Developing Symphony Applications for Cell BE Tutorial

Platform Symphony™
Version 5.1
April 2011

Copyright

We’d like to hear from you

You can help us make this document better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this document, please address your comments to doc@platform.com.

Your comments should pertain only to Platform documentation. For product support, contact support@platform.com.

Document redistribution and translation

Internal redistribution

Trademarks

® LSF is a registered trademark of Platform Computing Corporation in the United States and in other jurisdictions.

™ ACCELERATING INTELLIGENCE, PLATFORM COMPUTING, PLATFORM SYMPHONY, PLATFORM JOB SCHEDULER, PLATFORM ISF, PLATFORM ENTERPRISE GRID ORCHESTRATOR, PLATFORM EGO, and the PLATFORM and PLATFORM LSF logos are trademarks of Platform Computing Corporation in the United States and in other jurisdictions.

® UNIX is a registered trademark of The Open Group in the United States and in other jurisdictions.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.

® Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Intel®, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other products or services mentioned in this document are identified by the trademarks or service marks of their respective owners.

Third-party license agreements

http://www.platform.com/Company/third.part.license.htm

Third-party copyright notices

http://www.platform.com/Company/Third.Party.Copyright.htm

Contents

Tutorial: Developing a Synchronous Symphony Application for the IBM Cell BE

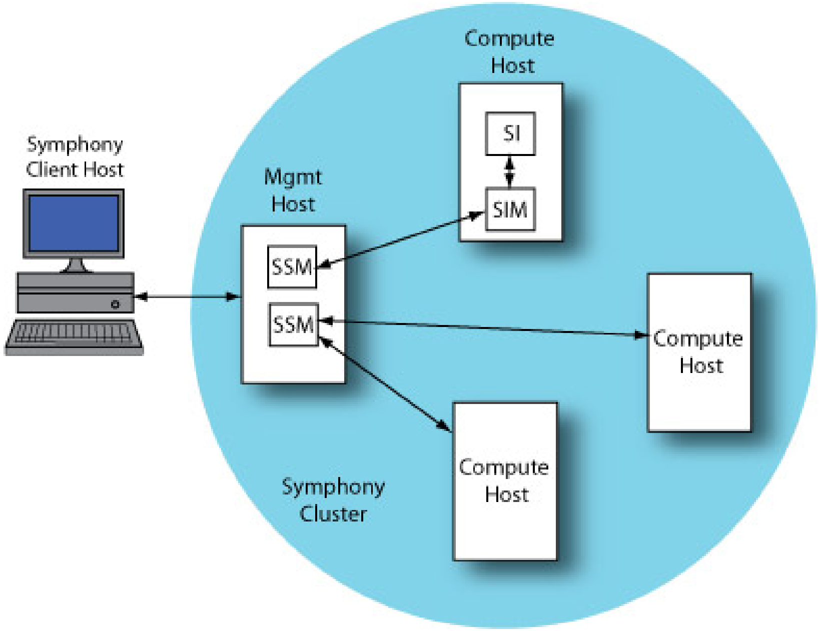
Goal

This tutorial walks you through the sample application code and guides you through the process of building, packaging, deploying, and running the sample client and service.

You will learn the minimum amount of code that you need to create a Symphony application for the IBM Cell Broadband Engine (BE).

Overview

This section describes the high-level interaction between a client and a Symphony service. The following diagram shows the message flow between the client and the hosts in a Symphony cluster.

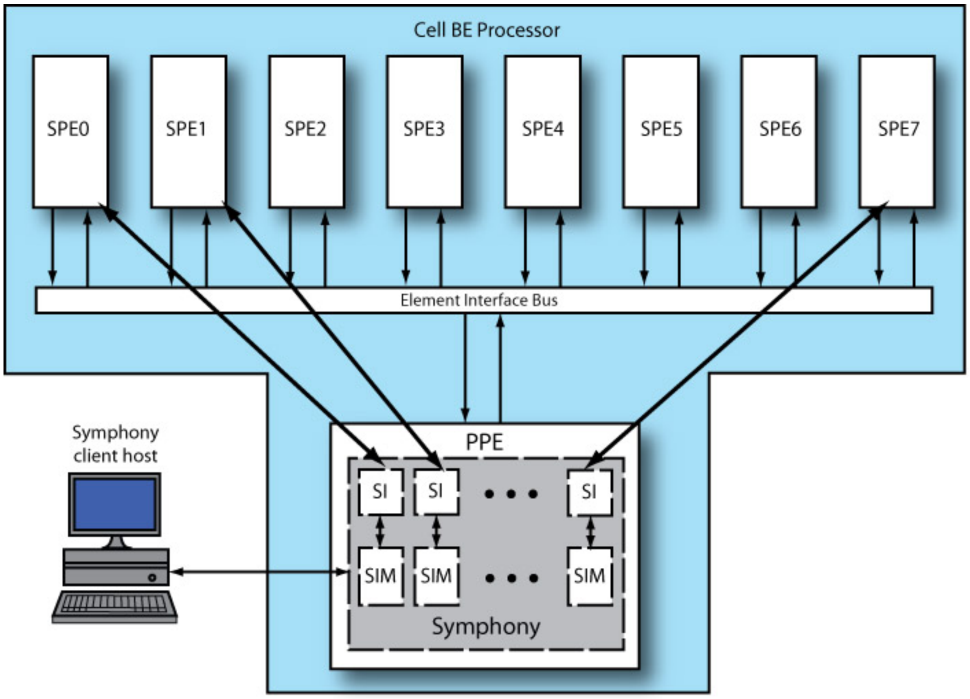


The client opens a session with the session director (not shown) on the management host. Once the client is authenticated, the client communicates directly with the session manager assigned to the application.

The Symphony session manager (SSM), which also runs on the management host, is the workload manager associated with a single application. The session manager routes messages from the client to the compute hosts and from the compute hosts to the clients.The session manager obtains resources to service its sessions and starts/manages service instance manager (SIM) processes on compute hosts.

The service instance manager starts, monitors, and manages a service instance (SI), passing inputs and outputs between the session manager and service instance.

The service code sample, which this tutorial is based on, is designed to be run on a minimum of one Cell blade with two BE processors. The following diagram shows the architecture of a single Cell BE processor with Symphony installed.



