXPORT_API_LIST_END - This macro declares the ending of computational kernel API exporting definition section.

DESCRIPTION

This macro must be the last statement of the definition section.

User-provided computational kernel APIs

This section lists the prototypes of accelerator APIs that you need to implement. Some of these functions are optional functions, which you do not need to implement if not required.

Note: For documentation purposes, names are provided for these different prototype APIs. However, you can choose your own function names for your implementations of these functions.

Computing kernel prototypes for work block tasks

This section describes the following APIs:

alf_accel_comp_kernel function

NAME

alf_accel_comp_kernel - Computes the work blocks.

SYNOPSIS

int alf_accel_comp_kernel(void* p_task_ctx, void *p_parm_ctx_buffer, void *p_input_buffer, void *p_output_buffer, void* p_inout_buffer, unsigned int current_iter, unsigned int num_iter);

Parameters

`p_task_context [IN]`	A pointer to the local memory block where the task context buffer is kept.
`p_parm_ctx_data [IN]`	A pointer to the local memory block where the parameter and context data are kept.
`p_input_buffer [IN]`	A pointer to the local memory block where the input data is loaded.
`p_output_buffer [IN]`	A pointer to the local memory block where the output data is written.
`p_inout_buffer [IN]`	A pointer to the accelerator memory block where the in/out buffers are located.
`current_iter [IN]`	The current iteration count of multi-use work blocks. This value starts at 0. For single-use work blocks, this value is always 0.
`num_iter [IN]`	The total number of iterations of multi-use work blocks. For single-use work blocks, this value is always 1.

DESCRIPTION

This is the computational kernel that does the computation of the work blocks. The ALF runtime ensures that all input data are available before invoking this call. You must provide an implementation for this function.

RETURN VALUE

0	The computation finished correctly.
less than `0`	An error occurred during the computation. The error code is passed back to you to be handled.

alf_accel_input_dtl_prepare function

NAME

alf_accel_input_dtl_prepare - Defines the data transfer lists for input data.

SYNOPSIS

int alf_accel_input_dtl_prepare (void* p_task_context, void *p_parm_context, void *p_dtl, unsigned int current_iter, unsigned int num_iter);

Parameters

`p_task_context[IN]`	Pointer to the task context buffer in accelerator memory.
`p_parm_ctx_buffer[IN]`	Pointer to the work block parameter context buffer in accelerator memory.
`p_dtl[IN]`	Pointer the data transfer list the generated data transfer list should be saved.
`current_iter[IN]`	The current iteration count of multi-use work blocks. This value starts at 0. For single-use work blocks, this value is always 0.
`num_iter[IN]`	The total number of iterations of multi-use work blocks. For single-use work blocks, this value is always 1.

DESCRIPTION

This function is called by the ALF runtime when it needs the accelerator to define the data transfer lists for input data. One important point to consider is that because the ALF framework may do double buffering, the function only refers to the information provided by the `p_parm_ctx_buffer`. This function should generate the data transfer lists for the input buffer (`ALF_BUF_IN`), the overlapped input buffer (`ALF_BUF_OVL_IN`), and the overlapped I/O buffer (`ALF_BUF_OVL_INOUT`) when these buffers are enabled. For the overlapped I/O buffer (`ALF_BUF_OVL_INOUT`), the data transfer list generated in this function is reused by the runtime to push the data back to host memory.

This function is an optional function. It is only called if the task descriptor sets the `ALF_TASK_DESC_PARTITION_ON_ACCEL` to true. When this attribute is not set or set to false, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that returns zero when weak link is not supported.

RETURN VALUE

0	The computation finished correctly.
less than 0	An error occurred during the call. The error code is passed back to you to be handled.

alf_accel_output_dtl_prepare function

NAME

alf_accel_output_dtl_prepare - Defines the partition of output data.

SYNOPSIS

int alf_accel_output_dtl_prepare (void* p_task_context, void *p_parm_ctx_buffer, void *p_io_container, unsigned int current_iter, unsigned int num_iter);

Parameters

`p_task_context[IN]`	Pointer to the task context buffer in accelerator memory.
`p_parm_ctx_buffer[IN]`	Pointer to the work block parameter context in accelerator memory.
`p_dt_list_buffer[IN]`	Pointer to the buffer where the generated data transfer list should be saved.
`current_iter[IN]`	The current iteration count of multi-use work blocks. This value starts at 0. For single-use work blocks, this value is always 0.
`num_iter[IN]`	The total number of iterations of multi-use work blocks. For single-use work blocks, this value is always 1.

DESCRIPTION

This function is called by the ALF runtime when it needs the accelerator to define the partition of output data. Because the ALF may be doing double buffering, the function should only refer to the information provided by the `p_parm_ctx_buffer`. This function generates the data transfer lists for the output buffer (`ALF_BUF_OUT`) and the overlapped output buffer (`ALF_BUF_OVL_OUT`) when these buffers are enabled.

This function is only called if the task descriptor sets the `ALF_TASK_DESC_PARTITION_ON_ACCEL` to true. When this attribute is not set or set false, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that return zero when weak link is not supported.

RETURN VALUE

0	The computation finished correctly.
less than 0	An error occurred during the call. The error code is passed back to you to be handled.

alf_accel_task_context_setup function

NAME

alf_accel_task_context_setup - Initializes a task.

SYNOPSIS

int alf_accel_task_context_setup (void* p_task_context);

Parameters

`p_task_context [IN/OUT]` Pointer to task context in accelerator memory.

DESCRIPTION

This function is called by the ALF runtime when a task starts running on an accelerator. The runtime loads the initial task context to the local memory and calls this function to do some task instance specific initialization.

The ALF runtime only invokes this API when the task has a task context. When the task does not have a task context or the application does not need extra setup of the initial context, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that returns zero when weak link is not supported.

RETURN VALUE

0	The API call finished correctly.
less than 0	An error happened during the call. The error code is passed back to you to be handled.

alf_accel_task_context_merge function

NAME

alf_accel_task_context_merge - Merges the context after a task has stopped running.

SYNOPSIS

int alf_accel_task_context_merge (void* p_task_context_to_be_merged, void* p_task_context);

Parameters

`p_task_context_to_merge[IN]`	Pointer to the local memory block where the to be merged task context buffer is kept.
`p_task_context[IN/OUT]`	Pointer to the local memory block where the to be target task context buffer is kept.

DESCRIPTION

This function is called by the ALF runtime when a task stops running on an accelerator. The runtime loads the corresponding task context to the memory of an accelerator that is running this task and calls this function to do the context merge.

The ALF runtime only invokes this API only when the task has a task context. If the task does not have a task context or the application does not need to do context merge, you can choose not to implement this API when the programming environment supports weak link or to implement an empty function that returns zero when weak link is not supported.

RETURN VALUE

0	The API call finishes correctly.
less than 0	An error occurred during the call. The error code is passed back to you to be handled.

Computational kernel prototypes for lightweight tasks

This section describes the following APIs:

alf_accel_lts_main function

NAME

alf_accel_lts_main - This is the prototype of the main entry point of a lightweight task.

SYNOPSIS

int alf_accel_lts_main (void* p_task_ctx, int instance_id, int number_of_instance);

Parameters

`p_task_ctx [IN]`	Pointer to task context data in local storage. ALF runtime loads the data.
`instance_id [IN]`	The current task instance id, starting from 0 to num_instances-1
`number_of_instance [IN]`	The number of instances of this task.

DESCRIPTION

This is the prototype of the main entry point of a lightweight task. You must provide an implementation according to this prototype and declare it by using the `ALF_ACCEL_EXPORT_API` as ALF symbols. The user-implemented function can use any valid identifier, but the definition must follow the argument formats of this prototype. The ALF runtime invokes this entry point on each instance only when all the instances of the same task are ready to run. The specific instance of the task is ended when this function returns. The task ends when all the instances end.

RETURN VALUE

0	The API call finishes correctly.
not 0	An error occurred during the task. The error code is passed back to be handled.

Runtime APIs

This section lists the APIs that accelerator side ALF runtime provides.

alf_accel_num_instances function

NAME

alf_accel_num_instances - Returns the number of instances that are running this computational kernel.

SYNOPSIS

int alf_accel_num_instances (void);

Parameters
None

DESCRIPTION

This function returns the number of instances that are currently executing this computational kernel. This function should only be used when a task is created with the task attribute ALF_TASK_ATTR_SCHED_FIXED. If user calls this function without ALF_TASK_ATTR_SCHED_FIXED, the number returned might change from one invocation to the next as the ALF runtime dynamically loads and unloads task instances.

RETURN VALUE

>0	number of accelerators that are executing this compute task
less than 0	Internal error

alf_accel_instance_id function

NAME

alf_accel_instance_id - Returns the number of instances that are running this computational kernel.

SYNOPSIS

int alf_accel_instance_id (void);

Parameters
None

DESCRIPTION

This function returns the current instance ID of the task. This ID ranges from 0 to `alf_accel_num_instances`.

RETURN VALUE

>=0	Returns the ID of the current accelerator. This is guaranteed to be unique within the reserved accelerators for ALF runtime
less than 0	Internal error

Work block task-specific APIs

This section describes the following APIs:

ALF_ACCEL_DTL_BEGIN function

NAME

ALF_ACCEL_DTL_BEGIN - Marks the beginning of a data transfer list for the specified target `buffer_type`.

SYNOPSIS

ALF_ACCEL_DTL_BEGIN (void* p_dtl, ALF_BUF_TYPE_T buf_type, unsigned int offset);

Parameters

`p_dtl[IN/OUT]`	Pointer to buffer for the data transfer list data structure.

`buf_type`	`ALF_BUF_IN`
	`ALF_BUF_OUT`
	`ALF_OVL_IN`
	`ALF_OVL_OUT`
	`ALF_OVL_INOUT`

`offset[IN]`	Offset to the input or output buffer pointer in local memory to which the data transfer list refers to.

DESCRIPTION

This utility marks the beginning of a data transfer list for the specified target `buffer_type`. Further calls to function `ALF_ACCEL_DTL_ENTRY_ADD` refer to the currently opened data transfer list. You can create multiple data transfer lists per buffer type. However, only one data transfer list is opened for entry at any time.

Note: This API is for accelerator node side to generate the data transfer list entries. It may be implemented as macros on some platforms.

RETURN VALUE

None.

ALF_ACCEL_DTL_ENTRY_ADD function

NAME

ALF_ACCEL_DTL_ENTRY_ADD - Fills the data transfer list entry.

SYNOPSIS

ALF_ACCEL_DTL_ENTRY_ADD (void *p_dtl, unsigned int data_size, ALF_DATA_TYPE_T data_type, alf_data_addr64_t p_host_address);

Parameters

`p_dtl[IN]`	Pointer to buffer for the data transfer list data structure.
`data_size[IN]`	Size of the data in unit of the data type.
`data_type[IN]`	The type of data. This value is required if data endianess conversion is necessary when moving the data.
`host_address[IN]`	Address of the host memory.

DESCRIPTION

This function fills the data transfer list entry.

