An introduction to the IDE for the Cell Broadband Engine SDK

Building a sample project with the Eclipse-based Cell BE development environment

Skill Level: Introductory

Sean Curry (seancurry.ut@gmail.com)
Software Engineer
IBM

02 Nov 2006

Updated 30 Mar 2007

This introductory tutorial, updated for the Cell Broadband Engine™ (Cell BE) SDK V2.1, explores the Interactive Development Environment (IDE) of the Cell BE processor and gives developers a click-for-click walk-through of building a simple project in this environment.

Section 1. Before you start

About this tutorial

This tutorial explains how to create, build, and run POWER™ Processing Unit (PPU) and Synergistic Processor Unit (SPU)-managed make projects. During this process, you will learn how to use some of the main features of the Cell IDE for Eclipse. First, we'll show you two managed make C projects, and you'll learn to configure these projects to utilize the Embed SPU tool. Next, you'll see how to use the combined debugger, static and dynamic performance analysis, and more.

This tutorial uses screen captures to show the steps described. Depending on your system configuration, the exact appearance of user interface elements may vary. Use the screen captures to follow along with the tutorial's progress.

Before starting this tutorial you should download and install the Cell BE SDK, Eclipse V3.2, the C/C++ Development Tools (CDT) V3.1 for Eclipse, and the Cell IDE; see Page 7 of this tutorial for links. This tutorial assumes only minimal familiarity with the Eclipse IDE, although familiarity with the IDE, and with the concepts of the Cell BE architecture, will be beneficial.

If you wish to know more about the Cell BE architecture, or about development for it, you might like reading some of these developerWorks articles and tutorials:

- "An introduction to compiling for the Cell Broadband Engine architecture"
- "Meet the experts: David Krolak on the Cell Broadband Engine EIB bus"
- "Cell Broadband Engine processor DMA engines"
- "Cell Broadband Engine Architecture and its first implementation"

And for a list of all Cell BE-related materials published to date, see the developerWorks Cell BE resource center.

Section 2. Creating the SPU project

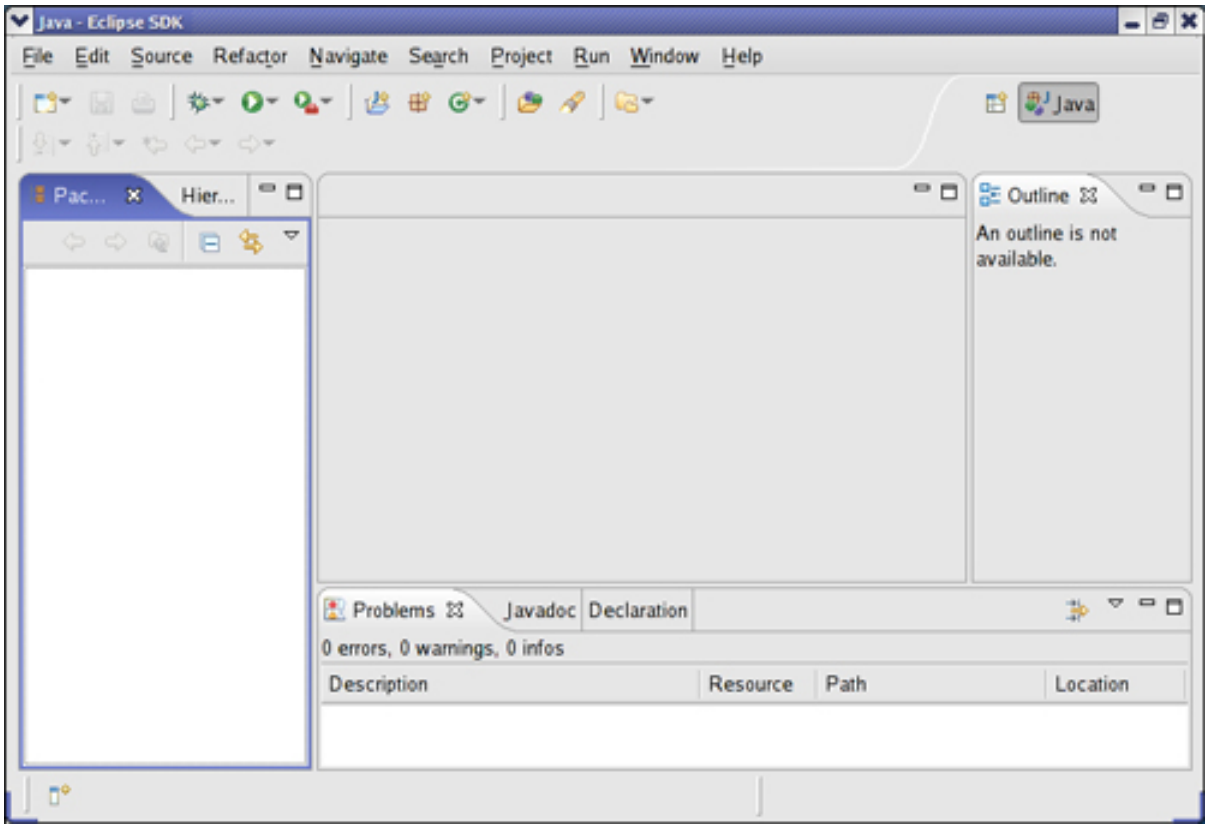
First things first

The first step in building the sample project is to create a project for the test program that will run on the SPU components of the Cell BE processor. This project creates one of the building blocks used by the next project.

Eclipse

Start by opening Eclipse. You should see a window similar to the screen capture below.

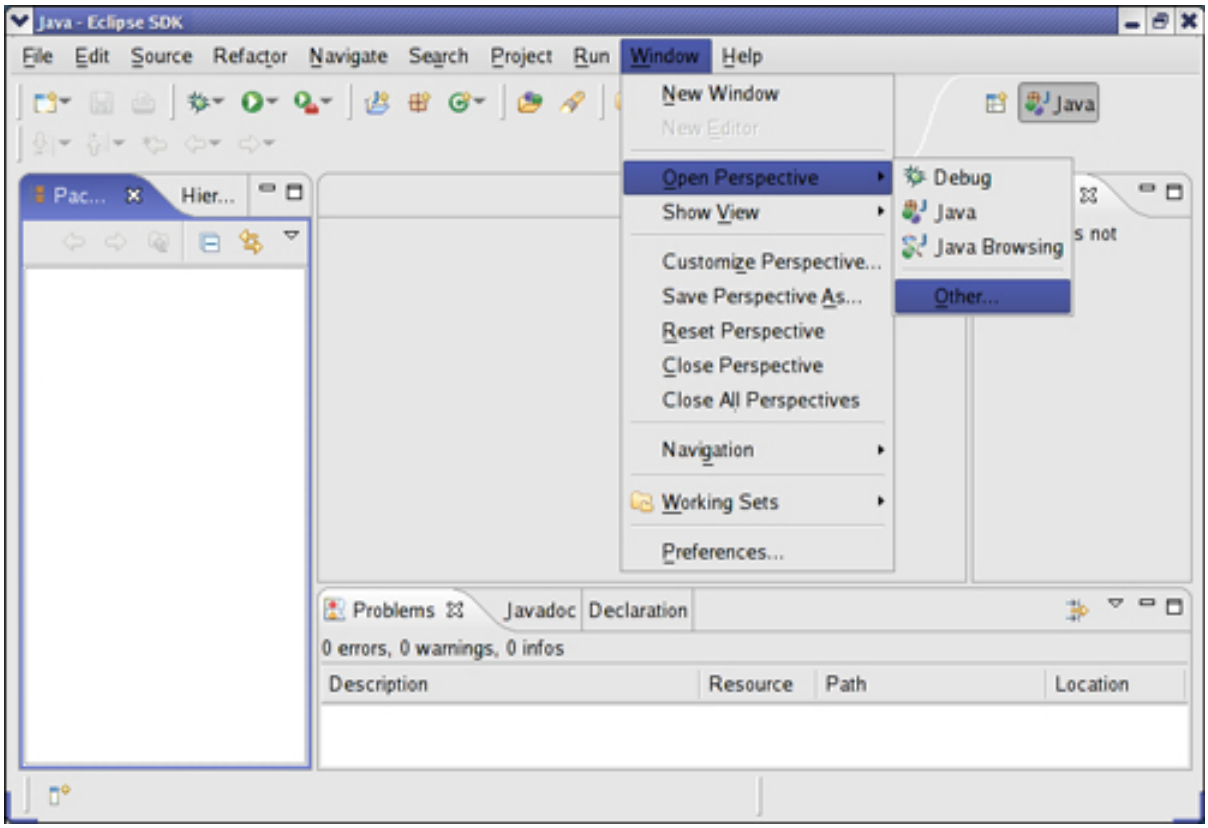
Figure 1. Eclipse



Open C/C++ perspective

Open the C/C++ perspective by selecting **Window > Open Perspective > Other...**.

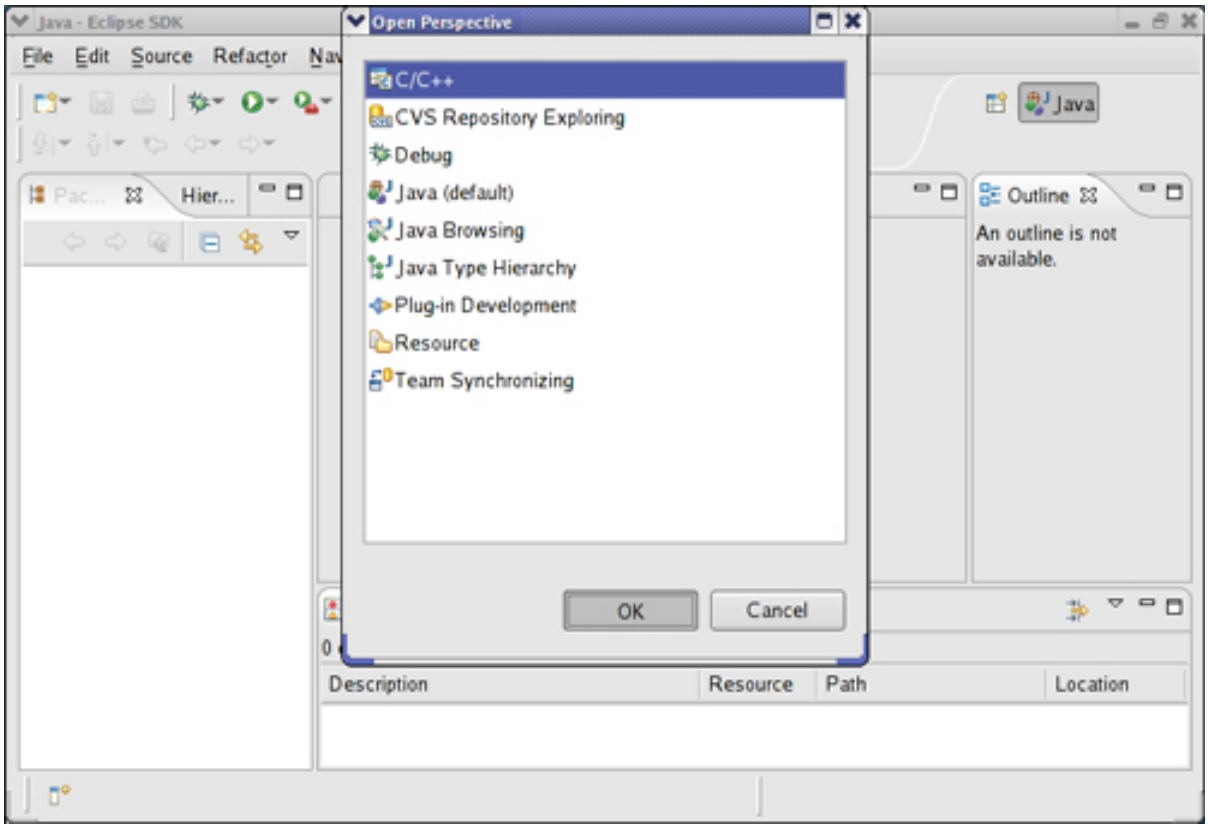
Figure 2. Open C/C++ perspective



Choose C/C++ perspective

Choose **C/C++** from the list and click **OK**.

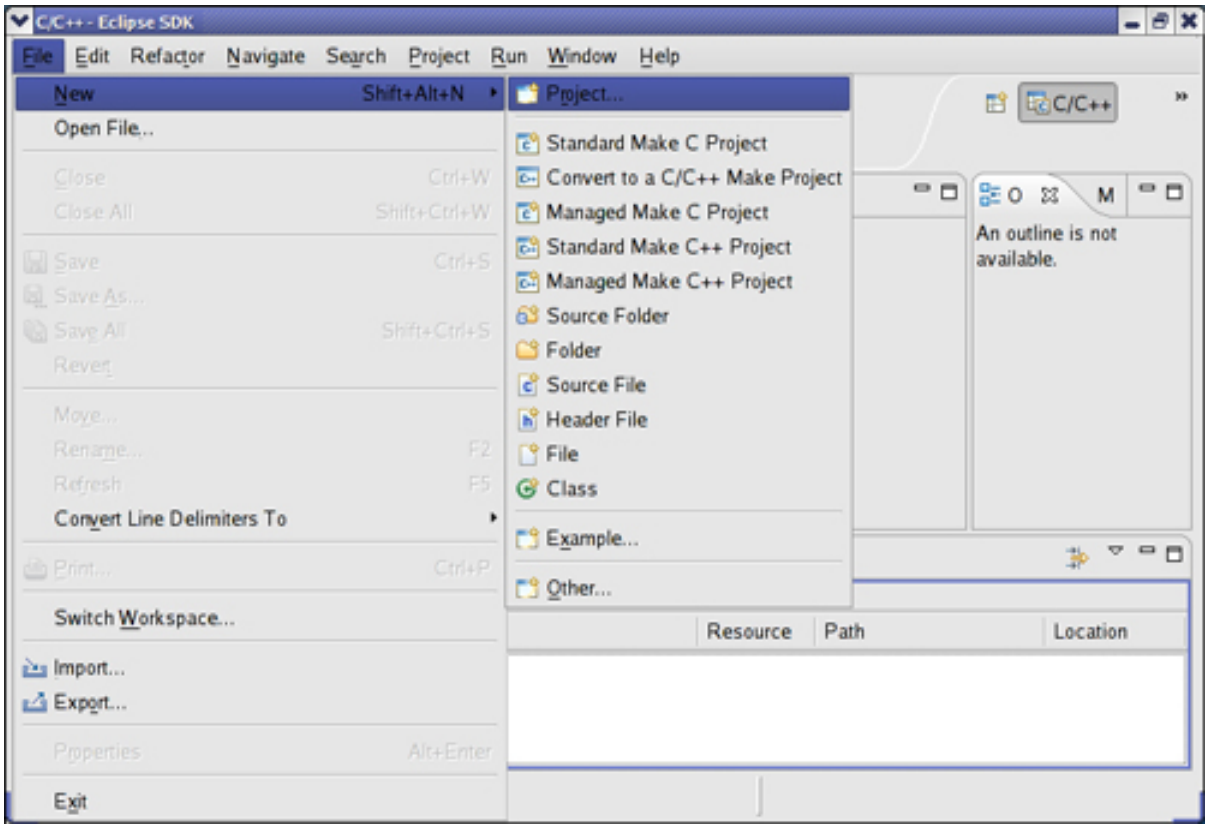
Figure 3. Choose C/C++ perspective



Create a project

Create a new C project by clicking **File > New > Project**.

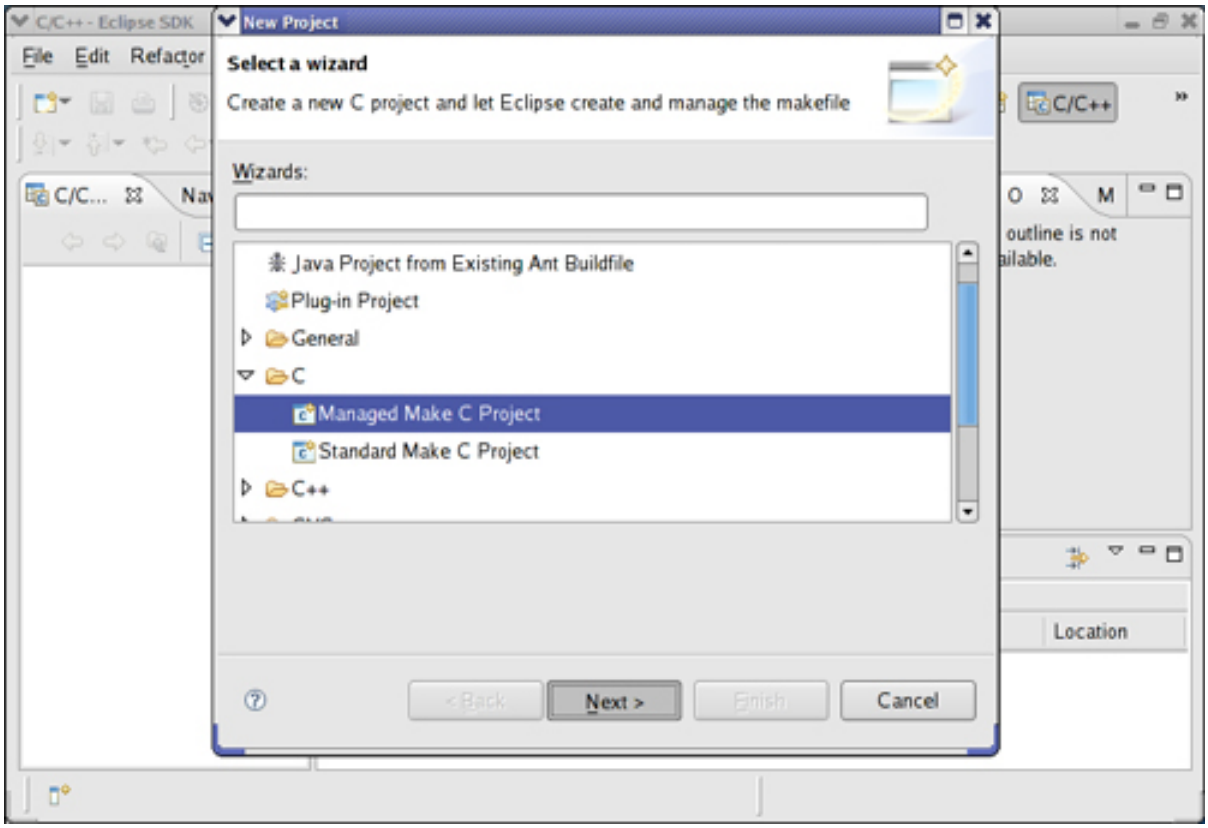
Figure 4. Create a project



New Project wizard

You will now see the New Project wizard. For this example, expand the section **C** and select **Managed Make C Project**. A Standard Make C/C++ Project requires you to provide a makefile; a Managed Make Project creates one for you. So, if your project already has a makefile, or if you would like to create your own makefile, you would choose a Standard Make Project. For this tutorial, use a Managed Make Project. Click **Next**.

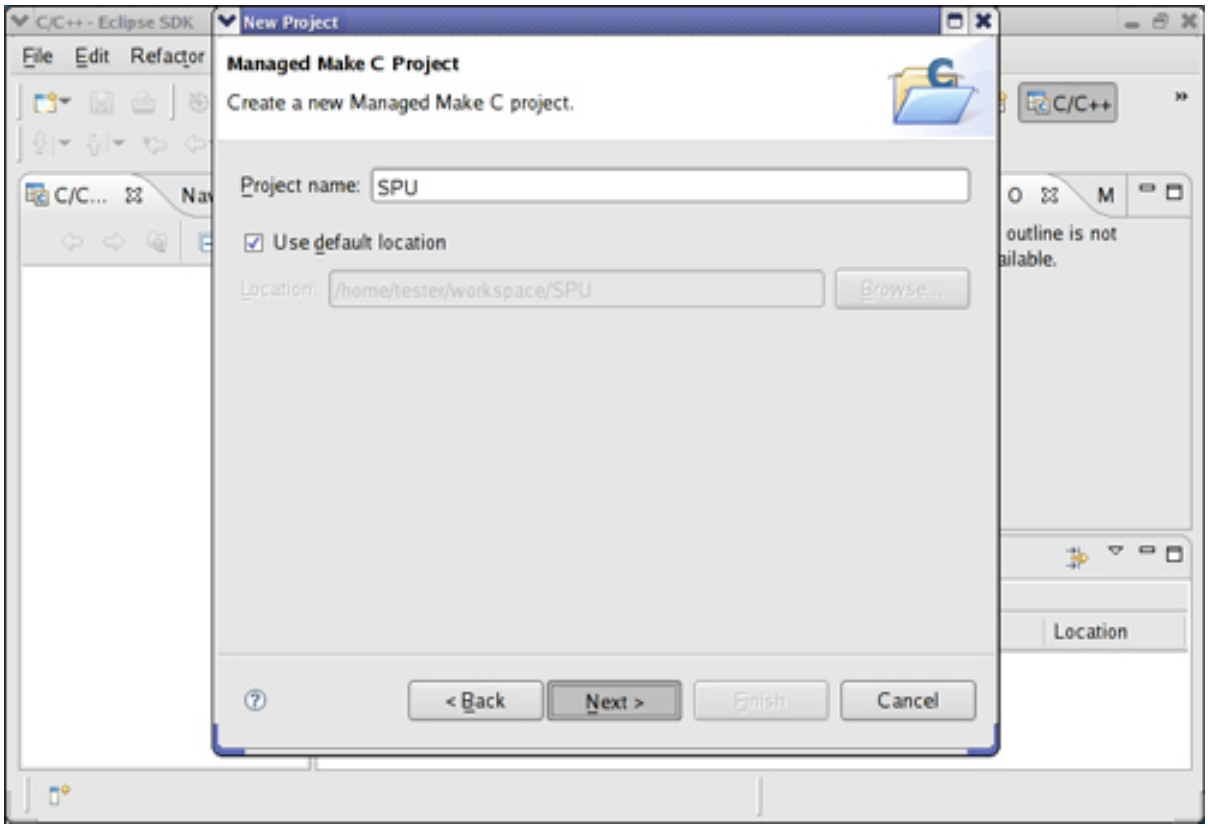
Figure 5. New Project wizard



Project name

Your project needs a unique name. Since this is the SPU project, enter the Project name of SPU. Click **Next**.

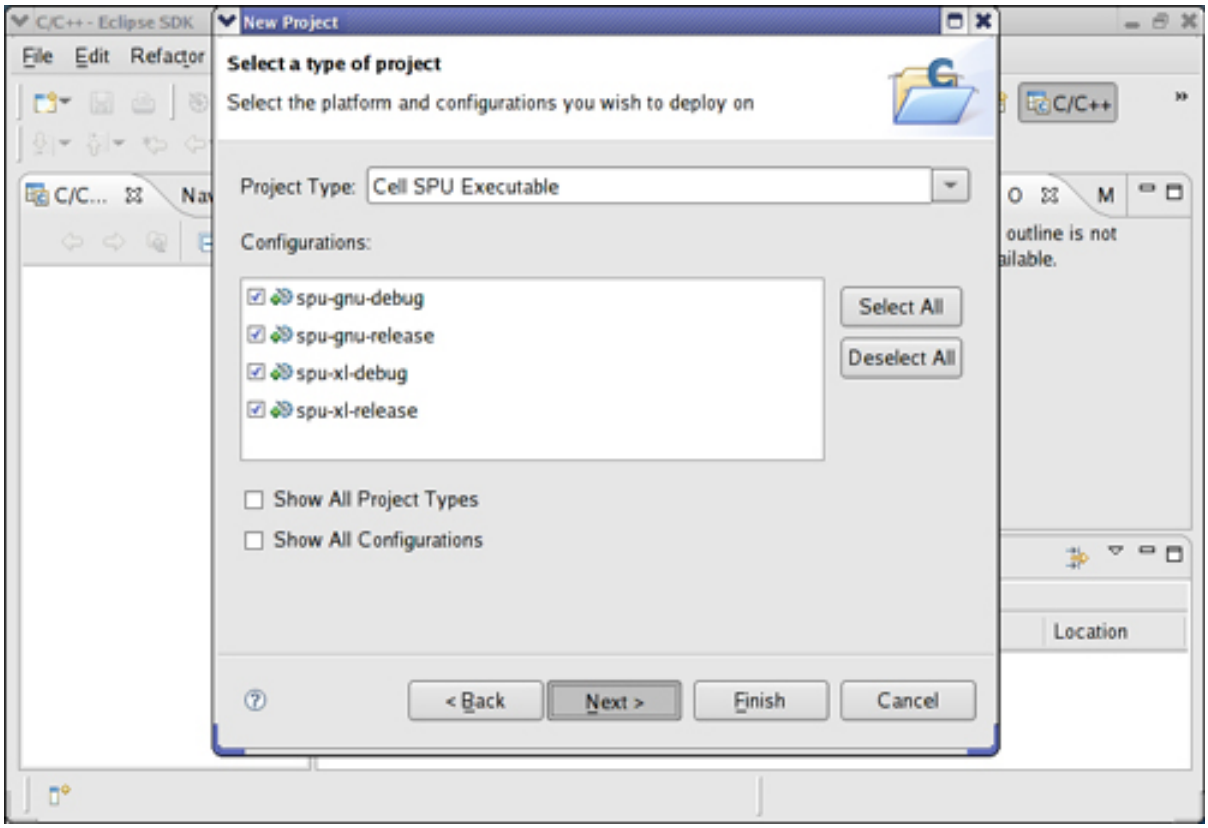
Figure 6. Project name



Select the Project type

You can now 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 can see all of the different available built-in build configurations for that project type. Later in the tutorial, you will see how to create new build configurations, and will learn how to modify the default settings of the built-in configurations. For Project Type, select **Cell SPU Executable**, and then click **Next**.