The Power Processor Element (PPE) is the main processor in the Cell BE. The PPE is responsible for overall control of the system and runs the operating system for all applications on the Cell BE. The Symphony service runs on the PPE, and individual computational tasks are off-loaded to the SPEs. The PPE then waits for and coordinates the results returning from the SPEs.

The Synergistic Processor Element (SPE) handles the compute-intensive tasks. Each SPE is an independent processor, and is optimized to run SPE threads spawned by the PPE.

In this sample, the client sends a configurable amount of tasks to the Symphony service. The tasks contain the input data for the calculation program that performs a simple addition of two integers on the SPEs. The service, which runs on the PPE, spawns one thread on each SPE that passes the input data to the calculation program. The calculation program is executed on each SPE concurrently. When the programs have completed the work, the results are collected by the service and relayed back to the client.

Application development methodology

When developing Symphony applications for the Cell BE, here is the programming process model:

1. Prepare the code:
    1. Write SPE calculation code
    2. Write PPE code (Symphony integration service)
    3. Write client code
2. Get SPE binary:
    1. Compile SPE code to SPE object file

    2. Link SPE object file to SPE executable
    3. Convert SPE executable to PPE embedded object file
3. Get PPE binary:
    1. Compile PPE code to PPE object file
    2. Link PPE object file and PPE embedded object file to PPE executable (Symphony integration service)

Note:

The PPE and SPE source programs use different compilers. The correct compiler and libraries must be used for the intended processor.

Symphony service API scope

All the Symphony Service classes and methods are applicable to the PPE. Because of the memory limitation of the SPE, none of the Symphony Service classes and methods are applicable to the SPE. For example, the Message object cannot be passed to the SPE or instantiated in the SPE and none of its methods can be executed in the SPE.

All basic data type objects can be passed between the PPE and SPE. For example, the integer value stored in the Message object can be accessed in the PPE and passed to the SPE. The value can also be passed from the SPE to PPE.

For more information about the scope of Symphony service APIs, refer to *Appendix A: Symphony API Summary* on page 26.

Prerequisites

Client host

- Operating system: all platforms supported by Symphony
- Compiler: gcc 3.4, 4.0, 4.1, Intel C++ 9.1, Visual Studio C++ (version 6 and higher), .NET, Java, CC

Note:

Although the client API is accessible on a Cell BE host, it is not recommended to run Symphony clients on this type of host. The Cell BE is intended primarily as an accelerator to speed up calculations.

Management host

Operating system: all platforms supported by Symphony

Service host

- Operating system: RHEL 5.1
- Compiler:
    - PPE

      ppu-g++, g++
    - SPE

  spu-gcc, spu-g++
- IBM Cell BE: QS21, QS22/SDK 3.0

Installing Symphony

Since there is no standalone Developer Edition for the Cell BE that would allow development and testing on a single host, the Cell BE requires the following components to be installed, as a minimum:

- client host
- management host
- compute host

These components form the basic building blocks of a Symphony cluster. Platform provides the following Symphony packages for setting up the compute host and application development environment:

- symcomputehostSetup5.1.0_linux2.6-glibc2.5-ppc64_CellBE-nnnnnn.bin
- symphonySDK-linux2.6-glibc2.5-ppc64_CellBE-5.1.0-nnnnnn.tar.gz

The Symphony package for the management host is dependent on the management host platform. To obtain the Symphony package, go to my.platform.com and select Products > Platform Symphony > Symphony 5.1 > Product Packages. Download the appropriate package for your host.

For getting started with Symphony, go to my.platform.com and select Products > Platform Symphony > Symphony 5.1 > Install Platform Symphony.

Important:

If you run egosh ego start/shutdown commands using sudo, you may not be able to access Symphony environment variables that were set in a non-root account. In this case, you must configure the /etc/sudoers file to access the variables.

Where to find the documentation

Additional documentation is available from the Knowledge Center located at*`$SOAM_HOME/`*`docs` on the compute hosts. For details on all API programming calls, refer to the Platform Symphony C++ API Reference.

Limitations

symping5.1 and symexec5.1 applications

To run the symping5.1 or symexec5.1 application with the Cell BE on the grid, you must re-register the latest application profile since it includes the Cell BE OS type. To re-register the application:

1. Log on to the Cell BE host.
2. Set the command-line environment.

   For example, if you installed Symphony in /opt/ego:

   - For csh or tcsh, use cshrc.platform:

```
source /opt/ego/cshrc.platform
```

- For sh, ksh, or bash, use profile.platform:

```
. /opt/ego/profile.platform
```

3. Change the current directory to the directory in which `symping5.1.xml`and `symexec5.1.xml` are located:

```
cd $SOAM_HOME/5.1/linux2.6-glibc2.5-ppc64_CellBE/bin/
```

4. Register the application profile:

```
soamreg symping5.1.xml
```

```
soamreg symexec5.1.xml
```

Review and understand the sample

Review the sample application code to learn how you can create a simple synchronous application for the IBM Cell BE.

Locate the code samples

Files	Location of Code Sample
Client	`$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/SyncClient`
Message object	`$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Common`
Service code	`$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Service`
Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/SampleAppCell.xml`
Output directory	`$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Output`
Makefile	`$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Makefile`

What the sample does

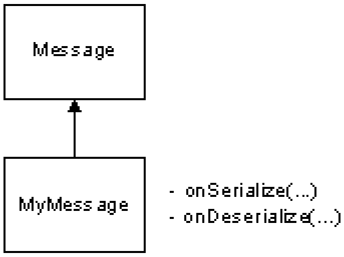
The sample allows you to enter two integers via command line when running the client. One integer is stored in class `MyMessage (MyMessage.h` and `MyMessage.cpp`) and the other is stored in class `MyCommonData` (`MyCommonData.h` and `MyCommonData.cpp`). `MyMessage` and `MyCommonData` are passed to the service. On the service side, the two integers are fetched from `MyMessage` and `MyCommonData` and passed to the SPE where they are added together. The result is passed back to the PPE and then sent back to the client where it is displayed on the screen.

When you run the client, it opens a session and sends *n* input messages (tasks) to the service running on the PPE of the Cell BE. The service spawns *m* threads that run concurrently on each SPE and perform the simple addition. The client application is synchronous so it sends input and blocks the output until all the results are returned.

Review the sample message code

Input and output: declare the message class

Your client application needs to handle data that it sends as input, and output data that it receives from the service.



Tip:

Clients and services share the same message class.

