Software Development Kit for Multicore Acceleration
Version 3.1

IDE
Tutorial and User's Guide

SC34-2561-00

Software Development Kit for Multicore Acceleration
Version 3.1

IDE
Tutorial and User's Guide

SC34-2561-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 153.

Edition notice

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

Contents

About this publication

This publication describes in detail how to use the IBM Software Development Kit for Multicore Acceleration Integrated Development Environment for Cell Broadband Engine™ (IDE).

Who should use this book

The target audience for this document is application programmers using the IBM Software Development Kit for Multicore Acceleration (SDK). You are expected to have a basic understanding of programming on the Cell Broadband Engine (Cell/B.E.) platform and common terminology used with the Cell/B.E. platform.

Typographical conventions

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

Table 1. Typographical conventions

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

Related information

For a list of SDK documentation, see Related documentation.

How to send your comments

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

Part 1. Getting started

These topics describe what IDE is, the hardware requirements and how to install IDE.

- Chapter 1, “Overview of the IBM SDK for Multicore Acceleration Integrated Development Environment,” on page 3
- Chapter 2, “Supported operating environments,” on page 5
- Chapter 3, “Installing IDE,” on page 7
- Chapter 4, “Uninstalling IDE,” on page 11

Chapter 1. Overview of the IBM SDK for Multicore Acceleration Integrated Development Environment

The IBM® SDK for Multicore Acceleration Integrated Development Environment (IDE) is built upon the Eclipse and C Development Tools (CDT) platform.

It integrates the Cell/B.E. GNU tool chain, compilers, IBM Full-System Simulator (simulator) for the Cell/B.E. processor, and other development components to provide a comprehensive, user-friendly development platform that simplifies Cell/B.E. application development. The key features include the following:

- A C/C++ editor that supports syntax highlighting, a customizable template, and an outline window view for procedures, variables, declarations, and functions that appear in the source code
- A rich visual interface for the Power Processing Element (PPE), Synergistic Processing Element (SPE), and the GNU debugger (GDB)
- A seamless integration of simulator into Eclipse
- An automatic builder, performance tools, and several other enhancements

Chapter 2. Supported operating environments

The following are the prerequisites for using IDE for all operating environments.

Architecture: x86, x86_64, PPC, PPC_64, Cell/B.E.

Operating System: Linux®

Software:
- Cell BE SDK 3.1
- Java™ 1.5 (IBM or Sun.)

Note: The Java™ that comes installed in some Linux® installations (GCJ) is *not* sufficient.

Chapter 3. Installing IDE

This topic describes how to install IDE.

If you have installed the SDK and wants to use the optional IES Eclipse IDE component, install IDE by following these steps:

1. The GCJ JRE installed with Linux distributions is *not* sufficient for IDE. Download and install J2SE Version 5.0, SR8 or greater for 32–bit xSeries® from `http://www.ibm.com/developerworks/java/jdk/linux/download.html`. SDK provides an IBM 32-bit JRE 1.5 SR8 that you can install by invoking the following commands as root (admin):
   - `yum install ibm-java2-ppc-jre` (PowerPc version)
   - `yum install ibm-java2-i386-jre` (i386 version)
2. Configure your environment to use the new JRE.
   a. Eclipse uses the Java virtual machine (JVM) to which your Linux **PATH** variable points. To find out which version of Java is being used, run the command:
      ```
      java -version
      ```
      If this returns a JVM other and GCJ, you can go to next step.
   b. When you start Eclipse, use the -vm command to specify the IBM JVM. For example:
      ```
      eclipse -vm /opt/java/jdk1.5.0_08/jre/bin
      ```
   c. If you want to modify your environment permanently, configure your **PATH** variable to reference the IBM JVM. You can modify your **PATH** variable by editing the file $HOME/.bash_profile. For example, add this line:
      ```
      PATH=$HOME/bin:/opt/java/jdk1.5.0_08/jre/bin:$PATH
      ```
3. Install the IDE rpm. To install the IDE rpm, log in as root (admin) user and perform one of the two installation methods:
   - Download the rpm from the developerWorks® IDE site. To install it, use the rpm command:
     ```
     > rpm -ivh cellide-3.1.0-[RELEASE_NUM].[ARCHITECTURE].rpm
     ```
   - If you have the cell-install rpm installed, you can use yum as follows:
     ```
     > yum install cellide
     ```

   This procedure installs an IES Eclipse bundled with CDT 4.0, PTP Remotetools 2.0 and the latest IDE plug-ins into the `/opt/cell/ide/eclipse` directory.
4. Install the Accelerated Library Framework (ALF) IDE template, Performance Debugging Tool (PDT) and SPU Timing packages.
   a. If you want to use the ALF IDE Wizard, you need to install the ALF IDE template rpm (alf-ide-template). Make sure you have the SDK's cell-install rpm package installed, then invoke the following command:
      ```
      > yum install alf-ide-template
      ```
   b. To enable the PDT feature on IDE, make sure you have the SDK's cell-install rpm package installed, then invoke the following command:
      ```
      yum install pdt
      ```
   c. To use the SPU Timing feature, make sure you have the SDK's cell-install rpm package installed, then invoke the following command:

```
yum install cell-spu-timing
```

What to do next: For more information about IDE, see the Eclipse help topic.

To access IDE help, in Eclipse, click: **Help → Help Contents → IDE for Cell Broadband Engine SDK**.

Troubleshooting: 32–bit IES Eclipse fails to load on 64–bit Fedora 9

This occurs because there are no specific 32–bit libraries provided with the Fedora 9 x86_64 version.

Solution:

1. Install the cellide package.
2. Install libcairo 32–bit version:
   ```
   yum install libcairo.i386
   ```
3. Install libXtst 32–bit version:
   ```
   yum install libXtst.i386
   ```

Troubleshooting: 32–bit IES Eclipse fails to load on x86 with RHEL 5.2

This is a known issue caused by a library called xulrunner. The original installation of this library in RHEL 5.2 makes 32-bit IES Eclipse to crash. To solve this problem, remove the original version of `xulrunner` from your RHEL 5.2 installation:

```
yum erase xulrunner
```

Chapter 4. Uninstalling IDE

This topic describes how to uninstall IDE for all platforms.

To uninstall IDE, you must remove the cellide package from the system. You can do this in many ways, for example:

- As root user, issue the command:
  ```
  yum erase cellide
  ```
- Use any graphical packaging management tool to remove the cellide package

When you remove the cellide package, the following components are deleted:

- IES Eclipse bundled with CDT 4.0 and PTP Remotetools 2.0
- The IDE plug-ins installed in the /opt/cell/ide/eclipse directory

Note: The IDE uninstallation process does not erase any plug-ins or workspaces located on the user's space, but it does erase all the contents of /opt/cell/ide/eclipse and its subdirectories.

Part 2. Using IDE

These topics describe how to use IDE to run and debug applications.

It describes the following topics:

- Chapter 5, “Creating a Cell/B.E. environment,” on page 15
- Chapter 6, “Running IDE on Cell/B.E. systems,” on page 17
- Chapter 7, “Running IDE on Power-PC systems,” on page 21
- Chapter 8, “Using the CDT XLC error parser,” on page 23
- Chapter 9, “Configuring public key-based authentication using OpenSSH,” on page 25
- Chapter 10, “Troubleshooting and work-arounds,” on page 27

Chapter 5. Creating a Cell/B.E. environment

This topic describes the Cell/B.E. environments in which you can run IDE.

You can use IDE to run and debug Cell/B.E. applications in any of the following types of Cell/B.E. environments:

- **Attached Cell/B.E. simulator** - Cell/B.E. simulator running on a remote system. You must manually start the simulator on the remote system. This is different to the **remote Cell/B.E. simulator** environment, where IDE starts the simulator on the remote host.
- **Cell/B.E. box** - Physical Cell/B.E. system (remote or native Cell/B.E. system running IDE).
- **Local Cell/B.E. simulator** - Cell/B.E. simulator running on the native system.
- **Remote Cell/B.E. simulator** - Cell/B.E. simulator running on a remote system. IDE starts the remote simulator. This is different to the **attached Cell/B.E. simulator** environment, whereby you must start the simulator manually on the remote host.

Before you can run or debug a Cell/B.E. application, you must create and start a Cell/B.E. environment as following:

1. Open the **Cell Environments** view.
2. Right click any of the four available Cell/B.E. machine types, then select **Create**.
3. Enter any necessary configuration values, then click **Finish**.
4. Start the environment. To do this, click the green **Start** arrow button.

   After the Cell/B.E. environment has started, it is ready to be used for Cell/B.E. application deployment.

More information:

- Running IDE on Cell/B.E. systems
- Running IDE on Power-PC systems

Chapter 6. Running IDE on Cell/B.E. systems

IDE can be run on a system with a Cell/B.E. processor (as of version 3.0).

The following topics describe how to run IDE on such a system:

- "Managing builder paths for Cell/B.E. systems"
- "Debugging Cell/B.E. applications"
- "Running Cell/B.E. applications" on page 19

Managing builder paths for Cell/B.E. systems

The GNU and XL tool paths are at different locations on a native Cell/B.E. system than on a ″typical″, non-Cell/B.E. system. IDE's default paths reflect the default Cell/B.E. SDK tool paths of a non-Cell/B.E. computer, so these paths must be corrected. This can be done as follows:

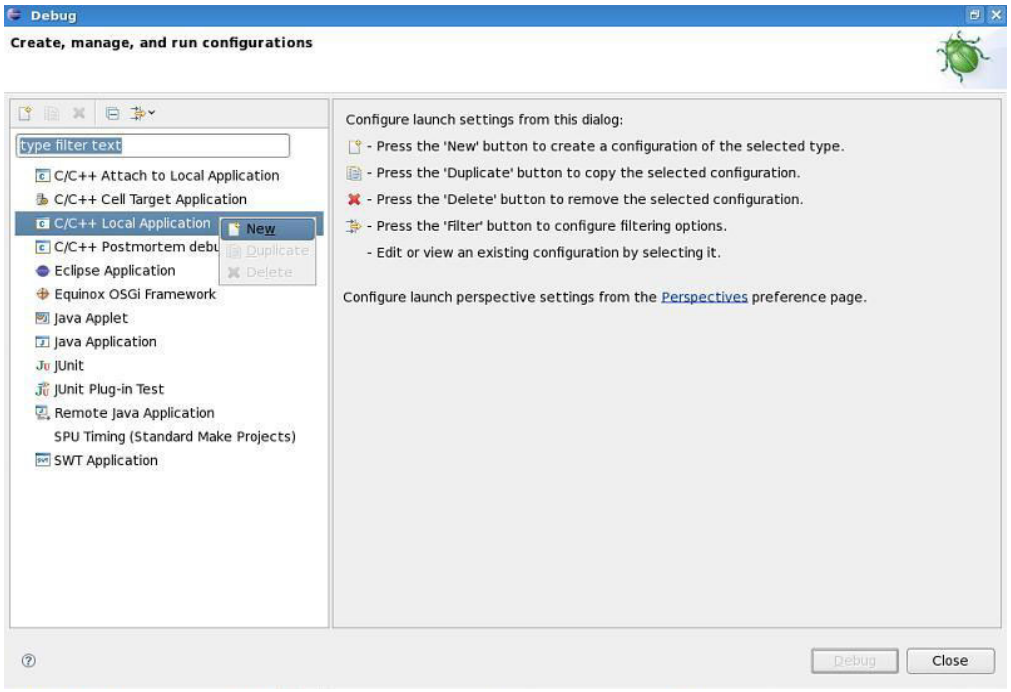
1. After installing IDE, open Eclipse preferences. To do this, click **Window → Preferences**.
2. Open the **GNU Tools Managed Build Paths** preference page: **Cell → Managed Builder Paths → GNU Tools Managed Build Paths**.
3. Click **Find it!** to automatically find the GNU tools location, or click **Browse** to manually select the location
4. Do the same for the **XL Tools Managed Build Paths** preference page

Debugging Cell/B.E. applications

Debugging Cell/B.E. applications while running the IDE on a native Cell/B.E. machine requires a few changes to the debugger configuration in order to function properly. Unlike attempting to debug Cell/B.E. applications remotely via SSH, debugging locally does not require the use of gdbserver. Also, instead of creating a **C/C++ Cell Target Application** debug configuration, you can simply create a **C/C++ Local Application** configuration (see Figure 1 on page 18).

Note: Other debugger configurations and combinations are possible, only one of them is described here.

1. Click **Run → Debug...** to open the debug configuration window.
2. Create a new **C/C++ Local Application** debug configuration (Figure 1 on page 18).



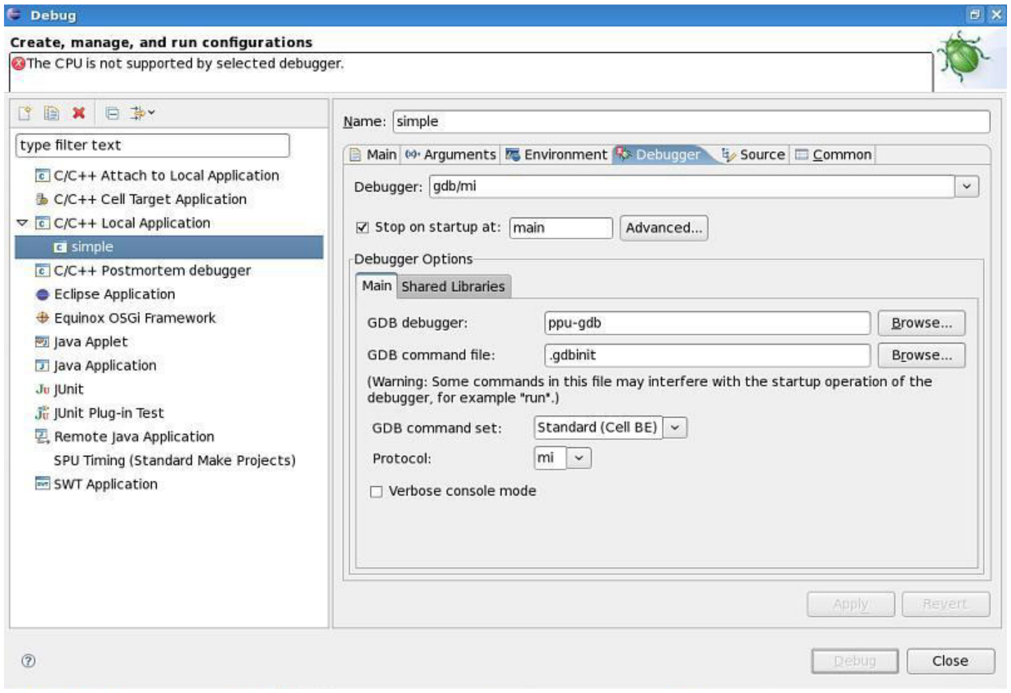
Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

Figure 1. Creating a new C/C++ Local Application

After you have created a new **C/C++ Local Application** launcher, you must properly configure the **Debugger** tab (see Figure 2 on page 19):

1. Select the **gdb/mi** debugger.
2. Change the **GDB Debugger** to **ppu-gdb** in order to be able to debug the SPU parts of the application as well as the PPU part.
3. Change the **GDB command set** field to **Standard (Cell BE)**.


Debug

Create, manage, and run configurations

The CPU is not supported by selected debugger.

type filter text

- C/C++ Attach to Local Application
- C/C++ Cell Target Application
- C/C++ Local Application
  - simple
- C/C++ Postmortem debugger
- Eclipse Application
- Equinox OSGi Framework
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Remote Java Application
- SPU Timing (Standard Make Projects)
- SWT Application

Name: simple

Main | Arguments | Environment | Debugger | Source | Common

Debugger: gdb/mi

☑ Stop on startup at: main   Advanced...

Debugger Options

Main | Shared Libraries

GDB debugger: ppu-gdb   Browse...

GDB command file: .gdbinit   Browse...

(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

GDB command set: Standard (Cell BE)

Protocol: mi

☐ Verbose console mode

Figure 2. Correctly configured Debugger tab

Running Cell/B.E. applications

To run applications locally on a Cell/B.E. system, you can use either the **C/C++ Local Application** or the **C/C++ Cell Target Application** launch configuration.

The **C/C++ Cell Target Application** launcher was initially designed to allow for the running/debugging of applications on a Cell/B.E. simulator or system, while the development of that application was done on a computer without a Cell/B.E. processor. However, you can use this same launcherto deploy an application locally if you are running IDE on a system with a Cell/B.E. processor. In order to accomplish this, you need to create a **Cell Box** environment (in the **Cell Environments** view) with the configuration option **Localhost** selected (instead of **Remote host** being selected). After you have clicked the green arrow to start the localhost **Cell Box**, it can then be used as the target environment in the **C/C++ Cell Target Application** launcher.

Chapter 7. Running IDE on Power-PC systems

IDE can be run on a Power-PC (PPC) system.

The following topics are worth discussing if you plan to use this platform to run the IDE:

- "Managing builder paths for a PPC system"

This section discusses any issues that may come up when running the Cell IDE on a Power-PC (PPC) system.

Managing builder paths for a PPC system

The SDK's GNU and XL tool paths are at different locations on a PPC-based system than on systems with Intel® processors. IDE's default paths reflect the default Cell/B.E. SDK tool paths of a Intel-based computer, so these paths must be corrected if you wish to run IDE on PPC. This can be done as follows:

1. After installing IDE, open Eclipse preferences. To do this, click **Window → Preferences**.
2. Open the **GNU Tools Managed Build Paths** preference page: **Cell → Managed Builder Paths → GNU Tools Managed Build Paths**.
3. Click **Find it!** to automatically find the GNU tools location, or click **Browse** to manually select the location
4. Do the same for the **XL Tools Managed Build Paths** preference page

Chapter 8. Using the CDT XLC error parser

This topic describes how to use the CDT XLC error parser with IDE.

The IDE is integrated with the Cell/B.E. GNU compilers and also with the Cell BE XL compilers. The error messages printed by these compilers are parsed by the appropriate error message parser in order to identify the error information and display it correctly in the graphical user interface. This makes it easier for you to solve problems that the compilers encountered with your source code.

The GNU C Compiler error message parser comes integrated with CDT. Previous versions of the CDT did not support XL C/C++ compiler error messages and for this reason IDE provides its own XLC error parser. However, from version 3.1.2 onwards CDT provides an XL C/C++ error parser, which you can install from the CDT update site. If you prefer to use this XL C/C++ error parser instead of the one provided by IDE, you need to follow the following instructions to the install CDT XLC error parser as it is not installed by default when CDT is installed:

1. Install Java, Eclipse and CDT as described in steps 1 through 4 of the IBM SDK for Multicore Acceleration IDE Installation Instructions.

   Note: The CDT XLC Error Parser is available from **CDT version 3.1.2** onwards.
2. Start Eclipse. Make sure you start Eclipse with a user that has write permission to the Eclipse installation directory.
3. From the menu, select **Help → Software Updates → Find and Install...**
4. Select **Search for new feature to install**, then press **Next**.
5. Click **New Remote Site...**.
6. Enter a name in the **Name** field that identifies the update site, for example, **CDT Update Site**.
7. Enter the following into the **URL** field
   ```
   http://download.eclipse.org/tools/cdt/releases/callisto
   ```
8. Click **OK**. The recently created entry appears selected in the **Sites to include in search** field.
9. Click **Finish** and follow the on-screen instructions.

After you have completed the above steps, a **CDT XLC Error Parser** entry appears in the **Error Parser** tab. You can access this tab through any of the following:

- Select **Window → Preferences... → C/C++ → Make → New Make Projects**
- From the **C/C++ Build** dialog page of a **Managed Make C/C++** project Properties
- From the **C/C++ Make Project** dialog page of a **Standard Make C/C++** project Properties
- From the final dialog of the project creation wizard for **Managed Make C/C++** and **Standard Make C/C++** projects.

From this **Error Parser** tab, you can deselect the IDE **XLC Error Parser** and select the **CDT XLC Error Parser**.

Chapter 9. Configuring public key-based authentication using OpenSSH

The following topics describe how to configure your client and server (as well as how to configure the IDE's environments) to use the public key authentication method.

IDE establishes connections to remote Cell/B.E. environments using Secure Shell (SSH). Two SSH authentication methods are supported:

- Password-based
- Public key-based

Step 1. Verify the software

First, confirm that OpenSSH is the SSH software installed on the client system. Public key generation can be different for different implementations of SSH. Use the `ssh-V` command to check the version of your SSH software as follows:

```
> $ ssh -V
OpenSSH_4.3p2, OpenSSL 0.9.8b 04 May 2006
```

Step 2. Generate a key pair

You must generate a RSA public/private key pair on the client system. The public key is copied to the remote server that is being connected to, the private key remains on the client system in a secure location. Use the `ssh-keygen` command to generate the key pair as follows:

```
client$ mkdir ~/.ssh
client$ chmod 700 ~/.ssh
client$ ssh-keygen -q -f ~/.ssh/id_rsa -t rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Step 3. Distribute the key

Next, copy the public key `~/.ssh/id_rsa.pub` to the remote server. After the public key has been copied, you need to append the key's contents to the file ~/.ssh/authorized_keys:

```
client$ scp ~/.ssh/id_rsa.pub user@server.example.com:
client$ ssh user@server.example.com

server$ mkdir ~/.ssh
server$ chmod 700 ~/.ssh
server$ cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
server$ chmod 600 ~/.ssh/authorized_keys
server$ rm ~/id_rsa.pub
```

Step 4. Confirm the functionality

Attempt to connect to the server to confirm that the public key authentication method is working. If you are using the public key authentication method, you are asked for the passphrase when you connect:

```
client$ ssh -o PreferredAuthentications=publickey
user@server.example.edu
Enter passphrase for key '/home/user/.ssh/id_rsa':
```

Step 5. Configure the Cell/B.E. environment

Finally, you need to configure the Cell/B.E. environment inside the IDE.

Note: Public key authentication can be used on any of the Cell/B.E. target environment types except for the **Local Cell Simulator** type.

The following steps explain how to configure your Cell/B.E. Environment:

1. In Eclipse, open the **Cell Environments** view.
2. Open the cell environment's configuration window by either creating a new environment (right click environment type and select **Create**) or by editing an existing environment (right click the existing environment and select **Edit**).
3. Select the option **Public key based authentication**.
4. Press the **Browse** button next to the field **File with private key**, and select the private key file ~/.ssh/id_rsa.
5. Enter the passphrase that you entered when you created the public/private key pair.
6. Click **Finish** and use the green start arrow to start the environment.

Chapter 10. Troubleshooting and work-arounds

This topic describes solutions to problems you can encounter when you use IDE.

The following are known issues with IDE:
- "Using the Java API with the simulator"
- "Reconnecting to the GDB server"
- "Ensuring that the remote launch directory is writable for all users"
- "Configuring the localhost" on page 28
- "Specifying MAC addresses for the local Cell/B.E. simulator" on page 28

Using the Java API with the simulator

There is a known bug with the Cell/B.E. simulator's capacity to use the Java API along with the simulator GUI window. To enable the use of the Java API with the simulator, you must set the option **USE_JAVA_API** to true:

```
USE_JAVA_API=true
```

This parameter can be found in the file **parameters.properties**, which is located at:

```
eclipse/plugins/com.ibm.celldt.simulator/com/ibm/celldt/simulator/conf
```

If the **USE_JAVA_API** option is set to true, the simulator GUI window cannot be used.

Reconnecting to the GDB server

If you are debugging a Cell/B.E. application remotely using gdbserver and the connection to the gdbserver is broken, IDE tries to reconnect automatically. However, this is not successful even though the gdbserver is still running on the target system.

If you stop this debug session and try to start a new one, IDE issues an error because the gdbserver is still occupying the previous configuation port on the target system and IDE is not able to launch another instance of gdbserver in the same port unless the debug launcher configuration is modified.

The workaround for this is to kill the gdbserver application on the target system before you launch another debug session again.

Ensuring that the remote launch directory is writable for all users

The Cell/B.E. Target Application launcher automatically creates a remote directory for launching the application. By default, this directory is given by following macros:

```
${system_workspace}/${user_workspace}/${project_name}/${timestamp}
```

If you are sharing the remote (Cell/B.E.) system, make sure that all users can write to the directory provided by `${system_workspace}`.

The first user that connects to the machine creates `${system_workspace}`, subsequent users create their directories inside `${system_workspace}`. However, if `${system_workspace}` was not created with full write privileges, then the subsequent users will not be able to launch.

Configuring the localhost

When IDE uses the Cell/B.E. Target Application launcher and Local Cell/B.E. Simulator, it calls `Inet4Address.getLocalHost()` several times. In some cases, this method can fail if your system is not configured correctly. This failure can result in IDE refusing to create new simulator environments or cause failures or both with the Cell/B.E. Target Application launcher.

If you encounter this problem, check the /etc/hosts file and ensure that the following line is present:

```
127.0.0.1 localhost mymachine
```

where `mymachine` is the name of the system as given by the environment variable $HOSTNAME.

Specifying MAC addresses for the local Cell/B.E. simulator

The local simulator configuration window allows you to choose your own MAC address. If you decide to specify your own MAC address, the following rules apply:

- The first number of the address MUST NOT be odd. It MUST be even. It should be n*4+2 (02, 06, 10, ...)
  - If the first number is odd, then SSH will not connect, even if simulator pings successfully
  - If the first digit is not n*4+2, then it does not conform to RFC
- It is recommended that the second and third numbers are 00
- The recommended MAC addresses are: 02-00-00-00-00-00, 02-00-00-00-00-01, 02-00-00-00-00-02, 02-00-00-00-00-03

Part 3. IDE tutorial

Chapter 11. IDE tutorial

This introductory tutorial explores IDE and provides click-by-click lessons on how to use many of the main features of the IDE.

Purpose of this tutorial

In this tutorial you will learn how to:
- Create, build, and run PPU/SPU managed make projects
- Use the local Cell/B.E. simulator environment
- Use the performance analysis tools (static and dynamic)
- Configure and use the **C/C++ Cell Target Application** launcher to run and debug your Cell/B.E. applications
- Use the ALF IDE wizard by means of an example usage scenario
- Use the IDE PDT to generate trace profiles

Preparation

Before you start with the tutorial you must prepare your system. You must have IDE installed and running on the computer you want to use for the tutorial. For information about how to install IDE, refer to Chapter 3, “Installing IDE,” on page 7.

Learning time

You should allow two and a half hours to complete the tutorial.

Conventions

The following conventions are used in the task descriptions:
- Select **File → New** means ″Select item ′New′ from the ′File′ menu.″
- Click **OK** means ″click the OK button″.

Table of contents

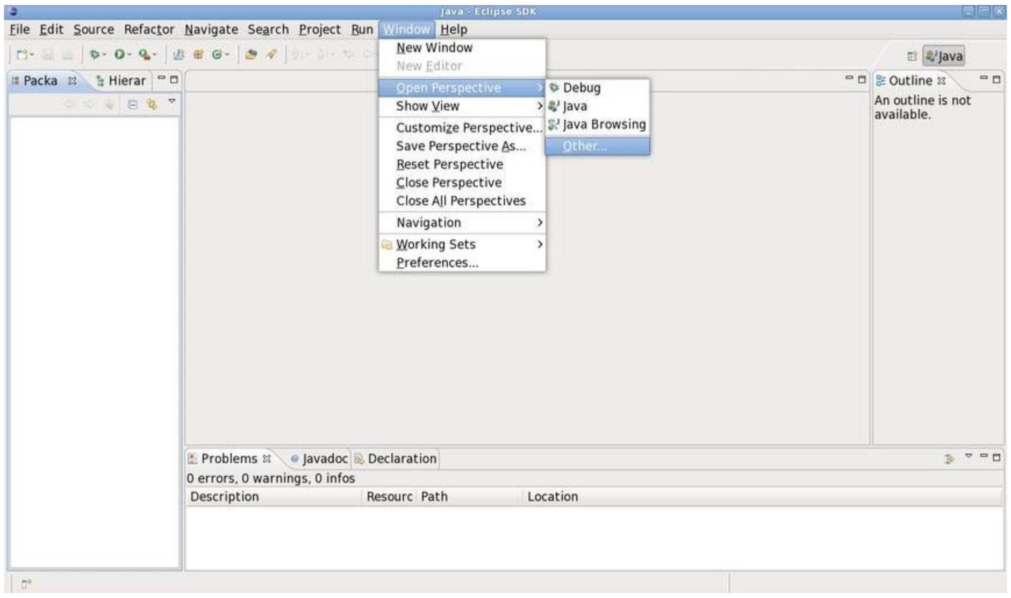
Click one of the following sections to jump to a particular section of the tutorial, or click here to start at the beginning.

You can return to the table of contents at any time by clicking the ″Parent topic″ link in the tutorial.

10. “View the IDE environment preferences” on page 119
11. “IDE PDT plugin” on page 104

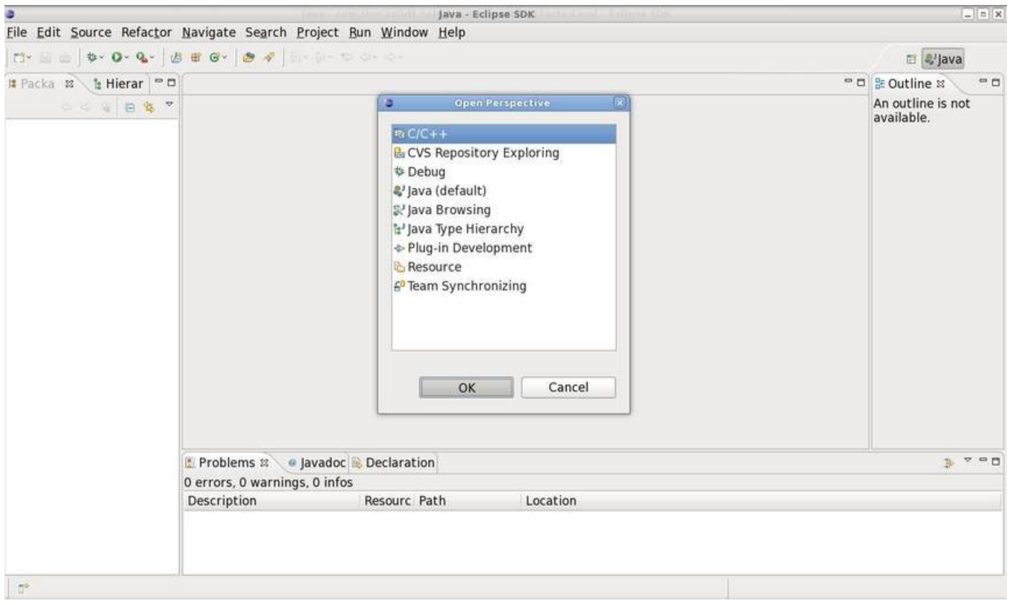
Getting started - open Eclipse

With CDT and IDE installed into your eclipse directory, open Eclipse and switch to the C/C++ perspective. To do this click **Window → Open Perspective → Other...**



Getting started - select C/C++ Perspective

Select **C/C++**, then click **OK**.



An outline is not available.

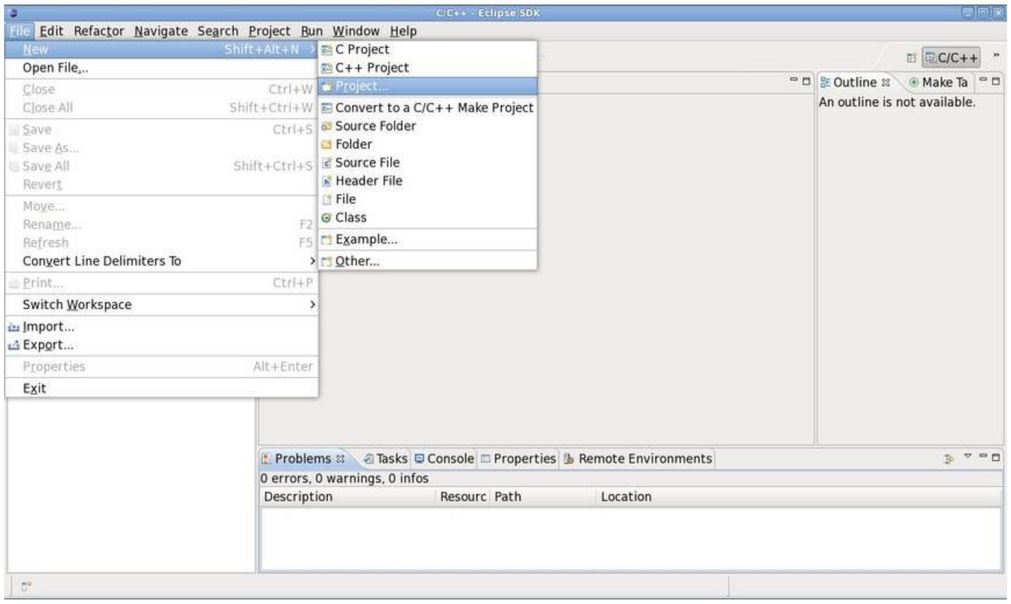
0 errors, 0 warnings, 0 infos

SPU project - create SPU project

Objectives: After you have completed this part of the tutorial you will know how to:

- Create and configure an SPU project
- Add the directory that contains the profile.h header file to the list of the compiler's include paths so that you can use the dynamic performance analysis tool
- Create a C source file
- Automatically build the project

Create a new C project. Click **File → New → Project**.



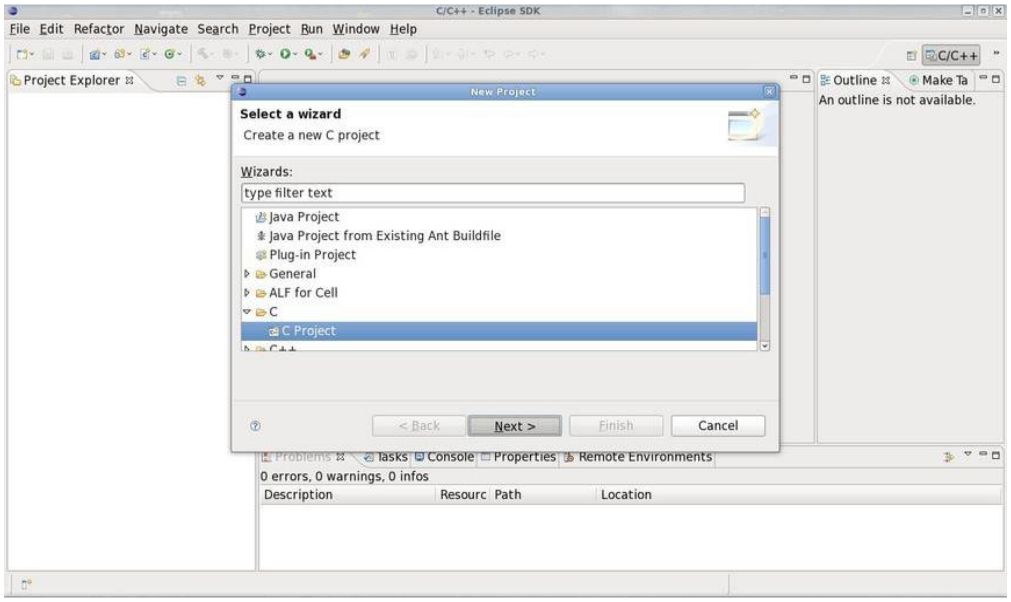
An outline is not available.

0 errors, 0 warnings, 0 infos

SPU project - select the C Project Wizard

The **New Project** wizard is displayed.

1. Expand the section **C**.
2. Select **C Project**.



Create a new C project

type filter text

An outline is not available.

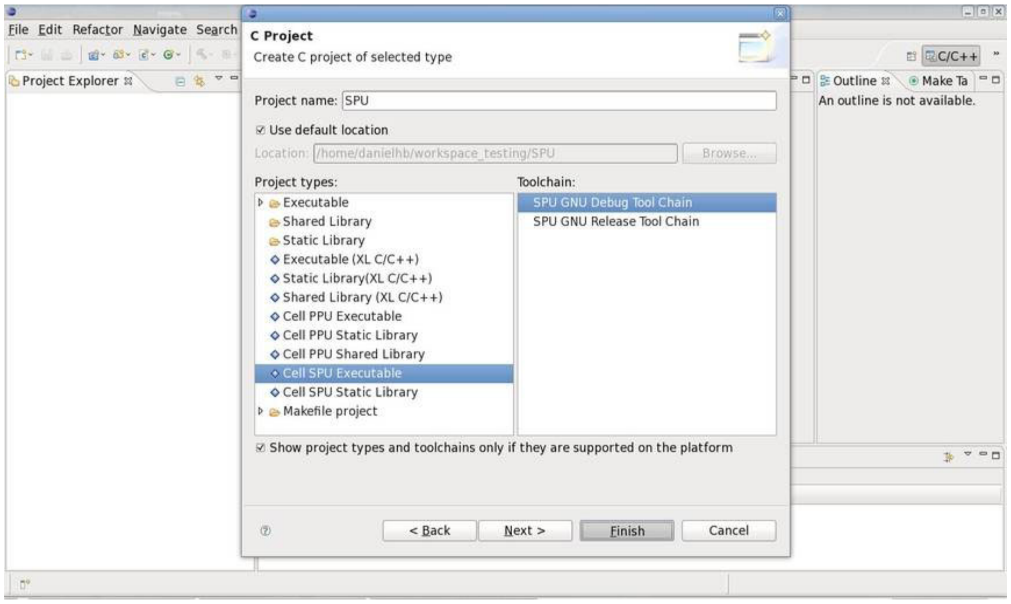
0 errors, 0 warnings, 0 infos

3. Click **Next**.

SPU project - enter the project name and define the project type

For **Project name**, enter SPU. You can select the appropriate **Project Type** (Cell PPU Executable, Cell SPU Static Library, and so on) for the project you are creating. When you select a project type, you see the default **Configurations** available for that project type. Later in the tutorial you will learn how to create new build configurations, as well as how to modify the existing built-in configurations' settings.

1. For **Project Type**, select **Cell SPU Executable**.
2. Click **Finish**.



C Project
Create C project of selected type

Project name: SPU

☑ Use default location

Location: /home/danielhb/workspace_testing/SPU

Project types:
- Executable
- Shared Library
- Static Library
- Executable (XL C/C++)
- Static Library(XL C/C++)
- Shared Library (XL C/C++)
- Cell PPU Executable
- Cell PPU Static Library
- Cell PPU Shared Library
- Cell SPU Executable
- Cell SPU Static Library
- Makefile project

Toolchain:
- SPU GNU Debug Tool Chain
- SPU GNU Release Tool Chain

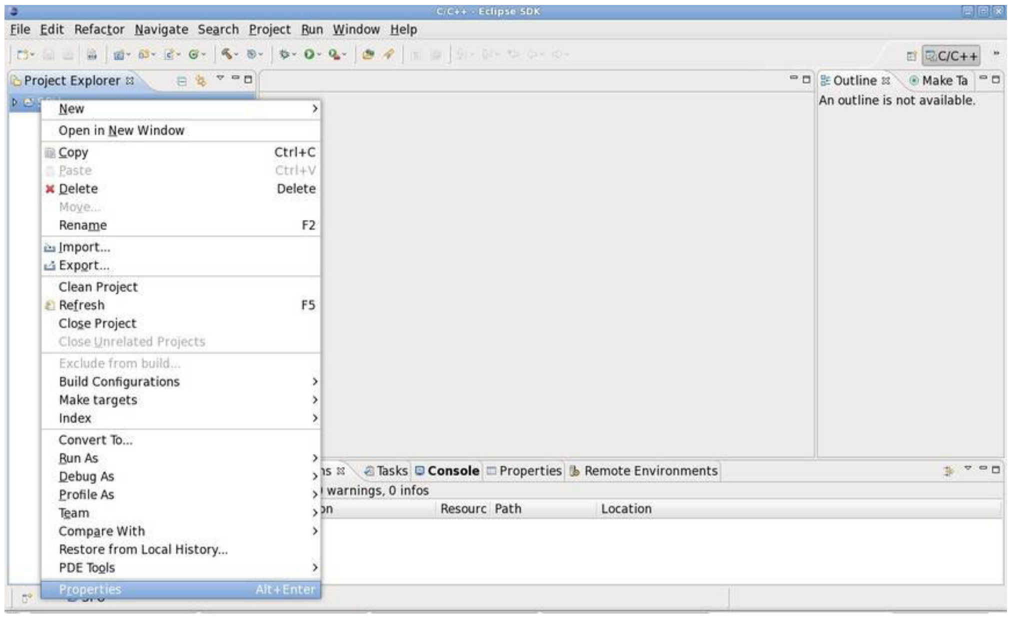
☑ Show project types and toolchains only if they are supported on the platform

An outline is not available.

SPU project - configure SPU Project

In the **C/C++ Projects** view:

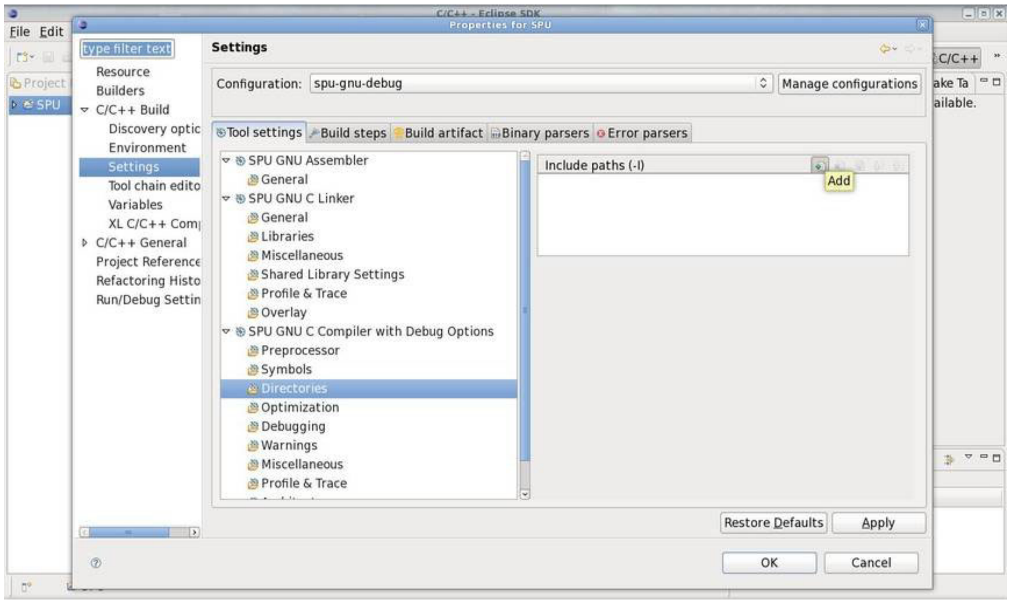
1. Right click the **SPU** project.
2. Select **Properties**.



SPU project - add directory the SPU compiler include paths list

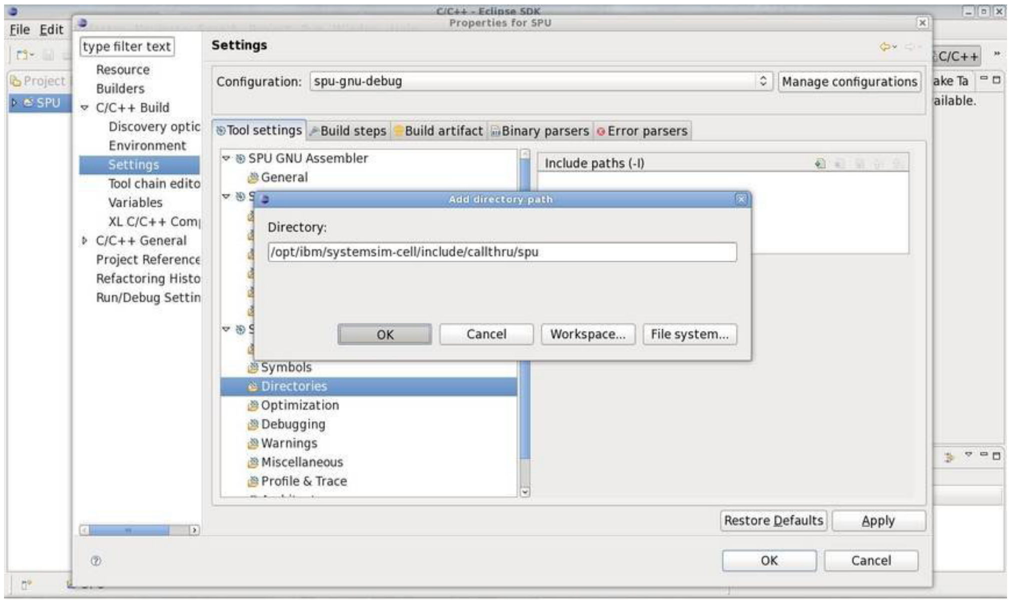
You need to add the directory that contains the **profile.h** header file to the list of the compiler's include paths so that you can use the dynamic performance analysis tool.

1. Click **C/C++ Build** in the left pane.
2. Under **SPU GNU C Compiler with Debug Options**, select **Directories**.
3. In the **Include Paths** pane on the right, click the **Add** button.



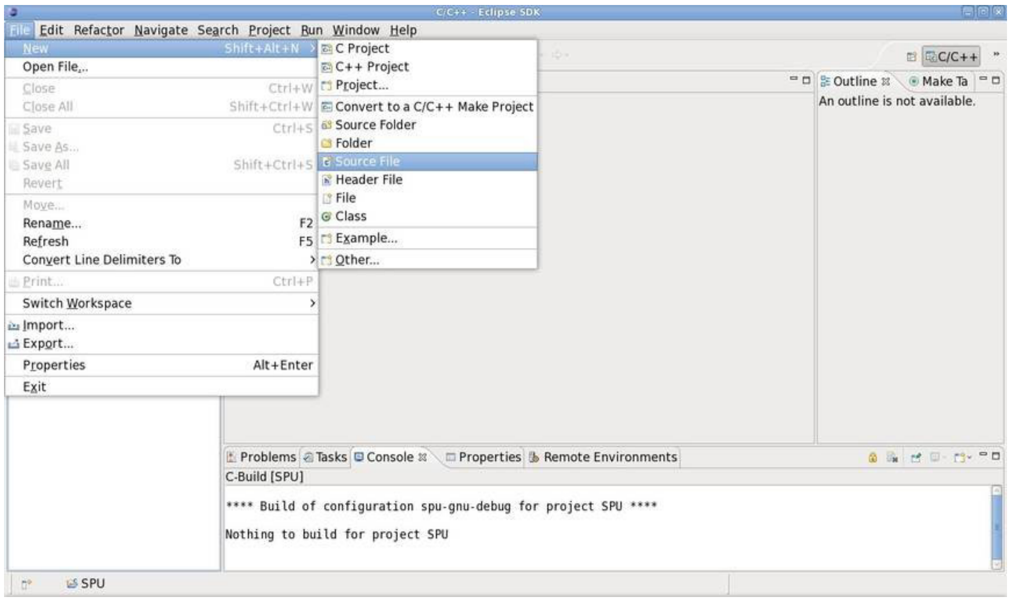
SPU project - add the directory path

1. Enter `/opt/ibm/systemsim-cell/include/callthru/spu`
2. Click **OK** twice to return to the **C/C++ Perspective**.



SPU project - create a new source file

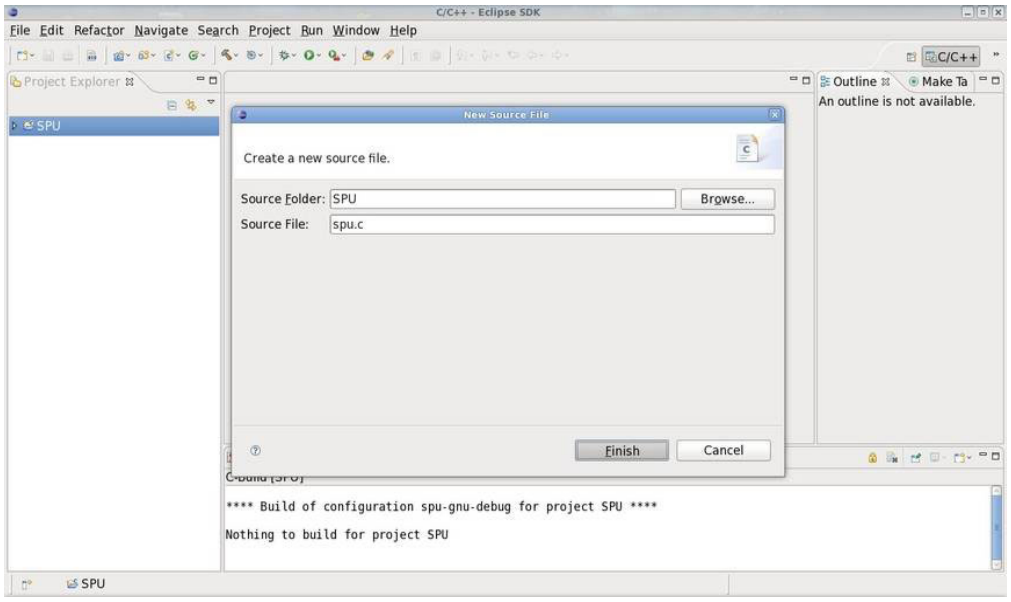
Create a C source file. Click **File → New → Source File**.



