Code Composer Studio v3.0
Getting Started Guide

Literature Number SPRU509E
September 2004

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Preface

Read This First

About This Manual

To get started with Code Composer Studio™ IDE, you must go through the first two chapters of this book. The remaining chapters contain information that can be useful to you, depending on your needs and the tools you are using. To determine whether you can utilize the features in these chapters, please review the online help provided with Code Composer Studio.

How to Use This Manual

This document contains the following chapters:

Chapter	Title	Description
1	Introduction	Introduces TI's eXpressDSP technology initiative, and includes Code Composer Studio's simple and very basic development flow.
2	Getting Started Quickly with Code Composer Studio IDE v3	Provides an abridged overview of some of the basic features and functionalities in Code Composer Studio.
3	Target and Host Setup	Provides information on how to define and set up your target configuration, and how to customize several of the general IDE options.
4	Code Creation	Provides options available to create code and build a basic Code Composer Studio project.
5	Debug	Reviews the debug tools and shows you how to use them.
6	Analyze/Tune	Discusses the various tools to help developers analyze and tune their applications.
7	Additional Tools, Tips	Gives information on how to find additional help for documentation, updates, and with customizing your Code Composer Studio installaton.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

  ```
  0011  0005  0001          .field    1, 2
  0012  0005  0003          .field    3, 4
  0013  0005  0006          .field    6, 3
  0014  0006                .even
  ```

  Here is an example of a system prompt and a command that you might enter:

  ```
  C:  csr -a /user/ti/simuboard/utilities
  ```

Related Documentation From Texas Instruments

For additional information on your target processor and related support tools, see the online manuals provided with the CCStudio IDE.

To access the online manuals:

Help→CCStudio Documentation→Manuals

Related Documentation

You can use the following books to supplement this user's guide:

American National Standard for Information Systems-Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C)

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

Programming in C, Kochan, Steve G., Hayden Book Company

Trademarks

To Help Us Improve Our Documentation . . .

If you would like to make suggestions or report errors in documentation, please email us. Be sure to include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Email: support@ti.com

Contents

Figures

Chapter 1

Introduction

This chapter introduces TI's eXpressDSP technology initiative. It also includes Code Composer Studio's simple and very basic development flow.

Topic **Page**

1.1 Welcome to the World of eXpressDSP

TI has a variety of development tools available that enable quick movement through the DSP-based application design process − from concept, to code/ build, through debug analysis, tuning, and on to testing. Many of the tools are part of TI's real-time eXpressDSP™ Software and Development Tool strategy, designed to enable innovators and inventors to speed new products to market and turn ideas into reality. These tools prove very helpful in quickly getting started as well as saving valuable time in the design process, allowing you to concentrate on differentiating your product in the marketplace. TI's real-time eXpressDSP Software and Development Tool strategy includes three tightly knit ingredients that will empower developers to tap the full potential of TMS320™ DSPs:

❑ The world's most powerful DSP-integrated development tools: Code Composer Studio™ Development Tools.

❑ eXpressDSP Software including:

  ■ a scalable, real-time software foundation: DSP/BIOS™ kernel,

  ■ standards for application interoperability and reuse: TMS320 DSP Algorithm Standard, and

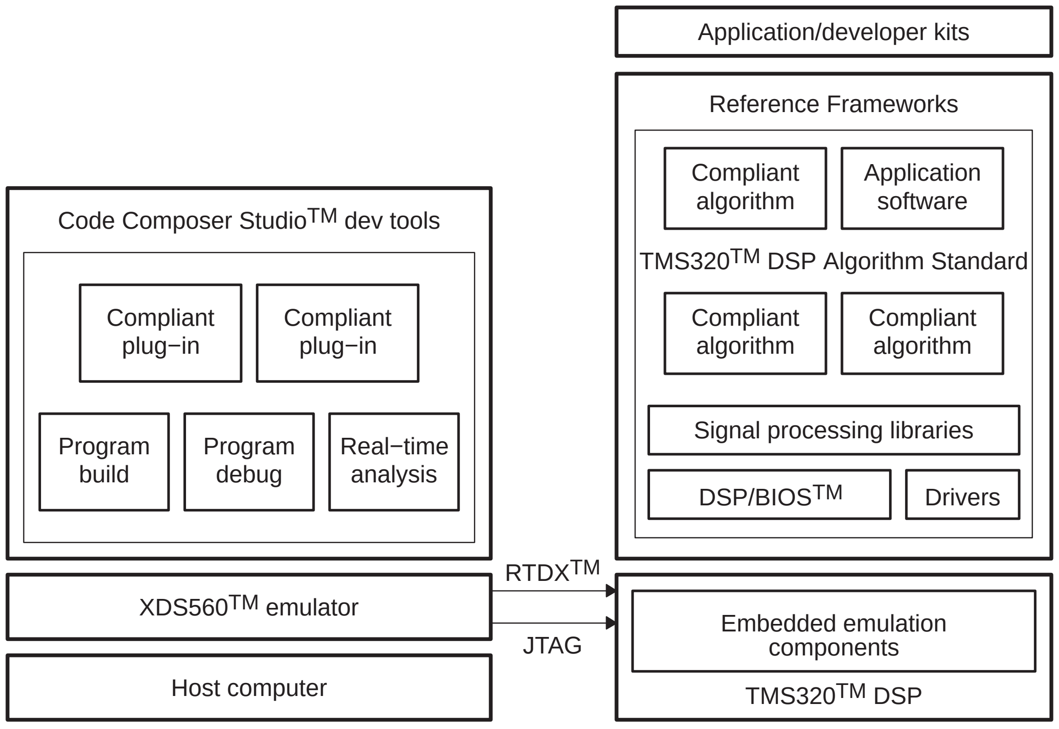
  ■ design-ready code that is common to many applications to get you started quickly on DSP design: eXpressDSP Reference Frameworks.

❑ A growing base of TI DSP-based products from TI's DSP Third Party Network, including eXpressDSP-compliant products that can be easily integrated into systems.

Each element is designed to simplify DSP programming and move development from a custom-crafted approach, to a new paradigm of interoperable software from multiple vendors supported by a worldwide infrastructure.

There has been an explosive growth in real-time applications demanding the real-time processing power of TI DSPs. eXpressDSP enables innovators and inventors to speed new products to market and turn ideas into reality. Previously unimagined applications, including virtual reality, medical imaging, auto navigation, digital audio, and Internet telephony now rely on the crucial real-time computing power that can only be found in a DSP.

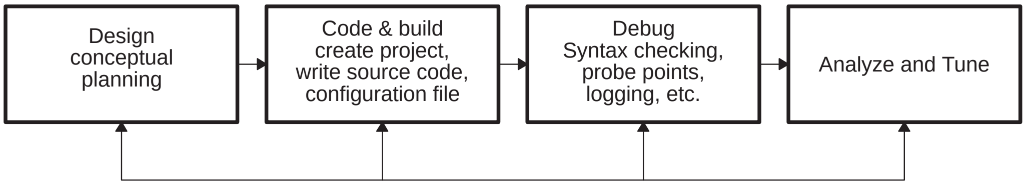
Figure 1−1. eXpressDSP ™ Software and Development Tools



1.2 Development Flow

The development flow of most DSP−based applications consists of four basic phases: application design, code creation, debug, and analyze/tune. Code Composer Studio is the key element of TI’s eXpressDSP software and development tools which integrate many of the tools needed to assist the developer in the development flow. This user’s guide will provide basic procedures and techniques in program development flow.

