Technical notes on using Analog Devices products and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Getting Started with CrossCore® Embedded Studio 1.1.x

Contributed by DH and Joe B.

Rev 1 – March 13, 2015

Introduction

CrossCore® Embedded Studio (CCES) is the latest Integrated Development Environment (IDE) from Analog Devices, Inc. (ADI), used during development of embedded applications on ADI's processors. The CCES IDE is built upon Eclipse (http://www.eclipse.org) technology, a multi-language, open-source software development environment. CCES provides complete graphical control of the edit, build, and debug processes and allows for very easy switching between them.

This EE-Note describes concepts/terminology associated with the Eclipse IDE, explains some useful CCES features, and provides a walkthrough demonstrating how to create a new application from project inception through the debug process, concluding with having a full application programmed to boot memory on an evaluation system using CCES release 1.1.x. This note was written using the ADSP-BF707 Blackfin® processor as the reference processor, as it has a dedicated hardware evaluation system and simulator support in the CCES environment, but the topics addressed apply to all ADI processors.

Introducing CCES

The CCES environment is a completely new departure from the previous VisualDSP++® development tools. In addition to the IDE itself changing, which provides numerous feature improvements in the way of code generation tools, debug capabilities, etc., this introduces a certain level of complexity for legacy users already familiar with the VisualDSP++ environment. Additionally, the way in which middleware is provided is also different in CCES, as is the means of providing documentation associated with the processors and supporting software tools.

While VisualDSP++ provided a single installation comprised of the IDDE, tool chain, example directories for all available evaluation platforms (including board-level drivers and application-specific example projects), and ADI-developed middleware supporting operating systems, file systems, and USB/Ethernet stacks, this is no longer the case with CCES. Rather, CCES employs a modular approach in support of ADI evaluation platforms and a partnership with Micrium for separately licensed middleware such as the µC/OS-III™ and µC/OS-II™ real-time kernels, µC/USB Device and Host™ stacks, USB device class drivers, and µC/FS™ file system (handled as *add-ins* into CCES during project definition, which will be covered as part of this note).

Board Support Packages (BSP)

Board Support Packages are downloadable plug-ins for CCES which support processor evaluation systems and EI3 expansion cards. They contain specific code examples and drivers in support of the processor and other system components resident on the evaluation board (e.g., parallel/serial memory, converters, audio codecs, etc.). When installed, all the included code examples and sketches can be indexed by CCES features like **Search** and the **Example Browser**, and the supporting documentation is installed into the CCES Online Help system. These BSPs are available from *www.analog.com* on the respective product pages.

To make full use of this Getting Started Guide, ensure that the latest BSP for the ADSP-BF707 EZ-Kit is installed on top of CCES 1.1.x.

Providing these BSPs outside of the base CCES installation as unique installs allows for more rapid deployment when new features are added or incremental changes are required.

Online Help

CCES Online Help contains all the documentation in support of embedded application development, including the assembler, compiler, linker, and loader tool chain manuals, the firmware (system services and device drivers, etc.) user guides, the processor/board hardware and programmer's reference manuals (including those for any installed BSPs), and any installed middleware manuals. To access CCES Online Help, go to **Help→Help Contents**, as shown in Figure 1:

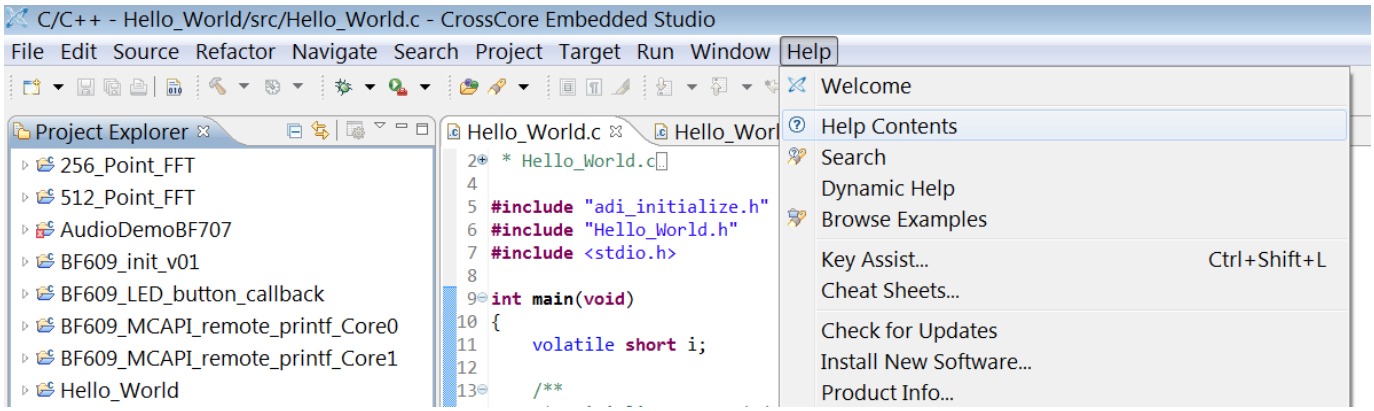

Figure 1. Accessing CCES Online Help

The Help Contents update every time any component related to CCES is added or updated, whether it is the IDE itself, a BSP, or middleware. As seen in Figure 2, Help will display each version of all installed components and provides the means to manually navigate the manuals or do a text search of all the documentation to retrieve specific content. It is highly recommended to utilize CCES On-Line Help in the reading of this note, as there is much more information available regarding the concepts and methods being discussed.

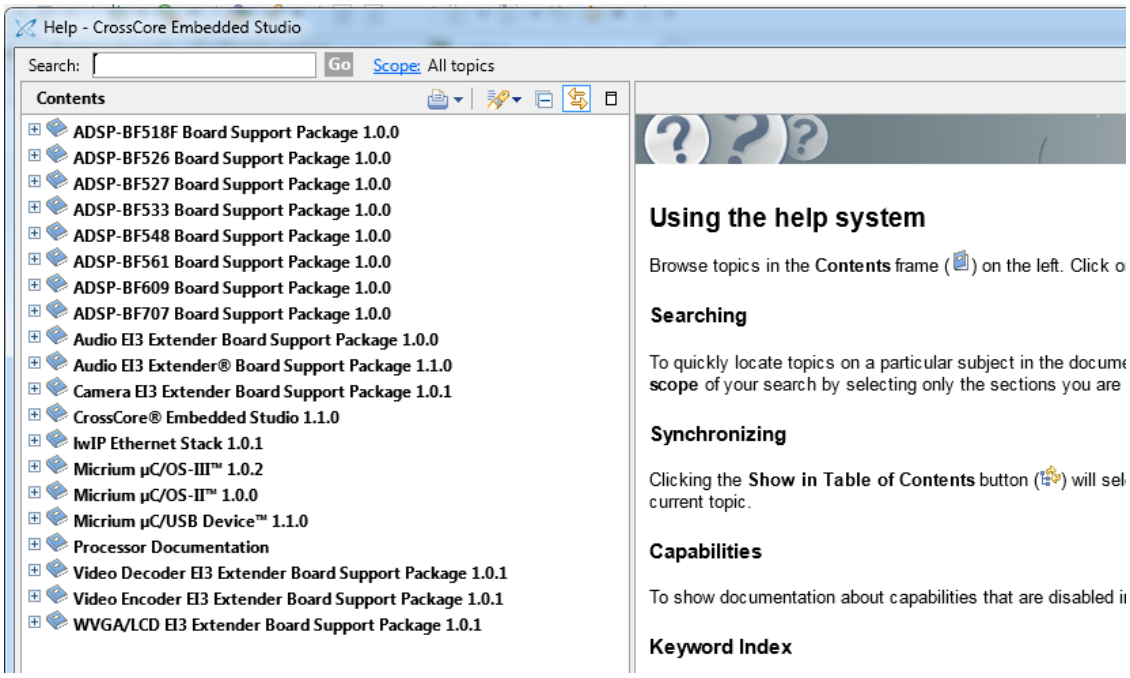

Figure 2. CrossCore Embedded Studio Online Help

Example Browser

The Example Browser indexes all the examples included in CCES and any installed BSP, sorted into two categories – **Example Projects** and **Code Sketches**.

Example Projects are complete examples that are specific to a processor platform, usually an EZ-Kit or an expansion board. When selected, these projects are imported into CCES as complete, functional examples including all the required source, headers, libraries, LDF, and System Configuration files. Examples include the Power-On Self-Test (POST) to bring a board up and test all the interfaces, audio talkthrough, LED management functions, parallel/serial flash memory programming drivers, etc.

Code Sketches are simply snippets of code that properly configure a specific feature or function. These can be copied into a project source file to set up something like the core timer, at which point edits can be made to customize the configuration to specific requirements. To open the Example Browser, use the **Help→Browse Examples** pull-down, as shown in Figure 3.

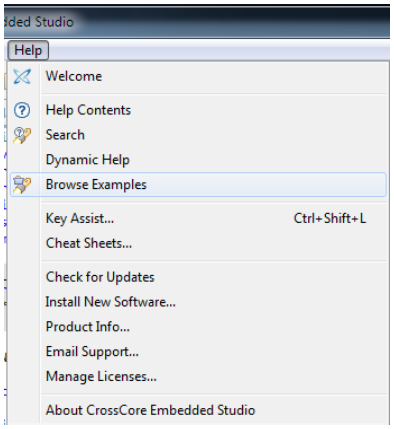

Figure 3. Launching Example Browser

When the Example Browser is launched, a wide range of filter criteria is available, allowing for fine navigation all the way down to a specific add-in for a specific processor. In the **Search results** section, unique icons designate whether the example is an Example Project ( ) or a Code Sketch ( ), as shown in Figure 4:

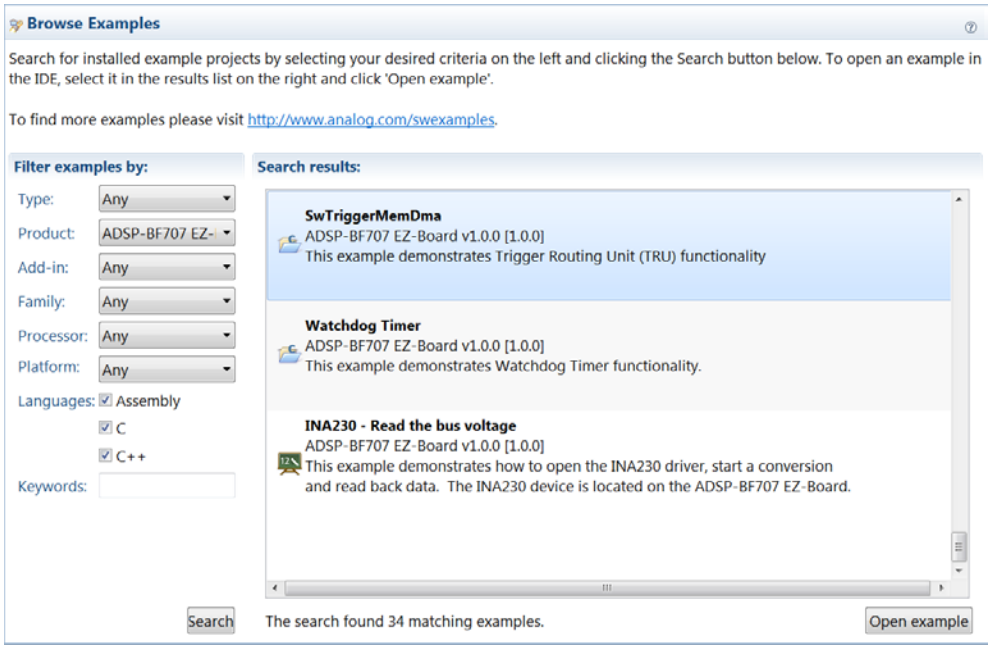

Figure 4. Example Browser

Importing Projects Manually

While the Example Browser provides a simple GUI to bring an individual existing project into the workspace, another method is to do this manually (for individual or multiple projects), which is referred to as *importing* projects. To import projects, use the **File→Import** pull-down, as shown in Figure 5:

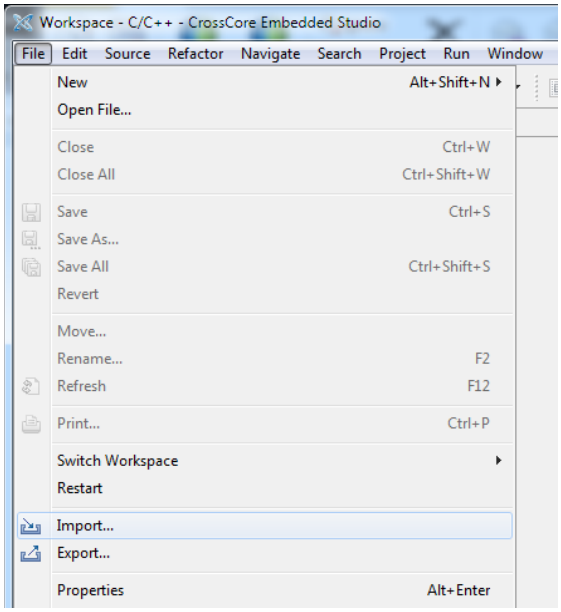

Figure 5. Importing a Project

In the **Select** window that comes up, under the **General** folder, select **Existing Projects into Workspace** and click **Next** (Figure 6).

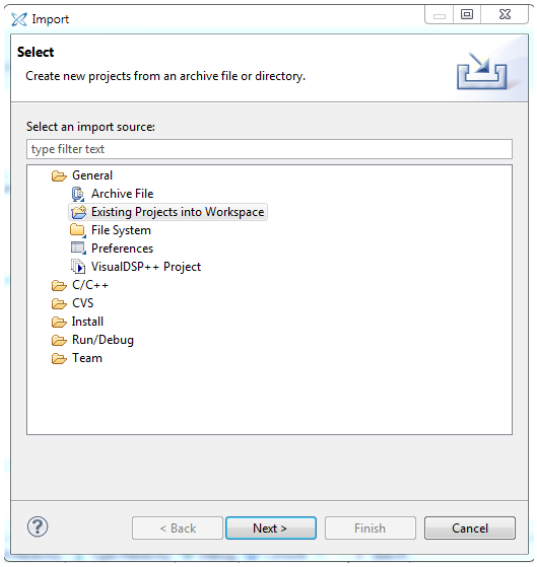

Figure 6. Existing Projects into Workspace

In the **Import Projects** window (Figure 7), click the **Browse…** button next to the **Select root directory** text box and navigate to the root folder of the project(s) to import. Once the root is selected, any projects in that directory (or in any sub-directory beneath the root) will appear in the **Projects** box. Check the box for the project(s) to import, and then click **Finish**.

Make sure the **Copy projects into workspace** option is selected if the original project should be preserved on the host machine. This will make a local copy to the default workspace such that edits will not overwrite what was installed on the machine.

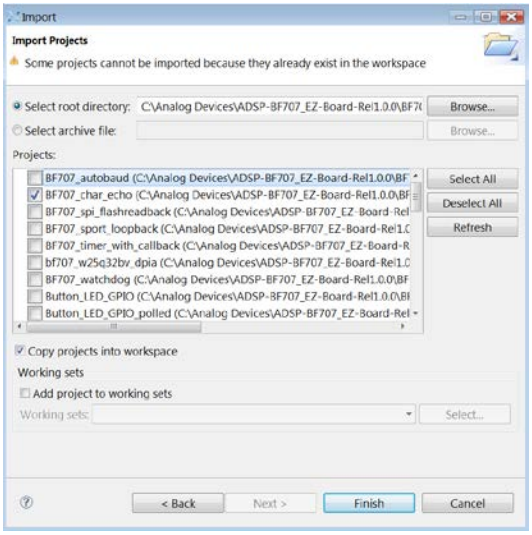

Figure 7. Import Projects

Once completed, all imported projects will appear in the open CCES session in the *C/C++ Perspective*.

CCES Terminology - Perspectives & Views

Before proceeding with creating a project from scratch, it is useful to understand some basic terminology associated with the Eclipse-based IDE, namely *perspectives* and v*iews*. A *perspective* is an instantiation of the CCES IDE dedicated to a specific collection of tasks consisting of a unique set of windows/panes called *views*. The primary perspectives in CCES are the **C/C++** and **Debug** perspectives.

C/C++ Perspective

The C/C++ perspective is where the application code is developed (Figure 8). In this perspective, source and header files (known as *resources*) are created, edited, saved, and compiled into executable files (DXE), static libraries (DLB), or loader images (LDR), depending on the project's output file format.

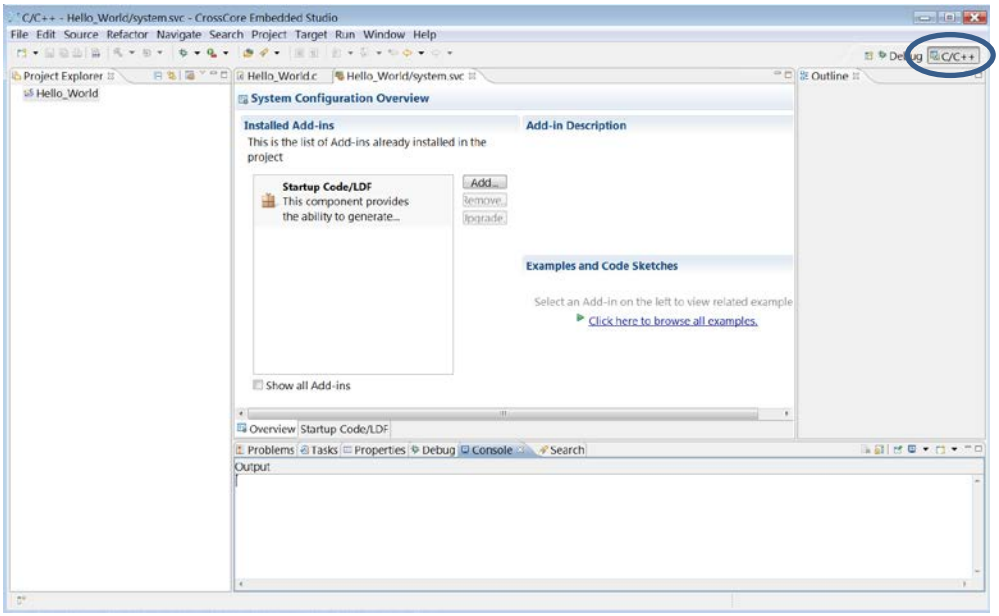

Figure 8. C/C++ Perspective

In the C/C++ perspective, there are several default views, namely the **Project Explorer**, **Editor, Outline** and **Console** views:

- The Project Explorer view to the left contains the directory tree for each of the projects currently open in CCES. Here is where all the individual project resources associated with the open projects are displayed.
- The Editor view in the center is where the open resources can be edited. The editing features built into CCES will dynamically change text color depending on if it's a recognized API, system variable, instruction, data type, or comment (called syntax coloring in Eclipse).
- The Console view at the bottom is where the tool chain outputs to when the project is built. In this view, each component of the tool chain (compiler, assembler, linker, and loader) will output the command-line used to invoke it and report any and all warnings and errors encountered in the process, providing convenient access to specific lines of code that generated compilation/assembler problems and specific error/warning messages that can then be searched in Online Help for further explanation.
- The Outline view to the right provides a skeletal mapping of the open resource showing included header files and any defined macros, functions, and global data, which allows for easy navigation within the resource to where the label is defined/used within the file.

Besides these default views, a number of others are also available via the **Window→Show View** pull-down to be added to the perspective and managed. Online Help can be consulted for specific guidance as to what the views are and how to use them.

Debug Perspective

The **Debug** perspective is available when the code is ready to be run on a hardware target or in the simulator (if there is simulator support for the targeted processor). This perspective offers the ability to set breakpoints, step through code, and access numerous views that assist greatly during development/debug efforts. As was the case with the C/C++ perspective, there are a number of default views associated with the Debug perspective as well, namely the **Debug, Disassembly, Editor,** and **Console** views:

- The Debug view to the upper-left shows each core associated with the target session (which can be a single processor featuring one core, a single processor with multiple cores, or a scan path comprised of any combination of the two) and the application loaded into that core. In the case of the ADSP-BF707, there is only a single core, Core0. Also visible here is the state of the core (Running or Suspended) and the call stack at the halt point.
- The Disassembly view to the right shows the compiler-generated assembly code at specific addresses after it has been downloaded to the target processor and external memory. In this view, the individual lines of C source code (and corresponding line number in the C source file) that generated the assembly sequence can be found interleaved. This code can be single-stepped, and breakpoints can be placed. Editing of the disassembly source is also possible by right-clicking a specific address and selecting **Edit OpCode…**
- The Editor view in the center functions similarly to the same view in the C/C++ perspective, but in the Debug perspective, breakpoints can be set in the C source files.
- The Console view at the bottom also functions similarly to the same view in the C/C++ perspective, except run-time console output from functions such as `printf()` will also appear here.

The open space to the upper right is a tabbed-area view where numerous low-level views can be opened to support specific debug tasks, all of which are available via the **Window→Show View**. Some useful views include:

- The **Memory Browser** view allows visibility into all mapped memory in the system, which includes on-chip RAM/ROM and any mapped external memory. If DDR and/or SPI memory is mapped for the ADSP-BF70x Blackfin processor, this tool can view it. The memory can be indexed by address or label.
- The **Register Browser** view provides the names and values of any of the core and system registers associated with the target processor, either pre-defined (logically grouped by function) or as a single user-defined set of specific registers. This view is extremely useful when debugging code and checking peripheral/interrupt/processor status, and the registers can be viewed on the whole or down to the bit and bit-field level.
- The **Expressions** view can be used as a hybrid Memory Browser and Register Browser view, providing quick access to global data (including content and addressing) and registers.
- If an image processing application is being used, the **Image Viewer** view allows the ability to view images stored on the PC or target processor memory while connected to an emulator session. This tool is extremely useful when using ADI's Image Processing Tool Kits and when working with the Pipeline Vision Processor (PVP) on the ADSP-BF60x processors.