```
**** Build of configuration spu-gnu-debug for project SPU ****

Nothing to build for project SPU
```

SPU project - enter the new source file name

1. In **Source File**, enter `spu.c`.
2. Click **Finish**.



An outline is not available.

Create a new source file.

Source Folder: SPU

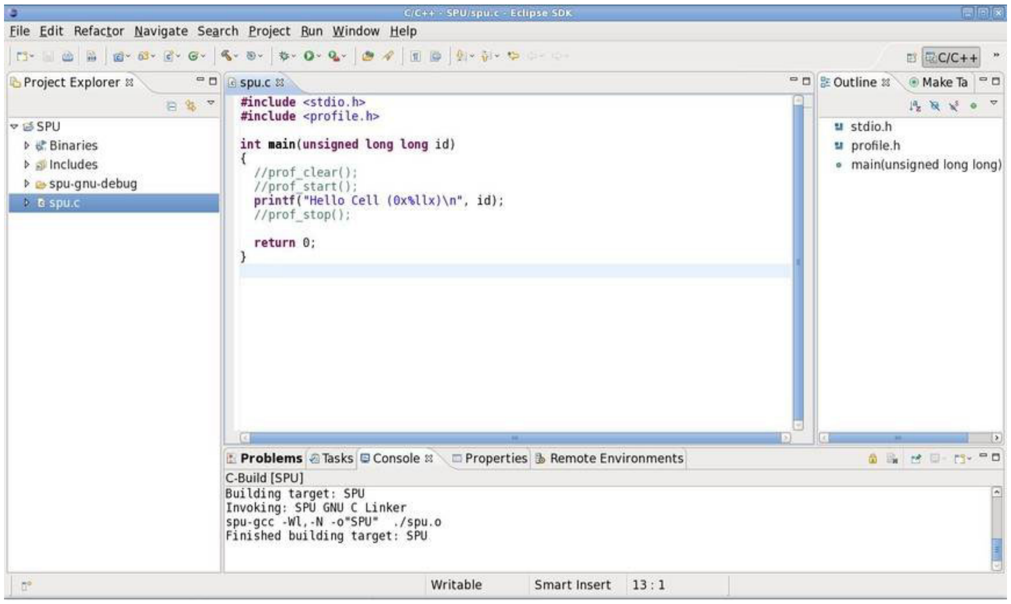
Source File: spu.c

```
**** Build of configuration spu-gnu-debug for project SPU ****

Nothing to build for project SPU
```

SPU project - edit the source file

A new editor is displayed so you can enter your source code. Copy and paste the following source code into your editor (you will uncomment the commented lines later):

```
#include <stdio.h>
#include <profile.h>
int main(unsigned long long id)
{
  //prof_clear();
  //prof_start();
  printf("Hello Cell (0x%llx)\n", id);
  //prof_stop();

  return 0;
}
```



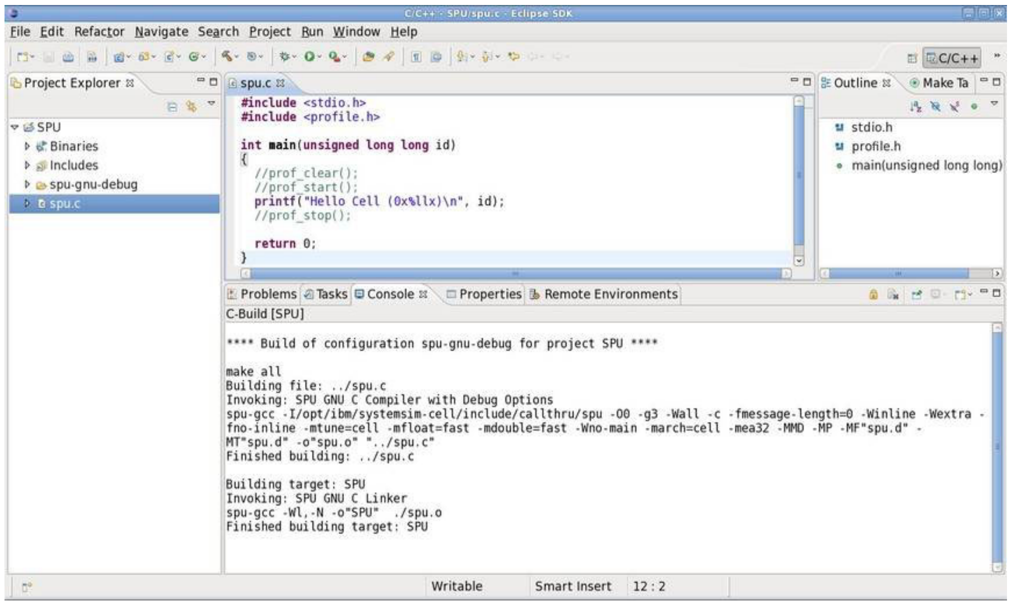
```
#include <stdio.h>
#include <profile.h>

int main(unsigned long long id)
{
  //prof_clear();
  //prof_start();
  printf("Hello Cell (0x%llx)\n", id);
  //prof_stop();

  return 0;
}
```

```
C-Build [SPU]
Building target: SPU
Invoking: SPU GNU C Linker
spu-gcc -Wl,-N -o"SPU"  ./spu.o
Finished building target: SPU
```

SPU project - automatically build the project

Save the source file (**Control + S**). The project is automatically built.

The build output is displayed in the **Console** view, and new resources, such as binaries and includes, are displayed in the **C/C++ Projects** view.



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

int main(unsigned long long id)
{
  //prof_clear();
  //prof_start();
  printf("Hello Cell (0x%llx)\n", id);
  //prof_stop();

  return 0;
}
```

```
**** Build of configuration spu-gnu-debug for project SPU ****

make all
Building file: ../spu.c
Invoking: SPU GNU C Compiler with Debug Options
spu-gcc -I/opt/ibm/systemsim-cell/include/callthru/spu -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -
fno-inline -mtune=cell -mfloat=fast -mdouble=fast -Wno-main -march=cell -mea32 -MMD -MP -MF"spu.d" -
MT"spu.d" -o"spu.o" "../spu.c"
Finished building: ../spu.c

Building target: SPU
Invoking: SPU GNU C Linker
spu-gcc -Wl,-N -o"SPU"  ./spu.o
Finished building target: SPU
```

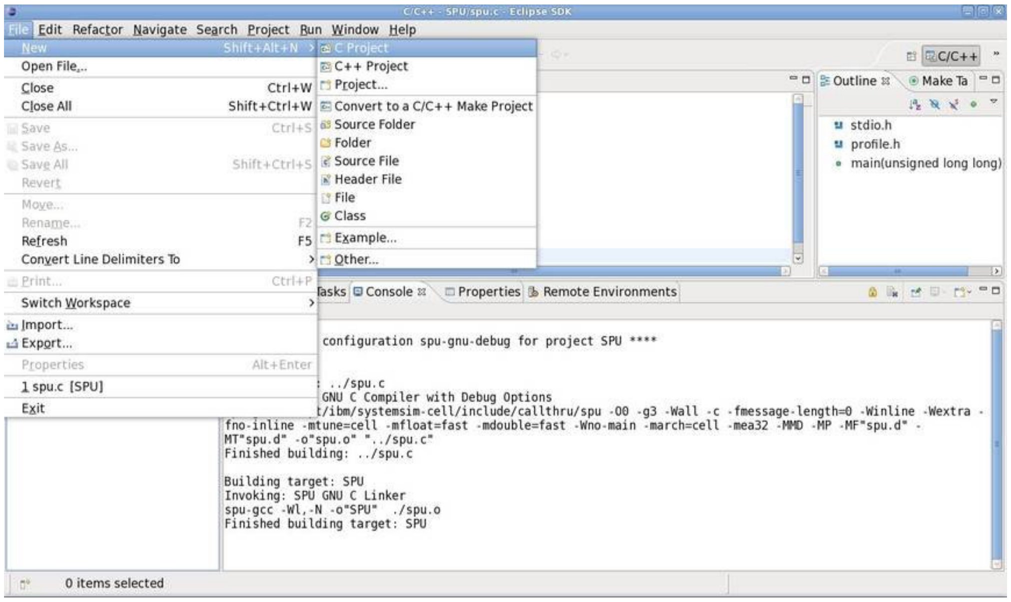
PPU project - create a PPU executable project

Objectives: After you have completed this part of the tutorial you will know how to:

- Create a PPU project
- Configure the PPU project
- Add the SPU executable as an embed SPU input

Now create a PPU executable project that uses the Embed SPU tool in your SPU project.

Click **File → New → C Project**.

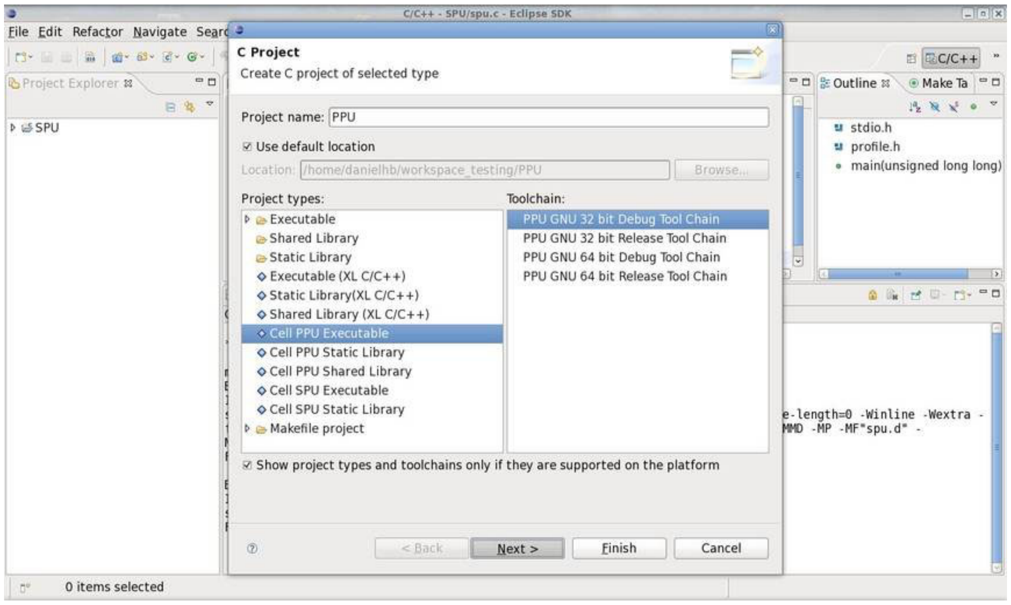


```
configuration spu-gnu-debug for project SPU ****

: ../spu.c
 GNU C Compiler with Debug Options
t/ibm/systemsim-cell/include/callthru/spu -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -
fno-inline -mtune=cell -mfloat=fast -mdouble=fast -Wno-main -march=cell -mea32 -MMD -MP -MF"spu.d" -
MT"spu.d" -o"spu.o" "../spu.c"
Finished building: ../spu.c

Building target: SPU
Invoking: SPU GNU C Linker
spu-gcc -Wl,-N -o"SPU"  ./spu.o
Finished building target: SPU
```

PPU project - create the PPU project

Create the PPU project:

1. For **Project name**, enter `PPU`.
2. For **Project Type**, select **Cell PPU Executable**
3. Click **Next**.



Project name: PPU

☑ Use default location

Location: /home/danielhb/workspace_testing/PPU Browse...

Project types:
- Executable
- Shared Library
- Static Library
- Executable (XL C/C++)
- Static Library(XL C/C++)
- Shared Library (XL C/C++)
- Cell PPU Executable
- Cell PPU Static Library
- Cell PPU Shared Library
- Cell SPU Executable
- Cell SPU Static Library
- Makefile project

Toolchain:
- PPU GNU 32 bit Debug Tool Chain
- PPU GNU 32 bit Release Tool Chain
- PPU GNU 64 bit Debug Tool Chain
- PPU GNU 64 bit Release Tool Chain

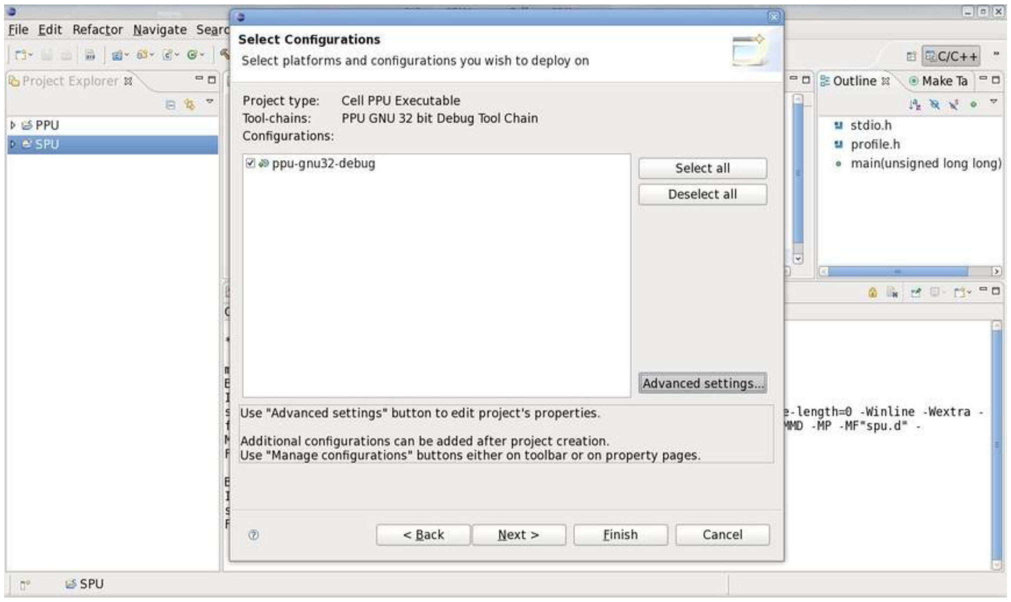
☑ Show project types and toolchains only if they are supported on the platform

```
e-length=0 -Winline -Wextra -
MMD -MP -MF"spu.d" -
```

PPU project - reference the SPU project - step 1

The project **SPU** that you created earlier is listed. Because you will embed the **SPU** project's executable into the **PPU** project, you need to reference it.

Click **Advanced settings**.



Select Configurations

Select platforms and configurations you wish to deploy on

Project type: Cell PPU Executable
Tool-chains: PPU GNU 32 bit Debug Tool Chain
Configurations:

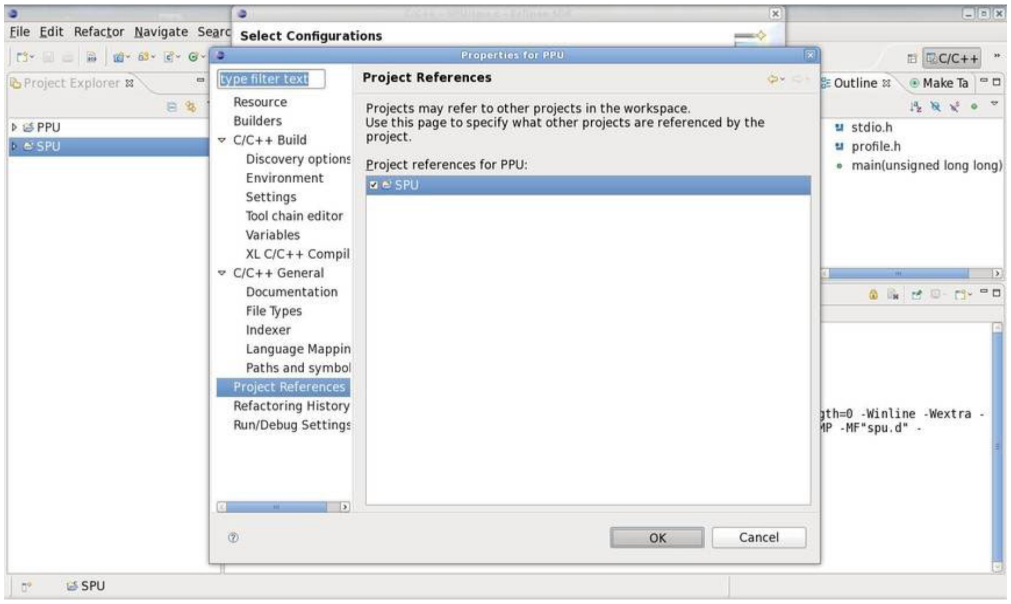
☑ ppu-gnu32-debug

Use "Advanced settings" button to edit project's properties.

Additional configurations can be added after project creation.
Use "Manage configurations" buttons either on toolbar or on property pages.

PPU project - reference the SPU project - step 2

1. Go to the **Project References** category.
2. Click **OK**.



Project References

Projects may refer to other projects in the workspace.
Use this page to specify what other projects are referenced by the project.

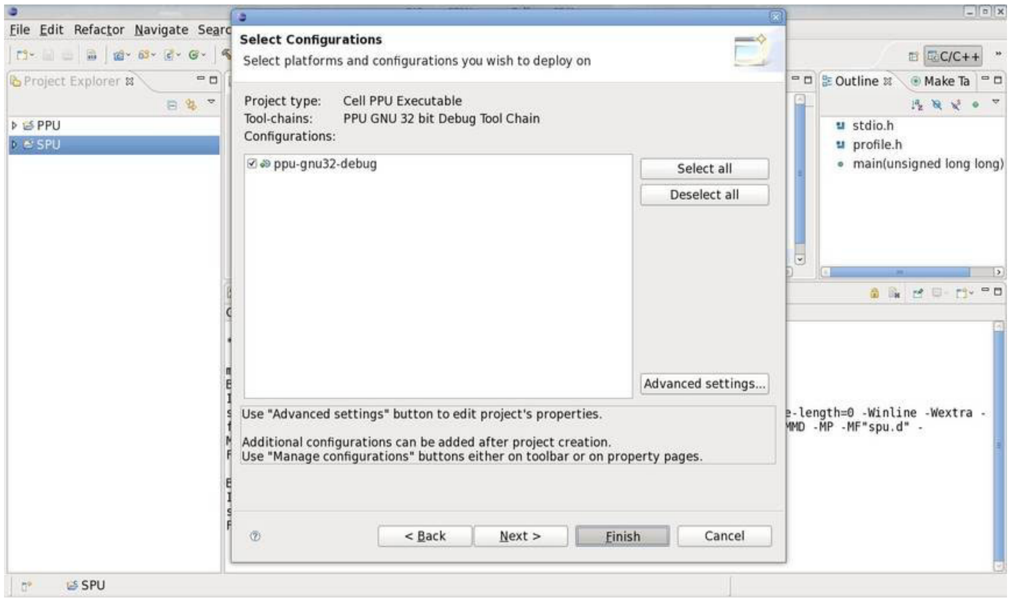
Project references for PPU:

☑ SPU

```
gth=0 -Winline -Wextra -
MP -MF"spu.d" -
```

PPU project - finish creating the PPU project

Click **Finish** to create the PPU project.



Select Configurations

Select platforms and configurations you wish to deploy on

Project type: Cell PPU Executable
Tool-chains: PPU GNU 32 bit Debug Tool Chain
Configurations:

☑ ppu-gnu32-debug

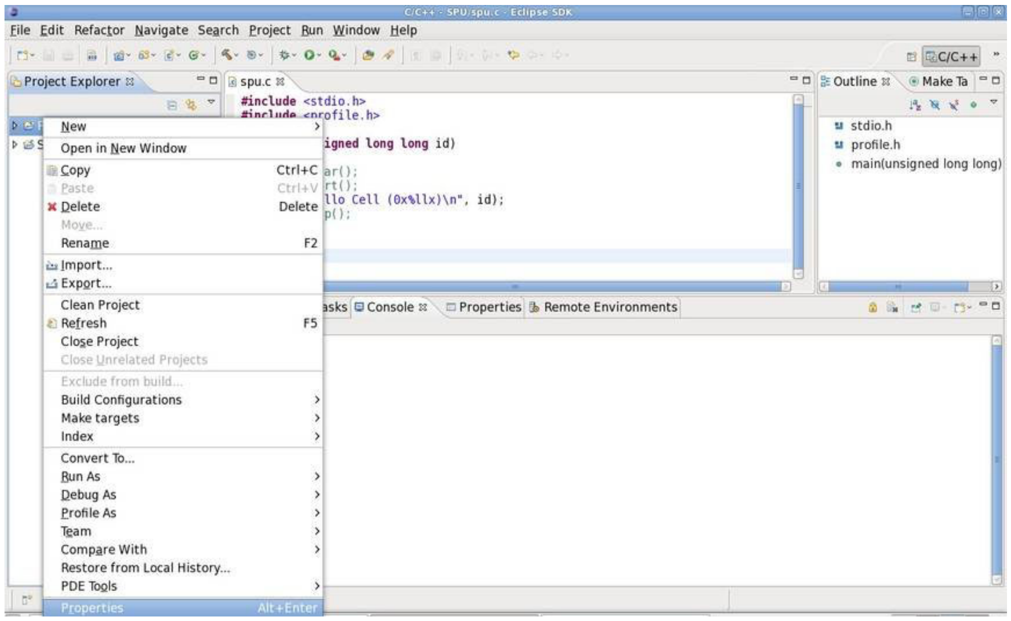
Use "Advanced settings" button to edit project's properties.

Additional configurations can be added after project creation.
Use "Manage configurations" buttons either on toolbar or on property pages.

PPU project - configure the PPU project

In the **C/C++ Projects** view:

1. Right click the **PPU** project.
2. Select **Properties**.

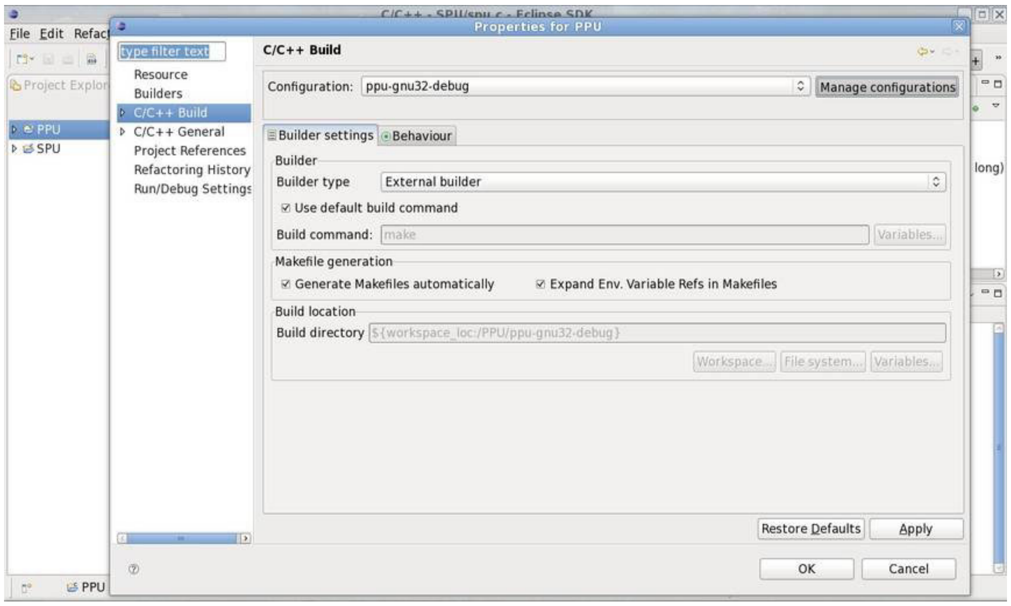


```
#include <stdio.h>
#include <profile.h>
igned long long id)
ar();
rt();
llo Cell (0x%llx)\n", id);
p();
```

PPU project - select the C/C++ build options

1. In the left pane, select **C/C++ Build**.

   In the **Configuration** group at the top, you can change the current configuration (for example. ppu-gnu32-debug, ppu-xl32-debug, and so on) or create your own. In the **Builder settings** tab, you can manipulate builder and makefile configurations.
2. Click **Manage configurations**.



C/C++ Build

Configuration: ppu-gnu32-debug

Builder
Builder type: External builder
☑ Use default build command
Build command: make

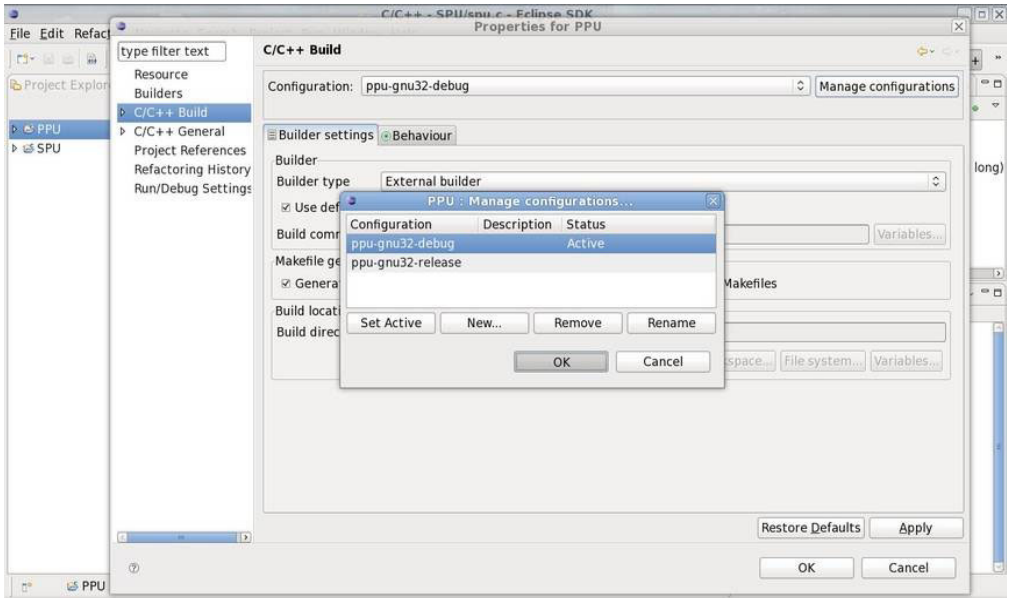
Makefile generation
☑ Generate Makefiles automatically
☑ Expand Env. Variable Refs in Makefiles

Build location
Build directory: ${workspace_loc:/PPU/ppu-gnu32-debug}

PPU project - view the Manage Configurations window

In this window you can rename, remove, or create new build configurations.