Figure 1−2. Simplified Code Composer Studio Development Flow



Chapter 2

Getting Started Quickly

This chapter provides an overview of some of the basic features and functionalities in Code Composer Studio v3, to guide you in creating and building simple projects. Experienced users should skip this chapter and proceed to the following chapters for more in-depth explanations of Code Composer Studio's various features.

Topic **Page**

2.1 Launching Code Composer Studio

To launch Code Composer Studio IDE for the first time, click the icon (shown below) on your desktop. A simulator is automatically configured by default. To configure Code Composer Studio for a specific target, please refer to Chapter 3 for more information.

2.1.1 Important Icons Used in Code Composer Studio v3

This list of icons is important in helping you to traverse through the Code Composer Studio IDE. These icons will be referred to throughout this manual.

Used to launch Code Composer Studio

Rebuilds the project

Builds the project incrementally

Halts execution

Toggle breakpoint toolbar button

Run toolbar button

Step into button

Step out of button

Step over button

2.2 Creating a New Project

You can create a working project by following these steps:

Step 1: If you installed Code Composer Studio in C:\CCStudio, create a folder called **practice** in the C:\CCStudio\myprojects folder.

Step 2: Copy the contents of C:\CCStudio\tutorial\target\consultant folder to this new folder. (Note: target refers to the current configuration of Code Composer Studio. By default, the target is sim64xx LE. For more about Code Composer Studio configurations, please refer to Chapter 3)

Step 3: From the Project menu, choose New.

Step 4: In the Project Name field, type **practice**.

Step 5: In the Location field, type or browse to the folder you created in step 1.

Step 6: By default, Project Type is set as Executable (.out) and Target is set as the current configuration of Code Composer Studio.

Step 7: Click Finish. Code Composer Studio creates a project file called practice.pjt This file stores your project settings and references the various files used by your project.

Step 8: Add files to the project by choosing Add Files to Project from the Project menu. You can also right-click the project in the Project View window on the left and then select Add Files to Project. Add main.c, DoLoop.c, and lnk.cmd (this is a linker command file that maps sections to memory) from the folder you created. Browse to the C:\CCStudio\c6000\cgtools\lib\ directory and add the rts.lib file for the target you are configured for.

Step 9: You do not need to manually add include files to your project, because the program finds them automatically when it scans for dependencies as part of the build process. After you build your project, the include files appear in the Project View.

2.3 Building Your Program

Now that you have created a functional program, you can build it. Since this is the first time the project is being built, it is recommended that you use the Build All function. An output window appears to show the build in process. When the build is finished, the output window will display “Build complete 0 errors, 0 warnings.”

Rebuild All is mainly used to rebuild the project when project options have been modified.

These build methods can also be accessed in the Project menu. For further information, please go to Chapter 4.

2.4 Loading Your Program

After the program has been built successfully, load the program by going to File→Load Program. By default, Code Composer Studio will create a subdirectory called Debug within your project directory, and store the .out file in it. Select practice.out and click Open to load the program.

Note: Remember to reload the program by choosing File→Reload Program if you rebuild the project after making changes.

2.5 Basic Debugging

You can see Code Composer Studio’s versatile debugger in action by completing the following exercises. For more in-depth information, please refer to Chapter 5.

2.5.1 Go to Main

To begin execution of the Main function, select Debug→Go main. The execution halts at Main.

2.5.2 Using Breakpoints

To set a breakpoint, place the cursor on the desired line and press F9. In addition, you can also set the breakpoint by selecting the toggle breakpoint toolbar button. To remove the breakpoint, simply press F9 or the button again. When a breakpoint has been set, a red icon will appear at the selection margin to the left of the code.

In main.c, set the breakpoint at the line “DoLoop(Input1, Input2, Weights, Output, LOOPCOUNT);”. Since execution was halted at the main function, now you can press F5, select Debug→Run or select the Run toolbar button to run the program. As you can see, once execution reaches the breakpoint, it halts.

2.5.3 Source Stepping

Source stepping is only possible when program execution has been halted. Since you halted at breakpoint, you can now execute the program line by line using source stepping.

Step Into the DoLoop function by selecting the Step Into button. Step through a few times to observe the executions.

The Step Over and Step Out Of functions are also available and those buttons are right below the Step Into button.

Assembly stepping is also available. Whereas source stepping steps through the lines of code, assembly stepping steps through the assembly instructions. For more information on assembly stepping, please go to section 5.2.1.

2.5.4 Viewing Variables

In the debugging process, it is often necessary to view the value of the variables to ensure that the function executes properly. Variables can be viewed in the watch window when the CPU has been halted. The watch window can be accessed by View→Watch Window. In the watch locals tab, all the relevant variables in the current execution will be shown.

As you continue to Step Into the while loop, you will see that the values of the variables change through each execution. In addition, you can view the values of specific variables by hovering the mouse pointer over the variable or by placing the variables in the Watch1 tab. For more information on variables and watch windows, please go to section 5.2.4.

2.5.5 Output Window

The Output window is located at the bottom of the screen by default. It can also be accessed by View→Output Window. By default, the printf function produces the same Output window. Information such as the contents of Stdout and the build log is displayed in the Output Window.

2.5.6 Symbol Browser

The symbol browser is a powerful tool that allows you to access all the components in your project with a single click. Select it through View→Symbol Browser. The symbol browser has multiple tabs: in this section we will discuss the Files, Functions, and Globals tabs.

When you expand the tree in the Files tab, you will see the source files in your project. A file is automatically accessed when you double-click on it; the same holds true for the Functions tab. The Globals tab will allow you to access the global symbols in your project.

For more information on the Symbol browser, please go to section 5.2.9.

These few steps have allowed you to successfully create, build, load and debug your first Code Composer Studio program.

2.6 Introduction to Help

Code Composer Studio provides a multitude of help tools accessed through the Help menu. Select Help →Contents to search by contents. Tutorials guide you through the Code Composer Studio development process.

Select Help →Web Resources to obtain the most current help topics and other guidance. User manuals are pdf files that provide information on specific features or processes.

Access Code Composer Studio’s newest features through Help→Update Advisor.

Chapter 3

Target and Host Setup

This chapter provides information on how to define and set up your target configuration for both single processor and multiprocessor configurations, and how to customize several of the general IDE options.

Topic **Page**

3.1 Define and Set Up Target

3.1.1 Code Composer Studio Setup Utility

This section discusses how to use the Setup utility to define and set up your target configuration, for both single processor and multiprocessor configurations.

3.1.1.1 Importing an Existing Configuration

The Setup utility allows you to configure the software to work with different hardware or simulator targets. You must select your own configuration in Setup before starting Code Composer Studio.

You can create a configuration using the provided standard configuration files, or create a customized configuration using your own configuration files (see the online help and/or the tutorial). For the purposes of this example, the standard configuration files are used.