In `MyMessage.h`:

- We declare the `MyMessage` class
- We declare serialization methods for input and output messages
- We declare methods to handle the data

Note:

For this example, we have defined the same class for input and output messages. However, you can define separate classes for input and output messages.

```
pragma once
#include "soam.h"
class MyMessage :
    public soam::Message
{
public:
    MyMessage();
    MyMessage(int taskInput, char* str, int spus, bool isSync);
    virtual ~MyMessage(void);
    void onSerialize(
        /*[in]*/ soam::OutputStreamPtr &stream) throw (soam::SoamException);
    void onDeserialize(
        /*[in]*/ soam::InputStreamPtr &stream) throw (soam::SoamException);
// accessors
public:
    int getSpus() const{return m_spus;}
    void setSpus(int _int) {m_spus = _int;}
    char* getString() const{return m_string;}
    void setString(const char* str) {freeString(m_string); m_string = copyString(str);}
    int getTaskInput(){return m_taskInput;}
    void setTaskInput(int taskInput){m_taskInput = taskInput;}
    bool getIsSync() const {return (m_isSync != 0);}
    void setIsSync(bool isSync) {m_isSync = isSync;}
private:
    char* copyString(const char* strSource);
    void freeString(char* strToFree);
private:
    int m_taskInput;
    char *m_string;
    int m_spus;
    bool m_isSync;
};
```

Implement the MyMessage object

Once your message class is declared, implement handlers for serialization and deserialization.

In `MyMessage.cpp`, we implement methods to handle the data. For data types that are supported by the Symphony SDK, see the appropriate API reference.

Note:

If you already have an application with a message object that is serialized, you can pass a binary blob through the DefaultBinaryMessage class.

```
MyMessage::MyMessage()
{
    m_taskInput = 0;
    m_string = copyString("");
    m_spus = 0;
    m_isSync = true;
}
MyMessage::MyMessage(int taskInput, char *str, int spus, bool isSync)
{
    m_taskInput = taskInput;
    m_string = copyString(str);
    m_spus = spus;
    m_isSync = isSync;
}
MyMessage::~MyMessage(void)
{
    freeString(m_string);
}
void MyMessage::onSerialize(OutputStreamPtr &stream) throw (SoamException)
{
    stream->write(m_taskInput);
    stream->write(m_string);
    stream->write(m_spus);
    stream->write(m_isSync);
}
void MyMessage::onDeserialize(InputStreamPtr &stream) throw (SoamException)
{
    stream->read(m_taskInput);
    freeString(m_string);
    stream->read(m_string);
    stream->read(m_spus);
    stream->read(m_isSync);
}
char* MyMessage::copyString(const char* strSource)
{
    SOAM_ASSERT(SOAM_NULL_PTR != strSource);
    size_t len = strlen(strSource);
    char* newString = new char[len+1];
    SOAM_ASSERT(SOAM_NULL_PTR != newString);
    strcpy(newString, strSource);
    return newString;
}
void MyMessage::freeString(char* strToFree)
{
    if (SOAM_NULL_PTR != strToFree)
    {
        delete []strToFree;
    }
}
```

Review the sample common data code

The `MyCommonData` class, which inherits from the `Message` class, handles the common data for the client and service. The class declaration and definition are contained in `MyCommonData.h` and MyCommonData.cpp

Declare the MyCommonData class

In `MyCommonData.h`:

- We declare the `MyCommonData` class
- We declare serialization methods for the common data object
- We declare methods (accessors) to handle the data

```
#pragma once
#include "soam.h"
/////////////////////////////////
// Common Data Object
/////////////////////////////////
class MyCommonData :
    public soam::Message
{
public:
    MyCommonData();
    MyCommonData(int i);
    virtual ~MyCommonData(void);
    void onSerialize(
        /*[in]*/ soam::OutputStreamPtr &stream) throw (soam::SoamException);
    void onDeserialize(
        /*[in]*/ soam::InputStreamPtr &stream) throw (soam::SoamException);
// accessors
public:
    int getInt() const {return m_int;}
    void setInt(int i) {m_int = i;}
private:
    int m_int;
};
```

Implement the MyCommonData object

Once your common data class is declared, implement handlers for serialization and deserialization.

In `MyCommonData.cpp`, we implement methods to handle the data. For data types that are supported by the Symphony SDK, see the appropriate API reference.

```
MyCommonData::MyCommonData()
{
    m_int = 0;
}
MyCommonData::MyCommonData(int i)
{
    m_int = i;
}
MyCommonData::~MyCommonData(void)
{
}
void MyCommonData::onSerialize(OutputStreamPtr &stream) throw (SoamException)
{
    stream->write(m_int);
}
void MyCommonData::onDeserialize(InputStreamPtr &stream) throw (SoamException)
{
    // we now own the int returned from the read call
    stream->read(m_int);
}
```

Review the sample client code

Enable the use of command options

To add flexibility, the client program is designed to receive up to five arguments for setting parameters or for displaying help, as follows:

Option	Default	Description
-c	0	Common data input value
-i	0	Task input value

Option	Default	Description
-s	1	Run the specified number of SPEs per task
-t	16	Perform the specified number of tasks
-h		Display help for command options on the screen

In this sample, we initialize the parameters with default values so that the program can run without passing arguments. In cases where arguments are used, a switch block parses the inputs and overwrites the default values.

```
...
//value that will be stored in MyMessage
int taskInput = 0;
//value that will be stored in MyCommonData
int commonDataInput = 0;
//number of tasks that will be sent
int tasksToSend = 16;
//number of SPUs that will run for each task
int spus = 1;
//Parse command line parameters
//Set taskInput, commonDataInput, tasksToSend and spus
for (int i=1; i<argc; i++)
{
    if (*argv[i] == '-')
    {
        switch (*(argv[i]+1))
        {
            case 't':
                i++;
                if (i < argc)
                {
                    tasksToSend = atoi(argv[i]);
                    if (tasksToSend < 0) tasksToSend = 0;
                }
                else
                {
                    printf("ERROR: Number of tasks is not specified.\n");
                    print_usage(argv[0]);
                }
                break;
```

```
            case 's':
                i++;
                if (i < argc)
                {
                    spus = atoi(argv[i]);
                    if (spus < 0) spus = 0;
                    if (spus > MAX_SPUS) spus = MAX_SPUS;
                }
                else
                {
                    printf("ERROR: Number of SPUs to use is not specified.\n");
                    print_usage(argv[0]);
                }
                break;
            case 'i':
                i++;
                if (i < argc)
                {
                    taskInput = atoi(argv[i]);
                    if (taskInput < 0) taskInput = 0;
                }
                else
                {
                    printf("ERROR: Task input value is not specified.\n");
                    print_usage(argv[0]);
                }
                break;
```

```
            case 'c':
                i++;
                if (i < argc)
                {
                    commonDataInput = atoi(argv[i]);
                    if (commonDataInput < 0) commonDataInput = 0;
                }
                else
                {
                    printf("ERROR: Common data input value is not specified.\n");
                    print_usage(argv[0]);
                }
                break;
```

```
            case 'h':
            default:
                print_usage(argv[0]);
                break;
        }
    }
    else
    {
        print_usage(argv[0]);
    }
}
```

Initialize the client

In `SyncClient.cpp`, when you initialize, you initialize the Symphony client infrastructure. You initialize once per client.