Click **OK** to continue.



Configuration: ppu-gnu32-debug

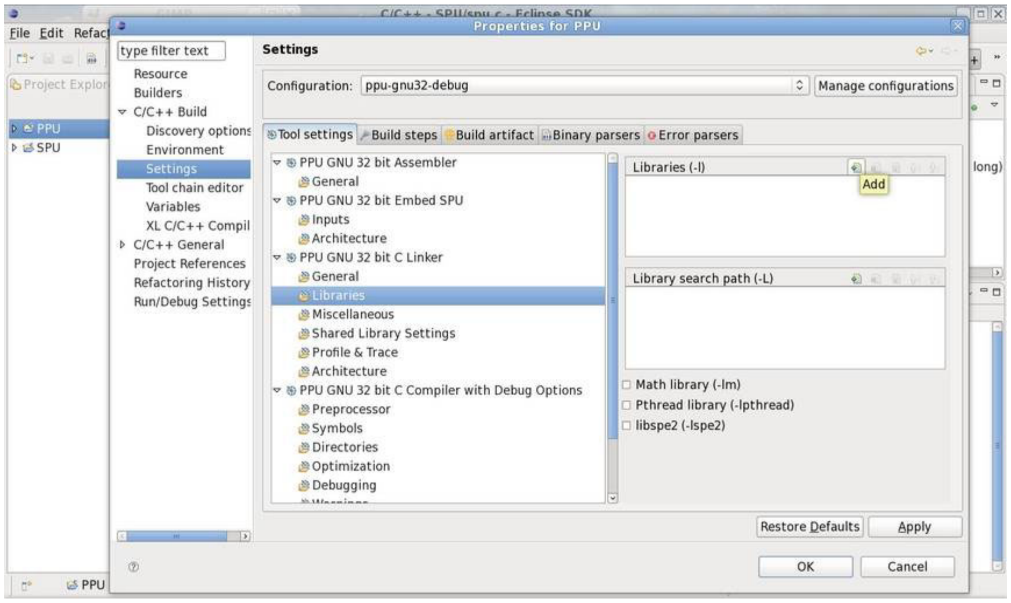
Builder type: External builder

Configuration	Description	Status
ppu-gnu32-debug		Active
ppu-gnu32-release		

PPU project - add the libspe2 library to the library linker list - step 1

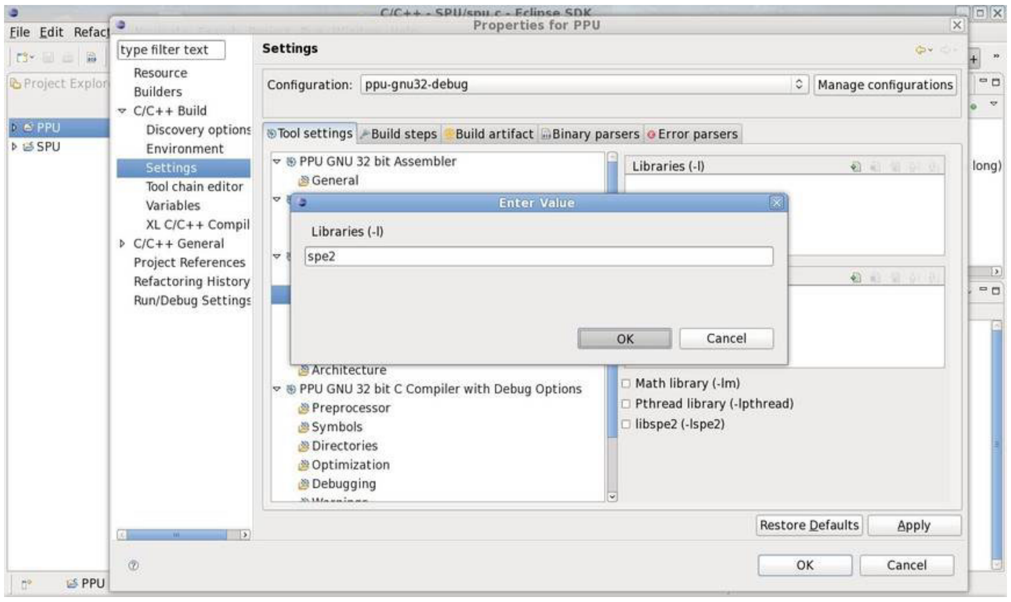
You use libspe2 in your PPU source code, so you need to add the library "spe2" to the linker's list of libraries.

1. In the **Tool Settings** tab, select the **Libraries** option, which is under the **PPU GNU 32 bit C Linker** category.
2. Click the **Add** button in the **Libraries** pane on the right.



PPU project - add the libspe2 library to the library linker list - step 2

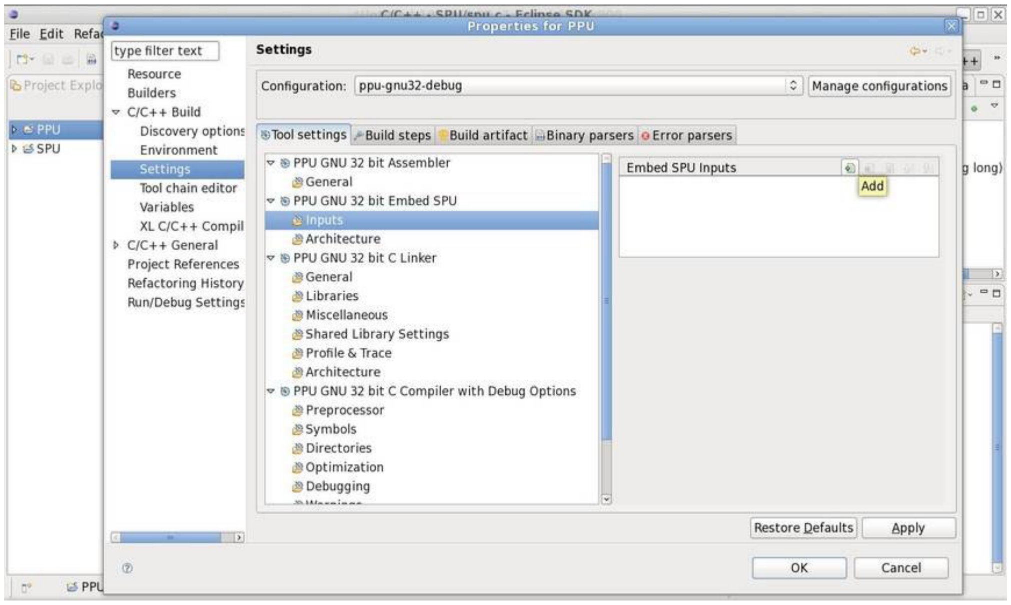
Enter `spe2`, then click **OK**.



PPU project - add the SPU executable as an embed SPU input - step 1

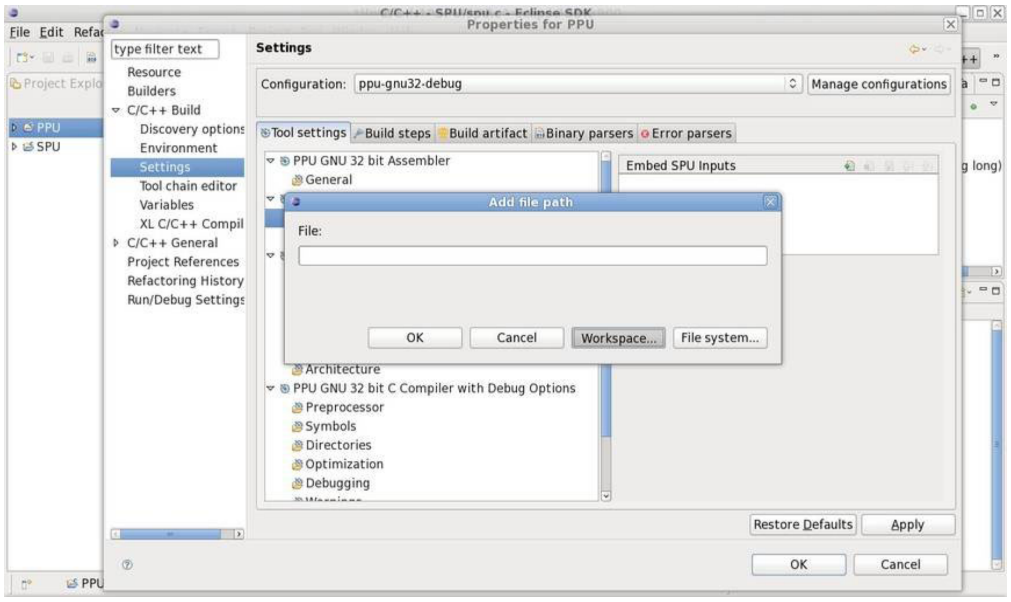
In order for the **PPU** project to embed the **SPU** project's executable, you need to add the **SPU** executable as an embed SPU input.

1. In **PPU GNU 32 bit Embed SPU**, select the **Inputs** option.
2. Click the **Add** button in the **Embed SPU Inputs** pane.



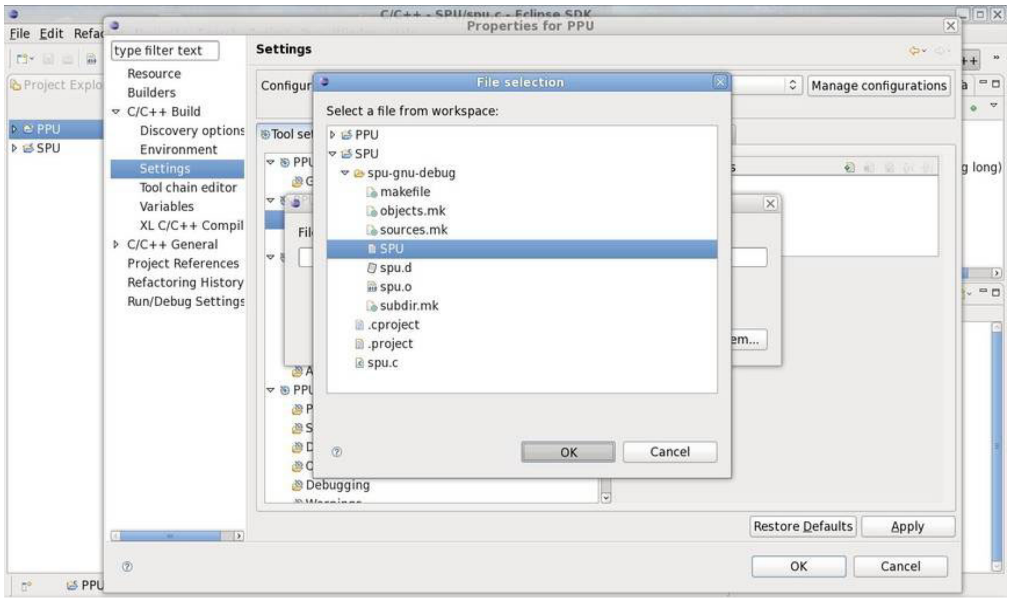
PPU project - add the SPU executable as an embed SPU input - step 2

Click the **Workspace** button.



PPU project - add the SPU executable as an embed SPU input - step 3

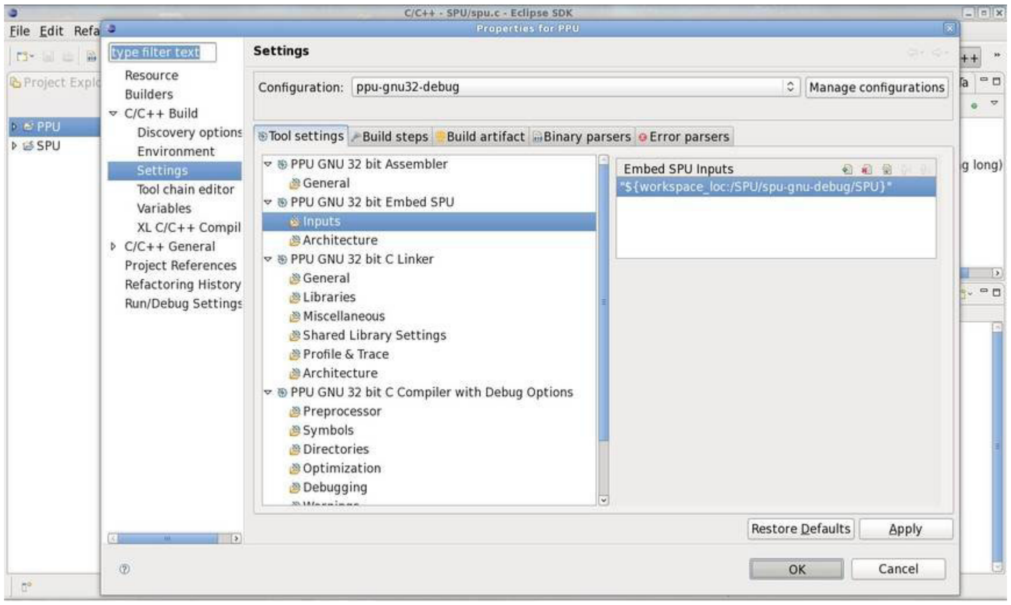
1. Select **SPU** → **spu-gnu-debug** → **SPU**.
2. Click **OK** twice to return to the **Properties for PPU** window.



PPU project - configure additional settings

You can configure additional settings via the other tabs (for example, **Build steps**, **Binary parsers**, and so on), and categories (such as **Project References**).

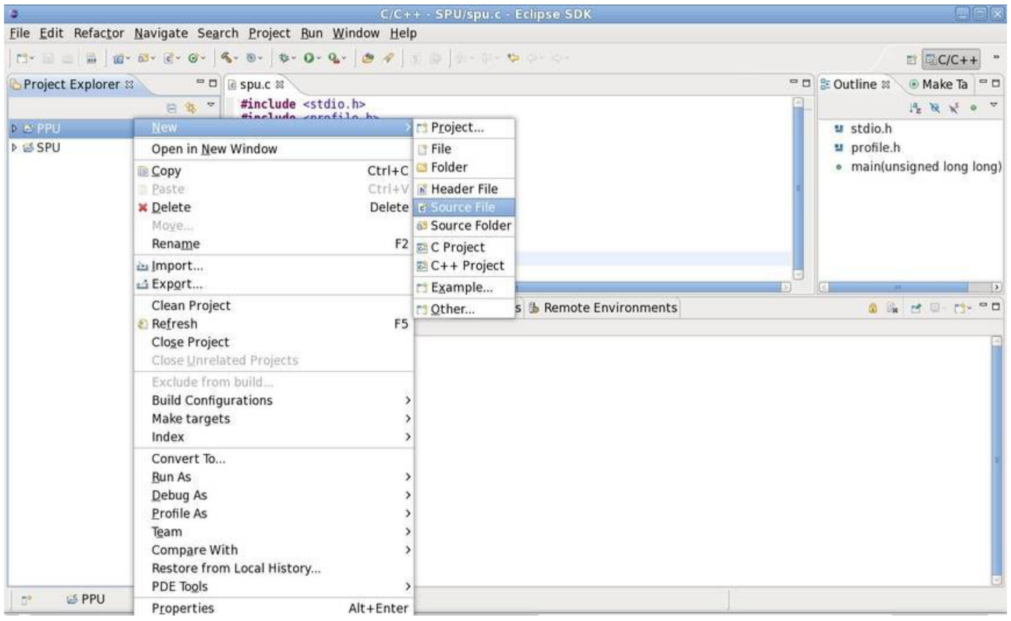
Take a minute to explore these other tabs and categories, then click **OK**.



"${workspace_loc:/SPU/spu-gnu-debug/SPU}"

PPU project - create another new source file

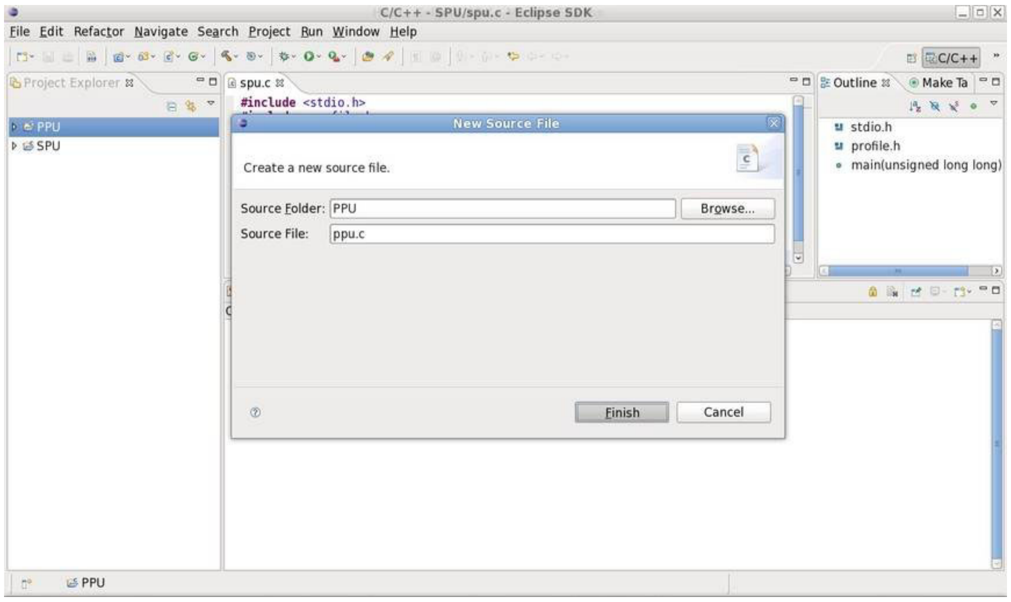
1. Right click the **PPU** project.
2. Select **New → Source File**.



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

PPU project - enter the name for the new source file

In the **Source File** field, enter `ppu.c`, then click **Finish**.



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

Create a new source file.

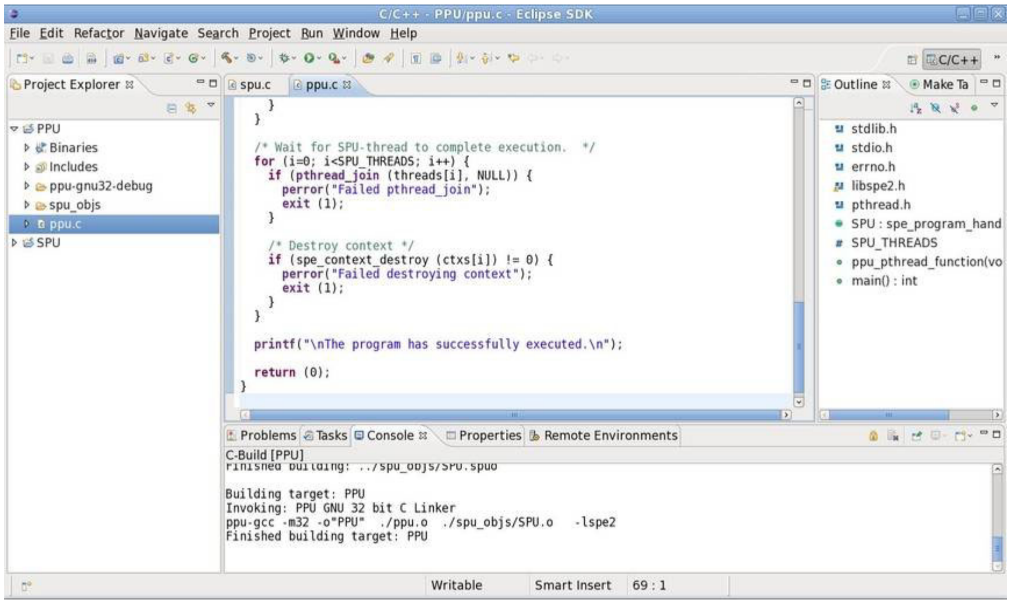
Source Folder: PPU

Source File: ppu.c

PPU project - edit the source code file

Copy and paste the following source code into your editor, then save it (**Ctrl+S**):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t SPU;

#define SPU_THREADS     8

void *ppu_pthread_function(void *arg) {
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  ctx = *((spe_context_ptr_t *)arg);
  if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0)
{
    perror ("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}

int main()
{
  int i;
  spe_context_ptr_t ctxs[SPU_THREADS];
  pthread_t threads[SPU_THREADS];

  /* Create several SPE-threads to execute 'SPU'.*/
  for(i=0; i<SPU_THREADS; i++)
    {
    /* Create context */
    if ((ctxs[i] = spe_context_create (0, NULL)) == NULL)
      {
      perror ("Failed creating context");
      exit (1);
      }
    /* Load program into context */
    if (spe_program_load (ctxs[i], &SPU))
      {
      perror ("Failed loading program");
      exit (1);
      }
    /* Create thread for each SPE context */
    if (pthread_create (&threads[i], NULL,
    &ppu_pthread_function, &ctxs[i]))
      {
      perror ("Failed creating thread");
      exit (1);
      }
  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<SPU_THREADS; i++)
    {
      if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }

    /* Destroy context */
    if (spe_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  printf("\nThe program has successfully executed.\n");

  return (0);
}
```



```
    }
  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<SPU_THREADS; i++) {
    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }

    /* Destroy context */
    if (spe_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  printf("\nThe program has successfully executed.\n");

  return (0);
}
```

```
Finished building: ../spu_objs/SPU.spuo

Building target: PPU
Invoking: PPU GNU 32 bit C Linker
ppu-gcc -m32 -o"PPU"  ./ppu.o  ./spu_objs/SPU.o   -lspe2
Finished building target: PPU
```

Cell/B.E. simulator - create the local Cell/B.E. simulator

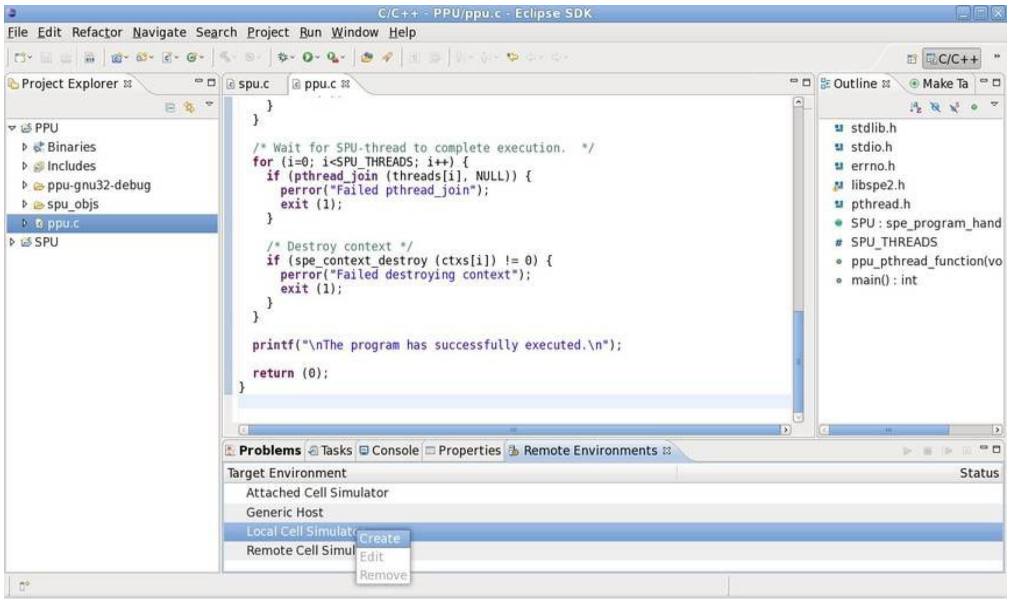
Objectives: After you have completed this part of the tutorial you will know how to:

- Create an environment
- Configure and start the simulator

Now that the projects are created and properly configured, you can test your program, but before you do this, you must create and start a Cell/B.E. environment. IDE integrates IBM's full-system simulator for the Cell/B.E. processor into Eclipse, so that you are only a few clicks away from testing your application on a Cell/B.E. environment, even if you do not have access to a physical Cell/B.E. processor.

In the **Remote Environments** view at the bottom:

1. Right click **Local Cell Simulator**
2. Select **Create**.

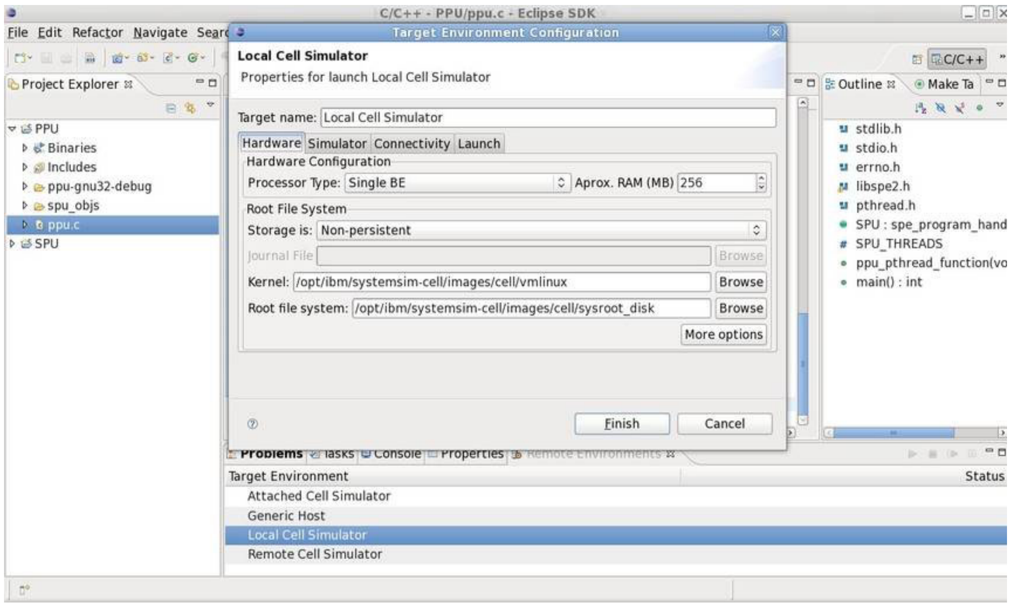


```
    }
  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<SPU_THREADS; i++) {
    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }

    /* Destroy context */
    if (spe_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  printf("\nThe program has successfully executed.\n");

  return (0);
}
```

Cell/B.E. simulator - configure the simulator

This is the **Local Cell Simulator** properties window, which you can use to configure the simulator to meet any specific needs that you might have. You can modify an existing Cell/B.E. environment's configuration at any time (as long as its not running) by right clicking the environment and selecting **Edit**.

1. Enter a name for this simulator (for example, `Local Cell Simulator`).
2. Click the **Simulator** tab.



Target Environment Configuration

Local Cell Simulator

Properties for launch Local Cell Simulator

Target name: Local Cell Simulator

Hardware | Simulator | Connectivity | Launch

Hardware Configuration

Processor Type: Single BE   Aprox. RAM (MB) 256

Root File System

Storage is: Non-persistent

Journal File

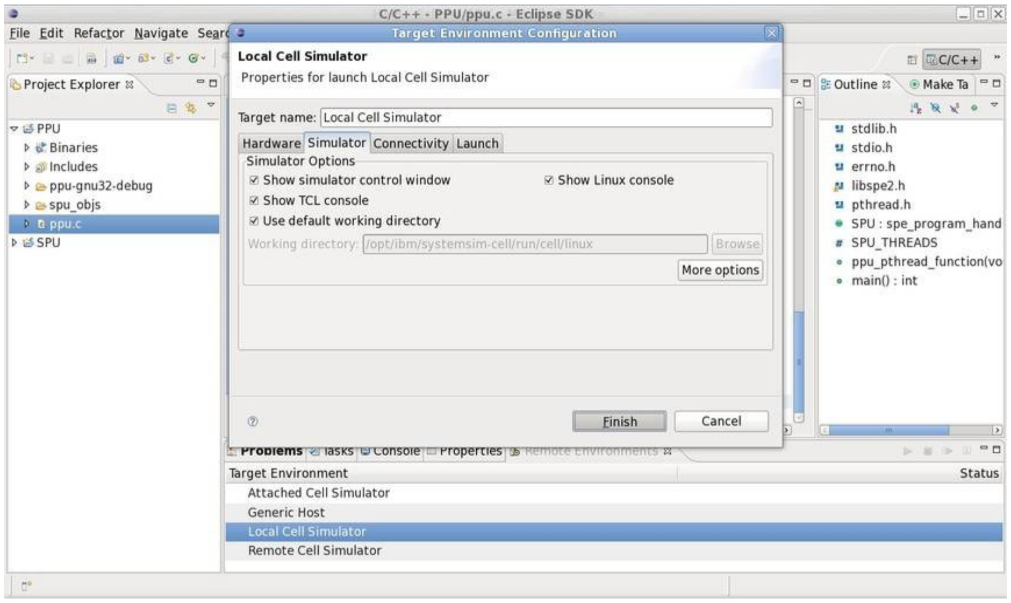
Kernel: /opt/ibm/systemsim-cell/images/cell/vmlinux

Root file system: /opt/ibm/systemsim-cell/images/cell/sysroot_disk

Target Environment	Status
Attached Cell Simulator	
Generic Host	
Local Cell Simulator	
Remote Cell Simulator	

Cell/B.E. simulator - configure the simulator options

In the **Simulator** tab, check the box labeled **Show TCL console**, then click **Finish**.



Target Environment Configuration

Local Cell Simulator

Properties for launch Local Cell Simulator

Target name: Local Cell Simulator

Hardware | Simulator | Connectivity | Launch

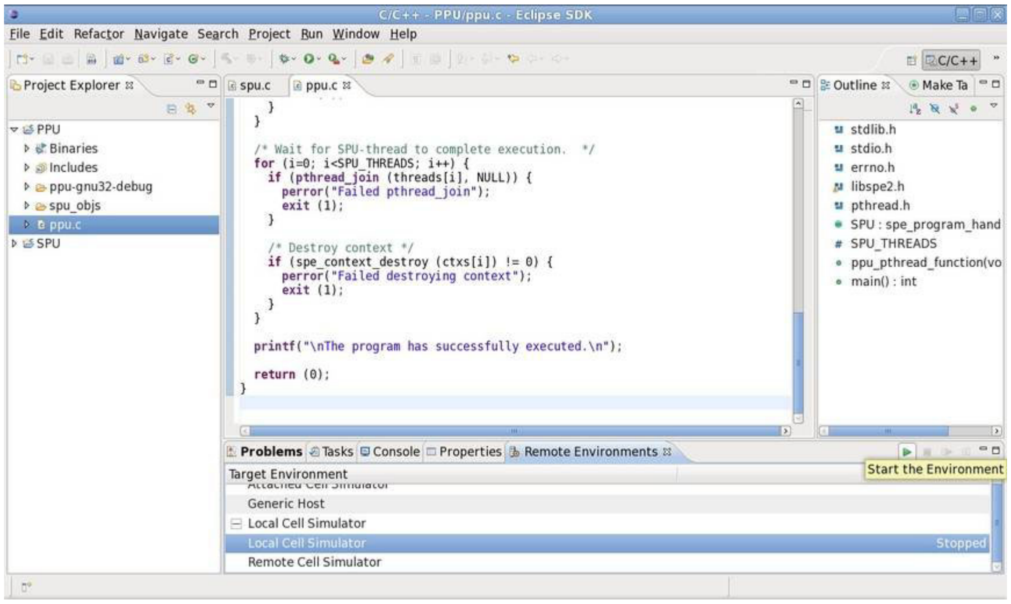
Simulator Options
- ☑ Show simulator control window
- ☑ Show Linux console
- ☑ Show TCL console
- ☑ Use default working directory

Working directory: /opt/ibm/systemsim-cell/run/cell/linux

Target Environment	Status
Attached Cell Simulator	
Generic Host	
Local Cell Simulator	
Remote Cell Simulator	

Cell/B.E. simulator - start the simulator - step 1

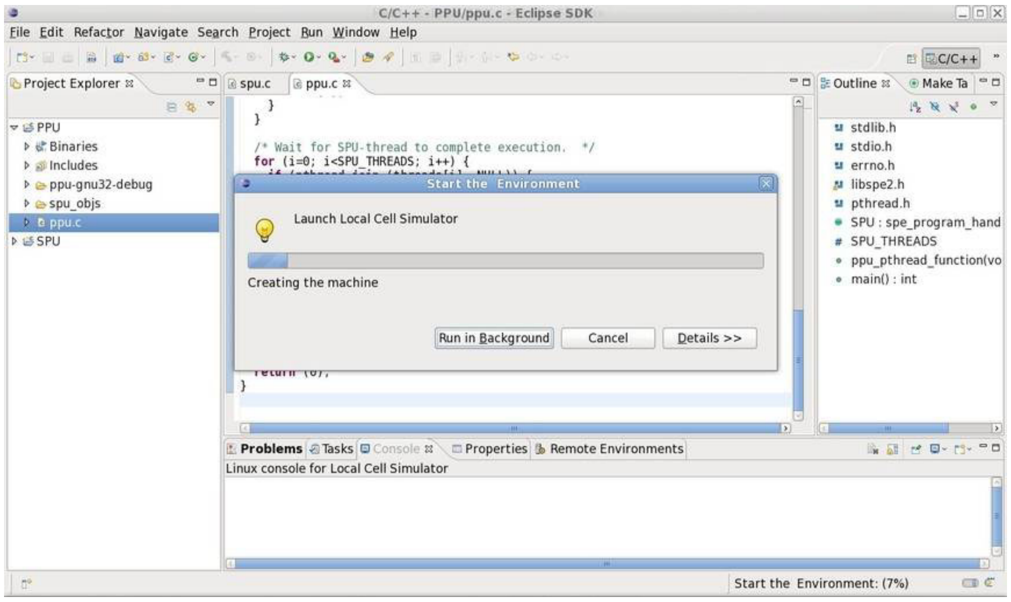
1. Click the + next to **Local Cell Simulator**.
2. Select the new simulator environment.
3. Click the **Start the Environment** (green arrow) button.



```
    }
  }

  /* Wait for SPU-thread to complete execution.  */
  for (i=0; i<SPU_THREADS; i++) {
    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }

    /* Destroy context */
    if (spe_context_destroy (ctxs[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }

  printf("\nThe program has successfully executed.\n");

  return (0);
}
```

Cell/B.E. simulator - start the simulator - step 2

The simulator now begins to launch (this can take a few minutes).



```
/* Wait for SPU-thread to complete execution.  */
for (i=0; i<SPU_THREADS; i++) {
```

Launch Local Cell Simulator

Creating the machine

Linux console for Local Cell Simulator

Start the Environment: (7%)

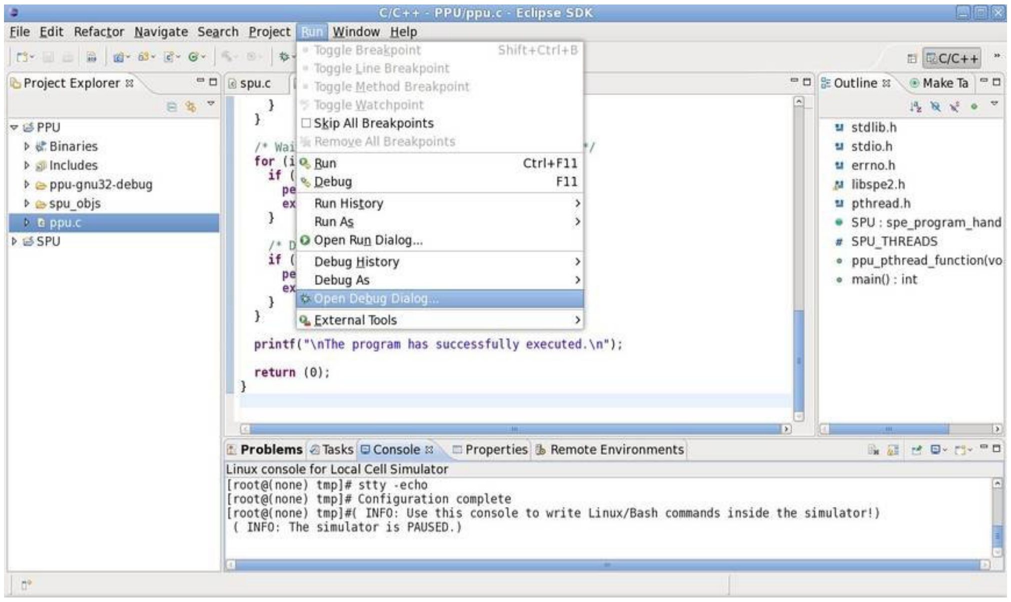
Create an application launch configuration

Objectives: After you have completed this part of the tutorial you will know how to:

- Create and configure a C/C++ Cell Application launch configuration

Next, you need to create and configure a **C/C++ Cell Application** launch configuration.

To do this, click **Run → Open Debug Dialog**.