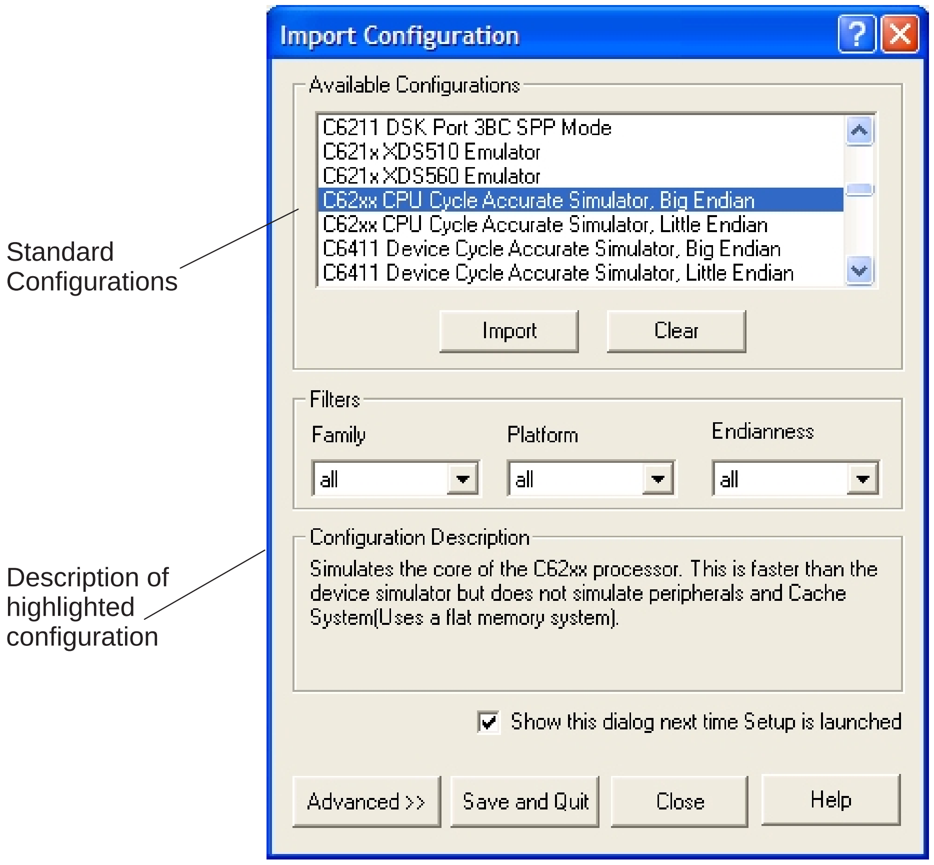
To create a system configuration using a standard configuration file:

Step 1: Double-click on the Setup Code Composer Studio desktop icon. Both the System Configuration dialog box and the Import Configuration dialog boxes appear, but we are concerned with the Import Configuration box for this step.

Step 2: Click the Clear button in the Import Configuration dialog box to remove any previously defined configuration.

Step 3: Click Yes to confirm the Clear command.

Step 4: From the list of Available Configurations, select the standard configuration that matches your system.



C6211 DSK Port 3BC SPP Mode
C621x XDS510 Emulator
C621x XDS560 Emulator
C62xx CPU Cycle Accurate Simulator, Big Endian
C62xx CPU Cycle Accurate Simulator, Little Endian
C6411 Device Cycle Accurate Simulator, Big Endian
C6411 Device Cycle Accurate Simulator, Little Endian

Family: all
Platform: all
Endianness: all

Simulates the core of the C62xx processor. This is faster than the device simulator but does not simulate peripherals and Cache System(Uses a flat memory system).

☑ Show this dialog next time Setup is launched

Determine if one of the available configurations matches your system. If none are adequate, you can create a customized configuration (see the online help and/or the tutorial).

Step 5: Click the Import button to import your selection to the system configuration currently being created. The configuration you selected now displays under the My System icon in the System Configuration pane of the Setup window.

If your configuration has more than one target, repeat steps 4 and 5 until you have selected a configuration for each board.

Step 6: Click the Save and Quit button to save the configuration.

Step 7: Click the Yes button to start the Code Composer Studio IDE with the configuration you just created.

You can now start a project. See Chapter 4 of this book, or the online help and tutorial for information on starting a project.

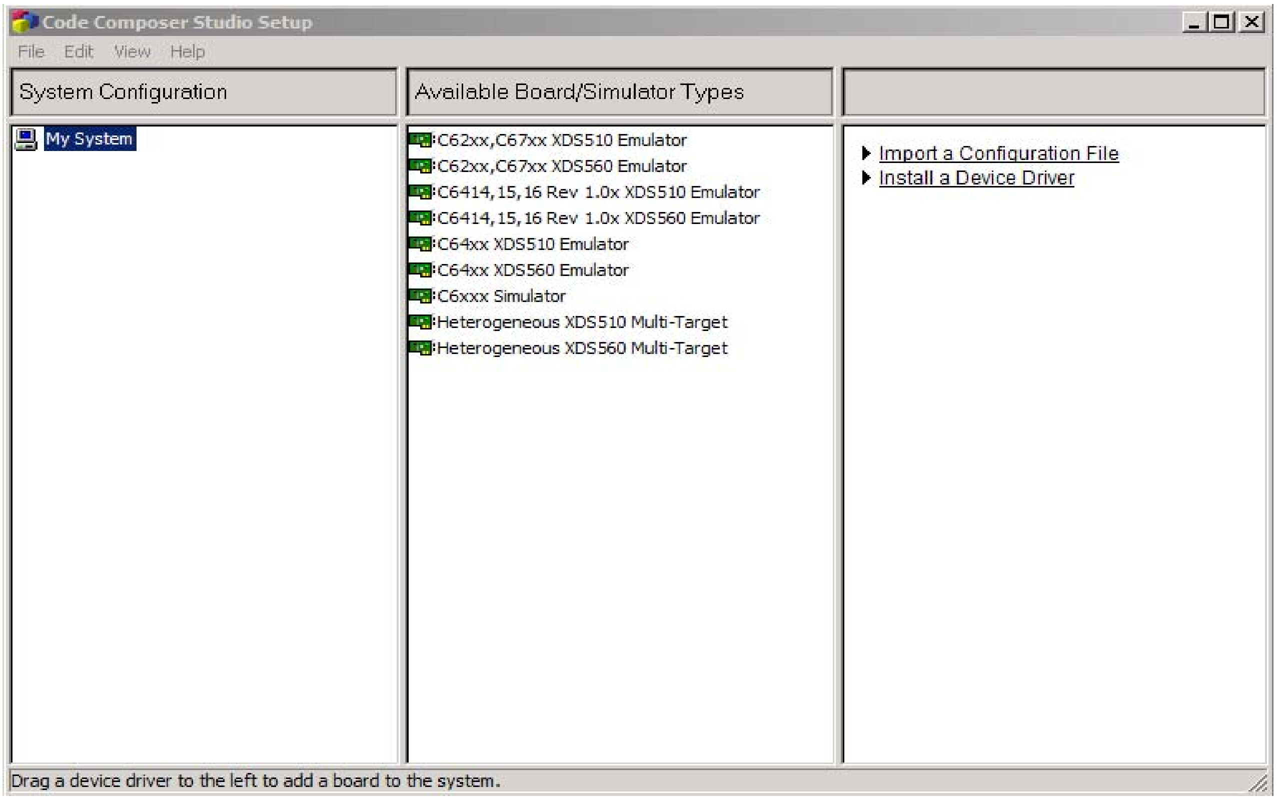
3.1.1.2 Creating a New System Configuration

To set up a new system configuration you will be working from the Code Composer Studio Setup dialog box, instead of the Import Configuration dialog box.

