

ClearSpeed[™]

User Guide

The CSPX Accelerator Interface Library

Document No. 06-UG-1555 Revision: 2.F

September 2010

Overview

This document is an introduction and reference manual for CSPX, an application-level software interface for ClearSpeed's accelerators.

The first part of this document provides a user guide for the CSPX functions. The second part is a reference to all of the functions, command line options, and so on.

User guide

Chapter 1: Getting started, presents an overview of what the CSPX library is and how it works. This includes a quick guide to getting started with a simple program running on the host that calls functions on an Advance accelerator.

Chapter 2: Basic process and object model, provides a more detailed explanation of the underlying process and object model used by CSPX using the C++ interface.

Chapter 3: C interface to processes and objects, describes how to use the C language interface.

Chapter 4: Double-buffered communication, explains how double-buffered data objects can be used to overlap computation and communication more effectively.

Chapter 5: Data pipes, describes the use of named pipes in CSPX.

Chapter 6: Profiling CSPX operations, shows how to use the ClearSpeed Visual Profiler with programs built using CSPX.

Reference

Chapter 7: Compiling and running CSPX programs, provides a summary of the commands and options used to build a program using CSPX, how errors are handled and how to run code on a simulator.

Chapter 8: C++ API reference, is a reference for the C++ host-side API.

Chapter 9: C API reference, is a reference for the standard C host-side API.

Chapter 10: Cⁿ API reference, is a reference to the Cⁿ API used by code on the accelerator.

The *Bibliography* provides a list of references and suggested further reading.

Open source

The source code for CSPX is available so that users can modify and extend it to meet their needs.

Table of contents

1	Getting started	9
1.1	CSPX overview	9
1.2	Remote procedure call	9
1.2.1	Using multiple accelerators	10
1.2.2	Parameter passing	10
1.2.3	Copying parameters	11
1.3	RPC example	12
1.3.1	Accelerator code	12
1.3.2	Host code	14
1.3.3	Running on a simulator	14
1.3.4	Handling errors	15
2	Basic process and object model	17
2.1	Process and communication model	17
2.2	CSPX processes	19
2.2.1	Calling functions from the host	19
2.2.2	Connecting to an accelerator	19
2.2.3	Running on a simulator	19
2.3	Object communication model	20
2.3.1	Objects	20
2.3.2	Communicating with an accelerator	20
2.3.3	Shared objects	22
2.3.4	Data alignment	22
2.3.5	Object attributes	22
2.4	Accelerator code	22
2.4.1	Exporting functions	22
2.5	Invoking a function on the accelerator	23
2.5.1	Passing parameters	23
2.5.2	Dealing with return values	24
2.6	Policy templates	26
2.6.1	Distribution and connection policies	26
2.6.2	Partition policies for objects	26
2.6.3	Custom policies	28

2.7	C++ example	29
2.7.1	Function arguments	29
2.7.2	Accelerator code	29
2.7.3	Host code	29
3	C interface to processes and objects	33
3.1	Creating processes	33
3.1.1	Running on a simulator	34
3.2	Object communication model	34
3.2.1	Operations	34
3.2.2	Common namespace	36
3.3	C example	36
3.3.1	Function arguments	36
3.3.2	Host code	36
3.3.3	Accelerator code	36
3.4	Process group interface	38
3.5	Example of process group use	38
3.5.1	Accelerator code	38
3.5.2	Host code	38
3.6	Extending the object model to the process group	40
4	Double-buffered communication	43
4.1	Using double-buffered objects	43
4.1.1	Double-buffered transfers to accelerator	44
4.1.2	Double-buffered transfers from accelerator	45
4.2	Example	45
4.2.1	Accelerator code	46
4.2.2	C++ host code	46
4.2.3	C host code	46
5	Data pipes	51
5.1	Basic pipe functions	51
5.2	Efficiency considerations	52
5.3	Groups of pipes	52

- 6 Profiling CSPX operations 55**
 - 6.1 Trace generation 55
 - 6.2 Events 55

- 7 Compiling and running CSPX programs 57**
 - 7.1 Header files 57
 - 7.2 Compiling RPC functions Cⁿ programs 57
 - 7.3 Linking Cⁿ programs 58
 - 7.4 Error handling 58
 - 7.4.1 Error codes 58
 - 7.4.2 Error functions 60
 - 7.4.3 C++ exceptions 60
 - 7.5 Running code on a simulator 61
 - 7.5.1 C++ interface 61
 - 7.5.2 C interface 61
 - 7.5.3 Environment variables 61

- 8 C++ API reference 63**
 - 8.1 Processes 63
 - 8.1.1 Classes 63
 - 8.1.2 Member functions 63
 - 8.2 Objects 71
 - 8.2.1 Classes 71
 - 8.2.2 Member functions 71
 - 8.3 Parameter objects 75
 - 8.3.1 Classes 75
 - 8.3.2 Member functions 75
 - 8.4 Double buffered objects 82
 - 8.4.1 Classes 82
 - 8.4.2 Member functions 82
 - 8.5 Connection policies 86
 - 8.5.1 Classes 86
 - 8.5.2 Member functions 86
 - 8.6 Distribution policies 87
 - 8.6.1 Classes 87
 - 8.6.2 Member functions 87

8.7	Partition policies	89
8.7.1	Classes	89
8.7.2	Member functions	89
8.8	Streams	91
8.8.1	Classes	91
8.8.2	Member functions	92
8.9	Asynchronous operation	95
8.9.1	Classes	95
8.9.2	Member functions	95
8.10	Semaphores	98
8.10.1	Classes	98
8.10.2	Member functions	98
8.11	Reduction functions	101
8.11.1	Functions	101
9	C API reference	105
9.1	Processes	105
9.1.1	Types	105
9.1.2	Functions	106
9.2	Objects	112
9.2.1	Constants	112
9.2.2	Types	112
9.2.3	Functions	113
9.3	Process groups	120
9.3.1	Types	120
9.3.2	Constants	120
9.3.3	Functions	120
9.4	Object groups	126
9.4.1	Types	126
9.4.2	Functions	126
9.5	Double buffered objects	130
9.5.1	Types	130
9.5.2	Functions	130
9.6	Pipes	137
9.6.1	Types	137
9.6.2	Functions	137

9.7	Error handling	142
9.7.1	Types	142
9.7.2	Functions	142
10	C" API reference	145
10.1	Pragmas	145
10.1.1	RPC pragma	145
10.1.2	Export pragma	146
10.2	Processes	146
10.2.1	Types	146
10.2.2	Functions	146
10.3	Objects	149
10.3.1	Constants	149
10.3.2	Types	149
10.3.3	Functions	149
10.4	Double buffered objects	154
10.4.1	Types	154
10.4.2	Functions	154
10.5	Pipes	159
10.5.1	Types	159
10.5.2	Functions	159
10.6	Error handling	163
10.6.1	Types	163
10.6.2	Functions	163
	Bibliography	165

1 Getting started

This chapter provides an overview of the CSPX interface and provides the basic instructions required to compile and run a program that calls functions on an accelerator.

1.1 CSPX overview

The CSPX application programming interface (API) provides a software interface for applications using ClearSpeed's acceleration products. This is a more 'application friendly' interface than the other APIs provided by the device driver and CSAPI. [Figure 1 on page 10](#) shows where CSPX fits in the software stack.

The software provides three levels of interface between the host and accelerators:

1. The device driver has a very low-level, hardware-oriented API which simply provides functions to read and write memory and registers in the target hardware. This is not intended to be used directly by the programmer.
2. The 'device-level' interface provided by CSAPI has basic support for loading and executing code on CSX processors. It also allows data to be transferred between host and accelerator memory, and supports access to hardware features such as semaphores, DMA operations and memory management.
3. CSPX provides a higher-level API that can easily be used by an application which is split across the host and accelerators. CSPX is used by a host application to call functions running on the accelerator hardware and pass data to and from those accelerated functions. CSPX is built on top of CSAPI. It complements and extends the features provided by CSAPI. This allows you to concentrate on decomposing the application problem for acceleration and on implementing the algorithms on the host and accelerator, rather than the interface between them.

CSPX has two modes of use. There is a simple remote procedure call model which allows applications to call function on the accelerator as if they were part of the same program. This allows you to get started quickly using ClearSpeed accelerators for your software.

Once your code is running, you may want to optimize the communication between the host application and the accelerator. The full set of CSPX functions provide complete control over what data is moved and when, and provide easy access to double-buffered transfers. The full interface to CSPX is described in [Chapter 2](#) onwards.

1.2 Remote procedure call

The simplest way of using CSPX allows a host application to directly call functions running on an accelerator exactly as if they were part of the host program.

A pragma in the **C** source code defines the functions which are available to the host using this remote procedure call (RPC) mechanism. The `csp_x_rpc` pragma defines how the parameters to the RPC function will be handled, and how the code is to be run on multiple accelerators. The pragma also causes the compiler to generate host source files which contain 'wrapper' code to enable this transparent RPC.

This basic RPC mechanism makes it easy to call an accelerated function passing data as arguments. Results can be returned in arguments and as the return value of the function.

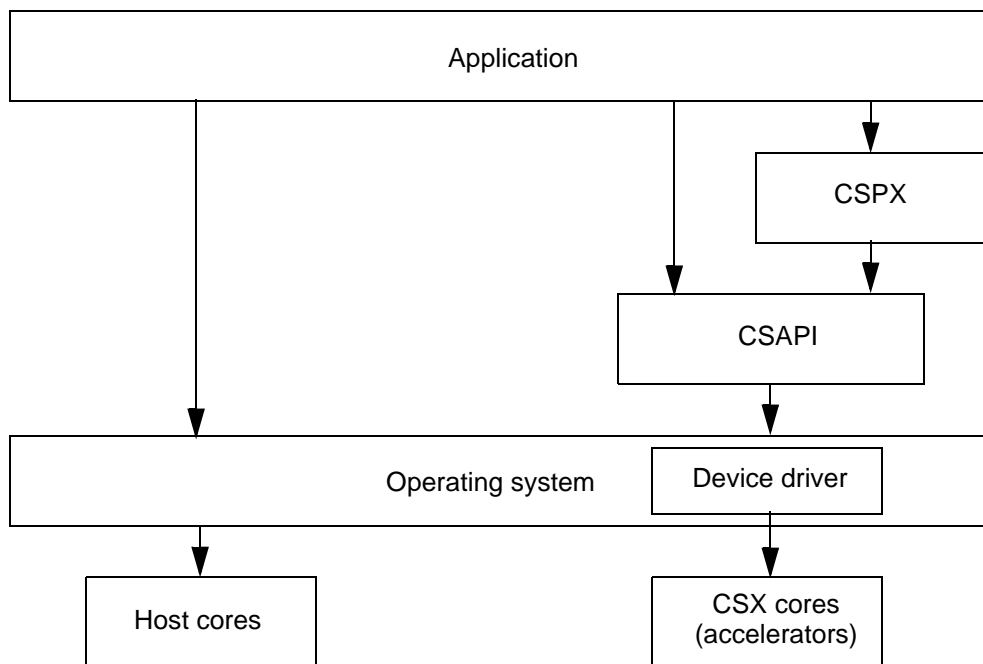


Figure 1. CSPX in the software stack

This can be an easy first step to porting an application to use an accelerator, before moving on to the greater flexibility offered by the rest of the CSPX API.

The RPC interface is implemented by automatically generating C++ code to call the accelerated functions. This can be used with an application written in either C++ or standard C as long as you use a C++ compiler to compile your application.

1.2.1 Using multiple accelerators

If multiple CSX processors are available, then a process can be created on each of them running the same compiled **Cⁿ** code. The default is to run the code on one accelerator. An alternative *distribution policy* can be specified in the RPC pragma. This can be one of the predefined ones or a custom policy (see [Section 8.6: Distribution policies on page 87](#) for more information on these policies).

1.2.2 Parameter passing

Data is passed to the accelerator as parameters of the function. Parameters can also be used to return results (even though they are not passed by reference). The return value of the function is also directly available on the host.

Function parameters can be of any type that does not involve pointers. Array parameters are treated slightly differently from scalar and other aggregate types in an RPC call.

Array parameters

Array parameters must have an associated integer parameter which specifies the size of the array (this is required because the size of an array declared on the host is not available to the **Cⁿ** compiler).

A variable passed as an array argument can be declared as an array or be a pointer to dynamically allocated memory (using `malloc`).

Normally, in a C program, arrays are passed as pointers and the content of the array is not copied. However, when making a remote procedure call, a copy of the array is used on the accelerator and a pointer to the copied data is passed to the called function.

Non-array parameters

Scalar and aggregate parameters can be of any type apart from those involving pointers.

1.2.3 Copying parameters

The host and accelerator have separate memory systems. This means that data needs to be copied to and from accelerators. As a result, the semantics of parameter passing are slightly different from a standard C function call.

The `cspcx_rpc` pragma defines how the parameters of the function will be used. This allows the system to optimize data movement and only copy data to or from the accelerator as necessary. Parameters can be defined as:

- **write:** used only to pass data from the host to the accelerator.
This behaves like a call by value: the data is copied to the accelerator and a pointer to the copied data is passed to the called function. The modified data is not copied back to the host.
- **read:** used only to return values from the accelerator.
Space is allocated for the variable but no data is copied *to* the accelerator. The value of the variable is copied back to the host when the function returns.
- **read/write:** used both for passing values to the accelerator and returning results.
This is similar to a call by reference: the data is copied to the accelerator and a pointer to the copied data is passed to the called function. When the function returns, the modified data is copied back to the host.

Sending data to accelerators: write parameters

Array write parameters are distributed across the accelerators. The default mode of operation is to divide the array into equal sized 'chunks' for each accelerator.

Scalar and other aggregate write parameters are broadcast to all of the accelerators used. In other words the same value is copied to every processor.

Results from accelerators: read parameters

An array read parameter contains the 'chunks' of data returned by each accelerator. These are combined in the same way as the write parameters are divided up; in other words, the default behavior is to concatenate the values returned by each processor into a contiguous array.

Scalar and other aggregate read parameters (and the function return value) will return a value from each processor. This means that the variable used for the returned value(s) must be defined as a vector in the host code. The alternative is to specify a reduction function to combine all the returned values to produce a single result. The reduction function to be used is specified in the RPC pragma.

Scalar and other aggregate read/write parameters *must* specify a reduction function (because they cannot be passed *to* the function as vectors).

1.3 RPC example

1.3.1 Accelerator code

The functions to be accelerated are written in **Cⁿ** and annotated with a pragma which defines how they are to be called from the host. For example, see the **Cⁿ** function in [Figure 2](#). This example simply calculates the square root of each of the input values, in batches, on all PEs concurrently.

```
#include <cspx.h>
#include <mathp.h>
#include <lib_ext.h>
#include <string_ext.h>

// export the function for remote calling
//  input parameters (inputs and count) are marked as 'write'
//  output parameter (results) is marked as 'read'
#pragma cspx_rpc write(inputs[i_size], count) read(results[r_size])

// a simple example of a Cn function
int square_root(double* inputs, unsigned int i_size,
               double* results, unsigned int r_size,
               int count) {
    int i;

    for (i = 0; i < count; i++) {
        poly double d;
        poly int index;

        // calculate address of data to operate on
        index = i * __NUM_PES__ + get_penum();
        // copy input from mono memory
        memcpym2p(&d, &inputs[index], sizeof(double));
        // perform calculation
        d = sqrtp(d);
        // write result back to mono memory
        memcyp2m(&results[index], &d, sizeof(double));
    }
}

int main(void) {
    // invoke cspx function to handle calls from the host
    CSPX_runtime();
}
```

Figure 2. Accelerator code: `sqrt_rpc.cn`

The `cspx_rpc` pragma applies to the immediately following function definition and specifies which function parameters are to be passed to or from the host (or both). In the example, the `inputs` array and `count` are written by the host and the `results` array is read by the host.

The pragma can also provide more information such as how the accelerator code is to be distributed to CSX processors. See the section [RPC pragma on page 145](#) for more details.

The RPC pragma causes the compiler to generate a source file with the C++ interfaces to the **Cⁿ** function. This has the same prototype as the **Cⁿ** function, with parameters changed

to vectors where required. It uses CSPX functions (as described in the following chapters) to move the arguments to and from the accelerators. The compiler also generates a C++ header file with a prototype for this function and the structure used by the wrapper function to pass the parameters to and from the accelerators.

By default, the generated files are named after the **C**ⁿ source file with `_interface` appended. In this example, the files generated from `sqrt_rpc.cn` will be `sqrt_rpc_interface.h` and `sqrt_rpc_interface.cpp`.

Handling calls from the host

The `main` function of the **C**ⁿ program must call the `CSPX_runtime` function. This handles calls from the host program, passing control to the appropriate **C**ⁿ function.

Compiling the **C**ⁿ code

When compiling **C**ⁿ source code containing the `cspcx_rpc` pragma, the interface files for the host code will be generated automatically. The code will also need to be linked with the CSPX library (specified on the command line as `-lcspcx`).

You will need to use the following `cscn` command line options:

- `-cspcx-csx-file filename` this is used to specify the name of the final CSX executable. This needs to be included in the generated C++ source code.
- `-cspcx-filename filename` this option specifies the name to be used for the generated interface files.

These options are required because the compilation process generates intermediate files with unpredictable names. Therefore the compiler needs to be explicitly given the names for the generated files.

These options are not handled by `cscn` program and so must be passed through to the **C**ⁿ compiler with the `-Wcn` option. For example the following command line⁽¹⁾ could be used to compile the `sqrt_rpc.cn` program:

```
cscn -Wcn, "-cspcx-csx-file sqrt_rpc", "-cspcx-filename ↵
sqrt_rpc_interface" -lcspcx -o sqrt_rpc.csx sqrt_rpc.cn
```

You can look at the makefiles included in the CSPX examples to see how these are compiled.

If there are multiple source files to be compiled, then they can all be included in the same `cscn` command. If multiple files define functions exported with the RPC pragma then the generated interface header and C++ files will include the information for all the functions.

If `cncc` is used to compile the source files, then they will have to be compiled individually. The compiler *appends* the generated code to the specified interface files so the same files can be used for all the RPC functions defined in the source files.

Note: *Because the compiler appends to the interface files, it is important to delete old or stale versions of these files before compiling.*

1. The symbol ↵ indicates that a single command has had to be split over more than one line in this document. The complete command should be entered on one line.

1.3.2 Host code

On the host side, the function can be called as if it were defined on the host itself as shown in [Figure 3](#). This includes the generated header file `sqrt_rpc_interface.h`.

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

// include the header files generated by the Cn compiler
#include "sqrt_rpc_interface.h"

#define NUM_PES          96
#define PROBLEM_SIZE    (NUM_PES * 1)

int main(void) {

    // An array of input values
    double inputs[PROBLEM_SIZE];

    // An array of result values
    double results[PROBLEM_SIZE];

    // initialize input values
    for (int i = 0; i < PROBLEM_SIZE; i++) {
        inputs[i] = i*i;
    }

    // call function on accelerator
    square_root(inputs,  PROBLEM_SIZE,
                results, PROBLEM_SIZE,
                PROBLEM_SIZE / NUM_PES);

    // print out results
    for (int i=0; i < PROBLEM_SIZE; i++) {
        printf("%d: %.2f\n", i, results[i]);
    }
    return 0;
}
```

Figure 3. Host code to call an accelerated function: `sqrt_rpc.c`

Compiling the host code

The host code has to be compiled and linked with the `sqrt_rpc_interface.cpp` file generated from the **Cⁿ** compiler.

When the host code is run, it will connect to as many processor cores as it can find, load the compiled CSX code on to them, and then call the function and wait for it to return.

1.3.3 Running on a simulator

It is possible to run the CSX code on a simulator if hardware is not available. See [Section 7.5: Running code on a simulator on page 61](#) for details.

1.3.4 Handling errors

The interface to the accelerator is written in C++. This may throw an exception if the program is unable to connect to the accelerator, or if the compiled **C** cannot be found. These errors can be caught by adding a try-catch block to the code. An example is shown in [Figure 4](#) which simply adds exception handling around the call of the square root function. If an exception is thrown, then the error message is displayed and the program exits.