Figure 9 shows an example Debug perspective for an ADSP-BF60x target:

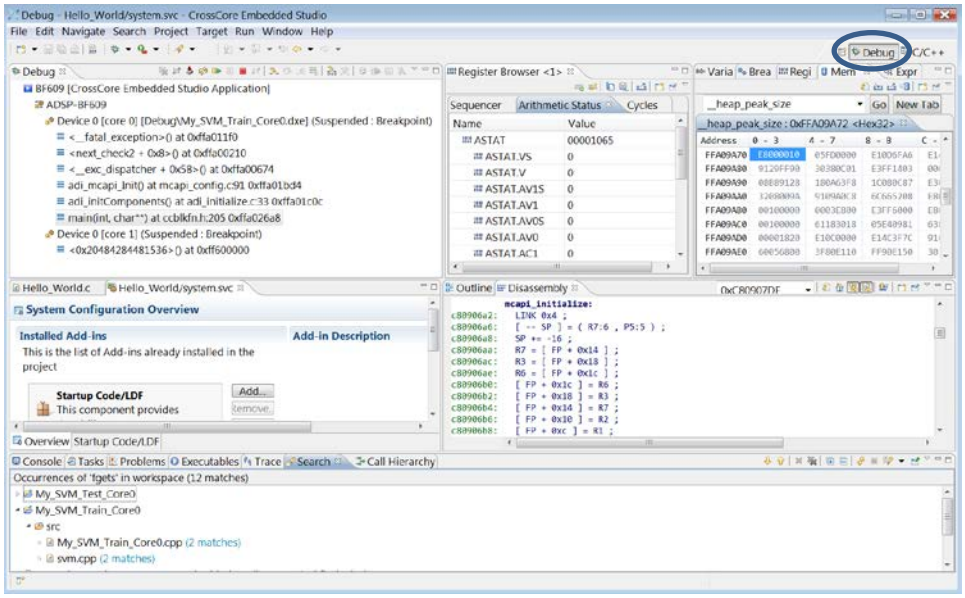

Figure 9. Debug Perspective - Not Connected to Target

Further information about these views and the others available is located in CCES Online Help.

Creating Projects in CCES

Now that the IDE has been introduced and the reader is familiar with the terminology and how the perspectives/views work, the next level is to start a project from scratch, which involves utilizing the **New Project Wizard**.

Using the New Project Wizard

Launch the New Project Wizard using the **File→New→CrossCore Project** pull-down (Figure 10). If this is the first time CCES is being opened, a window will appear asking where to create the default *Workspace* on the host machine. The workspace is where all newly created projects and sub-directories will be stored on the PC. Select the desired pathname, and check the box to remember that location.

If the Example Browser or Import Wizard were exercised during the reading of this EE-note, this process occurred when the first project was opened.

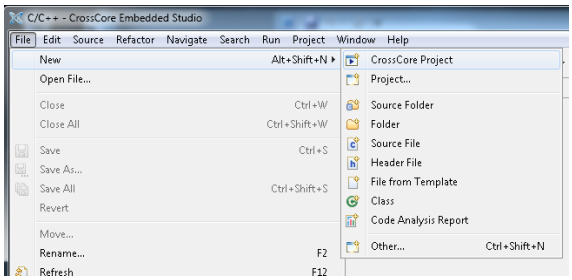

Figure 10. CrossCore Project

The **General Project Information** window will appear (Figure 11). This is where a name is given to the project being created. For the purposes of this guide, call this project "CCES Example", then click **Next**.

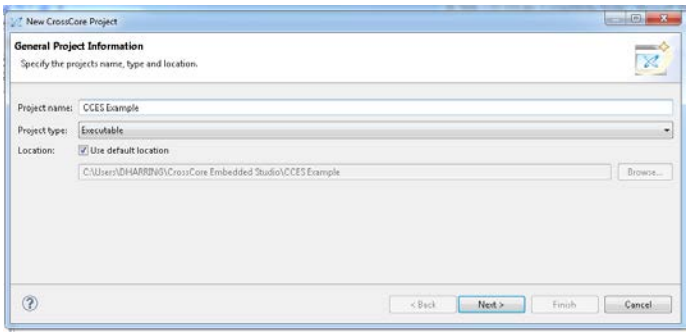

Figure 11. General Project Information

The next window is for **Processor Type and Tool Chain**, which is where the tool chain and processor being programmed are configured. Setting the **Processor family** will alter the ensuing pull-down options accordingly, and it is up to the user to correctly populate the rest of the fields depending on the target processor. Once the **Processor type** and its **Silicon revision** (branded on the processor package itself) have been set appropriately, click **Next**. Figure 12 depicts this window populated for a project targeting silicon revision 0.0 of the ADSP-BF707 Blackfin processor.

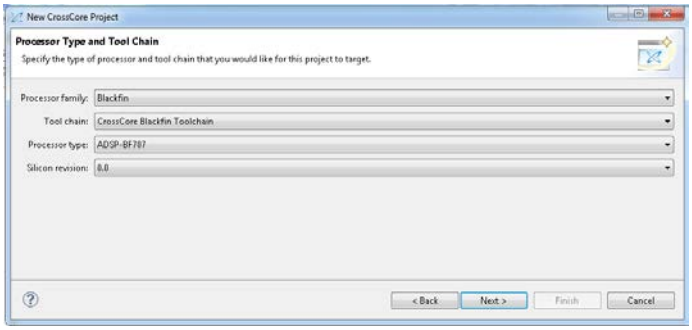

Figure 12. Processor Type and Tool Chain

Next, the **Add-In Selection** window appears (Figure 13). *Add-ins* are preconfigured software that is available for the target processor, including code-generation GUI tools (e.g., Startup Code/LDF, Pin Multiplexing, PVP Programmer, etc.) and middleware libraries supporting device drivers, system services, USB/Ethernet stacks, and operating systems. After the desired add-ins have been selected, click **Next**.

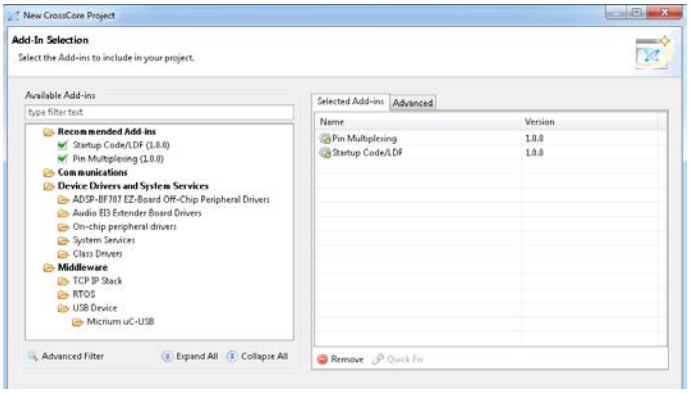

Figure 13. Add-In Selection Window

The **Template Code** window appears next (Figure 14). In this window, the option is provided to have CCES generate default skeletal code to begin development in **Assembly, C or C++.** When mixing source, be sure to choose the highest intended source-code level so that the appropriate tool chain options and syntax are utilized in the project. Once completed, click **Finish**.

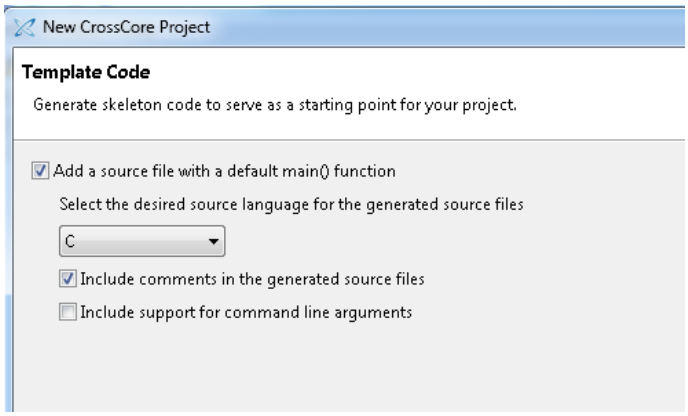

Figure 14. Template Code

The new project will appear in the C/C++ perspective's Project Explorer view, along with all of the generated code, system services, and device drivers requested. Further, the **System Configuration File** will be opened in the Editor view, as shown in Figure 15, which is discussed next.

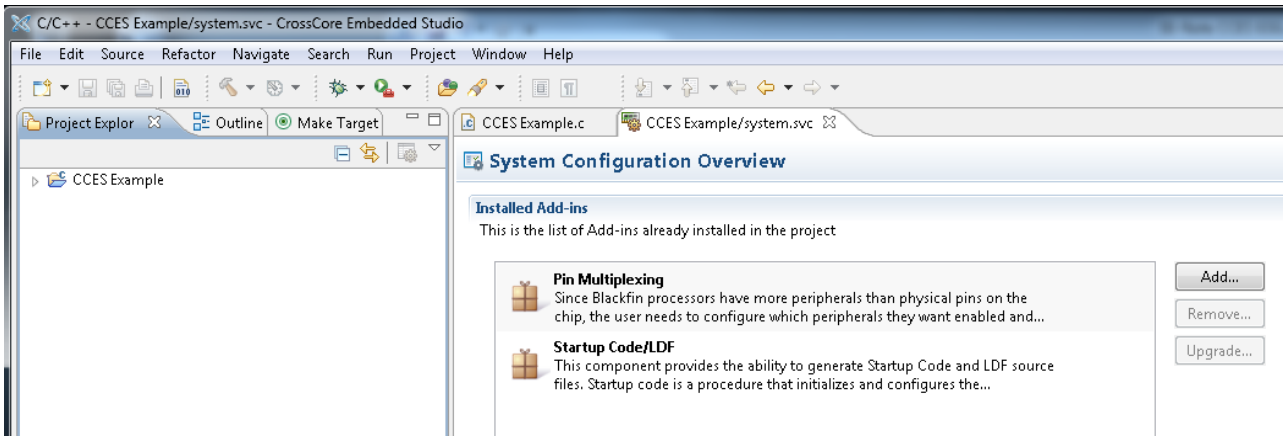

Figure 15. Project Ready

System Configuration File (system.svc)

Every CCES project contains a *system configuration* file called **system.svc**. This file is managed by the System Configuration Utility, which is the IDE's interface for adding, removing, and editing pre-written software components in a project's configuration, such as system services, device drivers, LDF/startup code, and any supported add-ins. It resides in the root of each project, appearing as the last resource in the project tree in the Project Explorer view. As mentioned in the previous section, it appears by default when a project is first created, but the System Configuration Utility can be launched at any time by double-clicking the system.svc file in the Project Explorer view.