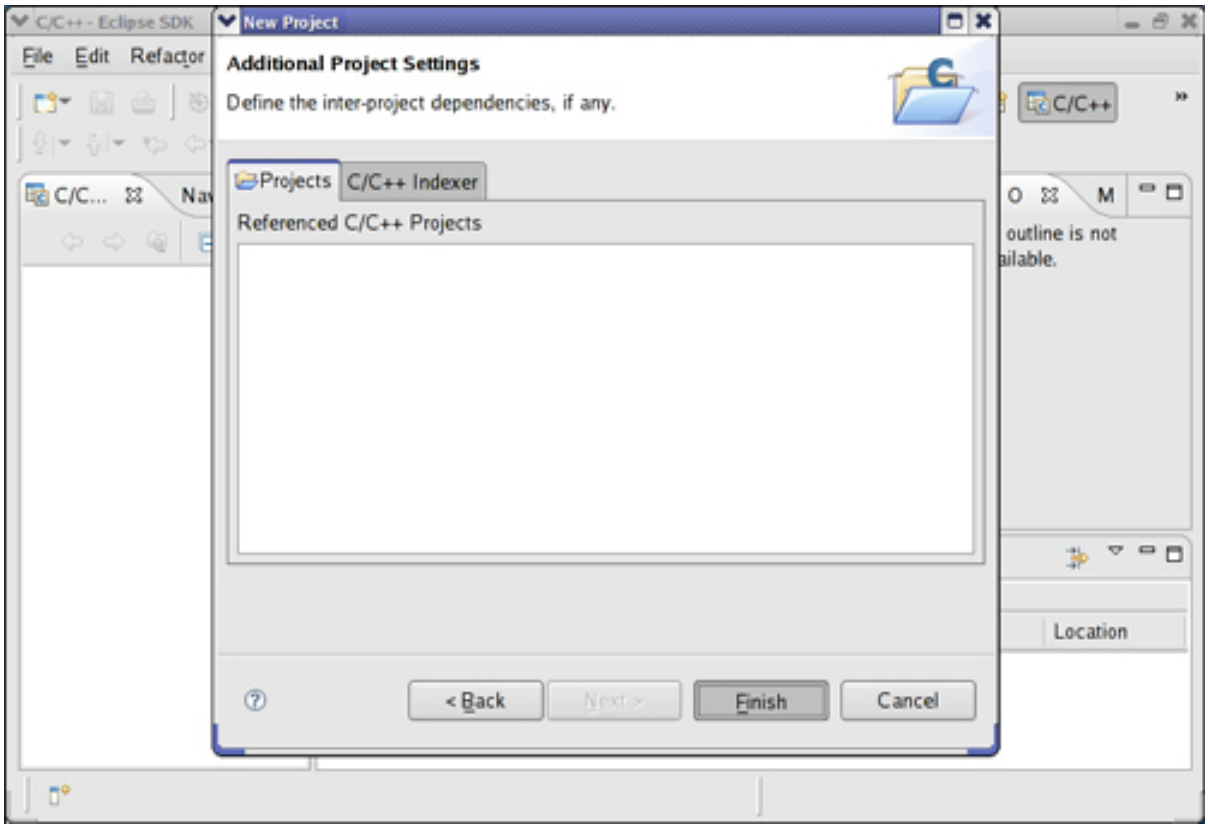
Figure 7. Select the Project type



Project references

The other projects in the workspace are listed here (in this case, there aren't any), and can be referenced by this project if needed. Click **Finish**.

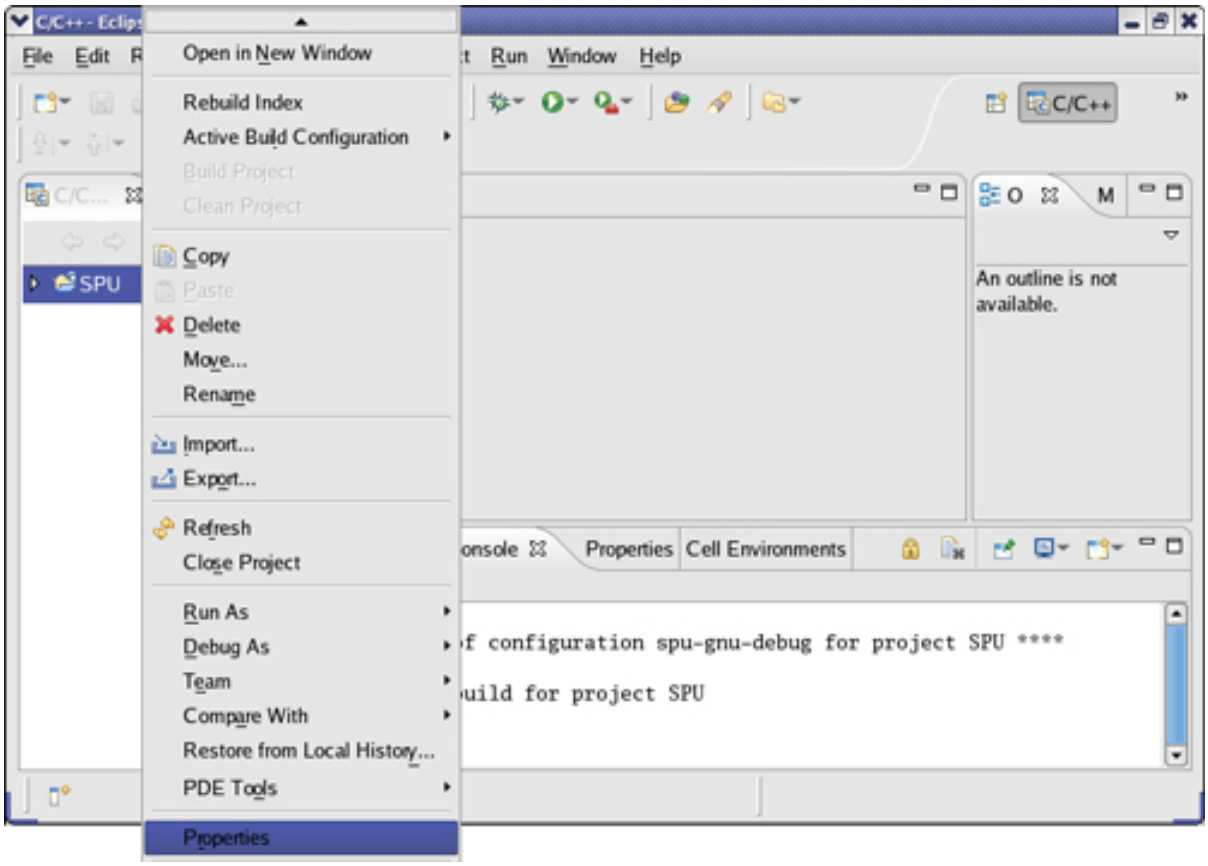
Figure 8. Project references



SPU project properties

In the C/C++ Projects view, right click on the SPU project, and select **Properties**.

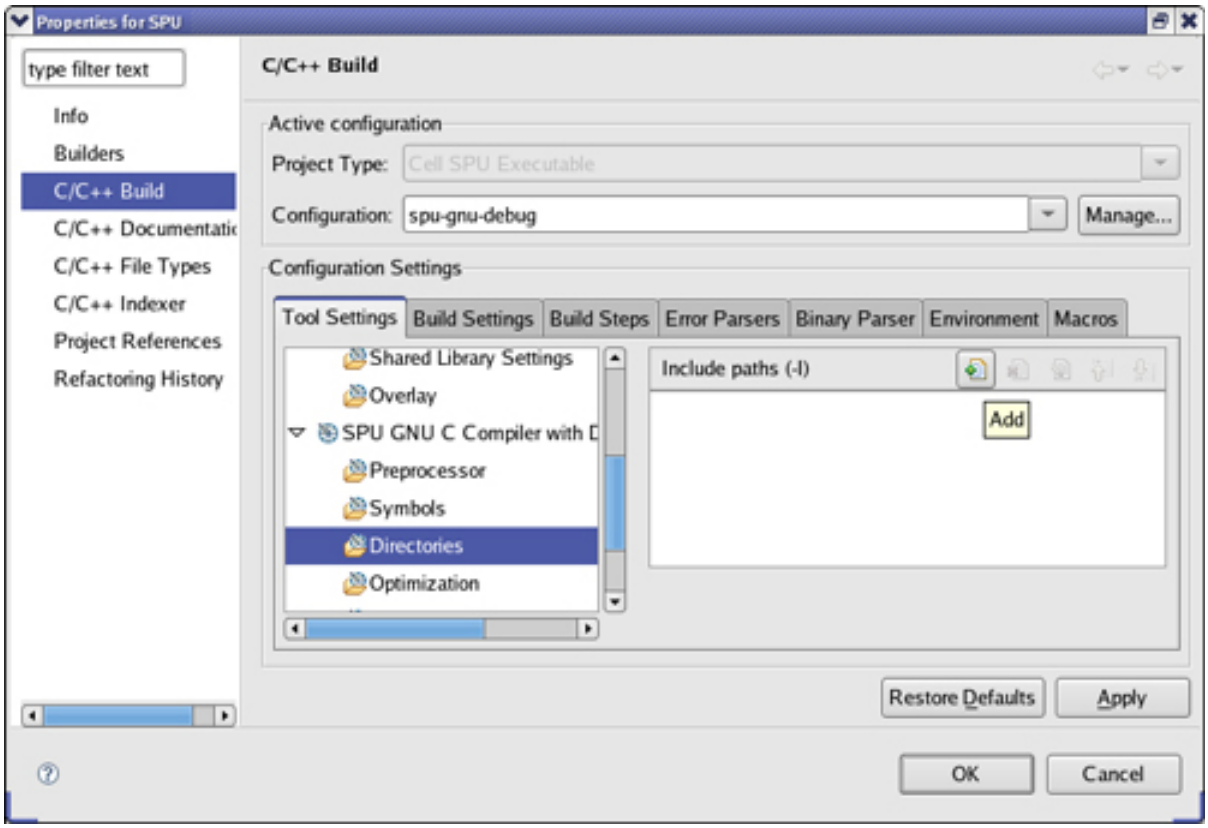
Figure 9. SPU project properties



SPU compiler include paths

You need to add the path of profile.h to the compiler's include paths, so that profile.h can be found. Click on **C/C++ Build** in the left pane, then under SPU GNU C Compiler with Debug Options, select **Directories**. In the Include Paths pane on the right, press **Add**.

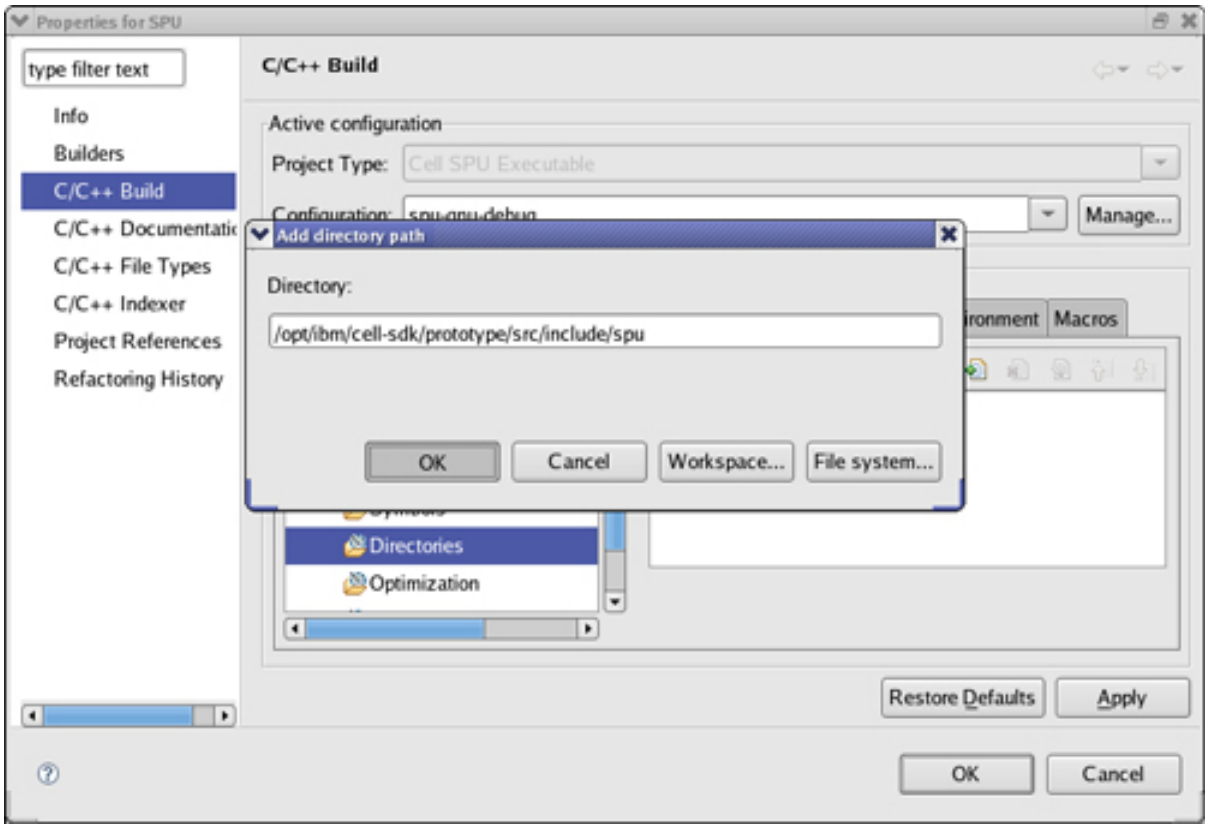
Figure 10. SPU compiler include paths



Add directory path

Type `/opt/ibm/cell-sdk/prototype/src/include/spu`, then return to the C/C++ Perspective by pressing **OK** twice.

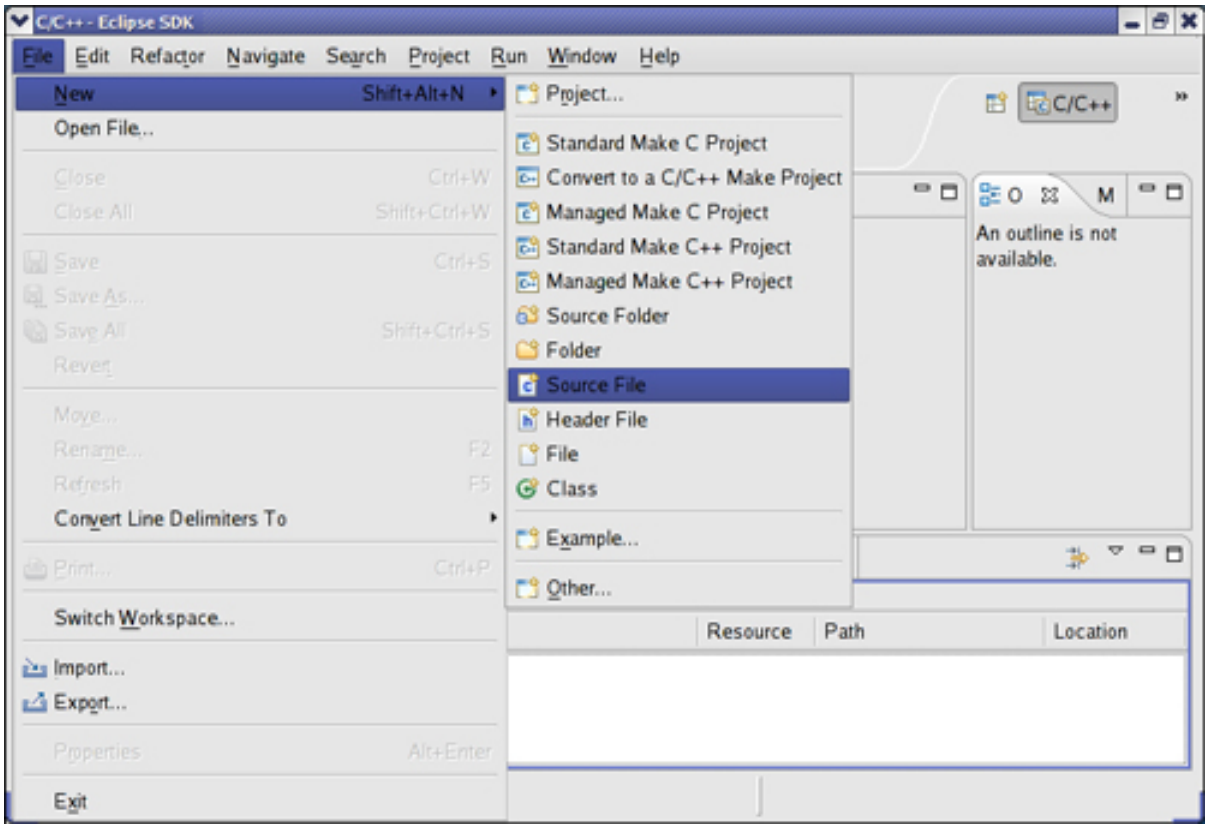
Figure 11. Add directory path



Create a new source file

Create a .c source file by clicking **File > New > Source File**.

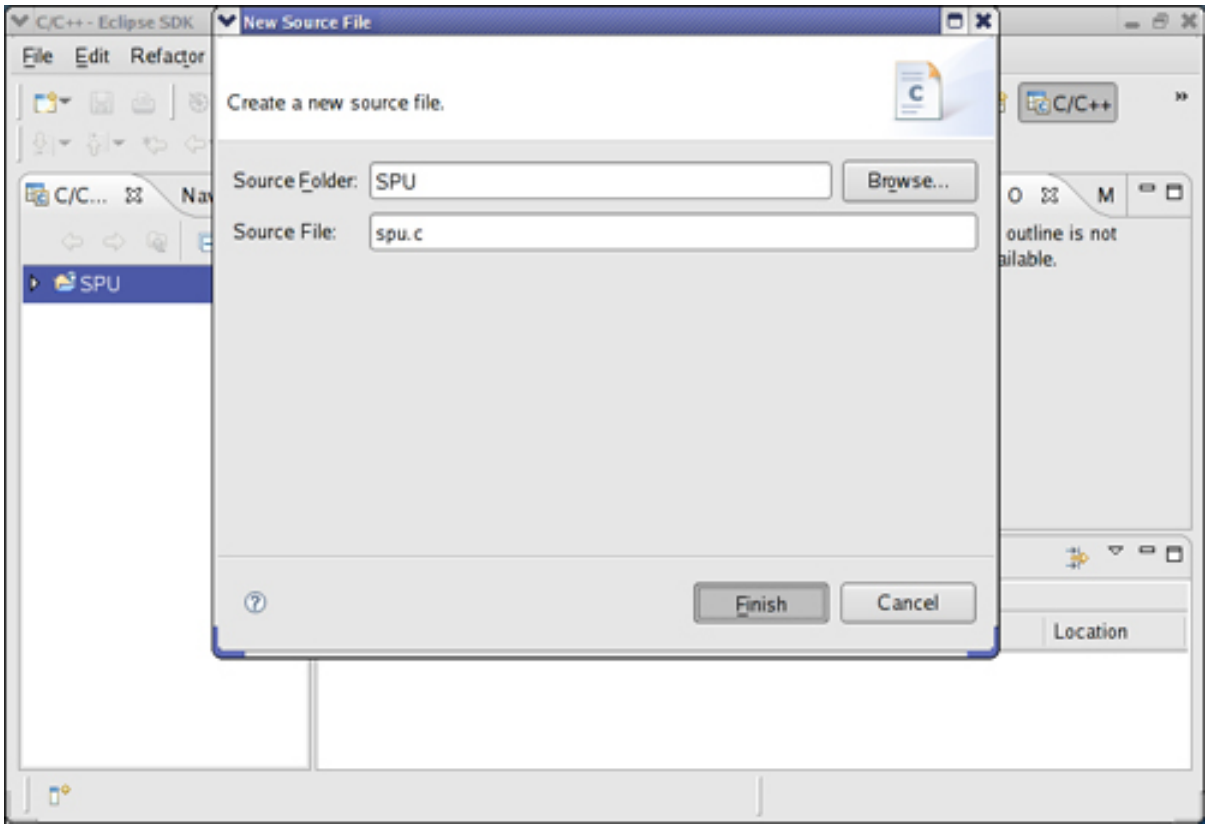
Figure 12. Create a new source file



New source file

For Source File, type in `spu.c`. Click **Finish**.

Figure 13. New source file

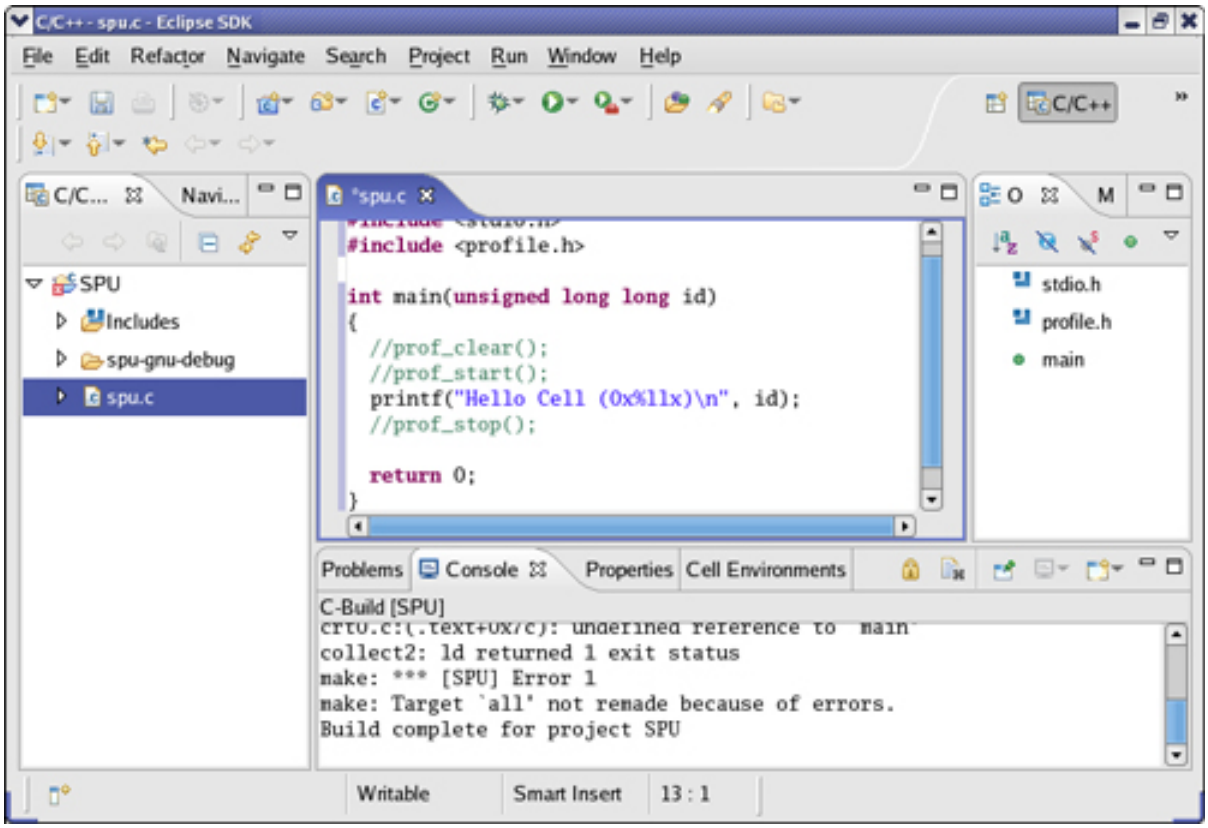


Edit the source file

A new editor appears so you can enter your source code. This program will run on the SPU, and this is where the bulk of processing would likely occur in a real application. Copy and paste the following source code into your editor:

```
#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;
}
```

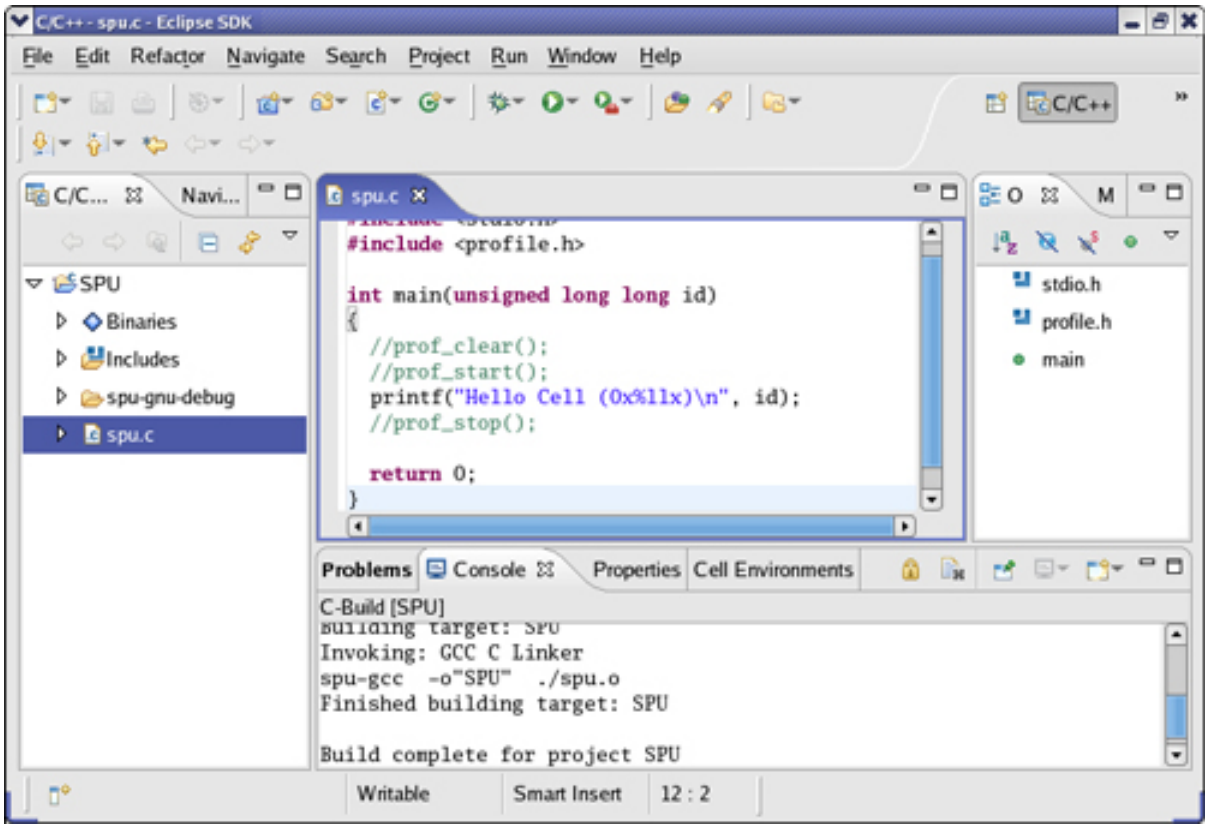
Figure 14. Edit the source file



Project build

Save the source file (**Control + S**). The project is built automatically. The build output is displayed in the Console view, and new files, such as binaries and includes, are shown in the C/C++ Projects view.

Figure 15. Project build



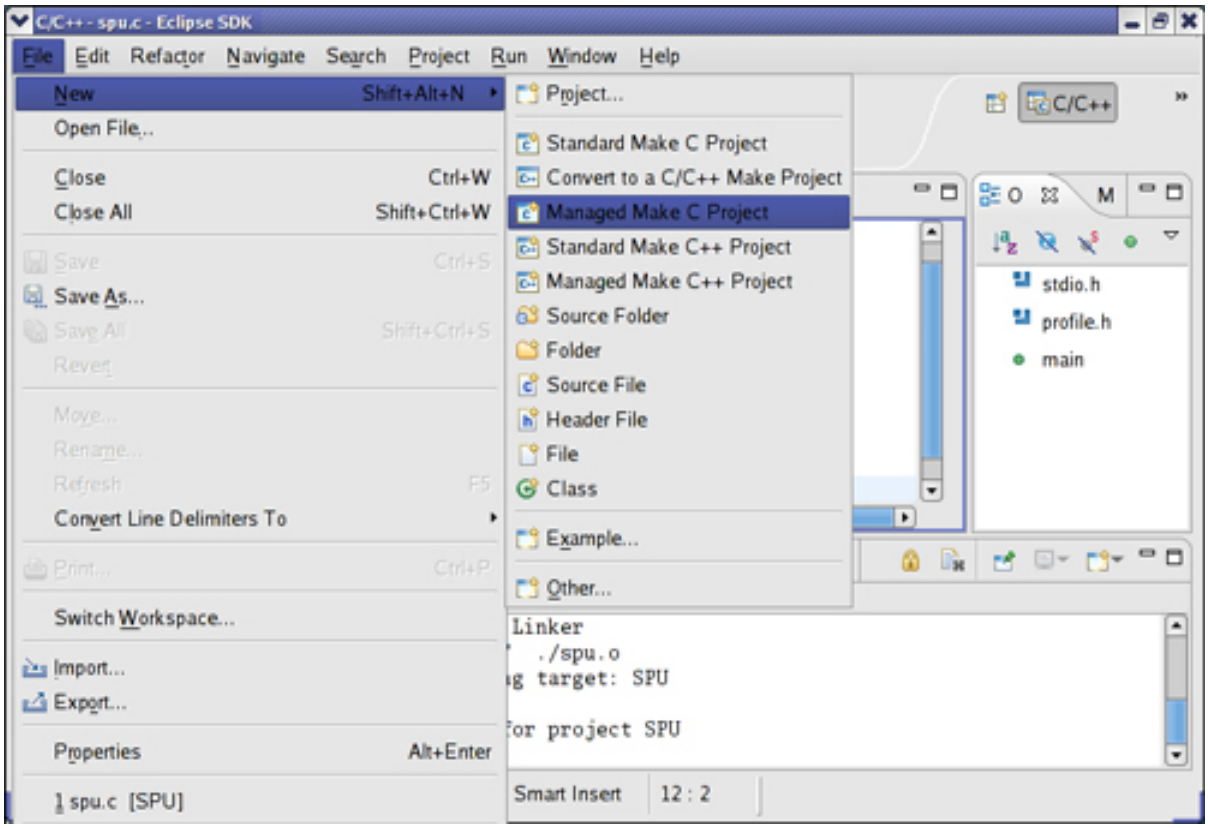
Section 3. Creating the PPU project

Create PPU project

Now you will create a PPU executable project which will embed the SPU program you just created in a PPU executable, by using the Embed SPU tool on the SPU project.