```
#include <CSPXException.h>

...

    try {
        // call function on accelerator
        square_root(inputs, PROBLEM_SIZE,
                    results, PROBLEM_SIZE,
                    PROBLEM_SIZE / NUM_PES);
    } catch (CSPX::Exception &e) {
        printf("%s\n", e.what());
        exit (1);
    }
```

Figure 4. Adding error handling to an RPC call

2 Basic process and object model

This chapter describes the CSPX process and object model which underlies the RPC mechanism. The explanation in this section uses the C++ library for the host functions. A C language interface is also available, see [Chapter 3: C interface to processes and objects on page 33](#).

The RPC mechanism described in the previous chapter is useful for simple programs. However, function calls are synchronous (blocking) and data can only be passed as the function parameters. The data is sent and returned on every function call which can lead to more communication than necessary in some cases. For example, you may wish to use a 'retained mode' where some data is sent to the accelerator once, for use by several subsequent function calls.

2.1 Process and communication model

The CSPX programming model is based on two basic concepts:

- a model of accelerator processes supporting remote function calls
- communication via object migration

An application is decomposed into a number of processes. Each of these runs on either a host core or an accelerator, as shown in [Figure 5 on page 17](#). The processes running on the accelerators export one or more functions which can be called from the host processes. Typically, the application will launch processes on the available accelerators and divide the computation between them.

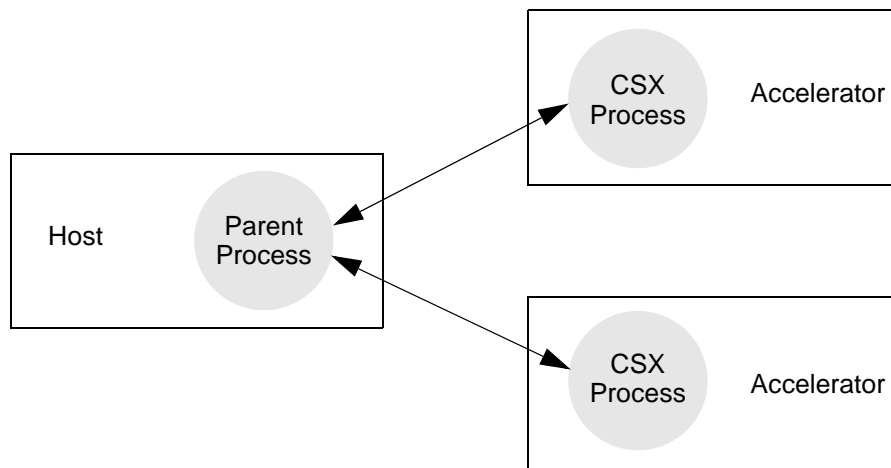


Figure 5. Process model

Communication between processes is supported using a concept of *objects* and *data migration*. Data to be shared between processes is represented using an object abstraction. CSPX objects provide a *handle* for data which is to be transferred between CSPX processes. Movement of data is performed by migrating these objects from one process to another using CSPX functions.

At any one time, an object can only be resident in one process. As well as copying data, object migration moves *ownership* of the object around.

- In [Figure 6](#), two objects are created in the host process.
- In [Figure 7](#), the objects are migrated to processes running on accelerators. This is done using a *move* function on the host to push the data to the CSPX processes. On the accelerators a *sync* function is used to synchronize execution with the availability of that data. Note that once they have been moved, the objects are no longer accessible by the host since objects exist uniquely in a single process.
- In [Figure 8](#), one of the objects is migrated back to the host. The contents of the object may have been modified or created by the accelerator process. The object is no longer accessible by the accelerator.
- In [Figure 9](#), the same object is migrated to another accelerator, showing how it is possible to move these objects to any process in the hierarchy.

 Object

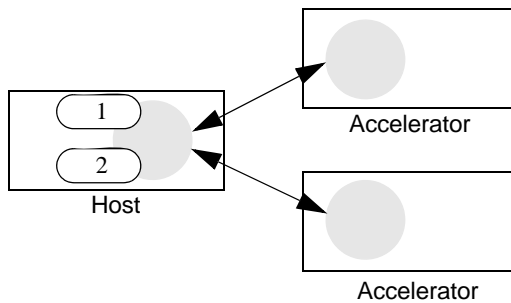


Figure 6. Object creation

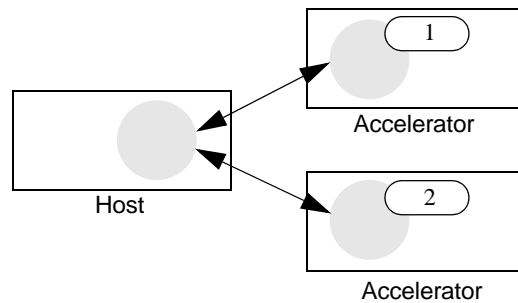


Figure 7. Objects sent to accelerators

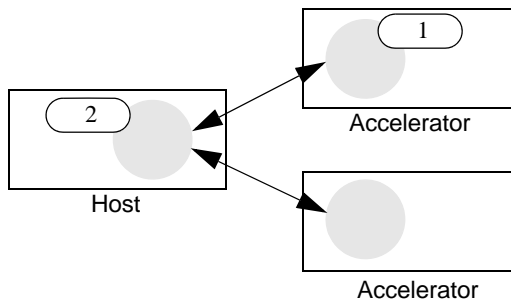


Figure 8. Object sent back to host

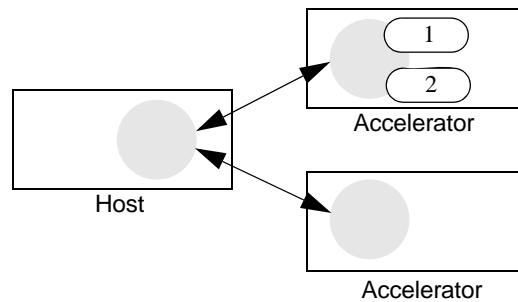


Figure 9. Object sent to other accelerator

This basic process and object model is used to build all the more sophisticated programming models in CSPX. As far as possible, the API is common to the host and accelerator processes. For example, the same functions are used by both host and accelerator code for moving data between them.

Note: In the current version of CSPX objects can only be moved between the host (parent) process and an accelerator, not between accelerators.

2.2 CSPX processes

A process in the standard operating system sense consists of a private address space and a set of threads which execute within this address space. CSPX provides the same model for making use of hardware accelerators such as the ClearSpeed Advance card.

CSPX support a *process group* model to support the single-program, multiple-data (SPMD) model of parallel computation where the processors run the same code to process pieces of the problem in parallel. In the C++ interface, single processes and groups are handled identically. *Policies* define how many processors are used and how the data is assigned to them.

A CSPX process is created by loading and running a **Cⁿ** program on a CSX processor. A host program can create one or more CSPX processes each running on a separate accelerator. Once the host process has created these CSPX processes it can make calls to functions exported by them.

2.2.1 Calling functions from the host

The fundamental classes provided by the C++ interface are `CSPX::State<>` and `CSPX::Object<>` which correspond to the process and object model, respectively.

These are defined as template classes rather than simple classes. This allows a great deal of flexibility through the use of policies. Policies are classes that can be used when an object of the `CSPX::State` or `CSPX::Object` class is declared to modify its behavior in various ways. Policies are described in more detail in [Section 2.6: Policy templates on page 26](#).

2.2.2 Connecting to an accelerator

A host process creates one or more CSPX processes by instantiating a `CSPX::State` object.

The `CSPX::State` class encapsulates the state associated with the connection to an accelerator (or accelerators). An instantiated `CSPX::State` object represents the connection to a group of accelerators and the program loaded onto them.

The simplest way to create such an object is as follows:

```
CSPX::State<> mystate( "compute.csx" );
```

This will initialize a connection to one accelerator core and load the file `compute.csx` onto that processor.

The `CSPX::State` class also provides member functions for the main CSPX operations. For example, `call` is used to call an accelerator function (remote function call) passing an object as the parameter to the function.

These member functions will be described in more detail below.

2.2.3 Running on a simulator

It is possible to run the CSX code on a simulator if hardware is not available. See [Section 7.5: Running code on a simulator on page 61](#) for details.

2.3 Object communication model

As well as a way of calling accelerated functions in processes running on ClearSpeed devices, an application requires a method of passing data efficiently between the host process and the accelerated processes. With CSPX this is provided via the object movement or migration model. The abstraction provided is that of an object (with an associated ‘lump’ of data) that is uniquely identified across all the host and CSPX processes.

Note: In the current implementation, only the host can create and attach data to objects.

There are two basic operations on objects:

1. **Move:** copies the object and associated data from the current process to another. This is asynchronous (the function returns immediately) and only allowed if the object is already resident in the process. Communication is only supported from the host, or parent, process to an accelerator and back again.
2. **Sync:** waits for the object to ‘arrive’ in the current process. This function blocks until the data is available.

Note: The move and sync operations on an object must be matched: there must be just one sync operation corresponding to each move.

2.3.1 Objects

The `CSPX::Object` template class corresponds to the basic CSPX object abstraction. The fundamental difference is that `CSPX::Object` inherently represents a group of objects (which may just be a single object).

The simplest way to create a `CSPX::Object` is as shown below:

```
CSPX::Object<double> my_object(buffer, size, numProcesses, CSPX_OBJECT_WRITE);
```

where `buffer` is a pointer to the buffer being used, `size` is the size of the buffer (number of elements, not number of bytes) and `numProcesses` is the number of target processes. The number of process can be obtained from the relevant `CSPX::State` object.

Note that the base type of the buffer being attached to the object is provided as an argument to the template class (this is `double` in this example).

The `CSPX::Object` class also provides a number of member functions for iterating over a group of objects. These are equivalent to the iterators provided in the C++ Standard Template Library.

- `CSPX::Object<basetype>::iterator` – the iterator type that is defined by the `CSPX::Object` template class
- `begin` – returns the start iterator for the group of objects
- `end` – returns the end iterator for the group of objects
- `dereference` (for example `*p`) – returns the underlying object contained in the `CSPX::Object` group

2.3.2 Communicating with an accelerator

Data to be moved to an accelerator must first be *attached* to a CSPX object. This object, with the associated data, is then moved to the accelerated process for computation. The host process can wait for the data to return after computation by synchronizing on the object. This is shown in [Figure 10](#). Here, an array of result values is created on the host and

passed to the accelerator. Note that no data is passed at this point: *ownership* of the result object is transferred to the accelerator and space for the array is allocated.

```
double results[PROBLEM_SIZE];
// create processes and communication objects
CSPX::State<> processes("cn_function.csx");
int numProcesses = processes.numProcesses();
CSPX::Object<double> resultObj(results, PROBLEM_SIZE, numProcesses, CSPX_OBJECT_READ);

processes.move(resultObj);          // move object to accelerator (no data copied)
processes.call(params, "cn_function");
processes.sync(resultObj);         // wait for the data to be returned
```

Figure 10. Sending data to an accelerator

On the accelerator, computation takes place and then the result object (and data) can be moved back to the 'parent' process, at this point the data is copied to the host. This is shown in [Figure 11](#).

```
CSPXObject results;
float *result_buffer;

// Wait for object and get address of data in this process
result_buffer = CSPX_object_sync_pointer(&results);

... // process

// Move object back to host process
CSPX_object_move(parent, results);
```

Figure 11. Operating on data in an accelerator

In a real application, one or more objects are likely to be used to send input data to the accelerator. The CSPX primitives allow this to be done in a very flexible way as shown in the code fragment in [Figure 12](#).

```
... // declare kernel, samples, results as CSPX objects

... // initialize data objects

process.move(results);          // pass result object over to accelerator
process.move(kernel);          // send first set of input data
for (i = 0; i < SAMPLES; i++) {
    process.move(p, samples);          // send data for each iteration
    process.call(p, "convolve");      // call one or more functions on accelerator
    process.call(p, "reduce");
}
process.sync(&results);        // wait for the final results to be returned
```

Figure 12. Processing data on an accelerator

2.3.3 Shared objects

For the object model to work, the host and accelerator must be able to refer to the same object, ideally by the same name. There are two ways of achieving this:

Join: objects declared on the host and accelerator can be explicitly joined. After a join, object variables on the host and accelerator will refer to the same object.

Parameter passing: objects can be passed as parameters to function calls. This provides the accelerator with a handle or reference to the object declared on the host.

These two methods are equivalent, you can choose whichever method fits best with your application. Most of the examples in this document pass the objects as parameters to function calls.

2.3.4 Data alignment

Data is normally transferred to and from the accelerators using Direct Memory Access (DMA) hardware. This imposes some constraints on the alignment of data. To ensure that data attached to an object is correctly aligned, there are various methods which can be used. You can choose whichever is more appropriate to your application. A couple of possibilities are:

1. Instead of declaring the data as an array of values, use `malloc` to allocate the data on the heap. This is guaranteed to be correctly aligned.
2. Your compiler may have directives for forcing the alignment of data. For example, using `gcc`, the alignment of a variable can be specified with the following syntax:

```
int x __attribute__((aligned (8)));
```

Similarly, with the Microsoft Visual C++ compiler, a declaration such as the following can be used:

```
__declspec(align(8)) int x;
```

Both of these examples force the data to be aligned to an 8-byte boundary.

2.3.5 Object attributes

In order to make data migration more efficient, objects can be declared as `READ`, `WRITE` or `READ_WRITE` with respect to the parent process. In the example shown, when the data is attached to the `CSPXObject` it is tagged as `READ_WRITE`. A `READ` object will only copy the attached data when the object is moved from a child process to the parent process (a read operation). Conversely a `WRITE` object will only copy the data when the object is moved from the parent process to the child. A `READ_WRITE` object will always copy data in both directions. When not copying data (for instance when writing a `READ` object from the parent process to a child process), the object is synchronized to allow the destination process to know that it 'owns' the object and can begin to operate on it.

2.4 Accelerator code

The **C** code for the accelerators provides a number of externally-callable functions and a dispatch function to handle calls to these from the host process.

2.4.1 Exporting functions

An exported function must conform to the following function prototype:

```
int f(CSPXProcess parent, void* const param);
```

An exported function has two parameters:

`CSPXProcess` – a handle to the calling (host) process

`void* param` – a pointer to the arguments to be passed to the function. This may be omitted if not required.

A function is marked as remotely callable using the `csp_x_export` pragma. For example, [Figure 13](#) shows the declaration of a function which can be called from the host.

```
#pragma csp_x_export(cn_function)

int cn_function (CSPXProcess parent, void* const params);
```

Figure 13. Exporting a C⁺⁺ function

The `params` parameter is typically a structure containing the parameters which would be passed to the function if called locally. This may contain any data types not involving pointers. This can include CSPX objects used to transfer larger amounts of data between the host and the accelerator.

Note that the data attached to any objects passed as parameters is not transferred automatically on a function call. The object passed is a handle to the data which must be explicitly transferred using `move` and `sync` functions, see [Section 2.3: Object communication model on page 20](#).

2.5 Invoking a function on the accelerator

The call member function for invoking a function on the accelerator card has already been mentioned as part of the `CSPX::State` description. To simplify passing parameters to a function called on the group of processes in the `CSPX::State` object, a special `CSPX::Parameter` class is provided.

2.5.1 Passing parameters

In this example, the program passes two objects and an integer to the function running on the accelerator. These parameters are encapsulated in a structure which is used by the accelerator process, `params` ([Figure 14](#)).

```
// A simple structure containing the parameters passed to the function
struct params {
    CSPXObject result; // read only
    CSPXObject input; // write only
    int count;         // implicit read/write
};
```

Figure 14. Structure for function parameters in C⁺⁺

It would be possible to create a `CSPX::Object`, attach an instance of the `params` structure and pass this to the called function. However, this can get complex when there is more than one accelerator process. The `CSPX::Parameter` class simplifies the task of passing parameters.

Consider the code in [Figure 15](#). Here each of the parameters is 'pushed' onto the `CSPX::Parameter` in the order they are defined in the structure. (Note that `results_obj` and `inputs_obj` are previously defined `CSPX::Object` objects.)

```

CSPX::Parameter params_obj(mystate.numProcesses());

// Corresponds to result field in cppwrapper_params
params_obj.push(results_obj);
// Corresponds to input field in cppwrapper_params
params_obj.push(inputs_obj);
// Corresponds to count field
params_obj.push(20 / mystate.numProcesses());

// move objects to accelerator
mystate.move(inputs_obj);
mystate.move(results_obj);

// call accelerator function passing parameter object
std::vector<int> call_returns = mystate.call(params_obj, "cn_function");

// wait for result object to arrive back
mystate.sync(results_obj);

```

Figure 15. Function call using CSPX::Parameter

The types of the parameters can be:

1. An object: in this case the partition policy of the object will determine how it is distributed to the processes
2. A scalar value: this will be broadcast to all processes
3. An STL vector of values: these will be distributed to the processes, one element per process.

2.5.2 Dealing with return values

There are two types of return value which need to be dealt with in the CSPX C++ interface. The first is the return value from a function call. This returns a vector of `int` values with a value corresponding to each process in the group. This value is the return value of the function called on each accelerator.

Function parameters

Values can also be returned in function parameters. For `CSPX::Object` parameters which are marked as read or read/write, then the partition policy will take care of how the data returned from each process is combined within the object.

Other parameter types are always tagged as being read/write. In this case the results from each process will need to be retrieved. For example, see the code in [Figure 17](#). In this example the parameters contain an input object (`inputs`) which is write only, and a result value (`res`). However, in the case where multiple accelerator processors are being used there will also be multiple result values. This requires a mechanism for returning or combining multiple values.

As shown in [Figure 17](#), when the `res` variable is pushed onto the `CSPX::Parameter`, a *handle* is returned. Once the call to the accelerator function has terminated a call is made to the `getValue` to return the vector of values returned for the parameter. At this point you could traverse the vector and process the values as required (for example, sum them).

Reduction functions

The `CSPX::Parameter` class provides a way to handle reductions automatically as demonstrated [Figure 18](#).

```
// structure used by the Cn code to represent parameters passed to
// an exported function

struct cppsum_params {
    CSPXObject input; // write only
    double result;    // read/write
};
```

Figure 16. Parameter definitions in C⁺⁺

```
CSPX::State<MaximumDistributionPolicy> mystate(csx_file);
CSPX::Object<double> inputs(a, N, mystate.numProcesses(), CSPX_OBJECT_WRITE);
CSPX::Parameter params(mystate.numProcesses());
double res;

params.push(inputs);
unsigned int handle = params.push(res);

mystate.move(inputs);
std::vector<int> returns = mystate.call(params, "csx_calculate_sum");

std::vector<double> results = params.getValue(handle);
```

Figure 17. Returning a parameter value

```
CSPX::State<MaximumDistributionPolicy> mystate(csx_file);
CSPX::Object<double> inputs(a, N, mystate.numProcesses(), CSPX_OBJECT_WRITE);
CSPX::Parameter params(mystate.numProcesses());
double res;

params.push(inputs);
unsigned int handle = params.push(res, &CSPX_double_sum);

mystate.move(inputs);
std::vector<int> call_returns = mystate.call(params, "csx_calculate_sum");

params.getValueReduce(res, handle);
```

Figure 18. Using reduction functions

In this example, when the result parameter value is pushed onto the `CSPX::Parameter` an additional function pointer is provided. The function pointer refers to a function that will be used to reduce the values returned by the function call. In this case a predefined function is used which returns the sum of two double arguments.

After the function call on the accelerator returns the function `getValueReduce` is called to invoke the specified reduction function on the values returned. This reduces the vector to a single value returned in the `res` variable.

There are a number of reduction functions defined in the C++ interface. You can easily augment these as the format is simple. The `CSPX_double_sum` function definition is shown in [Figure 19](#).

```

void *CSPX_double_sum(void * op1, void *op2) {
    double *d_op1 = (double *) op1, *d_op2 = (double *) op2;

    *d_op1 = *d_op1 + *d_op2;

    return (void *) d_op1;
}

```

Figure 19. Typical reduction function

2.6 Policy templates

2.6.1 Distribution and connection policies

There are default policies that are used by the `CSPX::State` template class. The declaration above ([Section 2.2.2: Connecting to an accelerator on page 19](#)) could also be written as follows, where the default policies are specified explicitly:

```

CSPX::State<CSPX::DefaultDistributionPolicy, CSPX::DefaultConnectionPolicy>
    mystate("compute.csx");

```

The `CSPX::DefaultDistributionPolicy` specifies that the `CSPX::State` object should use only one instance of an accelerator core.

The `CSPX::DefaultConnectionPolicy` specifies that `CSPX::State` should connect only to hardware (as opposed to a simulator).

The implementation provides alternative policies that can be used, defined as follows:

```

CSPX::MaximumDistributionPolicy – connect to as many accelerators as
available
CSPX::SimulatorConnectionPolicy – connect only to simulators

```

Note: When using the simulator the runtime system cannot determine the number of simulator instances that are running. This means that the maximum connection policy will only connect to a single simulator. When using the simulator, you should use the default distribution policy or a custom policy that connects to the number of simulator instances you are using.

2.6.2 Partition policies for objects

The `CSPX::Object` has a default policy which controls how the buffer will be distributed when there are more than one target processes. The previous object declaration ([Section 2.3.1: Objects on page 20](#)) could also explicitly specify the default policy:

```

CSPX::Object<double, CSPX::DefaultPartitionPolicy> my_object(buffer, size,
    numProcesses,
    CSPX_OBJECT_WRITE);

```

The `DefaultPartitionPolicy` will chop up the buffer into equally sized pieces to be distributed to each of the processes. As with the `CSPX::State` policies, this policy can be redefined. The default policy implementation is shown in [Figure 20](#).

A partition policy must provide the following member functions:

nextPartition: Return a pointer to the buffer for the processes specified by the index. In the default partition policy it returns the start pointer + ordinal * size of each

```

template <class T> class DefaultPartitionPolicy {
public:
    // Partition policy must implement a function that can partition a chunk of
    // memory pointing to a base type into the required number of pieces
    T* nextPartition(T* base, unsigned int total_size, unsigned int num_processes,
                    unsigned int ordinal) {
        // Default behaviour is very simple - chop up into equal sized pieces
        // Make sure that the chunks can be of equal size
        if (total_size % num_processes != 0) {
            std::cerr << "Warning : unable to partition object amongst processes evenly."
                << " Total size " << total_size << " number of processes "
                << num_processes << std::endl;
        }
        return base + (( total_size / num_processes ) * ordinal);
    }

    unsigned int partitionSize(size_t total_size, unsigned int num_processes,
                               unsigned int ordinal) {
        // Chop into equal sized pieces
        // In this case, the last chunk will fill out the surplus
        if (total_size % num_processes != 0) {
            std::cerr << "Warning : unable to partition object amongst processes evenly."
                << " Total size " << total_size << " number of processes "
                << num_processes << std::endl;
        }

        unsigned int partition_size = total_size / num_processes;
        if (ordinal == (num_processes - 1)) {
            // This is the last chunk
            return (partition_size + (total_size - (partition_size * num_processes))) *
                sizeof(T);
        }
        return partition_size * sizeof(T);
    }
};

```

Figure 20. Default partition policy

chunk. You could implement some much more sophisticated mechanisms, for instance serialization of a complex class hierarchy.

partitionSize: Return the size of the buffer for the process specified by the index. The default partition policy returns the total size / number of processes (except when the buffer cannot be divided equally and so the last chunk is a different size from the rest).

By defining a new distribution policy you can implement sophisticated ways of distributing a problem across the available accelerators. For example, if you wish to broadcast some seed values to all accelerators which are different for each process: you can create a structure containing the required information and then create a `CSPX::Object` based on this structure. You can then write a custom distribution policy which modifies this structure for each process according to the index. This means that this behavior can be encapsulated in one place, in the policy for the object.

2.6.3 Custom policies

Additionally, you can provide your own implementations of policies to define how a problem is distributed across a number of accelerators, and how the host should connect to them. For example, it would be simple to implement a distribution policy that allowed the program to connect to up to n cards but no more. In the same way, you could implement a new connection policy that attempted to connect to hardware or, if it were not available, to connect to a simulator instead.

To implement your own policies you need to define a new class that implements the same interface as the standard policy classes.

For example, the source code for the default distribution policies is shown in [Figure 21](#) to [Figure 23](#) (this is also available in the file `CSPXDistributionPolicy.h` in the software installation). We can see from this source code that there is a base class (`CSPX::DistributionPolicyBase`) that all distribution policies inherit from.

```
struct DistributionPolicyBase {
    static unsigned int availableProcessors(struct CSAPIState* state) {
        // Return the actual number available for the given state
        unsigned int n_chips_available;
        CSAPIErrno status;
        status = debug_try( CSAPI_num_processors( state, &n_chips_available));
        return n_chips_available;
    }
};
```

Figure 21. Distribution policy base class

```
// Default distribution policy is to run code on one process only
struct DefaultDistributionPolicy : public DistributionPolicyBase {
    static unsigned int numBoards() {
        // Default behaviour is to return 1 process
        return 1;
    }

    static unsigned int numProcessors() {
        // Default behaviour is to return 1 process
        return 1;
    }
};
```

Figure 22. Default distribution policy

A distribution policy must provide three member functions:

availableProcessors: return the number of processor cores that the CSAPI state is connected to.

numBoards: return the number of cards to connect to. (For the default case this is limited to 1, for the maximum case this is the number of cards that are available in the system).

numProcessors: return the number of processor cores to connect to on each card.

During initialization, the `CSPX::State` object will invoke these functions in order to connect to hardware or simulators as directed by the policies.

```

// Use the maximum number of accelerators available in the machine that this host
// code is running on
struct MaximumDistributionPolicy : public DistributionPolicyBase {
    static unsigned int numBoards() {
        CSAPIErrno status;
        unsigned int n_boards_in_system = 0;

        status = debug_try( CSAPI_num_cards ( &n_boards_in_system ));
        return n_boards_in_system;
    }

    static unsigned int numProcessors() {
        // Expect to find 2 cores per board for this policy
        return 2;
    }
};

```

Figure 23. Maximum distribution policy

2.7 C++ example

We can now take the square root example previously used in the RPC chapter ([Section 1.2: Remote procedure call on page 9](#)) and show how this can be implemented using the underlying CSPX primitives.

2.7.1 Function arguments

To use the CSPX remote function call, a structure is defined to hold the function arguments. This is defined in a header file which is used by the C⁺⁺ code and, in later examples, the host code. The structure definition is shown in [Figure 24](#).