```
    printf("\nThe program has successfully executed.\n");

    return (0);
}
```

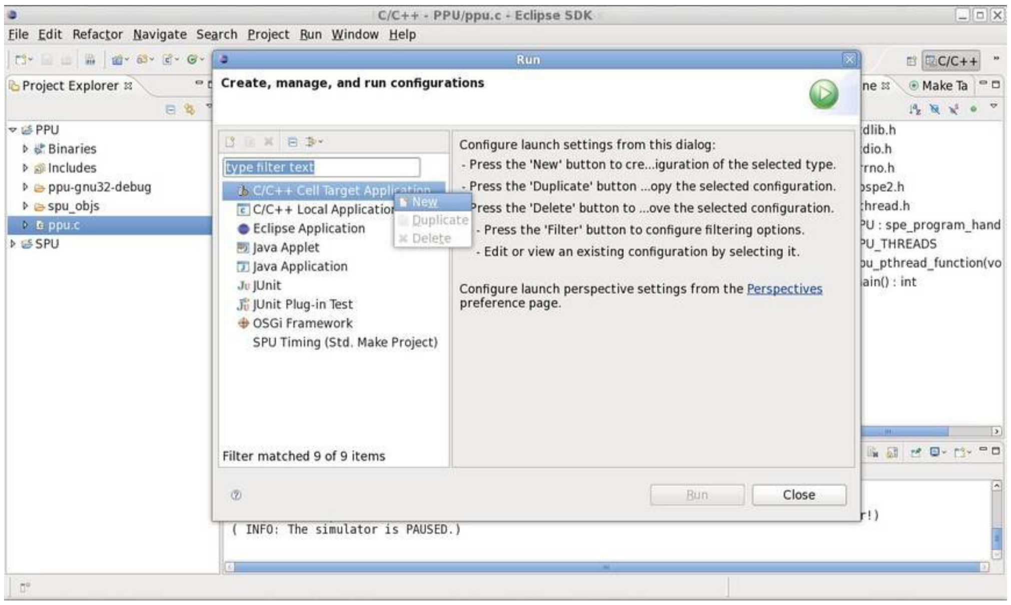
Linux console for Local Cell Simulator

```
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

Application launcher configuration - create a new C/C++ Cell/B.E. target application configuration

In the left pane:

1. Right click **C/C++ Cell Target Application**.
2. Select **New**.



Create, manage, and run configurations

Configure launch settings from this dialog:
- Press the 'New' button to cre...iguration of the selected type.
- Press the 'Duplicate' button ...opy the selected configuration.
- Press the 'Delete' button to ...ove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

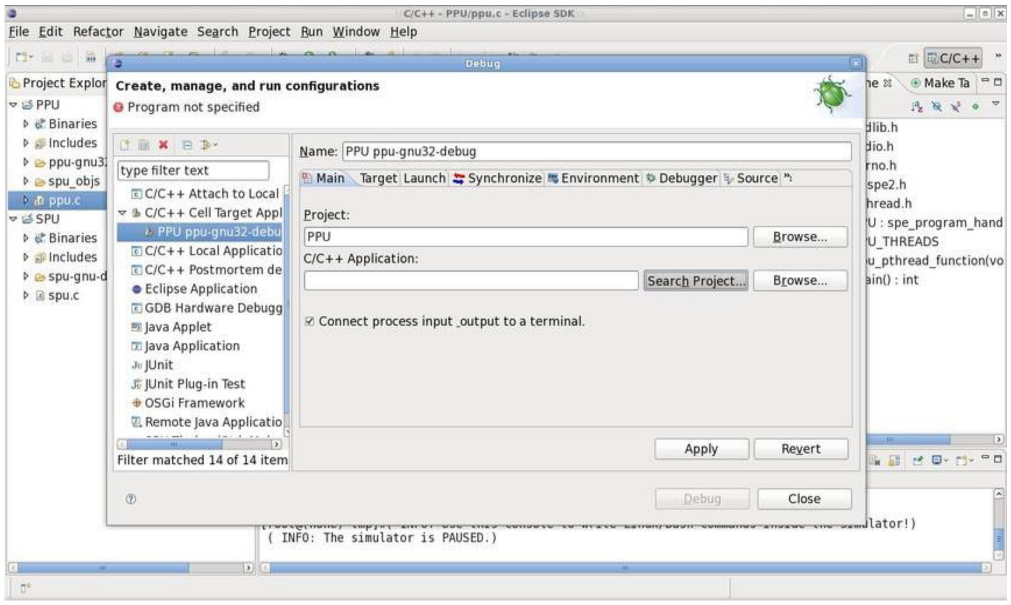
Filter matched 9 of 9 items

```
( INFO: The simulator is PAUSED.)
```

Application launcher configuration - modify the debug configuration

In the **Main** tab of the debug configuration, make sure that project **PPU** is in the **Project** field.

For the **C/C++ Application** field, click **Search Project**.



Name: PPU ppu-gnu32-debug

Project:
PPU

C/C++ Application:

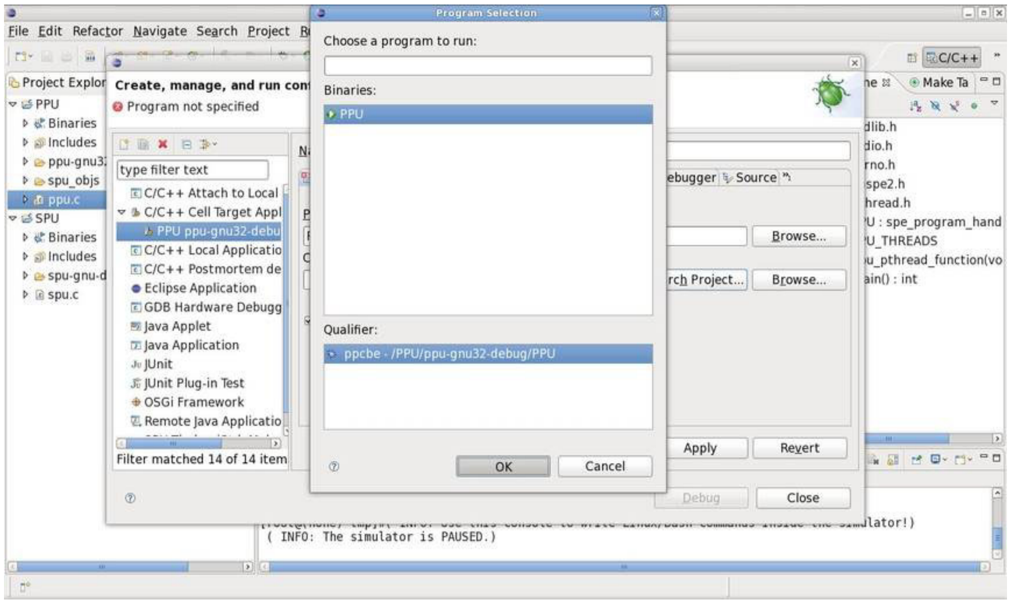
☑ Connect process input output to a terminal.

```
( INFO: The simulator is PAUSED.)
```

Application launcher configuration - select C/C++ application

The **Qualifier** section lists the different build configurations that can be used (32 or 64 bit gnu, xlc, and so on). The **Binaries** section lists the available binaries for the corresponding **Qualifier**. For the **PPU** project, you have not changed the default build configuration, so only one **Qualifier** is currently available.

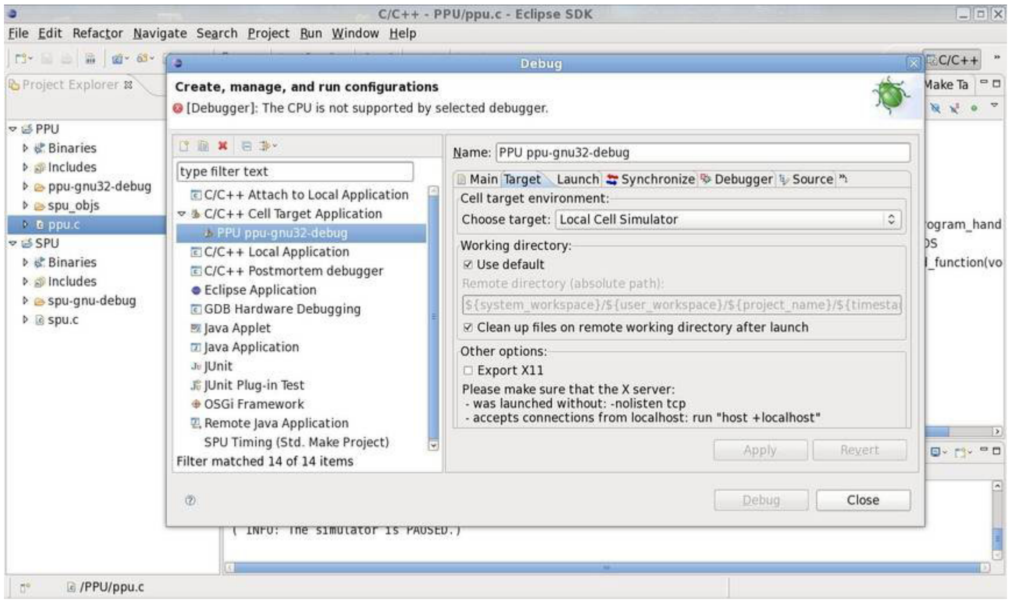
Select the **PPU** binary and click **OK**.



```
( INFO: The simulator is PAUSED.)
```

Application launcher configuration - select target environment

1. Navigate to the **Target** tab.
2. For the **Choose target** field, select the simulator that was just created, then go to the **Launch** tab.



Create, manage, and run configurations

[Debugger]: The CPU is not supported by selected debugger.

Name: PPU ppu-gnu32-debug

Cell target environment:
Choose target: Local Cell Simulator

Working directory:
☑ Use default
Remote directory (absolute path):
${system_workspace}/${user_workspace}/${project_name}/${timesta
☑ Clean up files on remote working directory after launch

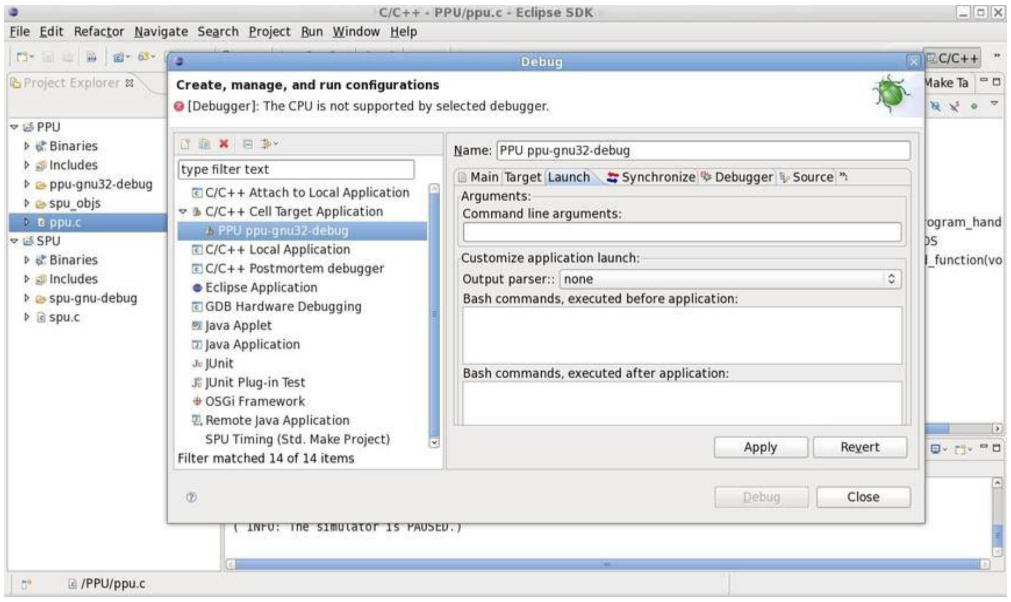
Other options:
☐ Export X11
Please make sure that the X server:
- was launched without: -nolisten tcp
- accepts connections from localhost: run "host +localhost"

```
( INFO: The simulator is PAUSED.)
```

Application launcher configuration - configure the Launch tab

Here you can specify command line arguments as well as bash commands that are executed before and/or after the Cell/B.E. application runs.

Next, go to the **Synchronize** tab.



Create, manage, and run configurations

[Debugger]: The CPU is not supported by selected debugger.

Filter matched 14 of 14 items

Name: PPU ppu-gnu32-debug

Arguments:

Command line arguments:

Customize application launch:

Output parser:: none

Bash commands, executed before application:

Bash commands, executed after application:

```
( INFO: The simulator is PAUSED.)
```

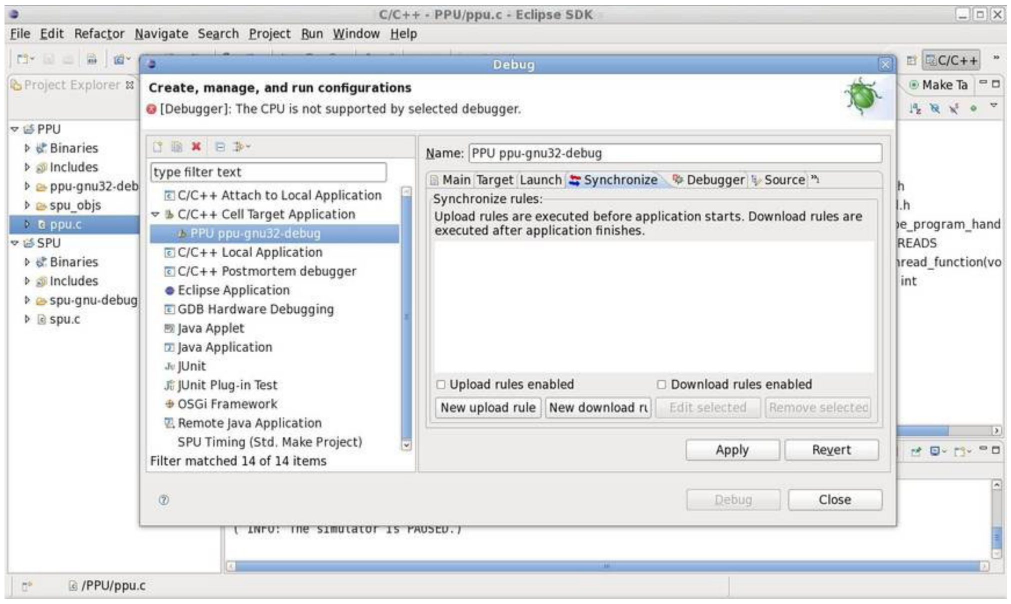
Application launcher configuration - specify resources to synchronize

In this tab, you can specify resources (such as input/output files) that need to be synchronized with the Cell/B.E. environment's file system before, or after the application runs or both.

Use **New upload rule** to specify resources to copy to the Cell/B.E. environment before the application runs, and use **New download rule** to copy resource(s) back to your local file system after the application has run.

Do not forget to check the **Upload rules enabled** and/or **Download rules enabled** boxes after adding any upload/download rules.

Now go to the **Debugger** tab.


Create, manage, and run configurations

[Debugger]: The CPU is not supported by selected debugger.

Name: PPU ppu-gnu32-debug

Synchronize rules:

Upload rules are executed before application starts. Download rules are executed after application finishes.

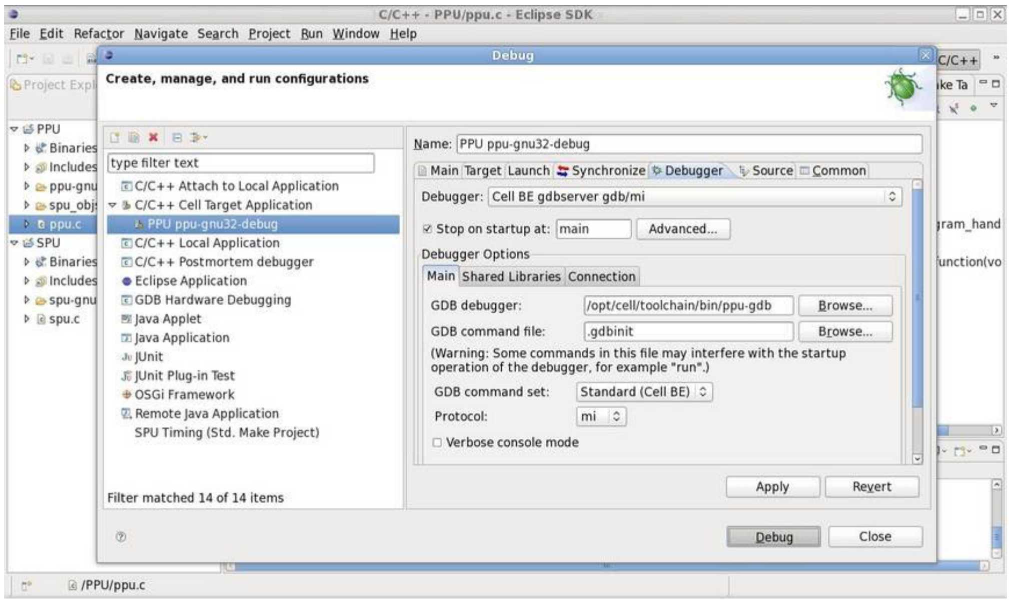
☐ Upload rules enabled ☐ Download rules enabled

Application launcher configuration - select debugger and launch the debug configuration

In the **Debugger** field, you can choose from **Cell PPU gdbserver**, **Cell SPU gdbserver**, or **Cell BE gdbserver**.

To debug only PPU or SPU programs, select **Cell PPU gdbserver** or **Cell SPU gdbserver**, respectively. The **Cell BE gdbserver** option is the combined debugger, which allows for debugging of PPU and SPU source code in one debug session.

1. Set the **Debugger** field to **Cell BE gdbserver**.
2. Click **Debug** to launch the debug session.


C/C++ - PPU/ppu.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
Debug
Create, manage, and run configurations
type filter text
C/C++ Attach to Local Application
C/C++ Cell Target Application
PPU ppu-gnu32-debug
C/C++ Local Application
C/C++ Postmortem debugger
Eclipse Application
GDB Hardware Debugging
Java Applet
Java Application
JUnit
JUnit Plug-in Test
OSGi Framework
Remote Java Application
SPU Timing (Std. Make Project)
Filter matched 14 of 14 items
Main Target Launch Synchronize Debugger Source Common
Debugger Options
Main Shared Libraries Connection
Advanced...
Browse...
Browse...
Apply
Revert
Debug
Close
/PPU/ppu.c

Name: PPU ppu-gnu32-debug

Debugger: Cell BE gdbserver gdb/mi

☑ Stop on startup at: main

GDB debugger: /opt/cell/toolchain/bin/ppu-gdb

GDB command file: .gdbinit

(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

GDB command set: Standard (Cell BE)

Protocol: mi

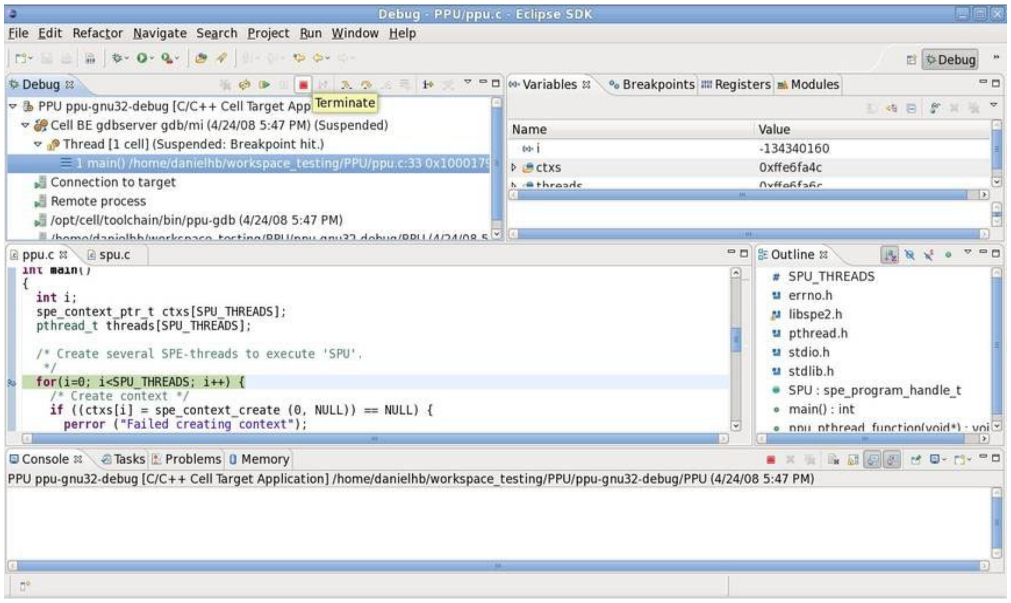
☐ Verbose console mode

Debug the application

Objectives: After you have completed this part of the tutorial you will know how to:

- Use the debugger functions to step through your source code

The debug perspective functions in a similar way to any other GUI debugger you have may have used in the past. You can use it to step through your Cell/B.E. application's source code, while you monitor variables and register values, so you can find and fix bugs faster.



```
PPU ppu-gnu32-debug [C/C++ Cell Target App
  Cell BE gdbserver gdb/mi (4/24/08 5:47 PM) (Suspended)
    Thread [1 cell] (Suspended: Breakpoint hit.)
      1 main() /home/danielhb/workspace_testing/PPU/ppu.c:33 0x100017
  Connection to target
  Remote process
  /opt/cell/toolchain/bin/ppu-gdb (4/24/08 5:47 PM)
```

Name	Value
i	-134340160
ctxs	0xffe6fa4c

```
{
  int i;
  spe_context_ptr_t ctxs[SPU_THREADS];
  pthread_t threads[SPU_THREADS];

  /* Create several SPE-threads to execute 'SPU'.
   */
  for(i=0; i<SPU_THREADS; i++) {
    /* Create context */
    if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
```

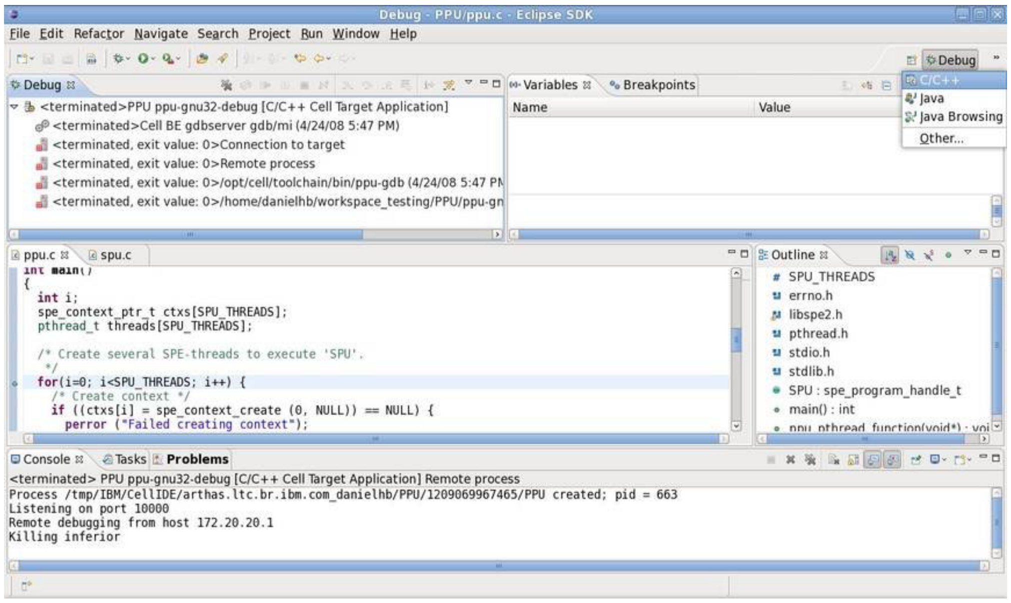
```
SPU_THREADS
errno.h
libspe2.h
pthread.h
stdio.h
stdlib.h
SPU : spe_program_handle_t
main() : int
```

```
PPU ppu-gnu32-debug [C/C++ Cell Target Application] /home/danielhb/workspace_testing/PPU/ppu-gnu32-debug/PPU (4/24/08 5:47 PM)
```

Debug the application - switch to the C/C++ perspective

Switch to the C/C++ perspective. To do this:

1. Click the double right arrows at the top right
2. Select **C/C++**.



```
<terminated>PPU ppu-gnu32-debug [C/C++ Cell Target Application]
  <terminated>Cell BE gdbserver gdb/mi (4/24/08 5:47 PM)
  <terminated, exit value: 0>Connection to target
  <terminated, exit value: 0>Remote process
  <terminated, exit value: 0>/opt/cell/toolchain/bin/ppu-gdb (4/24/08 5:47 PM
  <terminated, exit value: 0>/home/danielhb/workspace_testing/PPU/ppu-gn
```

```
int main()
{
  int i;
  spe_context_ptr_t ctxs[SPU_THREADS];
  pthread_t threads[SPU_THREADS];

  /* Create several SPE-threads to execute 'SPU'.
   */
  for(i=0; i<SPU_THREADS; i++) {
    /* Create context */
    if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
```

```
<terminated> PPU ppu-gnu32-debug [C/C++ Cell Target Application] Remote process
Process /tmp/IBM/CellIDE/arthas.ltc.br.ibm.com_danielhb/PPU/1209069967465/PPU created; pid = 663
Listening on port 10000
Remote debugging from host 172.20.20.1
Killing inferior
```

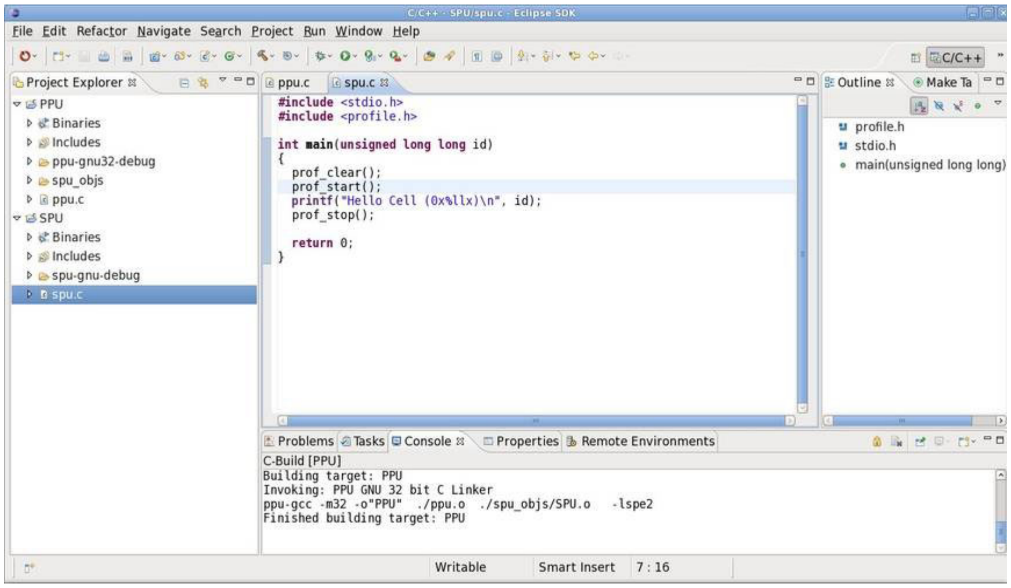
Dynamic profiling tool

Objective: After you have completed this part of the tutorial you will know how to:

- Include the profiling functions in the spu.c code
- Run and view the results of dynamic performance analysis

Next, you see how to use the dynamic profiling tool on the SPU code.

1. Open the editor for **spu.c**.
2. Uncomment the 3 lines **prof_clear();**, **prof_start();**, and **prof_stop();**.
3. Rebuild the project by saving the source file (**Ctrl+S**).



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

int main(unsigned long long id)
{
  prof_clear();
  prof_start();
  printf("Hello Cell (0x%llx)\n", id);
  prof_stop();

  return 0;
}
```

C-Build [PPU]

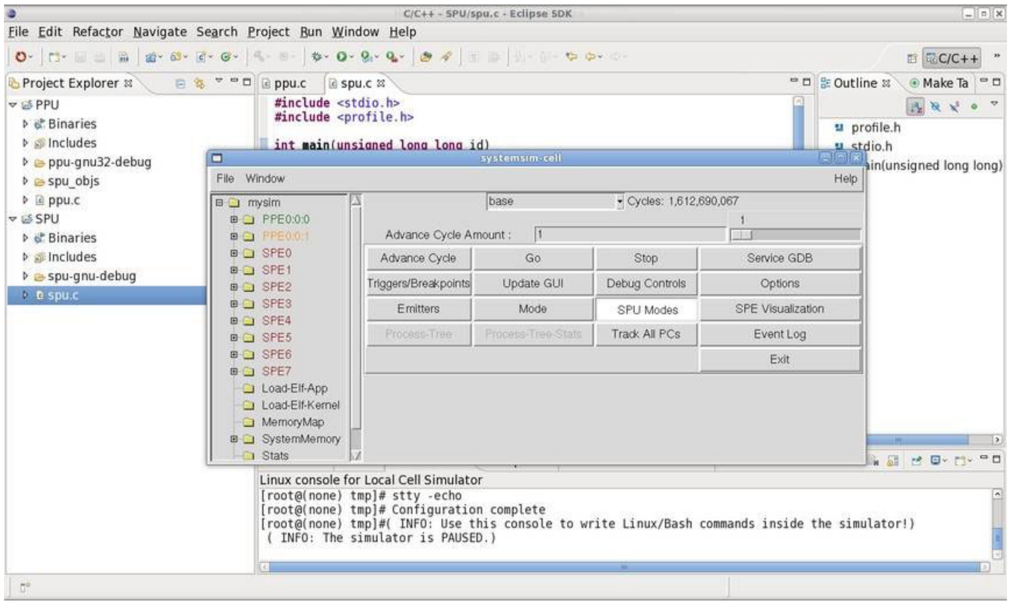
```
Building target: PPU
Invoking: PPU GNU 32 bit C Linker
ppu-gcc -m32 -o"PPU"  ./ppu.o  ./spu_objs/SPU.o   -lspe2
Finished building target: PPU
```

Dynamic profiling tool - change SPU modes in the simulator window

In order to use the dynamic profiling tool, the SPE's must be in **pipeline** mode.

Currently the SPE's are running in **fast** mode, so you need to change this.

1. Open the Simulator GUI window (**systemsim-cell**),
2. Click the button labeled **SPU Modes**.



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

int main(unsigned long long id)
```

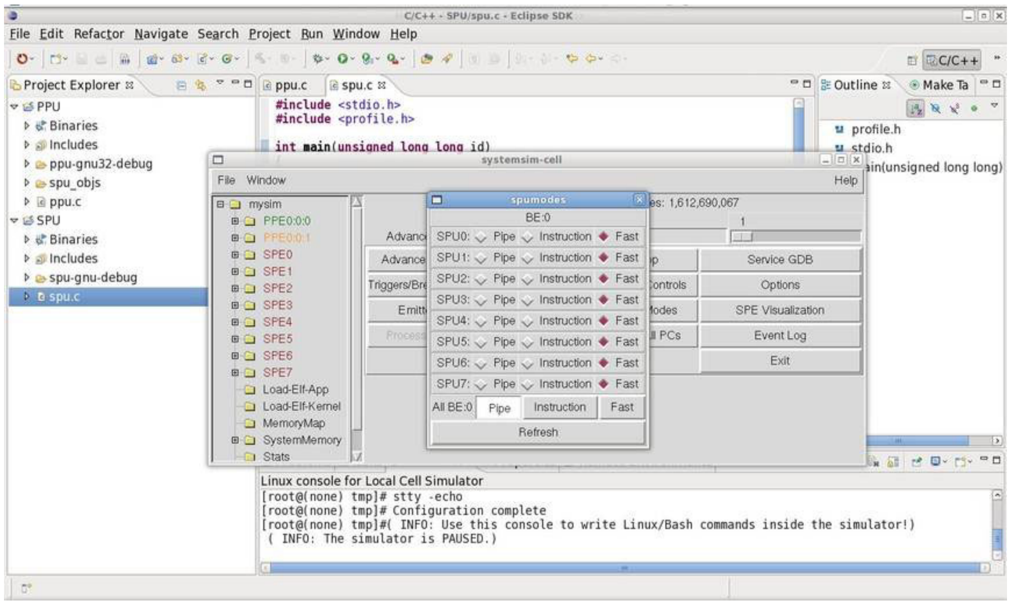
base   Cycles: 1,612,690,067

Advance Cycle Amount : 1

```
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

Dynamic profiling tool - change the SPEs to pipeline mode

To change all of the SPE's to **pipeline** mode, click the **Pipe** button at the bottom, then go back to Eclipse.



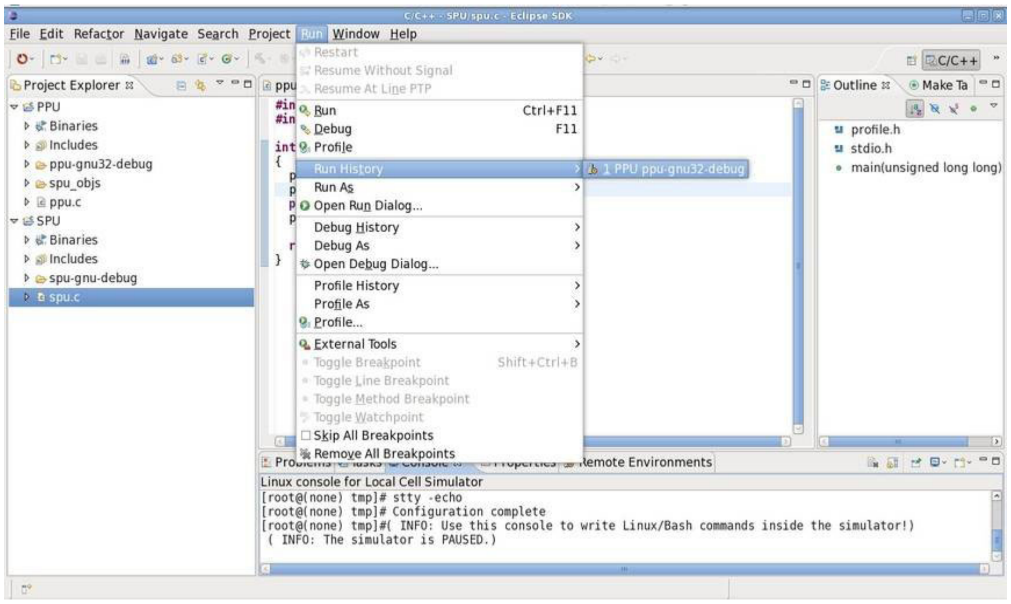
```
#include <stdio.h>
#include <profile.h>

int main(unsigned long long id)
```

```
Linux console for Local Cell Simulator
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

Dynamic profiling tool - launch the PPU application

Now that you have included the profiling functions in the **spu.c** code, and have the SPE's set to **Pipeline** mode, you can launch the application.

To do this, click **Run → Run History → PPU**.



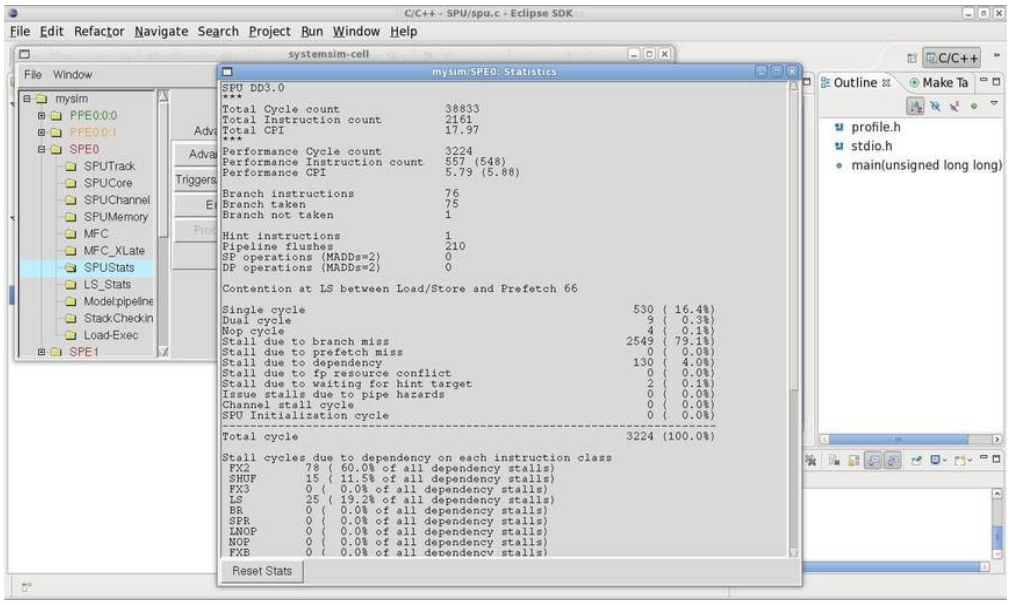
```
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

Dynamic profiling tool - view the results of the dynamic performance analysis

You can view the results of the dynamic performance analysis.

To do this:

1. Open the Simulator GUI window.
2. Expand the category **SPE0**
3. Open **SPUStats**.