Important:

Initialization is required. Otherwise, API calls fail.

```
...
    try
    {
        // Initialize the API
        SoamFactory::initialize();
...
```

Connect to an application

To send data to be calculated in the form of input messages, you connect to an application.

You specify an application name, a user name, and password. The application name must match that defined in the application profile. The default security callback encapsulates the callback for the user name and password.

Tip:

When you connect, a connection object is returned.

```
...
  // Set up application specific information to be supplied to Symphony
  char appName[]="SampleAppCell";
  // Set up application authentication information using the default security provider
  DefaultSecurityCallback securityCB("Guest", "Guest");
  // Connect to the specified application
  ConnectionPtr conPtr = SoamFactory::connect(appName, &securityCB);
  // Retrieve and print our connection ID
  cout << "connection ID=" << conPtr->getId() << endl;
...
```

Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. The tasks are sent and received synchronously.

When creating a session, you need to specify the session attributes by using the `SessionCreationAttributes` object. In this sample, we create a `SessionCreationAttributes` object called `attributes` and set four parameters in the object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command-line interface.

The second parameter is the session type. The session type is optional. You can leave this parameter blank and system default values are used for your session.

The third parameter is the session flag, which we specify as `ReceiveSync`. You must specify it as shown. This indicates to Symphony that this is a synchronous session.

The fourth parameter is the common data value that will be shared among tasks in the session.

We pass the `attributes` object to the `createSession()` method, which returns a pointer to the session.

Important:

The session type must be the same session type as defined in your application profile.

You define characteristics for the session with the session type in the application profile.

```
        // Set up session creation attributes
        SessionCreationAttributes attributes;
        attributes.setSessionName("mySession");
        attributes.setSessionType("ShortRunningTasks");
        attributes.setSessionFlags(Session::ReceiveSync);
        attributes.setCommonData(&commonData);
        // Create a synchronous session
        SessionPtr sesPtr = conPtr->createSession(attributes);
```

Send input data to be processed

In this step, we create *n* input messages to be processed by the service. When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
        // Now we will send some messages to our service
        for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
        {
            // Create a message
            MyMessage inMsg(taskInput, "", spus, true);
            // Create task attributes
            TaskSubmissionAttributes attrTask;
            attrTask.setTaskInput(&inMsg);
            // send it
            TaskInputHandlePtr input = sesPtr->sendTaskInput(attrTask);
            // Retrieve and print task ID
            cout << "task submitted with ID : " << input->getId() << endl;
        }
```

Retrieve output

Pass the number of tasks to the `fetchTaskOutput()` method to retrieve the output messages that were produced by the service. This method blocks until the output for all tasks is retrieved. The return value is an enumeration that contains the completed task results.Iterate through the task results and extract the messages using the `populateTaskOutput()` method. Display the task ID and the results from the output message.

```
        // Now get our results - will block here until all tasks retrieved
        EnumItemsPtr enumOutput = sesPtr->fetchTaskOutput(tasksToSend);
        // Inspect results
        TaskOutputHandlePtr output;
        while(enumOutput->getNext(output))
        {
            // Check for success of task
            if (true == output->isSuccessful())
            {
                // Get the message returned from the service
                MyMessage outMsg;
                output->populateTaskOutput(&outMsg);
                // Display content of reply
                cout << "Task Succeeded [" <<  output->getId() << "]" << endl;
                cout << outMsg.getString() << endl;
            }
            else
            {
                // Get the exception associated with this task
                SoamExceptionPtr ex = output->getException();
                cout << "Task Failed : " << ex->what() << endl;
            }
        }
```

Catch exceptions

Any exceptions thrown take the form of SoamException. Catch all Symphony exceptions to know about exceptions that occurred in the client application, service, and middleware.

The following sample code catches exceptions of type SoamException.

```
catch(SoamException& exp)
{
    // Report exception
    cout << "exception caught ... " << exp.what() << endl;
}
```

Uninitialize

Always uninitialize the client API at the end of all API calls. If you do not call uninitialize, the client API is in an undefined state, resources used by the client are held indefinitely, and there is no guarantee your client will be stable.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

```
    // uninitialize the API
    // This is the only means to ensure proper shutdown
    // of the interaction between the client and the system.
    SoamFactory::uninitialize();
...
```

Review the sample calculation code

The calculation code is contained in `service_spu.c`. This is the program that transfers, via DMA, the data from the PPE to the SPEs for execution. Memory flow controller (MFC) commands are used to transfer the data between the PPE and SPEs. You can see from this code sample that each SPE simply calculates the addition of the task input value and the common data value. from the client.