```

struct sqrt_args {
    CSPXObject inputs;
    CSPXObject results;
    int count;
};

```

Figure 24. Structure for function arguments: `sqrt.h`

2.7.2 Accelerator code

The code for the accelerator is very similar to the RPC example; the relevant function definition is shown in [Figure 25](#). Again, the object arguments and the attached data needs to be managed explicitly. The program starts by waiting for the data associated with objects to arrive. It then performs the same processing as the RPC example. Finally the result array is moved back to the host.

2.7.3 Host code

The main function for this version of the code is shown in [Figure 26 on page 31](#). This is more complex than the RPC example as it has to explicitly manage creation of the CSPX process on an accelerator and the movement of data to and from the process.

```

#include "sqrt.h"

// export the function for remote calling
#pragma cspix_export(square_root)

int square_root(CSPXProcess parent, void* const params) {
    struct sqrt_args *args = (struct sqrt_args *) params;
    double *inputs, *results;
    int i;

    // wait for the data to arrive from the host
    CSPX_object_sync(&args->inputs);
    CSPX_object_sync(&args->results);
    // get a pointer to the data attached to the objects
    inputs = CSPX_object_get_pointer(&args->inputs);
    results = CSPX_object_get_pointer(&args->results);

    for (i = 0; i < args->count; i += __NUM_PES__) {
        poly double d;
        poly int index;

        // calculate address of data to operate on
        index = i + get_penum();
        // copy input from mono memory
        memcpym2p(&d, &inputs[index], sizeof(double));
        // perform calculation
        d = sqrt(d);
        // write result back to mono memory
        memcpym2m(&results[index], &d, sizeof(double));
    }
    // return results back to host
    CSPX_object_move(parent, &args->results);
    // pass input object back to the host so it can be used again
    CSPX_object_move(parent, &args->inputs);

    return 0;
}

```

Figure 25. Accelerator code: sqrt.cn

In the C++ interface, we do not use the argument structure directly in the host code. Instead, the parameters are added to a parameter object in the same order they appear in the structure.

When the function is called, the parameter object is automatically copied across to the accelerator. However, the data attached to the objects must be explicitly moved to and from the accelerator. Note that only the data attached to the `inputs` object is actually copied to the accelerator (as it is marked with the 'write' attribute) and only the result data is copied from the accelerator.

By changing the declaration of the `CSPX::State` object to use the a different policy to distribution policy (for example, `CSPX::MaximumDistributionPolicy`) the same code can be used with any number of accelerators.

The array arguments to each process are automatically divided up among the accelerators used. This is handled by the partition policy used. In this case, we use the default which is simply to provide an equal part of the array to each process.

```
#include <CSPXState.h>
#include <CSPXObject.h>
#include <CSPXParameter.h>

#define NUM_PES          96
#define PROBLEM_SIZE    (NUM_PES*2)

int main(void) {

    // arrays of input and result values
    static double inputs[PROBLEM_SIZE], results[PROBLEM_SIZE];

    // object for the accelerator processes
    CSPX::State<> processes("sqrt.csx");
    int numProcesses = processes.numProcesses();

    // create objects to transfer the input and result arrays
    CSPX::Object<double> inputObj(inputs, PROBLEM_SIZE, numProcesses, CSPX_OBJECT_WRITE);
    CSPX::Object<double> resultObj(results, PROBLEM_SIZE, numProcesses, CSPX_OBJECT_READ);

    // Initialize the function arguments
    CSPX::Parameter params(processes.numProcesses());

    // add arguments to parameter object in the same order
    // that they are defined in the structure used in Cn (sqrt.h)
    params.push(inputObj);
    params.push(resultObj);
    params.push(PROBLEM_SIZE);

    // initialize input values
    for (int i = 0; i < PROBLEM_SIZE; i++) {
        inputs[i] = i*i;
    }

    // move the objects to the accelerator
    processes.move(inputObj);
    processes.move(resultObj);

    // call the function on the accelerator
    processes.call(params, "square_root");

    // and wait for the results to return
    processes.sync(resultObj);

    // print out results */
    for (int i=0; i < PROBLEM_SIZE; i++) {
        printf("%d: %.2f\n", i, results[i]);
    }
    return 0;
}
```

Figure 26. C++ host code

3 C interface to processes and objects

The C language interface to CSPX provides a number of types and functions to represent the processes and objects described in [Section 2.1: Process and communication model on page 17](#).

3.1 Creating processes

A host process creates a CSPX process by calling the function `CSPX_process_load` with the handle of an accelerator, created using CSAPI, and the name of a CSX program.

The initialization of an accelerator and creation of a CSPX process are shown in [Figure 27](#).

```
#include <csapi.h>
#include <cspix.h>

struct CSAPIState* card_handle;
CSPXProcess the_process;
char * csx_file = "compute_function.csx"; /* path to executable csx file */
CSAPIErrno err;

/* call CSAPI function to initialize connection to card */
card_handle = CSAPI_new();
err = CSAPI_connect(card_handle, CSH_Private, CSC_Direct,
                   NULL, CSAPI_INSTANCE_ANY, 0);

/* load the program onto the card and create a cspix process */
the_process = Process_load(card_handle, 0, csx_file, NULL, 0, NULL);
```

Figure 27. CSPX process initialization

The `CSPX_process_load` function returns a `CSPXProcess` handle. The parent process can then call any of the functions exported by the CSX program using this process handle. An example of calling an exported function is shown in [Figure 28](#).

```
#include <cspix.h>

CSPXErrno call_error_code;
int result;
struct my_parameters parameters;

call_error_code = CSPX_process_call(the_process,
                                   "cn_function",
                                   &parameters,
                                   sizeof(my_parameters),
                                   &result);
```

Figure 28. Calling an accelerator function through CSPX

The remote function call will behave exactly as if the function was called directly in the host code. This means that the call to `CSPX_process_call` will not return until the function in the CSX code returns.

3.1.1 Running on a simulator

It is possible to run the CSX code on a simulator if hardware is not available. See [Section 7.5: Running code on a simulator on page 61](#) for details.

3.2 Object communication model

As described in the previous chapter, there are a number of basic operations that must be provided for the object type:

- Data to be moved to an accelerator must first be *attached* to a CSPX object.
- A move operation copies the object and associated data from the current process to another. This is asynchronous (the function returns immediately) and only allowed if the object is already resident in the process. Communication is only supported from the host, or parent, process to an accelerator and back again.
- A sync function waits for the object to 'arrive' in the current process. This function blocks until the data is available.

As well as a way of calling accelerated functions in processes running on ClearSpeed devices, an application requires a method of passing data efficiently between the host process and the accelerated processes. With CSPX this is provided via the object movement or migration model. The abstraction provided is that of an object (with associated data) that is uniquely identified across the host and CSPX process.

3.2.1 Operations

There are two basic operations on objects:

1. Move: copies the object and associated data from the current process to another. This is asynchronous (the function returns immediately) and only allowed if the object is already resident in the process. Communication is only supported from the host, or parent, process to an accelerator and back again.
2. Sync: wait for the object to 'arrive' in the current process. This function blocks until the data is available.

Note: *The move and sync operations on an object must be matched: there must be just one sync operation corresponding to each move.*

Data to be moved to an accelerator must first be *attached* to a CSPX object.

An object can be created and moved, with the associated data, to the accelerator process for computation. The host process can wait for the data to return after computation by synchronizing on the object. This is shown in [Figure 29](#). Here, an array of result values is created on the host and passed to the accelerator. Note that no data is passed at this point:

ownership of the result object is transferred to the accelerator and space for the array is allocated.

```
CSPXObject result;
double result_array[SIZE_OF_DATA_ARRAY];

CSPX_object_new(&result);
CSPX_object_attach(&result, result_array, SIZE_OF_DATA_ARRAY * sizeof(double),
                  CSPX_OBJECT_READ);
CSPX_object_move(the_process, result); /* Move object to accelerator */
CSPX_process_call(process, "cn_function", ...);
CSPX_object_sync_pointer(&result); /* wait for the data to be returned */
```

Figure 29. Sending data to an accelerator

On the accelerator, computation takes place and then the result object (and data) can be moved back to the 'parent' process, at this point the data is copied to the host. This is shown in [Figure 30](#).

```
CSPXObject results;
float *result_buffer;

// Wait for object and get address of data in this process
result_buffer = CSPX_object_sync_pointer(&results);

... // process

// Move object back to host process
CSPX_object_move(parent, results);
```

Figure 30. Operating on data in an accelerator

In a real application, one or more objects are likely to be used to send input data to the accelerator. The CSPX primitives allow this to be done in a very flexible way as shown in the code fragment in [Figure 31](#).

```
CSPXObject kernel, samples, results;

... /* initialize data objects */

/* pass result object over to accelerator */
CSPX_object_move(p, results);
/* send first set of input data */
CSPX_object_move(p, kernel);
for (i = 0; i < SAMPLES; i++) {
    /* send data for each iteration */
    CSPX_object_move(p, samples);
    /* call one or more accelerator functions */
    CSPX_process_call(p, "convolve", ...
    CSPX_process_call(p, "reduce", ...
}
/* wait for the results to be returned */
CSPX_object_sync(&results);
```

Figure 31. Processing data on an accelerator

3.2.2 Common namespace

For the object model to work, the host and accelerator must be able to refer to the same object, ideally by the same name. There are two ways of achieving this:

Join: objects declared on the host and accelerator can be explicitly joined. After a join, object variables declared with the same name on the host and accelerator will refer to the same object.

Parameter passing: objects can be passed as parameters to function calls. This provides the accelerator with a handle or reference to the object declared on the host.

These two methods are equivalent, you can choose whichever method fits best with your application. Most of the examples in this document pass the objects as parameters to function calls.

3.3 C example

We can now take the square root example used ([Section 1.2: Remote procedure call on page 9](#)) and show how this can be implemented using the underlying CSPX primitives.

3.3.1 Function arguments

A structure is used to hold the function arguments. This is defined in a header file which will be used by both the host code and the **C** code. The structure definition is shown in [Figure 14 on page 23](#).

3.3.2 Host code

The host code which calls the function is shown in [Figure 32](#). This is more complex than the RPC example as it has to explicitly manage creation of the CSPX process on an accelerator (using the function `CSPX_process_create`) and the movement of data to and from the process.

When the function is called, the argument structure is copied across to the accelerator. However, the data attached to the objects must be explicitly moved to and from the accelerator. Note that only the data attached to the `inputs` object is copied to the accelerator (as it is marked with the 'write' attribute) and only the result data is copied from the accelerator.

This code could be extended to use multiple accelerators by creating a process for each one. The data to be processed would then need to be divided up and attached to objects which can be moved to each process. However, it is much simpler to use the process group model described below.

3.3.3 Accelerator code

The code for the accelerator is identical to that used before, see [Figure 25 on page 30](#). Again, the object arguments and the attached data needs to be managed explicitly. The program starts by waiting for the data associated with objects to arrive. It then performs the same processing as the RPC example. Finally the result array is moved back to the host.

```
#include <stdio.h>
#include <cspix.h>
#include "sqrt.h"

#define NUM_PES      96
#define PROBLEM_SIZE (NUM_PES * 2)

int main(void) {
    int i, res;
    CSPXErrno err;

    /* handle for the accelerator process */
    CSPXProcess the_process;

    /* Arrays of input and result values */
    double inputs[PROBLEM_SIZE], results[PROBLEM_SIZE];

    /* Structure for passing arguments */
    struct sqrt_args args;

    the_process = CSPX_process_create("sqrt.csx", 0, NULL, &err);

    /* create objects to transfer the input and result arrays */
    CSPX_object_new(&args.inputs);
    CSPX_object_attach(&args.inputs, inputs, PROBLEM_SIZE*sizeof(double),
                      CSPX_OBJ_WRITE);
    CSPX_object_new(&args.results);
    CSPX_object_attach(&args.results, results, PROBLEM_SIZE*sizeof(double),
                      CSPX_OBJ_READ);

    /* Initialize the function arguments */
    args.count = PROBLEM_SIZE;

    /* initialize input values */
    for (i = 0; i < PROBLEM_SIZE; i++)
        inputs[i] = i*i;

    /* Move the argument objects to the accelerator */
    CSPX_object_move(the_process, &args.inputs);
    CSPX_object_move(the_process, &args.results);

    /* Now call the function on the acclerator */
    CSPX_process_call(the_process, "square_root", &args, sizeof(args), &res);

    /* wait for the results object to be returned */
    CSPX_object_sync(&args.results);

    /* print out results */
    for (i=0; i < PROBLEM_SIZE; i++)
        printf("%d: %.2f\n", i, results[i]);
}
```

Figure 32. Calling accelerator function: sqrt.c

3.4 Process group interface

The C++ interface handles single processes and groups of processes in the same way. In the C interface, process groups need to be managed explicitly. Instances of the process group type (`CSPXProcess_group`) are constructed by starting a CSX program on a collection of accelerators, for example ClearSpeed Advance cards, as shown in [Figure 33](#).

```
CSPXProcessGroup the_group;

/* Request all available processor cores */
the_group = CSPX_process_group_create(AllAvailableProcessors, "compute.csx",
                                     NULL, 0, NULL);
```

Figure 33. Creating a process group

When creating a process group, you can specify the number of accelerators or request all available processors. The function will set up all the processors so that a problem can be split across them.

You can now use these accelerators as a ‘gang’ of processes (like the gang of threads in OpenMP). This is supported in CSPX by extending the basic remote function call mechanism to a process group. A function can be called across a group so each member of the group will compute the same function but on a separate portion of the problem data. The host can then combine (reduce) these partial results into a complete result.

The process group model includes an object for each process in the group to represent the arguments passed to that process. When a function is called on the process group, each function is passed the values from the corresponding argument object.

Functions can be defined to process the data passed to and from each process in the group. For example, this can be used for ‘deep copies’ of a parameter block which contains CSPX objects.

In general, the process group and associated objects is the most useful abstraction provided. If a problem follows the SPMD pattern and can be broken up into a number of independent pieces that can be operated on separately then the process group can be used to model this distribution. For accelerator cards with more than one processor core, the process group also gives a simple method to manage these processors. Note that the group size is specified as the number of *processor cores* not the number of accelerator cards.

3.5 Example of process group use

We can now take the example shown in the previous chapter and adapt it to use process groups. This simplifies the tasking of using multiple accelerators.

3.5.1 Accelerator code

The code running on the accelerator is identical to the C++ example, see [Figure 25 on page 30](#).

3.5.2 Host code

The main function for this version of the code is shown in [Figure 34](#).

The parameter structure is unchanged, as shown in [Figure 24 on page 29](#). However, in this example, we need an array of structures for passing arguments to each process.

```

int main(void) {
    int i, cores, slice_size, res;

    /* handle for the accelerator process */
    CSPXProcessGroup ps;

    /* An array of input and result values */
    double inputs[PROBLEM_SIZE], results[PROBLEM_SIZE];

    /* Create the group of processes
    ps = CSPX_process_group_create(AllAvailableProcessors, "sqrt.csx", NULL, 0, NULL);

    cores = CSPX_process_group_get_size(ps);
    printf("Running on %d processors\n", cores);
    slice_size = PROBLEM_SIZE / cores;

    /* Array of structures for passing arguments */
    struct sqrt_args *args = (struct sqrt_args*) malloc(cores*sizeof(struct sqrt_args));

    /* set up the parameters for the process group */
    for (i = 0; i < cores; i++) {

        /* create objects to transfer the input and result arrays */
        CSPX_object_new(&args[i].inputs);
        CSPX_object_new(&args[i].results);

        /* assign part of the input and result arrays to each process */
        CSPX_object_attach(&args[i].inputs, inputs+(i*slice_size),
                           slice_size*sizeof(double), CSPX_OBJECT_WRITE);
        CSPX_object_attach(&args[i].results, results+(i*slice_size),
                           slice_size*sizeof(double), CSPX_OBJECT_READ);
        /* initialise the count parameter */
        args[i].count = slice_size;

        /* Now attach the args structure to the parameter object for this process */
        CSPX_object_attach(CSPX_get_parameter_box(ps, i), &args[i],
                           sizeof(struct sqrt_args), CSPX_OBJECT_READ_WRITE);
    }

    /* initialize input values */
    for (i = 0; i < PROBLEM_SIZE; i++) {
        inputs[i] = i*i;
    }

    /* Now call the function on the acclerators */
    CSPX_process_call_group(ps, "square_root", move_param_out, move_param_in);

    /* print out results */
    for (i=0; i < PROBLEM_SIZE; i++) {
        printf("%d: %.2f\n", i, results[i]);
    }
    return 0;
}

```

Figure 34. Calling a function group

The arguments to each process are defined by slicing up the input and result arrays so that each accelerator has a separate part of the problem to work on. The parameters for each process are set up in a loop. A different slice of the `inputs` and `results` is passed to each function.

The process group model includes an object associated with each process to represent the arguments for that process. The elements of the array of argument structures are attached to the corresponding parameter objects for each process in the group.

In the single process example, the objects passed as parameters had to be explicitly moved to the accelerator before the function call. When using process groups, this would be awkward so, instead, two functions are defined which are automatically called when the parameter structure is copied to and from each process, see [Figure 35](#). The `move_param_out` function moves the objects in the parameter structure to the target process. The corresponding `move_param_in` function simply synchronizes the object coming back after the function has completed.

```
/* Function invoked when arguments copied to each process in group */
int move_param_out(CSPXProcess p, void *params) {
    struct sqrt_args *args = (struct sqrt_args *) params;

    CSPX_object_move(p, &args->inputs);
    CSPX_object_move(p, &args->results);

    return 0;
}

/* Function invoked on return from function in group */
int move_param_in(CSPXProcess p, void *params) {
    struct sqrt_args *args = (struct sqrt_args *) params;

    /* wait for the results object to be returned */
    CSPX_object_sync(&args->results);

    return 0;
}
```

Figure 35. Functions to handle parameter objects

3.6 Extending the object model to the process group

The standard use of the process group abstraction is to create a group of identical clone processes performing the same function. This also implies that each member of the process group will have the same CSPX communication mechanisms with the host process; to support this, CSPX provides group versions of all the object communication methods.

The process group model also provides an object group type. An object group can be created from a process group and creates one object per process in the group. An iterator can be defined to process the object group which will apply a function to each object in the group. By default an object group is created for the parameters to a function call across the group.

The functions to set up and manage the parameters for a process group function call are shown below:

CSPX_object_group_create: Create an object group for a process group, one object per member of the group.

CSPX_object_group_apply_function: Apply a function to each of the object members in the group.

4 Double-buffered communication

CSPX provides a *double-buffered* communication object to facilitate the overlapping of compute and IO on the CSX device. This mechanism supports the computation model where a problem can be broken into a number of individual computation steps but where the IO is a significant part of the overall time. To achieve the best performance it is necessary to perform the IO while the computation is on-going and to balance computation with data transfer to ensure computation is not stalled waiting for data transfer.

This mechanism relies on the move functions being asynchronous to achieve the overlap between compute and IO.

A double-buffered object contains two data buffers of equal size. It also maintains state on how the buffers are being used. While iteration n of a compute operation is executing on the accelerator using one set of data, then either:

1. the data for the next iteration ($n+1$) can be moved to the accelerator to be ready for the next iteration of processing, or
2. the computed results of the previous compute iteration ($n-1$) can be moved back to the host machine.

These two data movement models can be supported using CSPX double-buffered objects. Depending on the balance of compute and communication, you may wish to use both of these, for maximum overlap.

In a loop of double-buffered transfers and accelerator function calls, there will always be adjacent move and sync operations on the double-buffered object. These effectively do a *swap* of the two buffers: passing data in one and returning an empty buffer for the next iteration. At any time, one of the buffers is resident on the host and the other on the accelerator.

The internal state of the objects handles the initial case where both buffers are empty and therefore the first move to the accelerator, and first sync with the data back, are effectively 'null' operations.

4.1 Using double-buffered objects

The general principles of double buffered object are presented using the C++ interface. Fuller examples are provided below in both C++ and C.

There are a number of details which need to be handled in the first and last iterations. These are shown more in the full examples later.

A `CSPXDoubleObject` object is declared and initialized in the same way as a `CSPXObject` object. Two data buffers need to be attached to the object as shown in [Figure 36](#).

```
// create storage for the two sets of data
double buffer[2][PROBLEM_SIZE];

// create a double buffer object using the pair of buffers
CSPX::DoubleObject<double> dbObj(buffer[0], buffer[1], PROBLEM_SIZE,
                                numProcesses, CSPX_OBJECT_WRITE);
```

Figure 36. Initializing a double-buffered object

The double buffer type also has as process group equivalent which allows a group of double-buffered objects to be created, one object per member of the group.

4.1.1 Double-buffered transfers to accelerator

An outline of C++ host code using double buffering is shown in [Figure 37](#). In each iteration of the loop, the data for the next iteration is sent. As the move function is nonblocking (asynchronous) the data is transferred in parallel with the remote function call. This requires the data for the first iteration to be sent before the loop is entered. The timeline for these interactions is shown in [Figure 38](#). For simplicity, this only shows data transfers to the accelerator, a real application is likely to also transfer data back as well. The corresponding accelerator code is shown in [Figure 39](#).

```

... // initialize data in first buffer
process.move(dbObj); // set up: send data for iteration 1

for (int i = 0; i < ITERATIONS; i++) {
    process.sync(dbObj); // get handle back from accelerator
    ... // initialize data in next buffer
    process.move(dbObj); // and send to accelerator
    process.call(params, "cn_function"); // call function on previous buffer
}

// last call, don't need to send data this time
process.sync(dbObj); // get handle back from accelerator
process.call(params, "cn_function");
    
```

Figure 37. Host code

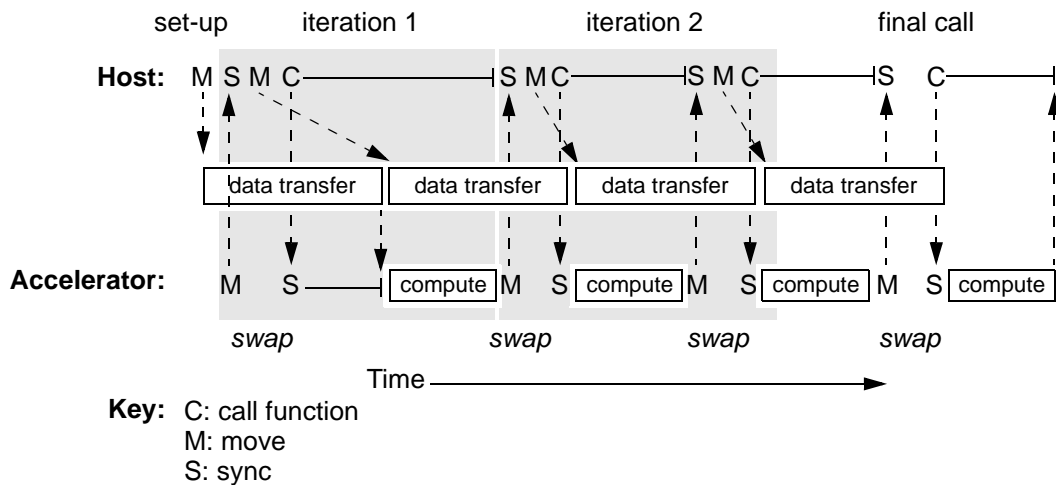


Figure 38. Transferring data to accelerator

```

int cn_function(CSPXProcess parent, void *param) {
    CSPXDoubleObject *obj = (CSPXDoubleObject *) param; // reference to dbl object

    data = CSPX_double_object_sync_pointer(obj); // get the current working buffer
    CSPX_double_object_move(parent, obj); // move the buffer back to the host

    do_compute(data);

    return 0;
}

```

Figure 39. Accelerator code

4.1.2 Double-buffered transfers from accelerator

The complementary case is where we move the results of iteration $n-1$ back on iteration n . In this case the accelerator code will be producing results to send back to the host. An outline of the C host code is shown in [Figure 40](#), the corresponding accelerator code is in [Figure 41](#). Again, for simplicity, this example only shows the data transfer *from* the accelerator.