The **System Configuration Overview** window lists all the installed add-ins that were selected when the project was created. Here, more add-ins can be added, removed, or upgraded (if the application was created with an older version of the add-ins). For a new ADSP-BF707 Blackfin processor project, the

default add-ins are **Startup Code/LDF** and **Pin Multiplexing** (which is only available for Blackfin projects). In addition to being listed in the **Installed Add-ins** window, individual attributes for each are accessible via the tabs along the bottom of the **System Configuration Overview**, as shown in Figure 16:

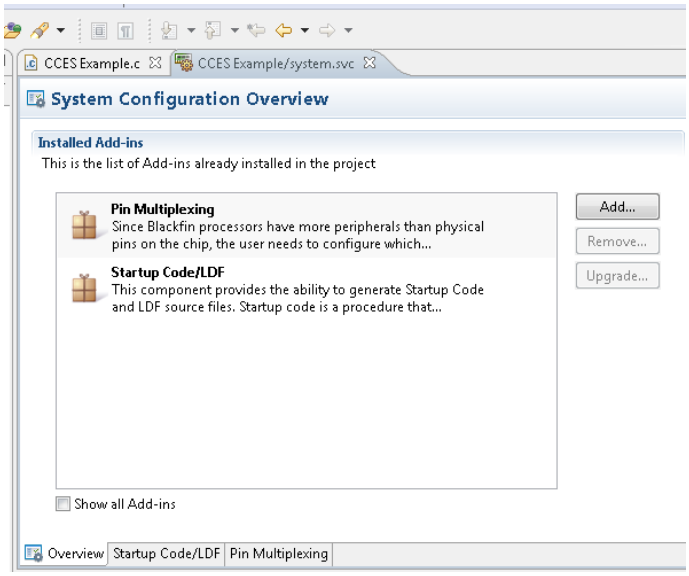

Figure 16. System Configuration Utility

Startup Code/LDF Add-In

The **Startup Code/LDF** add-in automatically generates the necessary startup code for the processor. This code is executed before the application's main function to perform required processor initialization based on user-defined parameters within this add-in. It configures the C run-time header (CRT) and installs default handlers for interrupts/exceptions, among other things. For example, the GUI available via the configuration tab allows the user to select Cache and Memory Protection, configure how memory is initialized, setup external memory, and allocate stacks and heaps, all of which cause updates to the startup code. This GUI also generates/updates the project-required Linker Description File (LDF), which defines the full memory system available to the processor and instructs the tool chain how/where to resolve the various sections of code and data that comprise the full application. The LDF itself is fully customizable as well and can be edited during project debug.

Pin Multiplexing Add-In

The **Pin Multiplexing** add-in provides a GUI to configure pin usage to support the various peripheral interface combinations (SPI, SMC, CAN, TWI, etc.) available on the target processor. The GUI provides all the information necessary to properly configure general-purpose ports on the processor to support the required peripherals and identify/configure pins that are available for GPIO use, including identifying pin conflicts. When the system.svc file is saved, this add-in generates all the required code to properly configure the processor ports to support the specific combination of GPIOs and peripherals designated by the user and updates the initialization code to call this newly generated code.

To install additional add-ins, click the **Add…** button, and the selection window in Figure 17 will appear.

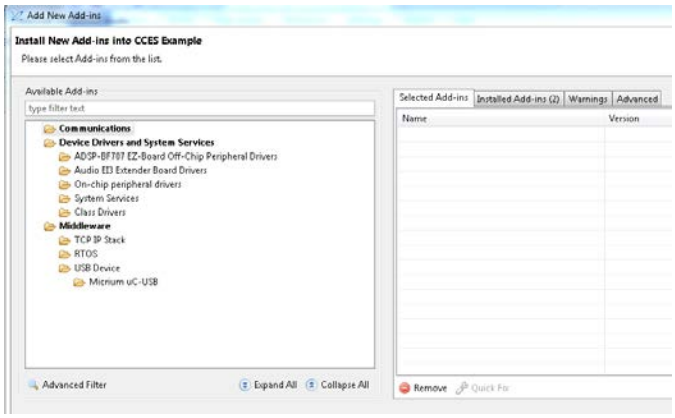

Figure 17. Add New Add-ins

There are also add-ins for the various system services (e.g., RTC and GPIO) and device drivers (for supported peripherals such as PPI, SPI, SPORT, etc.).

System Services and Device Drivers (SSDD)

System Services and Device Drivers provide easy-to-use C/C++ APIs to expedite application development. Device drivers are available for most on-chip infrastructure blocks/peripherals and for several external system components, such as flash memory, converters, audio/video codecs, etc. These drivers leverage the underlying System Services, which provide the same high-level APIs to work with power/clocks, DMA, interrupts, etc., and oftentimes the Device Drivers make the calls into the System Services automatically, removing the developer from bit-level concerns in configuration registers that may be required as a result of changes made at a high level. For example, if a change is made to the clock settings, the properly-used drivers and services will make sure that clock specifications are being met and that system-level adjustments such as DDR refresh rate are automatically checked and corrected, as needed.

With the introduction of CCES, the System Services and Devices Drivers have been upgraded to SSDD 2.0 from the 1.0 version that was available in VisualDSP++; therefore, code written in VisualDSP++ is incompatible with the CCES implementation and will need to be reworked. The SSDD documentation is available via Online Help under `CrossCore Embedded Studio 1.1.x -> System Run-Time Documentation -> System Services and Device Drivers`.

Project Properties

Once the project is fully defined and all associated setup code has been established via the System Configuration Utility, the next important concept is that of controlling how the project will be built. This is handled via the **Project Properties** page, accessible via the **Project Explorer** view by right-clicking the main project folder and selecting **Properties**. The Properties window will appear, providing access to multiple sub-pages of configurable options. The most commonly accessed options are located on the first four tabs of the **C/C++ Build→Settings** page shown in Figure 18.

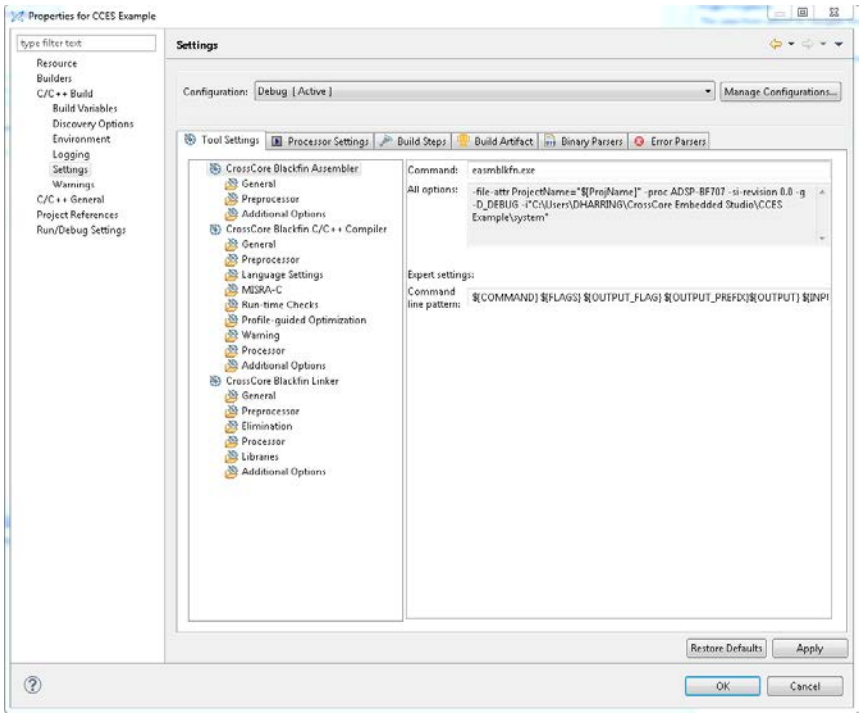

Figure 18. Setting Project Properties

- The **Tool Settings** tab is where all the tool chain components (compiler, assembler, linker and loader) can be configured. Each unique sub-page provides access to and descriptions for many of the supported command-line switches that can be invoked during the build process, giving the user quick control over things such as optimization settings.
- The **Processor Settings** tab describes the processor type and silicon revision, as setup during project creation. This is where an existing project can be modified to target a new processor and/or silicon revision after evaluation is complete (i.e., an application developed during evaluation on an ADSP-BF707 EZ-Kit needs to migrate to a custom target board designed around the lower-cost ADSP-BF704).
- The **Build Steps** tab provides a means of adding command-line directives before and after those defined by the Tool Settings tab to be invoked automatically when the project is built. For example, launching the Command-Line Device Programmer to program the flash automatically after the loader file is generated by the build is supported in the **Post-build steps** on this tab.
- The **Build Artifact** tab gives the flexibility to modify the output file type for the project to generate a processor executable (DXE), a bootable loader image (LDR), or a static library to be included as part of other projects (DLB), controlled via the **Artifact Type** pull-down. For example, a DXE is what is needed to debug the code in a simulator or emulator session, which employs the tool chain up to and including the linker. Once fully debugged, the DXE needs to be made into a defined boot stream, which requires the Loader component of the tool chain as well. Changing the Artifact type to **Loader File** and setting the appropriate fields in the Loader property pages on the Tool Settings tab is all that is required.

Developing a Simple Application (LED Blink Example)

To help get started using CCES, here are steps to create an example application to blink an LED on the ADSP-BF707 EZ-Kit. This step-by-step example assumes familiarity with the C/C++ programming language and how to access CCES Online Help for information on the APIs.

If not done already, follow the instructions in the Using the New Project Wizard section of this guide to create the **CCES Example** project and switch the Editor view to the *CCES Example.c* tab, as shown in Figure 19.

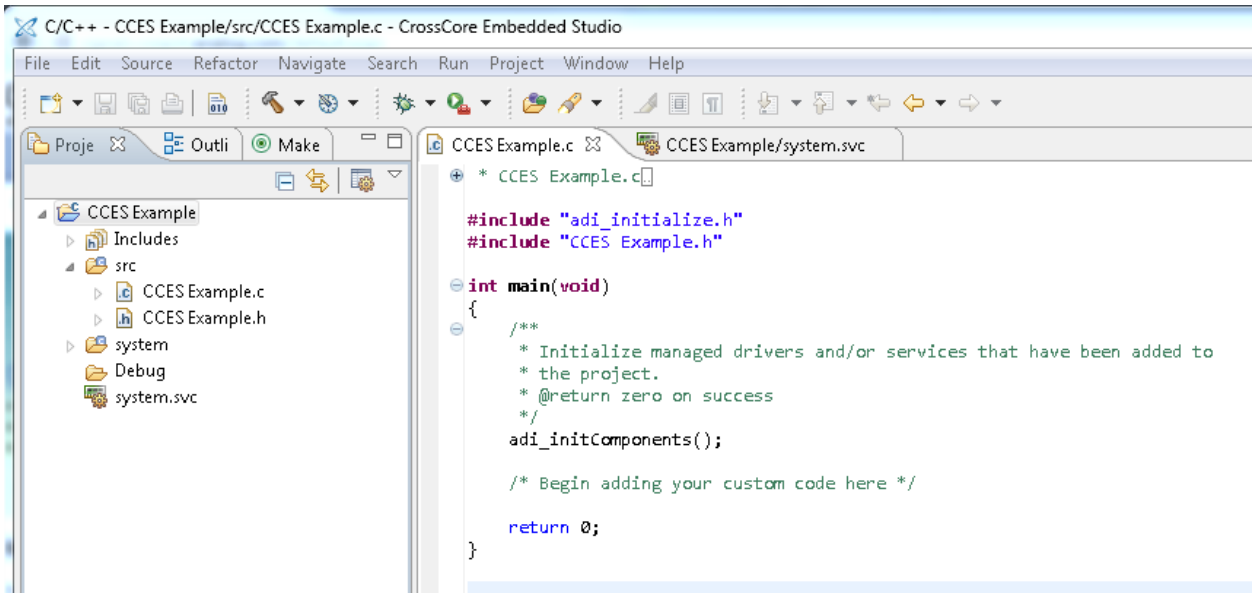

Figure 19. CCES Example.c

The template code generated at project start-up by CCES includes auto-generated .c and .h files that take the project name. In the *CCES Example.c* file, there is a skeletal `main()` function containing only a call to the `adi_initComponents()` function. The `adi_initComponents()` function is used to initialize drivers and services that were included using the system configuration utility. This is a necessary function in `main()` and must be executed before any custom code. Without this function call, the designated add-ins will not operate.