Click **File > New > Managed Make C Project**.

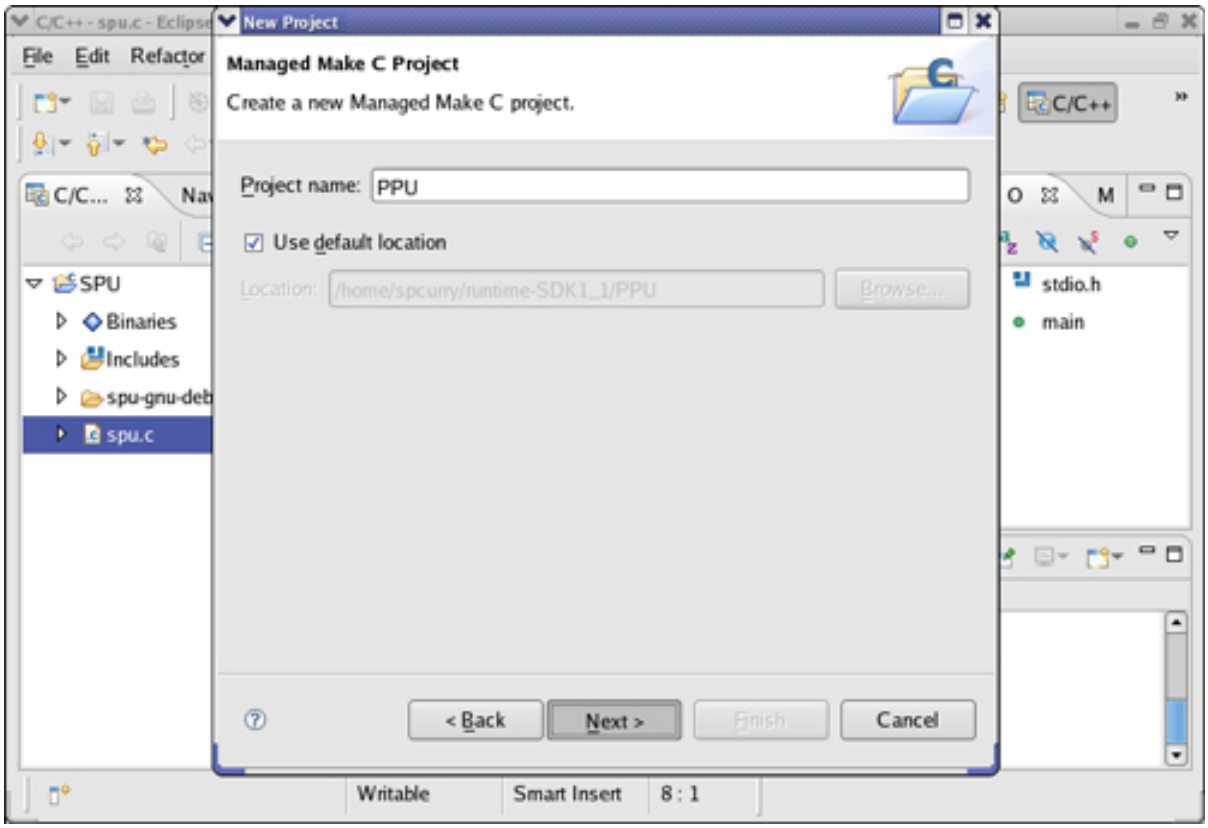
Figure 16. Create PPU project



Project name

For the project name, type in PPU. Click **Next**.

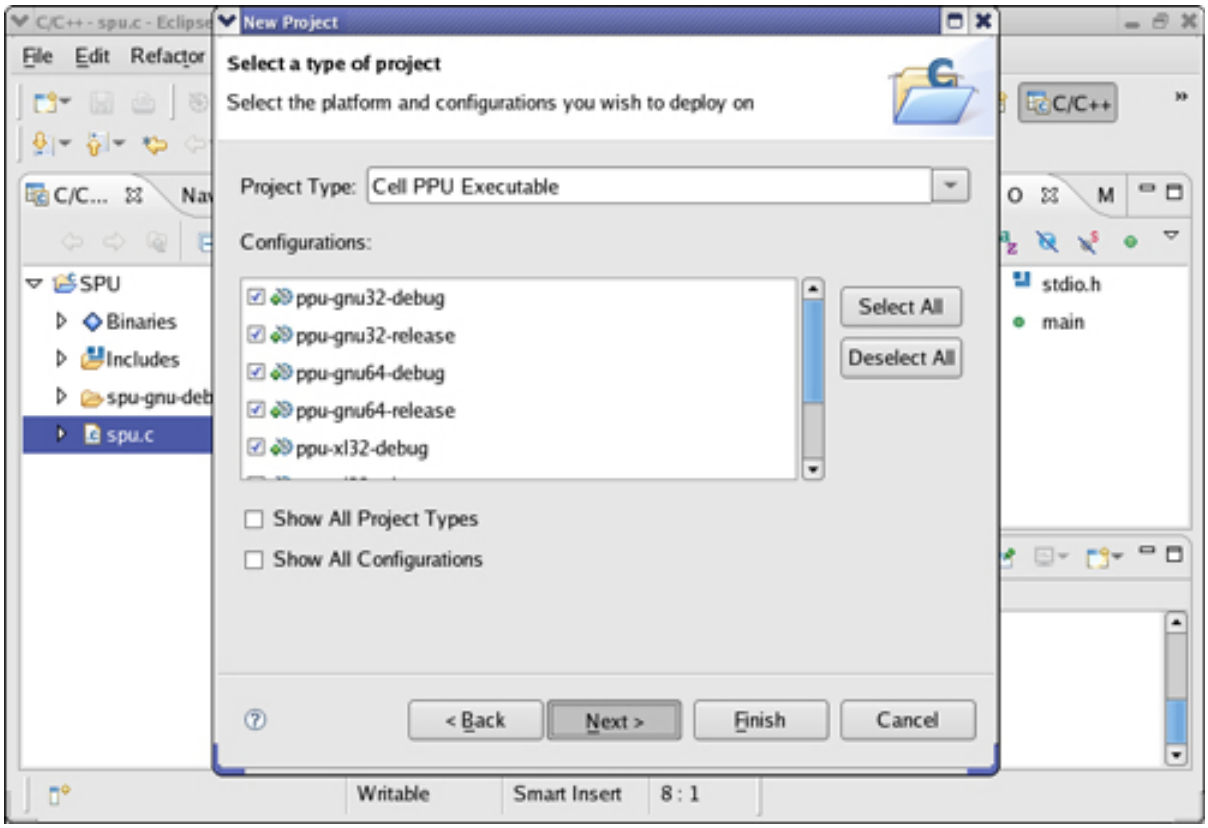
Figure 17. Project name



Project type

For Project Type, select **Cell PPU Executable**. Click **Next**.

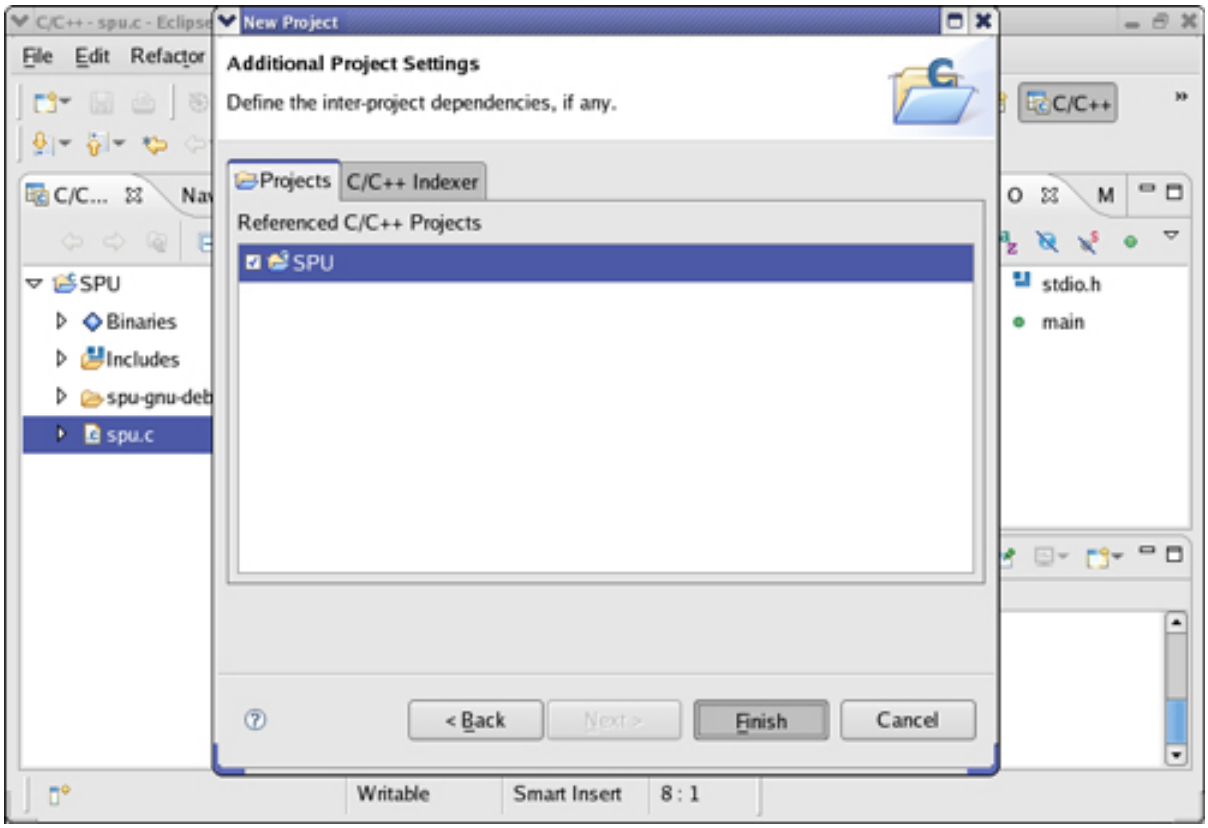
Figure 18. Project type



Project references

The SPU type project that you created earlier is now listed. You will need a reference to this SPU project in order to embed its object code. Check the box next to **SPU**, then click **Finish**.

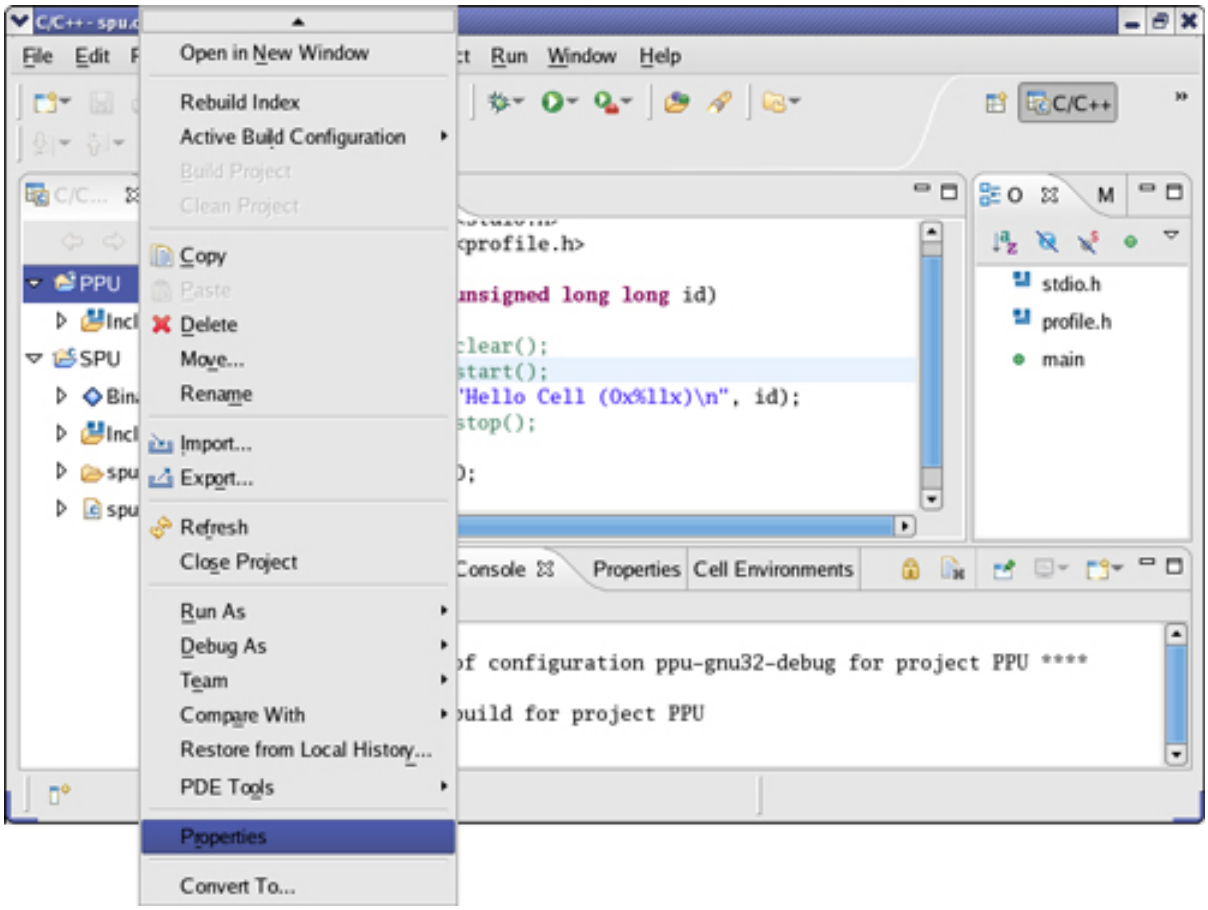
Figure 19. Project references



Configure project properties

In the C/C++ Projects view, right click on the **PPU** project, and select **Properties**.

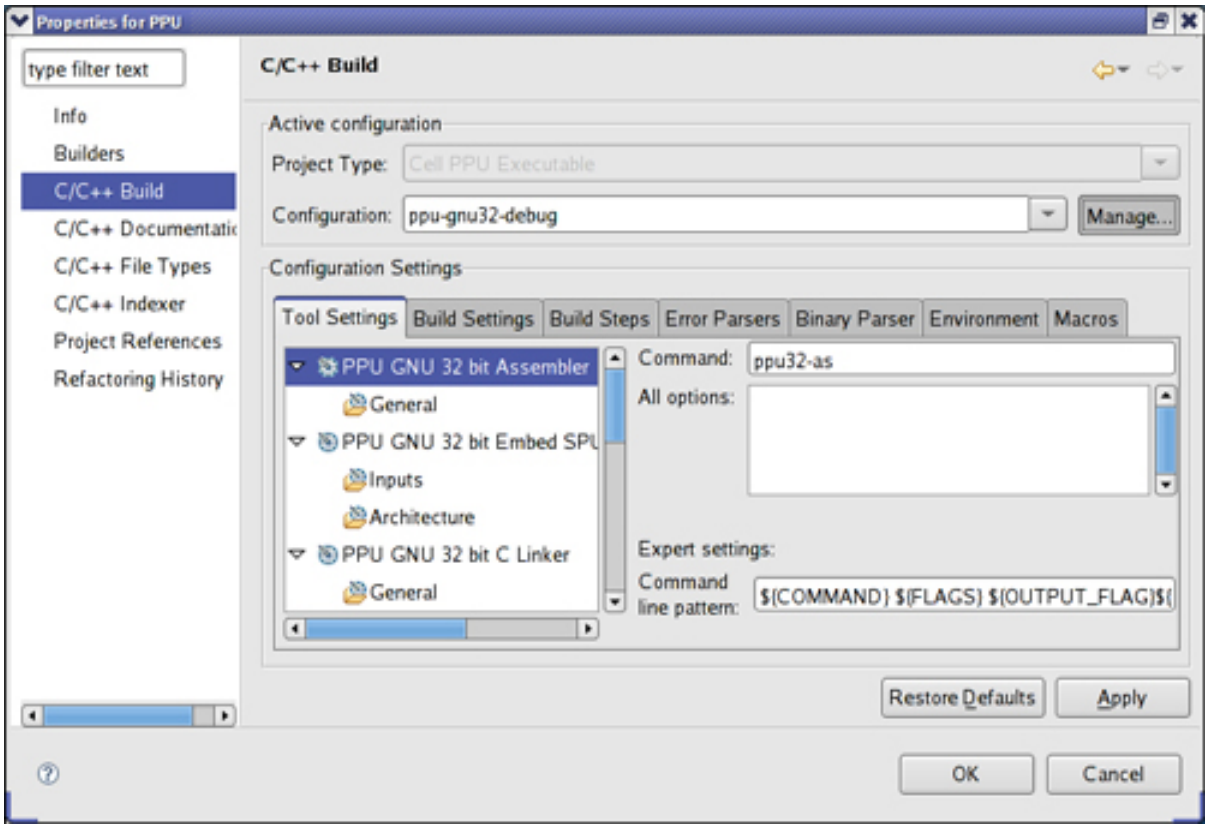
Figure 20. Configure project properties



C/C++ build options

In the left pane, select **C/C++ Build**. In the Active 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. Below, in the Tool Settings tab, you can manipulate the compiler, linker, assembler, and the embed SPU function (PPU project types only). Click **Manage...**

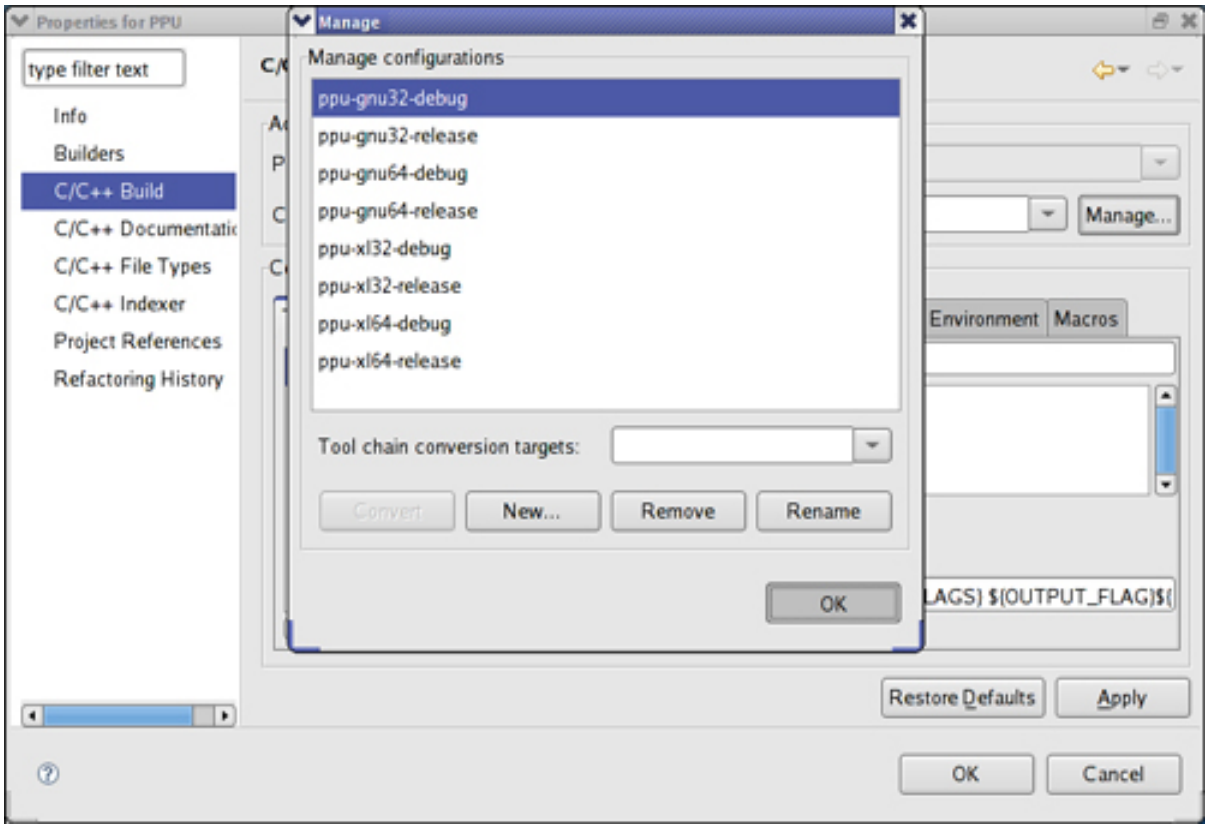
Figure 21. C/C++ build options



Manage configurations

Here you can create new build configurations. You can also rename or remove current configurations. Click **OK**.

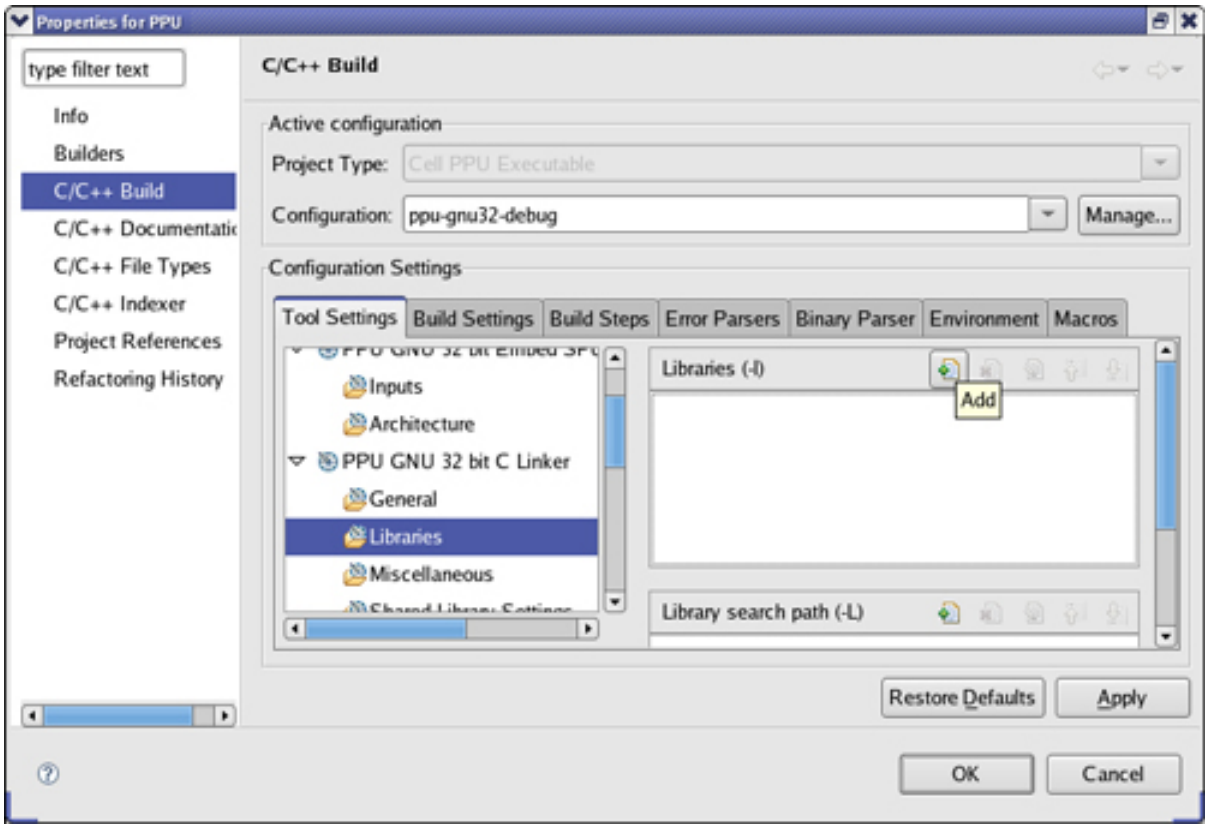
Figure 22. Manage configurations



PPU linker libraries

We will be using the library libspe.h in our PPU source code, so you need to add this library to the PPU Linker's Libraries. In the Tool Settings tab, click on **Libraries** (under the PPU GNU 32 bit C Linker category), then click **Add** in the Libraries pane on the right.

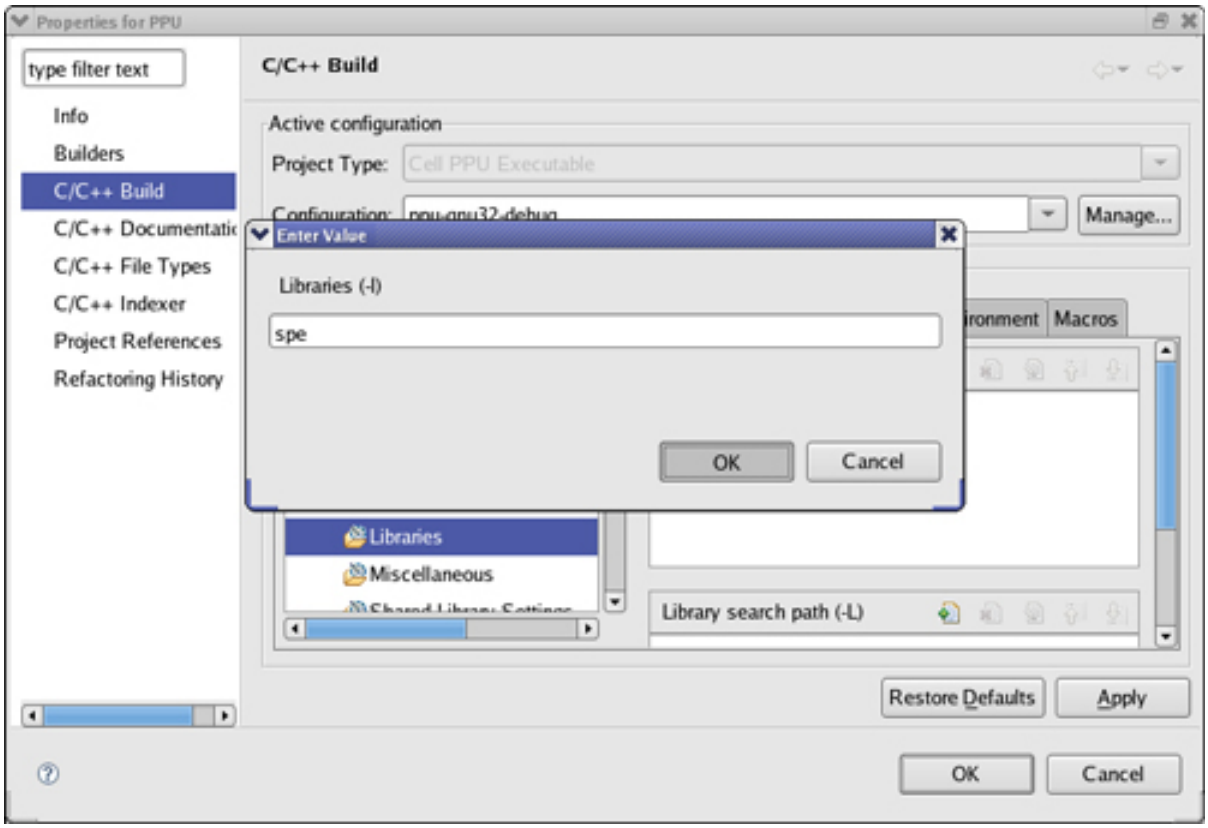
Figure 23. PPU linker libraries



Add library

Type in spe and click **OK**.