```
 int main(unsigned long long speid __attribute__ ((unused)), unsigned long long
 parms_ea)
 {
     int tag = 31, tag_mask = 1<<tag;

     //Fetch the parameters from PPU
     mfc_get(&parms, (unsigned long long)parms_ea, sizeof(parms), tag, 0, 0);
     mfc_write_tag_mask(tag_mask);
     mfc_read_tag_status_any();
     //Calculate
     parms.taskOutput = parms.taskInput + parms.commonDataInput;
     //Send parameters back to PPU
     mfc_put(&parms, (unsigned long long)parms_ea, sizeof(parms), tag, 0, 0);
     mfc_write_tag_mask(tag_mask);
     mfc_read_tag_status_any();

     return (0);
 }
...
```

Review the sample service code

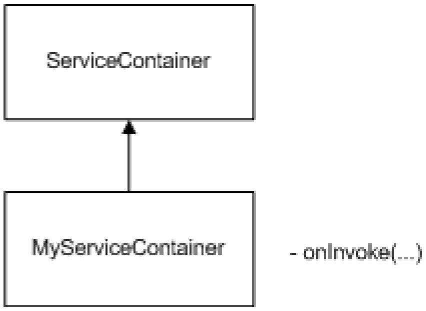
The Symphony service code provides inputs to calculation code that is executed on the SPEs. To take advantage of the Cell BE architecture, the service creates individual threads that run concurrently on individual SPEs. Each thread has its own context.

This sample uses the following basic algorithm to run multiple SPE contexts:

1. Create *n* SPE contexts (one for each SPE thread).
2. Load the SPE calculation program (executable object) into each SPE context's local memory.
3. Run an SPE context in each thread.
4. Wait for each SPE thread to terminate.
5. Destroy the SPE thread context.

Define a service container

For a service to be managed by Symphony, it needs to be in a container object. This is the service container.



In `SampleService.cpp`, we inherited from the ServiceContainer class.

```
class MyServiceContainer : public ServiceContainer
```

Run the container

The service is implemented within an executable. At a minimum, we need to create within our main function an instance of the service container and run it.

```
int main(int argc, char* argv[])
{
    // return value of our service program
    int retVal = 0;
    try
    {
        // Create the container and run it
        MyServiceContainer myContainer;
        myContainer.run();
    }
```

Retrieve the common data

Load the session common data into memory by implementing `onSessionEnter()` before the `onInvoke()` call. When common data is available, Symphony invokes `onSessionEnter()` once after the service is bound to your session.

Use the `populateCommonData()` method of the `sessionContext` object to load the common data.

```
void onSessionEnter (SessionContextPtr& sessionContext)
{
        // if common data exists now, delete it to prevent memory leak
        if (0 != m_commonData)
        {
            delete m_commonData;
            m_commonData =  0;
        }
        // populate our common data object
        m_commonData = new MyCommonData();
        sessionContext->populateCommonData(*m_commonData);
}
...
```

Process the input

Symphony calls `onInvoke()` on the service container once per task. Once you inherit from the ServiceContainer class, implement handlers so that the service can function properly.

To gain access to the data from the client, you must present an instance of the message object to the `populateTaskInput()` method on the task context. The task context contains all information and functionality that is available to the service during an `onInvoke()` call in relation to the task that is being processed.

Important:

Services are virtualized. As a result, a service should not read from stdin or write to stdout. Services can, however, read from and write to files that are accessible to all compute hosts.

You pass the message object, which comes from the client application, to `populateTaskInput()`. During this call, the data sent from the client is used to populate the message object. Task context such as the number of SPEs to run per task and the task input value are then loaded into local variables for use by the service code.

```
{
public:
virtual void onInvoke (TaskContextPtr& taskContext)
{
    // get the input that was sent from the client
    MyMessage inMsg;
    taskContext->populateTaskInput(inMsg);
    int spus = inMsg.getSpus();
    int taskInput = inMsg.getTaskInput();
    int commonDataInput = m_commonData->getInt();
...
```

Initialize the SPE threads

Since this service will use *n* SPEs concurrently, it is necessary for the service to create *n* threads. Each of these threads will run a single SPE context at a time. The thread runs on the SPE and is responsible for running the calculation program and retrieving the result.

Since we will be running tasks on *n* SPEs, we need to create an array to hold the parameters of each thread.

The `mm_parms` structure is instantiated as `parms`, which is used on the service side for passing messages between the PPE and SPE. The `mm_parms` structure is declared and defined in `params.h`.

Members of the `mm_params` structure include:

- `taskInput` stores the integer that is entered on the command line when running SyncClient with the -i parameter.
- `commonDataInput` stores the integer entered on the command line when running SyncClient with the -c parameter
- `taskOutput` stores the addition result in the SPE, which is passed back to the PPE.

```
    int i;
    for (i=0; i<spus; i++)
    {
        /* Initialize the thread structure and its parameters.
        */
        threads[i].parms.taskInput = taskInput;
        threads[i].parms.commonDataInput = commonDataInput;
        threads[i].parms.taskOutput = 0;
```

```
typedef struct _mm_parms
{
    unsigned int taskInput __attribute__ ((aligned (16)));
    unsigned int commonDataInput __attribute__ ((aligned (16)));
    //parameter sent to SPU and assigned as taskInput + commonDataInput in SPU
    unsigned int taskOutput __attribute__ ((aligned (16)));
} mm_parms;
```

Start the calculation program on the SPE

The PPE starts the calculation program by creating a thread on each SPE. The PPE uses the `spe_context_create()`, `spe_program_load()`, and `spe_context_run()` library calls provided in the SPE runtime management library.

The context for the SPE thread contains the persistent data about the SPE. Before being able to use an SPE, the SPE context data structure has to be created and initialized. This is done by calling `spe_context_create()`, which returns a pointer to the newly created SPE context when it is successfully created.

Before being able to run an SPE context, an SPE program has to be loaded into the context using the `spe_program_load()` call. You must pass a valid pointer to the SPE context and the address of the SPE program to `spe_program_load()`.

The `pthread_create()` function creates a new thread of control that executes concurrently with the calling thread. The `pthread_create()` function requires you to pass a variable that will hold the ID of the newly created thread, the function (`ppu_thread_function`) that the thread will execute, and the SPE context pointer. The `ppu_thread_function` receives the SPE context pointer as its sole argument and calls `spe_context_run()`, which executes the SPE context on a physical SPE. This subroutine causes the current PPE thread to transition to an SPE thread by passing its execution control from the PPE to the SPE whose context it is scheduled to run on.