The CCES system services include a GPIO service, which is used in this example application. This service can be used to set the direction of the GPIO pins, as well as read, set, toggle, and clear the states of each pin. Polling of pin states and interrupt methods are available through the GPIO service as well, which is documented in Online Help under `CrossCore Embedded Studio 1.1.x -> System Run-Time Documentation -> System Services and Device Drivers -> ADSP-BF70x API Reference -> Modules -> GPIO Service`.

Listing 1 is the full code example needed to blink the LED on the ADSP- BF707 EZ-Kit, and a description of the code follows:

```
/*****************************************************************************
 * CCES Example.c
 *****************************************************************************/

#include "adi_initialize.h"
#include "CCES Example.h"
#include <services/gpio/adi_gpio.h>

#define GPIO_MEMORY_SIZE (ADI_GPIO_CALLBACK_MEM_SIZE*2)

static uint8_t gpioMemory[GPIO_MEMORY_SIZE];

int main(void)
{
	/**
	 * Initialize managed drivers and/or services that have been added to
	 * the project.
	 * @return zero on success
	 */
	adi_initComponents();

	/* Begin adding your custom code here */

	ADI_GPIO_RESULT result;
	uint32_t gpioMaxCallbacks;
	/* initialize the GPIO service */
	result = adi_gpio_Init((void*)gpioMemory, GPIO_MEMORY_SIZE, &gpioMaxCallbacks);

	result = adi_gpio_SetDirection(ADI_GPIO_PORT_A, ADI_GPIO_PIN_0, ADI_GPIO_DIRECTION_OUTPUT);

	while(1)
	{
		result = adi_gpio_Toggle(ADI_GPIO_PORT_A, ADI_GPIO_PIN_0);
		for(int x = 0; x<5000000; x++);
	}

	return 0;
}
```

Listing 1. LED Blink Application for the ADSP-BF707 EZ-Kit

Each of the system services has a dedicated header file located in the **\services** sub-directory under the CCES root directory, and the first step in writing this LED Blink example is to include the header for the GPIO service after the default headers are included:

```
#include "adi_initialize.h"         ← Automatically included
#include "CCES Example.h"           ← Automatically included
#include <services/gpio/adi_gpio.h> ← Must Be Added
```

The next step is to define the object array which will contain the GPIO service data. This array will be used during the initialization of the GPIO service, and the macro definitions utilized can be found in the `adi_gpio.h` file added above.

```
#define GPIO_MEMORY_SIZE (ADI_GPIO_CALLBACK_MEM_SIZE*2)

static uint8_t gpioMemory[GPIO_MEMORY_SIZE];
```

A variable is then required for the return values from the system service and device driver APIs:

```
	ADI_GPIO_RESULT result;
```

Since the goal of this example is to build a simple blink application, the GPIO service needs to be initialized with the call to the `adi_gpio_Init()` function, and the direction of the I/O pin connected to the LED must be set as an output pin via the `adi_gpio_SetDirection()` function. The APIs have return values that indicate success or failure, which can provide valuable debug information, and this example uses `result` for each of the API calls to the GPIO service. Because of the simple nature of this example (i.e., only one pin is used, and only one service is being initialized), there is no error-checking implemented. A more complex system should utilize error-checking to ensure that the device drivers and system services are all being initialized and used properly.

Finally, to actually blink the LED, the `adi_gpio_Toggle()` GPIO toggle API is called from within an infinite `while(1)` loop, buffered on each iteration by the `for(int x = 0; x<5000000; x++)` delay loop to space out toggling of the GPIO state.

Debugging the Application

When all the above edits have been made to the *CCES Example.c* file, the project can be built. To do so, use the **Project→Build Project** pull-down.

When the above code is built, a warning is generated indicating that the `return 0;` instruction is unreachable. This instruction will never execute due to the infinite `while(1)` loop before it, but the build will complete because warnings do not cause an abort of the build process. The instruction can safely be deleted from the code and the project rebuilt to clear this warning, if desired.

Once the application is built, the **Debug** perspective can be launched by clicking **Debug** in the top right of the CCES window. To debug the application, select the **Run→Debug** pull-down to connect to the target processor. If a **Debug Configuration** has not yet been configured, a window will come up prompting for creation of a new **Debug Configuration** using the **Session Wizard**. In the **Select Processor** window, select the **ADSP-BF707** and click **Next** (Figure 20).

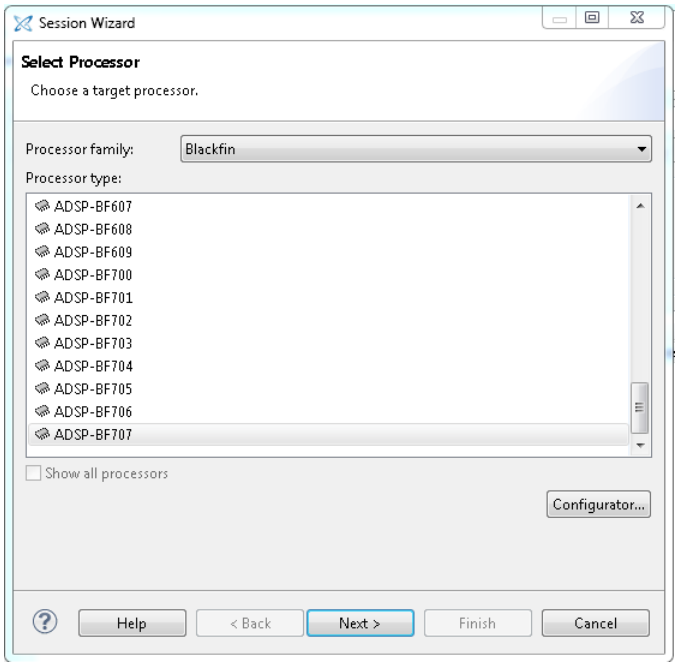

Figure 20. Session Wizard, Select a Processor

The next window asks to select the environment in which the target processor should be connected to the IDE, whether it is via an **Emulator**, as an **EZ-Kit via the Debug Agent**, or (for processors that support it) as a functional **Simulator**. This example is connecting to the board through an emulator, but the environment suitable for the target configuration is what should be selected here, then click **Next**.

The next window is the **Select Platform** window (Figure 21), which will vary depending on the previous selection. For Simulator and Debug Agent targets, there would be only one platform to pick here. For the Emulator environment, however, all the supported emulators will be displayed. Select the one appropriate for the emulator in use (e.g., **ADSP-BF707 via ICE-1000**), then click **Finish**.

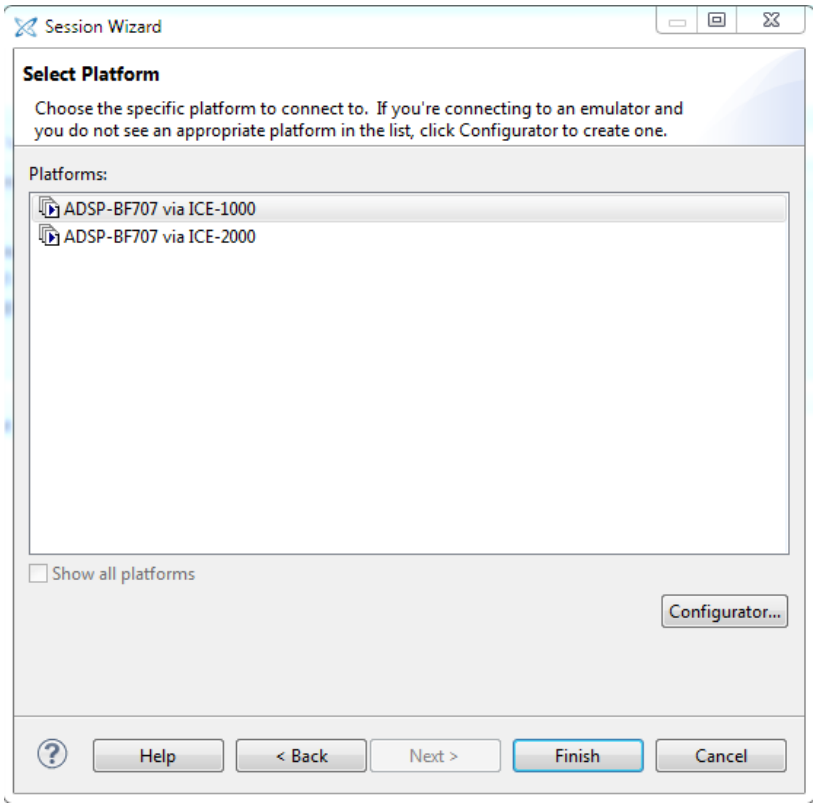

Figure 21. Select Emulator Platform

Before proceeding, make sure the target board is powered and properly connected to the PC (via Debug Agent or Emulator). Once hardware power and connections are confirmed, click the Debug icon ( ) to launch the Debug perspective and connect to the processor (Figure 22).