This API is for the accelerator node side to generate the data transfer list entries. It can be implemented as macros on some platforms.

Note: This API is for accelerator node side to generate the data transfer list entries. It can be implemented as macros on some platforms.

Cell/B.E. implementation details

The ALF runtime does not deal with the 16 KB Cell/B.E. DMA list entry size limitations transparently. Cell/B.E. programmers must deal with that limitation by dividing the entry into multiple entries not larger than 16 KB. Programmers also must be aware of the Cell/B.E. limitation on data transfer list size. A data transfer list on Cell/B.E. must not exceed 2048 entries. The ALF runtime does not do a strict check on these constraints by default due to performance requirements. The checks are only enabled for the debug build of ALF. Refer to Chapter 14, “Building and linking an application or an accelerated library,” on page 49 for more details about how to enable a debug build.

RETURN VALUE

None.

ALF_ACCEL_DTL_END function

NAME

ALF_ACCEL_DTL_END - Marks the ending of a data transfer list.

SYNOPSIS

ALF_ACCEL_DTL_END(void* p_dtl);

Parameters

p_dtl[IN]	Pointer to buffer for the data transfer list data structure.

DESCRIPTION

This utility marks the ending of a data transfer list.

RETURN VALUE

None.

Lightweight task-specific APIs

This section describes the following APIs:

alf_accel_instance_exit_if_canceled function

NAME

alf_accel_instance_exit_if_canceled - Tests for an optional cancellation point of a lightweight task instance.

SYNOPSIS

void alf_accel_instance_exit_if_canceled(void);

Parameters
N/A

DESCRIPTION

This function tests for an optional cancellation point of a lightweight task instance. It is provided as an alternative solution to specific implementations that do not support asynchronous cancellation of accelerator side threads. It is called at the point of the code when the task instance is allowed to have the cancellation processed, for example, when no locks are held. If task cancellation has already been requested then this function does not return to the caller.

alf_accel_host_addr_translate function

NAME

alf_accel_host_addr_translate - Translates the host memory address inside a dataset to appropriate address space that is necessary for DMA access on the accelerator side.

SYNOPSIS

int alf_accel_host_addr_translate(alf_data_addr64_t host_addr, unsigned long long host_size, ALF_DATASET_ACCESS_MODE_T host_access_mode, alf_data_addr64_t *trans_host_addr);

Parameters

`host_addr [IN]`
The host address that is to be translated. The address must be within the valid scope of a host buffer that is defined in the associated dataset.

`host_size [IN]`
The size of the memory region to be accessed in byte, starting form the given ″host_addr″. The ″host_addr + host_size″ must be within the valid scope of a host buffer that is defined in the associated dataset.

`host_access_mode [IN]`
- `ALF_DATASET_READ_ONLY`: The data set buffer is read-only.
- `ALF_DATASET_WRITE_ONLY`: The data set buffer is write-only.
- `ALF_DATASET_READ_WRITE`: The data set buffer allows both read and write access.

`trans_host_addr [IN/OUT]`
The pointer to the buffer to receive the result. This pointer can be NULL if the caller does not want the result.

DESCRIPTION

This API is only supported for the lightweight task. This is an API to translate the host memory address inside a dataset to appropriate address space that is necessary for DMA access on the accelerator side. The host address must be effectively defined in a dataset that is associated with the current task.

RETURN VALUE

0
Success, the EA of the problem space

Less than 0
Error:
- `ALF_ERR_2BIG`: The specified area size is out of the valid scope
- `ALF_ERR_FAULT`: The specified address is not a valid address
- `ALF_ERR_PERM`: The call is not allowed in the current task type
- `ALF_ERR_GENERIC`: Generic ALF internal error

Chapter 22. Cell/B.E. platform-specific extension APIs

These APIs are not part of the core ALF API. They are specific to Cell/B.E. architecture.

Work block task platform-specific APIs

The major purpose of these APIs is for performance improvements when the data movement patterns are very complex. Because the abstracted DTL APIs consider error conditions and also other boundary situations, the overhead is significant when the data movement patterns are very complex. The platform specific APIs can be used to get direct access into the internal of DTL data structure and thus you can generate the MFC DMA list directly.

The following example demonstrates how these APIs are used.

Note: The calling order of the APIs must follow the following order:

```
ALF_ACCEL_DTL_BEGIN
ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET
ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE
ALF_ACCEL_DTL_END
```

.

```
int  alf_accel_input_dtl_prepare (void* p_task_context, void *p_parm_context,
void *p_dtl, unsigned int current_iter, unsigned int num_iter)
{
   mfc_element_t *p_dma_list;
   unsigned int max_entry, cnt;
   alf_data_addr64_t ea_base = ((my_comon_data_t *)p_task_context)->ea_buf1;
   // to simplify the case, we assume 'ea_base' is within one 4GB segment

   ALF_ACCEL_DTL_BEGIN(p_dtl, ALF_IO_BUF_IN, 0);
   ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET(p_dtl, &p_dma_list, &max_entry);
   for(cnt=0; cnt<100 && cnt <max_entry; cnt++)
   {
       p_dma_list[cnt].eal = (unsigned int)(ea_base+offset);
       ...
   }
   ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE(p_dtl, cnt, ea_base);
   ALF_ACCEL_DTL_END(p_dtl);
}
```

ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET function

NAME

ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET - Gets the internal DMA list buffers.

SYNOPSIS

void ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET (void *p_dtl, void **pp_dma_list_buffer, unsigned int *p_max_entries);

Parameters

`p_dtl [IN]`	A pointer to the buffer for the data transfer list data structure.
`pp_dma_list_buffer [OUT]`	Returns a pointer to the internal DMA list buffer.
`p_max_entries [OUT]`	Returns the maximum allowed entries in this buffer

DESCRIPTION

This utility gets the internal DMA list buffers so that you can directly access them. It must be called after ALF_ACCEL_DTL_BEGIN and before ALF_ACCEL_DTL_END After this call, ALF_ACCEL_DTL_ENTRY_ADD must not be used before ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_UPDATE is called.

For the Hybrid implementation, the ALF runtime makes no attempt to do endian conversion for the buffers with DMA lists created manually through this call.

RETURN VALUE

0	The computation finished correctly.
less than `0`	An error occurred during the computation. The error code is passed back to the library developer to be handled.

ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_UPDATE function (Deprecated)

NAME

ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_UPDATE - Updates the internal data structure when the direct access completes.

SYNOPSIS

void ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_UPDATE (void *p_dtl, unsigned int num_entries);

Parameters

`p_dt_list_buffer [IN]`	A pointer to the buffer for the data transfer list data structure.
`num_entries [IN]`	The number of DMA list entries filled in during the direct access.

DESCRIPTION

This utility updates the internal data structure when the direct access completes. It must be called after `ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET` and before `ALF_ACCEL_DTL_END` Any further calls to `ALF_ACCEL_DTL_ENTRY_ADD` can only be done after this call.

Note: This API is deprecated due to a design limitation that limits this API to only support 32 bits address mode. It is recommended to use the updated API as `ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE`. The new API supports both 32 bits and 64 bits modes.

RETURN VALUE

Not specified

ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE function

NAME

ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE - Updates the internal data structure when the direct access completes.

SYNOPSIS

void ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_COMPLETE (void *p_dtl, unsigned int num_entries, alf_data_addr64_t base_ea);

Parameters

`p_dt_list_buffer [IN]`	A pointer to the buffer for the data transfer list data structure.
`num_entries [IN]`	The number of DMA list entries filled in during the direct access.
`base_ea [IN]`	The full 64 bits EA of the buffer to be accessed. Only the high 32 bits is used. Refer to the description for more details.

DESCRIPTION

This utility updates the internal data structure when the direct access completes. It must be called after `ALF_ACCEL_DTL_CBEA_DMA_LIST_BUFFER_GET` and before `ALF_ACCEL_DTL_END`. Any further calls to `ALF_ACCEL_DTL_ENTRY_ADD` can only be done after this call. Due to hardware limitations, one single DMA list can only access buffers that contain EA segments within the same 4 GB. This means that whatever buffers referenced in this DMA list, the high 32 bits EA address must be the same. If the addresses can be across different 4 GB EA segments, the DMA list needs to be broken into multiple ones, where each one covers one 4 GB segment.

RETURN VALUE

Not specified

Lighweight task platform-specific APIs

This section describes the following APIs:

alf_accel_instance_cbea_local_store_ea_get function

NAME

alf_accel_instance_cbea_local_store_ea_get - Retrieves the EA of the local store on the SPU that runs the specified task instance.

SYNOPSIS

alf_data_addr64_t alf_accel_instance_ cbea_ local_store_ea_get (int instance_id);

Parameters

`Instance_id [IN]`	The instance id to be queried

DESCRIPTION

This API retrieves the EA of the local store on the SPU that runs the specified task instance. This API is a Cell/B.E.-specific API.

RETURN VALUE

> 0	Success. The EA of the local store.
0	Error.

alf_accel_instance_cbea_ps_get_sig_notify1 function

NAME

alf_accel_instance_cbea_ps_get_sig_notify1 - Retrieves the EA of the signal notification area 1 of the SPU that runs the specified task instance.

SYNOPSIS

alf_data_addr64_t alf_accel_instance_cbea_ps_get_sig_notify1 (int instance_id);

Parameters

`Instance_id [IN]`	The instance ID to be queried

DESCRIPTION

This is an API to retrieve the EA of the signal notification area 1 of the SPU that runs the specified task instance.

RETURN VALUE

> 0	Success. The EA of the problem space.
0	Error.

alf_accel_instance_cbea_ps_get_sig_notify2 function

NAME

alf_accel_instance_cbea_ps_get_sig_notify2 - Retrieves the EA of the signal notification area 2 of the SPU that runs the specified task instance.

SYNOPSIS

alf_data_addr64_t alf_accel_instance_cbea_ps_get_sig_notify2 (int instance_id);

Parameters

`Instance_id [IN]`	The instance ID to be queried

DESCRIPTION

This is an API to retrieve the EA of the signal notification area 2 of the SPU that runs the specified task instance.

RETURN VALUE

> 0	Success. The EA of the problem space.
0	Error.