```
    //Create context for the SPU thread
    if ((threads[i].id = spe_context_create (0, NULL)) == NULL)
    {
        sprintf(errmsg,"INTERNAL ERROR: failed to create spu context %d. Error
        = %s\n", i, strerror(errno));
        throw new FatalException(errmsg);
    }
    //Load program into context
    if (spe_program_load (threads[i].id, &service_spu) != 0)
    {
        sprintf(errmsg, "INTERNAL ERROR: failed to load program %d. Error = %s
        \n", i, strerror(errno));
        throw new FatalException(errmsg);
    }
    //Execute context on SPU
    if (pthread_create (&threads[i].pthread, NULL, &ppu_pthread_function,
     &threads[i].id) != 0)
    {
        sprintf(errmsg, "INTERNAL ERROR: failed to create pthread %d. Error =
        %s\n", i, strerror(errno));
        throw new FatalException(errmsg);
    }
}
...
```

```
void *ppu_pthread_function(void *arg)
{
    struct _threads *data;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    data = (struct _threads *)arg;
    //This subroutine causes the current PPE thread to transition to an SPE thread
    if (spe_context_run(data->id, &entry, 0, (void *)(&(data->parms)), NULL, NULL) < 0)
    {
        throw new FatalException("Failed running context");
    }
    pthread_exit(NULL);
}
```

Wait for the results

Wait for all the SPEs to complete the calculations and then return execution control to the PPE. The `pthread_join()` function suspends execution of the calling thread until all the SPE threads have terminated.

```
...
    for (i=0; i<spus; i++)
    {
        //Wait for the SPU to complete
        if (pthread_join (threads[i].pthread, NULL) != 0)
        {
            sprintf(errmsg, "INTERNAL ERROR: failed to join pthread %d.
            Error = %s\n", i, strerror(errno));
            throw new FatalException(errmsg);
        }
```

Destroy the thread contexts

As each SPE thread terminates, destroy the thread context to release the associated resources and free the memory used by the SPE context data structures.

```
    //Destroy context
    if (spe_context_destroy (threads[i].id) != 0)
    {
        sprintf(errmsg, "INTERNAL ERROR: failed to destroy context %d. Error =
        %s\n", i, strerror(errno));
        throw new FatalException(errmsg);
    }
```

Retrieve the results

Once the computations are complete, we collect and format the results. When the results are completely assembled, they are added to the output message object. This object is then passed to the `setTaskOutput()` method, which sends the results to the client.

```
    //Set output
    MyMessage outMsg;
    ostringstream ostr;
    ostr<<"Task output:\n";
    //append calculation result to output
    for (i=0; i<spus; i++)
    {
        ostr<<"SPU["<<((i+1)<10?"0":"")<<(i+1)<<"]:
        "<<threads[i].parms.taskOutput<<(i==(spus-1)?"":"\n");
    }
    outMsg.setString(ostr.str().c_str());
    // set our output message
    taskContext->setTaskOutput(outMsg);
}
```

Review the sample service makefiles

During the build process for the service code, the makefiles at `$SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Service/` perform the following actions:

1. Change the current directory to the directory in which the SPE code is located:

   ```
   cd $SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Service/spu/
   ```
2. Compile SPE code to an SPE object file:

   ```
   spu-gcc -I .. -c -o service_spu.o service_spu.c
   ```
3. Link the SPE object file to an SPE executable:

   ```
   spu-gcc -o service_spu service_spu.o
   ```
4. Convert the SPE executable to a PPE-embedded object file (using ppu-embedspu):

   ```
   ppu-embedspu -m32 service_spu service_spu service_spu-embed.o
   ```
5. Archive the PPU-embedded object file to a .a file:

   ```
   ppu-ar -qcs lib_service_spu.a service_spu-embed.o;
   ```
6. Change the current directory to the directory in which the PPE code is located:

   ```
   cd $SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Service/
   ```
7. Compile PPE code to a PPE object file:

   ```
   ppu-g++ -W -Wall -Winline -Wno-unused -m32 -g -DGCC34 -DLINUX -I . -I ../../../../../include -I ../
   Common -c -o SampleService.o SampleService.cpp
   ```
8. Link the PPE object file and PPE-embedded object file to a PPE executable:

   ```
   g++ -m32 -o ../Output/SampleService SampleService.o -lspe2 spu/lib_service_spu.a -L ../Output
   -L ../../../../../linux2.6-glibc2.5-ppc64_CellBE/lib32 -lsampleCommon -lsoambase -lsoamapi
   ```

Build, package, deploy, and run the sample client and service

You can build client application and service samples at the same time.

1. Log on to the client host.
2. Locate `ego.conf`. The default location is `/opt/symphonySDK/SDK51/conf/ego.conf`. Set EGO_MASTER_LIST and EGO_KD_PORT in `ego.conf`, as follows:

   EGO_MASTER_LIST= *master hostname*

   EGO_KD_PORT= same value as `$EGO_TOP/kernel/conf/ego.conf`
3. Set the environment variable:
   - For `csh`, enter

     ```
     setenv SOAM_HOME /opt/symphonySDK/SDK51/
     ```
   - For `bash`, enter

     ```
     export SOAM_HOME=/opt/symphonySDK/SDK51/
     ```

4. Change the current directory to the `conf` directory:

   cd $SOAM_HOME/conf/
5. Source the environment:

   - For `csh`, enter

     source cshrc.symclient
   - For `bash`, enter

     . profile. Symclient
6. Change the current directory to the directory in which the samples are located:

   cd $SOAM_HOME/5.1/samples/CPP/IBMCell/cell/
7. Compile using the Makefile located in `$SOAM_HOME/5.1/samples/CPP/IBMCell/cell`:

   make
8. Change the current directory to the directory in which the compiled samples are located:

   cd $SOAM_HOME/5.1/samples/CPP/IBMCell/cell/Output/
9. Create the service package by compressing the service executable into a tar file:

   tar -cvf SampleService.tar SampleService

   gzip SampleService.tar

   You have now created your service package `SampleService.tar.gz`.
10. Deploy the service package with the `soamdeploy` command:

    soamdeploy add SampleServiceCell -p SampleService.tar.gz -c /SampleApplications/SOASamples

    The service package is deployed.
11. Check the list of deployed services with the `soamdeploy view` command:

    soamdeploy view -c /SampleApplications/SOASamples
12. Register the application with the `soamreg` command:

    soamreg ../SampleAppCell.xml
13. Check the list of registered applications with the `soamview app` command:

    soamview app
14. Run the client application:

    ./SyncClient -i 1 -c 1 -t 1 -s 1