```

CSPXDoubleObject input_output_obj; /* the double-buffer object */
...                               /* initialize object as a READ object */

for (i = 0; i < N ; i++) {
    CSPX_double_object_move(process, &input_output_obj); /* move data for next
iteration */
    CSPX_process_call(process, "compute_function", ...);
    CSPX_double_object_sync(&input_output_obj); /* wait for result to return */
}
CSPX_double_object_sync(&input_output_obj); /* and wait for the final result */

```

Figure 40. Host code

```

int cn_function(CSPXProcess parent, void *param) {
    CSPXDoubleObject *obj = (CSPXDoubleObject *) param; /* reference to dbl object */

    data = CSPX_double_object_sync_pointer(obj); /* get the current working buffer */

    do_compute(data);

    CSPX_double_object_move(parent, obj); /* move previous dbl object back,
on the first call this is a no-op */

    return 0;
}

```

Figure 41. Accelerator code

4.2 Example

We can now take the example program and add double buffering to transfer the data to the accelerator. In practice there are special cases to be handled when setting up the loop and for the final iteration. These are shown in this example.

Another important point is that the double buffered object type (`CSPX::DoubleObject` in C++ or `CSPXDoubleObject` in C and Cn) is a wrapper around the underlying CSPX objects which maintains the state of the buffers. This has two implications for your code.

Firstly, it is necessary to declare a double buffered object on the host and on the accelerator. Secondly, the parameter that is passed between the processes is the inner object (of type `CSPXDoubleObjectHandle`) rather than the double buffered object itself. Part of the initialization required is to assign this inner object to the double buffered objects on both the host and the accelerator.

The header file used by this example to share the parameters is shown in [Figure 42](#). The input parameter is declared as an object of type `CSPXDoubleObjectHandle`.

```
struct sqrt_args {
    CSPXDoubleObjectHandle inputs;
    CSPXObject results;
    int count;
};
```

Figure 42. Header file for double buffered example: `sqrt_dbl.h`

4.2.1 Accelerator code

The main accelerator code is shown in [Figure 43 on page 47](#). There are two key differences from the earlier examples. The input object is declared as a static variable so that it can maintain the state of the buffers between calls. Secondly, the code exports two functions to be called by the host. The first is an initialization function which is used to initialize the double buffered object before the first iteration of the loop, the other is a double-buffered version of the square root function.

The initialization code, initializes the object (with the `CSPX_double_object_new` function) and the assigns the input object passed as a parameter to the inner object. Finally it does an initial move of the object to the host to start the process of swapping buffers.

The `square_root` function is almost identical to the single-buffered version, apart from the difference in the type of the input object and the fact that the input object is a static variable rather than being a parameter to the function.

4.2.2 C++ host code

The body of the main function from the C++ host code is shown in [Figure 44 on page 48](#). In this case, the code declares a two dimensional array to hold the buffers for the input data. Both buffers are attached to the double buffered object. The initialization function on the accelerator is then called to pass this double buffered object to the accelerator.

Next the first set of input data is initialized and moved to the accelerator. Then the loop is entered and this synchronizes with the input buffer transferred from the accelerator. This combination of a move and a sync effectively swaps the buffers between the host and the accelerator. The next set of input data is then prepared and moved to the accelerator. The results object is also moved to the accelerator and the square root function called. Then, as before, the host waits for the results.

These steps are repeated for each iteration. Except for the last where there is no need to send any input data as the accelerator will be using the inputs sent on the previous iteration.

4.2.3 C host code

The C version of the host code is shown in [Figure 45 on page 49](#). This follows the same pattern as the C++ code. The main difference is that the C code has to explicitly extract the inner buffer object and assign it to the argument structure (this is done automatically by the

```

#pragma csp_xexport(square_root, init)

static CSPXDoubleObject double_inputs;

int init(CSPXProcess parent, void* const params) {
    struct sqrt_args *args = (struct sqrt_args *) params;

    // initialize the local double buffer object
    CSPX_double_object_new(&double_inputs);
    // attach the data buffer object to this object
    CSPX_double_object_set_handle(double_inputs, args->inputs);
    // move object to host before first iteration
    CSPX_double_object_move(parent, &double_inputs);
}

int square_root(CSPXProcess parent, void* const params) {
    struct sqrt_args *args = (struct sqrt_args *) params;
    double* inputs;
    double* results;
    int i;

    // get a pointer to the data attached to the objects
    inputs = (double *) CSPX_double_object_sync_pointer(&double_inputs);
    results = (double *) CSPX_object_sync_pointer(&args->results);

    for (i = 0; i < args->count; i += __NUM_PES__) {
        poly double d;
        poly int index;

        // calculate address of data to operate on
        index = i + get_penum();
        // copy input from mono memory
        memcpym2p(&d, &inputs[index], sizeof(double));
        // perform calculation
        d = sqrtm(d);
        // write result back to mono memory
        memcpyp2m(&results[index], &d, sizeof(double));
    }

    // return results back to host
    CSPX_object_move(parent, &args->results);
    // swap the input buffer back to the host
    CSPX_double_object_move(parent, &double_inputs);
}

```

Figure 43. Accelerator code for double buffered example: `sqrt_dbl.cn`

C++ class). Also, the C function provide access to the pointer to the current buffer. This means that the code does not have to keep track of which buffer to use on each iteration.

```
int buffer = 0; // pointer to buffer to use on each iteration

// an array of input and result values
double inputs[2][PROBLEM_SIZE], results[PROBLEM_SIZE];

// object for the accelerator processes
CSPX::State<> processes("sqrt_dbl.csx");
int numProcesses = processes.numProcesses();

// create objects to transfer the input and result arrays
CSPX::DoubleObject<double> inputObj(inputs[0], inputs[1], PROBLEM_SIZE,
                                     numProcesses, CSPX_OBJECT_WRITE);
CSPX::Object<double> resultObj(results, PROBLEM_SIZE, numProcesses, CSPX_OBJECT_READ);

// Initialize the function arguments
ParamObject params(processes.numProcesses());

// add arguments to parameter object
params.push(inputObj);
params.push(resultObj);
params.push(PROBLEM_SIZE);

processes.call(params, "init");

// initialize input values for first call
for (int i = 0; i < PROBLEM_SIZE; i++) {
    inputs[buffer][i] = i*i;
}

// move the objects to the accelerators
processes.move(inputObj);

for (int iter = 1; iter <= ITERATIONS; iter++) {
    buffer = buffer ^ 1; // update buffer number

    // get the previous buffer back from the accelerator
    processes.synch(inputObj);

    // not last call: send data for this iteration
    if (iter < ITERATIONS) {
        for (int i = 0; i < PROBLEM_SIZE; i++) {
            inputs[buffer][i] = iter+(i*i);
        }
        // move the buffer to the accelerator
        processes.move(inputObj);
    }

    // move the result object to the accelerator
    processes.move(resultObj);
    // call the function on the accelerator
    processes.call(params, "square_root");
    // and wait for the results to return
    processes.synch(resultObj);
}
```

Figure 44. C++ code for double buffered example: sqrt_dbl.cpp

```

int i, res, iter;
CSPXErrno err;
double * buff_ptr;
CSPXDoubleObject double_inputs;
CSPXProcess the_process;

/* an array of input and result values */
double inputs[2][PROBLEM_SIZE], results[PROBLEM_SIZE];
/* Structure for passing arguments */
struct sqrt_args args;

the_process = CSPX_process_create("sqrt_dbl.csx", 0, NULL, &err);

/* create objects to transfer the input and result arrays */
CSPX_double_object_new(&double_inputs);
CSPX_double_object_attach(&double_inputs, inputs[0], PROBLEM_SIZE*sizeof(double),
                        inputs[1], PROBLEM_SIZE*sizeof(double), CSPX_OBJECT_WRITE);

/* Note: pass the "internal" object to the accelerator */
args.inputs = CSPX_double_object_get_handle(double_inputs);

CSPX_object_new(&args.results);
CSPX_object_attach(&args.results, results, PROBLEM_SIZE*sizeof(double),
                  CSPX_OBJECT_READ);

/* initialize the other function arguments */
args.count = PROBLEM_SIZE;

/* pass the double buffered object across to the accelerator */
CSPX_process_call(the_process, "init", &args, sizeof(args), &res);

/* move the first buffer to the accelerator */
CSPX_double_object_move(the_process, &double_inputs);

for (iter = 1; iter <= ITERATIONS; iter++) {
    /* get the previous buffer back from the accelerator */
    CSPX_double_object_sync(&double_inputs);

    /* not last call: send data for this iteration */
    if (iter < ITERATIONS) {
        buff_ptr = (double *) CSPX_double_object_sync_pointer(&double_inputs);
        for (i = 0; i < PROBLEM_SIZE; i++) {
            buff_ptr[i] = iter+(i*i);
        }
        /* move the buffer to the accelerator */
        CSPX_double_object_move(the_process, &double_inputs);
    }

    /* move the result object to the accelerator */
    CSPX_object_move(the_process, &args.results);
    /* now call the function on the accelerator */
    CSPX_process_call(the_process, "square_root", &args, sizeof(args), &res);
    /* wait for the results object to be returned */
    CSPX_object_sync(&args.results);
}

```

Figure 45. C code for double buffered example: sqrt_dbl.c

5 Data pipes

CSPX provides a data pipe implementation to facilitate streaming data to and from accelerators. A CSPX data pipe is similar to a Unix/Linux pipe where data can be written by one process and read by another. A pipe is unidirectional and connects two processes where one process is the producer process writing to the pipe and the other process is the consumer process reading from the pipe. [Figure 46](#) shows a simple example with a number of doubles being written to a pipe by one the producer and read by the consumer.

<pre>// Producer Process CSPXPipe a; // Declare the pipe // Connect this process, the producer, // to the accelerator process, the consumer CSPX_pipe_init(&a, p, 4096, CSPX_OBJECT_WRITE); int i; double f; for (i = 0; i < 10; i++) CSPX_pipe_write(&a, &f, sizeof(f));</pre>	<pre>// Consumer Process CSPXPipe pipe; // Declare the pipe // get a pointer to the pipe from // structure passed as parameter pipe = params->pipe; int i; double f; for (i = 0; i < 10; i++) CSPX_pipe_read(&a, &f, sizeof(f));</pre>
---	---

Figure 46. Use of pipes between processes

Note that in this example the data is read back in the same chunk sizes as the data was written but this is not necessary, for example the consumer could have read back all the data in one single read.

A pipe is declared with a certain amount of buffer space allocated. In the above example the buffer size is 4KB. The `CSPX_pipe_write` functions will not block if there is sufficient space in the buffer for the data being written. Similarly, the `CSPX_pipe_read` function will not block if there is data in the buffer to satisfy the request.

As with other CSPX operations, the pipe API is symmetric and the same functions is available on the host and the accelerator processes. However, in the current implementation, pipes can only be created by the host process. The handle to the pipe (the `CSPXPipe` reference) is passed to the other process either as a function parameter or embedded in other CSPX objects.

A data pipe is a useful way to send some sort of data stream to an accelerator process for processing. For example digital video data can be streamed through the pipe to the accelerator process. A pair of stereo audio streams can be sent to the two processors on a ClearSpeed board for computation. A text file could be streamed through a data pipe to the accelerator as a simple way of implementing file I/O.

5.1 Basic pipe functions

The basic types functions used to create and use a data pipe are defined in the `CSPXPipe.h` header file. A pipe is declared and initialized as follows:

CSPX_pipe_init: Initialize a pipe connecting this process to an accelrator process. The internal buffer size is specified in bytes and an attribute parameter indicates

whether it is a read pipe or a write pipe (from the point of view of the creator of the pipe).

CSPX_pipe_write: Write the specified number of bytes through the pipe from the address given.. The function will not block if there is sufficient buffering to copy the data. However, if there is insufficient space the write will stall until there is space to complete the write. The function returns the number of bytes written.

CSPX_pipe_read: Read a number of bytes from pipe to the specified address. The function will not block if there is sufficient data in the buffer to satisfy the request. If there is insufficient data, the function will block until the required number of bytes arrive. The function returns the number of bytes read.

CSPX_pipe_drain: Returns once the pipe is empty. Called by the writing process to ensure that the pipe has drained.

5.2 Efficiency considerations

As with other data transfer operations between the host system and the accelerators the performance of the pipe depends on two critical factors: the size of the individual transfers and the alignment of the transfers. Small data transfers sizes will result in poor performance due to the overheads in transferring small amounts of data and better performance will be seen with larger data transfer blocks. The current implementation does not attempt to combine multiple small transfers into one larger one.

It is also important to maintain data alignments that are compatible with the DMA engine on the accelerator. CSPX will report an error if a pipe operation is misaligned. However, for some uses of pipes efficiency is not important, for example using a pipe as a simple debug aid to trace program progress.

Perhaps the most important consideration in using the pipe abstraction is to avoid stalling by balancing the consumer rate and the producer rate. If either process (consumer or producer) stalls then this will greatly reduce the efficiency of the pipe. The pipe is implemented in a 'lock-free' way so that the communication between producer and consumer is reduced to a minimum. The ClearSpeed Visual Profiler can be used to view the performance of the data pipe and adjustments made to either the producer or consumer rates to avoid any unnecessary stalls. It may be possible to avoid stalls by using larger internal buffer sizes.

5.3 Groups of pipes

Again like other CSPX abstractions, a pipe has a group equivalent, a `CSPXPipeGroup`. In its simplest use, this allows a host process to create a group of pipes connecting it to all the processes in a group. All the pipes in a single group have the same direction. The main functions on a pipe group are the following:

CSPX_create_pipe_group: Create a pipe group connecting this process to all the processes in the group. The size and attribute parameters are as in the initialization of a single pipe, i.e. the size of the internal buffering used for each pipe and the direction of the pipes. The pipe group can also be given a name which is useful for debugging and profiling purposes. The index gives a handle to the first pipe in the group. This can be used to apply an operation to a specific pipe in a group.

CSPX_apply_pipe_group: Calls a function on each member of the pipe group. The function can be a call to a pipe read or write function so this is the simplest way to read

from or write to a group of pipes. The function will return when all the reads or writes have completed.

6 Profiling CSPX operations

CSPX contains trace points for input to the ClearSpeed Visual Profiler using the profiler's standard host tracing library. These trace points provide a view of the behavior of the application in terms of CSPX functions.

In looking at a profiler trace of a program run using CSPX, the goals are the same as when using the profiler to view the execution of an accelerated application. That is, to look for:

- a) Unnecessary execution stalls where the processors are not active.
- b) Effective overlapping of data transfer and compute on the host and accelerator.

The profiler trace allows these events to be viewed in terms of CSPX entities. For example, a stall may be associated with a CSPX object or activity on the accelerator will be identified as being associated with a specific accelerator function call.

The CSPX objects are identified, by default, by displaying the file and line number where they are attached to real data. However, each object can be given a more descriptive name with the `CSPX_object_set_name` function.

6.1 Trace generation

CSPX trace generation is enabled by setting the environment variable `CSPX_TRACE`. The default name of the trace file is `cspcx.cst`. This default can be overridden. Note that the CSPX trace is generated for the entire run of the application hence representative samples of the application should be run to avoid generating huge trace files. Also, the trace file is generated using the `C atexit` functionality so that if the application terminates via a fault the trace file will not be generated.

It is also possible to add custom trace events to the host application via the `csvprof_trace.h` interface so that the overall behavior of the application can be seen with the CSPX operations in context.

6.2 Events

The specific events traced by default are abstraction specific, for example, the double-buffer events are different from the pipe events but a number of the events are common to all views. In the remainder of this section we will describe the common events and the abstraction specific events. The event names listed here correspond to the event names as rows in the profiler display. The events are generally also split into events related to host activity and events related to accelerator activity.

1. **csx_function_call**

The bars represent the start and end of each function called on the accelerator by the host using the CSPX remote function call mechanism. The function name is displayed along with the file and line number of the function call.

2. **csx function call async**

As for **csx_function_call** but for the asynchronous (non-blocking) function call.

3. **dma_idle**

The bars represent the times that the DMA channel to the accelerator is idle and gives a good indication of how much capacity remains unused. If stalls are observed

elsewhere, the periods where the DMA is idle are good candidates for being more efficiently used.

4. csx host: CSPX_object_move

The bar represents the duration of the data transfer for a CSPX object. The object is identified via a tool tip with the file and line number pair where the object was attached. Note that the bar only represents the time for the actual data transfer and not the time from the issuing of the corresponding `CSPX_object_move` request.

5. csx host: csx_synch_obj(stall)

These events represent the periods when the host is stalled waiting for an object to arrive back on the host process. This is the time stalled in the function `CSPX_object_sync` functions. Note that due to the way the synchronization works, even if the object is already in the host process and there is no stalling, a very small event will still be generated. This is an area that could be improved.

6. csx card: csx_synch_obj(stall)

These events correspond to the accelerator stalling waiting for objects to become resident in the accelerator process. The objects are identified in the usual way via their attach point file and line number.

The remaining events are mostly internal events to CSPX and are mainly of interest to CSPX developers. Note also that there are some events marked as 'unknown', these correspond to gaps in the time allocated to CSPX events and are generated by the profiler mechanisms. If custom profiler events are generated then the actual code that these unknown events correspond to can be more easily identified.

The internal events recorded correspond to two internal threads. The CSPX *agent* thread manages all the data transfers (DMA channels) for a board. The CSPX *support* thread performs operations on the host on behalf of the accelerator thus allowing the accelerator to perform the same operations as the host process.

7. csx_agent_read_event

Time spent by the per-accelerator host support thread in performing reads from the accelerator.

8. csx_agent_write_event

Time spent by the per-accelerator host support thread in performing writes to the accelerator.

9. csx_support_inferior_push_event

Time spent in moving object from accelerator to host, generated in response to `CSPX_object_move` operations on the accelerator processes.

7 Compiling and running CSPX programs

7.1 Header files

The header files and functions are documented in the API reference chapters. There is also a header file called `cspix.h` which will include the most commonly used headers.

The header files are under the include directory in the standard SDK installation. The headers which are only for use by host programs are in the `host` subdirectory. The header files which can be used by both C and C⁺⁺ programs are in the `common` subdirectory.

7.2 Compiling RPC functions C⁺⁺ programs

The ClearSpeed C⁺⁺ compiler (`cncc`) supports several options relevant to the CSPX remote procedure call (RPC) pragma. If you use the compiler driver program, `cscn`, instead of the compiler, `cncc`, then these options can be passed to `cncc` using the `-Wcn` option.

If you use the `cscn` driver with the `cspix_rpc` pragma then you will need to explicitly specify the names of the generated 'interface' files and the `.csx` file. This is because the driver will generate arbitrary file names for the intermediate files during the different phases of compilation. The example could be compiled using a series of commands similar to the following:

```
cscn -E sqrt_rpc.cn -o sqrt_rpc.csi
cncc sqrt_rpc.csi -o sqrt_rpc.is
cscn --dynamic -lcspix sqrt_rpc.is -o sqrt_rpc.csx
```

The generated C⁺⁺ wrapper function has to load the final compiled `.csx` file and so, by default, `cncc` assumes that the CSX executable file will be named after the source file. The name of the `.csx` file in the generated C⁺⁺ code can be changed with the `-cspix-csx-file` option. For example, to change the name of the CSX file, you could use the following commands:

```
cscn -E sqrt_rpc.cn -o sqrt_rpc.csi
cncc -cspix-csx-file output sqrt_rpc.csi -o sqrt_rpc.is
cscn --dynamic -lcspix sqrt_rpc.is -o output.csx
```

Note: The same name must be used both when compiling the file containing the RPC pragma and when generating the `.csx` file.

See [Compiling the C⁺⁺ code on page 13](#) for more details.

The relevant compiler command line options are:

-cspix-emit

Only emit the host based source files needed for the `cspix_rpc` pragma to work. This option stops the compiler from also emitting the assembler file that would normally be produced.

-cspix-dir <dir>

This specifies a directory in which any intermediate auto-generated host source files will be placed. The directory must already exist.

-csp_x-filename <name>

This specifies a name that will be used for the generated file. By default the compiler will use the original input file name and append `_interface` to it. This option forces the compiler to use an alternative output file name. The compiler will generate `name.cpp` and `name.h` files.

-csp_x-csx-file <name>

This command line option informs the compiler what the eventual CSX file name will be. This is required as the generated source for the host includes a step that loads this file. There should be one CSX file that is created. The default is the base file name for the input source file with a `.csx` appended (for example, `test.cn` will imply `test.csx` for the eventual CSX file).

7.3 Linking Cⁿ programs

Cⁿ programs must be dynamically linked (the default) and linked with the CSPX library using the command line option `-lcspx`.

7.4 Error handling

In order to minimize the overhead of CSPX function calls, no or limited checking is performed on parameters passed to functions. Functions that simply analyze a value and return a boolean will not do any error checking.

CSPX functions which modify or create state will check the results and flag an error when necessary.

The C++ interface uses exceptions to trap any runtime errors that occur.

The C and Cⁿ functions will provide an error code to the caller. This is done in one of two ways. Most functions return a result which indicates the success or failure of the operation. Functions which return a pointer will return NULL in the case of an error and will return the error in a parameter.

7.4.1 Error codes

The error codes returned by CSPX functions are listed in [Table 1](#). These are defined in `cspx_errno.h`. Errors may be returned by the underlying CSAPI functions. If so, the CSAPI error code (see the *Runtime User Guide*) is converted to a CSPX error code by adding an offset. An error string can be obtained from the error code by calling the function `CSPX_get_error_string`.

CSPXErrno	Description
<code>CSPXErrno_success</code>	No error.
<code>CSPXErrno_csapi_error</code>	This is added to any <code>CSAPIErrno</code> and returned as <code>CSPXErrno</code>
<code>CSPXErrno_double_object_card_unknown</code>	Double object is in an unknown state on the card side

Table 1. Error codes

CSPXErrno	Description
CSPXErrno_double_object_in_init	Operation attempted card side on double object in initial state
CSPXErrno_group_index_out_of_range	Index to a group out of range
CSPXErrno_object_not_initialised	Attempt to use an uninitialized object
CSPXErrno_object_delete_in_use	Attempting to delete an in-use object
CSPXErrno_object_attach_in_use	Attempting to attach to an in-use object
CSPXErrno_object_detach_non_home	Attempting to detach object in non-home process
CSPXErrno_object_handle_invalid	Attempting to use an invalid object handle
CSPXErrno_object_not_resident	Object not resident
CSPXErrno_object_null_address	Object home address is null
CSPXErrno_object_card_request	Card side object request is in an undefined state
CSPXErrno_pipe_wrong_key	Pipe descriptor has wrong key
CSPXErrno_pipe_double_wait	Both ends of a pipe are waiting
CSPXErrno_pipe_wait_count_discrepancy	Discrepancy in wait counters
CSPXErrno_pipe_drain_not_writer	Pipe drain not applied to pipe writer
CSPXErrno_pipe_card_bad_descriptor	Bad pipe descriptor on card side
CSPXErrno_pipe_card_bad_key	Bad pipe key on card side
CSPXErrno_pipe_card_deadlock	Card side pipe code about to deadlock
CSPXErrno_process_invalid_card_state	Invalid card state
CSPXErrno_process_function_not_found	Function not found (could not retrieve address)
CSPXErrno_process_free_buffer_error	Could not free buffer space associated with object
CSPXErrno_process_no_free_space	No free space found in migrated object space
CSPXErrno_process_job_queue_corrupt	The job queue has become corrupted
CSPXErrno_process_invalid_card_address	Invalid card side address specified for migrated object
CSPXErrno_process_completer_function_failed	Call of completer function failed
CSPXErrno_process_agent_unknown_request	Process agent thread unknown request received
CSPXErrno_process_support_fatal_error	Process support thread fatal error has occurred
CSPXErrno_process_inferior_req_unrecog	Process invalid inferior request
CSPXErrno_process_terminate_called	Card side terminate has been invoked (not actually an error)
CSPXErrno_ptr_no_more_handles	Unable to allocate handle, no more available
CSPXErrno_ptr_invalid_ptr	Invalid CSPX pointer
CSPXErrno_ptr_set_handle_failed	Failed to set handle for CSPX pointer
CSPXErrno_ptr_handle_out_of_range	Handle in CSPX pointer out of range
CSPXErrno_inf_protocol_beyond_buffer	Inferior protocol internal error pointer beyond buffer end

Table 1. Error codes (continued)

CSPXErrno	Description
CSPXErrno_inf_protocol_too_large	Inferior protocol internal error size larger than buffer
CSPXErrno_feature_not_implemented	Attempt to use an unimplemented feature
CSPXErrno_pthread_mutex_initialise	Unable to initialize mutex
CSPXErrno_pthread_cond_initialise	Unable to initialize condition variable
CSPXErrno_pthread_mutex_lock_failed	Failure in pthread_mutex_lock call
CSPXErrno_pthread_cond_broadcast_failed	Failure in pthread_cond_broadcast call
CSPXErrno_pthread_cond_wait_failed	Failure in pthread_cond_wait call
CSPXErrno_pthread_mutex_unlock_failed	Failure in pthread_mutex_unlock call
CSPXErrno_pthread_mutex_failure	Failure in a sequence of pthread mutex operations
CSPXErrno_pthread_thread_create_failed	Failed to create thread
CSPXErrno_trace_init_failed	Trace initialization failed
CSPXErrno_trace_unable_to_register_event	Trace unable to register event
CSPXErrno_trace_atexit_reg_failed	Trace atexit registration failed

Table 1. Error codes (continued)

7.4.2 Error functions

Some functions are provided to assist with error handling, see [Section 9.7: Error handling on page 142](#) for information on the C functions and [Section 10.6: Error handling on page 163](#) for details of the C++ functions.

7.4.3 C++ exceptions

The exceptions that can be thrown by CSPX functions are shown in [Table 2](#).

CSPX Exception	Description
ExceptionCallAlreadyInProgress	A non blocking call (see page 65) has been made before another has finished.
ExceptionCallNotInProgress	An attempt to wait (see page 66) when a non blocking call has not been made.
ExceptionConnectionFailed	CSPX failed to connect to the requested resources.
ExceptionGetProcessFailed	A CSPX function was unable to access the CSPAI process information.
ExceptionJoinFailed	CSPX was not able to join the specified objects (perhaps because the named object does not exist).
ExceptionNumCardsFailed	CSPX was not able to find the number of cards in the system.
ExceptionNumProcessorsFailed	CSPX was not able to find the number of processors on a card.
ExceptionOutOfRange	Index to a group out of range.
ExceptionPthreadFailure	An error occurred with the thread when using the CSPX::Handler (see page 95) class.

Table 2. CSPX exceptions

CSPX Exception	Description
ExceptionReductionUndefined	The reduction function for use by <code>getValueReduce</code> (see page 76) was not defined.
ExceptionRemoteCallFailed	CSPX was not able to call the remote function.
ExceptionSemaphoreAllocFailed	CSPX was unable to allocate semaphores for a process.
ExceptionSemaphoreSignalFailed	CSPX was unable to signal on a semaphore.
ExceptionSemaphoreSignalFailed	CSPX was unable to wait on a semaphore.
ExceptionSizeMismatch	The number of items pushed onto a parameter object does not match the number of processes.

Table 2. CSPX exceptions

7.5 Running code on a simulator

The CSPX interface can be used to run code on a simulator (`isim` or `casim`). This can be useful for code development and testing when accelerator hardware is not available.

7.5.1 C++ interface

When using the C++ interface, you can explicitly use the `SimulatorConnectionPolicy` to connect to a simulator. Alternatively, the `DefaultConnectionPolicy` which normally connects to accelerator hardware, can be made to connect to a simulator by setting the environment variable `CSAPI_CONNECT_SIM`. This allows you to decide whether to use hardware or a simulator at run time. See below for a description of all the relevant environment variables.

By default CSPX will use only a single simulator (see note on [page 26](#)) unless you write a custom connection policy, see [Section 2.6.3: Custom policies on page 28](#).

Remote procedure call

The RPC interface described in [Section 1.2: Remote procedure call on page 9](#), uses the C++ `DefaultConnectionPolicy`. This means that programs written using RPC can use the same environment variable to run on a simulator.

7.5.2 C interface

The C function `CSPX_process_create` also uses the environment variable to determine whether to connect to a simulator.

7.5.3 Environment variables

If the environment variable `CSAPI_CONNECT_SIM` is set to any value then the default connection policy and the `CSPX_process_create` function will attempt to connect to a simulator instead of acceleration hardware.

The simulator connection can be modified with the following environment variables:

`CSAPI_INSTANCE`: this specifies the simulator instance number to connect to. The simulator instance number can be specified with the `-i` option to `isim` and `casim`. The default instance number, if not specified, is 0.

`CSAPI_HOST`: the simulator can be run on a different machine to the host application. This environment variable can be used to specify the name or IP address of the machine running the simulator. The default, if not specified, is 'localhost'.

8 C++ API reference

This chapter describes the C++ interface to CSPX.

8.1 Processes

```
#include <CSPXState.h>
```

This class defines the C++ interface to an accelerator and the code that is loaded onto it (a CSX program). This embodies both the single process and the process group concepts.

8.1.1 Classes

CSPX::State<DistributionPolicy, ConnectionPolicy>

The `CSPX::State` class provides member functions to allow you to migrate data to and from the accelerator, and invoke functions on the accelerator. Note that this is a template class and you can override the distribution policy and connection policy when creating an object of this class. See [Section 8.5: Connection policies on page 86](#) and [Section 8.6: Distribution policies on page 87](#) for more information on these policies.

It is possible to create a CSPX process on a simulator if hardware is not available. See [Section 7.5: Running code on a simulator on page 61](#) for details.

8.1.2 Member functions

State

Prototype

```
State(const char* filename);
```

Description

Creates a new state object by loading the specified CSX object on the accelerators.

Parameters

`filename`: the name of the CSX executable file to load on to the accelerator. This can be an absolute or relative path name. If this is a relative path name then the `CSPATH` environment variable is used to search for the file.

Throws

```
CSPX::ExceptionConnectionFailed  
CSPX::ExceptionLoadFailed  
CSPX::ExceptionNumCardsFailed  
CSPX::ExceptionNumProcessorsFailed
```

~State

Prototype

```
~State(void);
```

Description

Default destructor, this unloads the program from the accelerators, disconnects and cleans up the state.

call

Prototype

```
std::vector<int> call(  
    CSPX::Object<U,V> obj,  
    char *function_name  
);
```

Description

Call an accelerator function. The parameters are attached to an object. The parameters should be in a structure that is shared with the **C** code on the accelerator. The accelerator function will expect to receive a parameter which corresponds to this structure. See [Section 8.2: Objects on page 71](#) for more details.

Parameters

obj: an object to be passed as a parameter to the called function
function_name: a string containing the name of the function to call

Throws

CSPX::ExceptionRemoteCallFailed

Returns

A vector of results values from the called functions.

call

Prototype

```
std::vector<int> call(  
    CSPX::Parameter obj,  
    char *function_name  
);
```

Description

Call an accelerator function. The parameters are passed in a parameter object. The arguments should be added to the parameter object in the same order they appear in the

structure parameter in the called function. See [Section 8.3: Parameter objects on page 75](#) for more details.

Parameters

`obj`: a parameter object passed to the called function

`function_name`: a string containing the name of the function to call

Throws

`CSPX::ExceptionRemoteCallFailed`

Returns

A vector of results values from the called functions.

`callNonBlock`

Prototype

```
void callNonBlock(  
    CSPX::Parameter obj,  
    char *function_name  
);
```

Description

Call an accelerator function. The parameters are passed in a parameter object. The arguments should be added to the parameter object in the same order they appear in the structure parameter in the called function. See [Section 8.3: Parameter objects on page 75](#) for more details.

This function will return immediately, allowing the accelerator function to continue asynchronously. The `callWait` function can be used to synchronize with the accelerator function.

Parameters

`obj`: a parameter object passed to the called function

`function_name`: a string containing the name of the function to call

Throws

`CSPX::ExceptionCallAlreadyInProgress`

`CSPX::ExceptionRemoteCallFailed`

Returns

Nothing.

callWait

Prototype

```
std::vector<int> callWait(void);
```

Description

Wait for a an accelerator function to terminate. The function on the accelerator must have already been called with `callNonBlock`.

Parameters

None.

Throws

```
CSPX::ExceptionCallNotInProgress
```

Returns

A vector of results values from the called functions.

getCSAPIState

Prototype

```
struct CSAPIState * getCSAPIState(unsigned int index);
```

Description

Gets the CSAPI state structure associated with the specified process.

Parameters

`index`: the index of the process

Throws

```
CSPX::ExceptionOutOfRange
```

Returns

A pointer to a CSAPI state structure for the process.

getProcess

Prototype

```
CSPXProcess getProcess(unsigned int index);
```

Description

Gets the process handle for the specified process.

Parameters

index: the index of the process

Throws

CSPX::ExceptionOutOfRange

Returns

The CSPX handle of the specified process.

join_named

Prototype

```
void join(  
    CSPX::Object<U,V> &obj,  
    const char *name  
);
```

Description

Joins objects on the host to objects on the accelerators.

Parameters

obj: the object to join to the target processes.

name: the name of the object on the target processes to join with.

Throws

CSPX::ExceptionJoinFailed

Returns

Nothing.

join_named

Prototype

```
void join(  
    CSPX::DoubleObject<U,V> &obj,  
    const char *name  
);
```

Description

Joins double objects on the host to double objects on the accelerators.

Parameters

obj: the object to join to the target processes.

name: the name of the double object on the target processes to join with.

Throws

CSPX::ExceptionJoinFailed

Returns

Nothing.

move

Prototype

```
void move(CSPX::Object<U,V> &obj);
```

Description

Moves an object to the target processes. This operation operates over the group of processes represented by the `CSPX::State` object. The data attached to the object will be sent to the accelerators according to the associated partition policy. This is a nonblocking, asynchronous call.

Parameters

obj: the object to move to the target processes.

Returns

Nothing.

move

Prototype

```
void move(CSPX::DoubleObject<U,V> &obj)
```

Description

Moves a double-buffered object to the target processes. This operation operates over the group of processes represented by the `CSPX::State` object. The data attached to the object will be sent to the accelerators according to the associated partition policy. This is a nonblocking, asynchronous call.

Parameters

`obj`: the object to move to the target processes.

Returns

Nothing.

numProcesses

Prototype

```
unsigned int numProcesses(void);
```

Description

Returns the number of processes that are defined in the `CSPX::State` object. This is used as an argument to many other member functions in other classes.

Parameters

None.

Returns

The number of processes.

sync

Prototype

```
void sync(CSPX::Object<U,V> &obj);
```

Description

Synchronizes with objects moved from the accelerator processes. This operation operates over the group of processes contained in the `CSPX::State` object. This is a blocking call which will return when all the data is resident on the host.

Parameters

`obj`: object to synchronize with

Returns

Nothing.

sync

Prototype

```
void sync(CSPX::DoubleObject<U,V> &obj);
```

Description

Synchronizes with double-buffered objects moved from the accelerator processes. This operation operates over the group of processes contained in the `CSPX::State` object. This is a blocking call which will return when all the data is resident on the host.

Parameters

`obj`: object to synchronize with

Returns

Nothing.

8.2 Objects

```
#include <CSPXObject.h>
```

This defines the class which represents objects used to communicate data with code running on accelerators.

8.2.1 Classes

CSPX::Object<T, PartitionPolicy>

This class defines the basic C++ data type which corresponds to `CSPXObject` (see [Section 9.2: Objects on page 112](#)). However, due to the nature of the C++ implementation the `CSPX::Object` is more like a *group* of objects (see [Section 9.3: Process groups on page 120](#)).

When declaring a `CSPX::Object` you must include the base type of the data being attached as a template argument. Optionally, you can also specify a `PartitionPolicy` template argument to specify how the data will be partitioned amongst the objects in the group (this defaults to `CSPX::DefaultPartitionPolicy`). See [Section 8.7: Partition policies on page 89](#) for more information on partition policies

8.2.2 Member functions

Object

Prototype

```
Object(  
    T* data,  
    size_t size,  
    unsigned int num_processes,  
    CSPXObjectAttribute attribute  
);
```

Description

The `Object` constructor creates an object or group of objects and attaches the data to them according to the partition policy for the object.

Parameters

- `data`: a pointer to the data to be attached
- `size`: the total size of the data: the number of items of type `T`
- `num_processes`: the number of processes the data is to be distributed to
- `attribute`: the read/write attribute of the data (see [Section 9.2: Objects on page 112](#))

begin

Prototype

```
iterator begin(void);
```

Description

Returns an iterator referring to the first element in the object group.

Parameters

None.

Returns

An iterator over the object group

end

Prototype

```
iterator end(void);
```

Description

An iterator to the element past the end of the sequence.

Parameters

None.

Returns

An iterator over the object group.

partitionCardinality

Prototype

```
size_t partitionCardinality(unsigned int index);
```

Description

Returns the number of data items in the partition specified by the index.

Parameters

index: the partition number, starting from 0

Throws

CSPX::ExceptionOutOfRange

Returns

The number of data items in the partition.

partitionSize

Prototype

```
size_t partitionSize(unsigned int index);
```

Description

Returns the size (in bytes) of the partition specified by the index.

Parameters

index: the partition number, starting from 0

Throws

CSPX::ExceptionOutOfRange

Returns

The size, in bytes, of the data in the partition.

purge

Prototype

```
void purge(void);
```

Description

Releases all resources allocated to this CSPX::Object. The function calls the CSPXObject_detach and CSPXObject_deinit functions for all CSPXObject objects

contained in the `CSPX::Object`. See [Section 9.2: Objects on page 112](#) for more details of these two functions.

Parameters

None.

Returns

None.

8.3 Parameter objects

```
#include <CSPXParameter.h>
```

This defines an object type that is used to pass function parameters in the C++ interface.

8.3.1 Classes

CSPX::Parameter

This class is used to create a parameter object to pass to a function called using the `CSPX::State` object. The accelerator code should expect to receive a parameter which is a structure with the items in the same order that they were pushed onto the parameter object.

Note that all parameters passed using the `CSPX::Parameter` are read/write and can be modified prior to function return. There are functions which allow you to access these values in various ways, either by reducing them or getting a vector of values.

8.3.2 Member functions

begin

Prototype

```
iterator begin(void);
```

Description

Returns an iterator referring to the first element in the object group.

Parameters

None.

Returns

An iterator over the object group.

end

Prototype

```
iterator end(void);
```

Description

An iterator to the element past the end of the sequence.

Parameters

None.

Returns

An iterator over the object group.

getValue

Prototype

```
std::vector<T> getValue(unsigned int handle);
```

Description

Returns the vector of values corresponding to the parameter item indicated by the handle.

Parameters

handle: a pointer to the parameter

Returns

A vector of result values.

getValueReduce

Prototype

```
T getValueReduce(T& msum, unsigned int handle)
```

Description

Reduces the parameter indicated by the handle (which was returned from the relevant push function used previously). The reduced value is returned and also placed into the argument

`msum`. An error is generated if the reduction function was not defined when the data was pushed.

Parameters

`msum`: a variable to store the result of the reduction

`handle`: a handle to the parameter to be reduced

Throws

`CSPX::ExceptionSizeMismatch`

Returns

The reduced value of the parameter.

push

Prototype

```
unsigned int push(  
    T data,  
    unsigned int alignment  
);
```

Description

Pushes a data item of type `T` onto the parameter block. The alignment of the item can also be specified, this defaults to the size of the data item if not specified. The data will be broadcast to all accelerators as a parameter.

Parameters

`data`: the value to be pushed on the parameter block

`alignment`: the desired alignment of the parameter (optional).

Returns

A handle which can be used in other functions that reference this parameter.

push

Prototype

```
unsigned int push(  
    std::vector<T> data,  
    unsigned int alignment  
);
```

Description

Pushes a vector of data items of type `T` onto the parameter block. The alignment of the items can also be specified, this defaults to the size of the data if not specified. Each

element of the vector will be passed to the corresponding process in the process group. The size of the vector must be the same as the number of processes.

Parameters

`data`: the vector of values to be pushed on the parameter block

`alignment`: the desired alignment of the parameter (optional).

Throws

`CSPX::ExceptionSizeMismatch`

Returns

A handle which can be used in other functions that reference this parameter.

push

Prototype

```
unsigned int push(  
    T data,  
    void*(*red_func)(void*, void*),  
    unsigned int alignment  
);
```

Description

Pushes a data item of type `T` onto the parameter block. A reduction function is associated with the parameter. The alignment of the item can also be specified, this defaults to the size of the data item if not specified. The data will be broadcast to all accelerators as a parameter.

Parameters

`data`: the value to be pushed on the parameter block

`red_func`: a pointer to a reduction function to be applied to the parameter when the function returns

`alignment`: the desired alignment of the parameter (optional).

Returns

A handle which can be used in other functions that reference this parameter. The return value should be saved for use as a handle to invoke the reduction function.

push

Prototype

```
unsigned int push(  
    std::vector<T> data,  
    void*(*red_func)(void*, void*),  
    unsigned int alignment  
);
```

Description

Pushes a vector of data items of type T onto the parameter block. A reduction function is associated with the parameter. The alignment of the items can also be specified, this defaults to the size of the data item if not specified. Each element of the vector will be passed to the corresponding process in the process group. The size of the vector must be the same as the number of processes.

Parameters

`data`: the vector of values to be pushed on the parameter block

`red_func`: a pointer to a reduction function to be applied to the parameter when the function returns

`alignment`: the desired alignment of the parameter (optional).

Throws

`CSPX::ExceptionSizeMismatch`

Returns

A handle which can be used in other functions that reference this parameter. The return value should be saved for use as a handle to invoke the reduction function.

push

Prototype

```
unsigned int push(CSPX::Object<T,U> &obj);
```

Description

Pushes a CSPX Object onto the parameter block. This must be a group of objects of the same size as the process group. Each object will be passed to the corresponding process in the process group.

Parameters

`obj`: the object to be pushed on to the parameter block

Returns

A handle which can be used in other functions that reference this parameter.

push

Prototype

```
unsigned int push(CSPX::DoubleObject<T,U> &obj);
```

Description

Pushes a double buffered CSPX Object onto the parameter block. This must be a group of objects of the same size as the process group. Each object will be passed to the corresponding process in the process group.

Parameters

`obj`: the object to be pushed on to the parameter block

Returns

A handle which can be used in other functions that reference this parameter.

push

Prototype

```
unsigned int push(  
    const CSPX::IStream &obj,  
    unsigned int alignment  
);
```

Description

Pushes an input stream onto the parameter block. The **C**" function will be passed the corresponding pipe in this parameter position.

Parameters

`obj`: the stream object to be pushed on to the parameter block

`alignment`: the desired alignment of the parameter

Returns

A handle which can be used in other functions that reference this parameter.

push

Prototype

```
unsigned int push(  
                const CSPX::OStream &obj,  
                unsigned int alignment  
                );
```

Description

Pushes an output stream onto the parameter block. The **C**" function will be passed the corresponding pipe in this parameter position.

Parameters

obj: the stream object to be pushed on to the parameter block
alignment: the desired alignment of the parameter

Returns

A handle which can be used in other functions that reference this parameter.

push

Prototype

```
unsigned int push(  
                const CSPX::IOStream &obj,  
                unsigned int alignment  
                );
```

Description

Pushes an input-output stream onto the parameter block. The **C**" function will be passed a pair of pipes corresponding to this parameter: the input pipe and then the output pipe.

Parameters

obj: the stream object to be pushed on to the parameter block
alignment: the desired alignment of the parameter

Returns

A handle which can be used in other functions that reference this parameter.

8.4 Double buffered objects

```
#include <CSPXDoubleObject.h>
```

This defines a double-buffered variant of the CSPX object.

8.4.1 Classes

CSPX::DoubleObject

This class defines the C++ data type which implements double buffering of objects. This corresponds to `CSPXDoubleObject` (see [Section 9.5: Double buffered objects on page 130](#)). Due to the nature of the C++ implementation the `CSPX::DoubleObject` is more like a *group* of objects (see [Section 9.3: Process groups on page 120](#)).

When declaring a `CSPX::DoubleObject` you must include the base type of the data being attached as a template argument. Optionally, you can also specify a `PartitionPolicy` template argument to specify how the data will be partitioned amongst the objects in the group (this defaults to `CSPX::DefaultPartitionPolicy`). See [Section 8.7: Partition policies on page 89](#) for more information on partition policies

8.4.2 Member functions

DoubleObject

Prototype

```
DoubleObject(
    T* data_1,
    T* data_2,
    size_t size,
    unsigned int num_processes,
    CSPXObjectAttribute attribute
);
```

Description

The `DoubleObject` constructor creates an object or group of objects and attaches the data to them according to the partition policy for the object.

Parameters

`data_1`: a pointer to the first data, or data buffer, to be attached

`data_2`: a pointer to the second data, or data buffer, to be attached

`size`: the total size of the data: the number of items of type `T`

`num_processes`: the number of processes the data is to be distributed to

`attribute`: the read/write attribute of the data (see [Section 9.2: Objects on page 112](#))

begin

Prototype

```
iterator begin(void);
```

Description

Returns an iterator referring to the first element in the object group.

Parameters

None.

Returns

An iterator over the object group.

end

Prototype