Part 6. Appendixes

Appendix A. Changes to the APIs for this release

The following table describes which APIs:

- Have been updated for this release
- Are new for this release
- Have been replaced for this release
- Were available in the previous release but have been removed for this release

Table 4. API changes

API Name	API has been updated for this release Y/N	New API for this release	Changes from previous release 3.0
Framework APIs			
`alf_handle_t`	N		
`alf_error_handler_t`	N		
`alf_init`	Y		
`alf_init_shared`		X	New for SDK 3.1
`alf_query_system_info`	N		
`alf_num_instances_set`	Y		
`alf_num_instances_query`		X	New for SDK 3.1
`alf_exit`	Y		
`alf_error_handler_register`	Y		
Compute APIs			
`alf_task_handle_t`	N		
`alf_task_desc_handle_t`	N		
`alf_task_desc_create`	N		
`alf_task_desc_destroy`	N		
`alf_task_desc_ctx_entry_add`	N		
`alf_task_desc_set_int32`	Y		
`alf_task_desc_set_int64`	N		
`alf_task_create`	Y		
`alf_task_finalize`	N		
`alf_task_wait`	Y		
`alf_task_query`	N		
`alf_task_destroy`	N		
`alf_task_depends_on`	N		
`alf_task_event_handler_register`	N		
Work block APIs			
`alf_wb_handle_t`	N		
`alf_wb_create`	Y		
`alf_wb_enqueue`	N		
`alf_wb_dtl_begin`	N		

Table 4. API changes (continued)

API Name	API has been updated for this release Y/N	New API for this release	Changes from previous release 3.0
alf_wb_parm_add	N		
alf_wb_dtl_entry_add	N		
alf_wb_dtl_end	N		
alf_wb_sync		X	Resupported for SDK 3.1
alf_wb_sync_wait		X	Resupported for SDK 3.1
sync_callback_func		X	Resupported for SDK 3.1
wb_sync_handle_t		X	Resupported for SDK 3.1
Data set APIs			
alf_dataset_handle_t	N		
alf_dataset_create	N		
alf_dataset_buffer_add	N		
alf_dataset_destroy	Y		
alf_task_dataset_associate	N		
Accelerator APIs			
ALF_ACCEL_EXPORT_API_ LIST_BEGIN	N		
ALF_ACCEL_EXPORT_API	N		
ALF_ACCEL_EXPORT_API_ LIST_END	N		
Computational kernel APIs			
alf_accel_comp_kernel	N		
alf_accel_input_dtl_prepare	N		
alf_accel_output_dtl_prepare	N		
alf_accel_task_context_setup	N		
alf_accel_task_context_merge	N		
alf_accel_lts_main		X	
Runtime APIs			
alf_accel_num_instances	N		
alf_accel_instance_id	N		
alf_accel_instance_exit if canceled		X	
alf_accel_host_addr_translate	Y Platform-specific for ALF for Hybrid		
ALF_ACCEL_DTL_BEGIN	Y		
ALF_ACCEL_DTL_ENTRY_ADD	N		
ALF_ACCEL_DTL_END	N		
Cell/B.E. platform specific APIs			
ALF_ACCEL_DTL_CBEA_ DMA_LIST_BUFFER_COMPLETE		X	New for SDK 3.1

Table 4. API changes (continued)

API Name	API has been updated for this release Y/N	New API for this release	Changes from previous release 3.0
ALF_ACCEL_DTL_CBEA_ DMA_LIST_BUFFER_UPDATE	Y		Deprecated from SDK 3.1
ALF_ACCEL_DTL_CBEA_ DMA_LIST_BUFFER_GET	Y		
alf_accel_instance_ cbea_local_store_ea		X	
alf_accel_instance_ ps_get_sig_notify1		X	
alf_accel_instance_ ps_get_sig_notify2		X	

Appendix B. API type and task type compatibility

Because of the differences between the work block tasks and the lightweight tasks, some ALF APIs are not applicable to a specific task type. The following table lists the API compatibility information. It is defined that APIs return `ALF_ERR_PERM` when invoked with an incompatible task or task descriptor handle. For APIs that are "Ignored", there is no error.

Table 5. API type and task type compatibility

API class and name	API attribute	ALF task type	
		Work block	**Lightweight**
Framework			
	*	Y	Y
Compute APIs			
`alf_task_desc_create`		Y	Y
`alf_task_desc_destroy`		Y	Y
`alf_task_desc_set_int32` family	ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE	Y	Ignore
	ALF_TASK_DESC_WB_IN_BUF_SIZE	Y	Ignore
	ALF_TASK_DESC_WB_OUT_BUF_SIZE	Y	Ignore
	ALF_TASK_DESC_WB_INOUT_BUF_SIZE	Y	Ignore
	ALF_TASK_DESC_NUM_DTL_ENTRIES	Y	Ignore
	ALF_TASK_DESC_PARTITION_ON_ACCEL	Y	Ignore
	ALF_TASK_DESC_TSK_CTX_SIZE	Y	Y
	ALF_TASK_DESC_MAX_STACK_SIZE	Y	Ignore
	ALF_TASK_DESC_TASK_TYPE	Y	Y
`alf_task_desc_set_int64` family	ALF_TASK_DESC_ACCEL_LIBRARY_REF_L	Y	Y
	ALF_TASK_DESC_ACCEL_IMAGE_REF_L	Y	Y
	ALF_TASK_DESC_ACCEL_KERNEL_REF_L	Y	Ignore
	ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L	Y	Ignore
	ALF_TASK_DESC_ACCEL_OUTPUT_DTL_REF_L	Y	Ignore
	ALF_TASK_DESC_ACCEL_CTX_SETUP_REF_L	Y	Ignore
	ALF_TASK_DESC_ACCEL_CTX_MERGE_REF_L	Y	Ignore
	ALF_TASK_DESC_ACCEL_LTS_MAIN_REF_L (new)	Ignore	Y
`alf_task_desc_ctx_entry_add`		Y	Y
Task			
`alf_task_create` (definition updated)		Y	Y
`alf_task_finalize` (definition updated)		Y	Y
`alf_task_wait`		Y	Y
`alf_task_destroy`		Y	Y
`alf_task_query`		Y	Y

Table 5. API type and task type compatibility (continued)

API class and name	API attribute	ALF task type	
		Work block	Lightweight
alf_task_depends_on		Y	Y
alf_task_event_handler_register		Y	Y
Work block APIs			
	*	Y	N
Data set APIs			
	*	Y	N
alf_accel_host_addr_translate		N	Y
Accelerator			
alf_accel_num_instances		Y	Y
alf_accel_instance_id		Y	Y
alf_accel_comp_kernel (function prototype)		Y	N
alf_accel_input_dtl_prepare (function prototype)		Y	N
alf_accel_output_dtl_prepare (function prototype)		Y	N
alf_accel_task_context_setup (function type)		Y	N
alf_accel_task_context_merge (function type)		Y	N
alf_accel_lts_main (new)		N	Y
alf_accel_instance_exit if canceled (new)		N	Y
ALF_ACCEL_DTL_BEGIN		Y	N
ALF_ACCEL_DTL_ENTRY_ADD		Y	N
ALF_ACCEL_DTL_END		Y	N
alf_accel_instance_ cbea_local_store_ea (new)		N	Y
alf_accel_instance_ ps_get_sig_notify1 (new)		N	Y
alf_accel_instance_ ps_get_sig_notify2 (new)		N	Y

Appendix C. Examples

The following examples are described in this section:

Basic examples

This section describes the following basic examples:

- "Matrix add - host data partitioning example." This example includes the source code.
- "Matrix add - accelerator data partitioning example" on page 170.

Matrix add - host data partitioning example

In this example, two large matrices are added together using ALF. The problem can be expressed simply as:

```
A[m,n] + B[m,n] = C[m,n]
```

where `m` and `n` are the dimensions of the matrices.

This simple example demonstrates how to:

- Start the ALF runtime environment
- Use task descriptor
- Start a task on the accelerators
- Create and add a work block to a task
- Exit the ALF runtime environment correctly

You can also use this sample as a template to build a more complicated application.

In this example, the host application:

- Initializes the ALF runtime environment
- Creates a task descriptor
- Creates a task based on that task descriptor
- Creates work blocks with the appropriate data transfer lists which start invocations of the computational kernel on the accelerator
- Waits for the computational kernel to finish and exits

The accelerator application includes a simple computational kernel that computes the addition of the two matrices.

The scalar code to add two matrices for a uni-processor machine is provided below:

```
float mat_a[NUM_ROW][NUM_COL];
float mat_b[NUM_ROW][NUM_COL];
float mat_c[NUM_ROW][NUM_COL];
int main(void)
{
  int i,j;
  for (i=0; i<NUM_ROW; i++)
     for (j=0; j<NUM_COL; j++)
        mat_c[i][j] = mat_a[i][j] + mat_b[i][j];
  return 0;
}
```

An ALF host program can be logically divided into several sections:

- Initialization
- Task setup
- Work block set up
- Task wait and exit

Source code

The following code listings only show the relevant sections of the code. For a complete listing, refer to the ALF samples directory

`matrix_add/STEP1a_partition_scheme_A/common/host_partition`

Initialization

The following code segment shows how ALF is initialized and accelerators allocated for a specific ALF runtime.

```
alf_handle_t alf_handle;
unsigned int nodes;

/* initializes the runtime environment for ALF*/
alf_init(&config_parms, &alf_handle;);

/* get the number of SPE accelerators available for from the Opteron */
rc = alf_query_system_info(alf_handle, ALF_QUERY_NUM_ACCEL, ALF_ACCEL_TYPE_SPE, &nodes;);

/* set the total number of accelerator instances (in this case, SPE) */
/* the ALF runtime will have during its lifetime */
rc = alf_num_instances_set (alf_handle, nodes);
```

Task setup

The next section of an ALF host program contains information about the description of a task and the creation of the task runtime. The `alf_task_desc_create` function creates a task descriptor. This descriptor can be used multiple times to create different executable tasks. The function `alf_task_create` creates a task to run an SPE program with the name *spe_add_program*.

```
/* variable declarations */
alf_task_desc_handle_t task_desc_handle;
alf_task_handle_t task_handle;
const char* spe_image_name;
const char* library_path_name;
const char* comp_kernel_name;

/* describing a task that's executable on the SPE*/
alf_task_desc_create(alf_handle, ALF_ACCEL_TYPE_SPE, &task_desc_handle;);
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_TSK_CTX_SIZE, 0);
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE, sizeof(add_parms_t));
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_IN_BUF_SIZE, H * V * 2 sizeof(float));
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_WB_OUT_BUF_SIZE, H * V * sizeof(float));
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_NUM_DTL_ENTRIES, 8);
alf_task_desc_set_int32(task_desc_handle, ALF_TASK_DESC_MAX_STACK_SIZE, 4096);
```

```
/* providing the SPE executable name */
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_IMAGE_REF_L,(unsigned long long) spe_image_name);
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_LIBRARY_REF_L,(unsigned long) library_path_name);
alf_task_desc_set_int64(task_desc_handle, ALF_TASK_DESC_ACCEL_KERNEL_REF_L,(unsigned long) comp_kernel_name);
```

Work block setup

This section shows how work blocks are created. After the program has created the work block, it describes the input and output associated with each work block. Each work block contains the input description for blocks in the input matrices of size `H * V` starting at location `matrix[row][0]` with `H` and `V` representing the horizontal and vertical dimensions of the block.