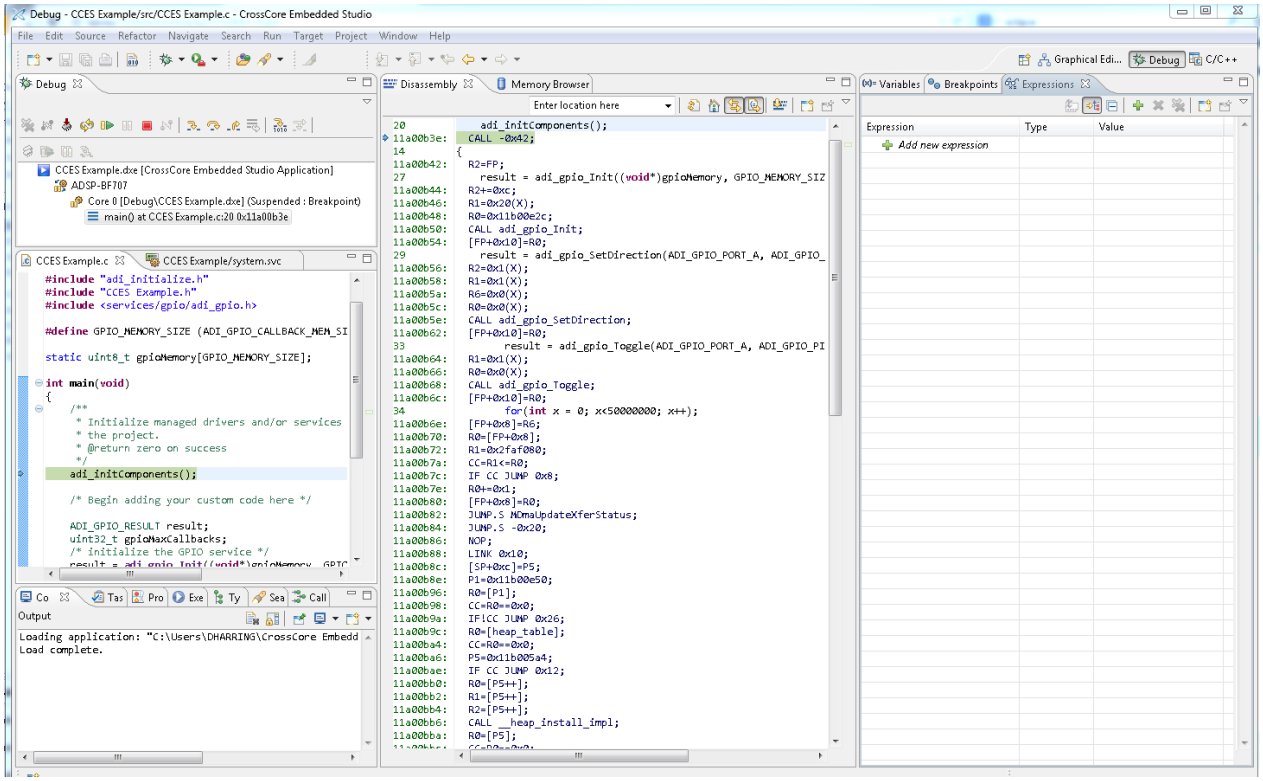

Figure 22. Debug Perspective after Connecting to ADSP-BF707 Processor

Once the processor has connected, the program will run to `main()` and halt. The **Run** pull-down contains the functions for running, stopping, pausing, and stepping through the application (Figure 23).

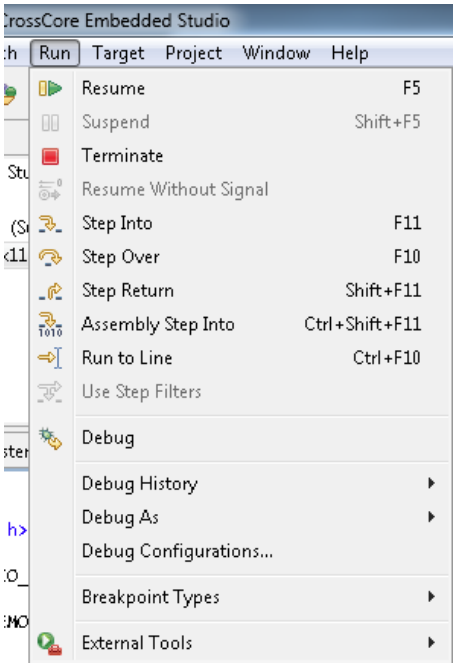

Figure 23. Run Menu

Setting Breakpoints

One of the most important features of debugging an application is being able to set breakpoints within the code. CCES provides two types of breakpoints - software breakpoints and hardware breakpoints.

Software breakpoints are handled in emulator/debug agent firmware. Essentially, the emulator keeps a record of all the places software breakpoints are established and replaces those instructions in memory with a private instruction with special bit encodings so that execution can stop at the breakpoint, at which point the emulator swaps back in the actual instruction that should be run when the program is resumed from the breakpoint.

Hardware breakpoints allow much greater flexibility than software breakpoints and require much more design thought and resources within the processor. At the simplest level, hardware breakpoints are helpful when debugging ROM code, where software breakpoints are not possible due to the need to write the instruction memory dynamically to support the breakpoint. As hardware breakpoint unit capabilities are increased, so are the benefits to the developer. At a minimum, an effective hardware breakpoint unit will have the capability to trigger a break on load, store, and fetch activities. Additionally, address ranges, both inclusive (bounded) and exclusive (unbounded) should be included.

Before breakpoints can be set, the project must be built. Further, project rebuilds are required every time code is modified, otherwise the DXE being run on the processor will not match the new DXE that would be generated after changing the code.

To set a breakpoint, go to the desired line of code and right-click on the gutter to the left of the Editor view (the blue shaded area). Select **Toggle Hardware Breakpoint** or **Toggle Software Breakpoint**, as shown in Figure 24. Once a breakpoint is set, this same toggle will remove the breakpoint, and control of the software breakpoints is also available in the **Breakpoint** view. In this example, a software breakpoint has been set on the `for()` loop.

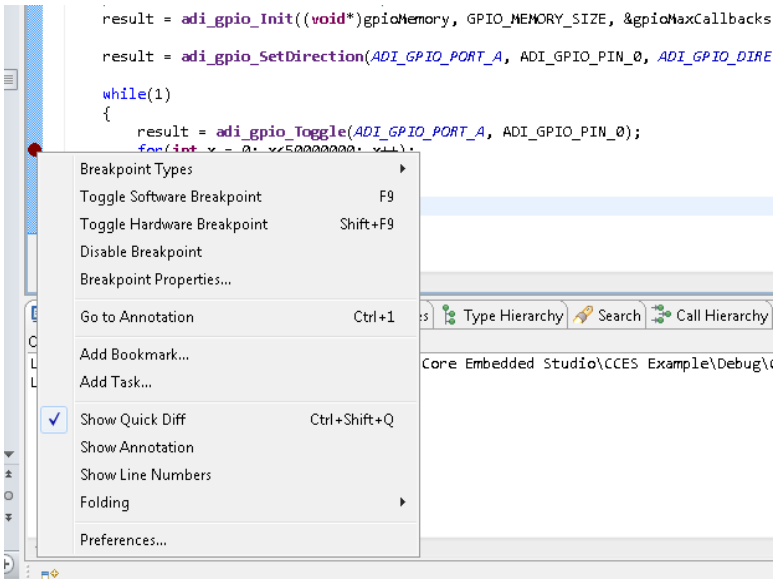

Figure 24. Setting a Breakpoint

Once the breakpoints have been set, the debug session can be resumed from the start of `main()` via the **Run→Resume** pull-down, by striking the **F5** key, or by clicking the green resume arrow ( ) in the **Debug** view toolbar. Once resumed, the application will suspend at the breakpoint, as shown in Figure 25.

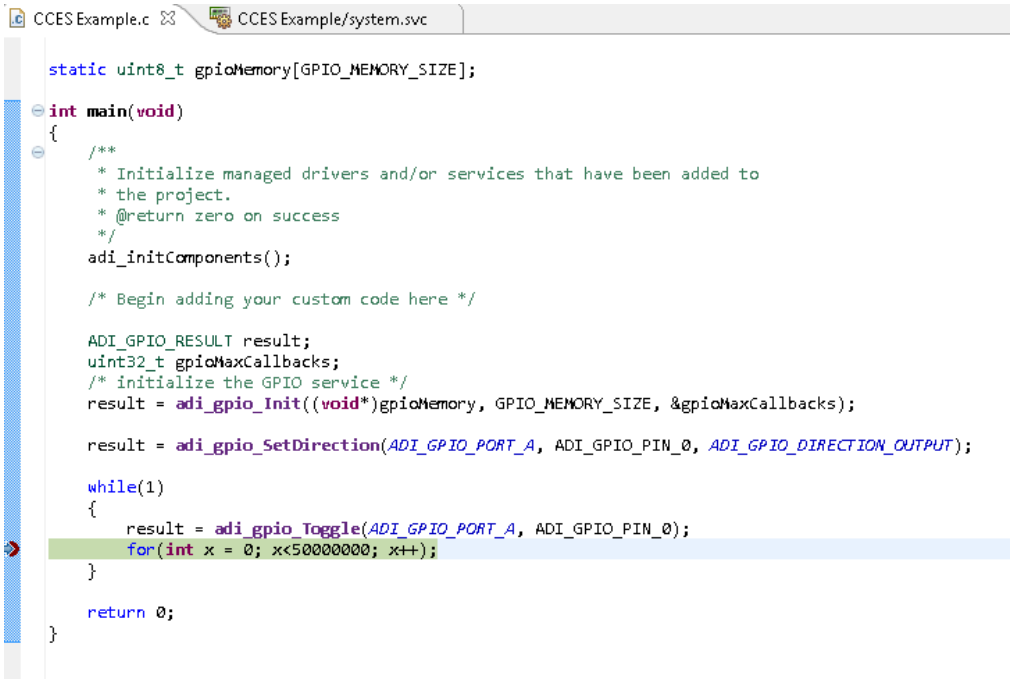

Figure 25. Application Suspended at Software Breakpoint

With the application stopped at the breakpoint, the processor context can be inspected, including analyzing local data like the loop counter `x`. Go to the **Expressions** view, accessible via the **Window→Show View→Expressions** pull-down. Click on *Add new expression*, and type "x" in the **New expression** text box. After striking Enter, the **Expressions** view updates to populate the **Type** and **Value** columns for the variable `x`, as shown in Figure 26.

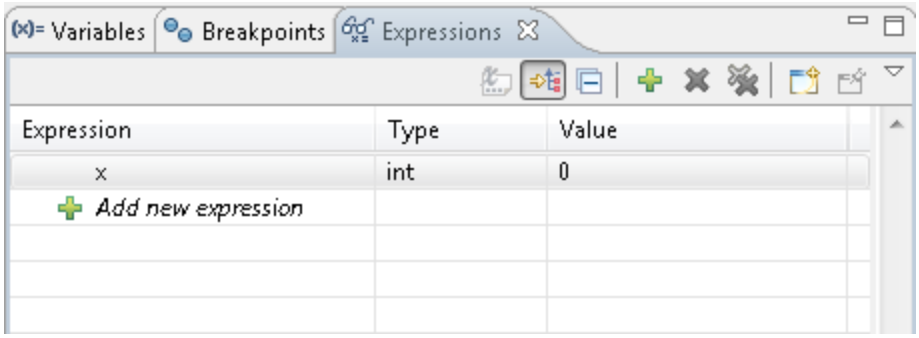

Figure 26. Expression View of Variable x

Since this breakpoint has stopped the application before the `for()` loop has run, `x` has its initial value of 0. Clicking resume at this point will result in the application again stopping at this same breakpoint due to the `while(1)` loop, and the **Expressions** view is updated, as shown in Figure 27. As can be seen, the `x` value is now the max value of the `for()` loop from the previous iteration of the loop, as it has not yet been initialized back to 0 for the next iteration of the same loop.