Start with a blank working configuration by selecting Clear from the File menu. (You may also start with a standard or imported configuration that is close to your desired system. In that case, begin at step three below after loading the starting configuration.)

Step 1: Select the My System icon in the System Configuration pane.

Step 2: In the Available Board/Simulator Types pane, select a target board or simulator that represents your system.


Code Composer Studio Setup
File Edit View Help
System Configuration
Available Board/Simulator Types
My System
Import a Configuration File
Install a Device Driver

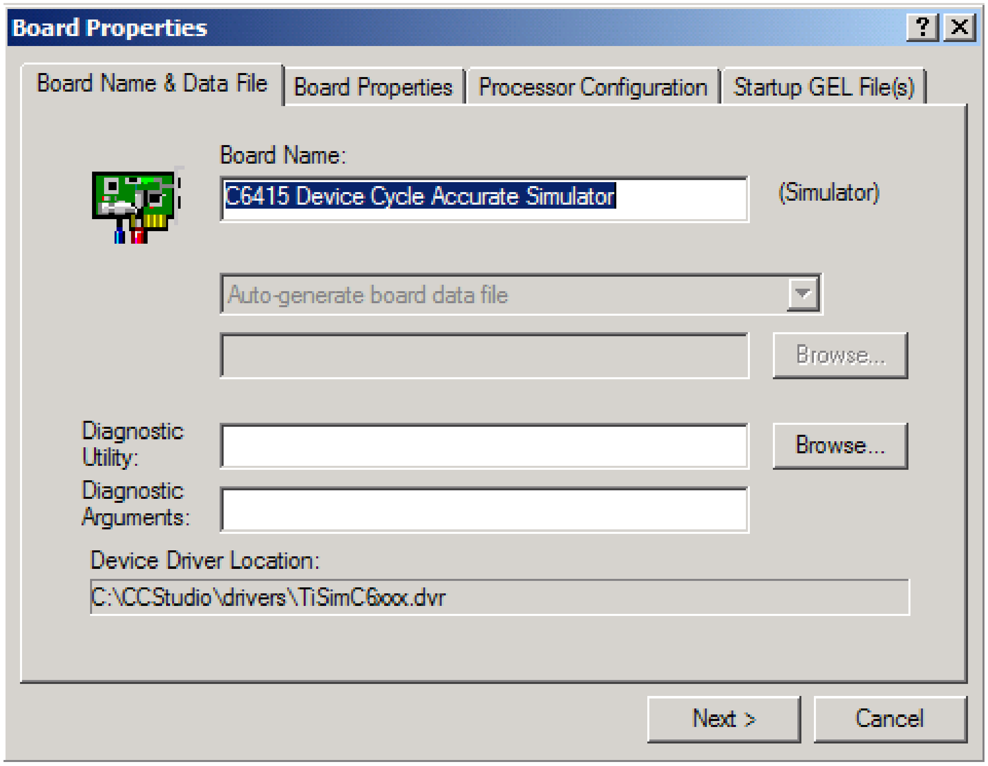
C62xx,C67xx XDS510 Emulator
C62xx,C67xx XDS560 Emulator
C6414,15,16 Rev 1.0x XDS510 Emulator
C6414,15,16 Rev 1.0x XDS560 Emulator
C64xx XDS510 Emulator
C64xx XDS560 Emulator
C6xxx Simulator
Heterogeneous XDS510 Multi-Target
Heterogeneous XDS560 Multi-Target

Drag a device driver to the left to add a board to the system.

If you want to use a target board or simulator that is not listed in the Available Board/Simulator Types pane, you must install a suitable device driver now. (For example, you may have received a device driver from a third-party vendor or you may want to use a driver from a previous version of Code Composer Studio.) Proceed to Installing/Uninstalling Device Drivers (select Help→Contents→Code Composer Studio Setup→How To…→Installing/Uninstalling Device Drivers) and then return to this section to complete your system configuration.

Step 3: Open the Board Properties dialog box using any of the following procedures:

- ❑ Double-click on the device driver in the Available Board/Simulator Types pane.
- ❑ Click on the device driver in the Available Board/Simulator Types pane and drag-and-drop the driver onto the System Configuration pane.
- ❑ Select the device driver in the Available Board/Simulator Types pane and then select the command, Add to System, in the Setup Commands/Information pane.
- ❑ Select the device driver in the System Configuration pane and then select the Properties command from the Edit menu.



Board Name: C6415 Device Cycle Accurate Simulator (Simulator)

Auto-generate board data file

Diagnostic Utility:

Diagnostic Arguments:

Device Driver Location: C:\CCStudio\drivers\TiSimC6xxx.dvr

Step 4: Edit the information in the Board Properties dialog. Board Properties is a tabbed dialog.

The tabs that appear and the fields that can be edited will differ depending on the board that you have selected.

After filling in the information in each tab, you can click the Next button to go to the next tab, or simply click on the next tab itself. When you are done, click the Finish button.

For more information on configuring the Board Properties dialog, see the online help (Help→Contents→Code Composer Studio Setup→Custom Setup).

3.1.1.3 Creating Multiprocessor Configurations

The most common configurations include a single simulator or a single target board with a single CPU. However, you can create more complicated configurations in the following ways:

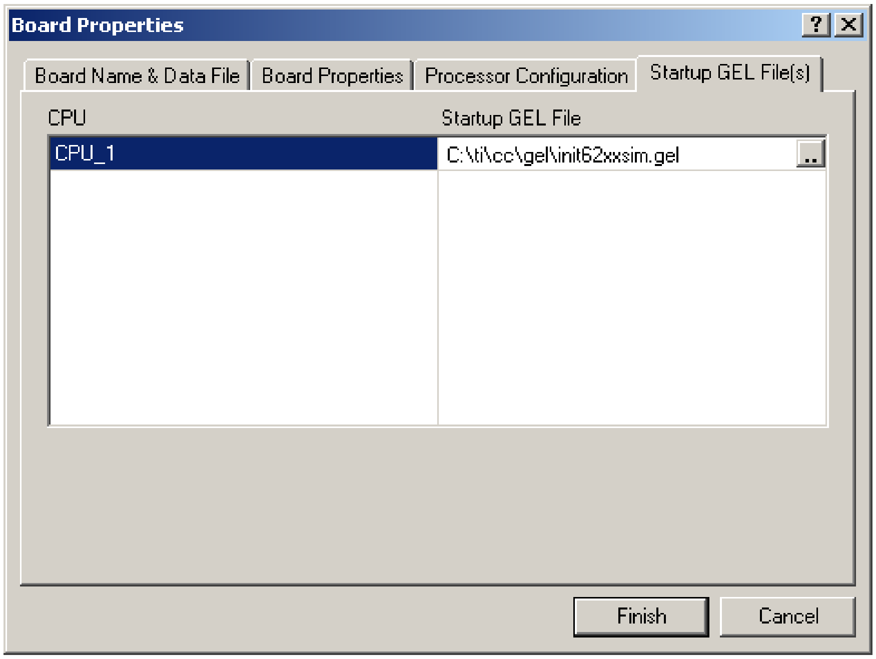
- ❑ Connect multiple emulators to your computer, each with its own target board.
- ❑ Connect more than one target board to a single emulator, using special hardware to link the scan paths on the boards.
- ❑ Create multiple CPUs on a single board, and the CPUs can be all of the same kind or they can be of different types (e.g., DSPs and microcontrollers).