In this example, assume that the accelerator memory can contain the two input buffers of size `H * V` elements and the output buffer of size `H * V`. The program calls `alf_wb_enqueue()` to add the work block to the queue to be processed. ALF employs an immediate runtime mode. As soon as the first work block is added to the queue, the task starts processing the work block. The function `alf_task_finalize` closes the work block queue.

```
alf_wb_handle_t wb_handle;
add_parms_t parm __attribute__((aligned(128)));
parm.h = H; /* horizontal size of the block */
parm.v = V; /* vertical size of the block */

/* creating work blocks and adding param & io buffer */
for (i = 0; i < NUM_ROW; i += H) {
     alf_wb_create(task_handle, ALF_WB_SINGLE, 0,&wb_handle);

     /* begins a new Data Transfer List for INPUT */
     alf_wb_dtl_set_begin(wb_handle, ALF_BUF_IN, 0);

     /* Add H*V element of mat_a as Input */
     alf_wb_dtl_set_entry_add(wb_handle, &matrix_a[i][0], H * V, ALF_DATA_FLOAT);

     /* Add H*V element of mat_b as Input */
     alf_wb_dtl_set_entry_add(wb_handle, &matrix_b[i][0], H * V, ALF_DATA_FLOAT);
     alf_wb_dtl_set_end(wb_handle);

     /* begins a new Data Transfer List OUTPUT */
     alf_wb_dtl_set_begin(wb_handle, ALF_BUF_OUT, 0);

     /* Add H*V element of mat_c as Output */
     alf_wb_dtl_set_entry_add(wb_handle, &matrix_c[i][0], H * V, ALF_DATA_FLOAT);
     alf_wb_dtl_set_end(wb_handle);

     /* pass parameters H and V to spu */
     alf_wb_parm_add(wb_handle, (void *) (&parm), sizeof(parm), ALF_DATA_BYTE, 0);

     /* enqueuing work block */
     alf_wb_enqueue(wb_handle);
}
alf_task_finalize(task_handle);
```

Task wait and exit

After all the work blocks are on the processing queue, the program waits for the accelerator to finish processing the work blocks. Then `alf_exit()` is called to cleanly exit the ALF runtime environment.

```
/* waiting for all work blocks to be done*/
alf_task_wait(task_handle, -1);
/* exit ALF runtime */
alf_exit(alf_handle, ALF_EXIT_WAIT, -1);
```

Accelerator side

On the accelerator side, you need to provide the actual computational kernel that computes the addition of the two blocks of matrices. The ALF runtime on the accelerator is responsible for getting the input buffer to the accelerator memory

before it runs the user-provided `alf_accel_comp_kernel` function. After `alf_accel_comp_kernel` returns, the ALF runtime is responsible for getting the output data back to host memory space. Double buffering or triple buffering is employed as appropriate to ensure that the latency for the input buffer to get into accelerator memory and the output buffer to get to host memory space is well covered with computation.

```
int alf_accel_comp_kernel(void *p_task_context,
            void *p_parm_context,
            void *p_input_buffer,
            void *p_output_buffer,
            void *p_inout_buffer,
            unsigned int current_count,
            unsigned int total_count)
{
 unsigned int i, cnt;
  vector float *sa, *sb, *sc;
  add_parms_t *p_parm = (add_parms_t *)p_parm_context;
  cnt = p_parm->h * p_parm->v / 4;
  sa = (vector float *) p_input_buffer;
  sb = sa + cnt;
  sc = (vector float *) p_output_buffer;
  for (i = 0; i < cnt; i += 4) {
      sc[i] = spu_add(sa[i], sb[i]);
      sc[i + 1] = spu_add(sa[i + 1], sb[i + 1]);
      sc[i + 2] = spu_add(sa[i + 2], sb[i + 2]);
      sc[i + 3] = spu_add(sa[i + 3], sb[i + 3]);
     }
  return 0;
}
```

Matrix add - accelerator data partitioning example

In this example, the same problem as presented in "Matrix add - host data partitioning example" on page 167 is solved, adding two large matrices using ALF. The code remains the same on the host except for the work block creation. The code also needs to specify that it uses accelerator data partitioning in the task descriptor.

An implementation for the `alf_accel_input_dtl_prepare` and `alf_accel_output_dtl_prepare` functions is also required.

For a complete listing of this sample, please refer to the ALF samples directory:
`matrix_add/common/accel_partitioning`

Task context examples

This section describes the following task context examples:

- "Table lookup example"
- "Min-max finder example" on page 172
- "Multiple vector dot products" on page 174
- "Overlapped I/O buffer example" on page 177

Table lookup example

This example shows how the task context buffer is used as a large lookup table to convert the 16 bit input data to 8 bit output data.

The lookup table has 65536 entries defined:

```
For all -32768 <= in <32768
          / 0,        in < -4096
Table(n)=|  in/32     -4096 <=in<4096
          \ 255,      in >= 4096
```

The following is the stripped down code list. The routines of less interest have been removed to allow you to focus on the key features. Because the task context buffer (the lookup table) is already initialized by the host code and the table is used as read-only data, you do not need the context setup and context merge functions on the accelerator side.

Data structures shared by the host and accelerator

The following code segment shows the data structures shared by both the host and the accelerators. The `my_task_context_t` data structure contains the lookup table. The `my_wb_parms_t` data structure represents the parameter and context data for each work block.

```
/* ------------------------------------------------ */
/* data structures shared by host and accelerator */
/* ------------------------------------------------ */
typedef struct _my_task_context_t
{
    alf_data_byte_t table[65536];
} my_task_context_t;

typedef struct _my_wb_parms_t
{
    alf_data_uint32_t num_data;  /* number of data in this WB */
} my_wb_parms_t;
```

Task descriptor setup

The following code segment shows how the task descriptor is set up for this application. The task context-related information marked in **bold** in the code.

```
alf_task_desc_create(alf_handle, 0, &task_desc_handle;);
  /* set up the task descriptor ... ... */
  /* the computing kernel name */
  alf_task_desc_set_int64(task_desc_handle,
     ALF_TASK_DESC_ACCEL_KERNEL_REF_L, "comp_kernel");

  /* the task context buffer size */
  alf_task_desc_set_int32(task_desc_handle,
     ALF_TASK_DESC_TSK_CTX_SIZE, sizeof(my_task_context_t));

  /* the work block parm buffer size */
  alf_task_desc_set_int32(task_desc_handle,
      ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE, sizeof(my_wb_parms_t));

  /* the input buffer size */
  alf_task_desc_set_int32(task_desc_handle,
     ALF_TASK_DESC_WB_IN_BUF_SIZE,
     PART_SIZE*sizeof(alf_data_int16_t));

  /* the output buffer size */
  alf_task_desc_set_int32(task_desc_handle,
     ALF_TASK_DESC_WB_OUT_BUF_SIZE,
     PART_SIZE*sizeof(alf_data_byte_t));

  /* the task context entry */
  alf_task_desc_ctx_entry_add(task_desc_handle, ALF_DATA_BYTE,
    sizeof(my_task_context_t)/sizeof(alf_data_byte_t));
```

Work block setup

The following code segment shows the code for work block creation.

```
  /* creating wb and adding param & io buffer */
  for (i = 0; i < NUM_DATA; i += PART_SIZE)
  {
```

```
        alf_wb_create(task_handle, ALF_WB_SINGLE, 0, &wb_handle);
        alf_wb_dtl_begin(wb_handle, ALF_BUF_IN, 0); /* input */
        alf_wb_dtl_entry_add(wb_handle, pcm16_in+i, PART_SIZE, ALF_DATA_INT16);
        alf_wb_dtl_end(wb_handle);
        alf_wb_dtl_begin(wb_handle, ALF_BUF_OUT, 0); /* output */
        alf_wb_dtl_entry_add(wb_handle, pcm8_out+i,PART_SIZE, ALF_DATA_BYTE);
        alf_wb_dtl_end(wb_handle);
        wb_parm.num_data = PART_SIZE;
        alf_wb_parm_add(wb_handle, (void *)&wb_parm, /* wb parm */
        sizeof(wb_parm)/sizeof(unsigned int), ALF_DATA_INT32, 0);
        alf_wb_enqueue(wb_handle);
  }
```

Accelerator code

The following code is the accelerator side code. The section of the code that modifies the task context is marked in **bold**.

```
/* ------------------------------------------------ */
/* the accelerator side code                        */
/* ------------------------------------------------ */
/* the computation kernel function */
int comp_kernel(void *p_task_context, void *p_parm_ctx_buffer,
                    void *p_input_buffer, void *p_output_buffer,
                    void *p_inout_buffer, unsigned int current_count,
                    unsigned int total_count)
{
    my_task_context_t *p_ctx = (my_task_context_t *) p_task_context;
    my_wb_parms_t *p_parm = (my_wb_parms_t *) p_parm_ctx_buffer;

    alf_data_int16_t *in = (alf_data_int16_t *)p_input_buffer;
    alf_data_byte_t *out = (alf_data_byte_t *)p_output_buffer;
    unsigned int size = p_parm->num_data;
    unsigned int i;

    // it is just a simple table lookup
    for(i=0;i<size;i++)
    {
       out[i] = p_ctx->table[(unsigned short)in[i]];
    }
    return 0;
}
```

Min-max finder example

This example shows how you can use the task context to keep the partial computing results for each task instance and then combine these partial results into the final result.

The example finds the minimum and maximum values in a large data set. The sequential code is a very simple textbook style implementation, it is a linear search across the whole data set, which compares and updates the best known values with each step.

You can use ALF framework to convert the sequential code into a parallel algorithm. The data set must be partitioned into smaller work blocks. These work blocks are then assigned to the different task instances running on the accelerators. Each invocation of a computational kernel on a task instance is to find the maximum or minimum value in the work block assigned to it. After all the work blocks are processed, you have multiple intermediate best values in the context of each task instance. The ALF runtime then calls the context merge function on accelerators to reduce the intermediate results into the final results.

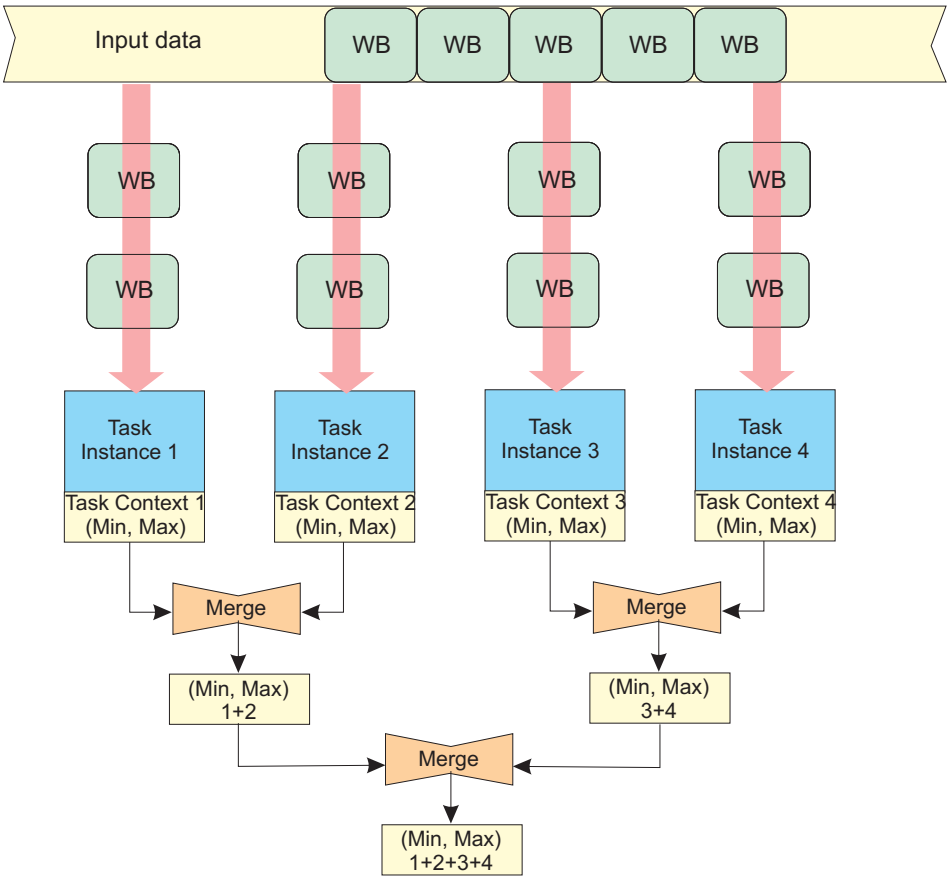


Figure 13. Min-max finder example

Source code

You can find the source code in the sample directory `task_context/min_max`.