```
SPU DD3.0
***
Total Cycle count              38833
Total Instruction count        2161
Total CPI                      17.97
***
Performance Cycle count        3224
Performance Instruction count  557 (548)
Performance CPI                5.79 (5.88)

Branch instructions            76
Branch taken                   75
Branch not taken               1

Hint instructions              1
Pipeline flushes               210
SP operations (MADDs=2)        0
DP operations (MADDs=2)        0

Contention at LS between Load/Store and Prefetch 66

Single cycle                                                   530 ( 16.4%)
Dual cycle                                                       9 (  0.3%)
Nop cycle                                                        4 (  0.1%)
Stall due to branch miss                                      2549 ( 79.1%)
Stall due to prefetch miss                                       0 (  0.0%)
Stall due to dependency                                        130 (  4.0%)
Stall due to fp resource conflict                                0 (  0.0%)
Stall due to waiting for hint target                             2 (  0.1%)
Issue stalls due to pipe hazards                                 0 (  0.0%)
Channel stall cycle                                              0 (  0.0%)
SPU Initialization cycle                                         0 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                                   3224 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2        78 ( 60.0% of all dependency stalls)
 SHUF       15 ( 11.5% of all dependency stalls)
 FX3        0 (  0.0% of all dependency stalls)
 LS         25 ( 19.2% of all dependency stalls)
 BR         0 (  0.0% of all dependency stalls)
 SPR        0 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
```

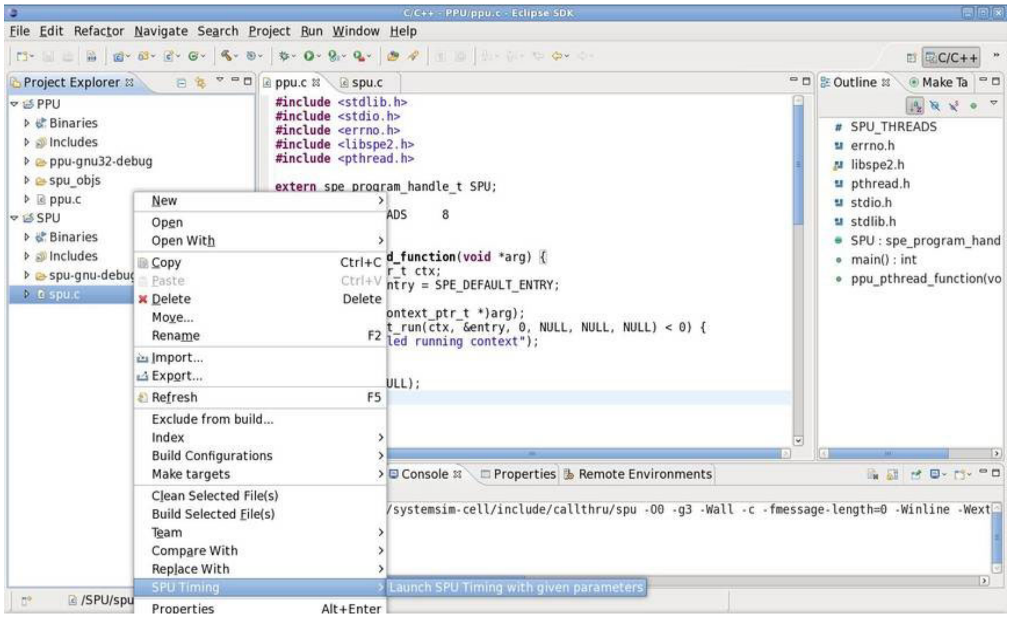
SPU timing tool - static timing analysis

Objectives: After you have completed this part of the tutorial you will know how to:

- Launch the SPU Timing tool
- View data collected by the SPU Timing tool

To launch the **SPU Timing** tool:

1. Right click **spu.c**
2. Select **SPU Timing → Launch SPU Timing with given parameters**



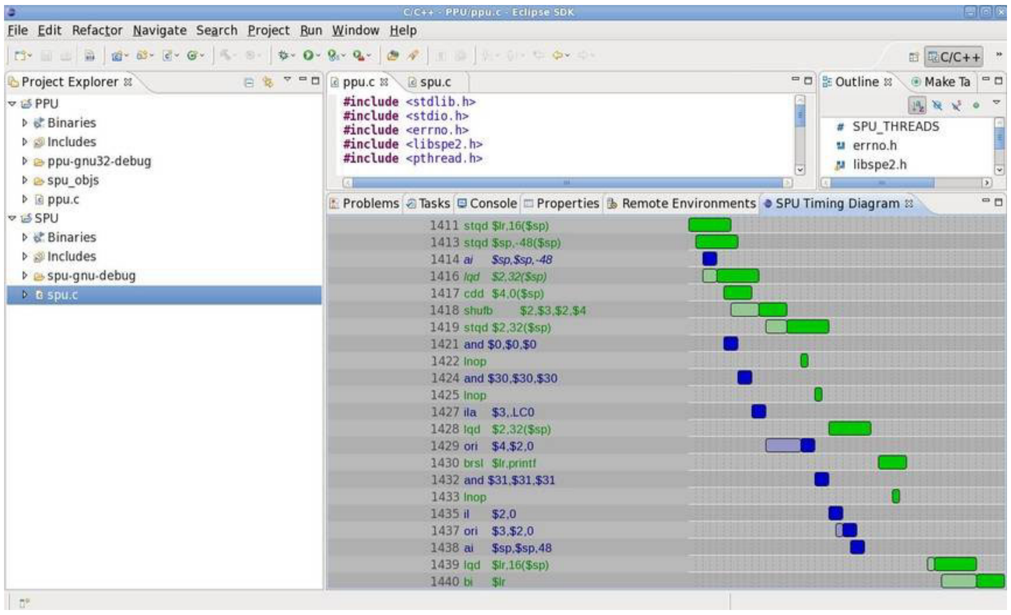
```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t SPU;

ADS     8

d_function(void *arg) {
r_t ctx;
ntry = SPE_DEFAULT_ENTRY;

ontext_ptr_t *)arg);
t_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
led running context");


ULL);
```

```
/systemsim-cell/include/callthru/spu -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wext
```

SPU timing tool - view the SPU timing output

You see the output from the SPU timing tool in the **Console** view.

Refresh your SPU project (right click SPU project, then click **Refresh**), to display a new directory named `sputiming`, which also contains the SPU timing output.



```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
```

```
1411 stqd $lr,16($sp)
1413 stqd $sp,-48($sp)
1414 ai   $sp,$sp,-48
1416 lqd  $2,32($sp)
1417 cdd  $4,0($sp)
1418 shufb     $2,$3,$2,$4
1419 stqd $2,32($sp)
1421 and $0,$0,$0
1422 lnop
1424 and $30,$30,$30
1425 lnop
1427 ila  $3,.LC0
1428 lqd  $2,32($sp)
1429 ori  $4,$2,0
1430 brsl $lr,printf
1432 and $31,$31,$31
1433 lnop
1435 il   $2,0
1437 ori  $3,$2,0
1438 ai   $sp,$sp,48
1439 lqd  $lr,16($sp)
1440 bi   $lr
```

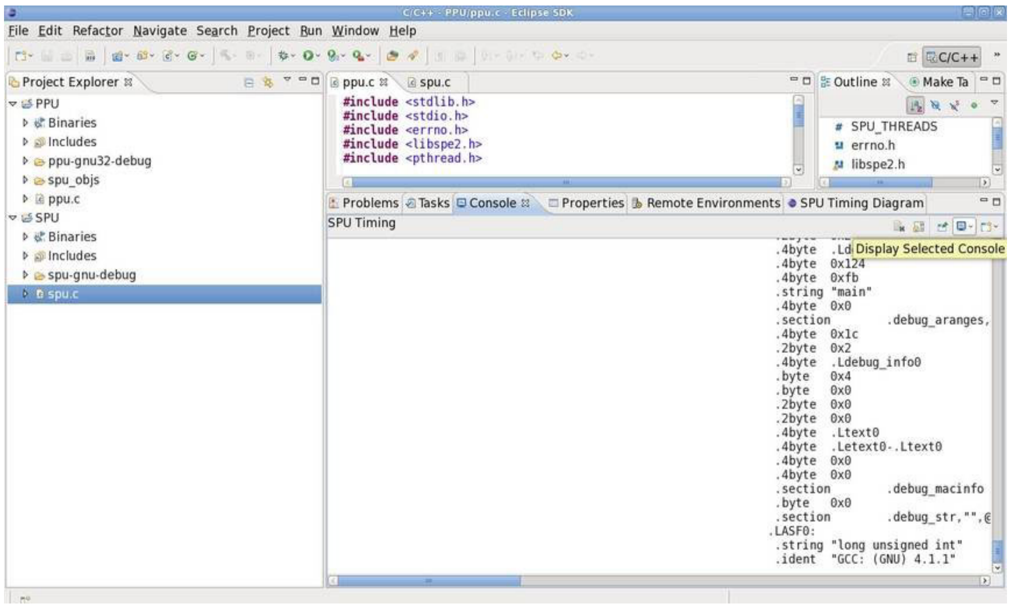
Simulator console - toggle the simulator console view

Objective: After you have completed this part of the tutorial you will know how to:

- Switch between available console views

In the **Console** view toolbar, a small blue monitor icon is displayed. You can use this icon to toggle between the available consoles.

Click the down arrow to the right of the blue monitor icon.

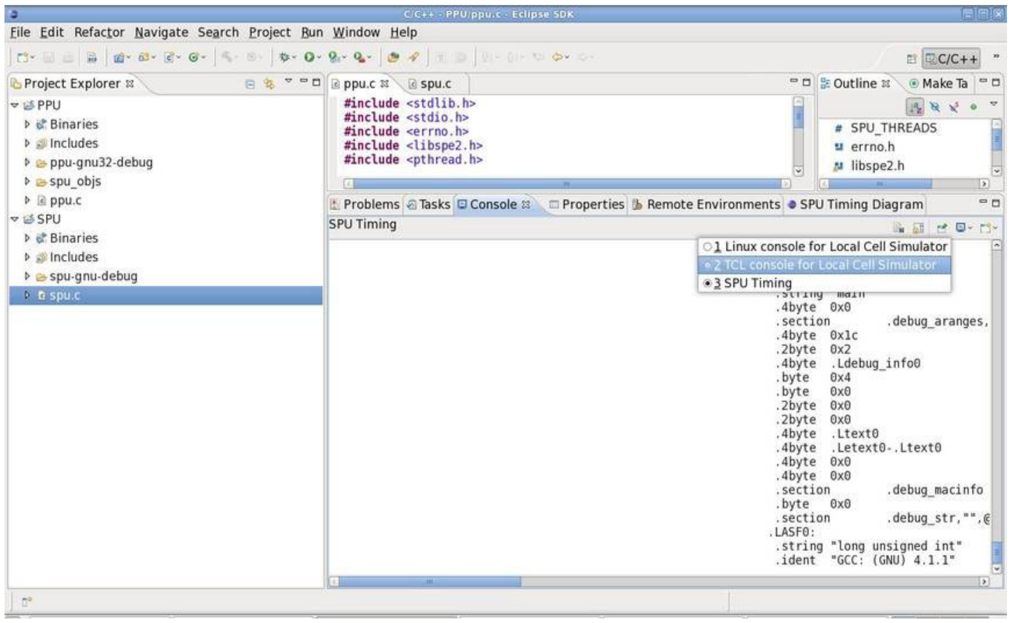


```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
```

```
 .4byte  .Ld
 .4byte  0x124
 .4byte  0xfb
 .string "main"
 .4byte  0x0
 .section        .debug_aranges,
 .4byte  0x1c
 .2byte  0x2
 .4byte  .Ldebug_info0
 .byte   0x4
 .byte   0x0
 .2byte  0x0
 .2byte  0x0
 .4byte  .Ltext0
 .4byte  .Letext0-.Ltext0
 .4byte  0x0
 .4byte  0x0
 .section        .debug_macinfo
 .byte   0x0
 .section        .debug_str,"",@
.LASF0:
 .string "long unsigned int"
 .ident  "GCC: (GNU) 4.1.1"
```

Simulator console - select the simulator TCL console

Select the option **TCL console for Local Cell Simulator**.

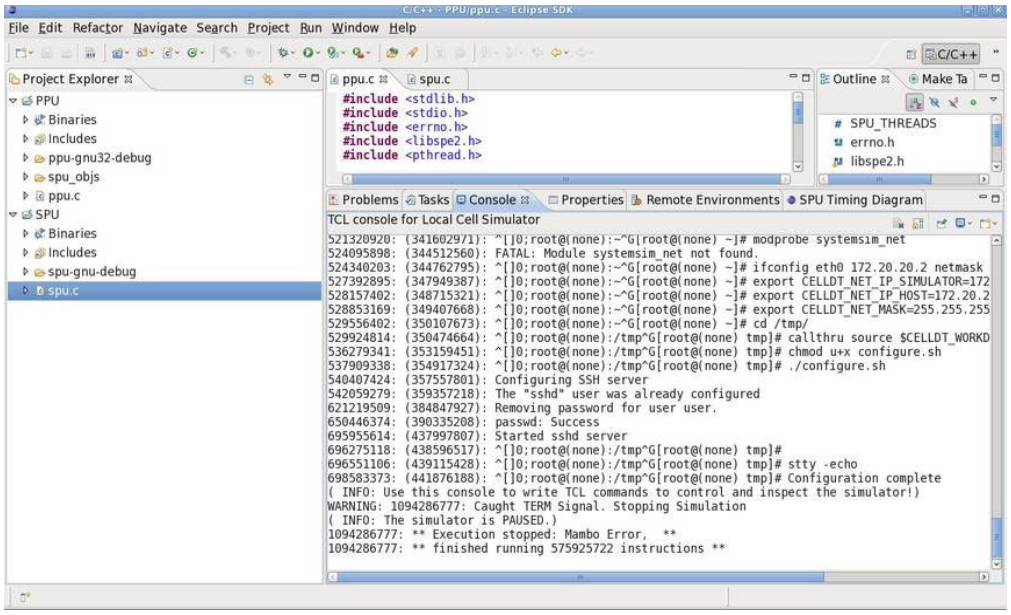


```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
```

```
 .string "main"
 .4byte  0x0
 .section        .debug_aranges,
 .4byte  0x1c
 .2byte  0x2
 .4byte  .Ldebug_info0
 .byte   0x4
 .byte   0x0
 .2byte  0x0
 .2byte  0x0
 .4byte  .Ltext0
 .4byte  .Letext0-.Ltext0
 .4byte  0x0
 .4byte  0x0
 .section        .debug_macinfo
 .byte   0x0
 .section        .debug_str,"",@
.LASF0:
 .string "long unsigned int"
 .ident  "GCC: (GNU) 4.1.1"
```

Simulator console - view the simulator's TCL console

This console can be used to pass TCL commands to the simulator.



```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
```

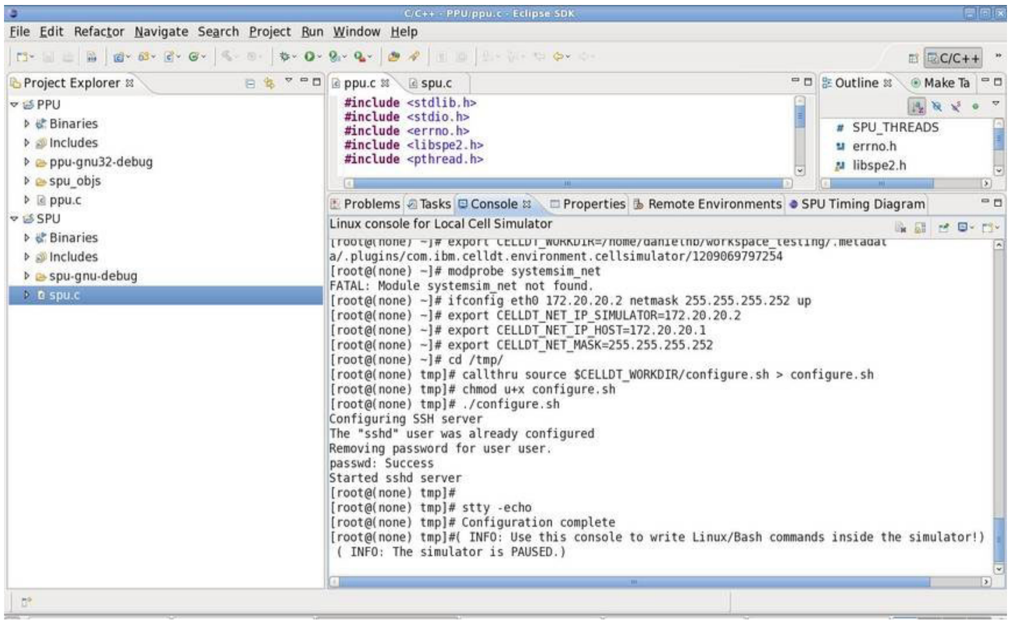
TCL console for Local Cell Simulator

```
521320920: (341602971): ^[]0;root@(none):~^G[root@(none) ~]# modprobe systemsim_net
524095898: (344512560): FATAL: Module systemsim_net not found.
524340203: (344762795): ^[]0;root@(none):~^G[root@(none) ~]# ifconfig eth0 172.20.20.2 netmask
527392895: (347949387): ^[]0;root@(none):~^G[root@(none) ~]# export CELLDT_NET_IP_SIMULATOR=172
528157402: (348715321): ^[]0;root@(none):~^G[root@(none) ~]# export CELLDT_NET_IP_HOST=172.20.2
528853169: (349407668): ^[]0;root@(none):~^G[root@(none) ~]# export CELLDT_NET_MASK=255.255.255
529556402: (350107673): ^[]0;root@(none):~^G[root@(none) ~]# cd /tmp/
529924814: (350474664): ^[]0;root@(none):/tmp^G[root@(none) tmp]# callthru source $CELLDT_WORKD
536279341: (353159451): ^[]0;root@(none):/tmp^G[root@(none) tmp]# chmod u+x configure.sh
537909338: (354917324): ^[]0;root@(none):/tmp^G[root@(none) tmp]# ./configure.sh
540407424: (357557801): Configuring SSH server
542059279: (359357218): The "sshd" user was already configured
621219509: (384847927): Removing password for user user.
650446374: (390335208): passwd: Success
695955614: (437997807): Started sshd server
696275118: (438596517): ^[]0;root@(none):/tmp^G[root@(none) tmp]#
696551106: (439115428): ^[]0;root@(none):/tmp^G[root@(none) tmp]# stty -echo
698583373: (441876188): ^[]0;root@(none):/tmp^G[root@(none) tmp]# Configuration complete
( INFO: Use this console to write TCL commands to control and inspect the simulator!)
WARNING: 1094286777: Caught TERM Signal. Stopping Simulation
( INFO: The simulator is PAUSED.)
1094286777: ** Execution stopped: Mambo Error,  **
1094286777: ** finished running 575925722 instructions **
```

Simulator console - use the simulator's Linux console

You can execute Linux commands inside the simulator (as long as the simulator is not stopped or paused).

In the **Console** view, switch to the console: **Linux Console for Local Cell Simulator**.



```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
```

```
[root@(none) ~]# export CELLDT_WORKDIR=/home/danielhb/workspace_testing/.metadat
a/.plugins/com.ibm.celldt.environment.cellsimulator/1209069797254
[root@(none) ~]# modprobe systemsim_net
FATAL: Module systemsim_net not found.
[root@(none) ~]# ifconfig eth0 172.20.20.2 netmask 255.255.255.252 up
[root@(none) ~]# export CELLDT_NET_IP_SIMULATOR=172.20.20.2
[root@(none) ~]# export CELLDT_NET_IP_HOST=172.20.20.1
[root@(none) ~]# export CELLDT_NET_MASK=255.255.255.252
[root@(none) ~]# cd /tmp/
[root@(none) tmp]# callthru source $CELLDT_WORKDIR/configure.sh > configure.sh
[root@(none) tmp]# chmod u+x configure.sh
[root@(none) tmp]# ./configure.sh
Configuring SSH server
The "sshd" user was already configured
Removing password for user user.
passwd: Success
Started sshd server
[root@(none) tmp]#
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

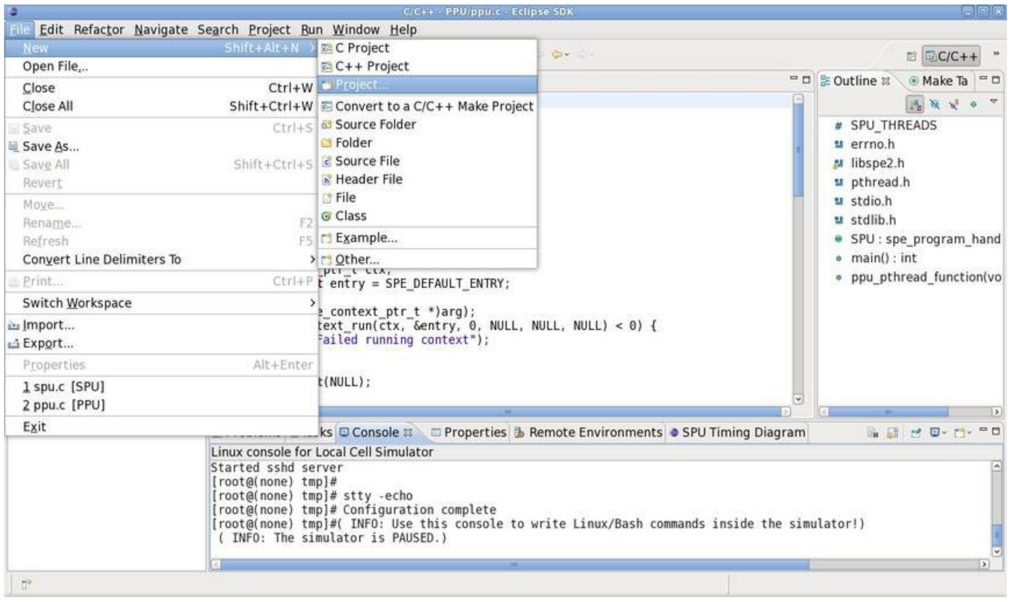
ALF API wizard

Objectives: After you have completed this part of the tutorial you will know how to use the ALF API wizard to:

- Create an ALF project
- Import the generated source code and configure the projects' properties to include the necessary libraries and to enable embedded SPU functionality
- Modify the ALF API wizard parameters
- Create and configure a new Cell/B.E. application launcher, so that the shared library project's .so file (libmy_alf_project.so) can be loaded during the remote application execution.
- Launch an ALF application

As of version 3.0, IDE includes a wizard which allows you to easily create a Cell/B.E. application that utilizes the Accelerated Library Framework (ALF) API. The ALF API provides a set of functions to a set of parallel problems on multi-core, host-accelerator type systems, such as the Cell/B.E. processor. Features of ALF include data transfer management, parallel task management, double buffering, and data partitioning. The next section of the tutorial walks you through an example use case of the ALF API Wizard.

To start using the wizard, click **File → New → Project**.

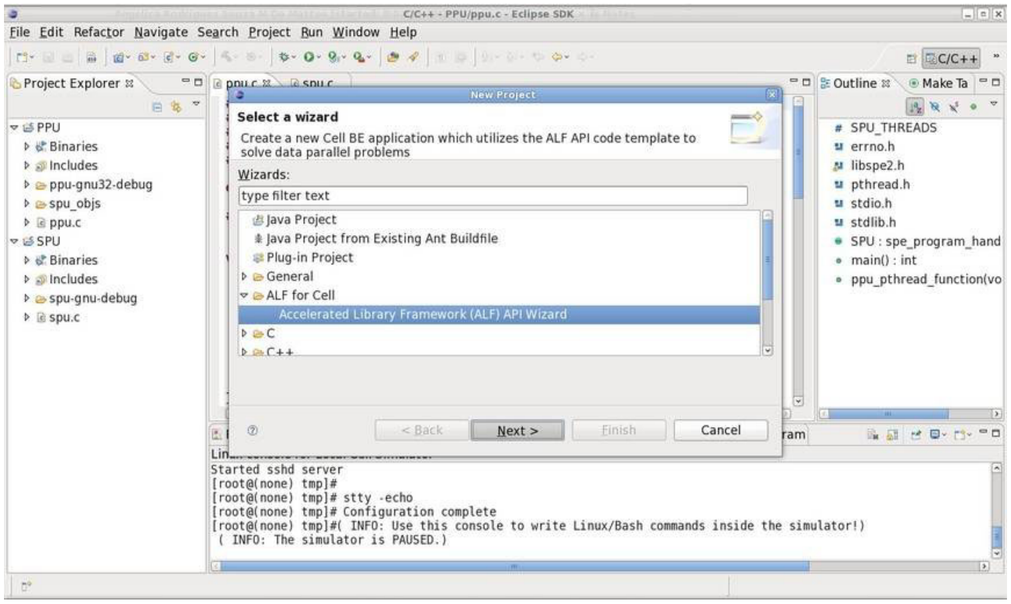


```
t entry = SPE_DEFAULT_ENTRY;

e_context_ptr_t *)arg);
text_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
Failed running context");

t(NULL);
```

```
Linux console for Local Cell Simulator
Started sshd server
[root@(none) tmp]#
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

ALF API wizard - start the ALF IDE wizard

1. Expand the **ALF for Cell** category.
2. Select the **Accelerated Library Framework (ALF) API Wizard** option.
3. Click **Next**.



Select a wizard

Create a new Cell BE application which utilizes the ALF API code template to solve data parallel problems

Wizards:

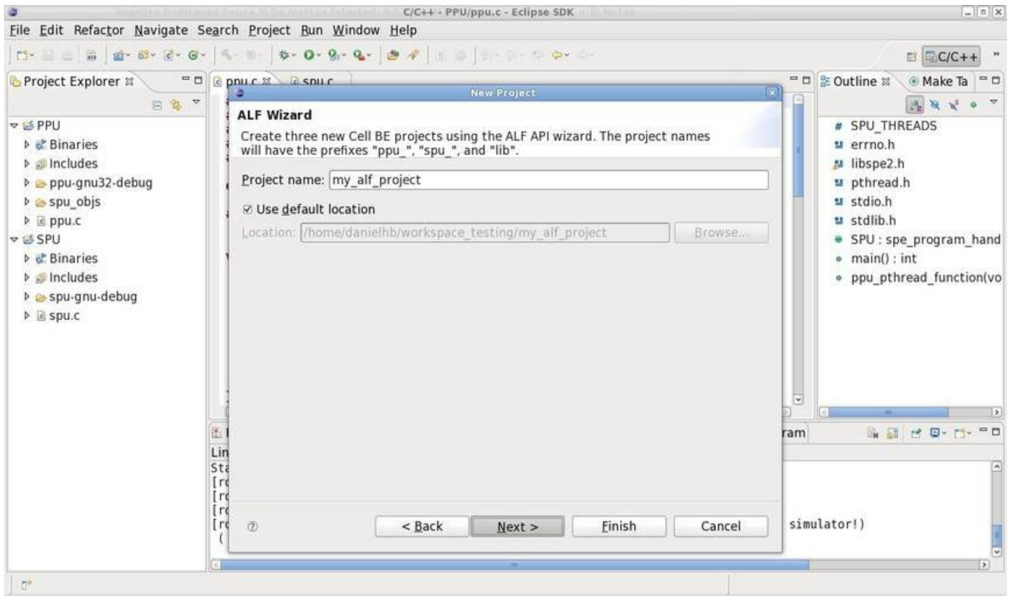
type filter text

- Java Project
- Java Project from Existing Ant Buildfile
- Plug-in Project
- General
- ALF for Cell
  - Accelerated Library Framework (ALF) API Wizard
- C
- C++

```
Started sshd server
[root@(none) tmp]#
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
```

ALF API wizard - enter a project name

Enter a project name, then click **Next**.



ALF Wizard

Create three new Cell BE projects using the ALF API wizard. The project names will have the prefixes "ppu_", "spu_", and "lib".

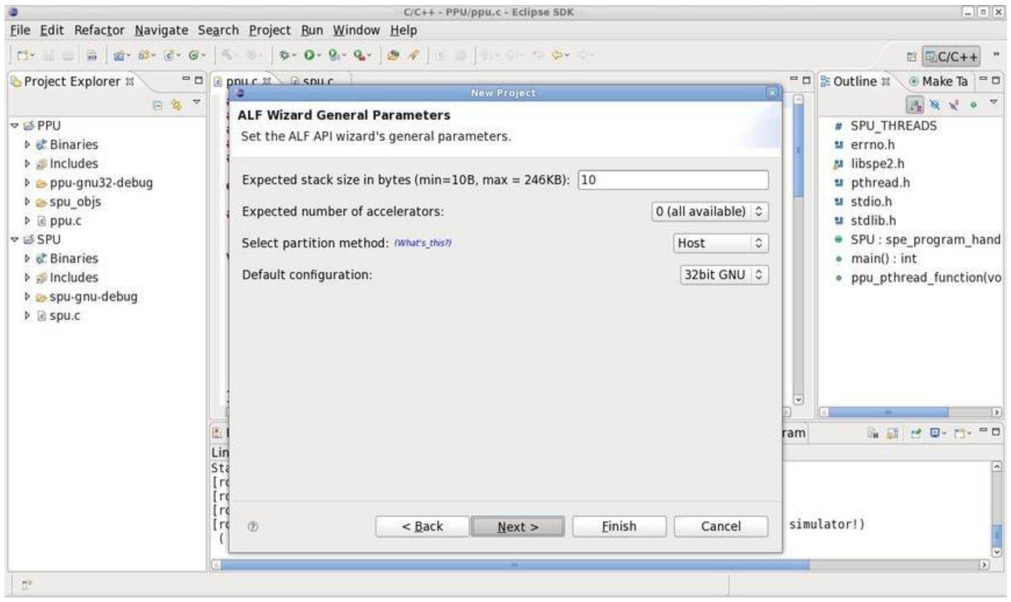
Project name: my_alf_project

☑ Use default location

Location: /home/danielhb/workspace_testing/my_alf_project

ALF API wizard - change general parameters

Here you can change the ALF IDE wizard's general parameters. You can return to this page at any time if you need to make any changes.



ALF Wizard General Parameters

Set the ALF API wizard's general parameters.

Expected stack size in bytes (min=10B, max = 246KB): 10

Expected number of accelerators: 0 (all available)

Select partition method: Host

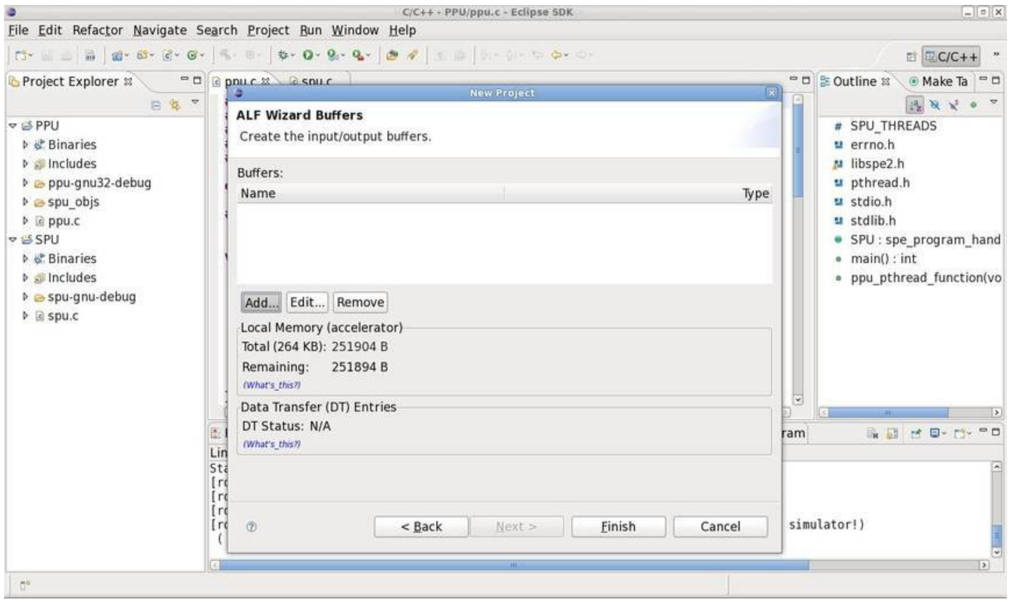
Default configuration: 32bit GNU

Click **Next**.

ALF API wizard - add ALF buffers

This page displays any existing buffers, allows you to create new buffers, displays remaining SPU local memory, and displays the data transfer entry status. Click the corresponding **What's this?** text to learn more about the local memory and data transfer status sections.

Click the **Add** button to create a new buffer.



ALF Wizard Buffers

Create the input/output buffers.

Buffers:

Name	Type

Local Memory (accelerator)

Total (264 KB): 251904 B

Remaining: 251894 B

(What's_this?)

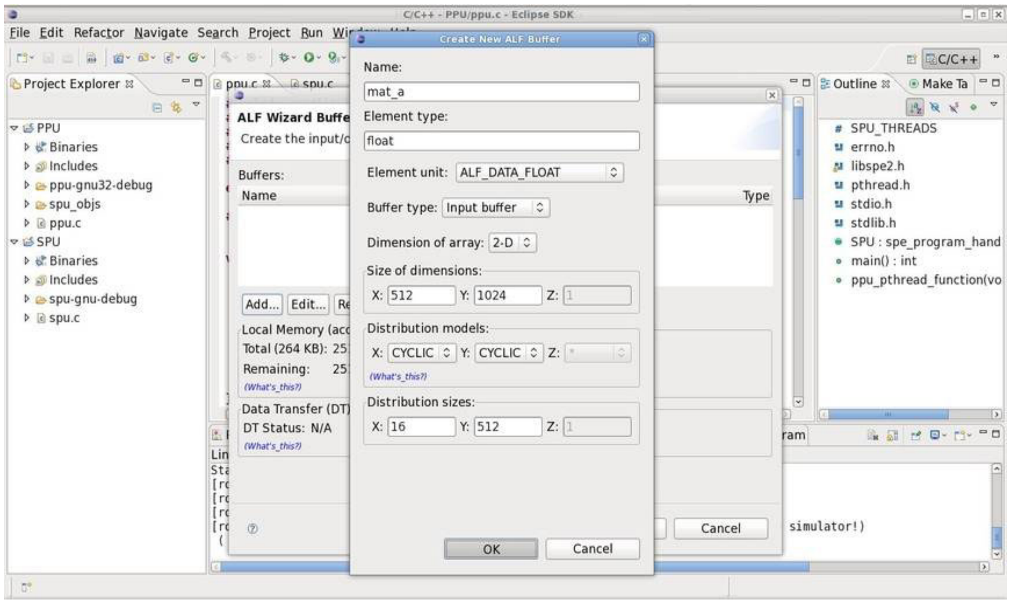
Data Transfer (DT) Entries

DT Status: N/A

(What's_this?)

ALF API wizard - add the first buffer

Enter information in the fields as shown below, then click **OK**.



Create New ALF Buffer

Name: mat_a

Element type: float

Element unit: ALF_DATA_FLOAT

Buffer type: Input buffer

Dimension of array: 2-D

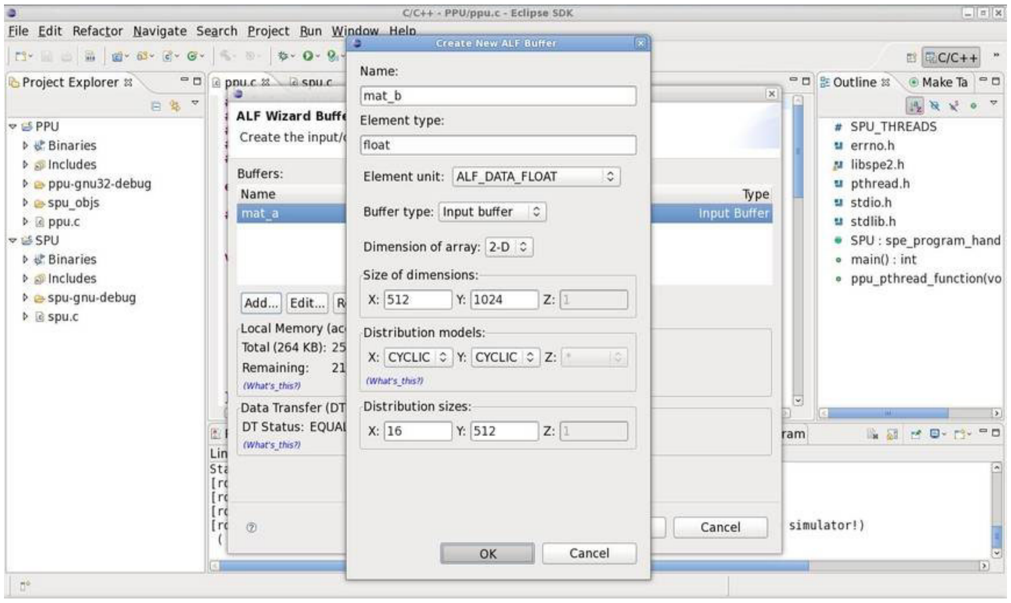
Size of dimensions: X: 512 Y: 1024 Z: 1

Distribution models: X: CYCLIC Y: CYCLIC Z:

Distribution sizes: X: 16 Y: 512 Z: 1

ALF API wizard - add a second buffer

Use the **Add** button to create another buffer, enter the parameters as shown below (use the same values as `mat_a`), then click **OK**.



Create New ALF Buffer

Name: mat_b

Element type: float

Element unit: ALF_DATA_FLOAT

Buffer type: Input buffer

Dimension of array: 2-D

Size of dimensions: X: 512 Y: 1024 Z: 1

Distribution models: X: CYCLIC Y: CYCLIC Z: *

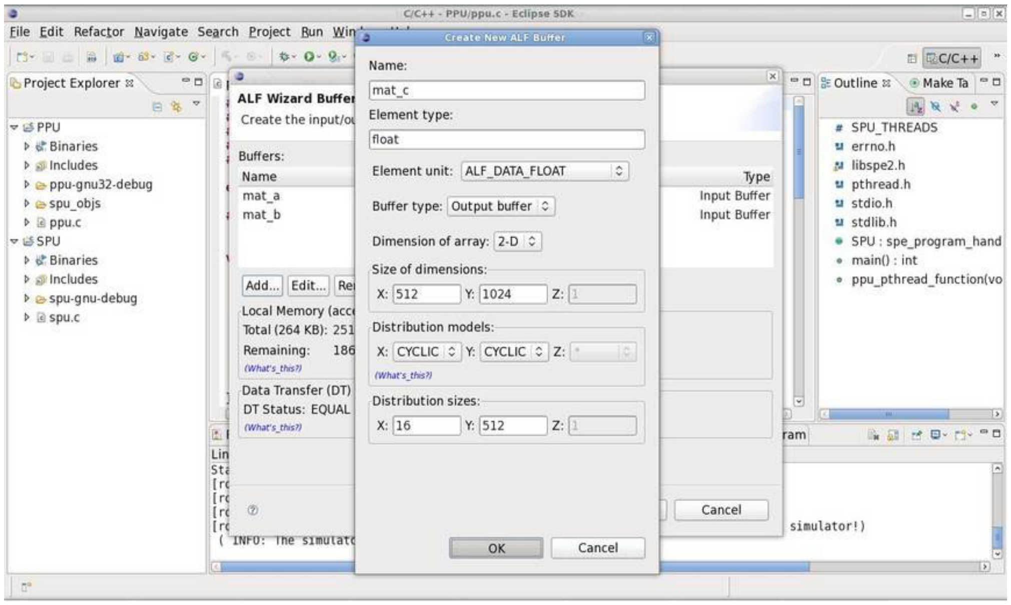
Distribution sizes: X: 16 Y: 512 Z: 1

OK Cancel

Name	Type
mat_a	Input Buffer

ALF API wizard - add the third buffer

Use the **Add** button to create the third and final buffer, enter the parameters as shown below (same values as `mat_a` and `mat_b`, except for the **Buffer type**), then click **OK**.

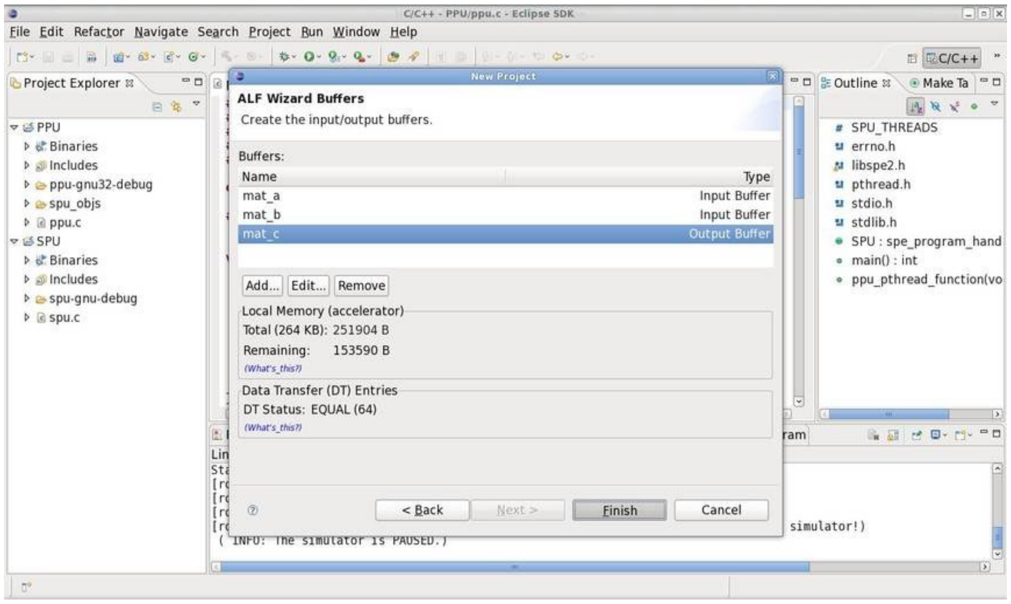


Name: mat_c
Element type: float
Element unit: ALF_DATA_FLOAT
Buffer type: Output buffer
Dimension of array: 2-D
Size of dimensions: X: 512 Y: 1024 Z: 1
Distribution models: X: CYCLIC Y: CYCLIC Z:
Distribution sizes: X: 16 Y: 512 Z: 1

Name	Type
mat_a	Input Buffer
mat_b	Input Buffer

ALF API wizard - finish using the ALF IDE wizard

Click **Finish** to complete the wizard. The wizard automatically creates and configures the three new projects.



ALF Wizard Buffers
Create the input/output buffers.

Buffers:

Name	Type
mat_a	Input Buffer
mat_b	Input Buffer
mat_c	Output Buffer

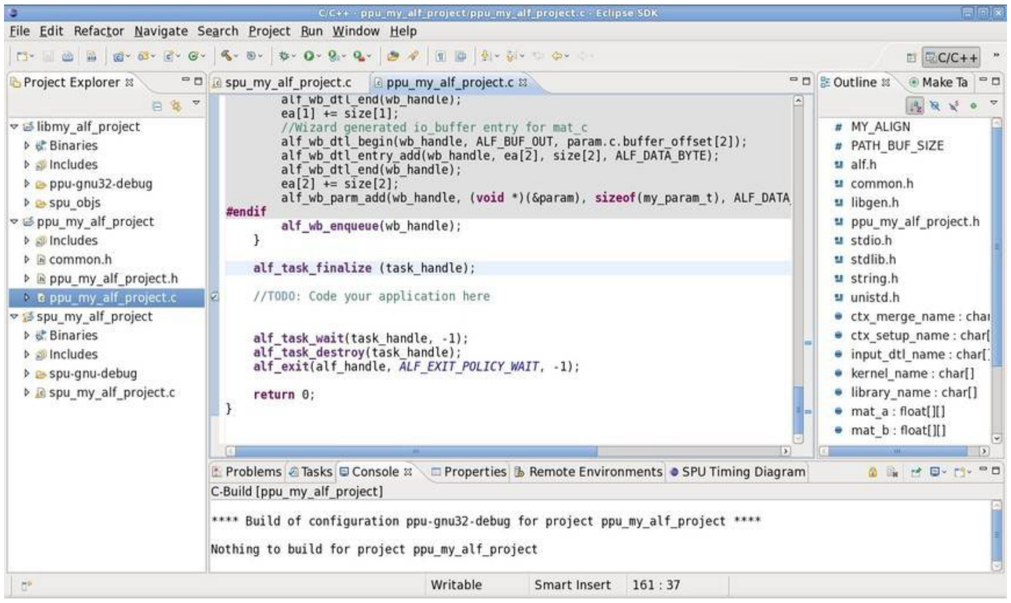
Local Memory (accelerator)
Total (264 KB): 251904 B
Remaining: 153590 B
(What's_this?)

Data Transfer (DT) Entries
DT Status: EQUAL (64)
(What's_this?)