```
Start dispatching Symphony task with following parameters:
Task number: 1
Task input: 1
Common data input: 1
Operation on SPU:  Task input + Common data input
SPU number: 1
connection ID=84e4981e-ffff-ffff-c000-00145ef51544-4118186896-2324
Session ID:2301
task submitted with ID : 1
Task Succeeded [1]
Task output:
SPU[01]: 2
All Done !!
Session execution time = 8.31 seconds
```

Cluster configuration

Configure HostType and HostModel

To enable the master host to correctly identify the HostType and HostModel of a Cell BE host, the following two lines must be added into *$EGO_CONFDIR/*`ego.shared` on the master host:

1. Add “**LINUXCELLBE**” between “Begin HostType” and “End HostType”.
2. Add “**CELLBE 13.5 (CellBroadbandEngine)**” between “Begin HostModel” and “End HostModel". Note the spacing of the text on the line.

   Example:

   ```
   Begin HostModel
   MODELNAME  CPUFACTOR   ARCHITECTURE # keyword
   ……
   PC1133          23.1    (x6_1189_PentiumIIICoppermine)
   CELLBE          13.5    (CellBroadbandEngine)
   PC6000         116.1    (x15_5980_IntelRPentiumR4CPU300GHz)
   ……
   End HostModel
   ```

Configuring host slots

Symphony uses the number of CPUs to derive the default number of slots. You must configure EGO_DEFINE_NCPUS in *$EGO_CONFDIR/*`ego.conf` on the master host to set the correct number of CPUs for the Cell BE host.

To make full use of the SPE, the following two modes are recommended.

1. Define the number of slots based on the number of CPU cores. Use this mode when you want to effectively share the SPEs among sessions and applications.

   To define the number of slots based on CPU cores, set EGO_DEFINE_NCPUS=cores and create one SPE thread for each task.

   Since one Cell BE host has two processors and each processor has eight cores, one Cell BE host will have 16 slots. Therefore, up to 16 tasks can run on one Cell BE host concurrently. If one task only creates one SPE thread, the 16 tasks can make full use of the SPEs.
2. Define the number of slots based on the number of processors. Configuring one slot per multiple SPEs is advantageous if your program is making use of advanced multi-core optimizations to speed up calculations.

   To define the number of slots based on processors, set EGO_DEFINE_NCPUS=procs and create eight SPE threads for each task.

   Since one Cell BE host has two processors, one Cell BE host will have two slots. Therefore, up to two tasks can run on one Cell BE host concurrently. If one task creates eight SPE threads, the two tasks can make full use of the SPEs.

A

Appendix A: Symphony API Summary

This section summarizes the scope of the service API with regard to the SPE.

Symphony API	Symphony class	Availability on SPE
SoamException (void) throw ()	SoamException	Not available
virtual const char * what () const throw ()	SoamException	pass return value to SPE when starting SPE thread
virtual SOAM_HRESULT getHR () const throw ()	SoamException	pass return value to SPE when starting SPE thread
virtual int getErrorCode (void) const throw ()	SoamException	pass return value to SPE when starting SPE thread
virtual const char * getErrorType (void) const throw ()	SoamException	pass return value to SPE when starting SPE thread
SoamException * getEmbeddedException (void) const throw ()	SoamException	Not available
SoamException & operator= (const SoamException &rhs)	SoamException	Not available
SoamException * operator-> () const	SoamExceptionPtr	Not available
bool isNull () const	SoamExceptionPtr	pass return value to SPE when starting SPE thread
void setNull ()	SoamExceptionPtr	Not available
SoamExceptionPtr & operator= (SoamExceptionPtr rhs)	SoamExceptionPtr	Not available
SoamExceptionPtr & operator= (SoamException *rhs)	SoamExceptionPtr	Not available

Symphony API	Symphony class	Availability on SPE
bool operator== (const SoamExceptionPtr &rhs) const	SoamExceptionPtr	pass return value to SPE when starting SPE thread
bool operator== (const SoamException *rhs) const	SoamExceptionPtr	pass return value to SPE when starting SPE thread
bool operator!= (const SoamExceptionPtr &rhs) const	SoamExceptionPtr	pass return value to SPE when starting SPE thread
bool operator!= (const SoamException *rhs) const	SoamExceptionPtr	pass return value to SPE when starting SPE thread
operator SoamException * () const	SoamExceptionPtr	Not available
FailureException (void) throw ()	FailureException	Not available
FailureException (const char *errorDescription, int errorCode=0) throw ()	FailureException	Not available
FailureException & operator= (const FailureException &rhs)	FailureException	Not available
void applyCustomizedDebugAction (bool shouldApply)	FailureException	Not available
FailureException * operator-> () const	FailureExceptionPtr	Not available
FatalException (void) throw ()	FatalException	Not available
FatalException (const char *errorDescription, int errorCode=0) throw ()	FatalException	Not available
FatalException & operator= (const FatalException &rhs)	FatalException	Not available
void applyCustomizedDebugAction (bool shouldApply)	FatalException	Not available
FatalException * operator-> () const	FatalExceptionPtr	Not available
T * operator-> () const	SoamSmartPtr	Not available
bool isNull () const	SoamSmartPtr	pass return value to SPE when starting SPE thread
void setNull ()	SoamSmartPtr	Not available
SoamSmartPtr< T > & operator= (SoamSmartPtr< T > other)	SoamSmartPtr	Not available
SoamSmartPtr< T > & operator= (T *obj)	SoamSmartPtr	Not available
bool operator== (const SoamSmartPtr< T > &a) const	SoamSmartPtr	pass return value to SPE when starting SPE thread

Symphony API	Symphony class	Availability on SPE
bool operator!= (const SoamSmartPtr< T > &a) const	SoamSmartPtr	pass return value to SPE when starting SPE thread
virtual void onCreateService (ServiceContextPtr &serviceContext)	ServiceContainer	Not available
virtual void onDestroyService ()	ServiceContainer	Not available
virtual void onSessionEnter (SessionContextPtr &sessionContext)	ServiceContainer	Not available
virtual void onSessionLeave ()	ServiceContainer	Not available
virtual void onInvoke (TaskContextPtr &taskContext)=0	ServiceContainer	Not available
virtual void onServiceInterrupt (ServiceContextPtr &serviceContext)	ServiceContainer	Not available
void run (void *stack=0, size_t stackSize=0)	ServiceContainer	Not available
int run (int argc, char *argv[], void *stack=0, size_t stackSize=0)	ServiceContainer	Not available
const char * getServiceName () const throw (SoamException)	ServiceContext	pass return value to SPE when starting SPE thread
InterruptEvent getLastInterruptEvent (SoamULong &gracePeriod) const throw (SoamException)	ServiceContext	pass return value/OUT parameter value to SPE when starting SPE thread
void setControlCode (SoamInt code) throw (SoamException)	ServiceContext	Not available
const char * getApplicationName () const throw (SoamException)	ServiceContext	pass return value to SPE when starting SPE thread
const char * getConsumerId () const throw (SoamException)	ServiceContext	pass return value to SPE when starting SPE thread