Computational kernel

The following code section shows the computational kernel for this application. The computational kernel finds the maximum and minimum values in the provided input buffer then updates the `task_context` with those values.

```
/* ---------------------------------------------- */
/* the accelerator side code                      */
/* ---------------------------------------------- */
/* the computation kernel function */
int comp_kernel(void *p_task_context, void *p_parm_ctx_buffer,
                void *p_input_buffer, void *p_output_buffer,
                void *p_inout_buffer, unsigned int current_count,
                unsigned int total_count)
{
    my_task_context_t *p_ctx = (my_task_context_t *) p_task_context;
    my_wb_parms_t *p_parm = (my_wb_parms_t *) p_parm_ctx_buffer;

    alf_data_int32_t *a = (alf_data_int32_t *)p_input_buffer;
    unsigned int size = p_parm->num_data;
    unsigned int i;

    /* update the best known values in context buffer */
    for(i=0;i<size;i++) {
           if(a[i]>p_ctx->max)
              p_ctx->max = a[i];
```

```
            else if(a[i]<p_ctx->min)
                    p_ctx->min = a[i];
    }     return 0;
}
```

Task context merge

The following code segment shows the `context_merge` function for this application. This function is automatically invoked by the ALF runtime after all the task instances have finished processing all the work blocks. The final minimum and maximum values stored in the task context per task instance are merged through this function.

```
/* the context merge function */
int ctx_merge(void* p_task_context_to_be_merged,
void* p_task_context)
{
    my_task_context_t *p_ctx = (my_task_context_t *) p_task_context;
    my_task_context_t *p_mgr_ctx = (my_task_context_t *)
    p_task_context_to_be_merged;
    if(p_mgr_ctx->max > p_ctx->max)
       p_ctx->max = p_mgr_ctx->max;
    if(p_mgr_ctx->min < p_ctx->min)
       p_ctx->min = p_mgr_ctx->min;
    return 0;
}
```

Multiple vector dot products

This example shows how to use the bundled work block distribution together with the task context to handle situations where the work block can not hold the partitioned data because of a local memory size limit. The example calculates the dot product of two lists of large vectors as:

Given two lists of vectors $\mathbf{A} = \{A_1, A_2, A_3, \ldots, A_m\}$ and $\mathbf{B} = \{B_1, B_2, B_3, \ldots, B_m\}$, where A_i and B_i are dimension N vectors;
Solve $\mathbf{C} = \{c_1, c_2, c_3, \ldots, c_m\}$, where $c_i = A_i \bullet B_i$.
The dot product "$\bullet$" operation of two dimension N vectors A and B is defined as $A \bullet B = \sum_{i=1}^{N} a_i \cdot b_i$ where a_i and b_i are members of vector A and B.

The dot product requires the element multiplication values of the vectors to be accumulated. In the case where a single work block can hold the all the data for vector Ai and Bi, the calculation is straight forward.

However, when the size of the vector is too big to fit into a single work block, the straight forward approach does not work. For example, with the Cell/B.E. processor, there are only 256 KB of local memory on the SPE. It is impossible to store two double precision vectors when the dimension exceeds 16384. In addition, if you consider the extra memory needed by double buffering, code storage, and so on, you are only be able to handle two vectors of 7500 double precision float point elements each (7500*8[size of double]*2[two vectors] * 2[double buffer] ≈ 240 KB of local storage). In this case, large vectors must be partitioned to multiple work blocks and each work block can only return the partial result of a complete dot product.

You can choose to accumulate the partial results of these work blocks on the host to get the final result. But this is not an elegant solution and the performance is also affected. The better solution is to do these accumulations on the accelerators and do them in parallel.

ALF provides the following two implementations for this problem:

- "Implementation 1: Making use of task context and bundled work block distribution"
- "Implementation 2: Making use of multi-use work blocks together with task context or work block parameter/context buffers" on page 176, with the limitation that accelerator side data partitioning is required

Source code

The source code for the two implementations is provided for you to compare with the shipped samples in the following directories:

- `task_context/dot_prod` directory: Implementation 1. task context and bundled work block distribution
- `task_context/dot_prod_multi` directory: Implementation 2. multi-use work blocks together with task context or work block parameter/context buffers

Implementation 1: Making use of task context and bundled work block distribution

For this implementation, all the work blocks of a single vector are put into a bundle. All the work blocks in a single bundle are assigned to one task instance in the order of enqueuing. This means it is possible to use the task context to accumulate the intermediate results and write out the final result when the last work block is processed.

The accumulator in task context is initialized to zero each time a new work block bundle starts.

When the last work block in the bundle is processed, the accumulated value in the task context is copied to the output buffer and then written back to the result area.

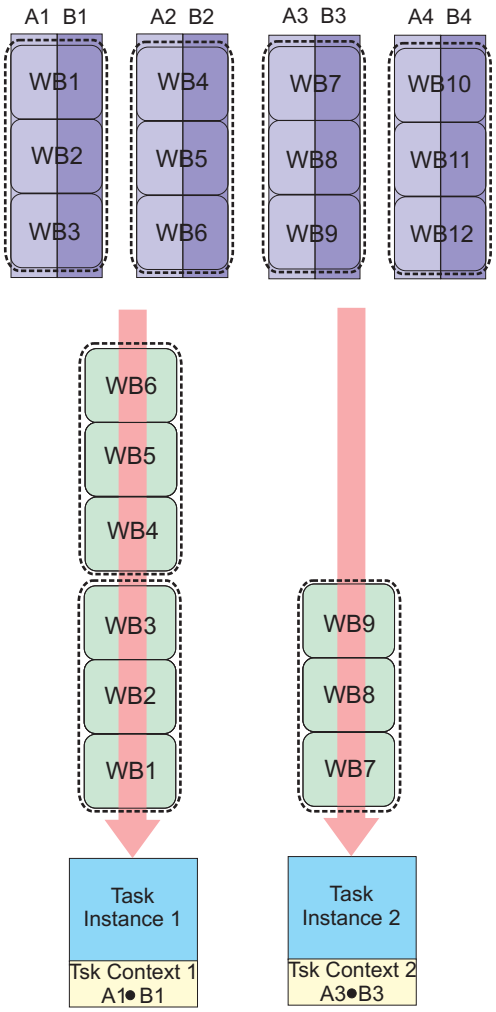


Figure 14. Making use of task context and bundled work block distribution

Implementation 2: Making use of multi-use work blocks together with task context or work block parameter/context buffers

The second implementation is based on multi-use work blocks and work block parameter and context buffers. A multi-use work block is similar to an iteration operation. The accelerator side runtime repeatedly processes the work block until it reaches the provided number of iteration. By using accelerator side data partitioning, it is possible to access different input data during each iteration of the work block. This means the application can be used to handle larger data which a single work block cannot cover due to local storage limitations. Also, the parameter and context buffer of the multi-use work block is kept through the iterations, so you can also choose to keep the accumulator in this buffer, instead of using the task context buffer.

Both methods, using the task context and using multi-use work block are equally valid.

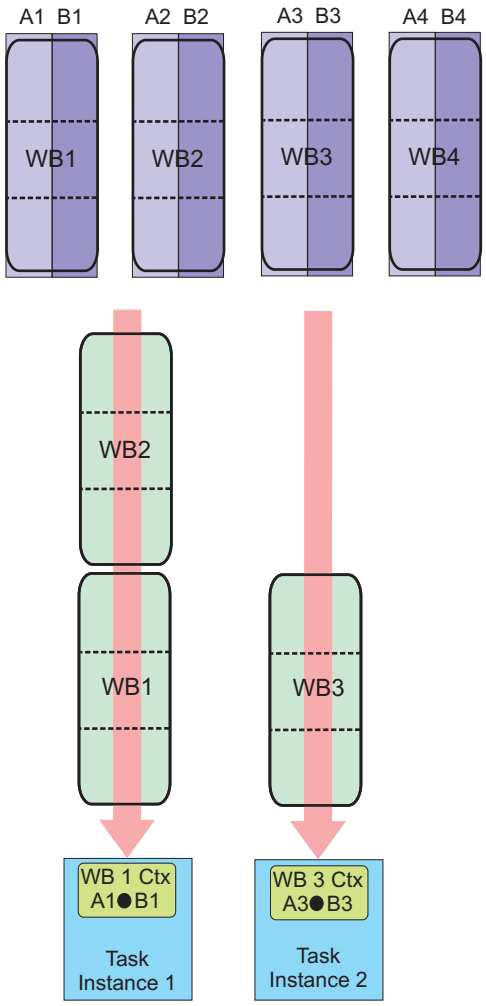


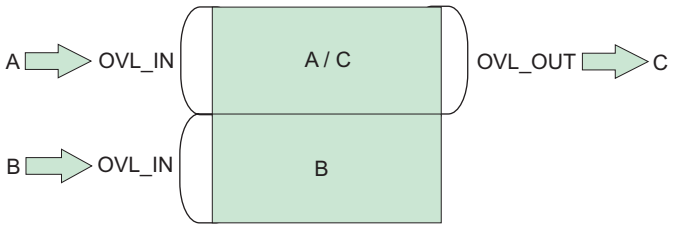
Figure 15. Making use of multi-use work blocks together with task context or work block parameter/context buffers

Overlapped I/O buffer example

The following two simple examples show the usage of overlapped I/O buffers. Both examples do matrix addition.

- The first example implements `C=A+B`, where `A`, `B`, and `C` are different matrices. There are three separate matrices on the host for matrix a, b, and c.
- The second example implements `A=A+B`, where matrix `A` is overwritten by the result. Storage is reserved on the host for matrix a and matrix b. The result of a+b is stored in matrix b.