Although a Code Composer Studio configuration is represented as a series of boards, in fact, each board is either a single CPU simulator or a single emulator scan chain that can be attached to one or more boards with multiple processors. The device driver associated with the board must be able to comprehend all the CPUs on the scan chain. More information may be found in the online help (Help→Contents→Code Composer Studio Setup→How To…→Configuring CCS for Heterogeneous Debugging).

3.1.1.4 Startup GEL Files

The general extension language (GEL) is an interpretive language, similar to C. GEL functions can be used to configure the Code Composer Studio development environment. They can also be used to initialize the target CPU. A rich set of built-in GEL functions is available, or you can create your own user-defined GEL functions.

The Startup GEL file(s) tab allows you to associate a GEL file (.gel) with each processor in your system configuration.



CPU	Startup GEL File
CPU_1	C:\ti\cc\gel\init62xxsim.gel

When Code Composer Studio is started, each startup GEL file is scanned and all GEL functions contained in the file are loaded. If the GEL file contains a StartUp() function, the code within that function is also executed. For example, the GEL mapping functions can be used to create a memory map that describes the processor's memory to the debugger.

```
StartUp(){  /*Everything in this function will be executed
on startup*/  GEL_MapOn();  GEL_MapAdd(0, 0, 0xF000, 1,
1);  GEL_MapAdd(0, 1, 0xF000, 1, 1);}
```

For more information, see the Code Composer Studio online help. Select Help→Contents→Creating Code and Building Your Project→Automating Tasks with General Extension Language).

3.1.1.5 Device Drivers

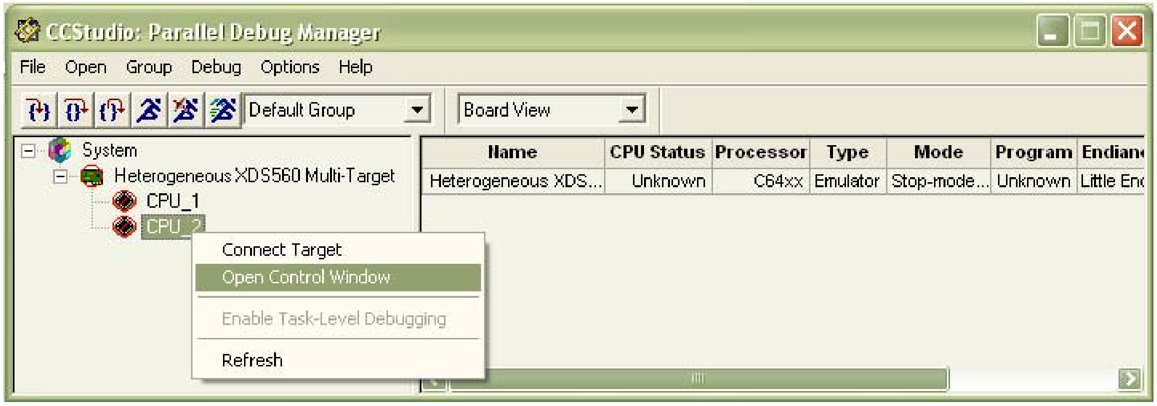
Special software modules called device drivers, are used to communicate with the target. Each driver file defines a specific target configuration: a target board and emulator, or simulator. Device drivers may either be supplied by Texas Instruments or by third-party vendors.

Each target board or simulator type listed in the Available Board/Simulator Types pane is physically represented by a device driver file. Code Composer

Studio does not support creating device drivers, but TI or third parties may ship device drivers separately from those which are pre-installed.

3.1.2 Parallel Debug Manager Plus (PDM+)

In multiprocessor configurations, invoking Code Composer Studio starts a special control known as the Parallel Debug Manager Plus (PDM+).



Name	CPU Status	Processor	Type	Mode	Program	Endian
Heterogeneous XDS...	Unknown	C64xx	Emulator	Stop-mode...	Unknown	Little End

The PDM+ allows you to open a separate Code Composer Studio session for each target device. Activity on the specified devices can be controlled in parallel using the PDM control.

The 3.0 version of Parallel Debug Manager (PDM+) has several changes from earlier versions:

❑ Users can connect or disconnect from targets “on the fly” by right-clicking the processor on the right pane of PDM+

❑ The interface allows an expanded view of processors, with several drop-down filters to reveal a list by group, by CPU or by board.

❑ Red highlighting on the processor icon (on the left pane) indicates that the processor is not connected to the system or that it has updated status information.

❑ Your can now put processors into loosely-coupled groups, (i.e., where the processors are not all on the same physical scan chain). Choosing Group View from the second dropdown menu and System on PDM’s left pane shows which groups are synchronous and which are not.

Global breakpoints work only when processors in a group belong to the same physical scan chain.

For further details on the PDM+, see the Code Composer Studio online help under Help→Contents→Debug→Advanced Debugging Features→Parallel Debug Manager(PDM).

3.1.3 Connect/Disconnect

Code Composer Studio IDE now makes it easier to connect and disconnect with the target dynamically, by using a new functionality called Connect/Disconnect. Connect/Disconnect allows you to disconnect from your hardware target and even to restore the previous debug state when connecting again.

By default, Code Composer Studio will not attempt to connect to the target when the control window is opened. Connection to the target can be established by going to Debug→Connect. The default behavior can be changed in the Debug Properties tab under Options→Customize.

The Status Bar will briefly flash a help icon to indicate changes in the target's status. When the target is disconnected, the status bar will indicate this fact, as well as the last known execution state of the target (i.e., halted, running, free running or error condition). When connected, the Status Bar will also indicate if the target is stepping (into, over or out), and the type of breakpoint that caused the halt (software or hardware).

After a connection to the target (except for the first connection), a menu option entitled Restore Debug State, will be available under the Debug Menu. Selecting this option will enable every breakpoint that was disabled at disconnect. You can also reset them normally by pressing F9 or selecting Toggle Breakpoints from the right-click menu. Breakpoints from cTools jobs and emu analysis will not be enabled.

If the PDM+ is open, you can connect to a target by right-clicking on the cell corresponding to the target device underneath the column marked, Name.

For further details on Connect/Disconnect, see the Code Composer Studio online help under Debugging→Connect/Disconnect.

3.2 Host Setup

3.2.1 IDE Customization

Once Code Composer Studio has been properly configured and the IDE launched, you can customize several of the general IDE options to adhere to your personal needs.