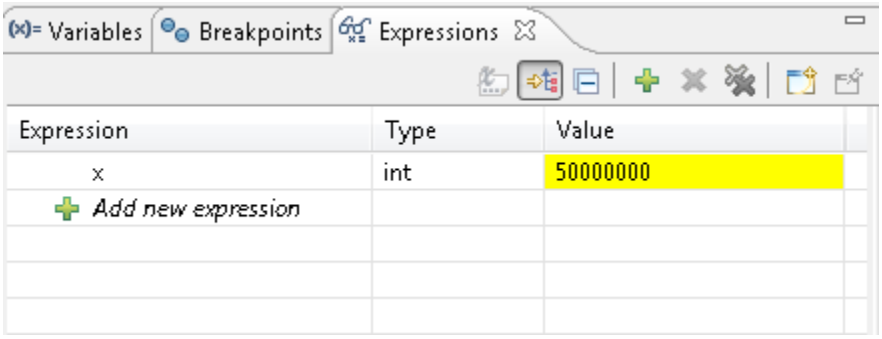

Figure 27. New Value of x after for() Loop Has Completed and while(1) Loop Has Iterated

Note that any value that has changed since the last suspesion point will be highlighted in yellow. This is extremely useful when debugging multiple variables simultaneously.

Creating Bootable Applications

To create a bootable application that will be loaded into the connected flash memory, a loader image (LDR) needs to be created. This requires the DXE developed in the previous section as the main application, but it will also require an initialization file if any external memory is needed by the application. This particular example is small enough to fit entirely in on-chip memory, but the concept is described to support much larger development efforts.

Initialization File

When using an emulator-based CCES debug session, the IDE is dynamically loading the application's executable file (DXE) to the processor's on-chip and external memory, thus bypassing the boot process altogether. For on-chip memory, there is no difference between how the emulator communicates with the processor and how the boot stream behaves. However, the JTAG interface used to target the processor does not communicate directly with the external DDR memory on the board. Instead, the data destined for DDR space is sent via JTAG to the processor, and the processor then writes it out over its external bus to the DDR memory. To support this, the emulator firmware must set registers on the processor to properly configure the clocks and external bus interface prior to attempting to resolve the application's code and data to external memory. Failure to account for this would result in hardware errors and exceptions when the processor attempts to load to an external device that has not yet been configured. When moving to a stand-alone boot image (i.e., when the emulator is not doing this automatically), the same initializations must be performed explicitly by the application, which is handled in an **Initialization file**.

An initialization file is a small processor DXE file that is prepended as a special block at the top of the LDR image, called an *initialization block*, before the actual application code. This block is booted into on-chip memory first and is run *before* any external accesses are attempted. In the boot stream, once the initialization block has loaded and run, flow returns to the boot ROM so that the rest of the application can then be loaded, which includes resolving code and data to off-chip memory as well as overwriting the on-chip memory that was used to run the initialization code.

For development board targets, the affected registers and their settings are defined in XML files. Specific to the ADSP-BF707 evaluation platform, the *ADSP-BF707-resets.xml* file located in the CCES installation in the **\System\ArchDef** directory contains this information. In this file, a list of Clock Generation Unit (CGU0) and Dynamic Memory Controller (DMC0) registers is provided. These are the registers that need

to be programmed to support the CLKIN and DDR memory specific to the ADSP-BF707 EZ-Kit Lite board and the values each register is being programmed to by the emulator, as shown in Listing 2:

```
CGU0_DIV = 0x41022241
CGU0_CTL = 0x00001000
DMC0_PHY_CTL4 = 0x00000001
DMC0_CAL_PADCTL2 = 0x0078283C
DMC0_CAL_PADCTL0 = 0xF0000000
DMC0_CFG = 0x00000522
DMC0_TR0 = 0x20B08232
DMC0_TR1 = 0x20270618
DMC0_TR2 = 0x00323209
DMC0_MR = 0x00000432
DMC0_EMR1 = 0x00000000
DMC0_EMR2 = 0x00000000
DMC0_CTL = 0x00002404
DMC0_DLLCTL = 0x0000054B
L1IM_ICTL = 0x00000000
USB0_SOFT_RST = 0x03
EPPI0_CTL = 0x00000000
```

Listing 2. Processor Registers Programmed By the Emulator for ADSP-BF707 EZ-Kit Lite Debug Session

CCES furnishes the project for the initialization requirements for the ADSP-BF707 EZ-Kit Lite, which can be modified to support custom boards with other CLKIN values and/or alternative DDR memory. The project can be found in the **\Blackfin\ldr\init_code\BF707_init\BF707_init_vxx** directory, where the xx indicates the board revision being used. For this exercise of generating a loader file, no modifications to the code are necessary, but the DXE file will need to be built if it hasn't been built before. To do this, open the BF707_init_vxx project and build it. After it builds, the **BF707_init_vxx.dxe** file will be generated in the **\Debug** directory of the project. Note the path to this DXE file, as it will be required later.

With the initialization DXE generated and the application DXE finalized, the loader file must be created:

1. Select the **Project→Build Configurations→Set Active→Debug** pull-down menu
2. In the **Project Explorer** view, right-click on the project name and select **Properties**
3. In the **Properties** window, go to the **C/C++ Build→Settings** page and select the **Build Artifact** tab. Under **Artifact Type**, select **Loader File**.
4. On the **Tool Settings** tab, go to the **CrossCore Blackfin Loader→General** page and input the settings as follows:
    a. Boot mode: **SPI0 Master**
    b. Boot format: **Intel Hex**
    c. Output width: **16 bits**
    d. Boot Code: **0x01**
    e. Make sure **Use default start address** is checked
    f. Add the initialization file by clicking **Browse…** and navigating to <CCES Root> \Blackfin\ldr\init_code\BF707_init\BF707_init_v00\Debug\BF707_init_v00.dxe
5. The window should now resemble Figure 28. Click **OK.**

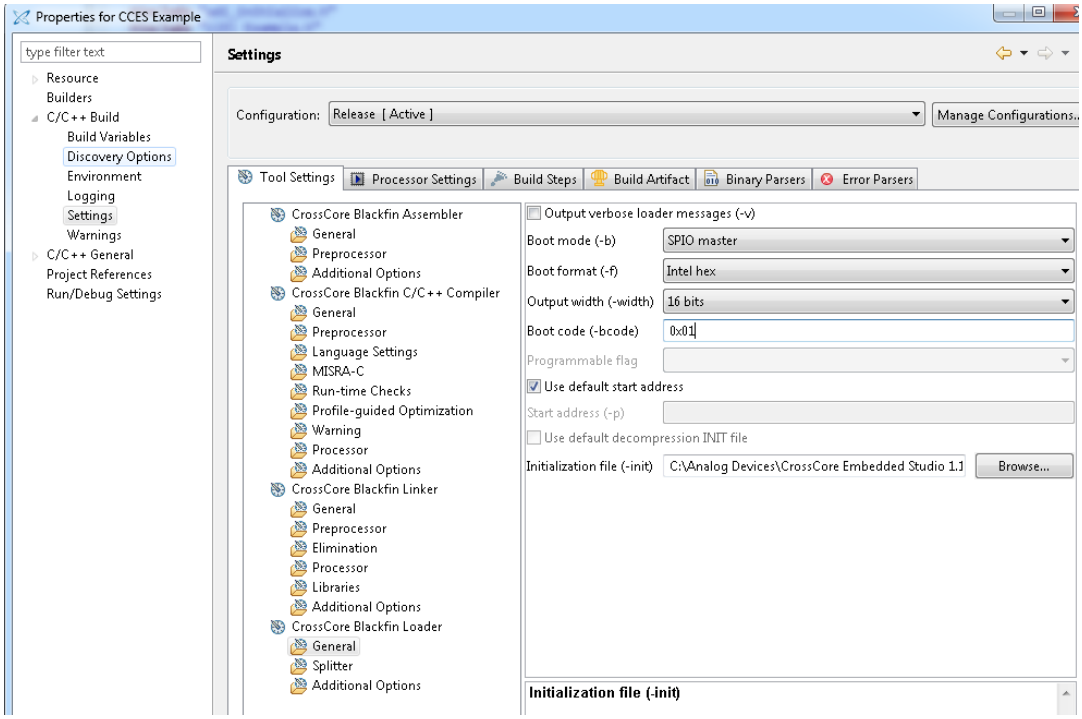

Figure 28. Loader File Settings in Project Properties

6. Use the **Project→Build Project** pull-down menu to generate the LDR file for the project.

With the loader image generated, it is ready to be programmed into the boot source memory, which will require use of the Command-Line Device Programmer (CLDP).

Dual-Core Processor Loader Considerations

While the ADSP-BF707 is not a dual-core processor, other processors such as the ADSP-BF609 processor have multiple cores. There are additional considerations when building the LDR image for an application that has a unique DXE file for each core. For example, the project name appends _Core0 and _Core1 to differentiate between the two unique executable files targeting multiple cores on a single processor (e.g., **BF609_MCAPI_remote_printf_Core0** and **BF609_MCAPI_remote_printf_Core1**). Each output directory contains a dedicated DXE file for that core, and special handling is required within the loader utility to create the appropriate LDR image to support the two unique DXE files in a single loader stream.

To build a LDR file supporting two cores, the LDR output itself must be generated by the Core1 project, as it is in the Core1 project that the dependencies on the Core0 project need to be set. As such, first build the Core0 project with the default DXE output and note the path to the DXE for inclusion later. Once the Core0 DXE is finalized, do the same for the Core1 project so that there are two finalized DXE files for the two unique cores on the processor.

If the initialization file for the ADSP-BF609 EZ-BOARD has not yet been generated, follow the guidance described for the ADSP-BF707 EZ-Kit in the previous section for the ADSP-BF609 EZ-BOARD and note the location of the generated initialization DXE for inclusion later as well.

Once the three DXE files are finalized, follow these steps to generate the required dual-core LDR file:

1. Click on the Core1 project name in the **Project Explorer** view to select it as the active project.
2. Right-click on the Core1 project name and select **Properties**, which opens the **Properties** window.
3. On the **C/C++ Build→Settings** page, select the **Build Artifact** tab, and use the **Artifact Type** pull-down to select **Loader File**.
4. On the **Tool Settings** tab, select the **CrossCore Blackfin Loader→General** page and verify the following settings:
    a. Boot mode: **Memory**
    b. Boot format: **Intel Hex**
    c. Output width: **16 bits**
    d. Set the **Boot code** field to **0x6** to select 16-bit flash boot.
    e. Verify that **Use default start address** is checked.
5. For the **Initialization file**, click **Browse…** and navigate to the **\Debug** directory for the initialization code built earlier and select the **BF609_init_vxx.dxe** file (where xx is the board revision).

It is at this point that the dependencies on the Core0 DXE need to be accounted for when generating the LDR file. As previously mentioned, the loader stream is broken up into blocks, with each block being sequentially processed by the boot ROM during boot time. Each block is prefixed with a specified header that contains special tag information that instructs the boot ROM how to process the data that follows the header. In a typical single-core application, the last block has a *Final* tag associated with it, indicating that the boot process will be complete when that block is processed. If this final tag is associated with the Core0 DXE's final block, it will cause the boot process to exit as soon as that block has been processed, which would result in nothing being booted for the Core1 application. To avoid this behavior, the loader needs to be configured to treat Core0's last block as a normal block such that booting will continue with the Core1 DXE after the Core0 DXE is fully loaded.