```
iterator end(void);
```

Description

An iterator to the element past the end of the sequence.

Parameters

None.

Returns

An iterator over the object group.

partitionCardinality

Prototype

```
size_t partitionCardinality(unsigned int index);
```

Description

Returns the number of data items in the partition specified by the index.

Parameters

index: the partition number, starting from 0

Throws

CSPX::ExceptionOutOfRange

Returns

The number of data items in the partition.

partitionSize

Prototype

```
size_t partitionSize(unsigned int index);
```

Description

Returns the size (in bytes) of the partition specified by the index.

Parameters

index: the partition number, starting from 0

Throws

CSPX::ExceptionOutOfRange

Returns

The size, in bytes, of the data in the partition.

purge

Prototype

```
void purge(void);
```

Description

Releases all resources allocated to this CSPX::Object. The function calls the CSPXObject_detach and CSPXObject_deinit functions for all CSPXObject objects

contained in the `CSPX::Object`. See [Section 9.2: Objects on page 112](#) for more details of these two functions.

Parameters

None.

Returns

None.

8.5 Connection policies

```
#include <CSPXConnectionPolicy.h>
```

This header defines policies for how the host will connect to tolerators. You can define your own variants to achieve the desired behavior.

8.5.1 Classes

CSPX::DefaultConnectionPolicy

This is the default connection policy which will connect to hardware or a simulator if the environment variable `CSAPI_CONNECT_SIM` is set (see [Section 7.5: Running code on a simulator on page 61](#) for details).

CSPX::SimulatorConnectionPolicy

This connection policy will connect to a simulator.

8.5.2 Member functions

connect

Prototype

```
CSAPIErrno connect(  
    struct CSAPIState* state,  
    unsigned int instance  
);
```

Description

Attempts to connect to an accelerator as defined by the policy. If the connection is successful the CSAPI state is initialized. See the ClearSpeed Runtime User Guide for more information about the CSAPI interface.

Parameters

`state`: a pointer to a CSAPI state structure to be used for the connection.
`instance`: the instance number of the hardware or simulator to connect to, instance numbers start from 0.

Throws

`CSPX::ExceptionConnectionFailed`

Returns

Zero for successful connection or a CSAPI error coded.

8.6 Distribution policies

```
#include <CSPXDistributionPolicy.h>
```

This header defines a number of policies that are used by the C++ CSPX interface. The distribution policy allows you to control how processors are used in the system.

8.6.1 Classes

CSPX::DistributionPolicyBase

The base class that all distribution policies are derived from. You can derive your own distribution policies as long as they provide the same static member functions.

CSPX::DefaultDistributionPolicy

This is the default policy. This will always attempt to connect to one processor core.

CSPX::MaximumDistributionPolicy

This policy connects to the maximum number of accelerator cores available in the system.

8.6.2 Member functions

availableProcessors

Prototype

```
unsigned int availableProcessors(struct CSAPIState* state)
```

Description

Returns the number of available processor cores in a CSAPI connection. The default implementation of this function calls the `CSAPI_num_processors` function.

Note: This will only recognize, and count, physical processors in the system. It will not find all simulator instances.

Parameters

`state`: a pointer to a CSAPI state structure for the connection

Throws

`CSPX::ExceptionNumProcessorsFailed`

Returns

The number of processor cores connected to by the CSAPI state.

numBoards

Prototype

```
unsigned int numBoards(void);
```

Description

Returns the number of accelerator cards which will be used by the policy. For the default distribution policy this returns 1. For the maximum distribution policy this returns the number of available accelerator cards.

Note: This will only recognize, and count, physical cards in the system. It will not find all simulator instances.

Parameters

None.

Throws

```
CSPX::ExceptionNumCardsFailed
```

Returns

The number of accelerator cards which will be used by the policy.

numProcessors

Prototype

```
unsigned int numProcessors(void);
```

Description

Returns the number of accelerator processor cores that the distribution policy will use on each card. For the default distribution policy this returns 1. For the maximum distribution policy this returns the number of cores on each card.

Parameters

None.

Throws

```
CSPX::ExceptionNumCardsFailed
```

Returns

The number of cores per card to be used by the policy.

8.7 Partition policies

```
#include <CSPXPartitionPolicy.h>
```

This header defines policies for how data will be partitioned amongst the processes. Again, you can define your own variants to achieve the desired behavior.

8.7.1 Classes

CSPX::DefaultPartitionPolicy

This is the default partition policy. Data is divided up into equal sized chunks. Any remainder is included in the final chunk.

CSPX::BroadcastPartitionPolicy

This policy sends the same data to all processes.

8.7.2 Member functions

partitionBase

Prototype

```
T* partitionBase(  
    T* base,  
    size_t total_size,  
    unsigned int num_processes,  
    unsigned int index  
);
```

Description

Returns a pointer to the start of the data for the partition specified by the index.

Parameters

base: the start address of the data to be partitioned
total_size: the total size of the data: number of data items of type T
num_processes: the number of processes the data is to be partitioned across
index: the index of the partition

Returns

A pointer to the data items in the specified partition.

partitionCardinality

Prototype

```
size_t partitionCardinality(  
    size_t total_size,  
    unsigned int num_processes,  
    unsigned int index  
);
```

Description

Returns the number of data items of type T in the partition specified by the index.

Parameters

`total_size`: the total size of the data: number of data items of type T
`num_processes`: the number of processes the data is to be partitioned across
`index`: the index of the partition

Returns

The number of data items in the specified partition.

partitionSize

Prototype

```
size_t partitionSize(  
    size_t total_size,  
    unsigned int num_processes,  
    unsigned int index  
);
```

Description

Returns the size (in bytes) of the partition specified by index.

Parameters

`total_size`: the total size of the data: number of data items of type T
`num_processes`: the number of processes the data is to be partitioned across
`index`: the index of the partition

Returns

The size, in bytes, of the data in the specified partition.

8.8 Streams

```
#include <CSPXStream.h>
```

Stream objects implement a streaming data interface between the host and accelerators. These are modelled on the `IOStream` class in the standard template library (STL).

A stream can be used with multiple accelerator processes. By default, output to a stream broadcasts the data to all the processes and input from a stream receives data from each process in turn. A stream can also be set to communicate with a single process.

Streams are built on top of pipes. On the card side, the stream appears as a pipe. A stream can be passed to an accelerator function call by using `CSPX::Parameter::push`, this will pass the corresponding pipe to the `C` function. The individual pipes underlying a stream can be accessed using iterators.

8.8.1 Classes

CSPX::OStream

This class implements an output stream, from host to accelerators.

CSPX::IStream

This class implements an input stream, from accelerators to host.

CSPX::IOStream

This class implements an input-output stream, between the host and the accelerators. This is implemented using two unidirectional pipes.

CSPX::BaseStream

The base class for all the stream classes.

8.8.2 Member functions

OStream

Prototype

```
OStream(  
    CSPX::State<T,U> &state,  
    size_t pipe_depth = 4096  
);
```

Description

Creates a pipe for each accelerator process and the corresponding output stream object.

Parameters

`state`: the CSPX:State object the stream will be associated with
`pipe_depth`: size of the buffer used in the pipe (optional)

Throws

CSPX::ExceptionInvalidState
CSPX::ExceptionPipeCreationFailed

IStream

Prototype

```
IStream(  
    CSPX::State<T,U> &state,  
    size_t pipe_depth = 4096  
);
```

Description

Creates a pipe for each accelerator process and the corresponding input stream object.

Parameters

`state`: the CSPX:State object the stream will be associated with
`pipe_depth`: size of the buffer used in the pipe (optional)

Throws

CSPX::ExceptionInvalidState
CSPX::ExceptionPipeCreationFailed

IOStream

Prototype

```
IOStream(  
    CSPX::State<T,U> &state,  
    size_t pipe_depth = 4096  
);
```

Description

Creates a pair of pipes for each accelerator process and the corresponding input-output stream object.

Parameters

`state`: the `CSPX::State` object the stream will be associated with
`pipe_depth`: size of the buffer used in the pipes (optional)

Throws

`CSPX::ExceptionInvalidState`
`CSPX::ExceptionPipeCreationFailed`

operator<<

operator>>

Operators for performing input (>>) and output (<<) on streams. Any data type can be used with streams, including the manipulator functions described below. The `IOStream` class implements both of these operators, the other classes only support one.

begin

Prototype

```
iterator begin(void);  
const_iterator begin(void);
```

Description

Returns an iterator referring to the first pipe in the stream object.

Parameters

None.

Returns

An iterator over the stream.

end

Prototype

```
iterator end(void);
```

Description

Returns an iterator referring to the last pipe in the stream object.

Parameters

None.

Returns

An iterator over the stream.

setProcess

Prototype

```
void setProcess(int index);
```

Description

Set the stream to communicate with a single process.

Parameters

`index`: the index of the process that the stream should communicate with

Returns

Nothing.

setProcess

Prototype

```
void setProcess(void);
```

Description

Reset the stream to the default communication model: broadcast for an output stream, round robin for an input stream.

Parameters

None.

Returns

Nothing.

8.9 Asynchronous operation

```
#include <CSPXHandler.h>
```

The CSPX handler class provides support for performing tasks asynchronously, such as transferring data while calling a function on the accelerators.

8.9.1 Classes

CSPX::Handler

This is an abstract class. You must derive your own class in order to execute functions in a separate thread. This can be used, for example, to create a thread to run code in parallel with functions being called on the accelerator.

Extra functions and state can be added derived class, and initialized in a constructor, in order to pass data to and from the asynchronous thread.

8.9.2 Member functions

Handler

Prototype

```
Handler(void);
```

Description

The default constructor.

Parameters

None.

Throws

```
CSPX::ExceptionPthreadFailure
```

doWork

Prototype

```
void doWork(void);
```

Description

Entry point for code to be run in a separate thread. This function can be overridden in the derived class to perform the asynchronous task.

The `doWork` function is called in a separate thread when the `startThread` member function is called. This is not intended to be called by the programmer directly.

Parameters

None.

Returns

Nothing

`isTerminated`

Prototype

```
bool isTerminated(void);
```

Description

Checks if the thread created by the handler has terminated.

Parameters

None.

Throws

```
CSPX::ExceptionPthreadFailure
```

Returns

True if the value has terminated, false otherwise.

`startHandler`

Prototype

```
int startHandler(void);
```

Description

Starts a thread running and invokes `doWork` on that thread.

Parameters

None.

Throws

```
CSPX::ExceptionPthreadFailure
```

Returns

Zero if the thread is created or nonzero on error.

terminateThread

Prototype

```
void terminateThread(void);
```

Description

Terminates the thread created by the handler.

Parameters

None.

Throws

```
CSPX::ExceptionPthreadFailure
```

Returns

Nothing.

waitHandler

Prototype

```
int waitHandler(void);
```

Description

This function will block until the handler thread finishes.

Parameters

None.

Throws

```
CSPX::ExceptionPthreadFailure
```

Returns

The exit code from the thread.

8.10 Semaphores

```
#include <CSPXSemaphore.h>
```

The CSPX semaphore class provides support for allocating, signalling and waiting on semaphores. Semaphores can be used to synchronize host and accelerator operations. The functions can also be used to allocate semaphores purely for use on the accelerators.

8.10.1 Classes

CSPX::Semaphore

This class manages semaphores for a set of processes. One semaphore is allocated for each accelerator process.

8.10.2 Member functions

Semaphore

Prototype

```
Semaphore(  
    CSPX::State<T,U> &state,  
    char *symbol_name=NULL  
);
```

Description

Allocates a semaphore for each process in the CSPX state. The semaphore will be assigned to the specified variable on the accelerator. If the variable name is `NULL` then the semaphore will not be passed to the accelerator code.

Parameters

`state`: the `CSPX::State` object to be allocated semaphores.

`symbol_name`: name of the static global variable in the CSX program to be assigned the allocated semaphore

Throws

`CSPX::ExceptionPthreadFailure`

`CSPX::ExceptionGetProcessFailed`

`CSPX::ExceptionSemaphoreAllocFailed`

getSemaphore

Prototype

```
CSAPISemaphore * getSemaphore(unsigned int index);
```

Description

Gets the semaphore allocated to the process specified by the index.

Parameters

index: the index of the CSPX process.

Throws

CSPX::ExceptionOutOfRange

Returns

The semaphore allocated to the process.

signal

Prototype

```
void signal(const CSPX::State<T,U> &state);
```

Description

Signals the semaphores on each process in the CSPX state.

Parameters

state: the CSPX::State object to signal.

Throws

CSPX::ExceptionSemaphoreSignalFailed

Returns

Nothing

wait

Prototype

```
void wait(  
    const CSPX::State<T,U> &state,  
    unsigned int timeout = CSAPI_NO_TIMEOUT  
);
```

Description

Waits on all of the semaphores in the CSPX state. The function returns when all of the semaphores have been signalled by the corresponding accelerator process or the wait times out.

Parameters

state: the `CSPX::State` object to wait on.

timeout: the timeout period, in milliseconds. The default is not to time out.

None.

Throws

`CSPX::ExceptionSemaphoreWaitFailed`

Returns

Nothing.

8.11 Reduction functions

```
#include <csp_x_reductions.h>
```

This header defines reduction functions which are used in other parts of the CSPX interface. These are provided as a starting point only. You can define extensions to these as long as they take the form:

```
void *CSPX_type_op(void * op1, void * op2);
```

Where *type* is the *type* of operands and *op* is the reduction operation. The function should use the first operand for the result of the operation and return a pointer to this.

8.11.1 Functions

CSPX_char_sum

Prototype

```
void * CSPX_char_sum(void * op1, void *op2);
```

Description

Sum of char values. The result overwrites the first argument.

Parameters

- op1: a pointer to the first operand, this is also used to store the result and so will be overwritten
- op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in *op1*

CSPX_double_sum

Prototype

```
void * CSPX_double_sum(void * op1, void *op2);
```

Description

Sum of double values. The result overwrites the first argument.

Parameters

- op1: a pointer to the first operand, this is also used to store the result and so will be overwritten
- op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in *op1*

CSPX_float_sum

Prototype

```
void * CSPX_float_sum(void * op1, void *op2);
```

Description

Sum of float values. The result overwrites the first argument.

Parameters

op1: a pointer to the first operand, this is also used to store the result and so will be overwritten

op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in op1

CSPX_int_sum

Prototype

```
void * CSPX_int_sum(void * op1, void *op2);
```

Description

Sum of integer values. The result overwrites the first argument.

Parameters

op1: a pointer to the first operand, this is also used to store the result and so will be overwritten

op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in op1

CSPX_short_sum

Prototype

```
void * CSPX_double_sum(void * op1, void *op2);
```

Description

Sum of short values. The result overwrites the first argument.

Parameters

op1: a pointer to the first operand, this is also used to store the result and so will be overwritten

op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in op1

CSPX_unsigned_char_sum

Prototype

```
void * CSPX_unsigned_char_sum(void * op1, void *op2);
```

Description

Sum of unsigned char values. The result overwrites the first argument.

Parameters

op1: a pointer to the first operand, this is also used to store the result and so will be overwritten

op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in op1

CSPX_unsigned_int_sum

Prototype

```
void * CSPX_unsigned_int_sum(void * op1, void *op2);
```

Description

Sum of unsigned integer values. The result overwrites the first argument.

Parameters

op1: a pointer to the first operand, this is also used to store the result and so will be overwritten

op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in op1

CSPX_unsigned_short_sum

Prototype

```
void * CSPX_unsigned_short_sum(void * op1, void *op2);
```

Description

Sum of unsigned short values. The result overwrites the first argument.

Parameters

op1: a pointer to the first operand, this is also used to store the result and so will be overwritten

op2: a pointer to the second operand

Returns

A pointer to the result of the reduction in op1

9 C API reference

This chapter describes the C interface to CSPX.

9.1 Processes

```
#include <csp_x_process.h>
```

This defines the `CSPXProcess` functions and types. These are used to implement the process model that forms the basis of CSPX.

9.1.1 Types

CSPXProcess

A “handle” to uniquely identify processes on both the host and the accelerators. A handle is generated for each process created on the accelerators. The host process passes its handle as a parameter to functions called on the accelerator.

CSPXProcessMoveInParamIterator

The type of the function pointer used to iterate over parameters passed to an accelerator process (see [CSPX_process_wait on page 111](#) and [CSPX_process_group_call on page 120](#)). The function prototype is:

```
int iterator(CSPXProcess process, void* data);
```

The function is called when object parameters are moved to an accelerator and will be passed a `CSPXProcess` handle for the called process and a `void*` that points to the data attached to the object.

CSPXProcessMoveOutParamIterator

The type of the function used to iterate over parameters used to return values from a function on an accelerator (see [CSPX_process_call_no_wait on page 106](#) and [CSPX_process_group_call on page 120](#)). The function prototype is:

```
int iterator(CSPXProcess process, void* data);
```

The function is called when object parameters are returned from an accelerator and will be passed a `CSPXProcess` handle for the called process and a `void*` that points to the data attached to the object.

struct CSPXRpcCallParameters

A structure used to pass parameters to and from the accelerator.

9.1.2 Functions

CSPX_process_call

Prototype

```
CSPXErrno CSPX_process_call(  
    CSPXProcess p,  
    const char * function_name,  
    void * const param_ptr,  
    size_t sizeo,  
    int * const result  
);
```

Description

Calls a function on an accelerator. This is the basic CSPX remote function call mechanism. This call blocks until the remote function returns.

The contents of the parameter block are user defined and must correspond on the host and accelerator side (usually done by having a common structure definition used by both the host and accelerator).

Parameters

`p`: the accelerator process to call
`function_name`: the function name to call on the process
`param_ptr`: a pointer to the parameters passed to the function
`size`: the size of the parameter block
`result`: the return code from the called function

Returns

An error code indicating the result of the operation.

CSPX_process_call_no_wait

Prototype

```
CSPXErrno CSPX_process_call_no_wait(  
    CSPXProcess p,  
    const char * function_name,  
    CSPXObject* param_obj,  
    struct CSPXRpcCallParameters *params,  
    CSPXProcessMoveOutParamIterator move_out  
);
```

Description

Calls a function on an accelerator. This function is asynchronous and returns immediately. The host can wait for the function to return by calling `CSPX_process_wait` (see [page 111](#)). Only one function call can be waiting at any time.

Parameters

`p`: the CSPX process
`function_name`: the name of the function to call
`params`: parameters required for the function call
`move_out`: a pointer to a function to be applied to each parameter passed to the function

Returns

An error code indicating the result of the operation.

CSPX_process_create

Prototype

```
CSPXProcess CSPX_process_create(  
    const char * program,  
    int argc,  
    char * argv[],  
    CSPXErrno *error_code  
);
```

Description

Creates a process on one accelerator.

It is possible to create a CSPX process on a simulator if hardware is not available. See [Section 7.5: Running code on a simulator on page 61](#) for details.

Parameters

`program`: the name of the CSX executable file to load on to the accelerator. This can be an absolute or relative path name. If this is a relative path name then the `CSPATH` environment variable is used to search for the file.
`argc`: the number of arguments in the `argv` array
`argv`: an array of argument values to be passed to the loaded CSX program
`error_code`: an error code indicating the result of the operation

Returns

The process or NULL in case of error.

CSPX_process_get_csapi_handle

Prototype

```
struct CSAPIState *CSPX_process_get_csapi_process(  
    CSPXProcess p,  
    CSPXErrno *error_code  
);
```

Description

Returns the CSAPI state structure for a CSPX process. This allows you to use CSAPI calls on running processes.

Parameters

`p`: the CSPX process handle

`error_code`: an error code indicating the result of the operation

Returns

A pointer to the CSAPI state structure for the process or NULL in case of error.

CSPX_process_get_csapi_process

Prototype

```
struct CSAPIProcess *CSPX_process_get_csapi_process(  
    CSPXProcess p,  
    CSPXErrno *error_code  
);
```

Description

Return the CSAPI process handle for a CSPX process. This allows you to use CSAPI calls on running processes.

Parameters

`p`: the CSPX process handle

`error_code`: an error code indicating the result of the operation

Returns

A pointer to the CSAPI process structure for the process or NULL in case of error.

CSPX_process_get_current

Prototype

```
CSPXProcess CSPX_process_get_current(CSPXErrno *error_code);
```

Description

Provides the handle of the current process.

Parameters

`error_code`: an error code indicating the result of the operation

Returns

Returns the CSPX handle of the process that called the function.

CSPX_process_get_index

Prototype

```
CSPXErrno CSPX_process_get_index(  
    CSPXProcess p,  
    int *index  
);
```

Description

Return the processor index for a CSPX process. This allows you to use CSAPI calls on running processes.

Parameters

`p`: the CSPX process handle

`index`: the index number of the processor that the process is running on

Returns

An error code indicating the result of the operation.

CSPX_process_load

Prototype

```
CSPXProcess CSPX_process_load(  
    struct CSAPIState * card,  
    unsigned proc_number,  
    const char * program,  
    int argc,  
    char * argv[],  
    CSPXErrno *error_code  
);
```

Description

This function loads and runs a program on an accelerator.

Parameters

`card`: a CSPAI state structure for the accelerator, created using the `CSAPI_new` and `CSAPI_connect` functions.

`proc_number`: the CSX core on the accelerator card. This is the processor index used by CSAPI. CSX cores are numbered from 0.

`program`: the name of the CSX executable file to load on to the accelerator. This can be an absolute or relative path name. If this is a relative path name then the `CSPATH` environment variable is used to search for the file.

`argc`: the number of arguments in the `argv` array

`argv`: an array of argument values to be passed to the loaded CSX program

`error_code`: an error code indicating the result of the operation

Returns

The processes handle or NULL in case of error.

CSPX_process_unload

Prototype

```
CSPXErrno CSPX_process_unload(CSPXProcess p);
```

Description

Cleans up any state created for the process. Must be invoked when a process has finished.

Parameters

`p`: the CSPX process handle

Returns

An error code indicating the result of the operation.

CSPX_process_wait

Prototype

```
CSPXErrno CSPX_process_wait(  
    CSPXProcess p,  
    CSPXProcessMoveInParamIterator move_in  
);
```

Description

Waits for an asynchronous accelerator function call to return.

Parameters

`p`: the CSPX process handle

`move_in`: a pointer to a function to be applied to each parameter used to return a value

Returns

An error code indicating the result of the operation.

CSPX_set_csvprof_filename

Prototype

```
CSPXErrno CSPX_set_csvprof_filename(const char * trace_file_name);
```

Description

Sets the name of the file to be used for profiling output.

Parameters

`trace_file_name`: the file name to be used

Returns

An error code indicating the result of the operation.

9.2 Objects

```
#include <csp_x_object.h>
```

This defines the functions and types used to implement the `CSPXObject` interface. Along with `CSPXProcess` this forms the basis for the CSPX system.

9.2.1 Constants

`CSPX_MAX_OBJECTS`

Defines the maximum number of objects that can exist. Space for CSPX objects is statically allocated.

9.2.2 Types

`CSPXObject`

This type provides a handle which can be used to uniquely identify objects across the host and accelerators.

`CSPXObjectAttribute`

This is an enumeration which defines the attributes of an object. It is used as parameter to a number of `CSPXObject` functions.

The values are:

`CSPX_OBJECT_READ`: the object is read only (from the perspective of the creating process)

`CSPX_OBJECT_WRITE`: the object is write only

`CSPX_OBJECT_READ_WRITE`: the object is read and written by the creating process

`CSPX_OBJECT_UNDEFINED`: uninitialised object state. This should not be used by explicitly

9.2.3 Functions

CSPX_object_attach

Prototype

```
CSPXErrno CSPX_object_attach(  
    CSPXObject *obj,  
    void *addr,  
    size_t size,  
    CSPXObjectAttribute attr  
);
```

Description

Attaches data to an initialized object and sets the attributes. The function takes an object and a pointer to the data to be attached.

Parameters

`obj`: a pointer to the object to which the data is to be attached
`addr`: a pointer to the data to be attached
`size`: the size of the data
`attr`: the attributes of the data (read or write)

Returns

An error code indicating the result of the operation.

CSPX_object_delete

Prototype

```
CSPXErrno CSPX_object_delete(CSPXObject* obj);
```

Description

Releases all resources that have been allocated to an object. The object can only be deleted by the process that created it and must be resident in that process. This is the opposite of the `CSPX_object_new` function. Any attached data must be detached from the object (using `CSPX_object_detach`) prior to calling `CSPX_object_delete`.

Parameters

`obj`: the object to be deleted

Returns

An error code indicating the result of the operation.

CSPX_object_detach

Prototype

```
CSPXErrno CSPX_object_detach(CSPXObject *obj);
```

Description

This function breaks the connection between data and an object. The data can only be detached by the process which originally attached it and the object must be resident in that process. Data must be detached from an object before `CSPX_object_delete` is called.

Parameters

`obj`: a pointer to the object from which data is to be detached

Returns

An error code indicating the result of the operation.

CSPX_object_get_attr

Prototype

```
CSPXErrno CSPX_object_get_attr(  
    CSPXObject *obj,  
    CSPXObjectAttribute attr  
);
```

Description

Returns the attribute of an object. See [CSPXObjectAttribute on page 112](#).

Parameters

`obj`: a pointer to the object

`attr`: the object's attribute

Returns

An error code indicating the result of the operation.

CSPX_object_get_index

Prototype

```
CSPXErrno CSPX_object_get_index(  
    CSPXObject *obj,  
    int *index  
);
```

Description

Returns the internal object index. This is primarily of use for debugging purposes and shouldn't normally be needed.

Parameters

obj: a pointer to the object
index: the index value for the object

Returns

An error code indicating the result of the operation.

CSPX_object_get_name**Prototype**

```
char *CSPX_object_get_name(  
    CSPXObject *obj,  
    CSPXErrno *error_code  
);
```

Description

Returns the name of the object. If a name has not been specified by the user then the function will return a default name which is the file and line number where the object was originally initialized.

Parameters

obj: a pointer to the object
error_code: an error code indicating the result of the operation

Returns

A pointer to the string. In the event of an error, the function returns NULL.

CSPX_object_get_pointer**Prototype**

```
void *CSPX_object_get_pointer(  
    CSPXObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Gets a pointer to the data attached to an object.