Implementation 1



Implementation 2

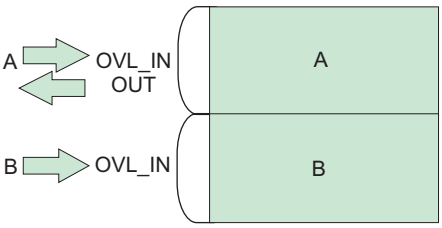


Figure 16. The two overlapped I/O buffer samples

Matrix setup

Note: The code is similar to the matrix_add example, see "Matrix add - host data partitioning example" on page 167. Here only the relevant code listing is shown.

```
/* ---------------------------------------------- */
/* matrix declaration for the two cases           */
/* ---------------------------------------------- */
#ifdef C_A_B // C = A + B
       alf_data_int32_t mat_a[ROW_SIZE][COL_SIZE];  // the matrix a
       alf_data_int32_t mat_b[ROW_SIZE][COL_SIZE];  // the matrix b
       alf_data_int32_t mat_c[ROW_SIZE][COL_SIZE];  // the matrix c
#else  // A = A + B
       alf_data_int32_t mat_a[ROW_SIZE][COL_SIZE];  // the matrix a
       alf_data_int32_t mat_b[ROW_SIZE][COL_SIZE];  // the matrix b
#endif
```

Work block setup

This code segment shows the work block creation process for the two cases.

```
for (i = 0; i < ROW_SIZE; i+=PART_SIZE){
     if(i+PART_SIZE <= ROW_SIZE)
        wb_parm.num_data = PART_SIZE;
     else
        wb_parm.num_data = ROW_SIZE - i;

     alf_wb_create(task_handle, ALF_WB_SINGLE, 0, &wb_handle);

     #ifdef C_A_B // C = A + B
            // the input data A and B
            alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_IN, 0);  // offset at 0
            alf_wb_dtl_entry_add(wb_handle, &mat_a[i][0], wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // A
            alf_wb_dtl_entry_add(wb_handle, &mat_b[i][0], wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // B
            alf_wb_dtl_end(wb_handle);

            // the output data C is overlapped with input data A
            // offset at 0, this is overlapped with A
            alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_OUT, 0);
            alf_wb_dtl_entry_add(wb_handle, &mat_c[i][0], wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // C
            alf_wb_dtl_end(wb_handle);

      #else // A = A + B
            // the input and output data A
            alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_INOUT, 0); // offset 0
            alf_wb_dtl_entry_add(wb_handle, &mat_a[i][0], wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // A
            alf_wb_dtl_end(wb_handle);

            // the input data B is placed after A
            // placed after A
            alf_wb_dtl_begin(wb_handle, ALF_BUF_OVL_IN, wb_parm.num_data*COL_SIZE*sizeof(alf_data_int32_t));
            alf_wb_dtl_entry_add(wb_handle, &mat_b[i][0], wb_parm.num_data*COL_SIZE, ALF_DATA_INT32); // B
```

```
        alf_wb_dtl_end(wb_handle);
    #endif
    alf_wb_parm_add(wb_handle, (void *)&wb_parm, sizeof(wb_parm)/sizeof(unsigned int), ALF_DATA_INT32, 0);
    alf_wb_enqueue(wb_handle);
}
```

Accelerator code

The accelerator code is shown here. In both cases, the output sc can be set to the same location in accelerator memory as sa and sb.

```
/* ------------------------------------------- */
/* the accelerator side code                   */
/* ------------------------------------------- */
/* the computation kernel function */
int comp_kernel(void *p_task_context, void *p_parm_ctx_buffer,
                void *p_input_buffer, void *p_output_buffer,
                void *p_inout_buffer, unsigned int current_count,
                unsigned int total_count)
{
  unsigned int i, cnt;
  int *sa, *sb, *sc;
  my_wb_parms_t *p_parm = (my_wb_parms_t *) p_parm_context;

  cnt = p_parm->num_data * COL_SIZE;

  sa = (int *) p_inout_buffer;
  sb = sa + cnt;
  sc = sa;

  for (i = 0; i < cnt; i ++)
       sc[i] = sa[i] + sb[i];

  return 0;
}
```

Task dependency example

This example shows how task dependency is used in a two stage pipeline application. The problem is a simple simulation.

An object P is placed in the middle of a flat surface with a bounding rectangular box. On each simulation step, the object moves in a random distance in a random direction. It moves back to the initial position when it hits the side walls of the bounding box. The problem is to calculate the number of hits to the four walls in a given time period.

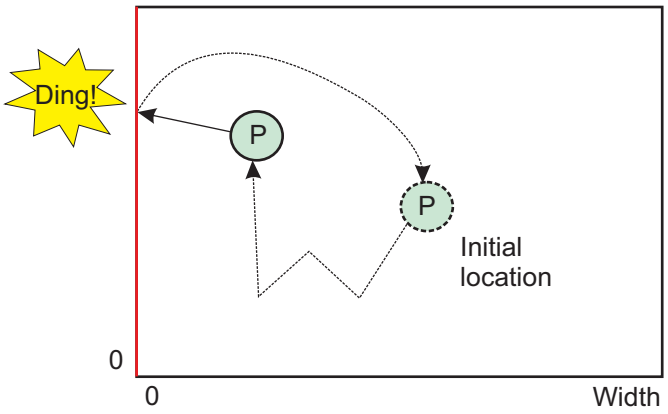


Figure 17. Object P randomly hits the side wall of the bounding box

A two stage pipeline is used to solve the problem so that the random number generation and the simulation can be paralleled:

- The first stage generates random numbers using a pseudo random number generator
- The second stage simulates the movements

Because ALF currently does not support pipeline directly, a pipeline structure is simulated using task dependency. There are two tasks which correspond to the two pipeline stages.

For this problem, each simulation step only needs a small amount of data just as a motion vector. Although ALF does not have a strict limit on how small the data can be, it is better to use larger data blocks for performance considerations. Therefore, the data for thousands of simulation steps is grouped into a single work block.

Stage 1 task: For the stage 1 task, a Lagged Fibonacci pseudo random number generator (PRNG) is used for simplicity. In this example, the algorithm is as follows:

```
Sn=(Sn-j^Sn-k)%232
```

where k > n > 0 and k = 71, j = 65

The algorithm requires a length k history buffer to save the older values. In this implementation, the task context is used for the history buffer. Because no input data is needed, the work block for this task only has output data.

Stage 2 task: For the stage 2 task, the task context is used to save the current status of the simulation including the position of the object and the number of hits to the walls. The work block in this stage only has input data, which are the PRNG results from stage 1.

Another target of pipelining is to overlap the execution of different stages for performance improvement. However, this requires work block level task synchronization between stages, and this is not yet supported by ALF. The alternative approach is to use multiple tasks whereby each task only handles a percentage of the work blocks for the whole simulation.

So there are now two stage tasks. For each chunk of work blocks, the following two tasks are created:

- The stage 1 task generates the random numbers and writes out the results to a temporary buffer
- The stage 2 task reads the random numbers from the temporary buffer to do the simulation

A task dependency is set between the two tasks to make sure the stage 2 task can get the correct results from stage 1 task. Because both the PRNG and the simulation have internal states, you have to pass the states data between the succeeding tasks of the same stage to preserve the states. The approach described here lets the tasks for the same stage share the same task context buffer. Dependencies are used to make sure the tasks access the shared task context in the correct order.

Figure 18 on page 181 (a) shows the task dependency as described in previous discussions. To further reduce the use of temporary intermediate buffers, you can use double or multi-buffering technology for the intermediate buffers. The task dependency graph for double buffering the intermediate buffers is shown in Figure 18 on page 181 (b), where a new dependency is added between the n-2th

stage 2 task and the nth stage 1 task to make sure the stage 1 task does not overwrite the data that may still be in use by the previous stage 2 task. This is what is implemented in the sample code.

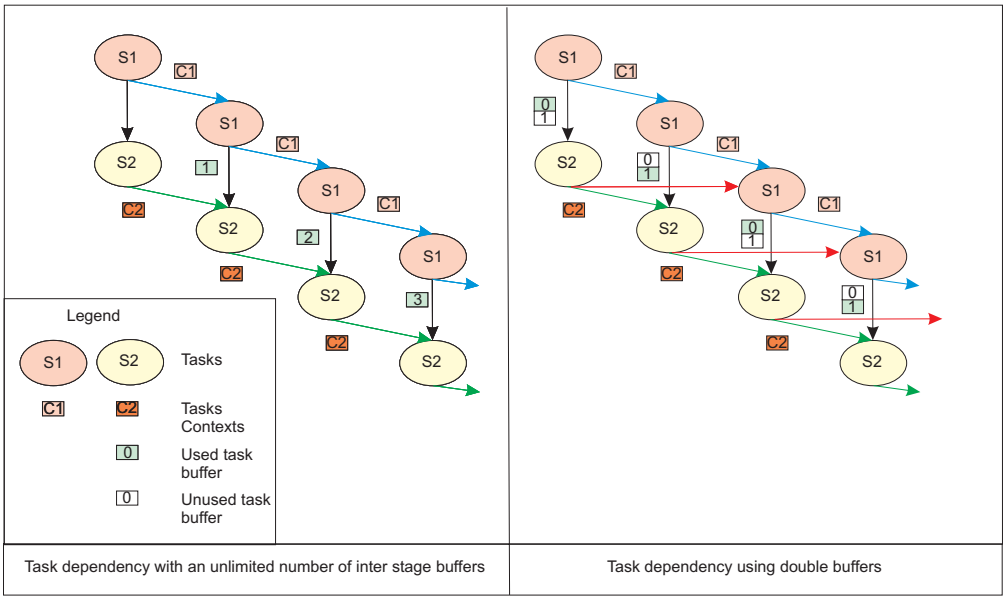


Figure 18. Task dependency examples

Source code

The complete source code can be found in the sample directory `pipe_line`.

Data set example

This example shows the addition of two matrixes and the creation and use of a data set within it.

The data set is created, and is referred to by `dataset_handle`. The data set consists of two buffers:

- `mat_a` with a size of NUM_ROW*NUM_COL*sizeof(float), and an `access_mode` of `ALF_DATASET_READ_ONLY`
- `mat_b` with a size of NUM_ROW*NUM_COL*sizeof(float), and an `access_mode` of `ALF_DATASET_READ_WRITE`

The data set is associated with the task referred to by `task_handle`.

The following section of code marked in **bold** shows the lines of code that have been added for data set support.

Source code

The source code is provided with the shipped samples in the following directory:

`matrix_add/common/dataset`

Appendix D. ALF trace events

The following shows the ALF trace events that are defined. In general, there are two trace hooks per API:

- The first traces the input parameters
- The second traces the output values as well as the time interval of the API call

ALF API hooks

Enabled with: TRACE_ALF_DEBUG.