```
simulator!)
( INFO: The simulator is PAUSED.)
```

ALF API wizard - ALF IDE wizard complete

The three new projects have now been created, the generated source code has been imported, and the projects' properties have been configured to include the necessary libraries and to enable embed SPU functionality. Looking at the source code, you notice that the ALF IDE wizard has generated code to handle many trivial tasks, such as input/output transfer list preparation, but you must write your own code for the computing kernel and data initialization.



```
            alf_wb_dtl_end(wb_handle);
            ea[1] += size[1];
            //Wizard generated io_buffer entry for mat_c
            alf_wb_dtl_begin(wb_handle, ALF_BUF_OUT, param.c.buffer_offset[2]);
            alf_wb_dtl_entry_add(wb_handle, ea[2], size[2], ALF_DATA_BYTE);
            alf_wb_dtl_end(wb_handle);
            ea[2] += size[2];
            alf_wb_parm_add(wb_handle, (void *)(&param), sizeof(my_param_t), ALF_DATA
#endif
            alf_wb_enqueue(wb_handle);
        }

    alf_task_finalize (task_handle);

    //TODO: Code your application here


    alf_task_wait(task_handle, -1);
    alf_task_destroy(task_handle);
    alf_exit(alf_handle, ALF_EXIT_POLICY_WAIT, -1);

    return 0;
}
```

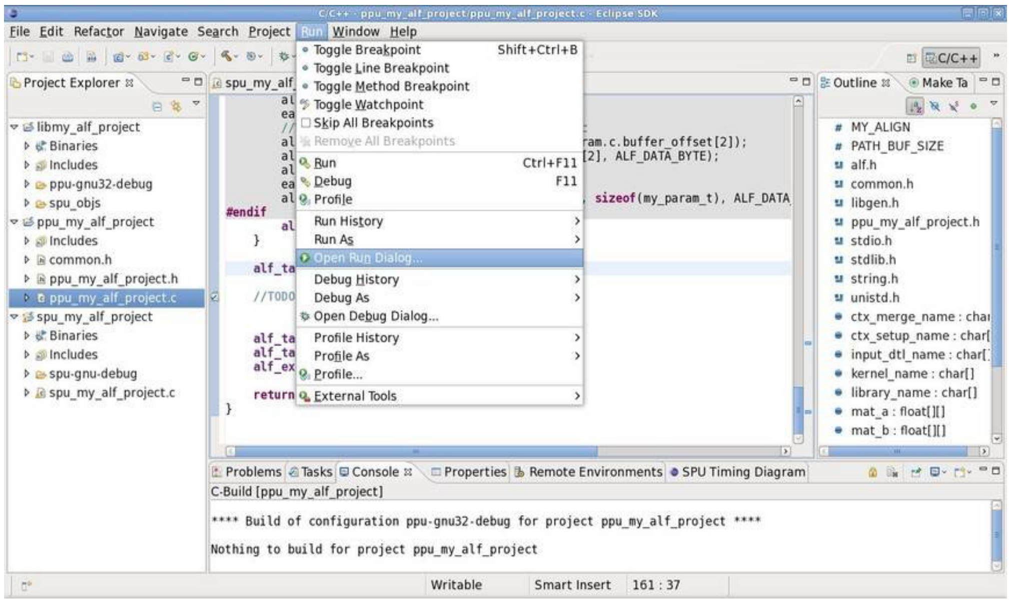
C-Build [ppu_my_alf_project]

```
**** Build of configuration ppu-gnu32-debug for project ppu_my_alf_project ****

Nothing to build for project ppu_my_alf_project
```

ALF API wizard - Create new launch configuration

Finally, you must create and configure a new Cell/B.E. application launcher, so that the shared library project's .so file (libmy_alf_project.so) can be loaded during the remote application execution.

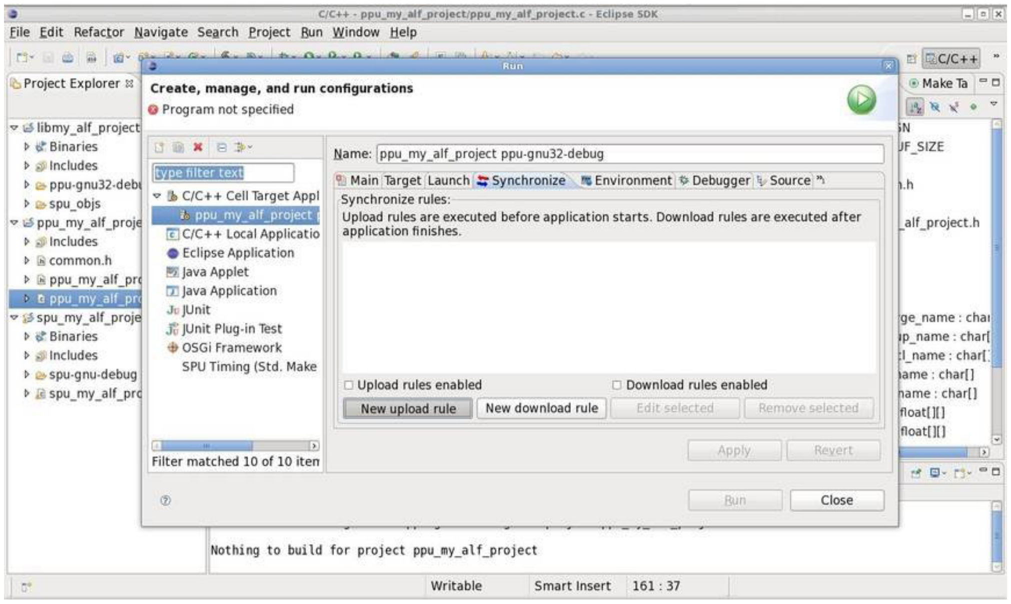
Click **Run** ➔ **Open Run Dialog...**



```
**** Build of configuration ppu-gnu32-debug for project ppu_my_alf_project ****

Nothing to build for project ppu_my_alf_project
```

ALF API wizard - add a new upload rule

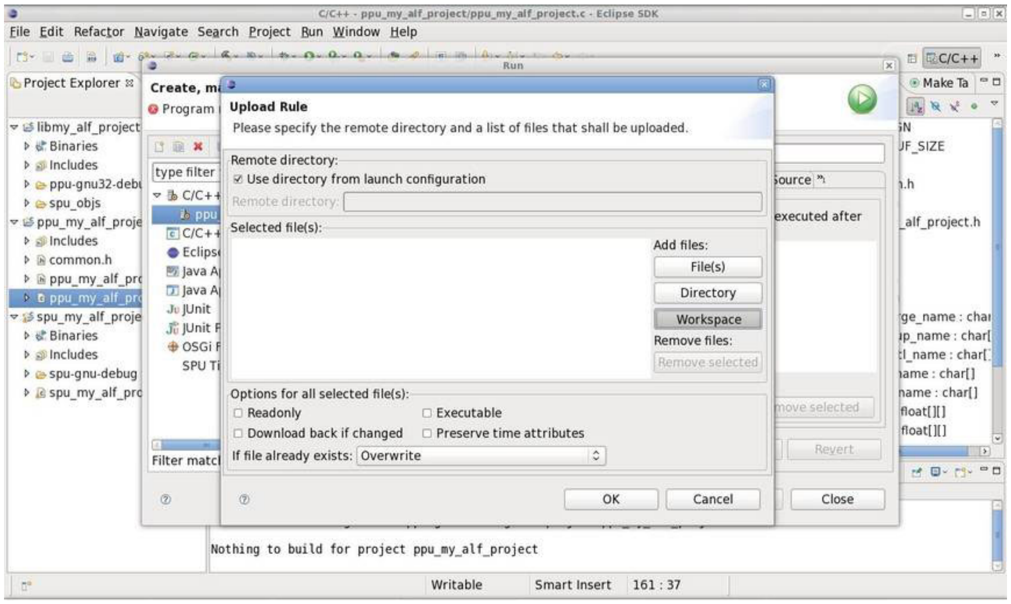
1. Create a new **C/C++ Cell Target Application** configuration (like you did earlier in the tutorial).
2. Go to the **Synchronize** tab, then click **New upload rule**.



Create, manage, and run configurations
Program not specified
Name: ppu_my_alf_project ppu-gnu32-debug
Main Target Launch Synchronize Environment Debugger Source
Synchronize rules:
Upload rules are executed before application starts. Download rules are executed after application finishes.
Upload rules enabled
Download rules enabled
New upload rule
New download rule
Edit selected
Remove selected
Apply
Revert
Run
Close
Filter matched 10 of 10 items
Nothing to build for project ppu_my_alf_project

ALF API wizard - create a new upload rule - step 2

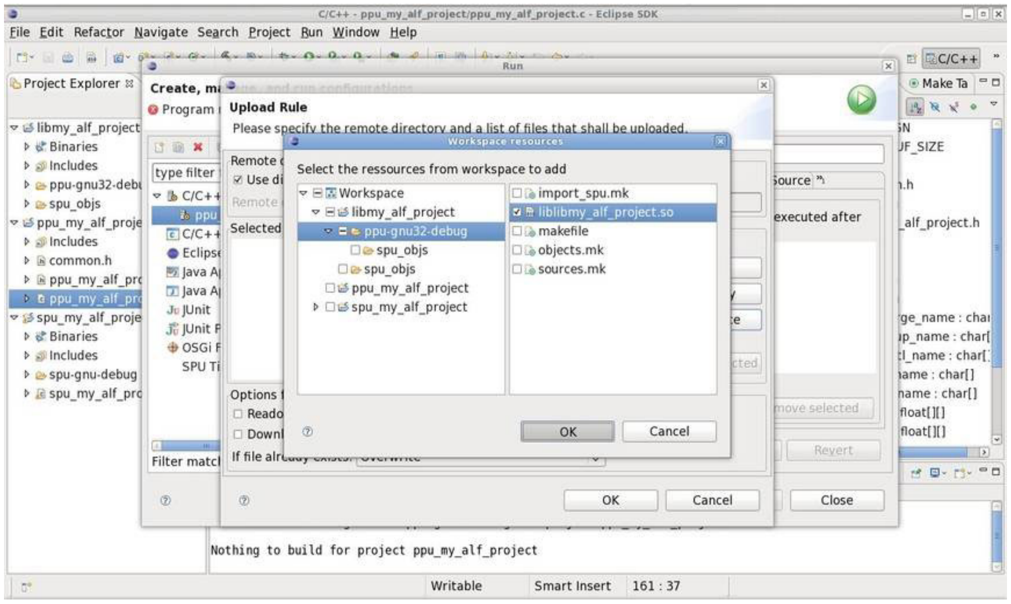
Make sure that the option **Use directory from launch configuration** is checked, then click the **Workspace** button.


Upload Rule
Please specify the remote directory and a list of files that shall be uploaded.
Remote directory:
☑ Use directory from launch configuration
Remote directory:
Selected file(s):
Add files:
File(s)
Directory
Workspace
Remove files:
Remove selected
Options for all selected file(s):
☐ Readonly
☐ Executable
☐ Download back if changed
☐ Preserve time attributes
If file already exists: Overwrite
OK
Cancel

```
Nothing to build for project ppu_my_alf_project
```

ALF API wizard - select the shared library .so file

1. Expand the **Workspace** and **libmy_alf_project** categories.
2. Click the **ppu-gnu32-debug** folder to view its contents.
3. Check the box for **libmy_alf_project.so**
4. Click **OK** twice to return to the **Synchronize** tab.

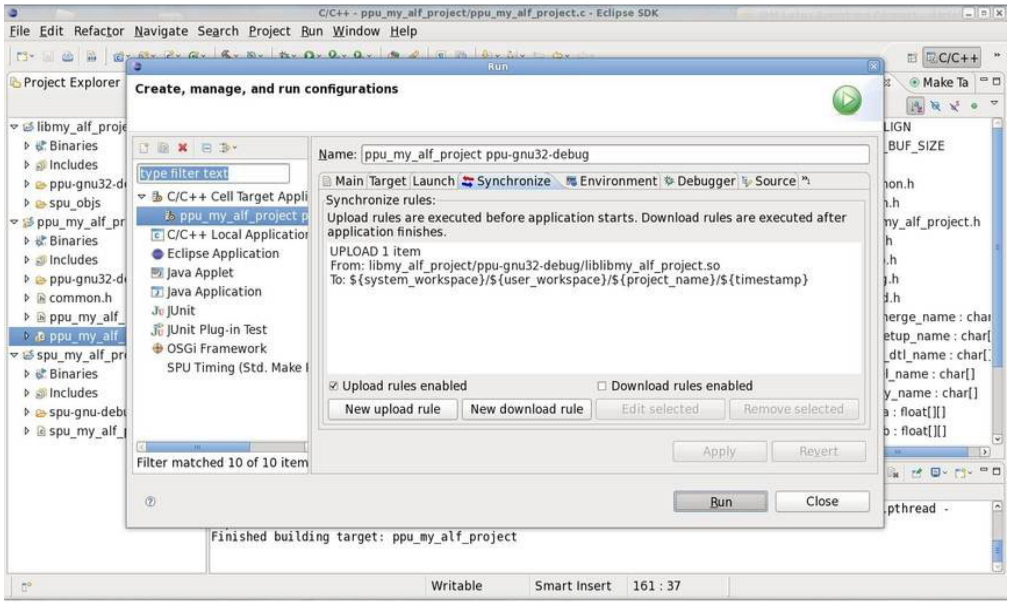


Nothing to build for project ppu_my_alf_project

ALF API wizard - launch an ALF application

All that remains to be done is to check the box labeled **Upload rules enabled**. The libmy_alf_project.so file will now be copied to the working directory automatically before the application runs, so the PPU project executable can load it during runtime.

Click **Run** to begin running the application.



Filter matched 10 of 10 item

Name: ppu_my_alf_project ppu-gnu32-debug

Synchronize rules:
Upload rules are executed before application starts. Download rules are executed after application finishes.

UPLOAD 1 item
From: libmy_alf_project/ppu-gnu32-debug/liblibmy_alf_project.so
To: ${system_workspace}/${user_workspace}/${project_name}/${timestamp}

☑ Upload rules enabled ☐ Download rules enabled

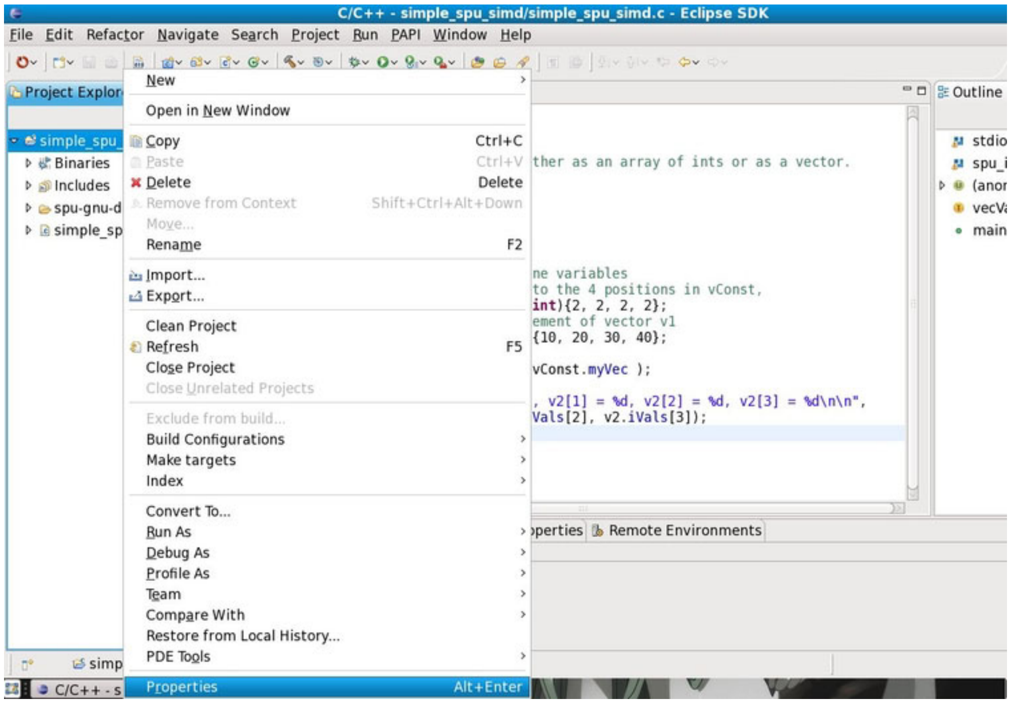
```
Finished building target: ppu_my_alf_project
```

IDE PDT plugin

Objective: After you have completed this part of the tutorial, you will know how to:

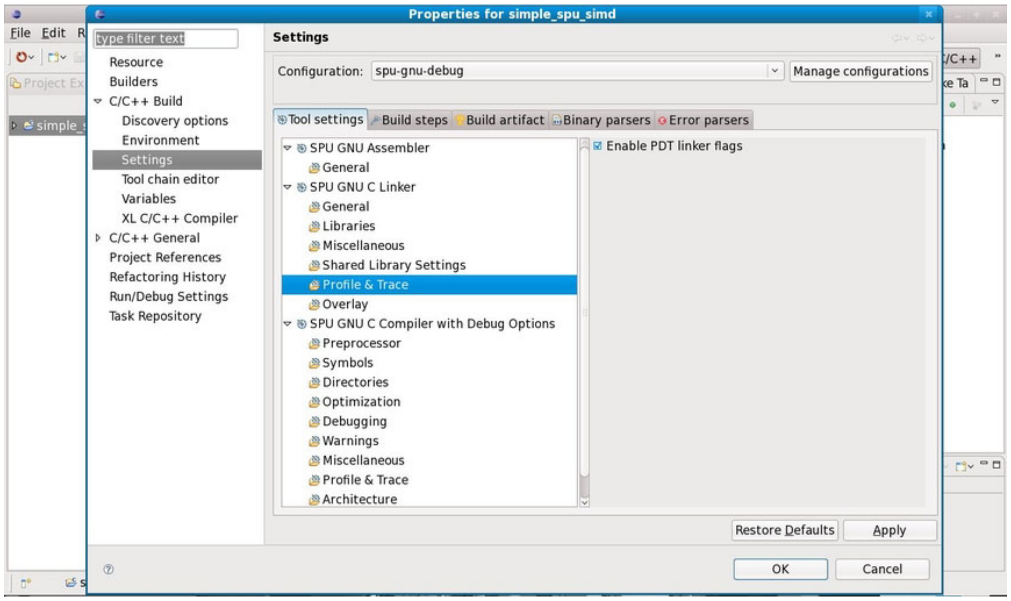
- Use the PDT to generate and view trace files

IDE provides an easy way to generate trace files using PDT - Performance Debugging Tools. First you must set PDT flags for your project. To do this, right click the project and choose **Properties**.



IDE PDT plugin - set up the C Linker flag

1. In the **C/C++ Build**, **Settings** category, go to C Linker options.
2. Select **Profile & Trace**.
3. Check **Enable PDT compiler flags**.



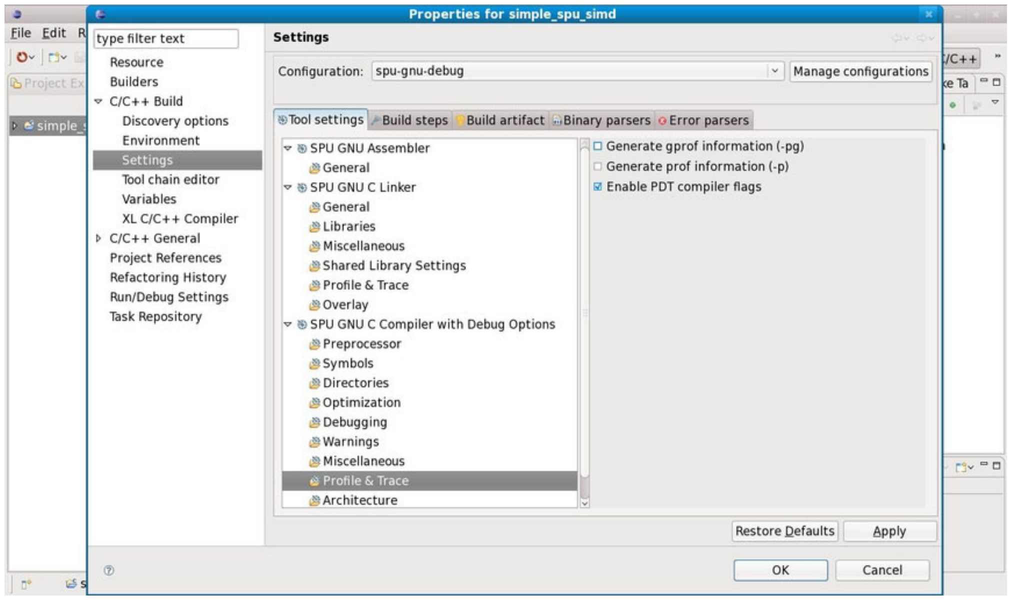
Configuration: spu-gnu-debug

☑ Enable PDT linker flags

IDE PDT plugin - set up the C compiler flag

To set up the C compiler flag, do the following:

1. Go to C Compiler with Debug Options.
2. Select **Profile & Trace**.
3. Check **Enable PDT compiler flags**.
4. Click **OK** to finish.

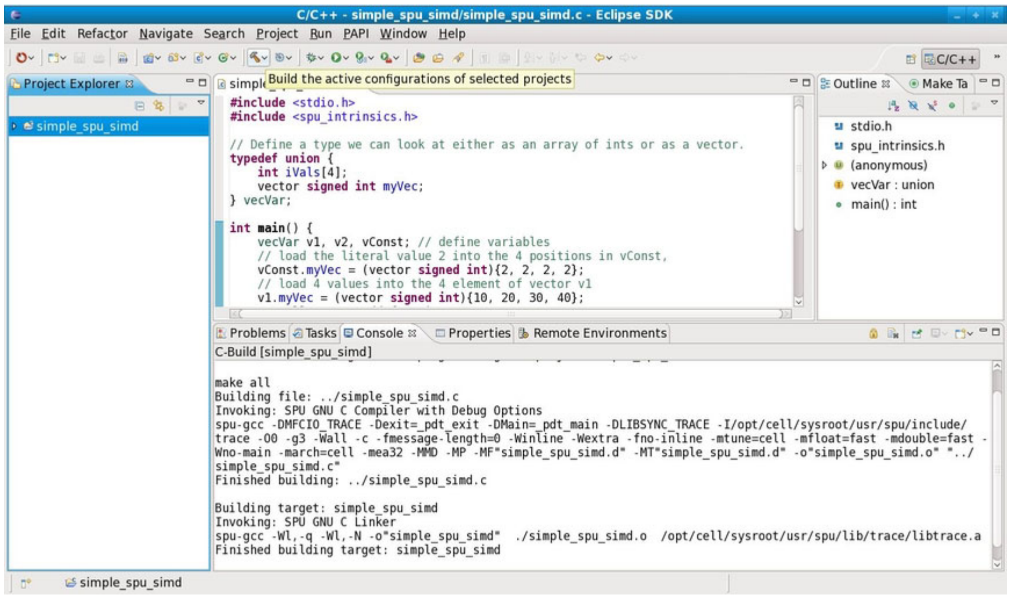


Configuration: spu-gnu-debug

☐ Generate gprof information (-pg)
☐ Generate prof information (-p)
☑ Enable PDT compiler flags

IDE PDT plugin - rebuild the project with PDT flags

Rebuild the project with PDT flags. The project must be built without any errors. Now you need to create a XML configuration file for PDT.

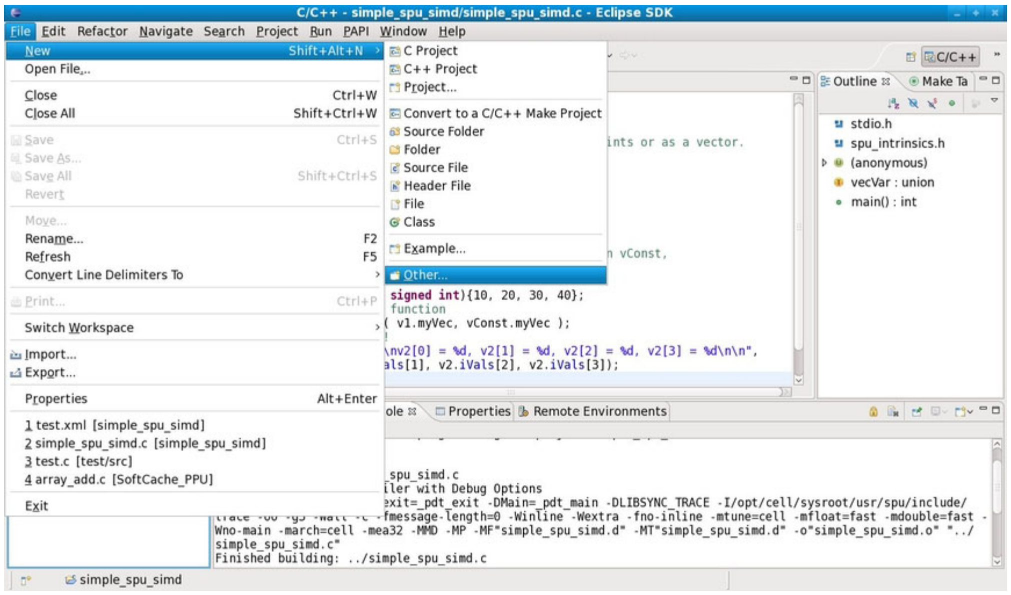

C/C++ - simple_spu_simd/simple_spu_simd.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run PAPI Window Help
C/C++
Project Explorer
simple_spu_simd
Build the active configurations of selected projects
Outline
Make Ta
stdio.h
spu_intrinsics.h
(anonymous)
vecVar : union
main() : int
Problems
Tasks
Console
Properties
Remote Environments
C-Build [simple_spu_simd]
simple_spu_simd

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

// Define a type we can look at either as an array of ints or as a vector.
typedef union {
    int iVals[4];
    vector signed int myVec;
} vecVar;

int main() {
    vecVar v1, v2, vConst; // define variables
    // load the literal value 2 into the 4 positions in vConst,
    vConst.myVec = (vector signed int){2, 2, 2, 2};
    // load 4 values into the 4 element of vector v1
    v1.myVec = (vector signed int){10, 20, 30, 40};
```

```
make all
Building file: ../simple_spu_simd.c
Invoking: SPU GNU C Compiler with Debug Options
spu-gcc -DMFCIO_TRACE -Dexit=_pdt_exit -DMain=_pdt_main -DLIBSYNC_TRACE -I/opt/cell/sysroot/usr/spu/include/
trace -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -fno-inline -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -march=cell -mea32 -MMD -MP -MF"simple_spu_simd.d" -MT"simple_spu_simd.d" -o"simple_spu_simd.o" "../
simple_spu_simd.c"
Finished building: ../simple_spu_simd.c

Building target: simple_spu_simd
Invoking: SPU GNU C Linker
spu-gcc -Wl,-q -Wl,-N -o"simple_spu_simd"  ./simple_spu_simd.o  /opt/cell/sysroot/usr/spu/lib/trace/libtrace.a
Finished building target: simple_spu_simd
```

IDE PDT plugin - create a PDT configuration file

To create our XML configuration file for PDT, go to **File** → **New** → **Other**

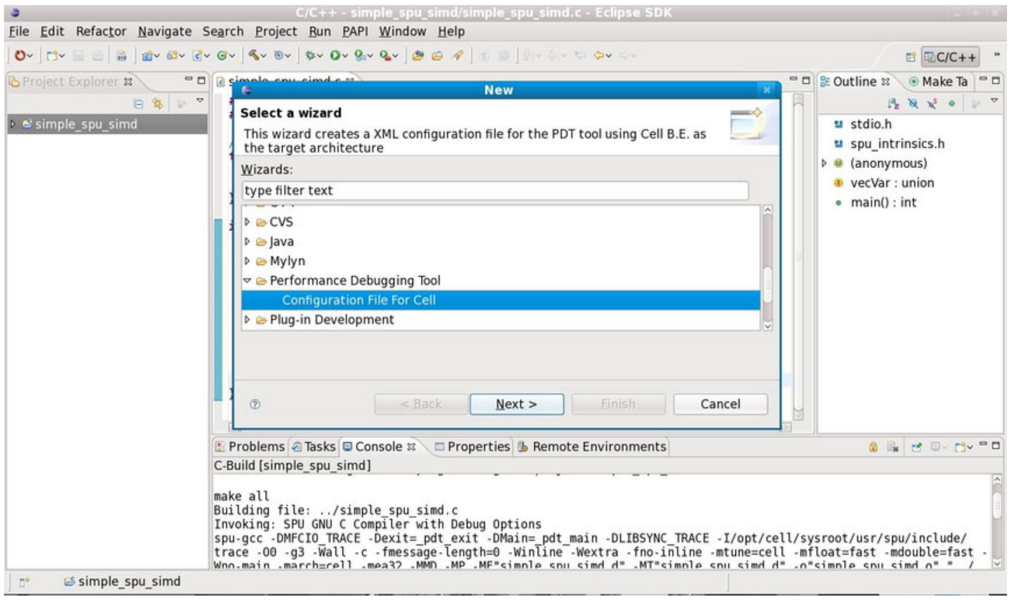


```
ints or as a vector.
n vConst,
signed int){10, 20, 30, 40};
function
( v1.myVec, vConst.myVec );
\nv2[0] = %d, v2[1] = %d, v2[2] = %d, v2[3] = %d\n\n",
als[1], v2.iVals[2], v2.iVals[3]);
```

```
_spu_simd.c
iler with Debug Options
exit=_pdt_exit -DMain=_pdt_main -DLIBSYNC_TRACE -I/opt/cell/sysroot/usr/spu/include/
-fmessage-length=0 -Winline -Wextra -fno-inline -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -march=cell -mea32 -MMD -MP -MF"simple_spu_simd.d" -MT"simple_spu_simd.d" -o"simple_spu_simd.o" "../
simple_spu_simd.c"
Finished building: ../simple_spu_simd.c
```

IDE PDT plugin - open the wizard

To open the wizard, in the **Performance Debugging Tool** category, select **Configuration File for Cell**, then click **Next**.