Figure 24. Add library

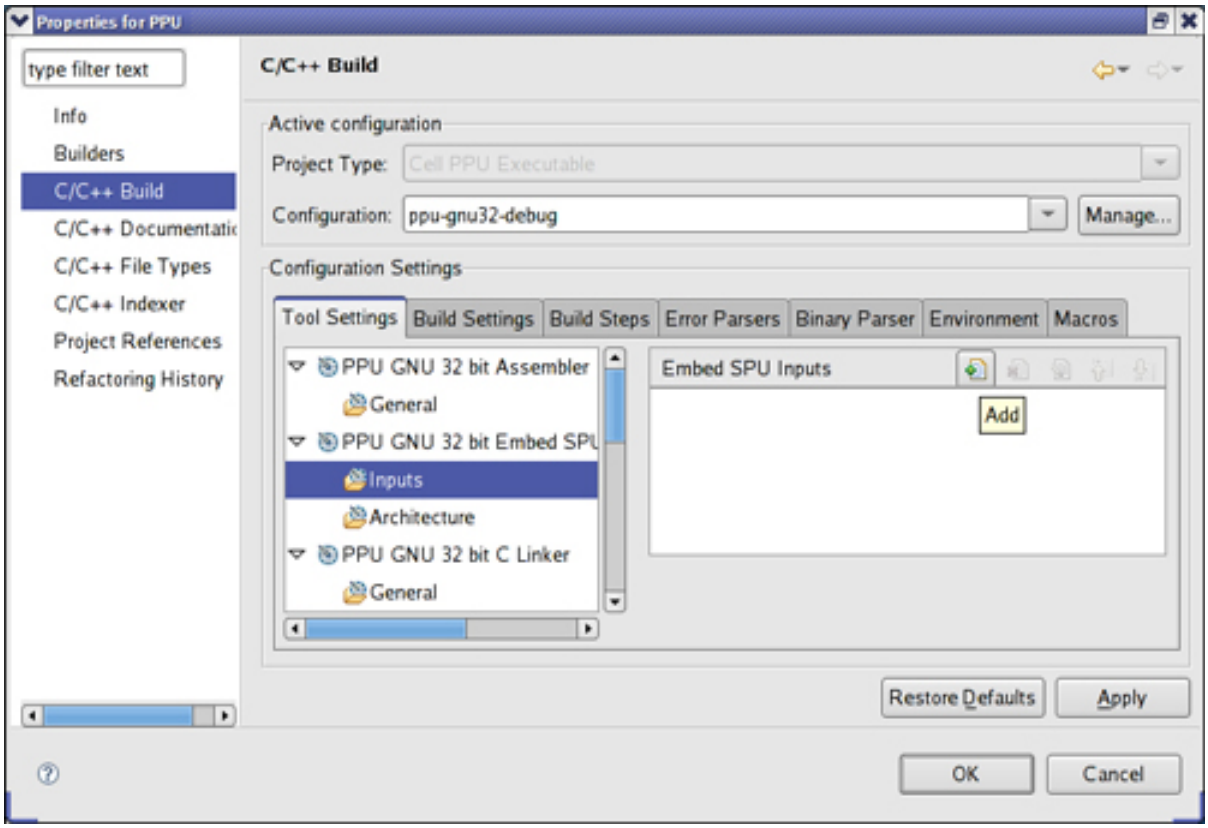


Section 4. Adding the embed SPU input

Add embed SPU input

To configure the PPU project to embed the spu.c program, you need to add an Embed SPU input. In the Tool Settings tab and under PPU GNU 32 bit Embed SPU, select the **Inputs** option, and then click **Add** in the Embed SPU Input pane.

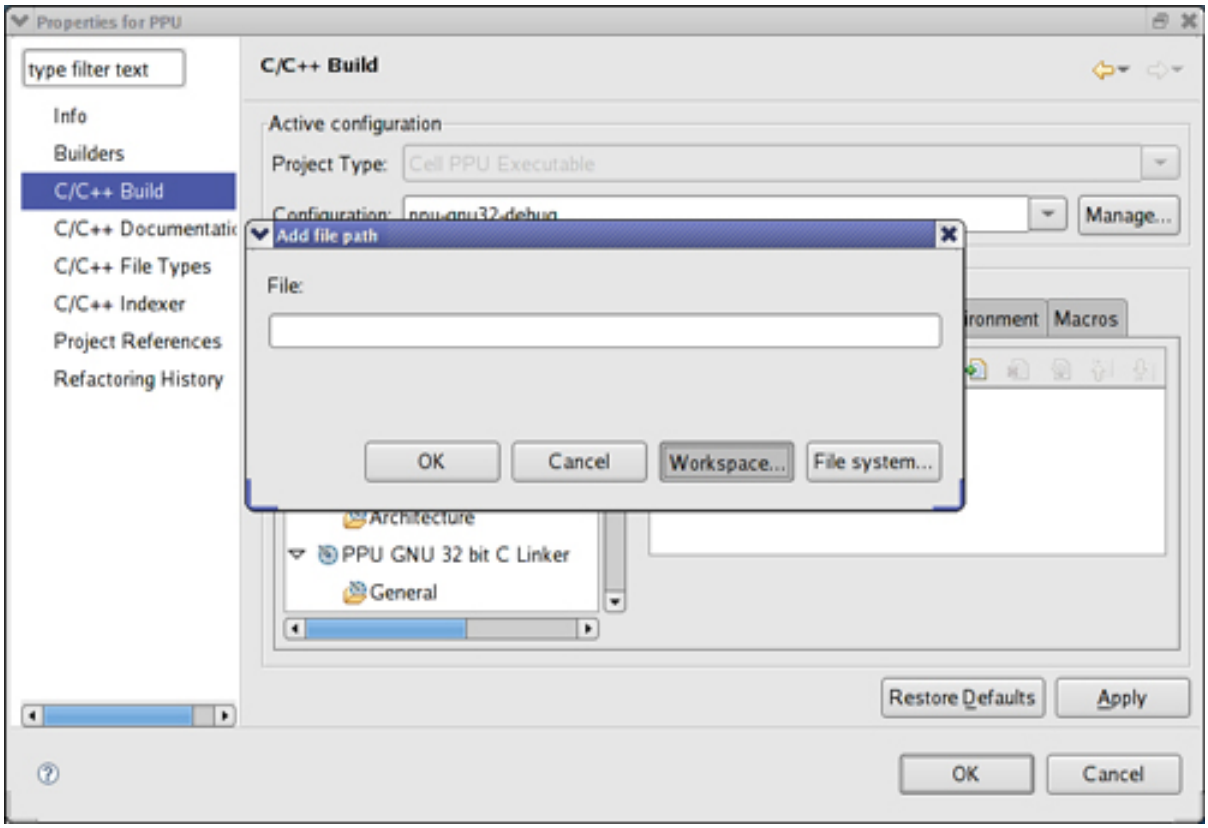
Figure 25. Add embed SPU input



Browse the workspace

Click the **Workspace** button.

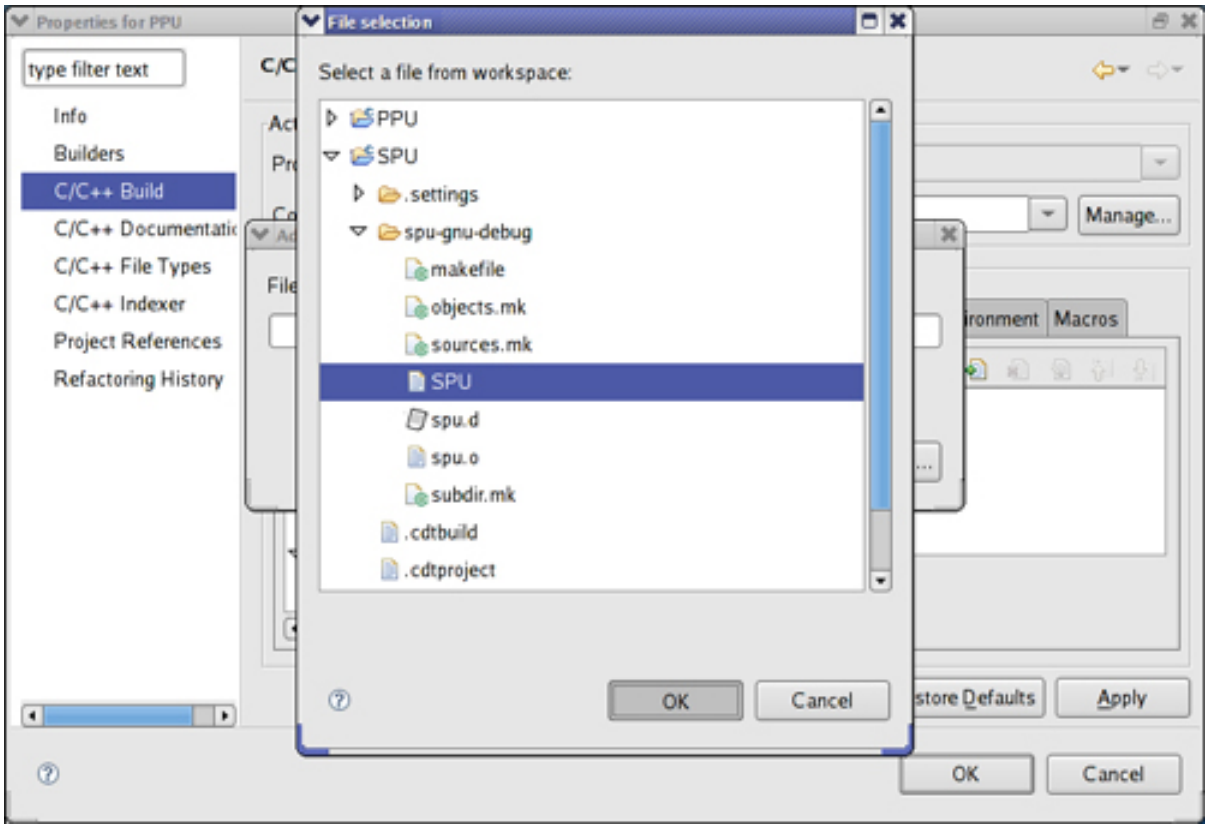
Figure 26. Browse the workspace



Select SPU executable

Select the file: **SPU > spu-gnu-debug > SPU**, and then return to the properties for PPU page by pressing **OK** twice.

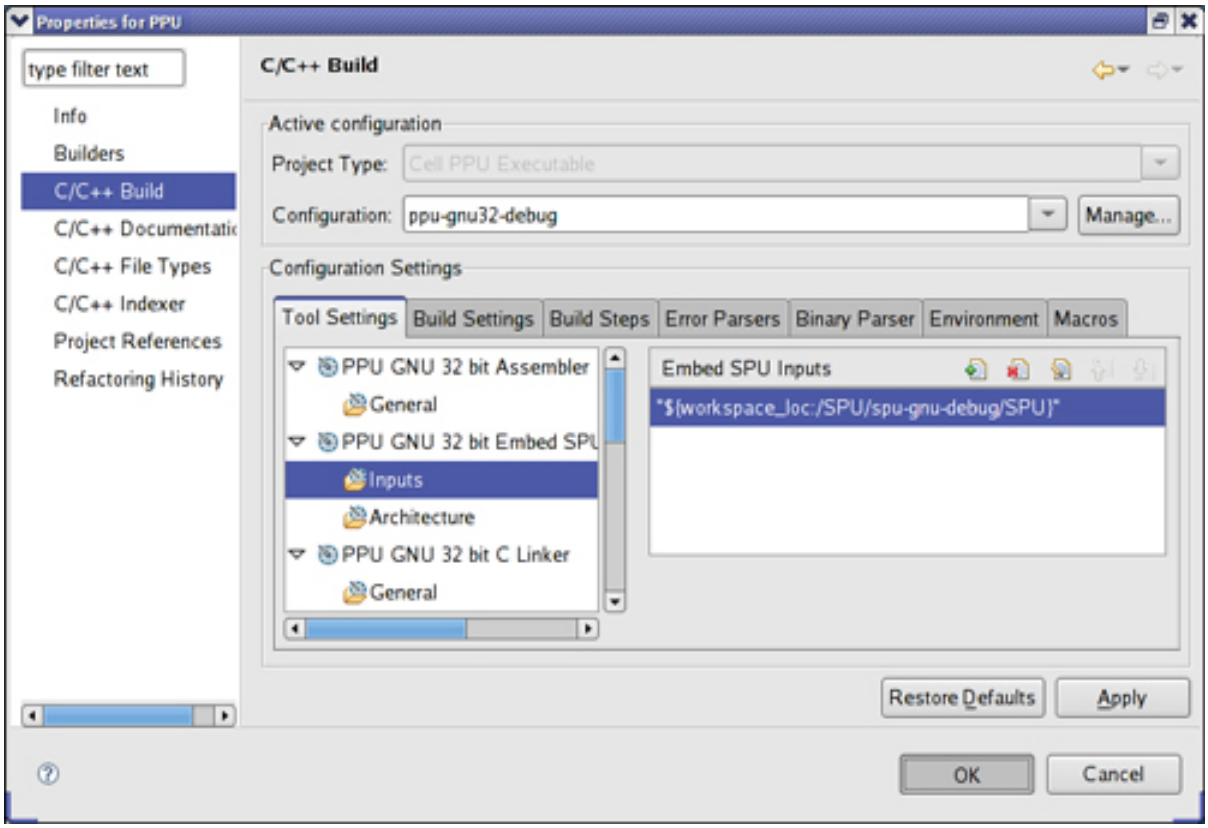
Figure 27. Select SPU executable



Other settings

You can configure various other settings on the other tabs (Build Settings, Build Steps, and so on) and in the other categories on the left pane (such as Project References). Take a minute to explore these other tabs and categories, but don't make any changes; then click **OK** when you're done.

Figure 28. Other settings

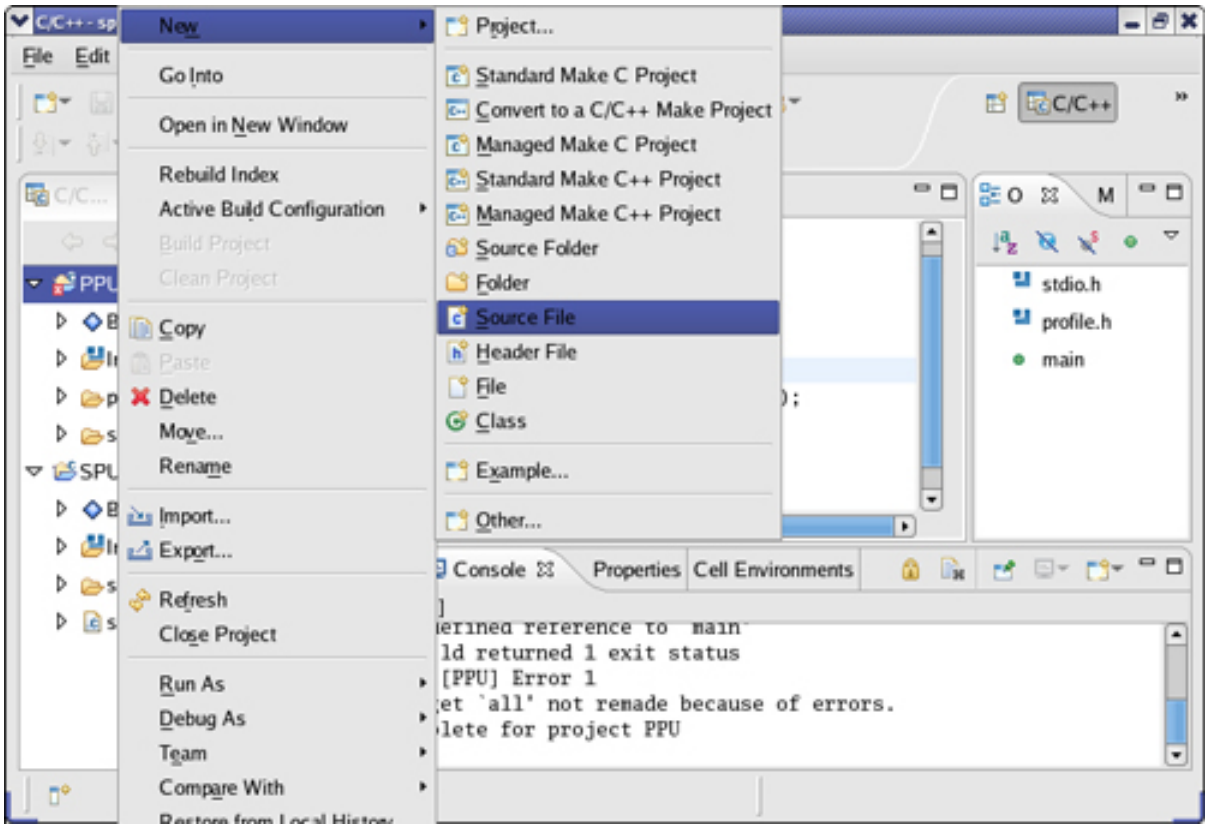


Section 5. Creating the PPU source file

Create another source file

Right click on the PPU project, and select **File > New > Source File**.

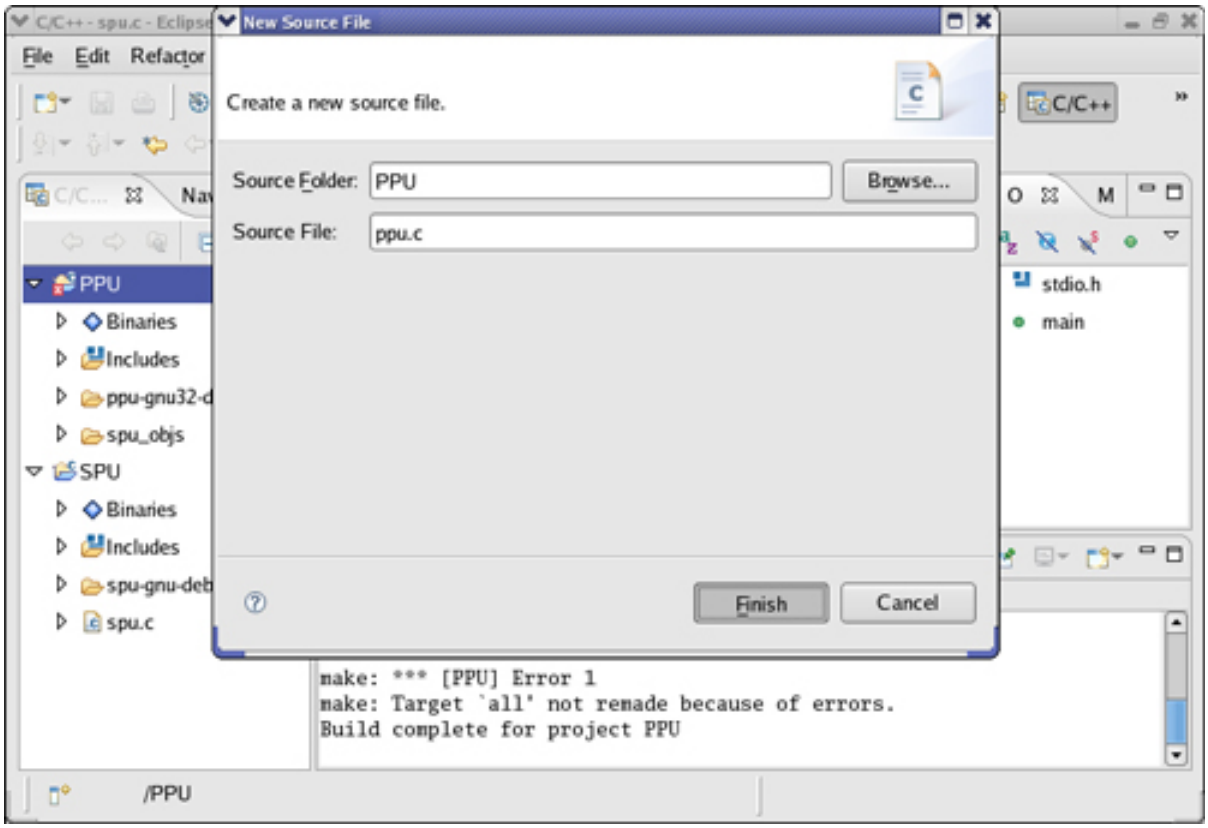
Figure 29. Create another source file



New source file

For the Source File field, type `ppu.c`. Click **Finish**.

Figure 30. New source file



Edit the source code file

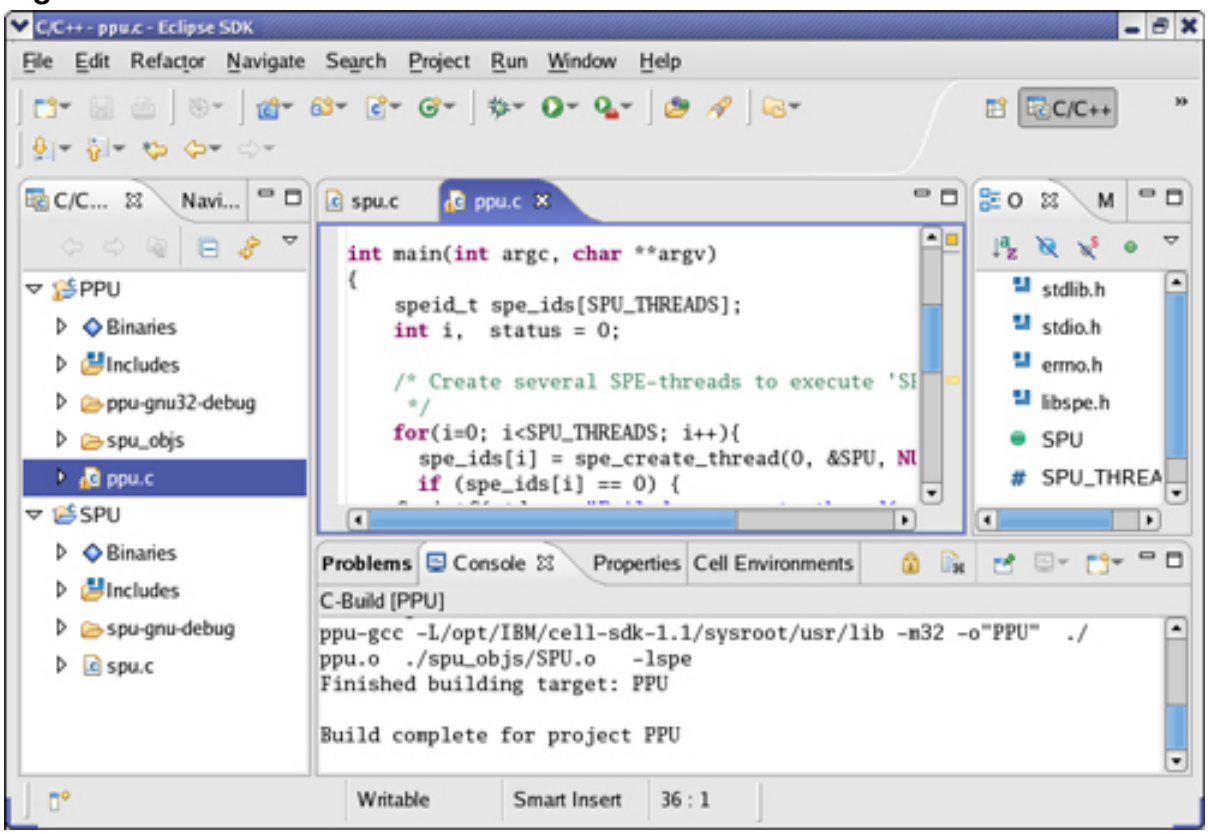
The PPU program is much more complicated than the SPU program. It creates a number of SPU threads, each of which is launched using the embedded SPU binary (now seen as `spe_program_handle_t SPU`).