Parameters

obj: a pointer to the object
error_code: an error code indicating the result of the operation

Returns

The local address of the data attached to an object. If the object is not resident, the function returns `NULL`.

CSPX_object_get_size

Prototype

```
CSPXErrno CSPX_object_get_size(  
    CSPXObject* obj,  
    size_t size  
);
```

Description

Gets the size of the data attached to the object.

Parameters

`obj`: a pointer to the object

`size`: the size of the data attached to the object.

Returns

An error code indicating the result of the operation.

CSPX_object_isnull

Prototype

```
int CSPX_object_isnull(CSPXObject * obj);
```

Description

Tests whether data is attached to an object.

Parameters

`obj`: a pointer to the object

Returns

Returns 1 if the object has no attached data or otherwise 0.

Note: This function may cause a segmentation fault if passed an invalid object pointer.

CSPX_object_join

Prototype

```
CSPXErrno CSPX_object_join(  
    CSPXProcess dest,  
    CSPXObject* obj  
);
```

Description

Unifies the name space between the host and the accelerator process for the given object. The object must be declared with the same name on both the host and the accelerator. After it is initialized on the host (using `CSPX_object_new`, see [page 118](#)) the join function can be used to make the separate host and accelerator declarations refer to the same object. An object can only be joined between one host and one accelerator process.

Parameters

`dest`: the handle of the process to join the object to
`obj`: a pointer to the host object

Returns

An error code indicating the result of the operation.

CSPX_object_join_named

Prototype

```
CSPXErrno CSPX_object_join_named(  
    CSPXProcess dest,  
    CSPXObject obj,  
    char* name  
);
```

Description

Joins a host and an accelerator object. The name of the object on the accelerator process is passed as a parameter. After the object is initialized on the host (using `CSPX_object_new`, see [page 118](#)) the join function can be used to make the separate host and accelerator declarations refer to the same object. An object can only be joined between one host and one accelerator process.

Parameters

`dest`: the handle of the process to join the object to
`obj`: a pointer to the host object
`name`: the name of the accelerator object to be joined

Returns

An error code indicating the result of the operation.

CSPX_object_move

Prototype

```
CSPXErrno CSPX_object_move(  
    CSPXProcess dest,  
    CSPXObject* obj  
);
```

Description

Moves an object to a target process. This performs the migration of the data attached to the object. The object must be resident in the process calling the function. Note that this operation is asynchronous and the function will return immediately.

Parameters

`dest`: the handle of the process to send the object to
`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_object_new

Prototype

```
CSPXErrno CSPX_object_new(CSPXObject* obj);
```

Description

Initializes an object and registers it with the object manager. All objects must be initialized before they are used.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_object_set_name

Prototype

```
CSPXErrno CSPX_object_set_name(CSPXObject *obj, char * name);
```

Description

Sets an optional name for the object. This is useful for debugging and profiling purposes.

Parameters

obj: a pointer to the object

name: a pointer to the string to use as the name

Returns

An error code indicating the result of the operation.

CSPX_object_sync**Prototype**

```
CSPXErrno CSPX_object_sync(CSPXObject* obj);
```

Description

Synchronizes with an object. This function will block until the object in question is resident in the calling process. If the object is already resident it will return immediately.

Parameters

obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_object_sync_pointer**Prototype**

```
void *CSPX_object_sync_pointer(  
    CSPXObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Synchronizes with an object and returns a pointer to the associated data. This function combines the `CSPX_object_sync` and `CSPX_object_get_pointer` functions.

Parameters

obj: a pointer to the object

error_code: an error code indicating the result of the operation

Returns

The local address of the attached data or `NULL` in error cases.

9.3 Process groups

```
#include <csp_x_process_group.h>
```

This is an extension to the basic CSPX functionality that provides a mechanism for creating collections of processes that are run on a number of accelerators. This is supported by the use of object groups with a one-to-one mapping between processes and objects in the group.

9.3.1 Types

CSPXProcessGroup

This is a unique handle for a group of processes

CSPXProcessGroupReturnFunction

Definition of the function pointer in `CSPX_process_group_get_return_code_reduced` (see [page 124](#)). This function has the form:

```
CSPXErrno function(int code, void * p);
```

The function is passed the return code. The parameter `p` is not currently used.

9.3.2 Constants

CSPX_ALL_AVAILABLE_PROCESSORS

A constant to request that `CSPX_process_group_create` use all the available accelerator cores to create the process group.

9.3.3 Functions

CSPX_process_group_call

Prototype

```
CSPXErrno CSPX_process_group_call(  
    CSPXProcessGroup ps,  
    const char *function_name,  
    CSPXProcessMoveOutParamIterator move_out,  
    CSPXProcessMoveInParamIterator move_in  
);
```

Description

Calls a function on every process in a process group. This is the equivalent to the standard `CSPX_process_call` function. When each function is called the `move_out` function is called on the parameters. The `move_in` function is called on the parameters when the

function returns. These are called on each parameter before moving the parameter to or from the other process. These are useful for carrying out deep copies of parameters. These should be NULL if not required.

Parameters

`ps`: the process group

`function_name`: the name of the function to call

`move_out`: a pointer to a function to be applied to each parameter passed to the function

`move_in`: a pointer to a function to be applied to each parameter used to return a value

Returns

An error code indicating the result of the operation.

CSPX_process_group_create

Prototype

```
CSPXProcessGroup CSPX_process_group_create(  
    int required_number,  
    const char * program,  
    int argc,  
    char * argv[],  
    CSPXErrno *error_code  
);
```

Description

Creates a process group with one process on each processor.

Parameters

`required_number`: the number of processes to be created. If this value is the constant `CSPX_ALL_AVAILABLE_PROCESSORS` then all available accelerators will be used to create the group.

`program`: the name of the CSX executable file to load on to the accelerator. This can be an absolute or relative path name. If this is a relative path name then the `CSPATH` environment variable is used to search for the file.

`argc`: the number of arguments in the `argv` array

`argv`: an array of argument values to be passed to the loaded CSX program

`error_code`: an error code indicating the result of the operation

Returns

The process group or NULL in case of error.

CSPX_process_group_free

Prototype

```
CSPXErrno CSPX_process_group_free(CSPXProcessGroup ps);
```

Description

Cleans up any state created for the process group.

Parameters

`ps`: the process group

Returns

An error code indicating the result of the operation.

CSPX_process_group_get_object_group

Prototype

```
CSPXObjectGroup CSPX_process_group_get_object_group(  
    CSPXProcessGroup ps,  
    const char * name,  
    CSPXErrno *error_code  
);
```

Description

Gets a named object group. A process group can have multiple object groups associated with it. This function returns the object group with a specified name.

Parameters

`ps`: the process group

`name`: the object group name to get

`error_code`: an error code indicating the result of the operation

Returns

The object group or NULL in case of error (for example, if there is no object group with the specified name).

CSPX_process_group_get_param_box

Prototype

```
CSPXObject* CSPX_process_group_get_param_box(  
    CSPXProcessGroup ps,  
    int index,  
    CSPXErrno *error_code  
);
```

Description

Returns a pointer to the object used as the parameter object for a specified process. This allows the parameters to be set up for each of the processes in the group.

Parameters

`ps`: the process group
`index`: the index number of the process in the group. The index starts from 0.
`error_code`: an error code indicating the result of the operation

Returns

The parameter object or in the event of an error (such as an index which is larger than the size of the group) NULL is returned.

CSPX_process_group_get_process

Prototype

```
CSPXProcess CSPX_process_group_get_process(  
    CSPXProcessGroup ps,  
    int index,  
    CSPXErrno *error_code  
);
```

Description

Gets a specified process from a process group.

Parameters

`ps`: the process group
`index`: the index number of the process in the group. The index starts from 0.
`error_code`: an error code indicating the result of the operation

Returns

The process or NULL in the event of an error.

CSPX_process_group_get_return_code

Prototype

```
CSPXErrno CSPX_process_group_get_return_code(  
    CSPXProcessGroup ps,  
    int index,  
    int *return_code  
);
```

Description

Gets the function return code for a specified process in a group.

Parameters

`ps`: the process group
`index`: the index number of the process in the group. The index starts from 0.
`return_code`: the return code for the function call on the specified process.

Returns

An error code indicating the result of the operation.

CSPX_process_group_get_return_code_reduced

Prototype

```
CSPXErrno CSPX_process_group_get_return_code_reduced(  
    CSPXProcessGroup ps,  
    CSPXProcessGroupReturnFunction return_function,  
    int *return_code  
);
```

Description

Reduces the return codes from all the functions calls on a group to a single value. This is done as the bitwise OR of all the return values. A function can be provided which is applied to each return value before the reduction.

Parameters

`ps`: the process group
`return_function`: the function to be applied to each function's return code. This can be NULL if no function is to be applied.
`return_code`: the bitwise OR of all the function return codes for the process group.

Returns

An error code indicating the result of the operation.

CSPX_process_group_get_size

Prototype

```
CSPXErrno CSPX_process_group_get_size(  
    CSPXProcessGroup ps,  
    int *size  
);
```

Description

Gets the number of processes in a group.

Parameters

`ps`: the process group
`size`: the number of processes in the group

Returns

An error code indicating the result of the operation.

9.4 Object groups

Object groups are provided to support the process group model. These are created with a one to one mapping between objects and processes.

```
#include <csp_x_process_group.h>
```

9.4.1 Types

CSPXObjectGroup

A handle for a group of objects used with a process group.

CSPXObjectGroupIterator

Definition of the function pointer used in `CSPX_object_group_apply_function` (see [page 126](#)). The function has the form:

```
CSPXErrno iterator(  
    int index,  
    CSPXObject* obj,  
    CSPXProcessGroup ps,  
    void* g  
);
```

The function is passed an integer `index` which corresponds to the position of the process in the process group `ps`. The function is passed a pointer to a CSPX object, `obj`, which is the corresponding object in the object group. Parameter `g` is not currently used.

9.4.2 Functions

CSPX_object_group_apply_function

Prototype

```
CSPXErrno CSPX_object_group_apply_function(  
    CSPXObjectGroup objects,  
    CSPXObjectGroupIterator function  
);
```

Description

Applies a function to each object in an object group.

Parameters

`objects`: the object group
`function`: the function to apply to each object

Returns

The OR of the return codes of all the function calls on the objects in the group.

CSPX_object_group_attach

Prototype

```
CSPXErrno CSPX_object_group_attach(  
    CSPXObjectGroup *gs,  
    void *addr,  
    size_t slice_size,  
    CSPXObjectAttribute attr  
);
```

Description

Attaches data to an object group and sets the attributes. The function takes an object and a pointer to the data to be attached. The data is divided equally between all the objects.

Parameters

`gs`: the group of objects to which the data is to be attached
`addr`: a pointer to the data to be attached
`size`: the size of the data to be attached to each object
`attr`: the attributes of the data (read or write)

Returns

An error code indicating the result of the operation.

CSPX_object_group_create

Prototype

```
CSPXObjectGroup CSPX_object_group_create(  
    CSPXProcessGroup ps,  
    const char * name,  
    int * index,  
    CSPXErrno *error_code  
);
```

Description

This function will create a group of CSPX objects for a process group, one object per process.

Parameters

`ps`: the process group for which the objects are to be created
`name`: the name to be assigned to the objects
`index`: not currently used
`error_code`: an error code indicating the result of the operation

Returns

The object group or NULL in case of error.

CSPX_object_group_join_named

Prototype

```
CSPXErrno CSPX_object_group_join_named(  
    CSPXObject obj,  
    char* name  
);
```

Description

Joins a host and an accelerator object. The name of the object on the accelerator process is passed as a parameter. After the object group is created on the host the join function can be used to make the separate host and accelerator declarations refer to the same object.

Parameters

obj: a pointer to the host object group
name: the name of the accelerator object to be joined

Returns

An error code indicating the result of the operation.

CSPX_object_group_move

Prototype

```
CSPXErrno CSPX_object_group_move(CSPXObjectGroup gs);
```

Description

Moves an object group to the associated processes. This performs the migration of the data attached to the object. The object must be resident in the host process calling the function. Note that this operation is asynchronous and the function will return immediately.

Parameters

gs: the object group to move

Returns

An error code indicating the result of the operation.

CSPX_object_group_sync

Prototype

```
CSPXErrno CSPX_object_group_sync(CSPXObjectGroup gs);
```

Description

Synchronizes with an object group. This function will block until all the object in the group are resident in the calling process. If the objects are already resident it will return immediately.

Parameters

`gs`: the object group to wait for

Returns

An error code indicating the result of the operation.

9.5 Double buffered objects

```
#include <csp_x_double_object.h>
```

Double buffered objects provide extra support for handling double-buffered transfers between host and accelerator. The double buffered object has two sets of data associated with it so that one can be processed while the other is being transferred. The object also maintains the current state: which set of data is currently being used.

The double buffered object, of type `CSPXDoubleObject`, contains status information and a handle to an “inner” object (of type `CSPXDoubleObjectHandle`) to which the two data buffers are attached.

9.5.1 Types

`CSPXDoubleObject`

The type for double buffered objects/

`CSPXDoubleObjectHandle`

The type of the object to which data is attached within a double buffered object.

9.5.2 Functions

`CSPX_double_object_attach`

Prototype

```
CSPXErrno CSPX_double_object_attach(  
    CSPXDoubleObject* obj,  
    void *addr_1,  
    void *addr_2,  
    size_t size,  
    CSPXObjectAttribute a  
);
```

Description

`obj`: a pointer to the object
`addr_1`: a pointer to the first data buffer
`addr_2`: a pointer to the second data buffer
`size`: the size of the data buffers
`a`: the attributes of the data

Parameters

Attaches two sets of data to an object. Both must be the same size and have the same attribute.

Returns

An error code indicating the result of the operation.

CSPX_double_object_delete**Prototype**

```
CSPXErrno CSPX_double_object_detach(CSPXDoubleObject* obj);
```

Description

Releases all resources that have been allocated to an object. The object can only be deleted by the process that created it and must be resident in that process. This is the opposite of the `CSPX_double_object_new` function. Any attached data must be detached from the object (using `CSPX_double_object_detach`) prior to calling `CSPX_double_object_delete`.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_detach**Prototype**

```
CSPXErrno CSPX_double_object_detach(CSPXDoubleObject* obj);
```

Description

Breaks the connection between data and an object. The data can only be detached by the process which originally attached it and the object must be resident in that process. Data must be detached from an object before `CSPX_object_delete` is called.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_get_handle

Prototype

```
CSPXErrno CSPX_double_object_get_handle(CSPXDoubleObject *obj);
```

Description

Gets the handle for the “inner” object that has the data buffers attached to it. This handle needs to be passed between the host and the accelerator using the double buffered object to allow them to share the buffers.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_get_name

Prototype

```
char *CSPX_double_object_get_name(  
    CSPXDoubleObject *obj,  
    CSPXErrno *error_code  
);
```

Description

Returns the name of the object. If a name has not been specified by the programmer then the function will return a default name which is the file and line number where the object was originally initialized.

Parameters

`obj`: a pointer to the object

`error_code`: an error code indicating the result of the operation

Returns

A pointer to the string. In the event of an error, the function returns `NULL`.

CSPX_double_object_get_pointer

Prototype

```
void *CSPX_double_object_get_pointer(  
    CSPXDoubleObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Gets a pointer to the appropriate data buffer attached to the object.

Parameters

`obj`: a pointer to the object
`error_code`: an error code indicating the result of the operation

Returns

The local address of the data buffer. If the object is not resident, the function returns `NULL`.

CSPX_double_object_join

Prototype

```
CSPXErrno CSPX_double_object_join(  
    CSPXProcess dest,  
    CSPXDoubleObject* obj  
);
```

Description

Unifies the name space between the host and the accelerator process for the given double object. The object must be declared with the same name on both the host and the accelerator. After it is initialized on the host (using `CSPX_double_object_new`, see [page 135](#)) the join function can be used to make the separate host and accelerator declarations refer to the same object. An object can only be joined between one host and one accelerator process.

Parameters

`dest`: the handle of the process to join the object to
`obj`: a pointer to the host object

Returns

An error code indicating the result of the operation.

CSPX_double_object_join_named

Prototype

```
CSPXErrno CSPX_double_object_join_named(  
    CSPXProcess dest,  
    CSPXDoubleObject obj,  
    char* name  
);
```

Description

Joins a host and an accelerator double object. The name of the object on the accelerator process is passed as a parameter. After the object is initialized on the host (using `CSPX_object_new`, see [page 135](#)) the join function can be used to make the separate host and accelerator declarations refer to the same object. An object can only be joined between one host and one accelerator process.

Parameters

`dest`: the handle of the process to join the object to
`obj`: a pointer to the host object
`name`: the name of the accelerator object to be joined

Returns

An error code indicating the result of the operation.

CSPX_double_object_move

Prototype

```
CSPXErrno CSPX_double_object_move(  
    CSPXProcess dest,  
    CSPXDoubleObject* obj  
);
```

Description

Moves an object to a target process. This performs the migration of the appropriate data buffer attached to the object. Note that this operation is asynchronous and the function will return immediately.

Parameters

`dest`: the handle of the process to send the object to
`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_new

Prototype

```
CSPXErrno CSPX_double_object_new(CSPXDoubleObject* obj);
```

Description

Initializes a double buffered object and registers it with the object manager. All objects must be initialized before they are used.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_reattach

Prototype

```
CSPXErrno CSPX_double_object_reattach(  
    CSPXDoubleObject* obj,  
    size_t stride  
);
```

Description

Changes the pointer attached to the current buffer. Implicitly performs a detach before attaching the new pointer. This is used to allow the double buffer to slide over an array a bit like a stream.

Parameters

`obj`: a pointer to the object

`stride`: the offset to be added to the current object pointer

Returns

An error code indicating the result of the operation.

CSPX_double_object_set_name

Prototype

```
CSPXErrno CSPX_double_object_set_name(  
    CSPXDoubleObject *obj,  
    char * name  
);
```

Description

Sets an optional name for a object. This is useful for debugging and profiling purposes.

Parameters

obj: a pointer to the object
name: the string to be assigned as the name of the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_sync

Prototype

```
CSPXErrno CSPX_double_object_sync(CSPXDoubleObject* obj);
```

Description

Synchronizes with an object. This function will block until the object in question is resident in the calling process. If the object is already resident it will return immediately.