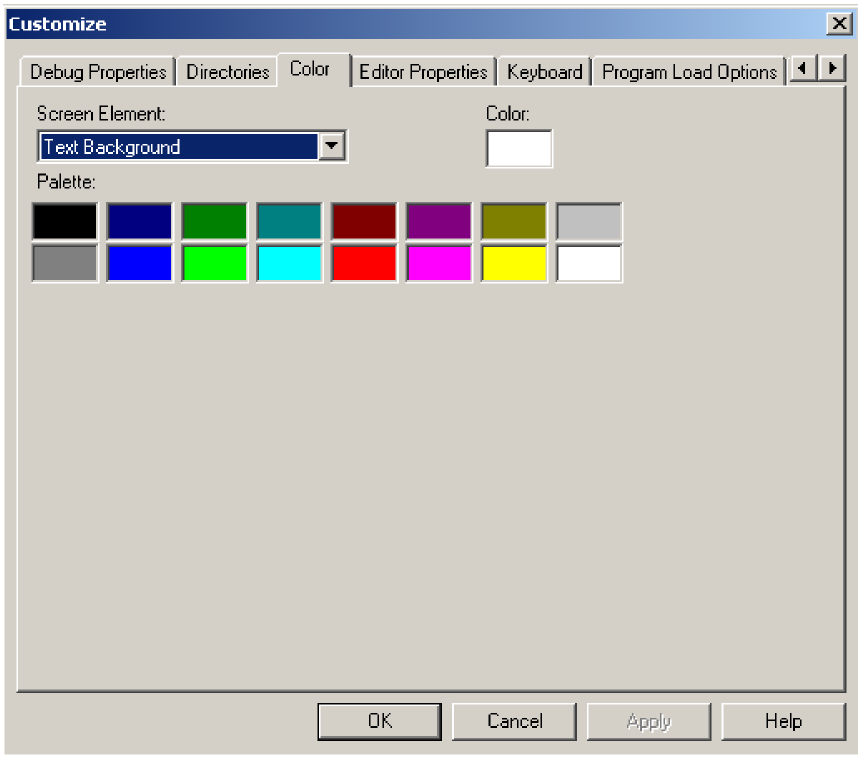
3.2.1.1 Setting Custom General IDE Options

Color

The color of various screen elements can be changed to suit your taste. All color changes are saved in the workspace.

To change the color of screen elements:

Step 1: Select Option→Customize, or right-click in a document window or Disassembly window and select Properties→Colors.



Screen Element: Text Background

Color:

Step 2: In the Customize dialog box, select the Color tab.

The color dialog offers the following options:

Screen Element. Click the drop-down list and select the screen element to change.

Color. The color field displays the current color of the selected screen element.

Palette. Select a different color from the palette. The color field is updated with the selected color.

Step 1: Click Apply to accept your selection.

Step 2: Click OK to exit the dialog box. Click Cancel to exit the dialog box without accepting your changes.

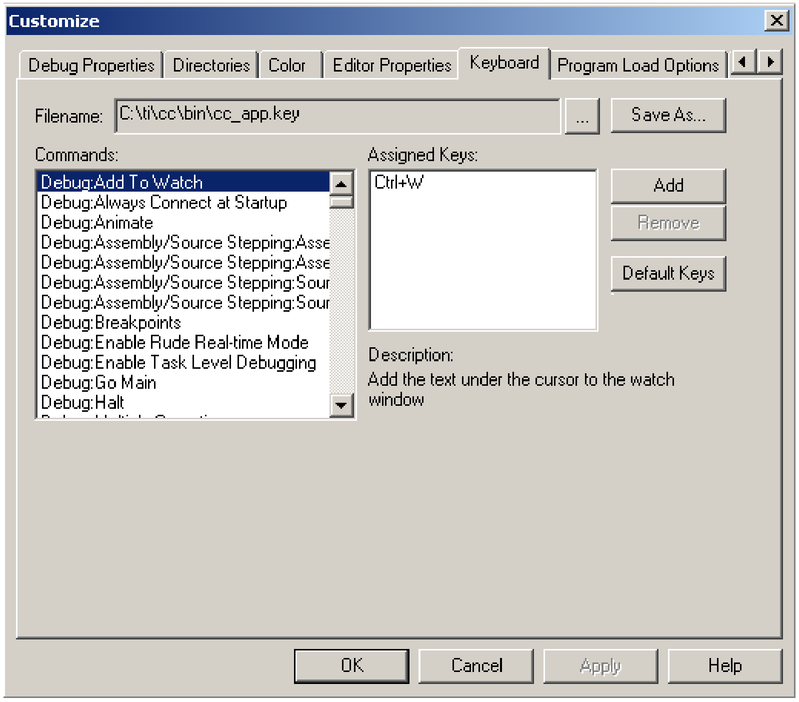
Keyboard

The default keyboard shortcuts can be changed and new keyboard shortcuts can be created for any editing or debugging commands that can be invoked from a document window.

To assign keyboard shortcuts:

Step 1: Select Option→Customize, or right−click in a document window and select Properties→Keyboard.

Step 2: In the Customize dialog box, select the Keyboard tab.



Filename: C:\ti\cc\bin\cc_app.key

Commands:

Debug:Add To Watch
Debug:Always Connect at Startup
Debug:Animate
Debug:Assembly/Source Stepping:Asse
Debug:Assembly/Source Stepping:Asse
Debug:Assembly/Source Stepping:Sour
Debug:Assembly/Source Stepping:Sour
Debug:Breakpoints
Debug:Enable Rude Real-time Mode
Debug:Enable Task Level Debugging
Debug:Go Main
Debug:Halt

Assigned Keys: Ctrl+W

Description:
Add the text under the cursor to the watch window

The Customize → Keyboard dialog offers the following options:

Filename. By default, the file that contains the standard keyboard shortcuts is displayed. To load a previously saved keyboard configuration file (*.key), enter the path and filename, or use the browse button (…) to navigate to the file.

Commands. Select the command you want to assign to a keyboard shortcut.

Assigned Keys. Displays the keyboard shortcuts that are assigned to the selected command.

Add. Click the Add button to assign a new key sequence for invoking the selected command. In the Assign Shortcut dialog box, enter the new key sequence, and then click OK.

Remove. To remove a particular key sequence for a command, select the key sequence in the Assigned Keys list and click the Remove button.

Default Keys. Immediately revert back to the default keyboard shortcuts by clicking the Default Keys button.

Save As. Click the Save As button to save your custom keyboard configuration in a file. In the Save As dialog box, navigate to the location where you want to save your configuration. Enter a name for your keyword configuration file (*.key). Click Save.

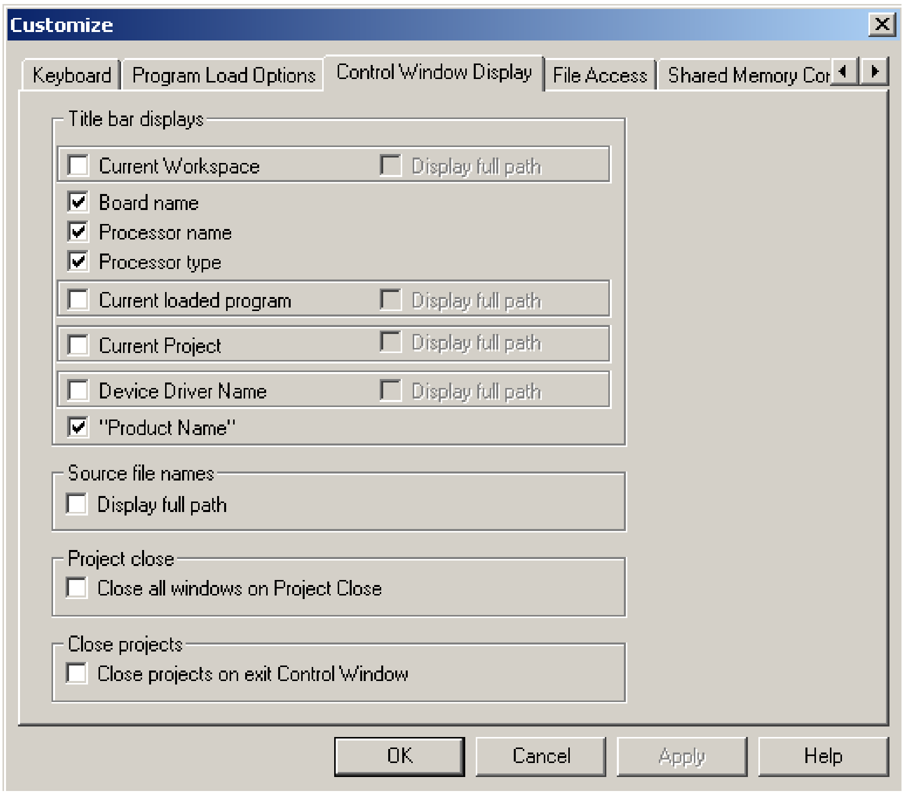
Step 3: Click OK to exit the dialog box.