Copy and paste the following source code into your editor, and then save it:

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe.h>

extern spe_program_handle_t SPU;

#define SPU_THREADS  8

int main(int argc, char **argv) {
    speid_t spe_ids[SPU_THREADS];
    int i,  status = 0;

    /* Create several SPE-threads to execute 'SPU'.  */
    for(i=0; i < SPU_THREADS; i++) {
      spe_ids[i] = spe_create_thread(0, &SPU, NULL, NULL, -1, 0);
      if (spe_ids[i] == 0) {
        fprintf(stderr, "Failed spu_create_thread(rc=%d, errno=%d)\n",
                spe_ids[i], errno); exit(1); } }
```

```
    /* Wait for SPU-thread to complete execution.  */
    for (i=0; i < SPU_THREADS; i++) {
      (void)spe_wait(spe_ids[i], &status, 0); }

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

    return (0);
}
```

Figure 31. Edit the source code file

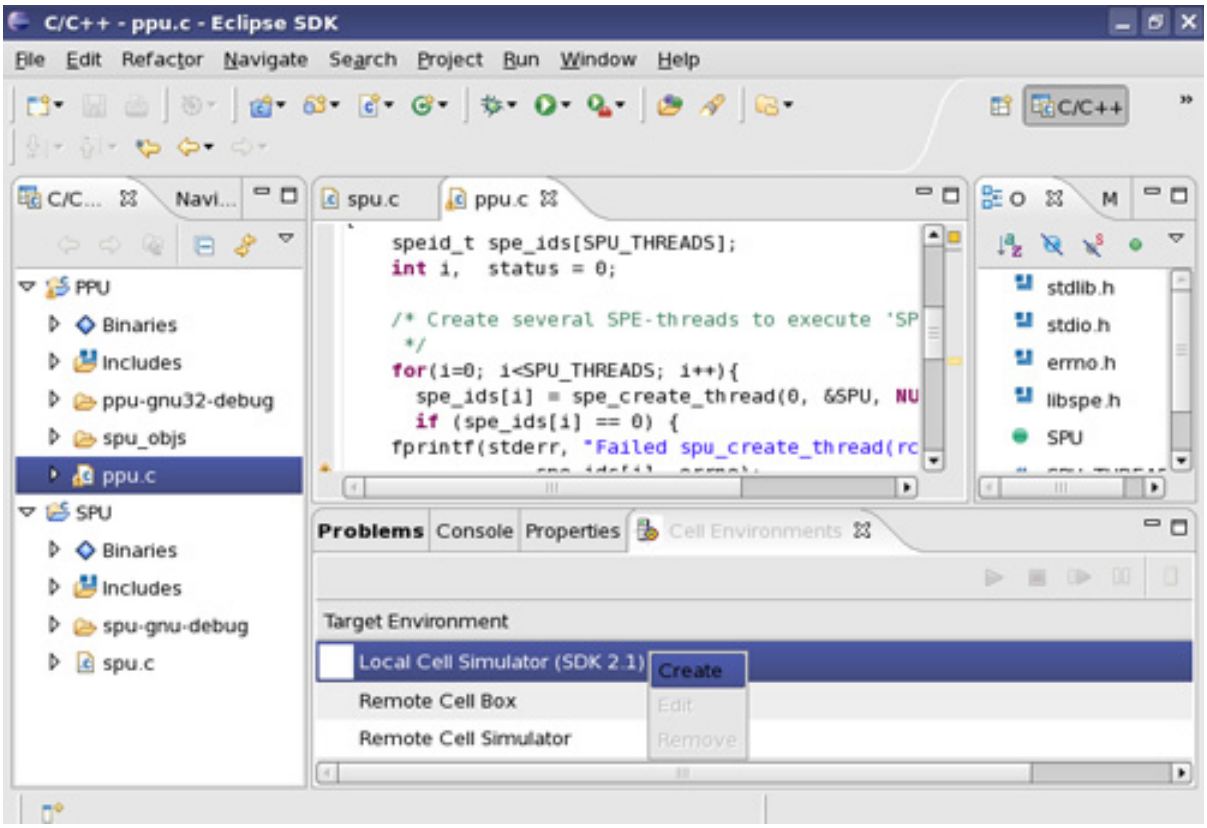


Section 6. Testing the program

Create Cell simulator

Now that you've created and properly configured the projects, you can test out the Embed SPU functionality, but first, you must create and start a Cell Environment configuration. In the Cell Environments view at the bottom, right click on **Local Cell Simulator** and select **Create**.

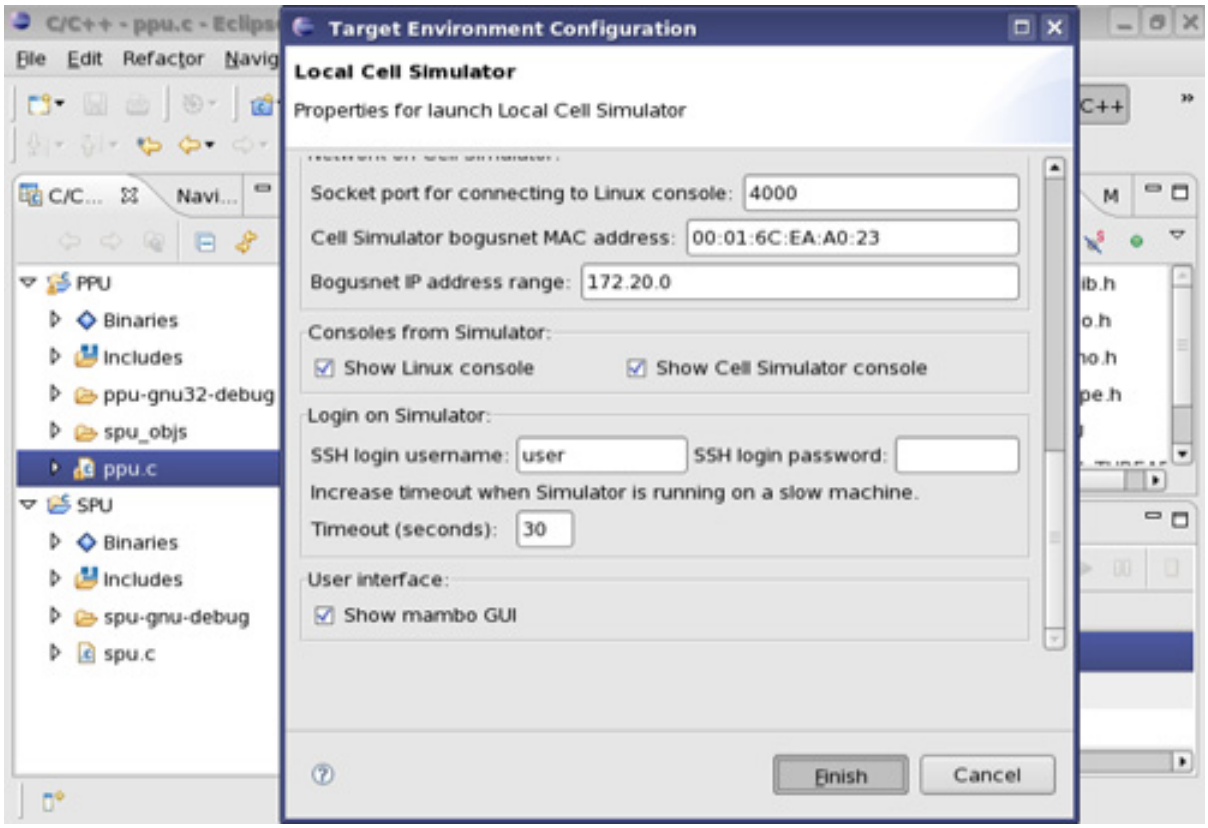
Figure 32. Create Cell simulator



Simulator configuration

Enter a name for this simulator configuration (for example, Sim). Also, ensure that the options *Show Linux Console* and *Show Cell Simulator Console* are both checked. Click **Finish**.

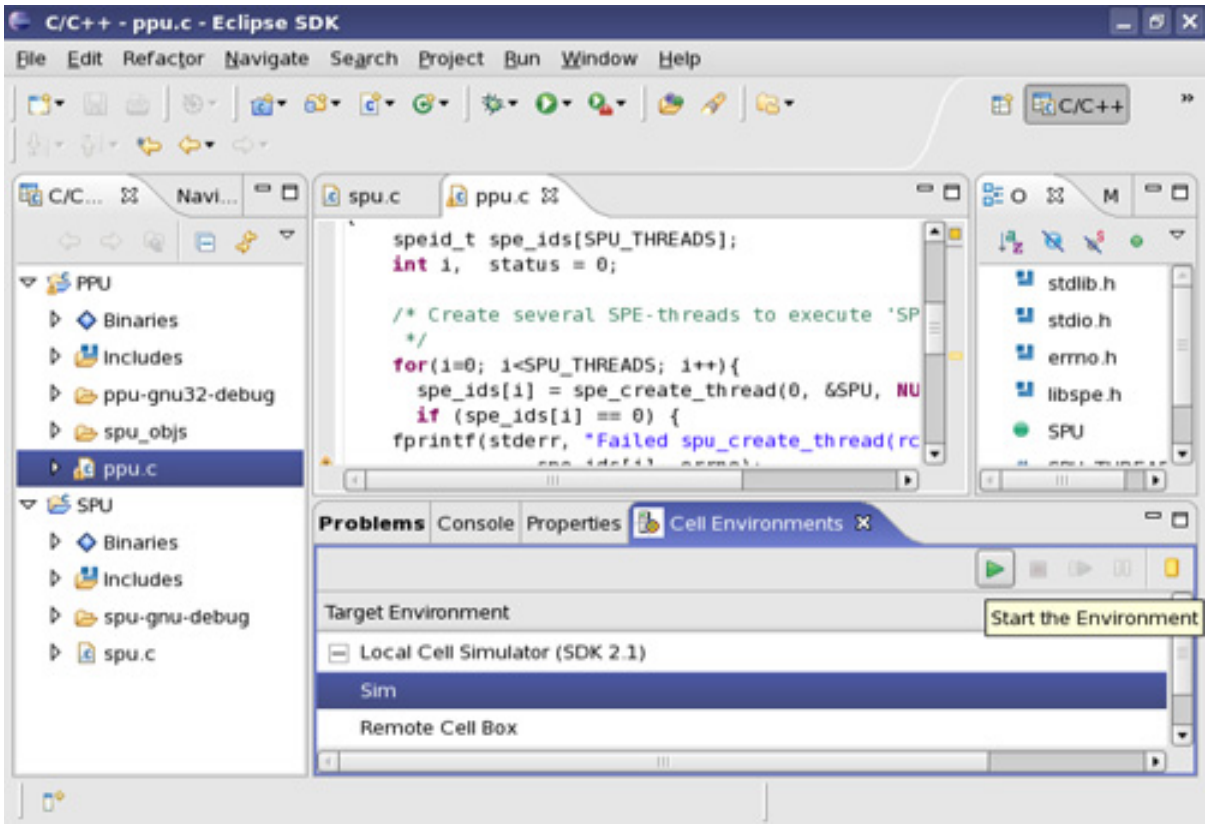
Figure 33. Simulator configuration



Start simulator

Click on the **+** next to Local Cell Simulator. The newly created simulator configuration is now visible. Select the simulator configuration, and click on **Start the Environment** (green arrow).

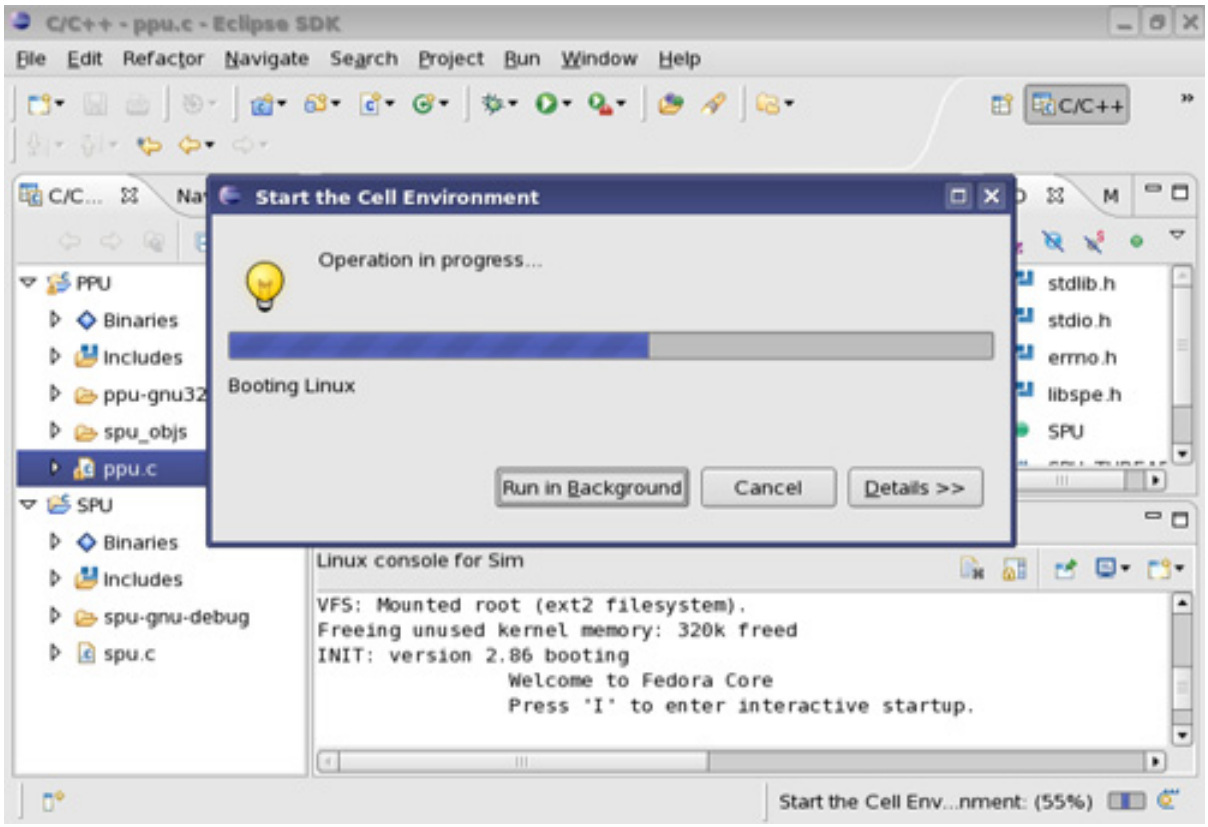
Figure 34. Start simulator



Simulator starting

The simulator launches now (this may take a few moments).

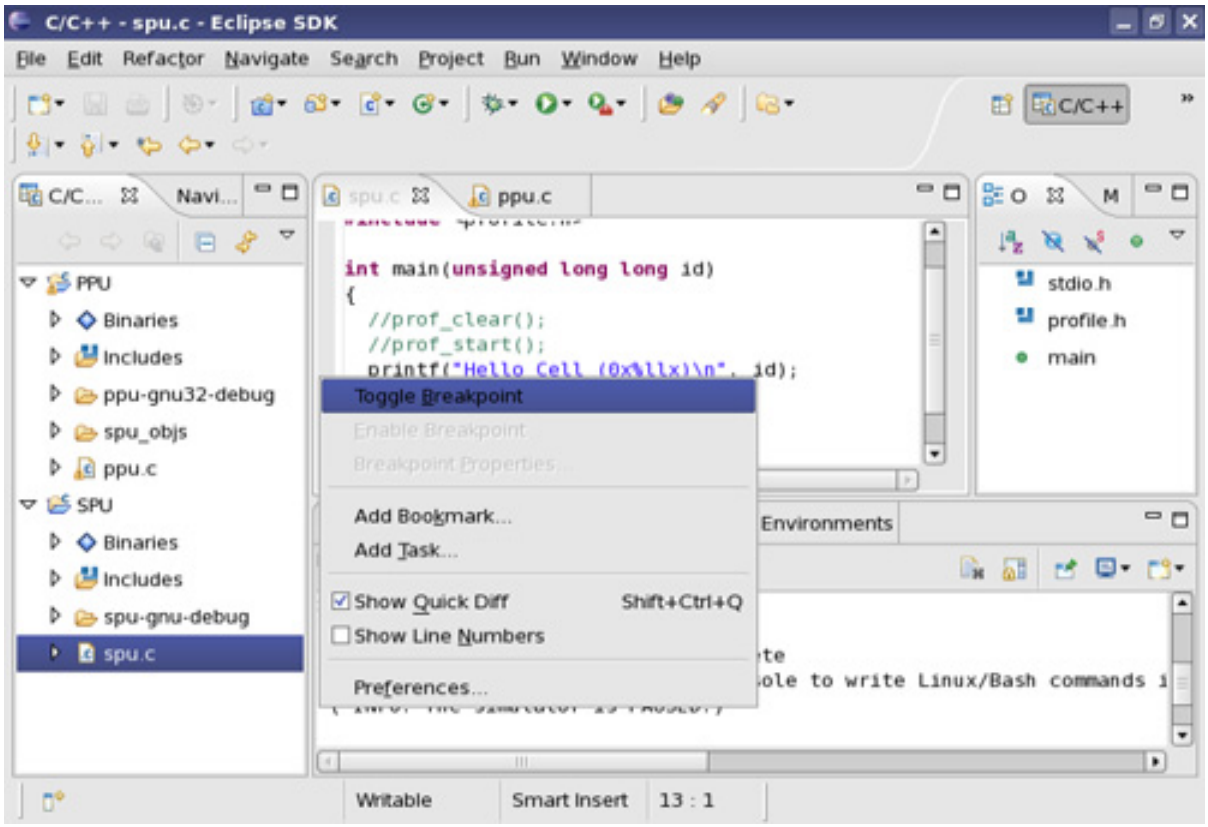
Figure 35. Simulator starting



Set a breakpoint

You can change numerous settings and options by right clicking on the gray column just to the left of the source code editor. Open the editor for the spu.c source file, and set a debug breakpoint by right clicking the gray column to the left of the printf statement, then select **Toggle Breakpoint**.

Figure 36. Set a breakpoint

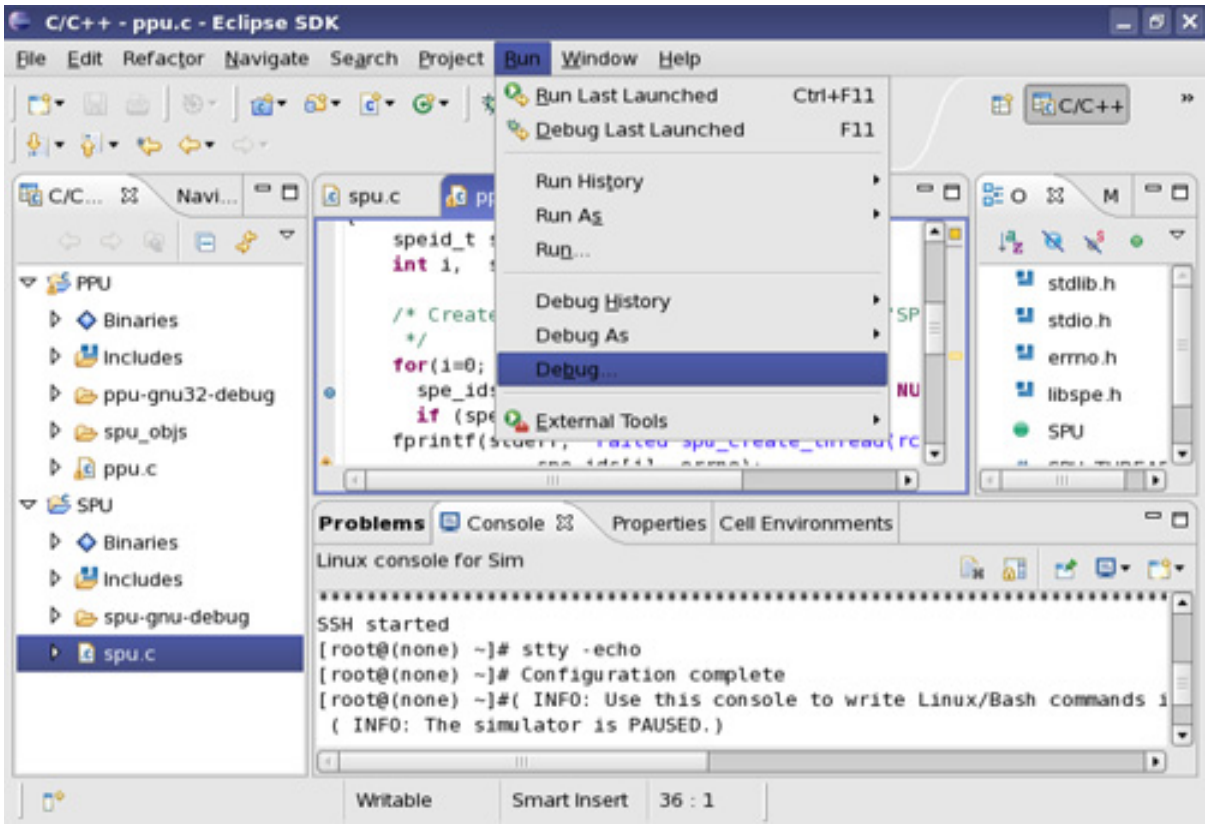


Create and configure a launch configuration

Switch back to the ppu.c source file editor, and set a breakpoint to the left of the line: spe_ids[i]=spe_create_thread…

A C/C++ Cell Application launch configuration needs to be created and configured. To do this, click **Run > Debug…**

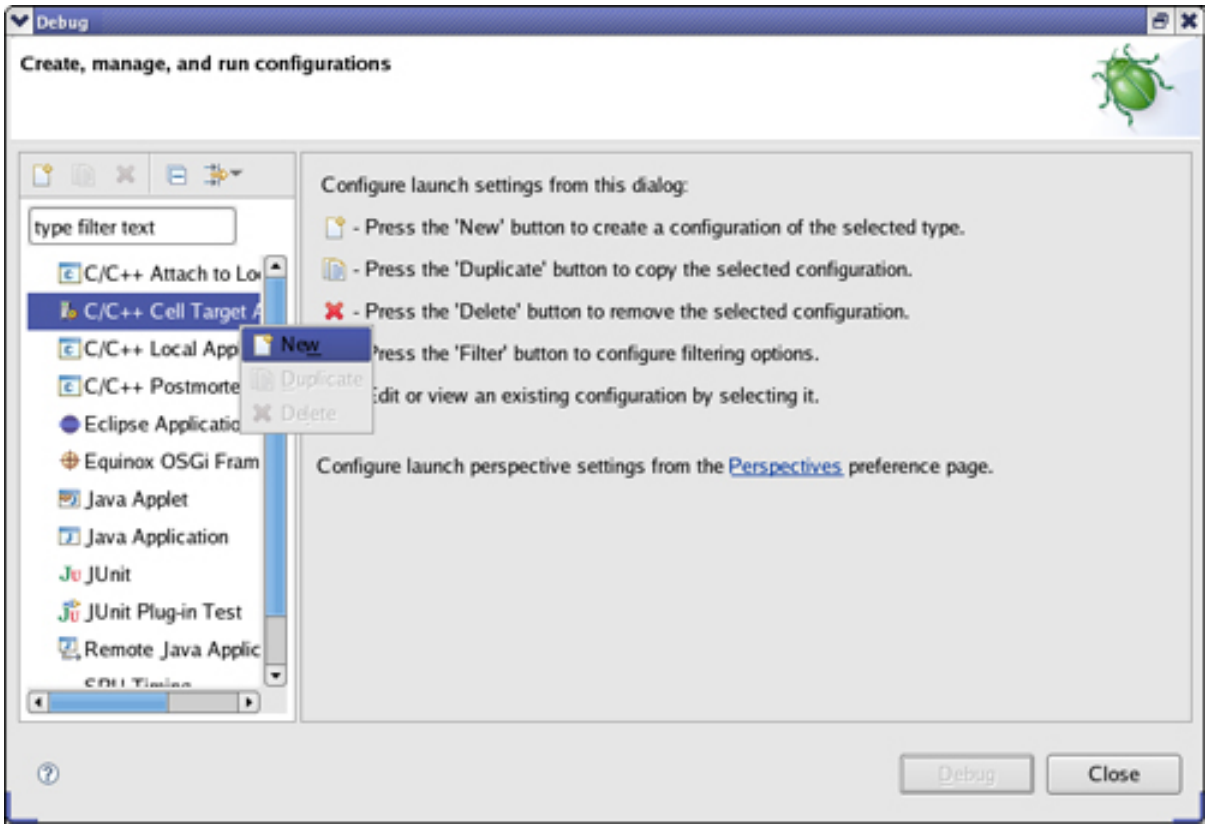
Figure 37. Create and configure a launch configuration



Create new C/C++ Cell Target Application configuration

In the left pane, right click on **C/C++ Cell Target Application**, and select **New**.

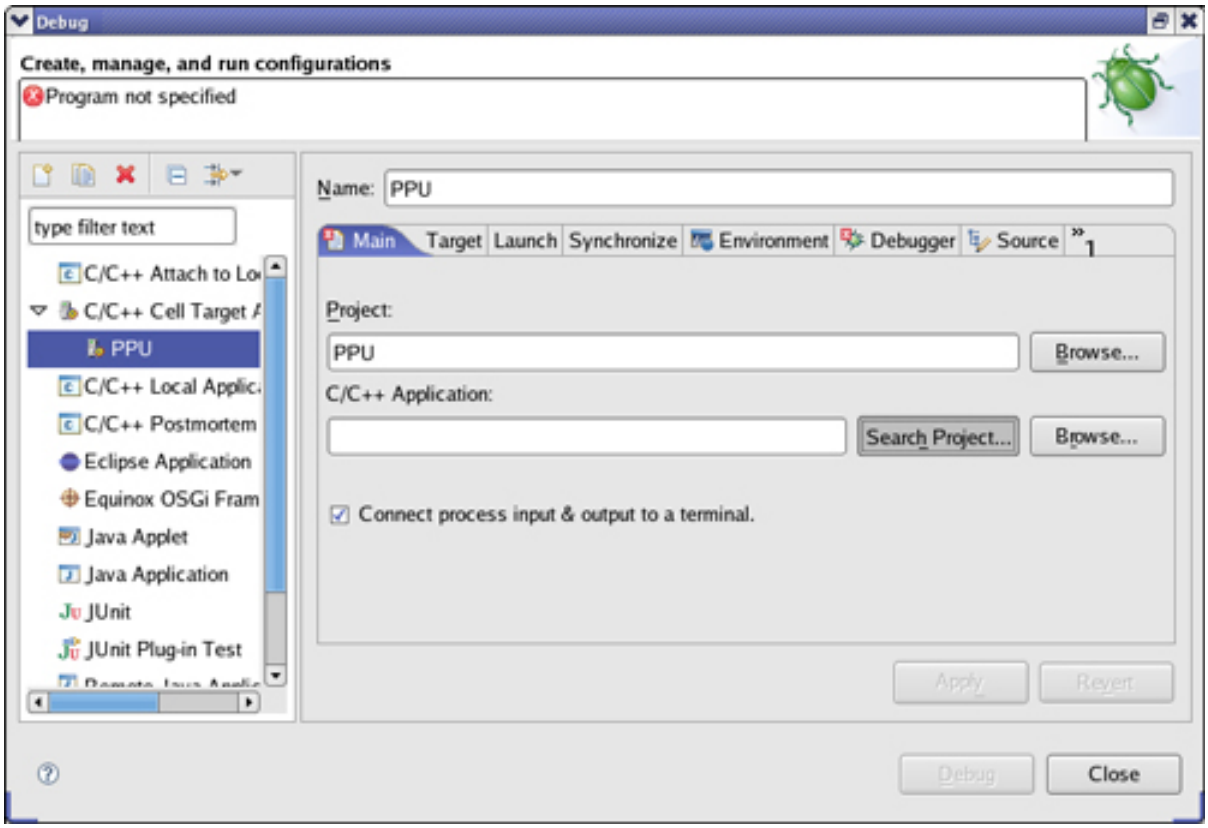
Figure 38. Create new C/C++ Cell Target Application configuration