Table 6. ALF debug hooks

Hook identifier	Traced values
_ALF_DATASET_ASSOCIATE_ENTRY	task_handle, dataset_handle
_ALF_DATASET_ASSOCIATE_EXIT_INTERVAL	retcode
_ALF_DATASET_BUFFER_ADD_ENTRY	dataset_handle, buffer, size, access_mode
_ALF_DATASET_BUFFER_ADD_EXIT_INTERVAL	retcode
_ALF_DATASET_CREATE_ENTRY	alf_handle, p_dataset_handle
_ALF_DATASET_CREATE_EXIT_INTERVAL	dataset_handle, retcode
_ALF_DATASET_DESTROY_ENTRY	dataset_handle
_ALF_DATASET_DESTROY_EXIT_INTERVAL	retcode
_ALF_EXIT_ENTRY	alf_handle, exit_policy, timeout
_ALF_EXIT_EXIT_INTERVAL	retcode
_ALF_GENERIC_DEBUG	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_ALF_INIT_ENTRY	sys_config_info, alf_handle_ptr
_ALF_INIT_EXIT_INTERVAL	rtn
_ALF_NUM_INSTANCES_SET_ENTRY	alf_handle, number_of_instances
_ALF_NUM_INSTANCES_SET_EXIT_INTERVAL	retcode
_ALF_QUERY_SYSINFO_ENTRY	alf_handle, query_info, accel_type, p_query_result
_ALF_QUERY_SYSINFO_EXIT_INTERVAL	query_result, retcode
_ALF_REGISTER_ERROR_HANDLER_ENTRY	alf_handle, error_handler_function, p_context
_ALF_REGISTER_ERROR_HANDLER_EXIT_INTERVAL	retcode
_ALF_TASK_CREATE_ENTRY	task_desc_handle, p_task_context_data, num_accelerators, tsk_attr, wb_dist_size, p_task_handle
_ALF_TASK_CREATE_EXIT_INTERVAL	task_handle, retcode
_ALF_TASK_DEPENDS_ON_ENTRY	task_handle_dependent, task_handle
_ALF_TASK_DEPENDS_ON_EXIT_INTERVAL	retcode
_ALF_TASK_DESC_CREATE_ENTRY	alf_handle, accel_type, task_desc_handle_ptr
_ALF_TASK_DESC_CREATE_EXIT_INTERVAL	desc_info_handle, retcode
_ALF_TASK_DESC_CTX_ENTRY_ADD_ENTRY	task_desc_handle, data_type, size
_ALF_TASK_DESC_CTX_ENTRY_ADD_EXIT_INTERVAL	retcode

Table 6. ALF debug hooks (continued)

Hook identifier	Traced values
_ALF_TASK_DESC_DESTROY_ENTRY	task_desc_handle
_ALF_TASK_DESC_DESTROY_EXIT_INTERVAL	retcode
_ALF_TASK_DESC_SET_INT32_ENTRY	task_desc_handle, field, value
_ALF_TASK_DESC_SET_INT32_EXIT_INTERVAL	retcode
_ALF_TASK_DESC_SET_INT64_ENTRY	task_desc_handle, field, value
_ALF_TASK_DESC_SET_INT64_EXIT_INTERVAL	retcode
_ALF_TASK_DESTROY_ENTRY	task_handle
_ALF_TASK_DESTROY_EXIT_INTERVAL	retcode
_ALF_TASK_EVENT_HANDLER_REGISTER_ENTRY	task_handle, task_event_handler, p_data, data_size, event_mask
_ALF_TASK_EVENT_HANDLER_REGISTER_EXIT_INTERVAL	retcode
_ALF_TASK_FINALIZE_ENTRY	task_handle
_ALF_TASK_FINALIZE_EXIT_INTERVAL	retcode
_ALF_TASK_QUERY_ENTRY	talk_handle, p_unfinished_wbs, p_total_wbs
_ALF_TASK_QUERY_EXIT_INTERVAL	unfinished_wbs, total_wbs, retcode
_ALF_TASK_WAIT_ENTRY	task_handle, time_out
_ALF_TASK_WAIT_EXIT_INTERVAL	retcode
_ALF_WB_CREATE_ENTRY	task_handle, work_block_type, repeat_count, p_wb_handle
_ALF_WB_CREATE_EXIT_INTERVAL	wb_handle, retcode
_ALF_WB_DTL_SET_BEGIN_ENTRY	wb_handle, buffer_type, offset_to_the_local_buffer
_ALF_WB_DTL_SET_BEGIN_EXIT_INTERVAL	retcode
_ALF_WB_DTL_SET_END_ENTRY	wb_handle
_ALF_WB_DTL_SET_END_EXIT_INTERVAL	retcode
_ALF_WB_DTL_SET_ENTRY_ADD_ENTRY	wb_handle, p_address, size_of_data, data_type
_ALF_WB_DTL_SET_ENTRY_ADD_EXIT_INTERVAL	retcode
_ALF_WB_ENQUEUE_ENTRY	wb_handle
_ALF_WB_ENQUEUE_EXIT_INTERVAL	retcode
_ALF_WB_PARM_ADD_ENTRY	wb_handle, pdata, size_of_data, data_type, address_alignment
_ALF_WB_PARM_ADD_EXIT_INTERVAL	retcode

ALF performance hooks

These trace hooks are enabled by LIBALF_PERF group (0x08) in the config file.

The COUNTERS and TIMERS hooks contain data that is accumulated during the ALF calls. Currently, that data and these trace events will get reported at various ALF exit calls.

Table 7. ALF performance hooks

Hook Identifier	Traced values
_ALF_GENERIC_PERFORM_HOST	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_ALF_GENERIC_PERFORM_SPU	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10
_ALF_HOST_COUNTERS	alf_task_creates, alf_task_waits, alf_wb_enqueues, thread_total_count, thread_reuse_count, x
_ALF_HOST_TIMERS	alf_runtime, alf_accel_utilize, x1, x2
_ALF_SPU_COUNTERS	alf_input_bytes, alf_output_bytes, alf_workblock_total, double_buffer_used, x1, x2
_ALF_SPU_TIMERS	alf_lqueue_empty, alf_wait_data_dtl, alf_prep_input_dtl, alf_prep_output_dtl, alf_compute_kernel, alf_spu_task_run, x1, x2
_ALF_TASK_BEFORE_EXEC_INTERVAL	task_flag
_ALF_TASK_CONTEXT_MERGE_INTERVAL	task_flag
_ALF_TASK_CONTEXT_SWAP_INTERVAL	task_flag
_ALF_TASK_EXEC_INTERVAL	task_flag
_ALF_THREAD_RUN_INTERVAL	task_flag
_ALF_WAIT_FIRST_WB_INTERVAL	task_flag, wb_flag, packet_flag
_ALF_WB_COMPUTE_KERNEL_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_DATA_TRANSFER_WAIT_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_DTL_PREPARE_IN_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_DTL_PREPARE_OUT_INTERVAL	task_flag, wb_flag, wb idx
_ALF_WB_LQUEUE_EMPTY_INTERVAL	task_flag, packet_flag

ALF SPU hooks

These trace hooks are enabled by LIBALF_SPU group (0x09) in the config file.

Table 8. ALF SPU hooks

Hook identifier	Traced values
_ALF_ACCEL_COMP_KERNEL_ENTRY	p_task_context, p_parm_ctx_buffer, p_input_buffer, p_output_buffer, p_inout_buffer, current_iter, num_iter
_ALF_ACCEL_COMP_KERNEL_EXIT	retcode
_ALF_ACCEL_DTL_BEGIN_ENTRY	p_dtl, buf_type, offset
_ALF_ACCEL_DTL_BEGIN_EXIT	p_dtl, retcode
_ALF_ACCEL_DTL_END_ENTRY	p_dtl
_ALF_ACCEL_DTL_END_EXIT _	retcode
_ALF_ACCEL_DTL_ENTRY_ADD_ENTRY	p_dtl, data_size, data_type, p_host_address
_ALF_ACCEL_DTL_ENTRY_ADD_EXIT	retcode
_ALF_ACCEL_INPUT_DTL_PREPARE_ENTRY	p_task_context, p_parm_ctx_buffer, p_dtl, current_iter, num_iter
_ALF_ACCEL_INPUT_DTL_PREPARE_EXIT	retcode
_ALF_ACCEL_NUM_INSTANCES	retcode

Table 8. ALF SPU hooks (continued)

Hook identifier	Traced values
_ALF_ACCEL_OUTPUT_DTL_PREPARE_ENTRY	p_task_context, p_parm_ctx_buffer, p_io_container, current_iter, num_iter
_ALF_ACCEL_OUTPUT_DTL_PREPARE_EXIT	retcode
_ALF_ACCEL_TASK_CONTEXT_MERGE_ENTRY	p_task_context_to_be_merged, p_task_context
_ALF_ACCEL_TASK_CONTEXT_MERGE_EXIT	retcode
_ALF_ACCEL_TASK_CONTEXT_SETUP_ENTRY	p_task_context
_ALF_ACCEL_TASK_CONTEXT_SETUP_EXIT	retcode
_ALF_INSTANCES_ID	retcode
_ALF_SPE_GENERIC_DEBUG	long1, long2, long3, long4, long5, long6, long7, long8, long9, long10

Appendix E. Attributes and descriptions

The following table is a list of attributes.

Table 9. Attributes and descriptions

Attribute name	Description
ALF_QUERY_NUM_ACCEL	Return the number of accelerators of a particular type accel_type in the system.
ALF_QUERY_HOST_MEM_SIZE	Return the memory size of control nodes up to 4T bytes, in units of kilo bytes (2^10 bytes).
ALF_QUERY_HOST_MEM_SIZE_EXT	Return the memory size of control nodes, in units of 4T bytes (2^42 bytes)
ALF_QUERY_ACCEL_MEM_SIZE	Return the memory size of accelerator nodes up to 4T bytes, in units of kilo bytes (2^10 bytes).
ALF_QUERY_ACCEL_MEM_SIZE_EXT	Return the memory size of accelerator nodes, in units of 4T bytes (2^42 bytes).
ALF_QUERY_HOST_ADDR_ALIGN	Return the basic requirement of memory address alignment on control node side, in exponential of 2.
ALF_QUERY_ACCEL_ADDR_ALIGN	Return the basic requirement of memory address alignment on accelerator node side, in exponential of 2.
ALF_QUERY_DTL_ADDR_ALIGN	Return the address alignment of data transfer list entries, in exponential of 2.
ALF_ACCEL_TYPE_SPE	Accelerator type.
ALF_EXIT_POLICY_FORCE	Perform a shutdown immediately and aborts all unfinished tasks if there are any.
ALF_EXIT_POLICY_WAIT	Wait for all tasks to be processed and then shuts down.
ALF_EXIT_POLICY_TRY	Return with a failure if there are unfinished tasks.
ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE	Size of the work block parameter buffer.
ALF_TASK_DESC_WB_IN_BUF_SIZE	Size of the work block input buffer.
ALF_TASK_DESC_WB_OUT_BUF_SIZE	Size of the work block output buffer.
ALF_TASK_DESC_WB_INOUT_BUF_SIZE	Size of the work block overlapped input/output buffer.
ALF_TASK_DESC_NUM_DTL_ENTRIES	Maximum number of entries for the data transfer list.
ALF_TASK_DESC_TSK_CTX_SIZE	Size of the task context buffer.
ALF_TASK_DESC_PARTITION_ON_ACCEL	Specifies whether the accelerator functions are invoked to generate data transfer lists for input and output data.
ALF_TASK_DESC_MAX_STACK_SIZE	Specify the maximum stack size.
ALF_TASK_DESC_ACCEL_LIBRARY_REF_L	Specify the name of the library that the accelerator image is contained in
ALF_TASK_DESC_ACCEL_IMAGE_REF_L	Specify the name of the accelerator image that's contained in the library.