For more information about the ADSP-BF609 processor boot process, see the *Boot ROM and Booting the Processor* chapter of the *ADSP-BF60x Blackfin Processor Hardware Reference*.

To configure the loader to properly boot to both cores (continuing from above):

6. Go to the **CrossCore Blackfin Loader→Additional Options** page.
7. In the **Additional options** pane, click the **Add…** icon in the header bar, as shown in Figure 29:

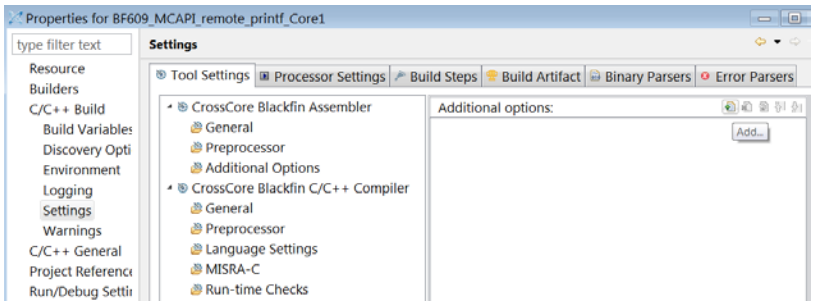

Figure 29. Setting Additional Options in the Loader Settings Window

In the window that pops up, two directives must be added to the command line for the LDR file to be appropriately generated. The first item is the path to the Core0 DXE file (relative to the Core1 output directory). For example, consider a project directory structure like this:

TestProject_Core0
    Debug
        TestProject_Core0.dxe
TestProject_Core1
    Debug
        TestProject_Core1.dxe

For this example, the correct pathname for the Core0 DXE relative to the location of the Core1 DXE would be:

..\..\TestProject_Core0\Debug\TestProject_Core0.dxe

The *–NoFinalTag* loader switch is also needed to skip insertion of the final tag for the indicated executable, as follows:

–NoFinalTag=..\..\TestProject_Core0\Debug\TestProject_Core0.dxe

Continuing with the example:

8. In the **Additional options** field in the pop-up window, enter the text for the Core0 DXE path and the loader switch. Per the example above, this would be (on a single line):
   ..\..\TestProject_Core0\Debug\TestProject_Core0.dxe -NoFinalTag=..\..\TestProject_Core0\Debug\TestProject_Core0.dxe

If the pathnames contain spaces, they must be cared for by employing double-quotes and escape characters, e.g., *"C:/Directory\ With\ Spaces/Debug/TestProject_Core0.dxe"*.

9. Click **OK** to close the pop-up window, and click **OK** to close the **Properties** window. If prompted to regenerate LDF/startup code, select **No.**
10. Build the LDR file using the **Project→Build Project** pull-down.

Command Line Device Programmer (CLDP)

With the LDR file generated, the final step is to program the blink example to the SPI flash memory on the ADSP-BF707 EZ-KIT Lite. This is handled via the Command Line Device Programmer (CLDP) utility.

As of CCES 1.1.1, the CLDP is a command-line tool, though a GUI version is planned for a future release.

Just as the JTAG does not directly access DDR memory when downloading the DXE during a debug session, the CLDP does not interact directly with the flash memory on the board either. Rather, it uses the processor to load a host driver to communicate with the flash memory, then it passes the LDR image in a data stream via JTAG so that the processor can take it and write it to the flash memory by issuing the proper write commands to the flash. To coordinate all of this, the **cldp.exe** program requires a number of

command-line switches and arguments to initialize the JTAG programmer, load the flash driver to the processor, and then load the LDR image to the flash memory on the ADSP-BF707 EZ-KIT Lite, as follows:

-proc: the target processor is the ADSP-BF707

-emu: indicates which emulator driver is in use for the active debug session (needed to send the LDR image to the processor)

Because this example utilizes the ICE-1000 emulator, this switch is set to **ICE-1000**. Consult CCES On-Line Help for the full description of the appropriate CLDP switch settings for the debug configuration being used.

-driver: indicates the flash memory device driver (included with the CCES installation) that the processor must use to work with the Winbond W25Q32BV flash device on the board (**bf707_w25q32bv_dpia.dxe**).

-cmd: the flash program command (**prog**) must be explicitly defined

-erase: flash memory must be erased before being programmed, designating only the **affected** region be erased

-file: indicates the LDR image which is to be written to the flash

For this example, per the above, the single command line that must be issued is:

cldp -proc ADSP-BF707 –emu ICE-1000 -driver "<BF707 EZ-KIT BSP root directory>\BF707_EZ-Board\Blackfin\Examples\Device_Programmer\bf707_w25q32bv_dpia.dxe" -cmd prog –erase affected -file "<workspace directory>\Debug\CCES Example.ldr"

The use of <*directory*> indicates that the full path to these files is required based on where the BSP is installed on the PC and where the CCES project was created.

To have CCES invoke the CLDP at the end of the build process:

1. In the **Project Explorer** view, right-click on the **CCES Example** project name and select **Properties…**
2. On the **C/C++ Build→Settings** page, select the **Build Steps** tab.
3. Under the **Post-build steps** section, populate the **Command** field with the command line above, as in Figure 30.

Do *not* copy and paste the command line from above, as formatting differences will not translate properly and will cause the CLDP to fail due to incorrect parsing of the command-line in the post-build steps.

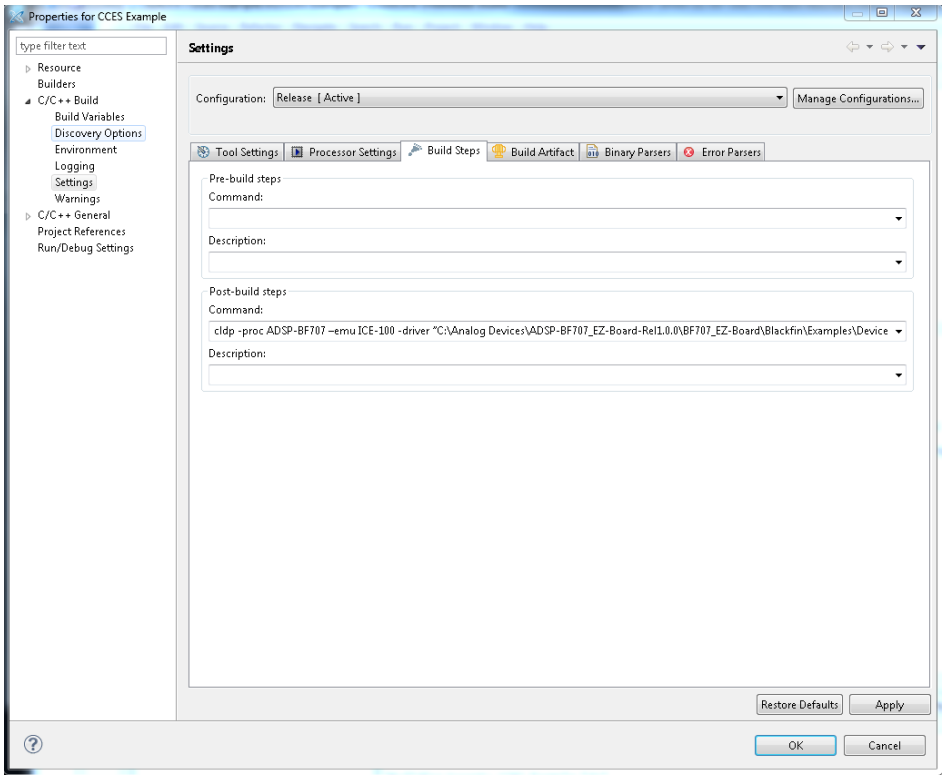

Figure 30. CLDP Command Line in Project Settings Build Steps Window

Click **OK** and build the project using **Project→Build Project**, and the CCES session should resemble Figure 31.

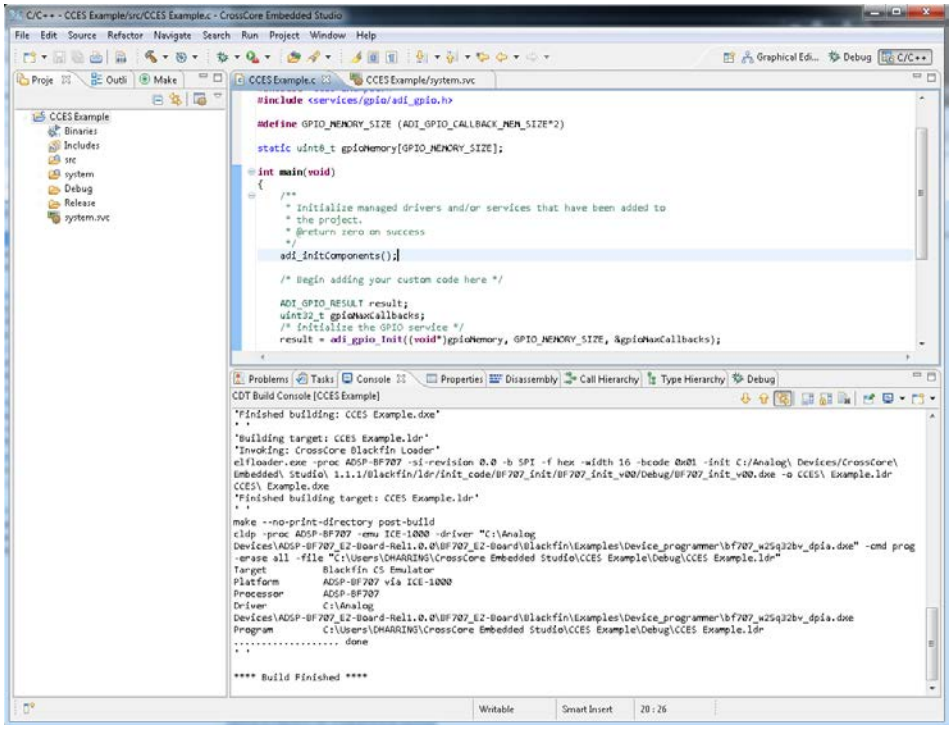

Figure 31. Command Prompt Window after Executing CLDP

The successful **. . . . . done** message in the Console view indicates that the device programming is complete, and the application is now present in the flash memory. To verify that the application boots properly outside the context of CCES, close CCES on the PC and verify that the boot selector switch (SW1) on the board is in position 1 to boot from SPI flash. If the board's power is cycled or the reset button is pushed, the LED blink application will boot from the SPI flash memory and behave as it did in the debug session.

References

[1] *ADSP-BF70x Blackfin Processor Hardware Reference.* Rev 0.2, May 2014. Analog Devices, Inc.

[2] *ADSP-BF70x Blackfin Embedded Processor Data Sheet.* Rev PrD, December 2014. Analog Devices, Inc.

[3] *Embedded Processing and DSP (http://www.analog.com/processors).* May 2005. Analog Devices, Inc.

Readings

[1] *ADSP-BF70x Blackfin Processor Programming Reference.* Rev 0.2, May 2014. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – March 13, 2015 *by DH and Joe B*	Initial Release