Modify debug configuration

In the Main tab of the debug configuration, ensure that project PPU is in the Project field. For the C/C++ Application field, click **Search Project...**.

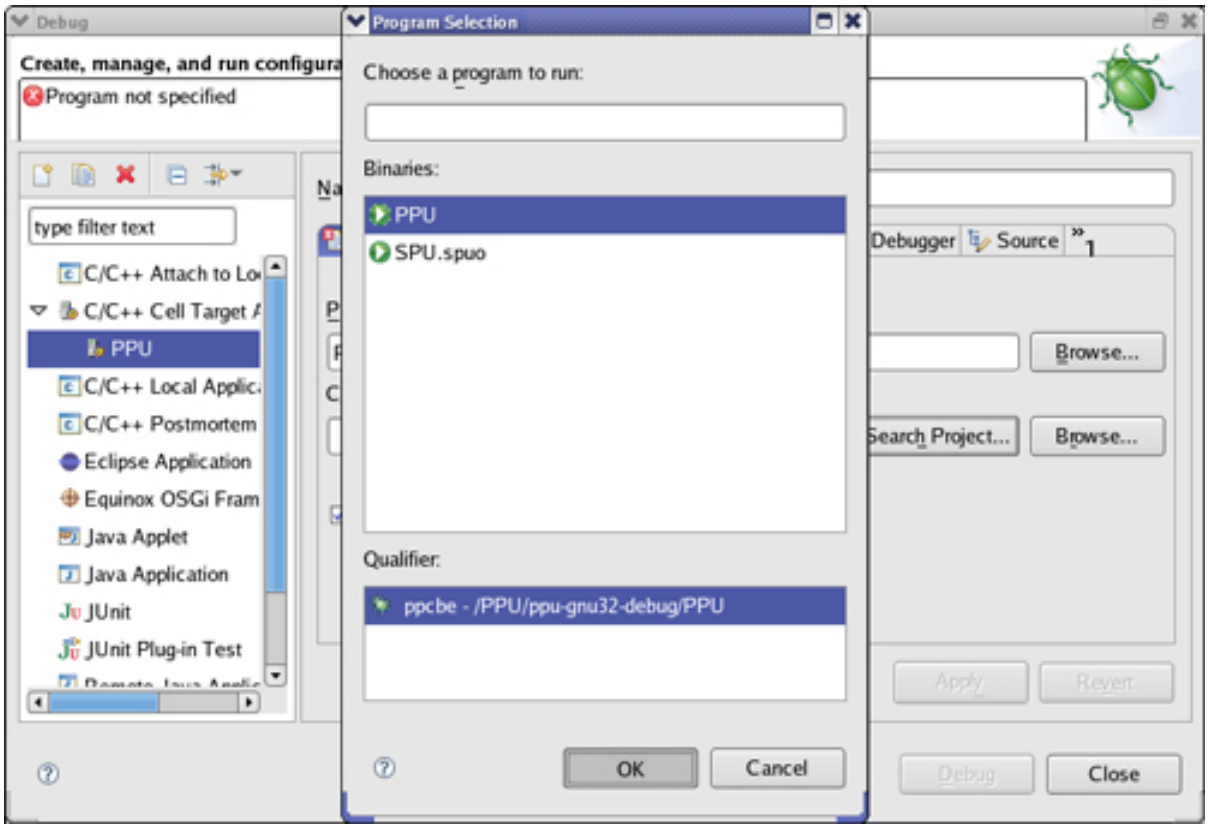
Figure 39. Modify debug configuration



C/C++ application selection

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, we have not changed the default build configuration, so only one Qualifier is currently available. Select **PPU** for the Binary, and click **OK**.

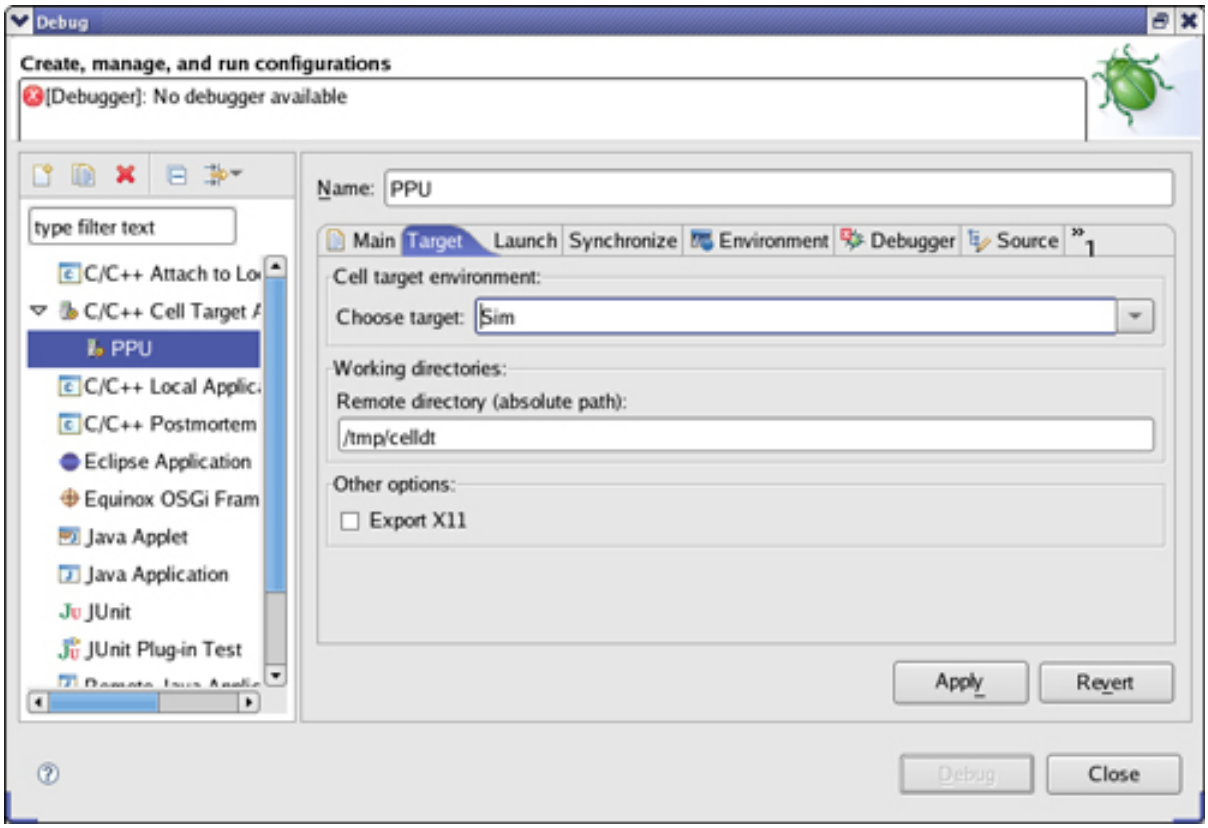
Figure 40. C/C++ application selection



Target selection tab

Navigate to the Target Selection tab. For the Choose Target field, select the simulator that was just created, then go to the Launch tab.

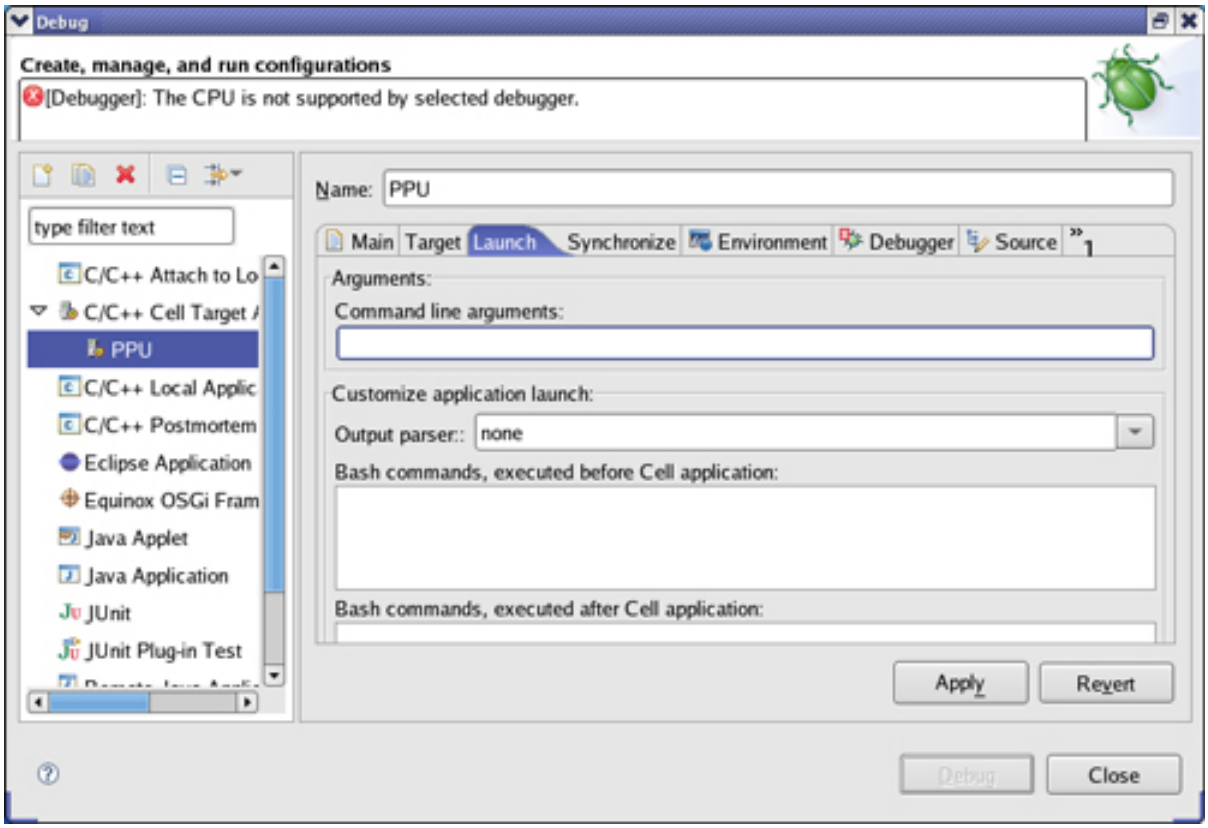
Figure 41. Target selection tab



Launch tab

In this tab you can specify command line arguments, as well as bash commands that will be executed before or after the Cell application executes.

Figure 42. Launch tab

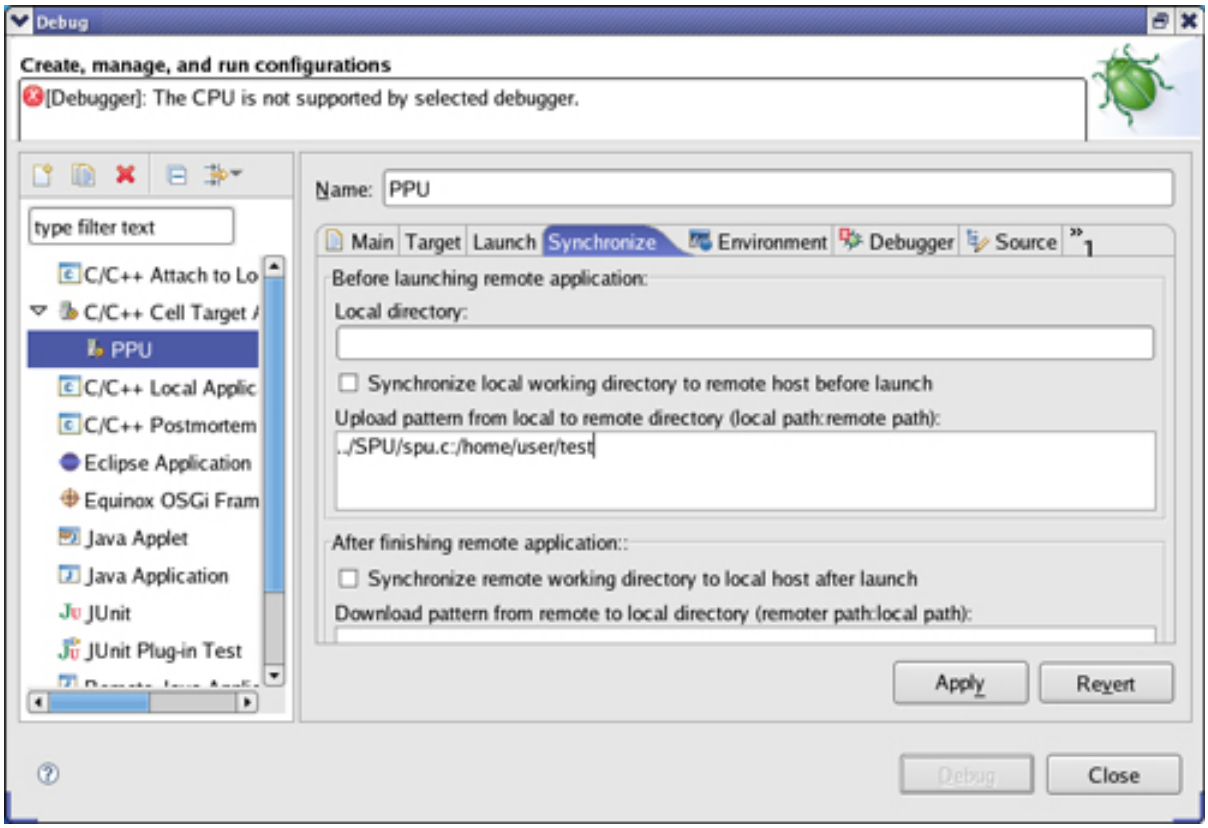


Synchronize tab

Here you can specify resources (such as input and output files) that need to be synchronized with the Cell environment's file system before or after the Cell application executes. To synchronize resources, use the following pattern: `local_path:remote_path`, where the local_path is relative (in this case) to the PPU project's path (or whichever project is selected in the Main tab). For example, to synchronize the source file spu.c with the Cell file system before program execution, you could type: `../SPU/spu.c:/home/user/test`.

Now go to the Debugger tab.

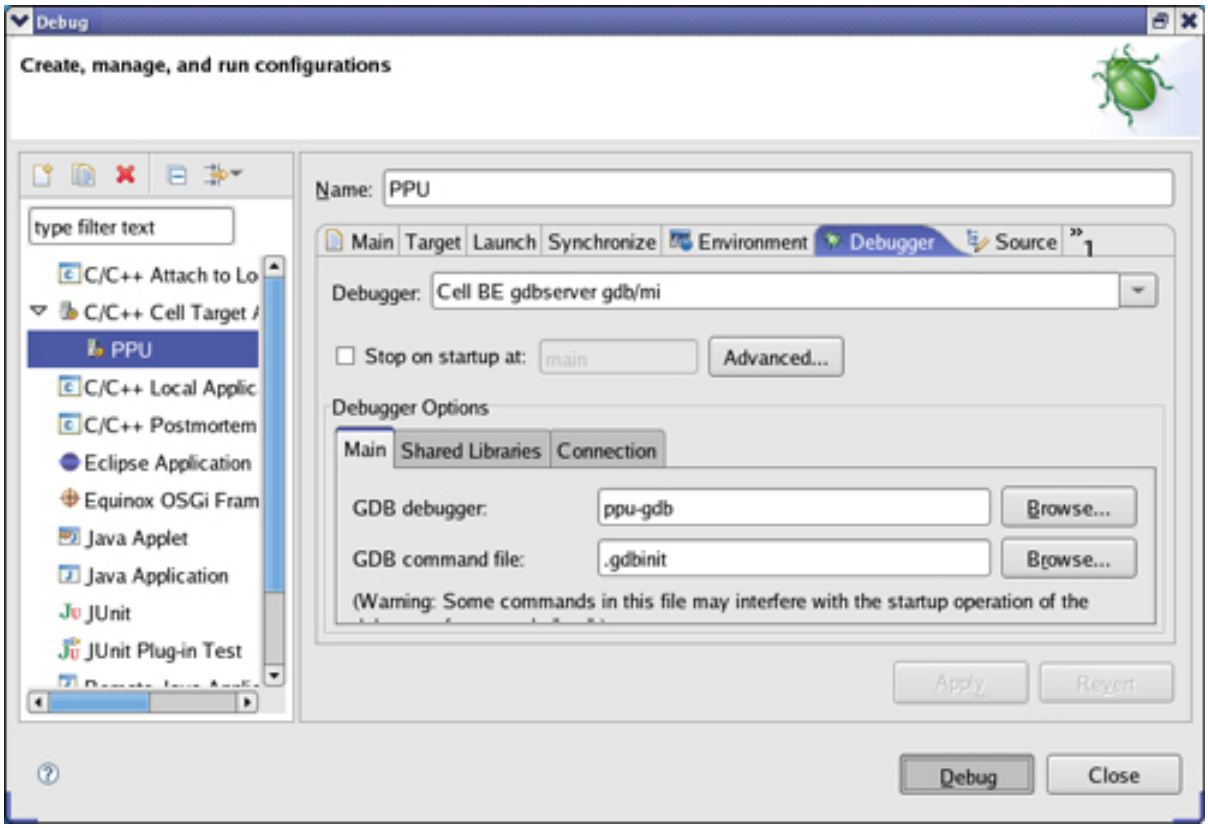
Figure 43. Synchronize tab



Debugger tab

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** or **Cell SPU gdbserver**, respectively. To debug both PPU and SPU projects at the same time, select **Cell BE gdbserver**, which is the combined debugger. Set the Debugger field to **Cell BE gdbserver**, uncheck **Stop on startup at: main**, then click **Debug** to launch the debug session.

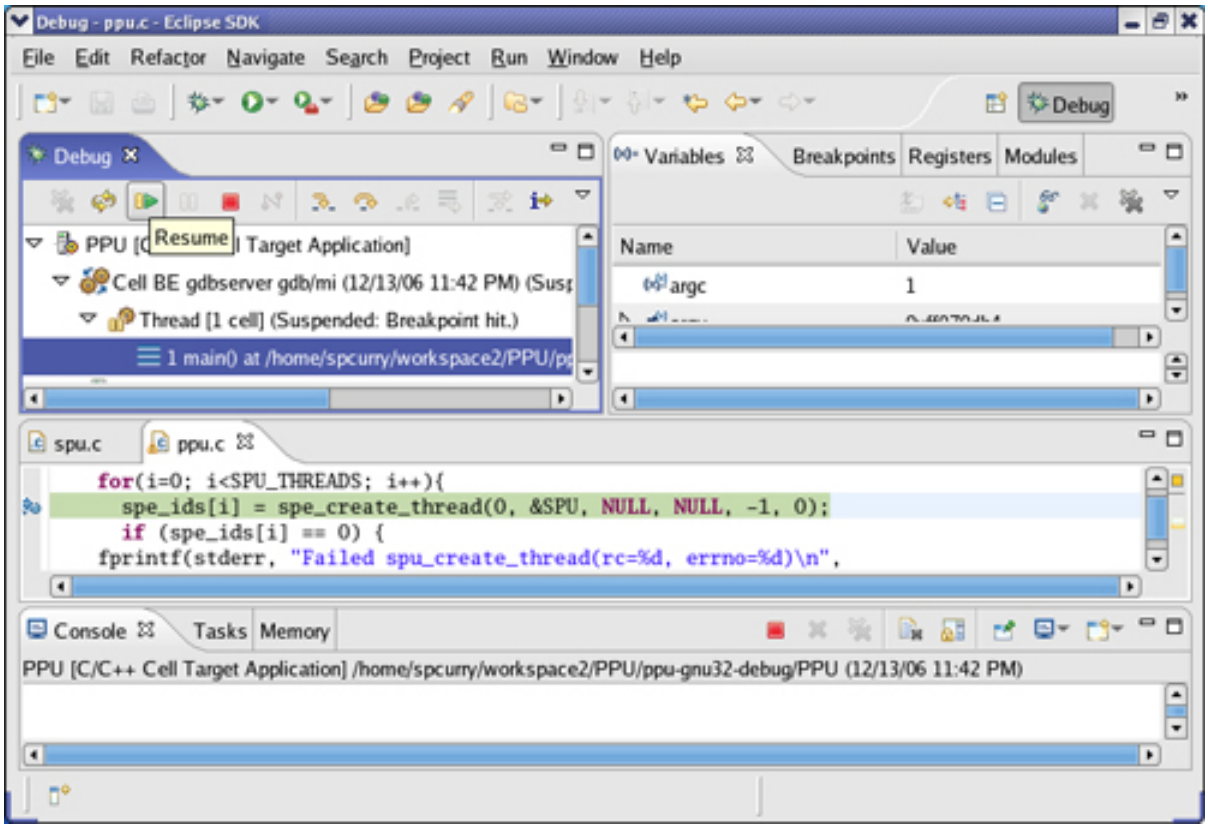
Figure 44. Debugger tab



The debug perspective

With the debug perspective you can step through your source code to find and correct bugs. This perspective also allows you to view variables and registers, along with their current values, and much more. Click on **Resume** to execute the spe_create_thread call, which will cause a second thread to be created for the spu.c program.

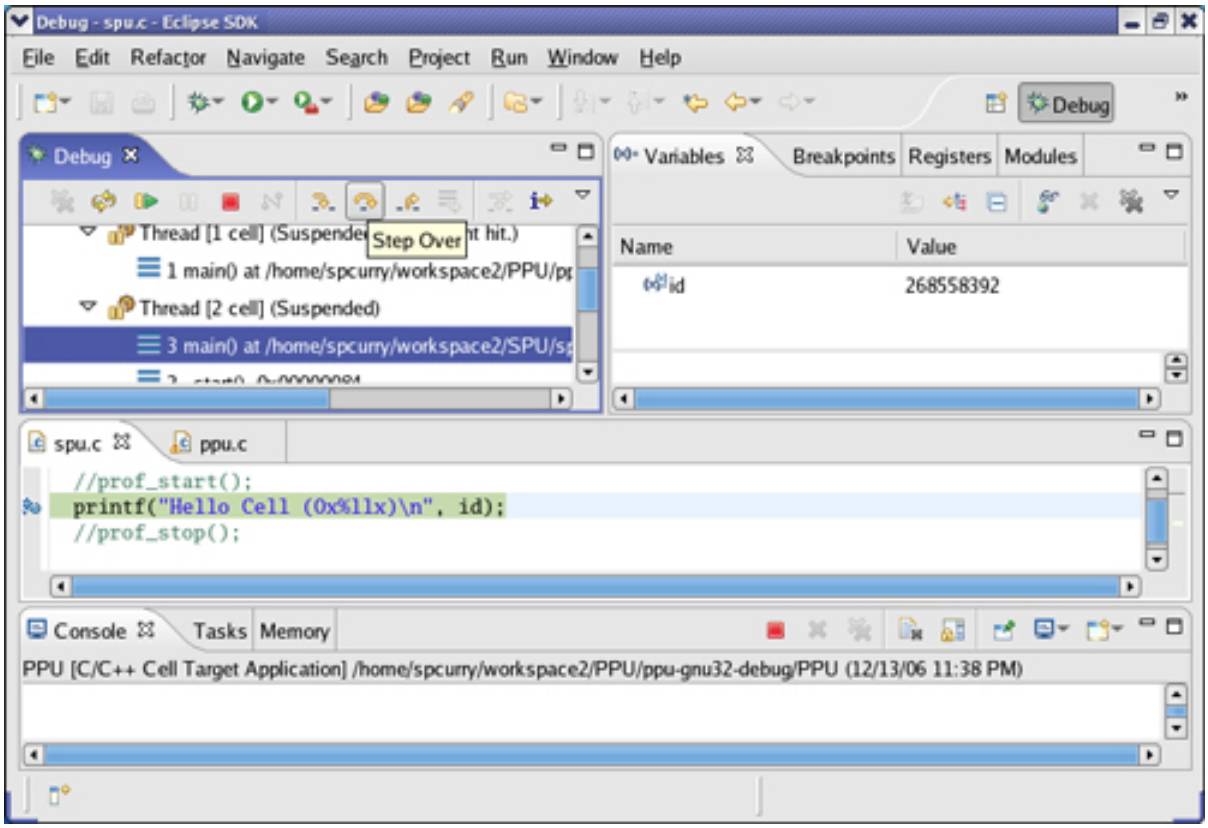
Figure 45. The debug perspective



SPU thread

In the Debug view, a second thread (Thread [2 cell]) has been created. Expand this thread and select **main()**. Execution has stopped in the spu.c source file because of the spe_create_thread call (and because of the breakpoint we set). Click **Step Over** to step through the spu.c program.

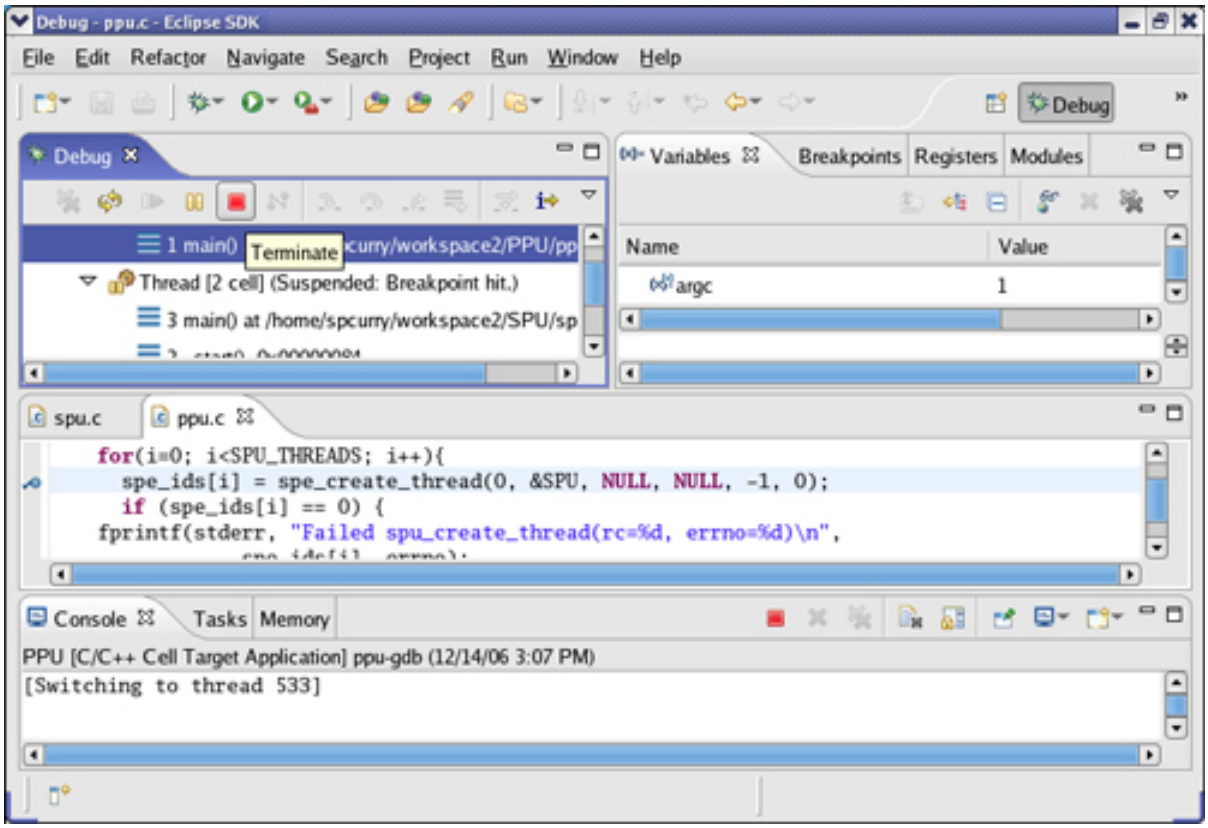
Figure 46. SPU thread



Terminate execution

When you are ready to terminate the debug session, click on the **Terminate** icon (red square, towards the top).