Table 9. Attributes and descriptions (continued)

Attribute name	Description
ALF_TASK_DESC_ACCEL_KERNEL_REF_L	Specify the name of the computational kernel function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L	Specify the name of the input list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_OUTPUT_DTL_REF_L	Specify the name of the output list prepare function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_CTX_SETUP_REF_L	Specify the name of the context setup function, this usually is a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_ACCEL_CTX_MERGE_REF_L	Specify the name of the context merge function, this usually a string constant that the accelerator runtime could use to find the correspondent function.
ALF_TASK_DESC_TASK_TYPE	The task type. The following is defined as the value parameter:
ALF_TASK_DESC_ACCEL_LTS_MAIN_REF_L	Specify the name of the main entry point function of the lightweight task. This is usually a string constant that the accelerator runtime uses to find the corresponding function pointer.
ALF_TASK_ATTR_SCHED_FIXED	The task must be scheduled on `num_instances` of accelerators.
ALF_TASK_ATTR_WB_CYCLIC	The work blocks for this task is distributed to the accelerators in a cyclic order as specified by `num_instances`.
ALF_TASK_EVENT_TYPE_T	Defined as followed: • ALF_TASK_EVENT_FINALIZED: This task has been finalized. No additional work block can be added to this task. • ALF_TASK_EVENT_READY: This task has been scheduled for execution. • ALF_TASK_EVENT_FINISHED: All work blocks in this task have been processed. • ALF_TASK_EVENT_INSTANCE_START: One new instance of the task is started on an accelerator after the event handler returns • ALF_TASK_EVENT_INSTANCE_END: One existing instance of the task ends and the task context has been copied out to the original location or has been merged to another current instance of the same task. • ALF_TASK_EVENT_DESTROY: The task is destroyed explicitly
ALF_WB_SINGLE	Create a single use work block.

Table 9. Attributes and descriptions (continued)

Attribute name	Description
ALF_WB_MULTI	Create a multi use work block. This work block type is only supported when the task is created with ALF_TASK_DESC_PARTITION_ON_ACCEL set to non-zero.
ALF_BUF_IN	Input to the input-only buffer.
ALF_BUF_OUT	Output from the output only buffer.
ALF_BUF_OVL_IN	Input to the overlapped buffer.
ALF_BUF_OVL_OUT	Output from the overlapped buffer.
ALF_BUF_OVL_INOUT	In/out to/from the overlapped buffer.
ALF_DATASET_READ_ONLY	The dataset is read-only. Work blocks referencing the data in this buffer cannot update this buffer as an output buffer.
ALF_DATASET_WRITE_ONLY	The dataset is write-only. Work blocks referencing the data in this buffer as input data results in indeterminate behavior.
ALF_DATASET_READ_WRITE	The dataset allows both read and write access. Work blocks can use this buffer as input buffers and output buffers and/or inout buffers.
ALF_TASK_TYPE_T	Defined as follows: • ALF_TASK_TYPE_WORKBLOCK Work block task type • ALF_TASK_TYPE_LIGHWEIGHT Lightweight task type

Appendix F. Error codes and descriptions

The following table is a list of the ALF error codes.

Table 10. Error codes and descriptions

Error	Error code	Description
ALF_ERR_PERM	1	No permission
ALF_ERR_SRCH	3	No such task
ALF_ERR_2BIG	7	I/O buffer request exceeds limitations
ALF_ERR_NOEXEC	8	Cannot execute task
ALF_ERR_BADF	9	Bad handle
ALF_ERR_AGAIN	11	Try again
ALF_ERR_NOMEM	12	Out of memory
ALF_ERR_FAULT	14	Invalid address
ALF_ERR_BUSY	16	Resource busy
ALF_ERR_INVAL	22	Invalid argument
ALF_ERR_RANGE	34	Out of range
ALF_ERR_NOSYS	38	Function not implemented
ALF_ERR_BADR	53	Resource request cannot be fulfilled
ALF_ERR_NODATA	61	No more data available
ALF_ERR_TIME	62	Time out
ALF_ERR_COMM	70	Communications error
ALF_ERR_PROTO	71	Internal protocol error
ALFF_ERR_BADMSG	74	Unrecognized message
ALF_ERR_OVERFLOW	75	Overflow
ALF_ERR_INCOMPAT	76	Accelerator incompatibility
ALF_ERR_NOBUFS	105	No buffer space available
ALF_ERR_ACCEL	2000	Generic accelerator error

Appendix G. Related documentation

This topic helps you find related information.

Document location

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

`http://www.ibm.com/developerworks/power/cell/`

Click the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*

Programming

- *Cell Broadband Engine Programmer's Guide*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Programming Tutorial*

Library

- *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference*
- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Fast Fourier Transform Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*
- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Runtime Extensions Library Programmer's Guide and API Reference*

- *Three dimensional FFT Prototype Library Programmer's Guide and API Reference*

Installation

- *SDK for Multicore Acceleration Version 3.1 Installation Guide*

Tools

- *Getting Started - XL C/C++ for Multicore Acceleration for Linux*
- *Compiler Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Language Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Programming Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Installation Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Getting Started - XL Fortran for Multicore Acceleration for Linux*
- *Compiler Reference - XL Fortran for Multicore Acceleration for Linux*
- *Language Reference - XL Fortran for Multicore Acceleration for Linux*
- *Optimization and Programming Guide - XL Fortran for Multicore Acceleration for Linux*
- *Installation Guide - XL Fortran for Multicore Acceleration for Linux*
- *Performance Analysis with the IBM Full-System Simulator*
- *IBM Full-System Simulator User's Guide*
- *IBM Visual Performance Analyzer User's Guide*

IBM PowerPC Base

- *IBM PowerPC Architecture™ Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *IBM PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

Appendix H. Accessibility features

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

The following list includes the major accessibility features:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are tactilely discernible and do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

IBM and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able/ for more information about the commitment that IBM has to accessibility.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "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. IBM 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 non-IBM Web sites 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 IBM product and use of those Web sites is at your own risk.

IBM 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:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

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, 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 non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

All IBM prices shown are IBM'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 IBM, 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. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

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

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

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 is a trademark 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.

Glossary

ABI

Application Binary Interface. This is the standard that a program follows to ensure that code generated by different compilers (and perhaps linking with various, third-party libraries) run correctly on the Cell BE. The ABI defines data types, register use, calling conventions and object formats.

accelerator

General or special purpose processing element in a hybrid system. An accelerator can 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.

ALF

Accelerated Library Framework. This an API that provides a set of services to help programmers solving data parallel problems on a hybrid system. ALF supports the multiple-program-multiple-data (MPMD) programming style where multiple programs can be scheduled to run on multiple accelerator elements at the same time. ALF offers programmers an interface to partition data across a set of parallel processes without requiring architecturally-dependent code.

API

Application Program Interface.

ATO

Atomic Unit. Part of an SPE's MFC. It is used to synchronize with other processor units.

Broadband Engine

See *CBEA*.

C++

C++ is an object-orientated programming language, derived from C.

cache

High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.

CBEA

Cell Broadband Engine Architecture. A new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the Cell Broadband Engine are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

Cell/B.E. processor

The Cell/B.E. processor is a multi-core broadband processor based on IBM's Power Architecture.

Cell Broadband Engine processor

See *Cell BE*.

cluster

A collection of nodes.

compiler

A programme that translates a high-level programming language, such as C++, into executable code.

computational 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.

data set

An ALF data set is a logical set of data buffers.

DMA

Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

DMA command

A type of MFC command that transfers or controls the transfer of a memory location containing data or instructions. See *MFC*.

DMA list

A sequence of transfer elements (or list entries) that, together with an initiating DMA-list command, specify a sequence of DMA transfers between a single area of LS and discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA-list command such as `getl` or `putl`. DMA-list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

DMA-list command

A type of MFC command that initiates a sequence of DMA transfers specified by a DMA list stored in an SPE's LS. See *DMA list*.

EA

See *Effective address*.

effective address

An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective address space is 2^{64} bytes.

exception

An error, unusual condition, or external signal that may alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled. See interrupt.

FFT

Fast Fourier Transform.

GCC

GNU C compiler

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.

HTTP

Hypertext Transfer Protocol. A method used to transfer or convey information on the World Wide Web.

Hybrid

A module comprised of two Cell BE cards connected via an AMD Opteron processor.

IDL

Interface definition language. Not the same as CORBA IDL

kernel

The core of an operating which provides services for other parts of the operating system and provides multitasking. In Linux or UNIX operating system, the kernel can easily be rebuilt to incorporate enhancements which then become operating-system wide.

latency

The time between when a function (or instruction) is called and when it returns. Programmers often optimize code so that functions return as quickly as possible; this is referred to as the low-latency approach to optimization. Low-latency designs often leave the processor data-starved, and performance can suffer.

local store

The 256-KB local store associated with each SPE. It holds both instructions and data.

LS

See *local store*.

main storage

The effective-address (EA) space. It consists physically of real memory (whatever is external to the memory-interface controller, including both volatile and nonvolatile memory), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices (all I/O is memory-mapped), and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. See also *local store*.

main thread

The main thread of the application. In many cases, Cell BE 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.

MFC

Memory Flow Controller. Part of an SPE which provides two main functions: it moves data via DMA between the SPE's local store (LS) and main storage, and it synchronizes the SPU with the rest of the processing units in the system.

MPMD

Multiple Program Multiple Data. Parallel programming model with several distinct executable programs operating on different sets of data.

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).

PDF

Portable document format.

pipelining

A technique that breaks operations, such as instruction processing or bus transactions, into smaller stages so that a subsequent stage in the pipeline can begin before the previous stage has completed.

PPE

PowerPC Processor Element. The general-purpose processor in the Cell.

PPE

PowerPC Processor Element. The general-purpose processor in the Cell BE processor.

PPU

PowerPC Processor Unit. The part of the *PPE* that includes the execution units, memory-management unit, and L1 cache.

process

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

program section

See *code section*.

SDK

Software development toolkit for Multicore Acceleration. A complete package of tools for application development.

section

See *code section*.

SIMD

Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SPE

Synergistic Processor Element. Extends the PowerPC 64 architecture by acting as cooperative offload processors (synergistic processors), with the direct memory access (DMA) and synchronization mechanisms to communicate with them (memory flow control), and with enhancements for real-time management. There are 8 SPEs on each cell processor.

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

synchronization

The order in which storage accesses are performed.

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.

vector

An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.

virtual memory

The address space created using the memory management facilities of a processor.

virtual storage

See *virtual memory*.

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.

workload

A set of code samples in the SDK that characterizes the performance of the architecture, algorithms, libraries, tools, and compilers.

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.

x86

Generic name for Intel-based processors.

XLC

The IBM optimizing C/C++ compiler.

Index

E

F

H

I

L

M

O

P

R

S

T

W

Printed in USA

SC33-8333-03