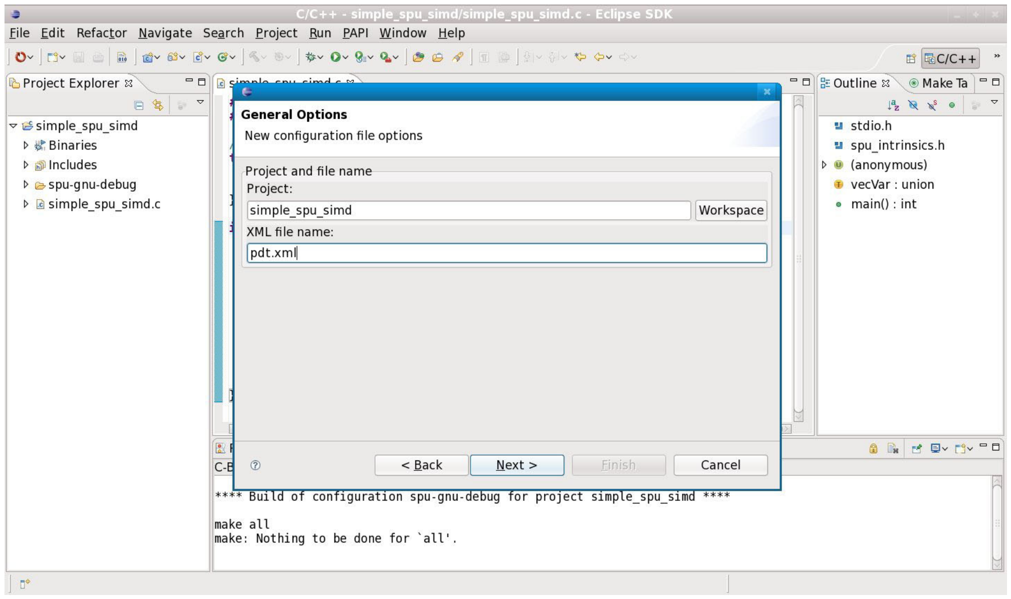


This wizard creates a XML configuration file for the PDT tool using Cell B.E. as the target architecture

```
make all
Building file: ../simple_spu_simd.c
Invoking: SPU GNU C Compiler with Debug Options
spu-gcc -DMFCIO_TRACE -Dexit=_pdt_exit -DMain=_pdt_main -DLIBSYNC_TRACE -I/opt/cell/sysroot/usr/spu/include/
trace -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -fno-inline -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -march=cell -mea32 -MMD -MP -MF"simple_spu_simd.d" -MT"simple_spu_simd.d" -o"simple_spu_simd.o" "../
```

IDE PDT plugin - enter a file name

Select a location and a name for the XML file, then click **Next**.



General Options

New configuration file options

Project and file name

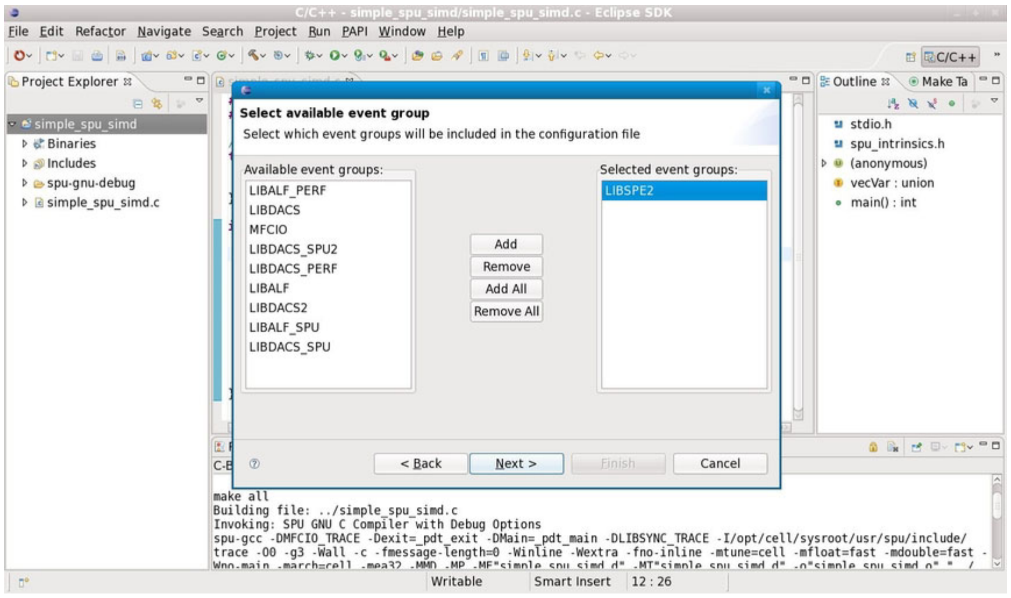
Project: simple_spu_simd

XML file name: pdt.xml

```
**** Build of configuration spu-gnu-debug for project simple_spu_simd ****

make all
make: Nothing to be done for `all'.
```

IDE PDT plugin - select event groups

In this window you can select which event groups your PDT session will capture. Select any event group and click **Next**.



Select available event group

Select which event groups will be included in the configuration file

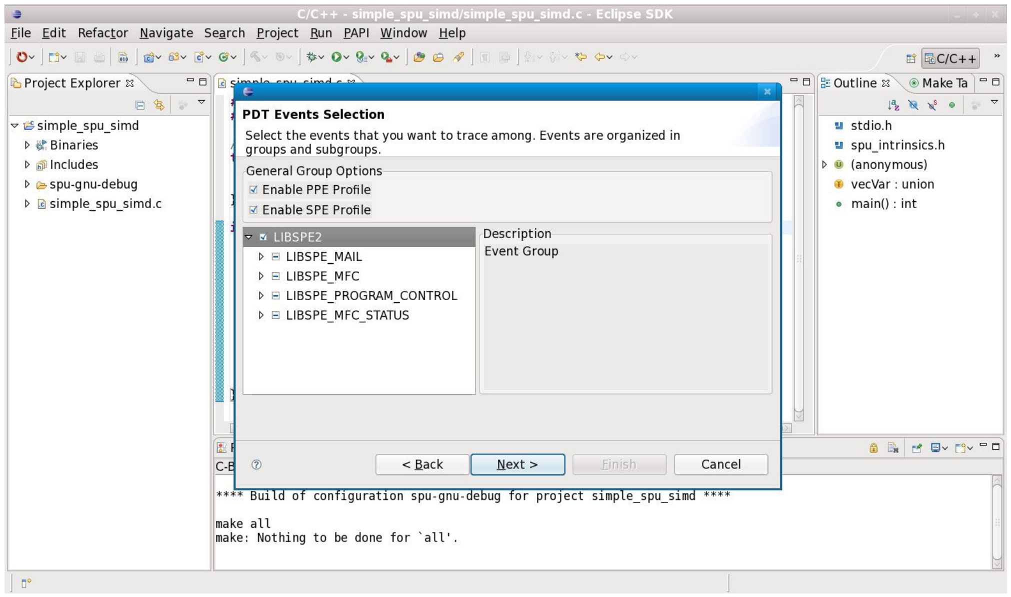
Available event groups:	Selected event groups:
LIBALF_PERF	LIBSPE2
LIBDACS	
MFCIO	
LIBDACS_SPU2	
LIBDACS_PERF	
LIBALF	
LIBDACS2	
LIBALF_SPU	
LIBDACS_SPU	

```
make all
Building file: ../simple_spu_simd.c
Invoking: SPU GNU C Compiler with Debug Options
spu-gcc -DMFCIO_TRACE -Dexit=_pdt_exit -DMain=_pdt_main -DLIBSYNC_TRACE -I/opt/cell/sysroot/usr/spu/include/
trace -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -fno-inline -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -march=cell -mea32 -MMD -MP -MF"simple_spu_simd.d" -MT"simple_spu_simd.d" -o"simple_spu_simd.o" "../
```

IDE PDT plugin - select sub-events

Next you need to fine tune your event selection by selecting events inside the events groups you chose before.

Click **Next** when you have finished.



Select the events that you want to trace among. Events are organized in groups and subgroups.

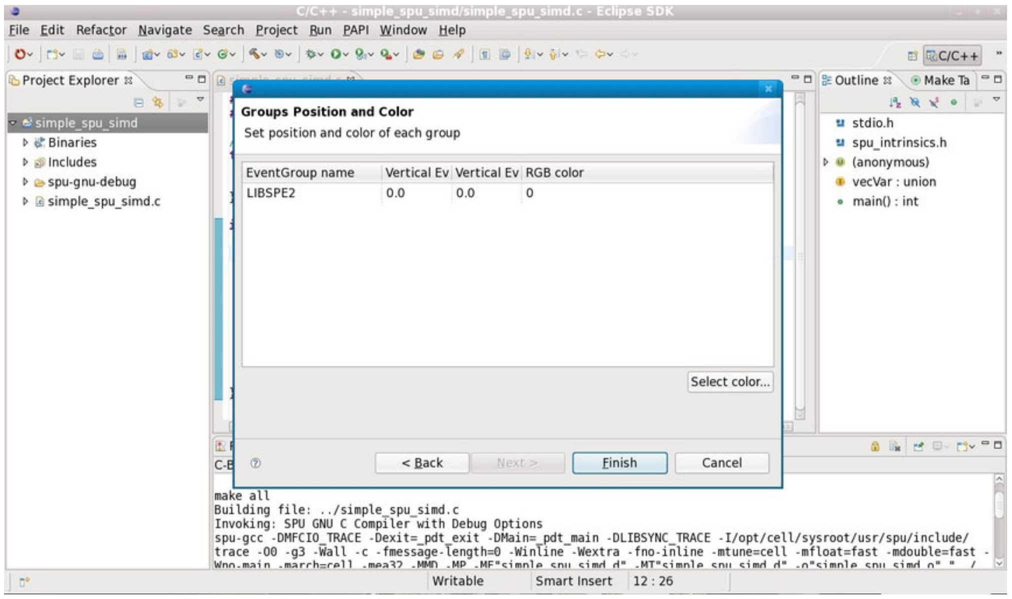
☑ Enable PPE Profile

☑ Enable SPE Profile

```
**** Build of configuration spu-gnu-debug for project simple_spu_simd ****

make all
make: Nothing to be done for `all'.
```

IDE PDT plugin - set colors for events

In this last step, you select different colors for the events groups, which you chose on previous steps. This information is part of the PDT's XML configuration file and is used to generate PDT trace files. Click **Finish** to create the XML configuration file.



Groups Position and Color

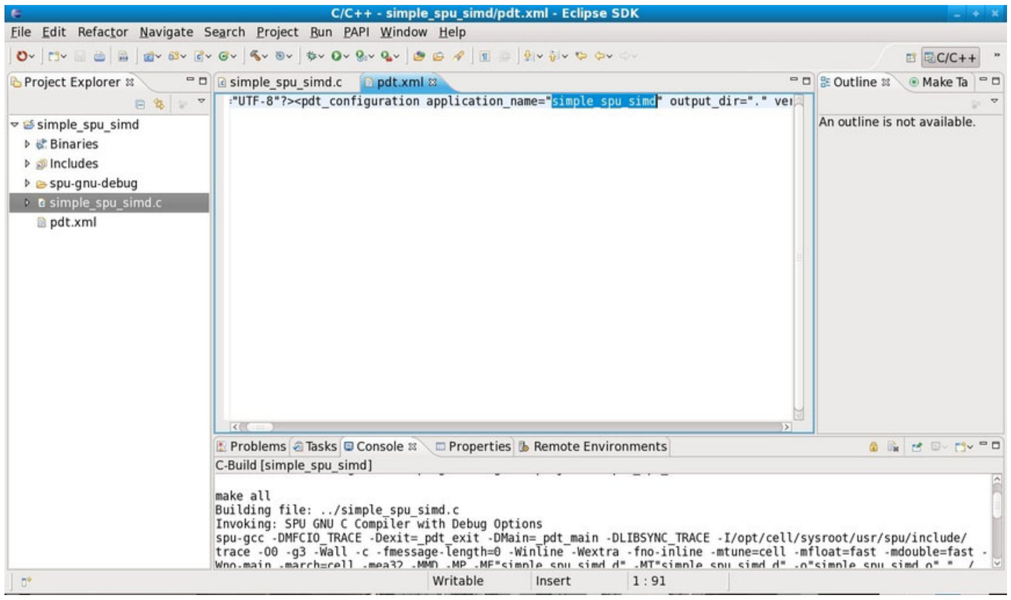
Set position and color of each group

EventGroup name	Vertical Ev	Vertical Ev	RGB color
LIBSPE2	0.0	0.0	0

```
make all
Building file: ../simple_spu_simd.c
Invoking: SPU GNU C Compiler with Debug Options
spu-gcc -DMFCIO_TRACE -Dexit=_pdt_exit -DMain=_pdt_main -DLIBSYNC_TRACE -I/opt/cell/sysroot/usr/spu/include/
trace -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -fno-inline -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -march=cell -mea32 -MMD -MP -MF"simple_spu_simd.d" -MT"simple_spu_simd.d" -o"simple_spu_simd.o" "../
```

IDE PDT plugin - modify the XML configuration file

1. Open the created XML file and set the **application_name** attribute to match the binary that will be used with PDT.
2. Save the files.
3. Press F5 to refresh the workbench.



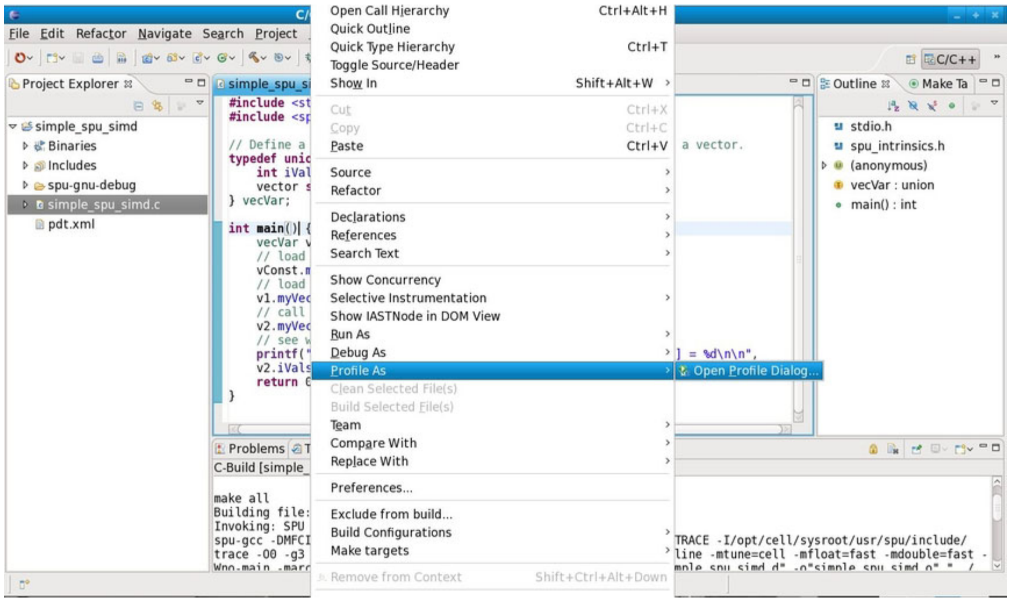
```
:"UTF-8"?><pdt_configuration application_name="simple_spu_simd" output_dir="." ver
```

```
make all
Building file: ../simple_spu_simd.c
Invoking: SPU GNU C Compiler with Debug Options
spu-gcc -DMFCIO_TRACE -Dexit=_pdt_exit -DMain=_pdt_main -DLIBSYNC_TRACE -I/opt/cell/sysroot/usr/spu/include/
trace -O0 -g3 -Wall -c -fmessage-length=0 -Winline -Wextra -fno-inline -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -march=cell -mea32 -MMD -MP -MF"simple_spu_simd.d" -MT"simple_spu_simd.d" -o"simple_spu_simd.o" "../
```

IDE PDT plugin - using the Profile Launcher

You can now launch our application with PDT support. To do this:

1. Right-click the editor with the C file that will generate the executable.
2. Select **Open Profile Dialog**.



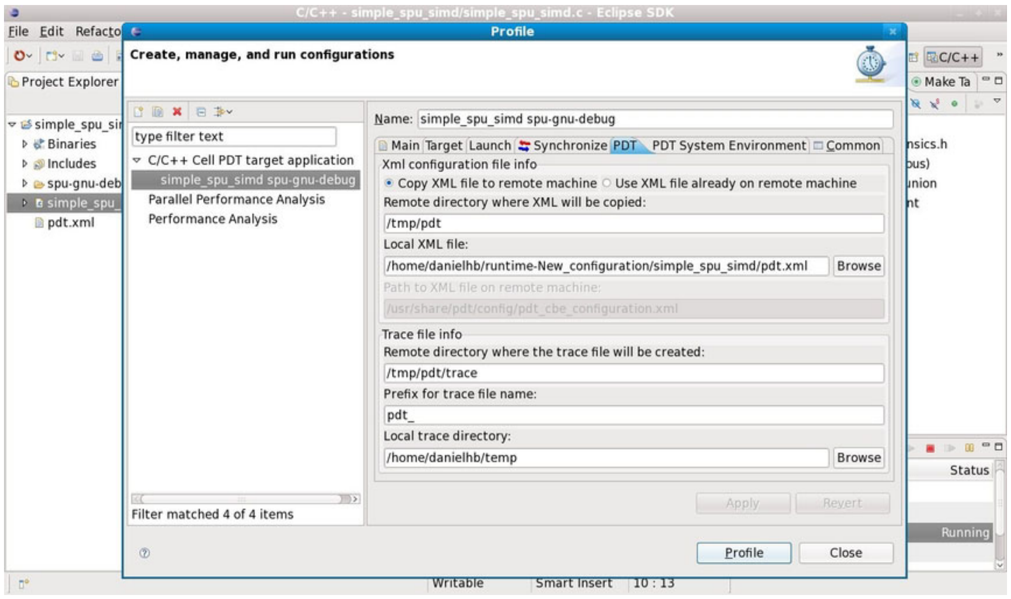
```
#include <st
#include <sp

// Define a                          a vector.
typedef unic
    int iVal
    vector s
} vecVar;

int main()| {
    vecVar v
    // load
    vConst.m
    // load
    v1.myVec
    // call
    v2.myVec
    // see w
    printf("                         ] = %d\n\n",
    v2.iVals
    return 0
}
```

```
make all
Building file:
Invoking: SPU
spu-gcc -DMFCI                       TRACE -I/opt/cell/sysroot/usr/spu/include/
trace -O0 -g3                        line -mtune=cell -mfloat=fast -mdouble=fast -
Wno-main -marc                       mple_spu_simd.d" -o"simple_spu_simd.o" "../
```

IDE PDT plugin - launch the application with PDT support

1. Double left-click **C/C++ Cell PDT target application** to create a new launch configuration
2. In the **PDT** tab, select the option **Copy XML file to remote machine**. You need to configure:
   - Remote directory to copy the XML file - must be a valid directory
   - Local XML file. Select the configuration file you created on previous steps
   - Remote directory to create the trace files - must be a valid directory
   - Prefix for trace file name - any valid ASCII string
   - Local trace directory, where the generated trace files will be downloaded from the remote environment - must be a valid directory

Click **Profile** to run the application with PDT support.


C/C++ - simple_spu_simd/simple_spu_simd.c - Eclipse SDK
Profile
Create, manage, and run configurations
type filter text
C/C++ Cell PDT target application
simple_spu_simd spu-gnu-debug
Parallel Performance Analysis
Performance Analysis
Filter matched 4 of 4 items
Main
Target
Launch
Synchronize
PDT
PDT System Environment
Common
Browse
Apply
Revert
Profile
Close

Name: simple_spu_simd spu-gnu-debug

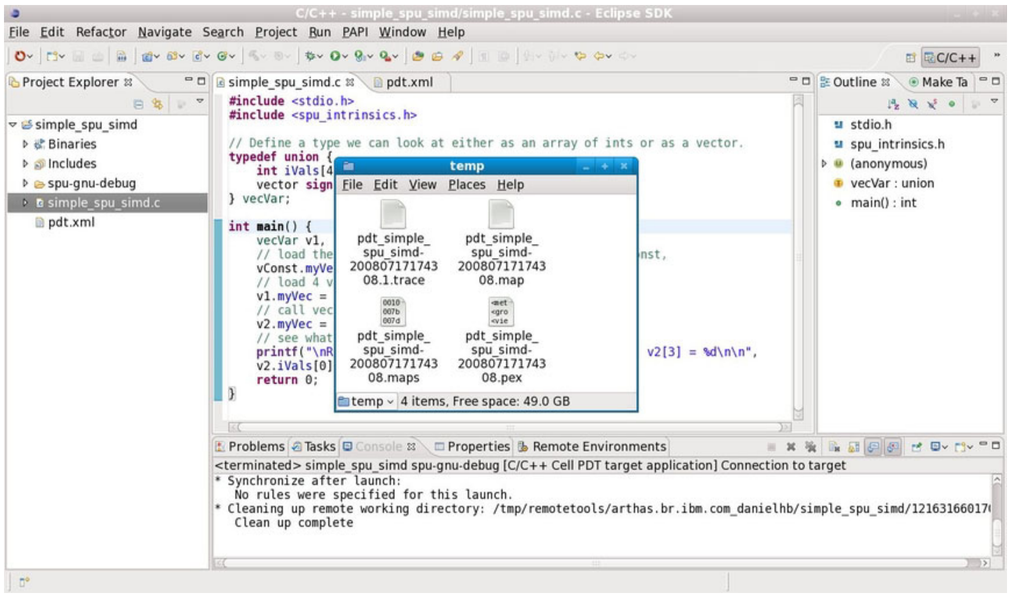
Xml configuration file info
● Copy XML file to remote machine ○ Use XML file already on remote machine
Remote directory where XML will be copied: /tmp/pdt
Local XML file: /home/danielhb/runtime-New_configuration/simple_spu_simd/pdt.xml
Path to XML file on remote machine: /usr/share/pdt/config/pdt_cbe_configuration.xml

Trace file info
Remote directory where the trace file will be created: /tmp/pdt/trace
Prefix for trace file name: pdt_
Local trace directory: /home/danielhb/temp

IDE PDT plugin - check generated trace files

After the launch finishes, you can check the results of our profiling by opening the local directory set to receive the trace files generated.

These trace files can be viewed with the Eclipse-based VPA tool using the Trace Analyzer perspective, for example.


C/C++ - simple_spu_simd/simple_spu_simd.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run PAPI Window Help
C/C++
Project Explorer
simple_spu_simd
Binaries
Includes
spu-gnu-debug
simple_spu_simd.c
pdt.xml
simple_spu_simd.c
pdt.xml
Outline
Make Ta
stdio.h
spu_intrinsics.h
(anonymous)
vecVar : union
main() : int
temp
File Edit View Places Help
pdt_simple_spu_simd-20080717174308.1.trace
pdt_simple_spu_simd-20080717174308.map
pdt_simple_spu_simd-20080717174308.maps
pdt_simple_spu_simd-20080717174308.pex
temp
4 items, Free space: 49.0 GB
Problems
Tasks
Console
Properties
Remote Environments

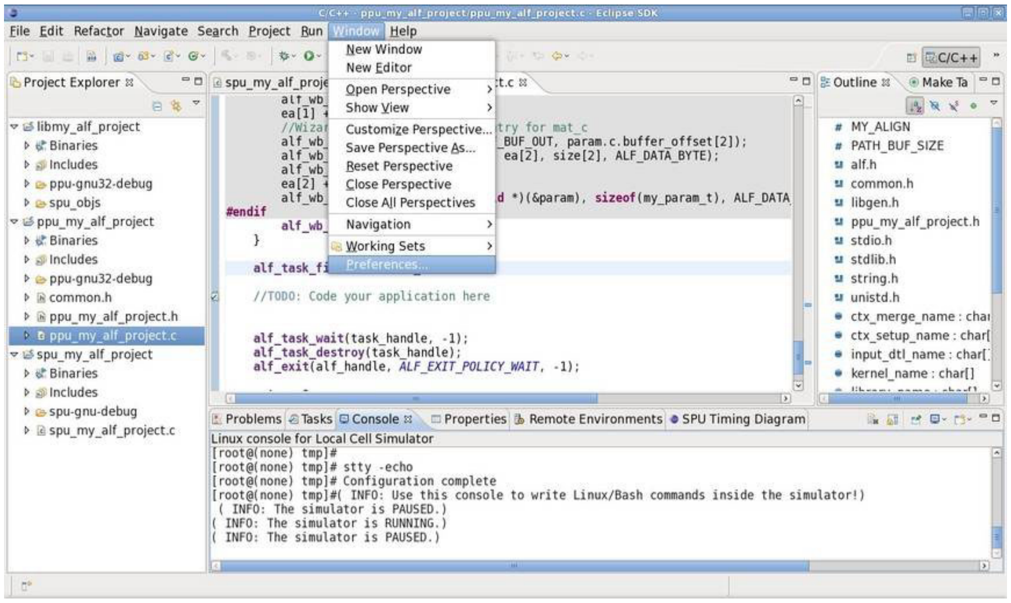
```
#include <stdio.h>
#include <spu_intrinsics.h>

// Define a type we can look at either as an array of ints or as a vector.
typedef union {
    int iVals[4
    vector sign
} vecVar;

int main() {
    vecVar v1,
    // load the                                                   nst,
    vConst.myVe
    // load 4 v
    v1.myVec =
    // call vec
    v2.myVec =
    // see what
    printf("\nR                                                   v2[3] = %d\n\n",
    v2.iVals[0]
    return 0;
}
```

```
<terminated> simple_spu_simd spu-gnu-debug [C/C++ Cell PDT target application] Connection to target
* Synchronize after launch:
  No rules were specified for this launch.
* Cleaning up remote working directory: /tmp/remotetools/arthas.br.ibm.com_danielhb/simple_spu_simd/12163166017
  Clean up complete
```

View the Eclipse preferences window

You can view and modify numerous cell environment settings in the **Preferences** window.

Click **Window → Preferences...**

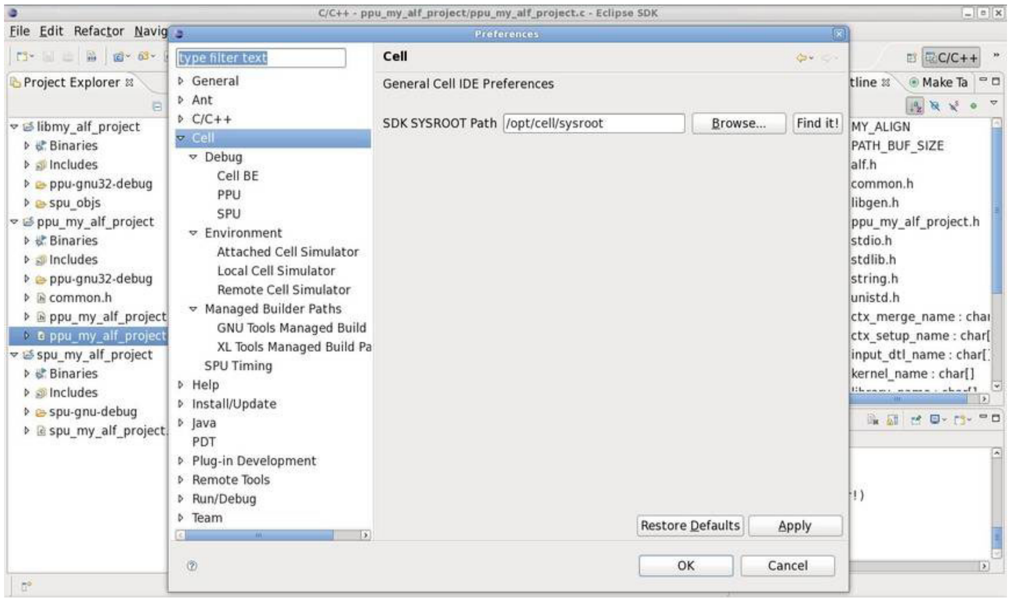


```
    //TODO: Code your application here


    alf_task_wait(task_handle, -1);
    alf_task_destroy(task_handle);
    alf_exit(alf_handle, ALF_EXIT_POLICY_WAIT, -1);
```

```
Linux console for Local Cell Simulator
[root@(none) tmp]#
[root@(none) tmp]# stty -echo
[root@(none) tmp]# Configuration complete
[root@(none) tmp]#( INFO: Use this console to write Linux/Bash commands inside the simulator!)
 ( INFO: The simulator is PAUSED.)
( INFO: The simulator is RUNNING.)
( INFO: The simulator is PAUSED.)
```

View the IDE environment preferences

On the left side, expand the **Cell** category. Here you can change many different options regarding IDE including **Debug** options, **Cell Environment** settings, and **SPU Timing** binary location.



SDK SYSROOT Path /opt/cell/sysroot

Tutorial finished!

This concludes the IDE tutorial!

Part 4. ALF for IDE programmer's guide

The following topics describe how to use ALF distribution template provided by IDE.

Chapter 12. ALF for IDE overview

The ALF API provides a set of functions to solve a set of parallel problems on multi-core memory hierarchy systems. This implementation of this API is focused on data parallel problems on a host-accelerator type hybrid system. ALF offers an interface to write data parallel applications without requiring architecturally dependent code. Features of ALF include data transfer management, parallel task management, double buffering, and data partitioning.

The ALF data distribution template provided by IDE, is designed to support data parallel programming style for solving data parallel problem for arrays. Similar to High Performance FORTRAN (HPF), the ALF data distribution template supports three data distribution models, which are:

- *
- CYCLIC
- BLOCK

The data distributions are presented as a serial of data transfer list entries. Data distributions provided by ALF data distribution template are for arrays of one, two, and three dimensions.

The data distributions are applied to two types of data, which are input buffer and output buffer.

Chapter 13. Data distribution overview

ALF IDE introduces data distribution directives to provide you with control over locality and data partition.

A DISTRIBUTE directive is used to indicate how data is to be partitioned among continuous memories. It specifies, for each dimension of an array, a mapping of array indices to accelerated nodes. Each dimension of an array may be distributed in one of three ways:

- * : No distribution
- BLOCK(n) : Block distribution (default: n= N/P), in which N is the size of array and P is the number of accelerated node.
- CYCLIC(n) : Cyclic distribution

BLOCK distribution

Let N be the number of elements in an array dimension, and let P be the number of accelerated nodes assigned to that dimension. Then, as illustrated in Figure 3,Figure 4, and Figure 5 on page 126, a BLOCK distribution divides the indices in a dimension into contiguous, equal-sized blocks of size N/P.

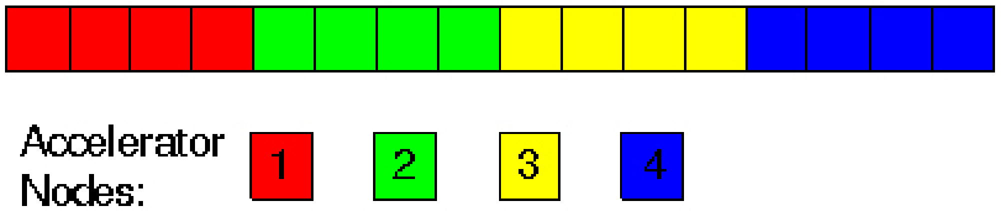


Figure 3. A 1D array BLOCK distribution for four accelerator nodes

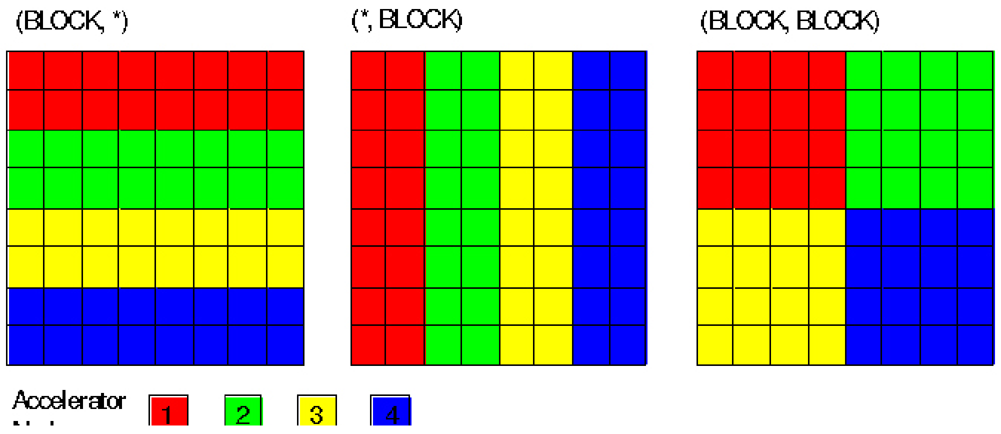


Figure 4. 2D array BLOCK distributions for four accelerator nodes

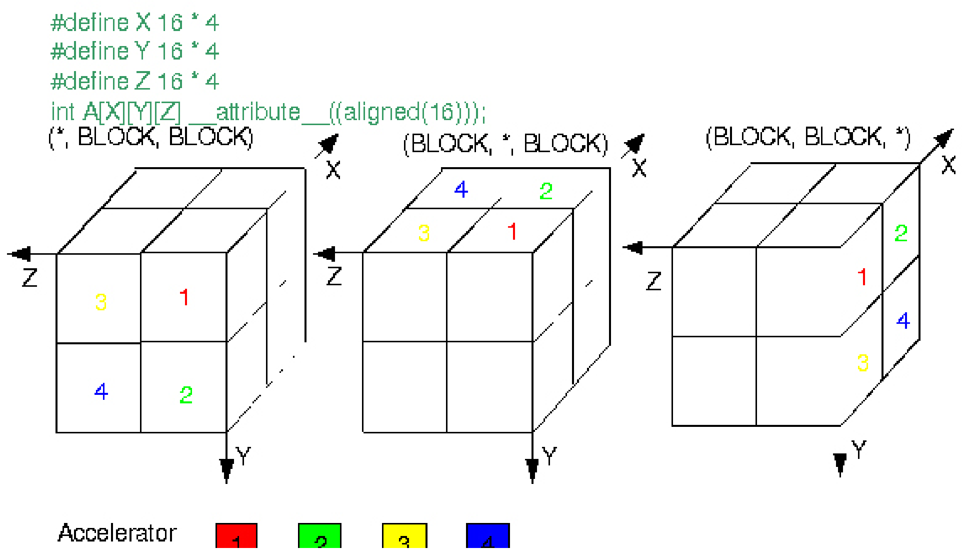


Figure 5. 3D array BLOCK distributions for four accelerator nodes

BLOCK(m) implies that after giving each accelerator node the array should have been distributed; if there are any elements left over an error has occurred. It implies that each accelerator node must get at most one block of elements

CYCLIC distribution

Cyclic distribution distributes elements of an array to processors in a round robin fashion. If an array, A has elements N and is mapped onto P accelerator nodes, each node gets (a maximum) total of ┌N / P┐ separate elements. See for and as examples***WRITER COMMENT; THIS SENTENCE DOES NOT MAKE SENSE. 3D CYCLIC distribution is inefficient and confusing and is therefore NOT supported in the ALF data distribution template.

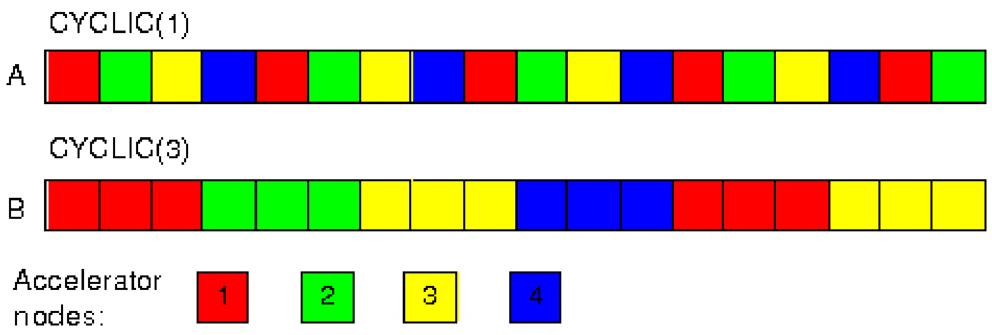


Figure 6. 1D array CYCLIC distributions for four accelerator nodes

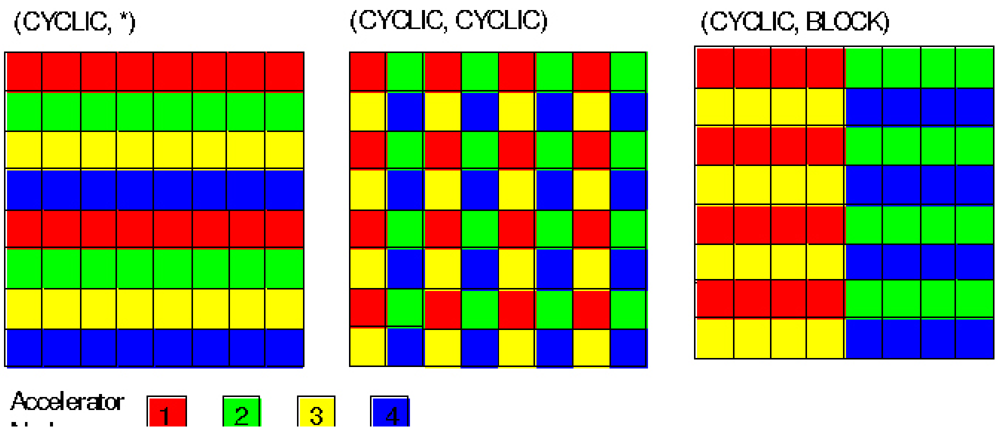


Figure 7. 2D array CYCLIC distributions for four accelerator nodes

There are some limitation for CYCLIC distribution, they are:

- The memory address of each CYCLIC should be 16 Byte aligned.
- The size (Bytes) of each CYCLIC must be 1, 2, 4, 8 and integral times of 16 (instance: 1, 2, 4, 8, 16, 32 ...). Refer to the API definition descriptions for further information.