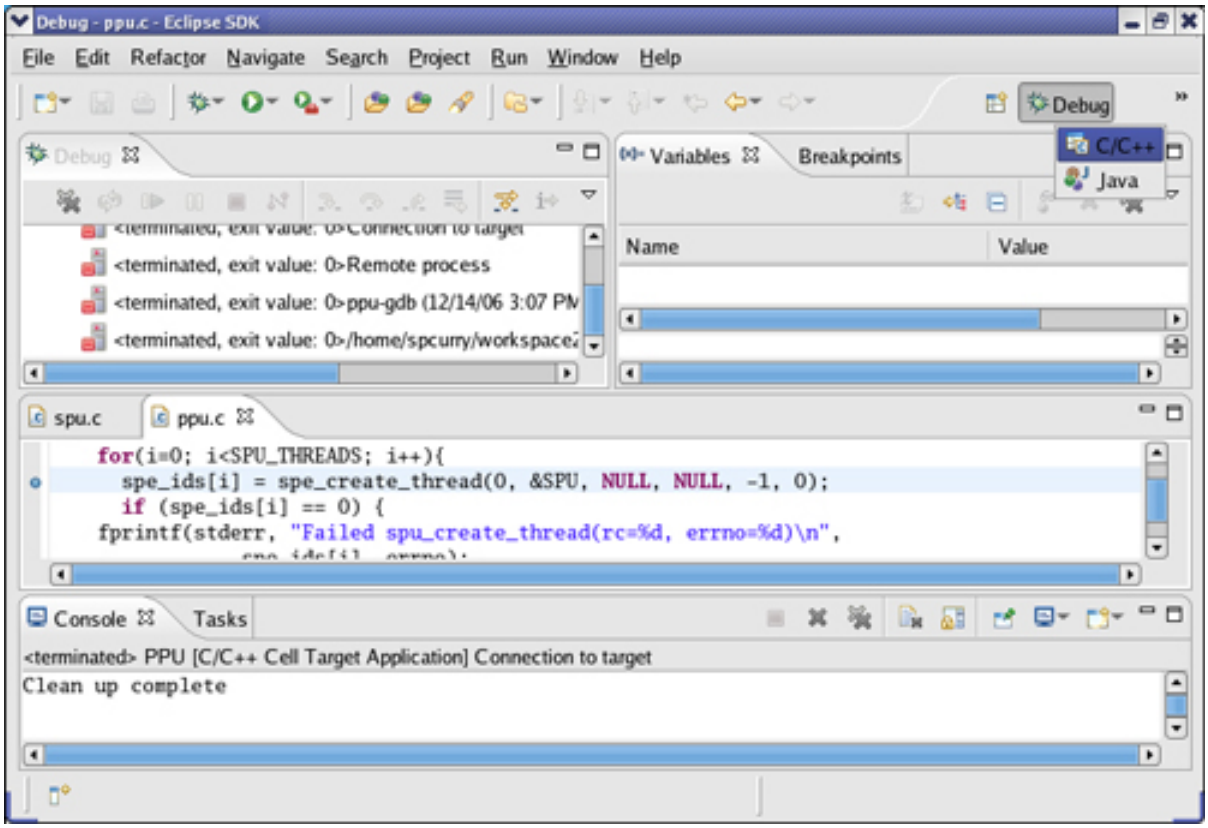
Figure 47. Terminate execution



C/C++ perspective

Switch back to the C/C++ perspective by clicking on the **double right arrows** at the top right and selecting **C/C++**.

Figure 48. C/C++ perspective

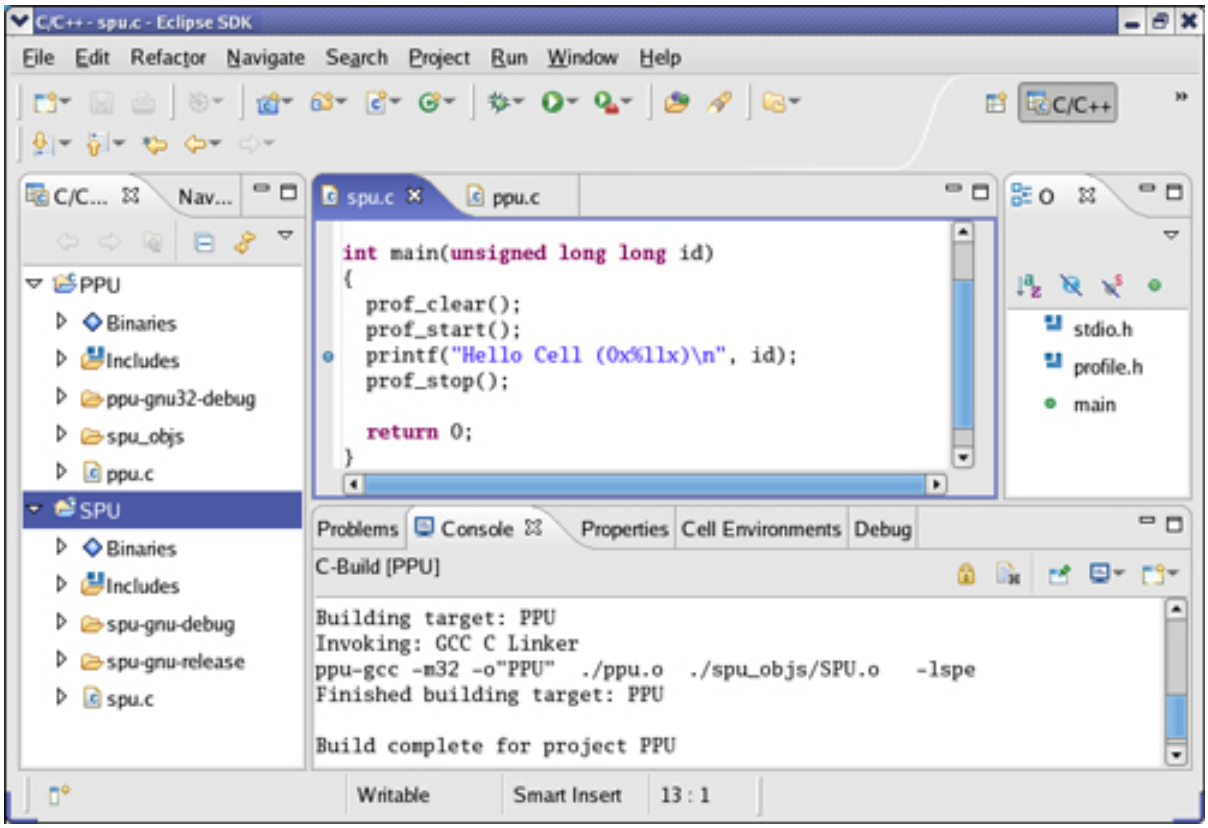


Section 7. Using the static/dynamic profiling tools

Using dynamic profiling

To use dynamic profiling, you must include profile.h in your SPU source code, use the three timing functions prof_clear(), prof_start(), and prof_stop(), and the SPU cores must be set to Pipeline mode. First, open the spu.c source file editor, and uncomment the 3 lines: prof_clear();, prof_start();, and prof_stop();

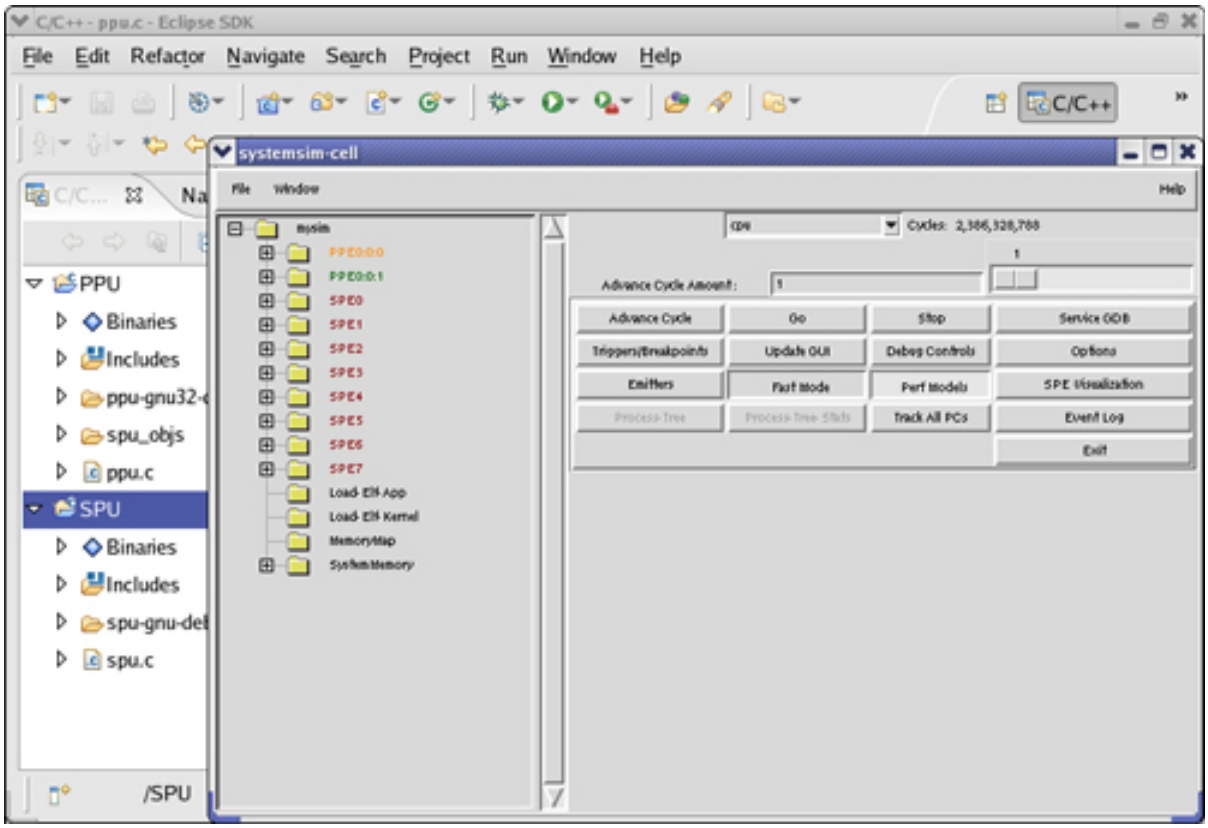
Figure 49. Using dynamic profiling



Change SPU modes

To use dynamic profiling, the SPUs must be in pipeline mode. Open the Simulator GUI window (systemsim-cell), and click on **Perf Models**.

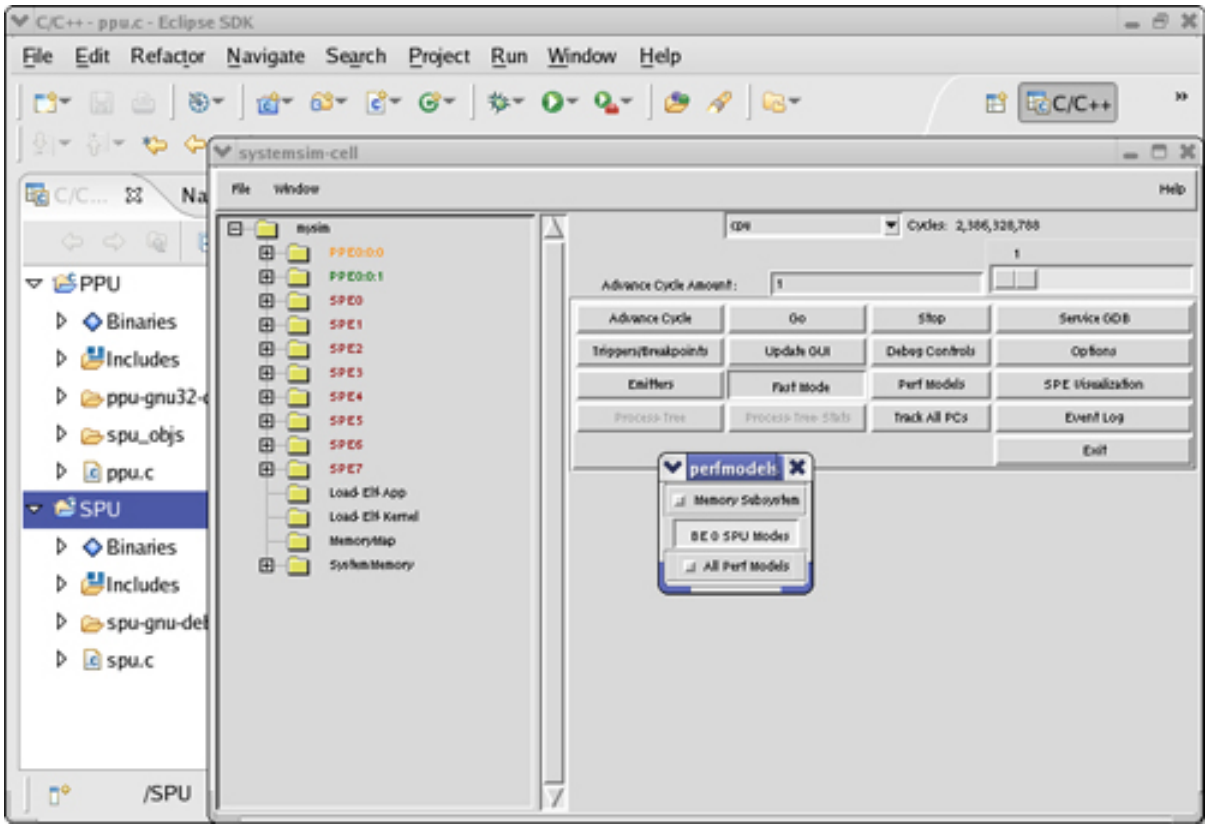
Figure 50, Change SPU modes



Performance models

Click **BE 0 SPU Modes**.

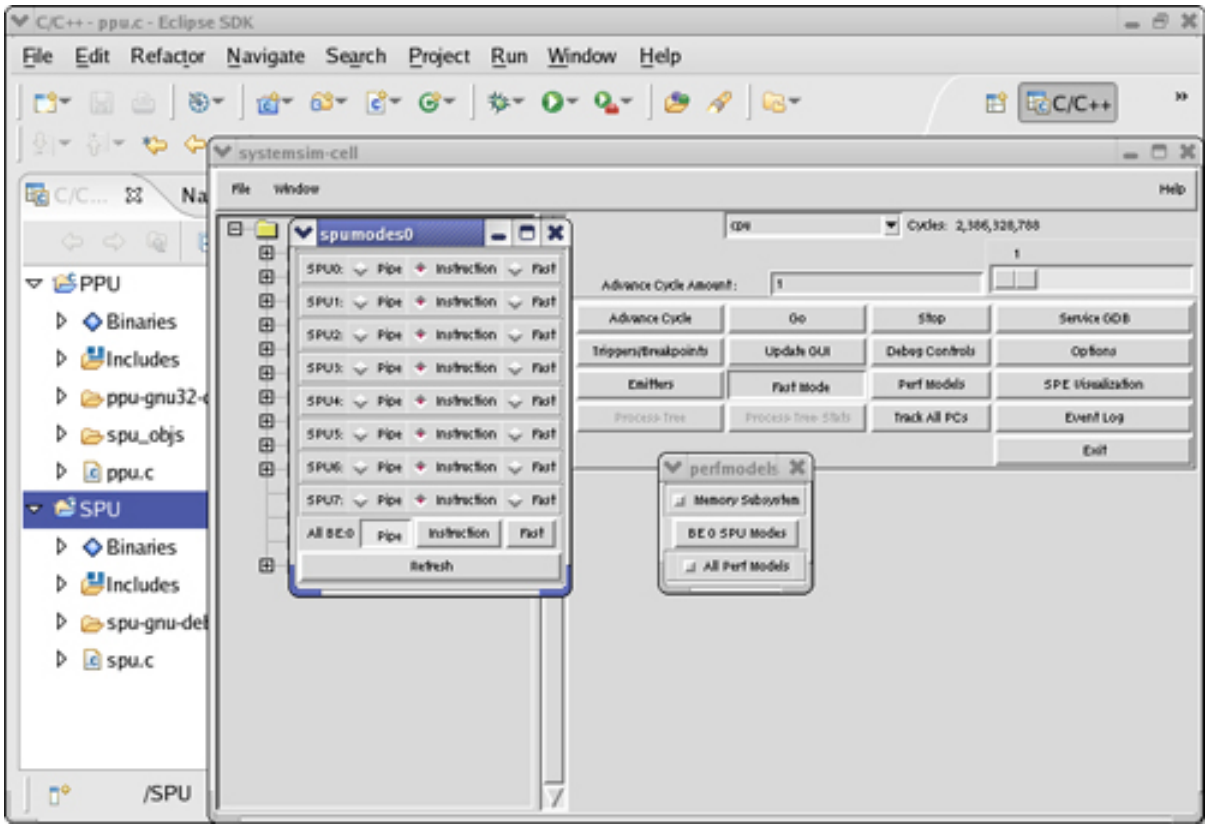
Figure 51. Performance models



Set SPU modes to pipeline

To change all of the SPUs to pipeline mode, click **Pipe**, then go back to Eclipse.

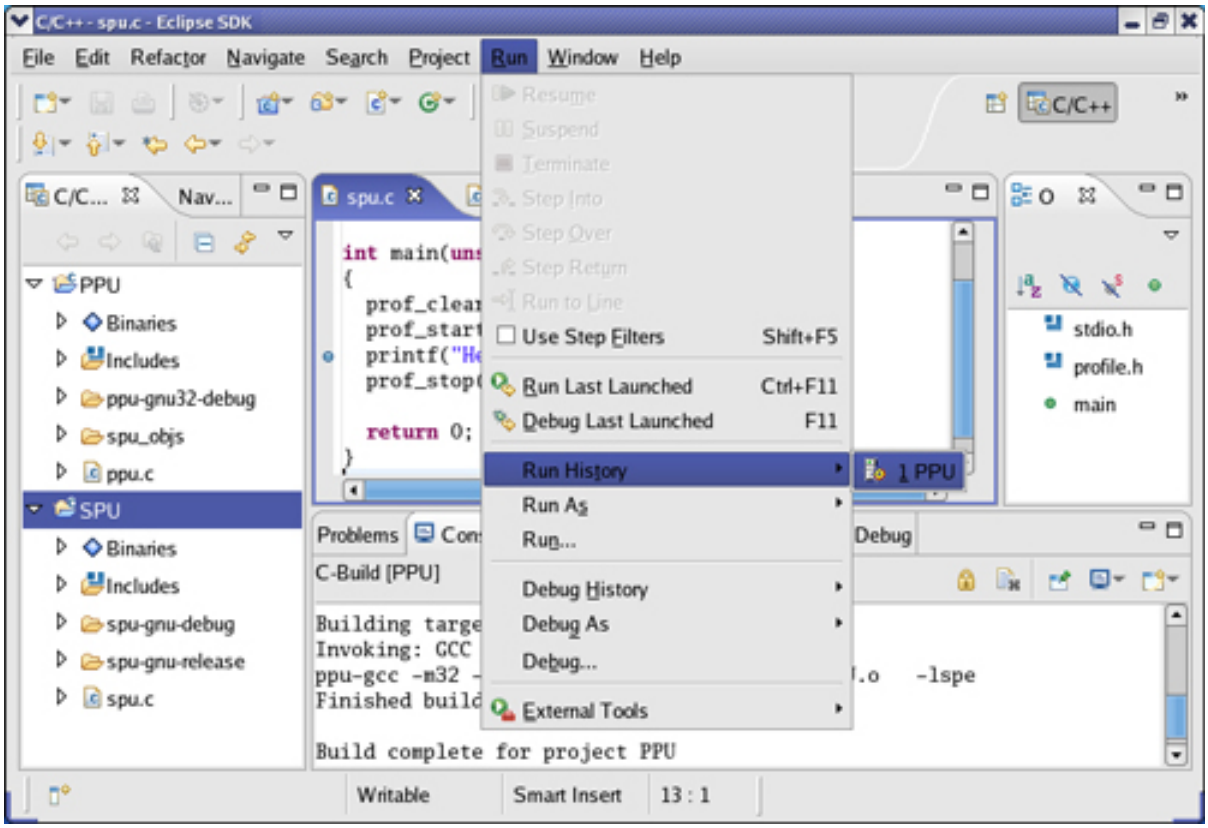
Figure 52. Set SPU modes to pipeline



Run the Cell launch configuration

Now that the profiling functions are in the spu.c source file and the SPUs are set to pipeline mode, we can launch the application so the performance results can be recorded and viewed. Click **Run > Run History > PPU**.

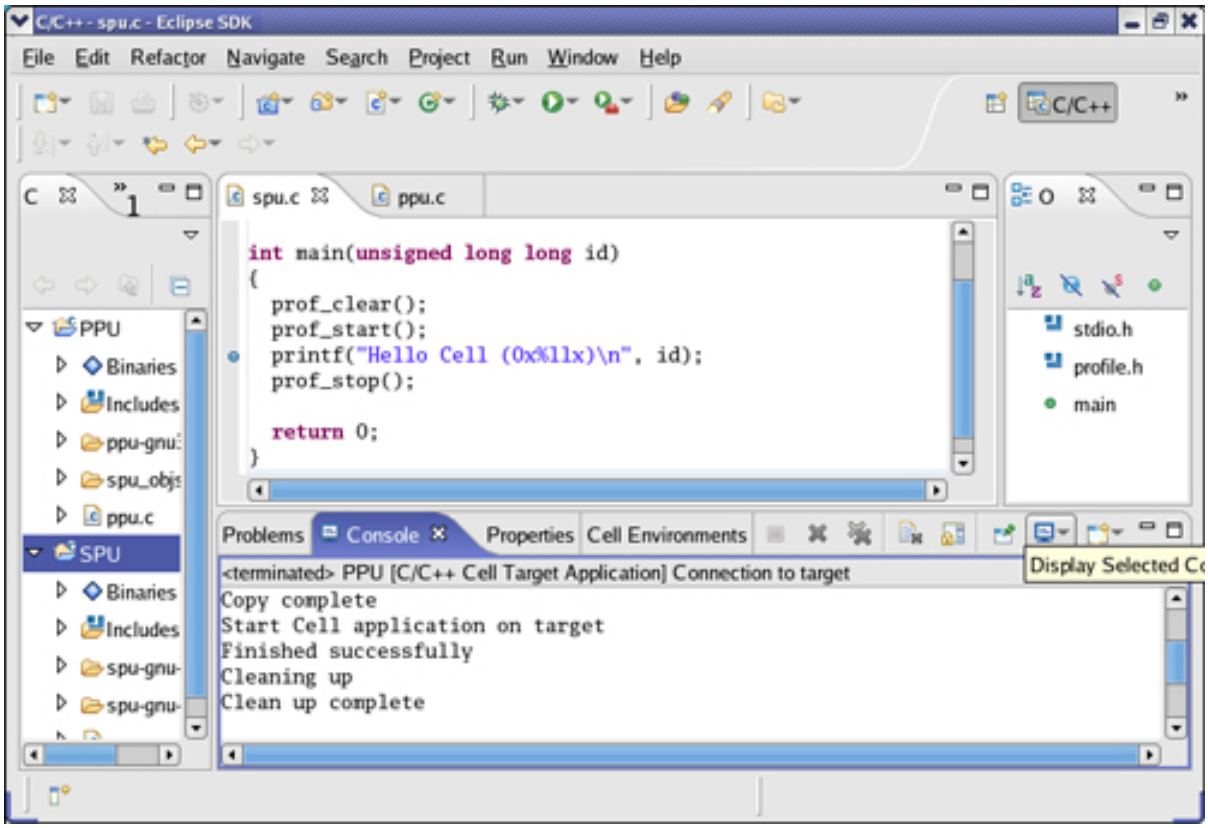
Figure 53. Launch the Cell launch configuration



Switch console

On the Console view's toolbar, you will see a little blue monitor icon. You can use this icon to switch between the available consoles. Click on the **down arrow** to the right of this icon.

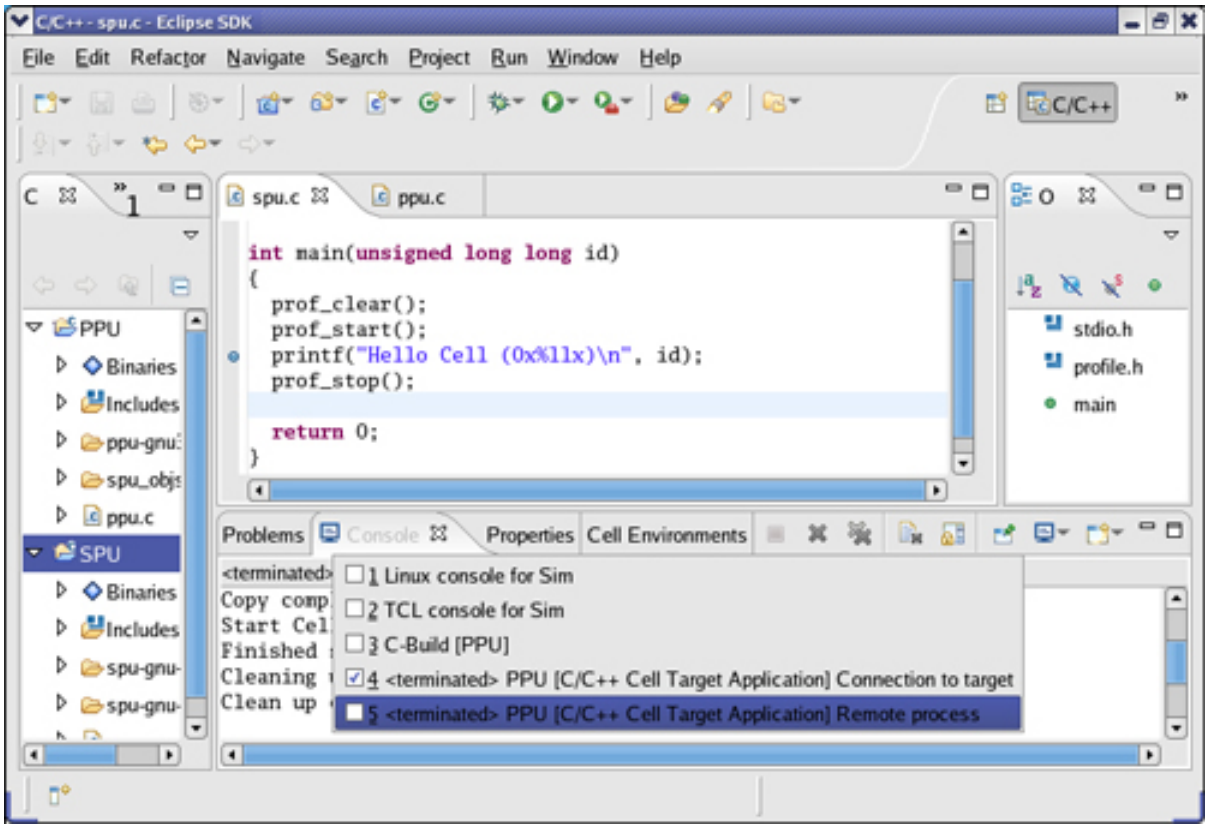
Figure 54. Switch console



Open remote process console

Select the **Remote Process** console to view the output of running the Cell application.

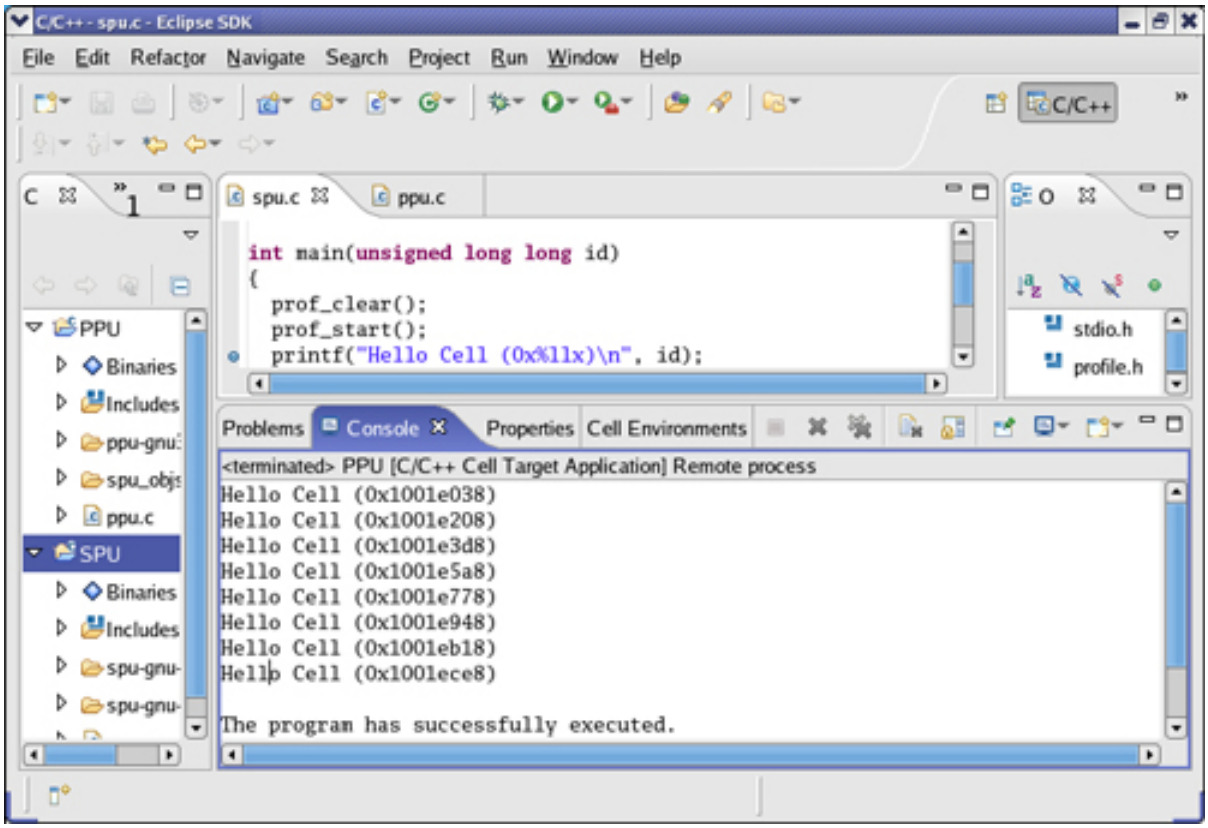
Figure 55. Open remote process console



Cell application output

Now in the Console view, you see `Hello Cell` printed eight times, once per SPU core.

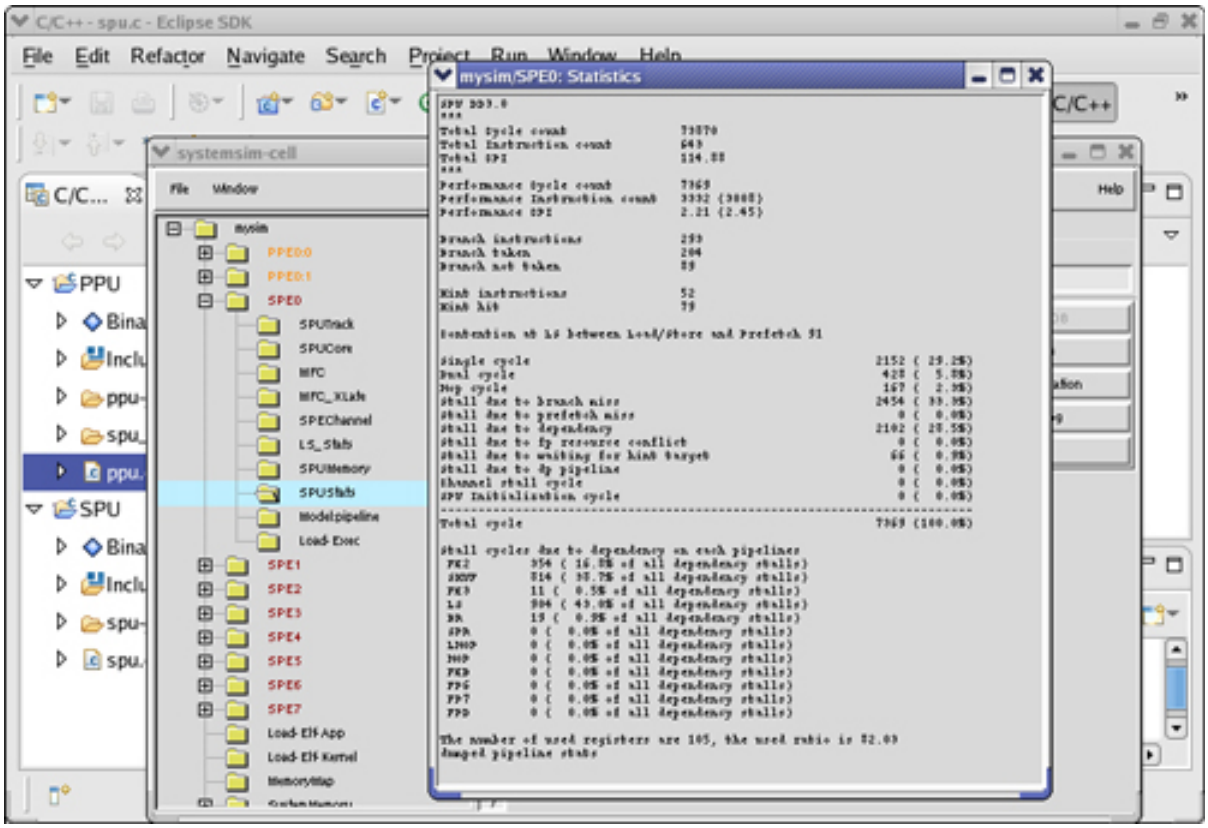
Figure 56. Cell application output



Dynamic profiling statistics

Open the Simulator GUI window, expand the category **SPE0**, and open **SPUStats**. These are the results of the dynamic profiling tool. These results, when used in conjunction with static profiling (covered next), can be used to locate and optimize code segments that might not be performing as well as you had hoped.

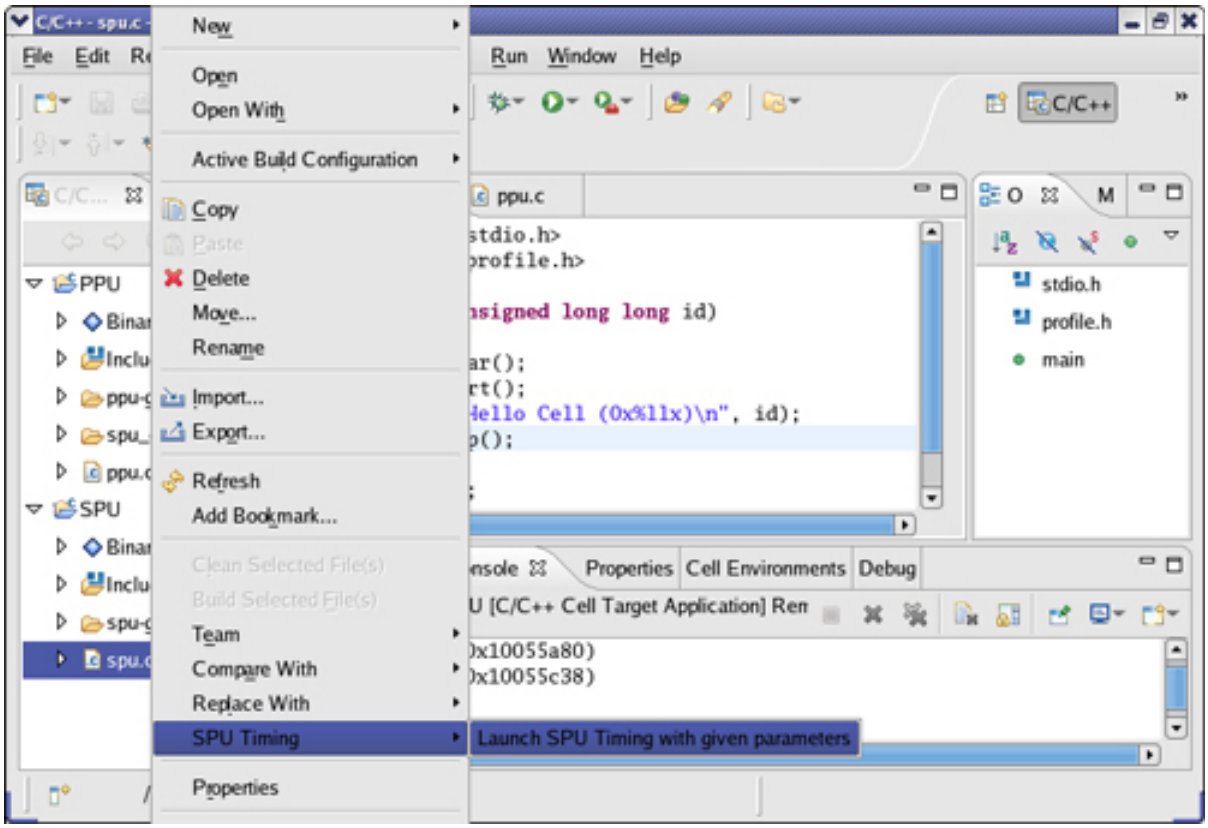
Figure 57. Dynamic profiling statistics



SPU timing -- static performance analysis

To launch the SPU Timing tool, right click on **spu.c**, then select **SPU Timing > Launch SPU Timing with given parameters**.

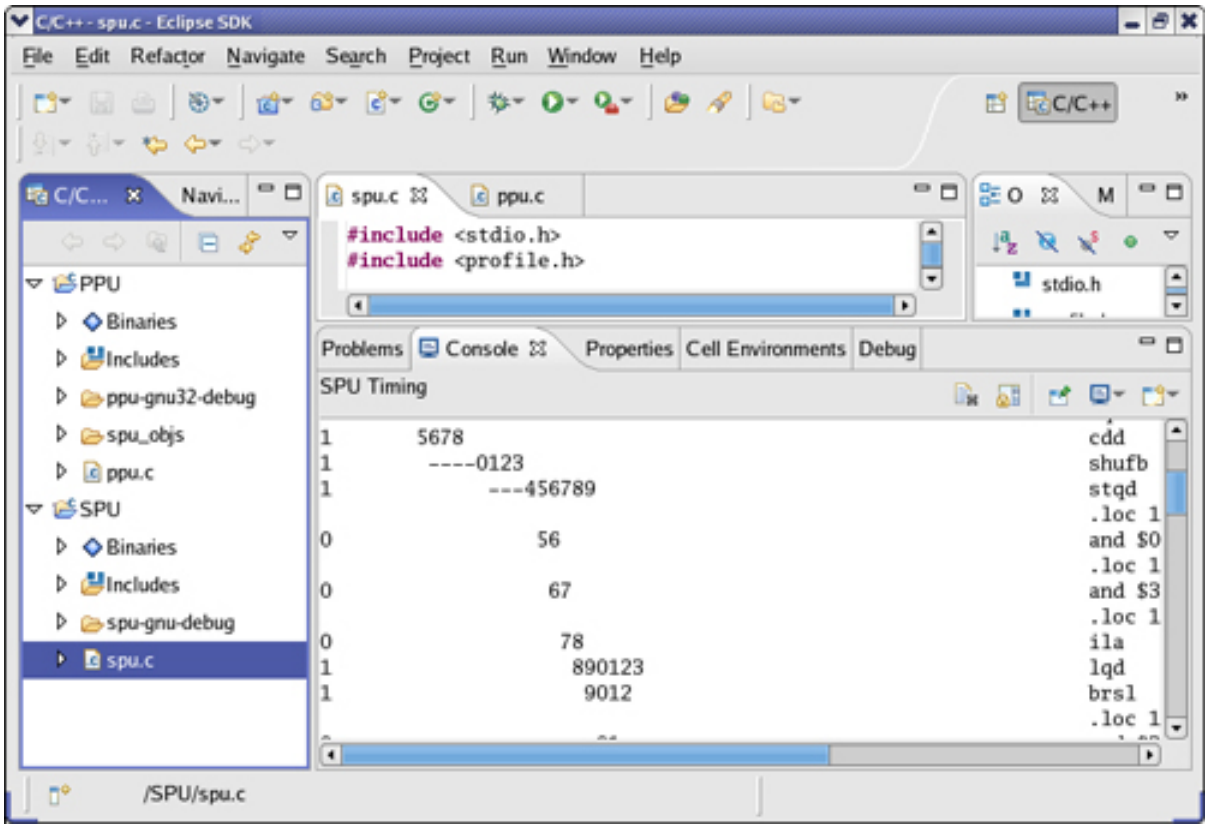
Figure 58. Using SPU timing – static performance analysis



SPU timing output