Parameters

obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_sync_pointer

Prototype

```
void * CSPX_double_object_sync_pointer(  
    CSPXDoubleObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Synchronizes with an object and returns a pointer to the associated data. This function combines the `CSPX_double_object_sync` and `CSPX_double_object_get_pointer` functions.

Parameters

obj: a pointer to the object
error_code: an error code indicating the result of the operation

Returns

The local address of the attached data or NULL in error cases.

9.6 Pipes

```
#include <csp_x_pipe.h>
```

CSPX provides a data pipe implementation to facilitate streaming data to and from accelerators. A CSPX pipe is similar to a Unix pipe where data can be written by one process (the *producer*) and read by another (the *consumer*). A pipe is unidirectional. One end must be the host process, the other being an accelerator.

9.6.1 Types

CSPXPipe

The type used to represent pipes on the host and accelerators.

9.6.2 Functions

CSPX_pipe_drain

Prototype

```
CSPXErrno CSPX_pipe_drain(CSPXPipe* pipe);
```

Description

Blocks until the pipe has drained. Can be called by a producer to wait for the consumer to read the entire contents of the pipe.

Parameters

`pipe`: a pointer to the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_get_index

Prototype

```
CSPXErrno CSPX_pipe_get_index(  
    CSPXPipe* pipe,  
    int *index  
);
```

Description

Returns the internal index of the pipe. This is primarily of use for debugging purposes and shouldn't normally be needed.

Parameters

`pipe`: a pointer to the pipe
`index`: the returned index value of the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_get_name

Prototype

```
char *CSPX_pipe_get_name(  
    CSPXPipe* pipe,  
    CSPXErrno *error_code  
);
```

Description

Returns the name of the pipe. If a name has not been set then the function will return a default name which is the file and line number where the object was originally initialized.

Parameters

`pipe`: a pointer to the pipe
`error_code`: an error code indicating the result of the operation.

Returns

A pointer to the string. In the event of an error, the function returns `NULL`.

CSPX_pipe_get_stats

Prototype

```
CSPXErrno CSPX_pipe_get_stats(  
    CSPXPipe* pipe,  
    struct CSPXPipePerf * perf_info  
);
```

Description

Returns performance information on the pipe such as the amount of data transferred and stall rate.

Parameters

`pipe`: a pointer to the pipe
`perf_info`: a pointer to a `CSPXPipePerf` structure which return the performance data

Returns

An error code indicating the result of the operation.

CSPX_pipe_new

Prototype

```
CSPXErrno CSPX_pipe_new(  
    CSPXPipe* pipe,  
    CSPXProcess p,  
    size_t buffer_depth,  
    CSPXObjectAttribute a  
);
```

Description

Initialize a pipe connecting two processes. The buffer depth is the depth of the buffering used within the pipe. The larger the buffer the more likely it is that the pipe can handle different patterns of producer/consumer rates without blocking.

Parameters

`pipe`: a pointer to the pipe
`p`: the process at the other end of the pipe
`buffer_depth`: the size of the internal buffer in the pipe
`a`: the attribute of the pipe (read or write)

Returns

An error code indicating the result of the operation.

CSPX_pipe_read

Prototype

```
CSPXErrno CSPX_pipe_read(  
    CSPXPipe* pipe,  
    void* data,  
    size_t bytes  
);
```

Description

Reads from a pipe. This will block until the specified number of bytes has been read.

Parameters

`pipe`: a pointer to the pipe
`data`: a pointer to the destination for the data
`bytes`: the number of bytes to read from the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_reader

Prototype

```
CSPXProcess CSPX_pipe_reader(  
    CSPXPipe* pipe  
    CSPXErrno *error_code  
);
```

Description

Returns the handle of the process reading from the pipe.

Parameters

`pipe`: a pointer to the pipe
`error_code`: an error code indicating the result of the operation

Returns

The process handle for the consumer or NULL on error.

CSPX_pipe_reset_stats

Prototype

```
CSPXErrno CSPX_pipe_reset_stats(CSPXPipe* pipe);
```

Description

Resets the performance statistics collected for a pipe.

Parameters

`pipe`: a pointer to the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_set_name

Prototype

```
CSPXErrno CSPX_pipe_set_name(  
    CSPXPipe *pipe,  
    char * name  
);
```

Description

Sets an optional name for the pipe. This is useful for debugging and profiling purposes.

Parameters

pipe: a pointer to the pipe

name: a pointer to the string to use as the name

Returns

An error code indicating the result of the operation.

CSPX_pipe_write**Prototype**

```
CSPXErrno CSPX_pipe_write(  
    CSPXPipe* pipe,  
    const void *data,  
    size_t bytes  
);
```

Description

Writes to a pipe. This will not block if there is sufficient buffering to take the data.

Parameters

pipe: a pointer to the pipe

data: a pointer to the source of the data

bytes: the number of bytes to write to the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_writer**Prototype**

```
CSPXProcess CSPX_pipe_writer(  
    CSPXPipe* pipe,  
    CSPXErrno *error_code  
);
```

Description

Returns the handle of the process reading from the pipe.

Parameters

pipe: a pointer to the pipe

error_code: an error code indicating the result of the operation

Returns

The process handle for the producer or NULL on error.

9.7 Error handling

```
#include <csp_x_errno.h>
```

These functions provide support for handling and reporting errors returned by CSPX functions.

9.7.1 Types

CSPXErrno

The `CSPXErrno` enumeration defines the set of values that can be returned to indicate an error. See [Section 7.1: Error codes on page 63](#) for details of the error codes and their meanings.

9.7.2 Functions

CSPX_get_error_string

Prototype

```
CSPXErrno CSPX_get_error_string(  
    struct CSAPIState* const s,  
    CSPXErrno error_code,  
    char * const error_string,  
    unsigned int max_string_length  
);
```

Description

Returns the error string associated with the error code. If the error code is a CSAPI error, then the `CSAPIState` parameter will be used to determine the cause of the error.

Parameters

`s`: the state created by `CSAPI_new`, can be NULL if a valid state is not available. You can use the `CSPX_process_get_csapi_handle` function to get this handle from a `CSPXProcess`

`error_code`: the error code

`error_string`: a pointer to a pre-allocated buffer to receive error string.

`max_string_length`: the maximum length of the error string buffer.

Returns

An error code indicating the result of the operation.

CSPX_get_error_string_length

Prototype

```
CSPXErrno CSPX_get_error_string_length(  
    struct CSAPIState* const s,  
    CSPXErrno error_code,  
    unsigned int max_string_length  
);
```

```
    struct CSAPIState* const s,  
    CSPXErrno error_code,  
    unsigned int* const length  
);
```

Description

Returns the length of the description string for the given error code. This allows a buffer to be allocated that will hold the error string returned by `CSPX_get_error_string`.

Parameters

`s`: the state created by `CSAPI_new`, can be NULL if a valid state is not available. You can use the `CSPX_process_get_csapi_handle` function to get this handle from a `CSPXProcess`.

`error_code`: the error code

`length`: the value to return the string length

Returns

An error code indicating the result of the operation.

10 Cⁿ API reference

This chapter defines the accelerator (Cⁿ) interface to CSPX.

10.1 Pragmas

10.1.1 RPC pragma

This pragma defines a function to be called directly from the host application. It can only be called from a C++ host program.

The pragma takes the form:

```
#pragma cspx_rpc options*

options :
    write( parameter+ )
    | read( parameter+ )
    | read_write( parameter+ )
    | reduce( reduce_param+ )
    | dist( distributionPolicy )

parameter :
    array_param[ size_param ]
    | scalar_param

reduce_param :
    reduction_function:reduce

reduce :
    parameter
    | return
```

The pragma applies to the immediately following function declaration.

The `read`, `write` and `read_write` options specify whether the function parameters are:

- only passed from the host to the function (`write`)
- only returned from the function to the host (`read`)
- or transferred to and from the function (`read_write`).

Parameters can be arrays or pointers, in which case they must be followed by a size parameter, or scalar values.

Scalar values which return a value (either `read` parameters or the function return value) must specify a reduction function or be represented as vectors in the host code. Scalar parameters defined as `read_write` values must have an associated reduction function.

The `reduce` option specifies the reduction function to be used or the specified parameter (or the function return value).

Reduction functions have names of the form *function_type* (for example, `CSPX_double_sum` for a sum function with a parameter of type double). The reduction functions can be one of those included in the CSPX C++ implementation (see [Section 8.11: Reduction functions on page 101](#)) or a user defined function.

10.1.2 Export pragma

The export pragma specifies a list of functions to be made available to a host program. This pragma can appear anywhere in the compilation unit containing the function definitions.

```
#pragma csp_x_export(function+)
```

For example, the following example will export the two function `foo` and `bar`:

```
#pragma csp_x_export(foo, bar)
```

The exported functions must be of the form:

```
int foo(CSPXProcess parent, void* const param);
```

10.2 Processes

```
#include <csp_x_process.h>
```

This defines the `CSPXProcess` functions and types. These are used to implement the process model that forms the basis of CSPX.

10.2.1 Types

CSPXProcess

This type defines a “handle” to uniquely identify processes. Processes can only be created by the host. The handle of the host process is passed as a parameter to functions called on the accelerator.

10.2.2 Functions

CSPX_dispatch

Prototype

```
CSPXErrno CSPX_dispatch(void);
```

Description

Handles the movement of the parameter objects between the host and the accelerator, and invokes the requested function.

Parameters

None.

Returns

An error code indicating the result of the operation. The dispatch loop should be terminated if the function returns anything other than `CSPXErrno_success`.

CSPX_process_get_current

Prototype

```
CSPXProcess CSPX_process_get_current(CSPXErrno *error_code);
```

Description

Provides the handle of the current process.

Parameters

`error_code`: an error code indicating the result of the operation

Returns

Returns the handle of the process that called the function.

CSPX_process_init

Prototype

```
CSPXErrno CSPX_process_get_current(void);
```

Description

Initializes the process on the accelerator.

Parameters

None.

Returns

An error code indicating the result of the operation.

CSPX_runtime

Prototype

```
CSPXErrno CSPX_runtime(void);
```

Description

This provides a handler for CSPX function calls by the host. It calls `CSPX_process_init` and then repeatedly calls `CSPX_dispatch` to handle the calls from the host. This will run until the dispatch function returns a nonzero value. This will usually occur under error conditions or when the host calls the `CSPX_process_unload` function.

Parameters

None.

Returns

An error code indicating the result of the operation.

10.3 Objects

```
#include <csp_x_object.h>
```

This defines the functions and types used to implement the `CSPXObject` interface. Along with `CSPXProcess` this forms the basis for the CSPX system.

10.3.1 Constants

`CSPX_MAX_OBJECTS`

Defines the maximum number of objects that can exist. Space for CSPX objects is statically allocated.

10.3.2 Types

`CSPXObject`

This type provides a handle which can be used to uniquely identify objects across the host and accelerators.

`CSPXObjectAttribute`

This is an enumeration which defines the attributes of an object. It is used as parameter to a number of `CSPXObject` functions.

The values are:

`CSPX_OBJECT_READ`: the object is read only (from the perspective of the creating process)

`CSPX_OBJECT_WRITE`: the object is write only

`CSPX_OBJECT_READ_WRITE`: the object is read and written by the creating process

`CSPX_OBJECT_UNDEFINED`: uninitialised object state. This should not be used by explicitly

10.3.3 Functions

`CSPX_object_get_attr`

Prototype

```
CSPXErrno CSPX_object_get_attr(  
    CSPXObject *obj,  
    CSPXObjectAttribute attr  
);
```

Description

This functions returns the attribute of the object. See [CSPXObjectAttribute on page 149](#).

Parameters

`obj`: a pointer to the object
`attr`: the object's attribute

Returns

An error code indicating the result of the operation.

CSPX_object_get_index

Prototype

```
CSPXErrno CSPX_object_get_index(  
    CSPXObject *obj,  
    int *index  
);
```

Description

Returns the internal object index. This is primarily of use for debugging purposes and shouldn't normally be needed.

Parameters

`obj`: a pointer to the object
`index`: the index value for the object

Returns

An error code indicating the result of the operation.

CSPX_object_get_name

Prototype

```
char *CSPX_object_get_name(  
    CSPXObject *obj,  
    CSPXErrno *error_code  
);
```

Returns the name of the object. If a name has not been specified by the user then the function will return a default name which is the file and line number where the object was originally initialized.

Parameters

`obj`: a pointer to the object
`error_code`: an error code indicating the result of the operation

Returns

A pointer to the string. In the event of an error, the function returns `NULL`.

CSPX_object_get_pointer

Prototype

```
void *CSPX_object_get_pointer(  
    CSPXObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Gets a pointer to the data attached to an object.

Parameters

`obj`: a pointer to the object

`error_code`: an error code indicating the result of the operation

Returns

The local address of the data attached to an object. If the object is not resident, the function returns NULL.

CSPX_object_get_size

Prototype

```
CSPXErrno CSPX_object_get_size(  
    CSPXObject* obj,  
    size_t size  
);
```

Description

Gets the size of the data attached to the object.

Parameters

`obj`: a pointer to the object

`size`: the size of the data attached to the object.

Returns

An error code indicating the result of the operation.

CSPX_object_isnull

Prototype

```
int CSPX_object_isnull(CSPXObject* obj);
```

Description

Tests whether data is attached to an object.

Parameters

obj: a pointer to the object

Returns

Returns 1 if the object has no attached data or otherwise 0.

CSPX_object_move**Prototype**

```
CSPXErrno CSPX_object_move(  
    CSPXProcess dest,  
    CSPXObject* obj  
);
```

Description

Moves an object to a target process. This performs the migration of the data attached to the object. The object must be resident in the process calling the function. Note that this operation is asynchronous and the function will return immediately.

Parameters

dest: the handle of the process to send the object to
obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_object_sync**Prototype**

```
CSPXErrno CSPX_object_sync(CSPXObject* obj);
```

Description

Synchronizes with an object. This function will block until the object in question is resident in the calling process. If the object is already resident it will return immediately.

Parameters

obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_object_sync_pointer

Prototype

```
void *CSPX_object_sync_pointer(  
    CSPXObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Synchronizes with an object and returns a pointer to the associated data. This function combines the `CSPX_object_sync` and `CSPX_object_get_pointer` functions.

Parameters

`obj`: a pointer to the object

`error_code`: an error code indicating the result of the operation

Returns

The local address of the attached data or `NULL` in error cases.

10.4 Double buffered objects

```
#include <csp_x_double_object.h>
```

Double buffered objects provide extra support for handling double-buffered transfers between host and accelerator. The double buffered object has two sets of data associated with it so that one can be processed while the other is being transferred. The object also maintains the current state: which set of data is currently being used.

The double buffered object, of type `CSPXDoubleObject`, contains status information and a handle to an “inner” object (of type `CSPXDoubleObjectHandle`) to which the two data buffers are attached.

10.4.1 Types

`CSPXDoubleObject`

The type for double buffered objects/

`CSPXDoubleObjectHandle`

The type of the object to which data is attached within a double buffered object.

10.4.2 Functions

`CSPX_double_object_delete`

Prototype

```
CSPXErrno CSPX_double_object_detach(CSPXDoubleObject* obj);
```

Description

Releases all resources that have been allocated to an object. The object can only be deleted by the process that created it and must be resident in that process. This is the opposite of the `CSPX_double_object_new` function.

Note: In general, it is only the host that can create an object to move data to and from the accelerator. However, the double buffer object is a wrapper for the “inner object” (used to move the data buffers) and status information about the buffer currently being used. As such, a double buffer object needs to be created and initialized on both the host and the accelerator in order to manage the state on each side.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_get_handle

Prototype

```
CSPXErrno CSPX_double_object_get_handle(CSPXDoubleObject *obj);
```

Description

Gets the handle for the “inner” object that has the data buffers attached to it. This handle needs to be passed between the host and the accelerator using the double buffered object to allow them to share the buffers.

Parameters

`obj`: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_get_name

Prototype

```
char *CSPX_double_object_get_name(  
    CSPXDoubleObject *obj,  
    CSPXErrno *error_code  
);
```

Description

Returns the name of the object. If a name has not been specified by the programmer then the function will return a default name which is the file and line number where the object was originally initialized.

Parameters

`obj`: a pointer to the object

`error_code`: an error code indicating the result of the operation

Returns

A pointer to the string. In the event of an error, the function returns `NULL`.

CSPX_double_object_get_pointer

Prototype

```
void *CSPX_double_object_get_pointer(  
    CSPXDoubleObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Gets a pointer to the appropriate data buffer attached to the object.

Parameters

obj: a pointer to the object

error_code: an error code indicating the result of the operation

Returns

The local address of the data buffer. If the object is not resident, the function returns `NULL`.

CSPX_double_object_move

Prototype

```
CSPXErrno CSPX_double_object_move(  
    CSPXProcess dest,  
    CSPXDoubleObject* obj  
);
```

Description

Moves an object to a target process. This performs the migration of the appropriate data buffer attached to the object. Note that this operation is asynchronous and the function will return immediately.

Parameters

dest: the handle of the process to send the object to

obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_new

Prototype

```
CSPXErrno CSPX_double_object_new(CSPXDoubleObject* obj);
```

Description

Initializes a double buffered object and registers it with the object manager. All objects must be initialized before they are used.

Note: In general, it is only the host that can create an object to move data to and from the accelerator. However, the double buffer object is a wrapper for the "inner object" (used to move the data buffers) and status information about the buffer currently being used. As such, a double buffer object needs to be created and initialized on both the host and the accelerator in order to manage the state on each side.

Parameters

obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_sync

Prototype

```
CSPXErrno CSPX_double_object_sync(CSPXDoubleObject* obj);
```

Description

Synchronizes with an object. This function will block until the object in question is resident in the calling process. If the object is already resident it will return immediately.

Parameters

obj: a pointer to the object

Returns

An error code indicating the result of the operation.

CSPX_double_object_sync_pointer

Prototype

```
void * CSPX_double_object_sync_pointer(  
    CSPXDoubleObject* obj,  
    CSPXErrno *error_code  
);
```

Description

Synchronizes with an object and returns a pointer to the associated data. This function combines the `CSPX_double_object_sync` and `CSPX_double_object_get_pointer` functions.

Parameters

`obj`: a pointer to the object

`error_code`: an error code indicating the result of the operation

Returns

The local address of the attached data or NULL in error cases.

10.5 Pipes

```
#include <csp_x_pipe.h>
```

CSPX provides a data pipe implementation to facilitate streaming data to and from accelerators. A CSPX pipe is similar to a Unix pipe where data can be written by one process (the *producer*) and read by another (the *consumer*). A pipe is unidirectional. One end must be the host process, the other being an accelerator.

10.5.1 Types

CSPXPipe

The type used to represent pipes on the host and accelerators.

10.5.2 Functions

CSPX_pipe_get_index

Prototype

```
CSPXErrno CSPX_pipe_get_index(  
    CSPXPipe* pipe,  
    int *index  
);
```

Description

Returns the internal index of the pipe. This is primarily of use for debugging purposes and shouldn't normally be needed.

Parameters

`pipe`: a pointer to the pipe
`index`: the returned index value of the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_get_name

Prototype

```
char *CSPX_pipe_get_name(  
    CSPXPipe* pipe,  
    CSPXErrno *error_code  
);
```

Description

Returns the name of the pipe. If a name has not been set then the function will return a default name which is the file and line number where the object was originally initialized.

Parameters

`pipe`: a pointer to the pipe
`error_code`: an error code indicating the result of the operation.

Returns

A pointer to the string. In the event of an error, the function returns `NULL`.

CSPX_pipe_read

Prototype

```
CSPXErrno CSPX_pipe_read(  
    CSPXPipe* pipe,  
    void* data,  
    size_t bytes  
);
```

Description

Reads from a pipe. This will block until the specified number of bytes has been read.

Parameters

`pipe`: a pointer to the pipe
`data`: a pointer to the destination for the data
`bytes`: the number of bytes to read from the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_reader

Prototype

```
CSPXProcess CSPX_pipe_reader(  
    CSPXPipe* pipe  
    CSPXErrno *error_code  
);
```

Description

Returns the handle of the process reading from the pipe.

Parameters

`pipe`: a pointer to the pipe
`error_code`: an error code indicating the result of the operation

Returns

The process handle for the consumer or NULL on error.

CSPX_pipe_write

Prototype

```
CSPXErrno CSPX_pipe_write(  
    CSPXPipe* pipe,  
    const void *data,  
    size_t bytes  
);
```

Description

Writes to a pipe. This will not block if there is sufficient buffering to take the data.

Parameters

`pipe`: a pointer to the pipe
`data`: a pointer to the source of the data
`bytes`: the number of bytes to write to the pipe

Returns

An error code indicating the result of the operation.

CSPX_pipe_writer

Prototype

```
CSPXProcess CSPX_pipe_writer(  
    CSPXPipe* pipe,  
    CSPXErrno *error_code  
);
```

Description

Returns the handle of the process reading from the pipe.

Parameters

`pipe`: a pointer to the pipe

`error_code`: an error code indicating the result of the operation

Returns

The process handle for the producer or NULL on error.

10.6 Error handling

```
#include <csp_x_errno.h>
```

These functions provide support for handling and reporting errors returned by CSPX functions.

10.6.1 Types

CSPXErrno

The `CSPXErrno` enumeration defines the set of values that can be returned to indicate an error. See [Section 7.1: Error codes on page 63](#) for details of the error codes and their meanings.

10.6.2 Functions

CSPX_get_error_string

Prototype

```
CSPXErrno CSPX_get_error_string(  
    CSPXErrno error_code,  
    char * const error_string,  
    unsigned int max_string_length  
);
```

Description

Returns the error string associated with the error code.

Parameters

`error_code`: the error code
`error_string`: a pointer to a pre-allocated buffer to receive error string.
`max_string_length`: the maximum length of the error string buffer.

Returns

An error code indicating the result of the operation.

CSPX_get_error_string_length

Prototype

```
CSPXErrno CSPX_get_error_string_length(  
    CSPXErrno error_code,  
    unsigned int* const length  
);
```

Description

Returns the length of the description string for the given error code. This allows a buffer to be allocated that will hold the error string returned by `CSPX_get_error_string`.

Parameters

`error_code`: the error code

`length`: the value to return the string length

Returns

An error code indicating the result of the operation.

Bibliography

1. SDK Reference Manual
Document Number: 06-UG-1136
ClearSpeed Technology
2. The C[®] Standard Libraries Reference Manual
Document Number: 06-RM-1139
ClearSpeed Technology
3. CSX Processor Architecture
White Paper 02-WP-1110
ClearSpeed Technology
4. Instruction Set Reference Manual
Document Number: 06-RM-1137
ClearSpeed Technology
5. Runtime Software User Guide
Document Number: 06-UG-1345
ClearSpeed Technology
6. Patterns for Parallel Programming
Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill
ISBN: 0-321-22811-1
Addison Wesley, 2004

ClearSpeed Technology Ltd

130 Aztec West
Park Avenue
Bristol BS32 4UB
United Kingdom

Tel: +44 (0)1454 629 623

Fax: +44 (0)1454 629 624

Email: info@clearspeed.com

Web: <http://www.clearspeed.com>

Support: <http://support.clearspeed.com>

Acknowledgements:

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

1. Information and data contained in this document, together with the information contained in any and all associated ClearSpeed documents including without limitation, data sheets, application notes and the like ('Information') is provided in connection with ClearSpeed products and is provided for information only. Quoted figures in the Information, which may be performance, size, cost, power and the like are estimates based upon analysis and simulations of current designs and are liable to change.
2. Such Information does not constitute an offer of, or an invitation by or on behalf of ClearSpeed, or any ClearSpeed affiliate to supply any product or provide any service to any party having access to this Information. Except as provided in ClearSpeed Terms and Conditions of Sale for ClearSpeed products, ClearSpeed assumes no liability whatsoever.
3. ClearSpeed products are not intended for use, whether directly or indirectly, in any medical, life saving and/ or life sustaining systems or applications.
4. The worldwide intellectual property rights in the Information and data contained therein is owned by ClearSpeed. No license whether express or implied either by estoppel or otherwise to any intellectual property rights is granted by this document or otherwise. You may not download, copy, adapt or distribute this Information except with the consent in writing of ClearSpeed.
5. The system vendor remains solely responsible for any and all design, functionality and terms of sale of any product which incorporates a ClearSpeed product including without limitation, product liability, intellectual property infringement, warranty including conformance to specification and or performance.
6. Any condition, warranty or other term which might but for this paragraph have effect between ClearSpeed and you or which would otherwise be implied into or incorporated into the Information (including without limitation, the implied terms of satisfactory quality, merchantability or fitness for purpose), whether by statute, common law or otherwise are hereby excluded.
7. ClearSpeed reserves the right to make changes to the Information or the data contained therein at any time without notice.

© Copyright ClearSpeed Technology Ltd 2010. All rights reserved.

ClearSpeed, ClearConnect, Advance and the ClearSpeed logo are trade marks or registered trade marks of ClearSpeed Technology Ltd. All other brands and names are the property of their respective owners.