CYCLIC(m) retains characteristics of both BLOCK and CYCLIC distributions: in theory, blocks of m elements are grouped together which is useful for neighborhood calculations and the cyclic distribution policy should promote a reasonable degree of load balancing.

Practice with data distribution directives

The following topics describe how to work with data distribution directives:

- "A matrix addition example"
- A simple solution to solve the problem
- "Data layout and partition scheme" on page 128

A matrix addition example

This is a simple application that adds two 2D matrixes and stores the result to a third matrix. For a 2D matrix addition, the mathematical definition is as follows:

C = A + B, where:

$c_{ij} = a_{ij} + b_{ij}$

or

$$\begin{vmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{vmatrix} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} + \begin{vmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{vmatrix}$$

Understanding the problem

Simple solution

In the following analysis, assume the data to be a 1024 x 512 single precision floating point matrix. The following is a sample of C code that solves the problem:

```
float mat_a[1024][512];
float mat_b[1024][512];
float mat_c[1024][512];

int main(void)
{
   int i,j;

   for (i=0; i<1024; i++)
   for (j=0; j<512; j++)

   mat_c[i][j]= mat_a[i][j] + mat_b[i][j];

   return 0;

}
```

The limitation with this simple approach is that it cannot run faster on a system with many accelerators that can process the $c_{i,j} = a_{i,j} + b_{i,j}$ in parallel. There are parallel programming languages and models that can speed up the program. The following section describe steps to rewrite this program with ALF APIs for parallel acceleration.

Potential solution for parallel speed increase

In general, most matrix mathematical operations can be decomposed into similar operations on many submatrices. The operations on these submatrices can be done in parallel if there are no dependencies between them. For example, divide the matrix into 128 submatrices each of which has 64 x 64 elements. Then the operation can be done on each 64 x 64 submatrix in parallel. In theory, the computation of the 1024 x 512 matrix addition can be completed in 1/128th of the time of the simple serialized code. The following examples use the submatrix approach.

Data layout and partition scheme

Two dimensional matrices are usually represented in 2D arrays in C. The actual memory layout of the 2D array in C is in 1D arrays concatenated by the second/column index, as shown by Figure 8.

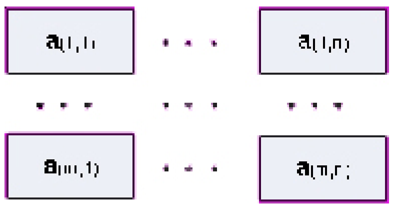
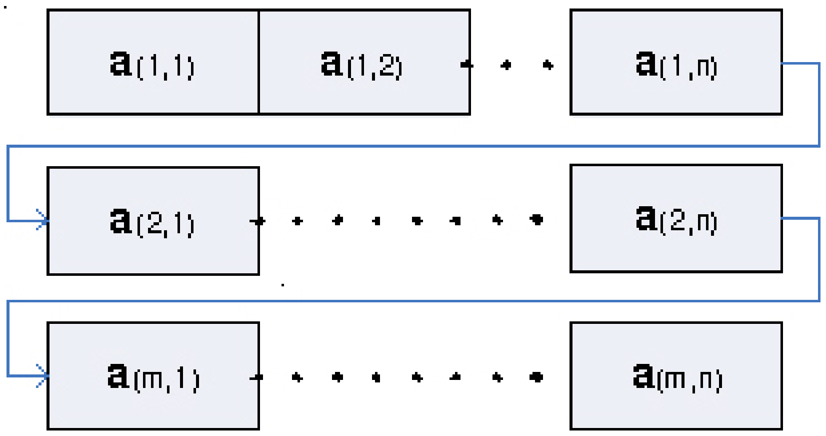


Figure 8. Memory organization of a 2D array "a[m][n]"



In the matrix addition example from the previous section, the submatrices were the basic unit of data. In the above C matrix data structure, a submatrix is part of the whole array as shown by Figure 9 and Figure 10 on page 130. This provides the following options to choose from:

- **Partition Scheme A**: With this partition scheme, the submatrices are a part of the whole column or row of the matrix. Or more formally, one of the submatrices of `a[m][n]` is defined as `sa[h][v]` where the `h < m` and `v <= n`. This is illustrated by the Figure 9.
- **Partition Scheme B**: With this partition scheme, we define the submatrices as a set of adjacent full length rows of the matrix. One of the submatrices of `a[m][n]` is defined as "`sa[h][v]`" where the `h < m` and `v == n`. See Figure 10 on page 130

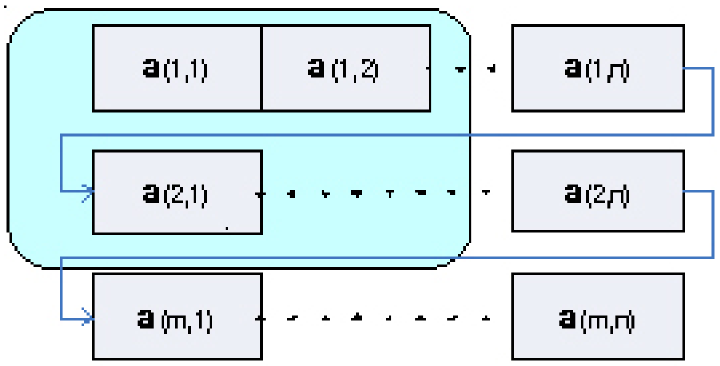


Figure 9. Partition scheme A: Data partition of 2D sub matrix

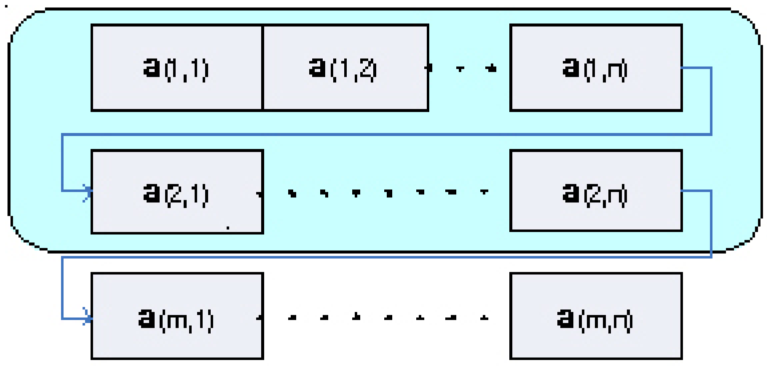


Figure 10. Partition Scheme B: Data partition of 2D sub matrix

Figure 7 on page 127 shows the differences between the two schemes. The data of the submatrix in scheme A is collected from disjointed segments in the data buffer of the matrix. For scheme B, the submatrix is from one contiguous segment of the matrix. Mathematically this makes no significant difference, but the data movement in our matrix addition example is significantly more complex in scheme A than in scheme B, as can be seen by the following example pseudo code with data distribution directives.

```
DECLARE ACCELERATE_NODES = 6;

float mat_a[1024][512];
float mat_b[1024][512];
float mat_c[1024][512];

CYCLIC(512,8) ONTO mat_a
CYCLIC(512,8) ONTO mat_b
CYCLIC(512,8) ONTO mat_c

for (i=0; i<512; ++i)
for (j=0; j<8; ++j)

mat_c[i][j]= mat_a[i][j] + mat_b[i][j];
```

Based on the above analysis, scheme B is preferred in this matrix addition example. Remember that this situation can change in some real world scenarios where large contiguous data movement might not be supported.

Chapter 14. ALF IDE wizard overview

This section describes the ALF IDE wizard.

It covers the following topics:
- "Basic structure of an ALF application"
- "What does the ALF IDE wizard do?"

Basic structure of an ALF application

The basic structure of an ALF application is shown in Figure 11. In general the flow is linear.

On the host side, you first initialize the ALF runtime and then create a compute task. After you have created the task, you add work blocks to the work queue of the task. Then, you wait for the task to complete and shut down the ALF runtime to free up allocated resources.

On the accelerator side, after an instance of the task is spawned, it waits for pending work blocks to be added to the work queue. Then the **alf_comp_kernel** is called for each work block. If the partition location attribute of a task is partitioned in accelerated node, **alf_accel_intput_list_prepare** is called before the compute kernel is invoked and **alf_accel_output_list_prepare** is called after the compute kernel exits.

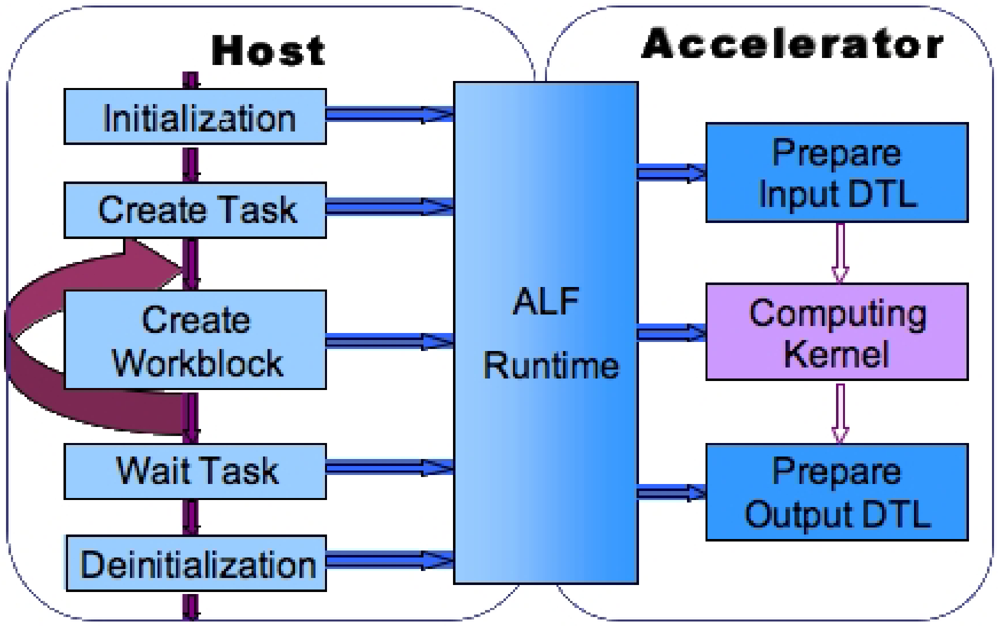


Figure 11. ALF application structure and execution flow

What does the ALF IDE wizard do?

The ALF IDE wizard aims to release you from trivial ALF programming tasks, and to help you focus on your problem logic.

The ALF IDE wizard automatically generates source code for host and accelerator code. In most cases, you only needs to add code for computing kernel and data initialization.

The ALF IDE wizard does several tasks works automatically, as described in the following figure.

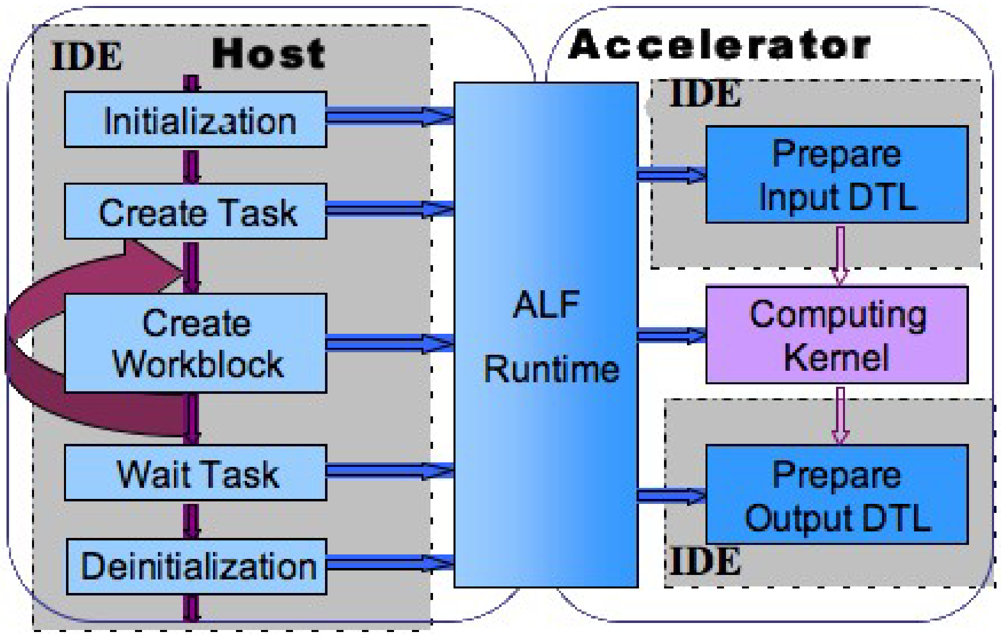


Figure 12. Tasks performed by the ALF IDE wizard

Figure 12 shows that the ALF IDE wizard reduces some trivial and routine tasks, such as initialization, and preparation of the input/output transfer list. The ALF IDE wizard also generates the correct Makefiles for you.

The ALF IDE Wizard does not automatically generate source code for the computing kernel and data initialization tasks. You need to generate code for your computing kernel and data initialization.

IDE also cannot handle the data transfer list for irregular data movement. In this case, it is suggested that you use the initialization and skeleton template for the ALF project to generate an empty ALF project, and code your own application.

If you need to add any other libraries or headers into application, you need to revise the project's properties to add these dependencies.

Chapter 15. Programming tips using the ALF IDE wizard

The following topics provide tips about how to program using the ALF IDE wizard.

Data layout and partition scheme

Data partition is crucial to the ALF programming model. Incorrect data partitioning and data layout design either prevents ALF from being applicable or results in degraded performance. For the best performance, use the following guidelines for data partitioning and data layout design.

Note: Data partition design and layout can be platform-dependent, so the best design for one architecture might not perform well on another.

Data partition and layout is closely coupled with compute kernel design and implementation, so you should them considered simultaneously.

- **Use the correct data partition size**

  Often, the accelerator local memory or data cache is very limited. A significant performance penalty can occur if the partitioned data cannot fit into this memory limitation. For example, on Cell/B.E. architecture, if a single block of partitioned data is larger than 128 KB, it might not be possible to support double buffering on the SPE side. This can result in up to 50% performance loss.
- **Minimize the number of data movements**.

  A large amount of data movement can cause performance loss in applications. Improve performance by avoiding unnecessary data movements. Use the correct data distribution size to improve the performance.
- **Know address alignment limitations on specific platforms**

  If data is stored on aligned addresses, the processor can access it faster than data on unaligned addresses. On platforms such as Cell/B.E., the data movement from the local memory of the SPE can only be based on 16 byte aligned addresses.

For more detail about data partitioning restrictions, refer to “Data partitioning limitations” on page 143.

ALF host program and data transfer lists

One important decision for data transfer list is to choose whether to use accelerator side data transfer list generation. Because there might be a large number of accelerators used in one compute task if the data transfer list is somewhat complex, the host might not be able to generate work blocks faster than the accelerators can process. In this case, you can supply the data needed for data transfer list generation in the parameters of the work block and use the accelerators to generate the data transfer lists based on these parameters.

You can also start a new task for queuing while another task is running. You can then prepare the work blocks for the new task in advance before the task actually runs.

ALF IDE wizard generates data transfer lists both in host and in accelerator simultaneously. You can use the macro ACCELERATOR_PARTITION defined in common.h to switch between them.

Chapter 16. Programming with ALF IDE wizard

This section describes how to program with the ALF IDE wizard.

It covers the following topics:

Using the ALF IDE wizard to create an ALF project

This topic describes what you need to do to create an ALF project.

You need to follow the following steps to generate an ALF project.

1. Create an ALF project and give the project a name.
2. Specify the expected stack size.
3. Select the expected number of accelerators (1, 2, 16, and 0: for all available).
4. Select the partition method (host or accelerator).
5. Describe the data distribution model for each buffer. For each new buffer input, performing the following steps:
   a. Enter the variable name, element type (the type for each element in the buffer), and basic unit (an ALF predefined type or the same as element type) of the element
   b. Specify the buffer type (INPUT or OUTPUT)
   c. Specify the dimension and size (1, ... dimension) for each dimensions of the array
   d. Specify the distribution model for this array (CYCLIC or BLOCK), and parameters for this distribution model
6. Finish. The ALF IDE wizard generates code according to your input.
7. Revise the generated code as required.

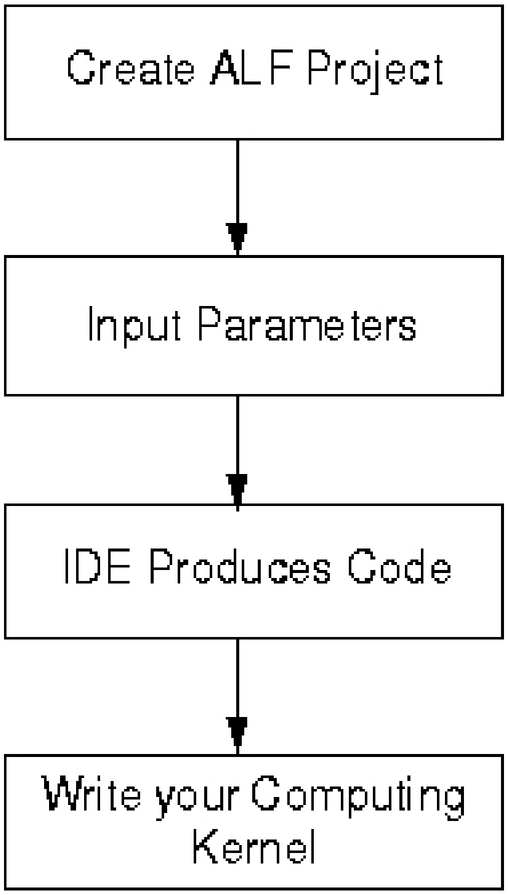


Figure 13. Creating ALF project

ALF IDE wizard parameters

The following table lists the user input parameters.

Table 2. Basic user input parameters

Token	Description	Data range
`$PROJECT_NAME`	Project name (Step 1)	Strings
`$STACK_SIZE`	Expected stack size (Step 2)	From 10 Bytes to 246 KBytes
`$EXP_ACCEL_NUM`	Expected number of accelerator (Step 3)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, and 0 for all available. **Note:** All available option is invalid when any buffer's data distribution model is BLOCK
`$PARTITION_METHOD`	Partition model, host, or accelerator (Step 4)	0 : Partition by host 1 : Partition by accelerator
`$VARIABLE_NAME`	Name of the data array (Step 5a)	Valid variable name in C language, it is case-sensitive. Different buffer should have different variable name.
`$ELEMENT_TYPE`	Type of element for the data array indicated by `$VARIABLE_NAME` (Step 5a)	C language built in type or user defined data type.

Table 2. Basic user input parameters (continued)

Token	Description	Data range

`$ELEMENT_UNIT`	Array data type. (Step 5a)	Basic data types. The choices are:
		• ALF data types. There are 6 types:
		`ALF_DATA_INT16`
		`ALF_DATA_INT32`
		`ALF_DATA_INT64`
		`ALF_DATA_FLOAT`
		`ALF_DATA_DOUBLE`
		`ALF_DATA_BYTE`
		• As defined by `$ELEMENT_TYPE`

`$BUFFER_TYPE`	Buffer type of the data array indicated by `$VARIABLE_NAME`. Input buffer or output buffer. (Step 5b)	`ALF_BUFFER_INPUT`: input buffer `ALF_BUFFER_OUTPUT`: output buffer
`$NUM_DIMENSION`	Number of dimension for array indicated by `$VARIABLE_NAME`. (Step 5c)	1, 2, or 3
`$DIMENSION_SIZE_X`	X dimension size for array indicated by `$VARIABLE_NAME`. For a 3D dimension array A, its dimensions represented as A[Z][Y][X] For a 2D dimension array B, its dimensions represented as B[Y][X] For a 1D dimension array C, its dimensions represented as C[X](Step 5c)	1, ...
`$DIMENSION_SIZE_Y`	Y dimension size for array indicated by `$VARIABLE_NAME`. Only available when the array dimension is 2 or 3. For a 3D dimension array A, its dimensions represented as A[Z][Y][X] For a 2D dimension array B, its dimensions represented as B[Y][X](Step 5c)	1, ... 0 when the dimension of array is less than 2.
`$DIMENSION_SIZE_Z`	Z dimension size for array indicated by `$VARIABLE_NAME`. Only available when the array dimension is 3. For a 3D dimension array A, its dimensions represented as A[Z][Y][X].(Step 5c)	1, ... 0 when the dimension of array is less than 3.
`$DISTRIBUTION_MODEL_X`	The distribution model for X dimension of data array which indicated by $VARIABLE_ (Step 5d)	BLOCK, CYCLIC, or *

Table 2. Basic user input parameters (continued)

Token	Description	Data range

`$DISTRIBUTION_SIZE_X`	The distribution size for X dimension distribution (indicated by `$VARIABLE_NAME`) for data array (indicated by `$VARIABLE_NAME`). (Step 5d)	CYCLIC: User input integer number, and limited into (1 - `$DIMENSION_SIZE_X`). For BLOCK distribution, it is calculated by IDE as:
		
		`$DISTRIBUTION_SIZE_X = $DIMENSION_SIZE_X / $EXP_ACCEL_NUM`
		`*: $DISTRIBUTION_SIZE_X = $DIMENSION_SIZE_X`

`$DISTRIBUTION_MODEL_Y`	The distribution model for Y dimension of data array which indicated by `$VARIABLE_NAME` Only available when the dimension of data array is 2 or 3.	BLOCK, CYCLIC, or Should not appear in the parameter file when the dimension of array is less than 2.

`$DISTRIBUTION_SIZE_Y`	The distribution size for Y dimension distribution (indicated by `$DISTRIBUTION_MODEL_Y`) for data array (indicated by `$VARIABLE_NAME`).
	
	Only available when the dimension of data array is 2 or 3.(Step 5d)
		CYCLIC: User input integer number, and limited into (1 - `$DIMENSION_SIZE_Y`). For BLOCK distribution, it is calculated by IDE as:
		
		`$DISTRIBUTION_SIZE_Y = $DIMENSION_SIZE_Y / $EXP_ACCEL_NUM`
		`*: $DISTRIBUTION_SIZE_Y = $DIMENSION_SIZE_Y`
		
		Should not appear in the parameter file when the dimension of array is less than 2.

`$DISTRIBUTION_MODEL_Z`	The distribution model for Z dimension of data array which indicated by `$VARIABLE_NAME` Only available when the dimension of data array is 3.(Step 5d)	BLOCK or * Should not appear in the parameter file when the dimension of array is less than 3.

`$DISTRIBUTION_SIZE_Z`	The distribution size for Z dimension distribution (indicated by `$DISTRIBUTION_MODEL_Z`) for data array (indicated by `$VARIABLE_NAME`).
	
	Only available when the dimension of data array is 3.(Step 5d)
		BLOCK:
		
		`$DISTRIBUTION_SIZE_Z = $DIMENSION_SIZE_Z / $EXP_ACCEL_NUM`
		`*: $DISTRIBUTION_SIZE_Z = $DIMENSION_SIZE_Z`
		
		Should not appear in the parameter file when the dimension of array is less than 3.

Data partition limitations

There are some limitations for data partitions.

The partition is eventually turned into a serial of data transfer entry. The number of these entries for each buffer after partitioning should be equal. The following examples illustrate some valid and invalid data distributions for arrays: A[1024][512], B[512][128].

- (CYCLIC(512), CYCLIC(16)) ONTO A, and (CYCLIC(128), CYCLIC(8) ONTO B is valid, because the number of DT entries for these two distributions are equal. (For A, the number of DT entry is (1024/512) * (512 / 16), and for B, the number of DT entry is (512 / 128) * (128/6))
- (*, CYCLIC(16)) ONTO A, and (CYCLIC(128), CYCLIC(16)) ONTO B is valid, because the numbers of DT entry for these two distributions are equal, it is 32.

- (CYCLIC(512), CYCLIC(16)) ONTO A, and (CYCLIC(128), CYCLIC(16)) is invalid, because the numbers of DT entry for these two distributions are not equal.

The data size of each data transfer entry must be a multiple of 16 Byte (16 Bytes, 32 Bytes, and so on).

The sum of total size of buffer and free stack size should not exceed the size of Local Storage (256 KBytes) for Cell/B.E. platform.

Using user-defined types

The ALF IDE wizard supports user-defined types. When your input is not a C-language built-in type, the ALF IDE wizard generates a user-defined type in common.h.

For example, when you input an array as `my_structure a[1024]`, a struct named `my_structure` is presented in common.h.

```
typedef
struct _my_structure_t
{
//TODO: Code your data structure here, make sure the size (Byte)
//of this structure is 1, 2, 4, 8 or power of 16.
char
pad[16];
}my_structure;
```

You can revise the structure generated by IDE as you require. You need to make sure that the size of the structure is 1, 2, 4, 8, or power of 16 (Byte) as this is a limitation of the Cell/B.E. platform.

The following types are treated as C programming language built-in types.

IDE supported C programming language built-in types

```
"char", "unsigned char", "signed char", "short", "unsigned short",
"int", "unsigned int","long", "signed", "unsigned","unsigned long",
"long long","unsigned long long", "float", "double", "long double",
"_Bool", "float _Complex", "double _Complex", "long double _Complex"
```

Example: Matrix addition

The following topics describe how to code your computing kernel.

Step-by-step instructions

Refer to the Chapter 11, “IDE tutorial,” on page 31 for a step by step walkthrough of how to use the ALF IDE wizard.

Coding your computing kernel

You must code your computing kernel and data initialization. In this step, you code the computing kernel in accelerated code.

By editing spu_matrix_add.c, enter the following code into **alf_accel_comp_kernel**. The computing kernel will look like the following example (user-defined code is marked as bold).

Computing kernel for matrix addition

```
int alf_accel_comp_kernel(void *p_task_context __attribute__
((unused)),void *p_parm_ctx_buffer,void *p_input_buffer,void
*p_output_buffer,void *p_inout_buffer __attribute__ ((unused)),unsigned
int current_count __attribute__ ((unused)),unsigned
int total_count __attribute__ ((unused)))
{
 float *mat_a;
 float *mat_b;
 float *mat_c;
 my_param_t *p_param = (my_param_t *) p_parm_ctx_buffer;
 mat_a = (float*) ((char*)p_input_buffer + p_param->c.buffer_offset[0]);
 mat_b = (float*) ((char*)p_input_buffer + p_param->c.buffer_offset[1]);
 mat_c = (float*) ((char*)p_output_buffer + p_param->c.buffer_offset[2]);
 //TODO: Code your computing kernel here
 int i;
 for(i=0; i<p_param->c.size[0]; ++i)
       {
        mat_c[i] = mat_a[i] + mat_b[i];
       }
 return 0;
}
```

We will also add a data initialization code in `ppu_matrix_add.c`, and call it in main (). As illustrated in the next sample and marked as bold.

Data preparation in PPU side

```
static void prepare_data()
{
int i, j;
for (i=0; i<1024; ++i)
for (j=0; j<512; ++j)
{
mat_a[i][j] = i * 2.0 + j;
mat_b[i][j] = i * 3.0 + j;
mat_c[i][j] = 0.0;}
}
...
int main()
{
prepare_data();
alf_init(NULL, &alf_handle);
if ((err = alf_query_system_info(alf_handle, ALF_QUERY_NUM_ACCEL, &nodes)) < 0)
{
fprintf(stderr, "Error in query system information\n");
return (-1);
}
...
}
```

Chapter 17. Platform-specific constraints for ALF IDE Wizard on Cell/B.E. architecture

The section describes the following platform-specific constraints:

- "Data distribution limitations"
- "Data partitioning limitations"
- "Local memory constraints for SPE" on page 144
- "User-defined type size limitations" on page 146

Data distribution limitations

This topic describes the limitations for the various ALF data distribution policies.

For BLOCK distribution, the expected accelerator number should be exact, `all available` is not allowed. This means if there has any buffer for which the distribution model is BLOCK, the option `all available` options cannot be selected.

CYCLIC distribution is only available when the dimension of an array is one or two.

Data partitioning limitations

For each array, the number of blocks after partitioning should be equal, as they are put together into the same ALF work block. Errors occur when the number of partitioned blocks is not equal.

The size of each block should be bound into 16 bytes (such as 16 byte, 32 byte, 48 byte, and so on). The following describes how to calculate the size of data block after partitioning.

- `data_transfer_size = sizeof($ELEMENT_TYPE) * $DISTRIBUTION_SIZE_X`, when the array dimension is one
- `data_transfer_size = sizeof($ELEMENT_TYPE) * $DISTRIBUTION_SIZE_X * $DISTRIBUTION_SIZE_Y`, when the array dimension is two
- `data_transfer_size = sizeof($ELEMENT_TYPE) * $DISTRIBUTION_SIZE_X * $DISTRIBUTION_SIZE_Y * $DISTRIBUTION_SIZE_Z`, when the array dimension is three

If `((data_stransfer_size % 16) != 0)`, an error occurs.

To sum up, for ALF forCell/B.E., be aware of the following data transfer list constraints:

- Data transfer information for a single working block can consist of up to eight data transfer lists for each direction transfer (local store to main memory and vice versa)
- Each data transfer list consists of up to 2048 data transfer entries
- Each entry can describe up to 16 KB of data transfer between the continuous area in main memory and local storage
- The local store area described by each entry within the same data transfer list must be contiguous

- The transfer size and effective address low 32 bits for each data transfer entry must be 16 bytes aligned

Local memory constraints for SPE

The size of local memory on the accelerator is 256 KB and is shared by code and data. Memory is not virtualized and is not protected. A typical memory map of an SPU program is given in Figure 14 on page 145. There is a runtime stack above the global data memory section. The stack grows from the higher address to the lower address until it reaches the global data section. Due to the limitation of programming languages and compiler/linker tools, you cannot predict the maximum stack usage when you develop the application and when the application is loaded. If the stack requires larger storage space than was allocated, there will be a stack overflow exception. The current approach is to dedicate all the free memory to the stack. New compiler tools also support the option to generate stack boundary check code according to the process given in the SPU ABI document. The SPU application will be shutdown when there is a stack overflow and a message is sent to the PPE.

ALF allocates the work block buffers directly from the memory region higher than the runtime stack. This is implemented by moving the stack pointer (or by pushing a large amount of data into the stack). For ALF, the larger the buffer is, the better it can optimize the performance of a task by using techniques such as double buffering. It is better to let ALF allocate as much as possible memory from the stack. Because of this, you must estimate the maximum stack usage information for the user code plus ALF and programming language runtime code. Estimate the stack usage and input the value into IDE. If the stack size is too small at runtime, a stack overflow occurs and it causes unexpected exceptions such as wrong results or a program crash. Be sure to leave enough local memory space for runtime stack usage in the early stage of your coding and data partition designs. Reduce the stack size to make more memory for ALF runtime usage only when the actual requirement can be estimated more accurately.

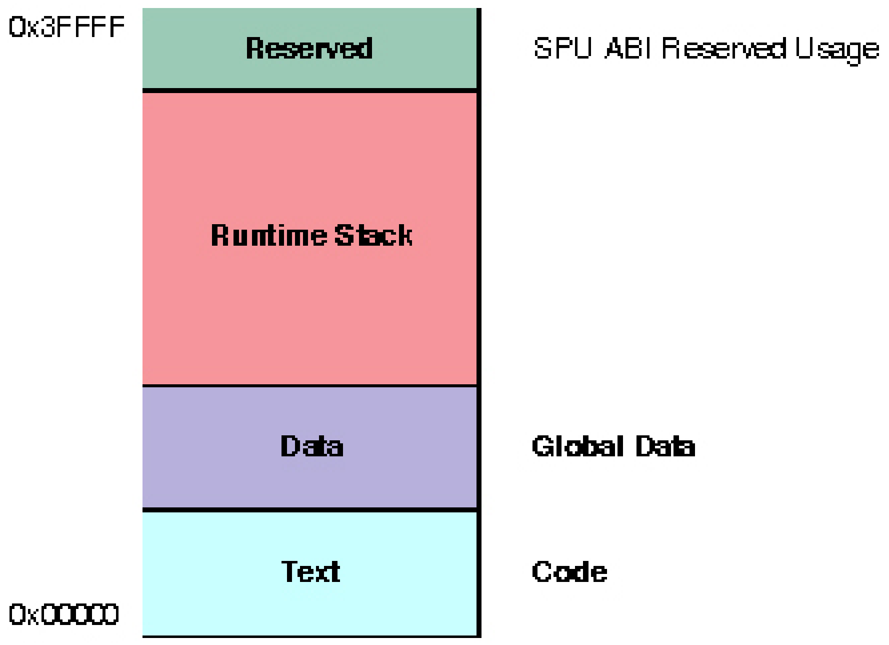


Figure 14. SPU local memory map - common Cell/B.E. application

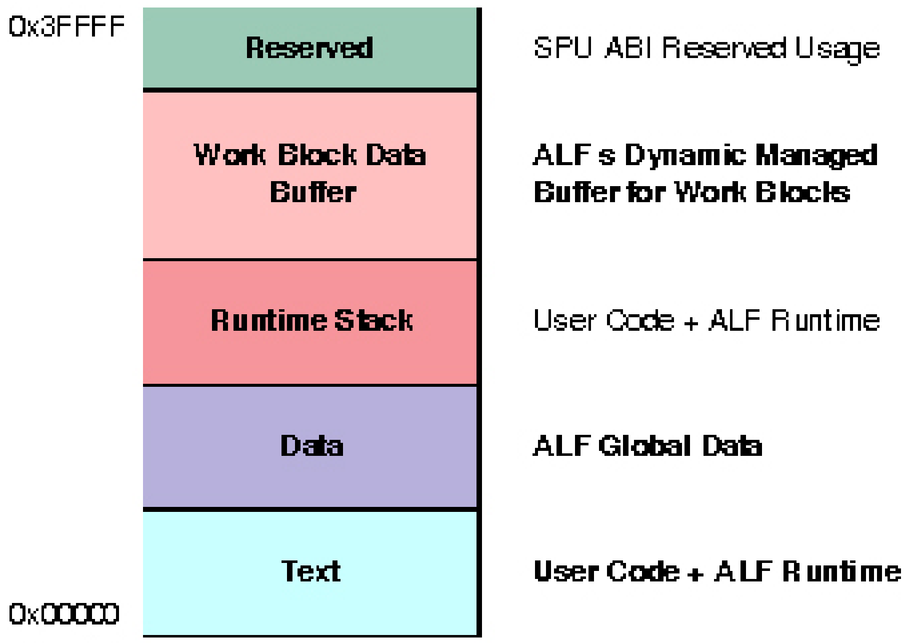


Figure 15. SPU local memory map - common ALF application

Because of SPE space limitations, the sum of data size partitioned into each accelerator node and the stack size should not exceed 246 KBytes (The SPE space is 256 KB). The data size for each accelerator can be calculated as follows (for the size of data transfer list, refer to "Data partitioning limitations" on page 143):

```
sum = 0;
for_each(IO_BUFFER)
{
    sum +=
    data_transfer_size;
}
sum +=
STACK_SIZE;
if (sum >246 * 1024) return error;
```

IDE reports an error when this limitation is exceeded.

User-defined type size limitations

The size of a user-defined type should be a multiple of 16 Bytes. IDE does not check this.

A tip for generating a 16 byte boundary type, is to declare data structure as follows, and the compiler will generate a data structure with a size that is bounded into 16 bytes:

User-defined type bounded into 16 bytes

```
typedef struct _my_type_t
{
   struct my_content_t
   {
    //Your parameters
   }c;
   unsigned char _pad[(sizeof(struct _my_type_t) + 15) & 0xFF10];
}
my_type_t;
```

Part 5. Appendixes

Appendix A. Related documentation

This topic helps you find related information.

Document location

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

`http://www.ibm.com/developerworks/power/cell/`

Click the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*

Programming

- *Cell Broadband Engine Programmer's Guide*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Programming Tutorial*

Library

- *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference*
- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Fast Fourier Transform Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*
- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Runtime Extensions Library Programmer's Guide and API Reference*

- *Three dimensional FFT Prototype Library Programmer's Guide and API Reference*

Installation

- *SDK for Multicore Acceleration Version 3.1 Installation Guide*

Tools

- *Getting Started - XL C/C++ for Multicore Acceleration for Linux*
- *Compiler Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Language Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Programming Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Installation Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Getting Started - XL Fortran for Multicore Acceleration for Linux*
- *Compiler Reference - XL Fortran for Multicore Acceleration for Linux*
- *Language Reference - XL Fortran for Multicore Acceleration for Linux*
- *Optimization and Programming Guide - XL Fortran for Multicore Acceleration for Linux*
- *Installation Guide - XL Fortran for Multicore Acceleration for Linux*
- *Performance Analysis with the IBM Full-System Simulator*
- *IBM Full-System Simulator User's Guide*
- *IBM Visual Performance Analyzer User's Guide*

IBM PowerPC® Base

- *IBM PowerPC Architecture™ Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *IBM PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

Appendix B. Accessibility features

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

The following list includes the major accessibility features:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are tactilely discernible and do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

IBM and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able/ for more information about the commitment that IBM has to accessibility.

Notices

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

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

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

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

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

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

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

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

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

Terms and conditions

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

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

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

Index

Printed in USA

SC34-2561-00