To view the output from the SPU Timing tool, open the Console view. (You may need to switch the console being displayed to the SPU Timing console, using the blue monitor icon as described before.) In the SPU Timing output, the dashes ('-') represent stalls (bad performance). Use the slope of the timing area to get a good overall indication of your program's performance.

Figure 59. SPU timing output

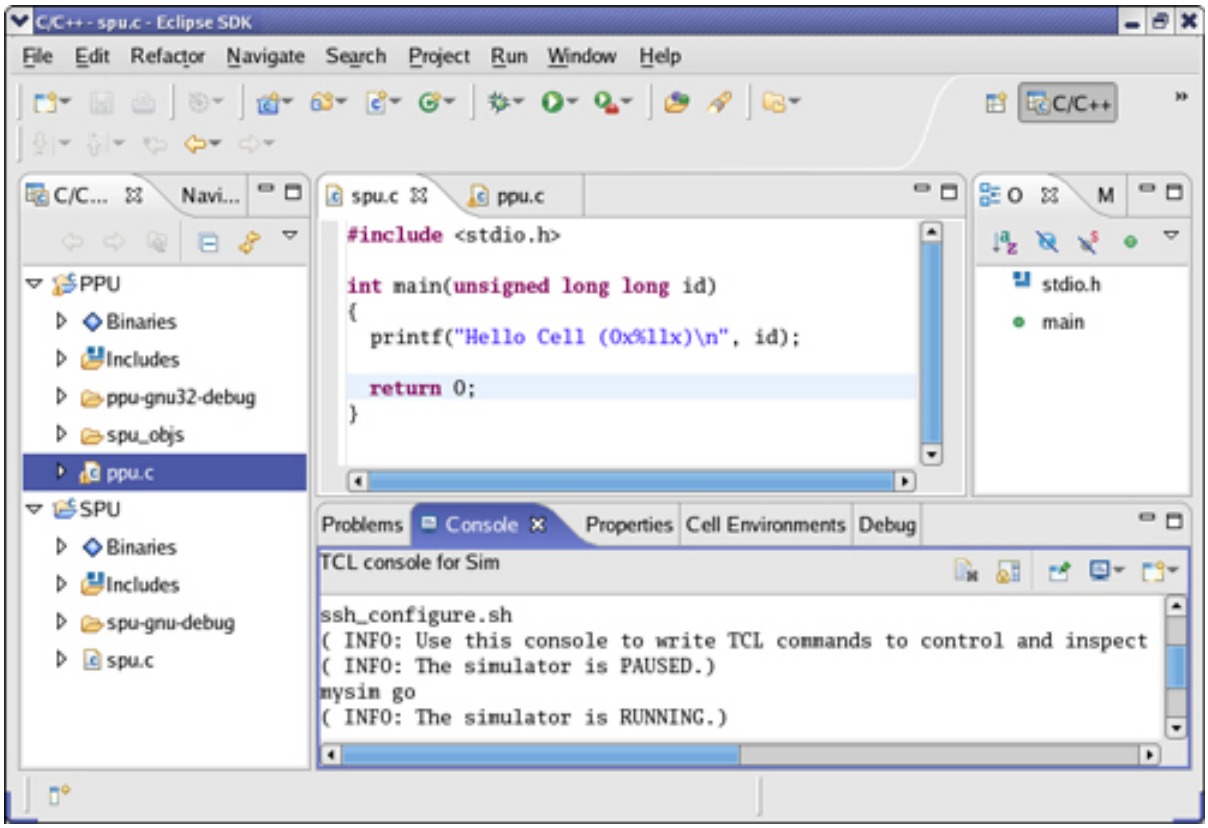


Section 8. Other tools and preferences

Simulator's TCL console

In the Console view, switch to the **TCL Console for Sim**. This console can be used to pass TCL commands directly to the simulator. For example, if the simulator is paused, rather than using the Resume button (in the Cell Environments view), you could type `mysim go` to start the simulator.

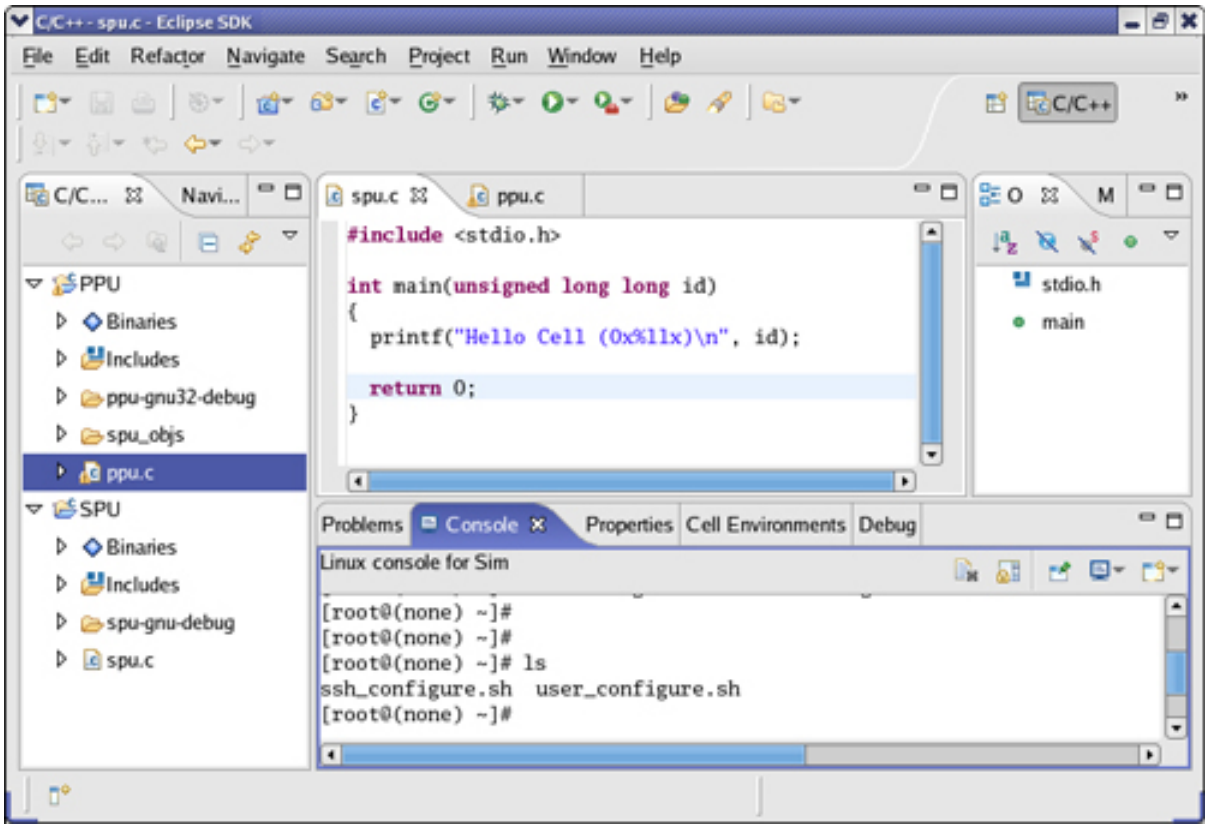
Figure 60. Simulator's TCL console



Simulator’s Linux console

Switch to the **Linux Console for Sim**. Here you can execute bash commands on the simulator.

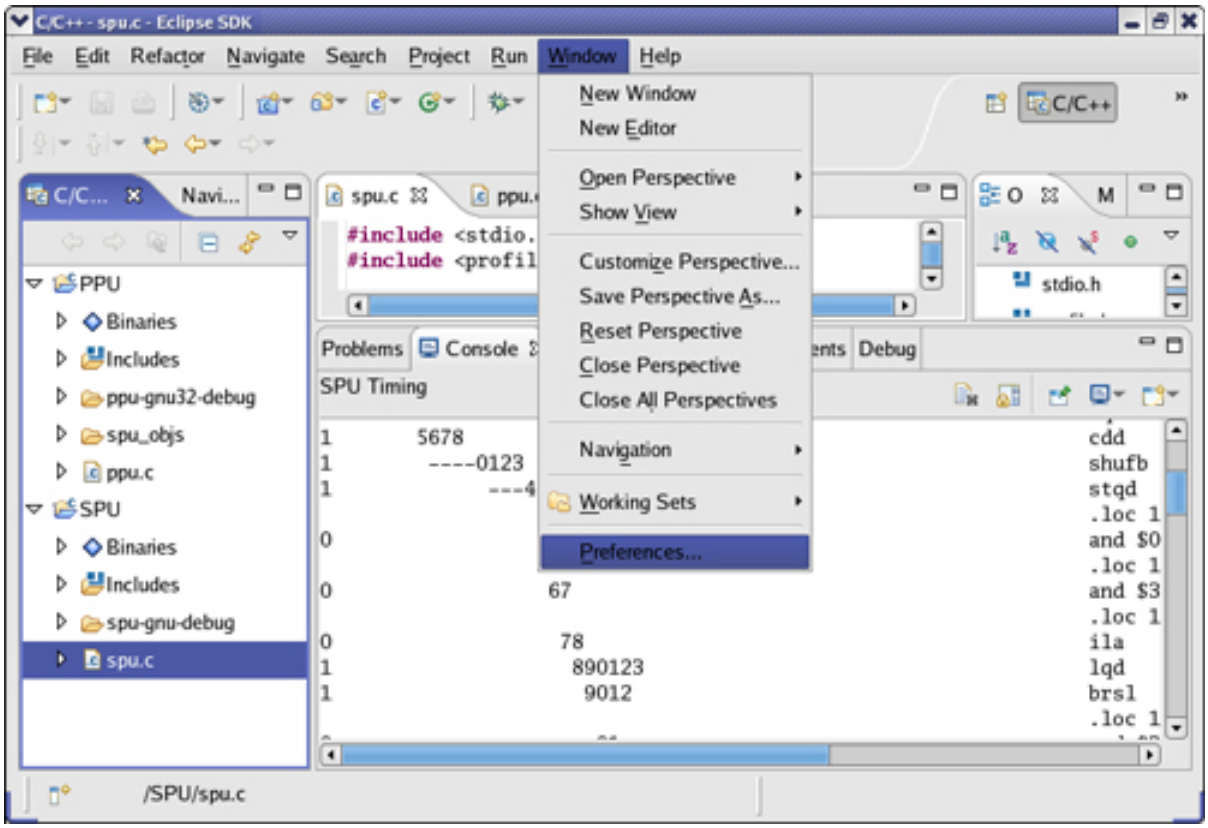
Figure 61. Simulator’s Linux console



Eclipse preferences

You can view and modify numerous Cell environment settings by visiting the Preferences page. Click on **Window > Preferences...**

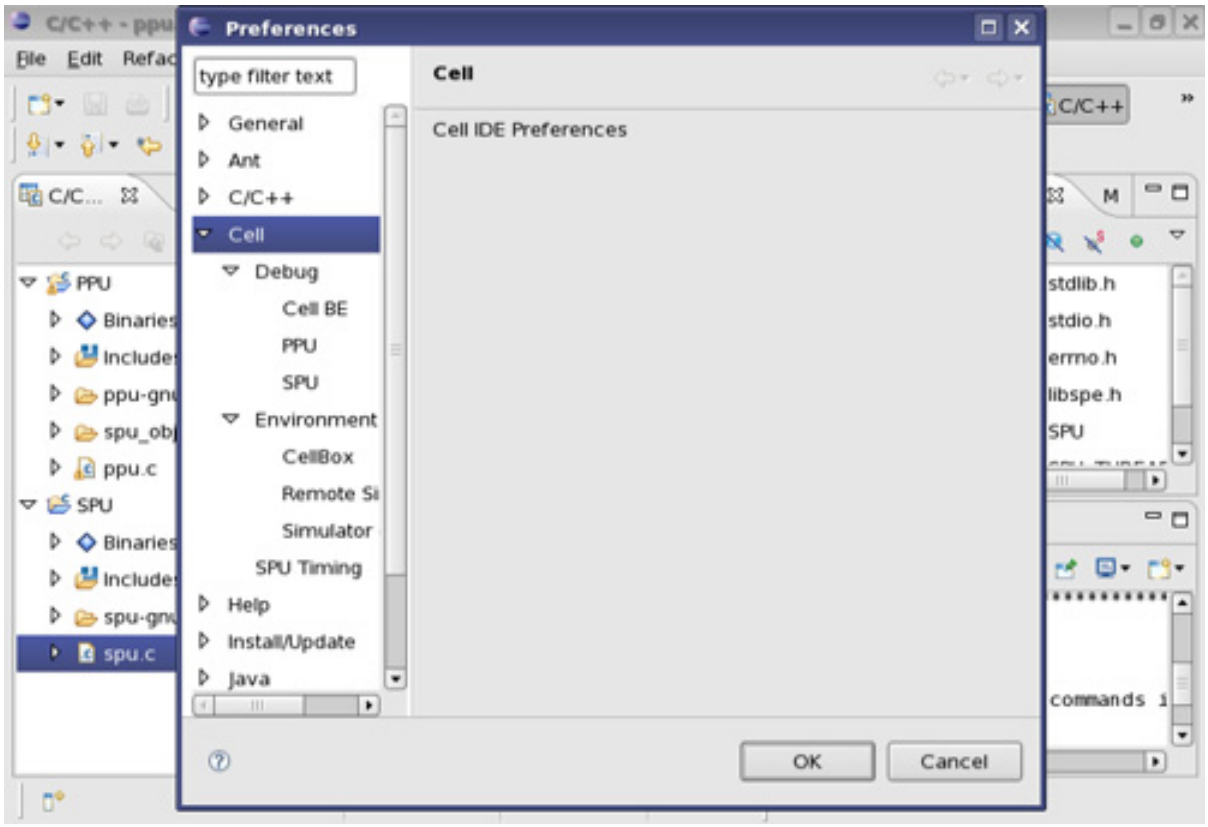
Figure 62. Eclipse preferences



Cell IDE environment preferences

On the left side, expand the Cell category. Here you can change many different options regarding the Cell IDE including PPU/SPU GDB Client Binary locations, SPU Timing Binary location, simulator path, and much more. Take a minute to look through some of the options, then click **OK**.

Figure 63. Cell IDE environment preferences



Tutorial finished!

This concludes the Cell IDE tutorial.

Downloads

- Product: Cell BE SDK
- Product: Eclipse V3.2
- Product: C/C++ Development Tools (CDT) V3.1 for Eclipse
- Product: Cell IDE

Resources

Learn

- The IBM Semiconductor solutions technical library Cell Broadband Engine documentation section lists specifications, user manuals, and more.
- Find all Cell BE-related articles, discussion forums, downloads, and more at the IBM developerWorks Cell Broadband Engine resource center: your definitive resource for all things Cell BE.
- Keep abreast of all the IBM semiconductor and Power Architecture-related news that's fit to print: subscribe to IBM microNews.

Get products and technologies

- Get the complete installation how-to for the Cell IDE.
- Get Cell BE-related downloads, including the IBM Full-System Simulator, an evaluation copy of the Visual Age XL C compiler for Cell, and the Cell SDK from IBM alphaWorks.
- Download the Cell IDE from alphaWorks.

Discuss

- Participate in the discussion forum for this content.

About the author

Sean Curry

Sean is an undergraduate student at The University of Texas at Austin, working toward a degree in Computer Sciences. As an intern at the IBM Linux Technology Center, Sean works as part of the Linux on Cell/B.E. team, specifically contributing to the IDE for Cell Broadband Engine SDK project. Sean develops source code and conducts integration testing and product release management.