Control Window Display

To set display options:

Step 1: Select Option→Customize.

Step 2: In the Customize dialog box, select the Control Window Display tab.



Customize

Title bar displays
- ☐ Current Workspace ☐ Display full path
- ☑ Board name
- ☑ Processor name
- ☑ Processor type
- ☐ Current loaded program ☐ Display full path
- ☐ Current Project ☐ Display full path
- ☐ Device Driver Name ☐ Display full path
- ☑ "Product Name"

Source file names
- ☐ Display full path

Project close
- ☐ Close all windows on Project Close

Close projects
- ☐ Close projects on exit Control Window

The Control Window Display dialog offers the following options.

Title bar displays

The following options control the information that is displayed in the title bar of the Control window:

Current Workspace. Displays the name of the current workspace. If selected, you can choose to display just the filename or display the full path.

Board name. Displays the name of the target board. This option is selected by default.

Processor name. Displays the name of the target processor. This option is selected by default.

Processor type. Displays the type of target processor. This option is selected by default.

Current loaded program. Displays the name of the current loaded program. If selected, you can choose to display just the output filename or display the full path.

Current Project. Displays the name of the active project. If selected, you can choose to display just the project filename or display the full path.

Device Driver Name. Displays the name of the device driver. If selected, you can choose to display just the device driver filename or display the full path.

Product Name. Displays the name of the Code Composer Studio product. By default, this option is selected.

Source file names

The following option controls the information that is displayed in the title bar of document windows and the Build Options dialog box:

Display full path. Displays the full path and filename. By default, only the filename is displayed.

Project close

Close all windows on Project Close. When a project is closed, close all document windows associated with the project. If a file has been modified, you will be prompted to save your changes. By default, associated windows are not closed.

Close projects

Close projects on exit Control Window. This option is only significant when using a multiprocessor setup. Starting Code Composer Studio with a multiprocessor setup opens the Parallel Debug Manager (PDM). From the PDM, you can launch a control window for each defined processor.

With this option disabled, projects remain open even when the control window is closed. For example, if you open a control window from the PDM, load a project, exit the control window, and then reopen the control window, you will see that the project is still open.

With this option enabled, projects that are opened within a control window are closed when you exit the control window.

Without a multiprocessor setup, closing the control window exits Code Composer Studio, which always closes all projects.

Step 1: Click OK to accept your selections and close the Customize dialog box.

File Access

The number of files and the format of the file names listed in the recent files list within the Code Composer Studio interface can be changed to suit your taste.

To Change the File Access Options:

Step 1: Select Option→Customize.

Step 2: In the Customize dialog box, select the File Access tab. Use the scroll arrows at the top of the dialog box to locate the tab.

The File Access dialog offers the following options:

Source files. Enter the maximum number of recent source files (File→Recent Source Files) to display. The value must be an integer in the range 1 to 10. The default value is 4.

Programs. Enter the maximum number of recent program files (File→Recent Program Files) to display. The value must be an integer in the range 1 to 10. The default value is 4.

Symbols. Enter the maximum number of recent symbol files (File→Recent Symbol Files) to display. The value must be an integer in the range 1 to 10. Default value is 4.

GEL files. Enter the maximum number of recent GEL files (File→Recent GEL Files) to display. The value must be an integer in the range 1−10. Default value is 4.

Projects. Enter the maximum number of recent project files (Project→Recent project files) to display. The value must be an integer in the range 1 to 10. The default value is 4.

Workspaces. Enter the maximum number of recent workspaces (File →Recent Workspaces) to display. The value must be an integer in the range 1 to 10. The default value is 4.

Reset file directories when opening a project. If you choose this option, when you try to open a file, Code Composer Studio will start you inside the directory of your active project. If you don't choose this option, Code Composer Studio will start you inside the last directory you used, regardless of which project you are in now.

Show time stamp for program files in recent file list. Enabling this option will display the time stamp along with the file name of each program file in the recent file list.

Step 3: Click OK to exit the Customize dialog box.

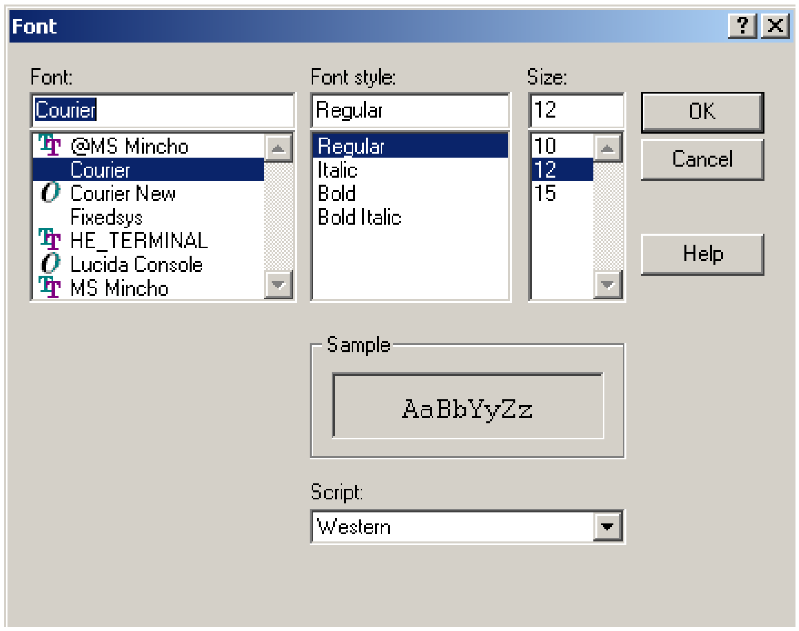
Font

The typeface, style, size, and color of the default font can be changed to suit your preference.

To restore the original font and font characteristics, select Courier font, Regular style, size 12.

To change fonts and font characteristics:

Step 1: Select Option→Font, or right-click in a document window or Disassembly window and select Properties→Fonts.



The Font dialog box offers the following options.

Font. Select a font from the list.

Font style. Select a style for the chosen font. The styles that are displayed vary depending on the selected font.

Size. Select a font size from the list. The sizes vary depending on the selected font.

Sample. The Sample field displays the selected font and font characteristics as they will appear within the IDE.

Script. Select a language script from the drop-down list.

Step 2: Click OK.

Chapter 4

Code Creation

This chapter gives a brief look at the options available to create code and build a basic Code Composer Studio project.

Topic **Page**

4.1 Create and Configure Project

A project stores all the information needed to build an individual program or library.