const char * getDeployDirectory () const throw (SoamException)	ServiceContext	pass return value to SPE when starting SPE thread
const char * getLogDirectory () const throw (SoamException)	ServiceContext	pass return value to SPE when starting SPE thread
const char * getSessionId (void) const throw (SoamException)	SessionContext	pass return value to SPE when starting SPE thread

Symphony API	Symphony class	Availability on SPE
void getCommonData (Message &commonData) const throw (SoamException)	SessionContext	Not available
void populateCommonData (Message &commonData) const throw (SoamException)	SessionContext	Not available
void discardCommonData (void) throw (SoamException)	SessionContext	Not available
const char * getSessionId (void) const throw (SoamException)	TaskContext	pass return value to SPE when starting SPE thread
const char * getTaskId (void) const throw (SoamException)	TaskContext	pass return value to SPE when starting SPE thread
void getInputMessage (Message &inMsg) const throw (SoamException)	TaskContext	Not available
void populateTaskInput (Message &inputMessage) const throw (SoamException)	TaskContext	Not available
void setOutputMessage (Message &outMsg) throw (SoamException)	TaskContext	Not available
void setTaskOutput (Message &outputMessage) throw (SoamException)	TaskContext	Not available
void discardInputMessage (void) throw (SoamException)	TaskContext	Not available
	TaskContextPtr	
virtual void onSerialize (OutputStreamPtr &stream)=0 throw (SoamException)	Message	Not available
virtual void onDeserialize (InputStreamPtr &stream)=0 throw (SoamException)	Message	Not available
DefaultBinaryMessage (void)	DefaultBinaryMessage	Not available
DefaultBinaryMessage (const char *buffer, unsigned long length, bool shouldCopy=false)	DefaultBinaryMessage	Not available
void onSerialize (OutputStreamPtr &stream) throw (SoamException)	DefaultBinaryMessage	Not available
void onDeserialize (InputStreamPtr &stream) throw (SoamException)	DefaultBinaryMessage	Not available
void setBuffer (const char *buffer, unsigned long length, bool shouldCopy=false)	DefaultBinaryMessage	Not available

Symphony API	Symphony class	Availability on SPE
const char * getBuffer (unsigned long &length) const	DefaultBinaryMessage	pass return value/OUT parameter value to SPE when starting SPE thread
DefaultByteArrayMessage (void)	DefaultByteArrayMessage	Not available
DefaultByteArrayMessage (const void *byteArray, SoamUInt length, SoamBool shouldCopy=false)	DefaultByteArrayMessage	Not available
void onSerialize (OutputStreamPtr &stream) throw (SoamException)	DefaultByteArrayMessage	Not available
void onDeserialize (InputStreamPtr &stream) throw (SoamException)	DefaultByteArrayMessage	Not available
void setByteArray (const void *byteArray, SoamUInt length, SoamBool shouldCopy=false)	DefaultByteArrayMessage	Not available
void getByteArray (char *&byteArray, SoamUInt &length, SoamBool shouldCopy=false) const	DefaultByteArrayMessage	pass OUT parameter value to SPE when starting SPE thread
DefaultTextMessage (void)	DefaultTextMessage	Not available
DefaultTextMessage (const std::string &text)	DefaultTextMessage	Not available
DefaultTextMessage (const char *text)	DefaultTextMessage	Not available
void onSerialize (OutputStreamPtr &stream) throw (SoamException)	DefaultTextMessage	Not available
void onDeserialize (InputStreamPtr &stream) throw (SoamException)	DefaultTextMessage	Not available
void setText (const std::string &text)	DefaultTextMessage	Not available
void setText (const char *text)	DefaultTextMessage	Not available
void getText (std::string &text) const	DefaultTextMessage	pass OUT parameter value to SPE when starting SPE thread
const char * getText () const	DefaultTextMessage	pass return value to SPE when starting SPE thread
operator const char * () const	DefaultTextMessage	pass return value to SPE when starting SPE thread
bool getNext (TaskOutputHandlePtr &taskOutputHandle) throw (SoamException)	EnumItems	pass return value to SPE when starting SPE thread
void reset () throw (SoamException)	EnumItems	Not available

Symphony API	Symphony class	Availability on SPE
void skip (SoamULong skipCount) throw (SoamException)	EnumItems	Not available
SoamULong getCount () throw (SoamException)	EnumItems	pass return value to SPE when starting SPE thread
EnumItemsPtr clone () throw (SoamException)	EnumItems	Not available
virtual void read (short &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (unsigned short &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (int &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (unsigned int &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (long &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (unsigned long &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (long long &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (unsigned long long &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (float &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (double &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (bool &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (char &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread

Symphony API	Symphony class	Availability on SPE
virtual void read (char *&x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void read (std::string &x) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void readBytes (void *x, unsigned long length) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void readByteArray (char *&x, unsigned long &length) throw (SoamException)	InputStream	pass OUT parameter value to SPE when starting SPE thread
virtual void write (short x) throw (SoamException)	OutputStream	Not available
virtual void write (unsigned short x) throw (SoamException)	OutputStream	Not available
virtual void write (int x) throw (SoamException)	OutputStream	Not available
virtual void write (unsigned int x) throw (SoamException)	OutputStream	Not available
virtual void write (long x) throw (SoamException)	OutputStream	Not available
virtual void write (unsigned long x) throw (SoamException)	OutputStream	Not available
virtual void write (long long x) throw (SoamException)	OutputStream	Not available
virtual void write (unsigned long long x) throw (SoamException)	OutputStream	Not available
virtual void write (float x) throw (SoamException)	OutputStream	Not available
virtual void write (double x) throw (SoamException)	OutputStream	Not available
virtual void write (bool x) throw (SoamException)	OutputStream	Not available
virtual void write (char x) throw (SoamException)	OutputStream	Not available
virtual void write (const char *x) throw (SoamException)	OutputStream	Not available
virtual void write (const std::string &x) throw (SoamException)	OutputStream	Not available
virtual void writeBytes (const void *x, unsigned long length) throw (SoamException)	OutputStream	Not available
virtual void writeByteArray (const char *x, unsigned int offset, unsigned int length) throw (SoamException)	OutputStream	Not available

Index