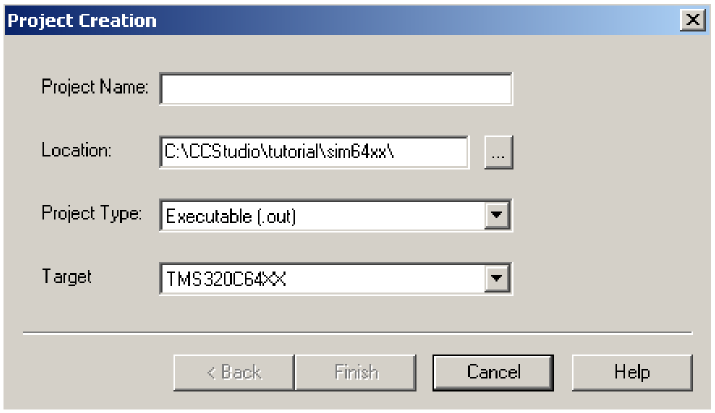
- ❑ Filenames of source code and object libraries
- ❑ Code generation tool options
- ❑ Include file dependencies

4.1.1 Creating a Project

The following procedure allows you to create single or multiple new projects (multiple projects can be open simultaneously). Each project's filename must be unique.

The information for a project is stored in a single project file (*.pjt).

Step 1: From the Project menu, choose New.
The Project Creation wizard window displays.


Project Creation

Project Name:

Location: C:\CCStudio\tutorial\sim64xx\

Project Type: Executable (.out)

Target: TMS320C64XX

Step 2: In the Project Name field, type the name you want for your project.

Step 3: In the Location field, specify the directory where you want to store the project file, Object files generated by the compiler and assembler are also stored here.
You can type the full path in the Location field or click the Browse button and use the Choose Directory dialog box. It is a good idea to use a different directory for each new project.

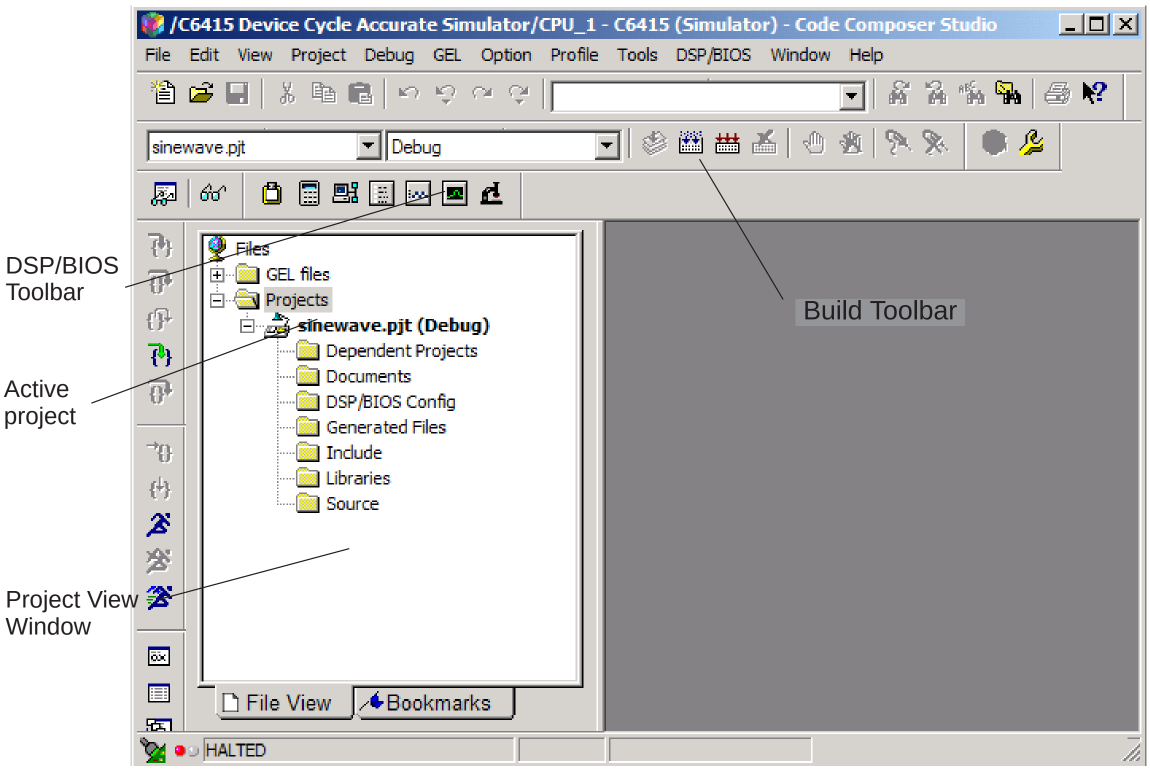
Step 4: In the Project Type field, select a Project Type from the drop-down list.
Choose either Executable (.out) or Library (lib). Executable indicates that the project generates an executable file. Library indicates that you are building an object library.

Step 5: In the Target field, select the target family that identifies your CPU. This information is necessary when tools are installed for multiple targets.

Step 6: Click Finish.
A project file called *projectname*.pjt is created. This file stores all files and project settings used by your project.

The new project and first project configuration (in alphabetical order) become the active project, and inherit TI-supplied default compiler and linker options for debug and release configurations.

Figure 4−1. Code Composer Studio IDE Basic Window



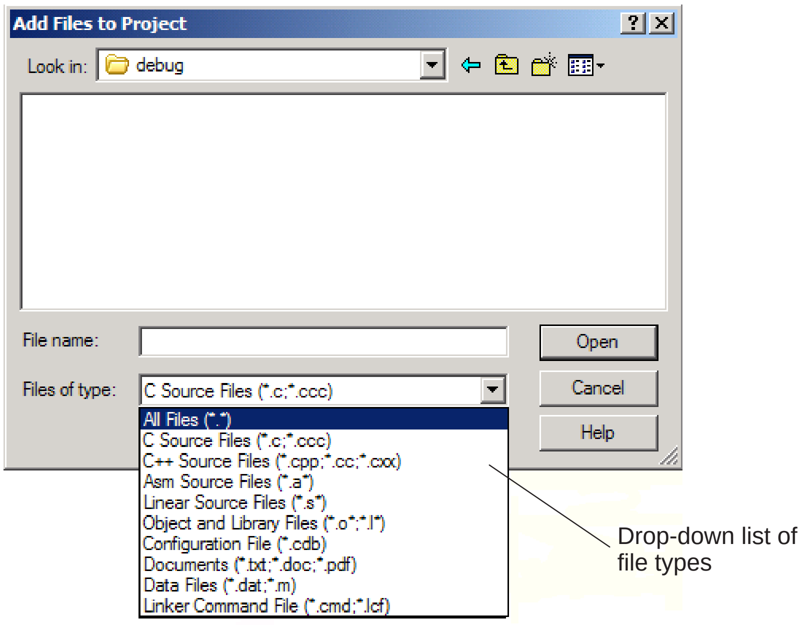
After creating a new project file, add the filenames of your source code, object libraries, and linker command file to the project list.

4.1.1.1 Adding Files to a Project

You can add several different files or file types to your project. The types are shown in the graphic below. To add files to your project:

Step 1: Select Project→Add Files to Project, or right-click on the project's filename in the Project View window and select Add Files.

The Add Files to Project dialog box displays.


Add Files to Project
Look in:
debug
File name:
Files of type:
C Source Files (*.c;*.ccc)
Open
Cancel
Help
All Files (*.*)
C Source Files (*.c;*.ccc)
C++ Source Files (*.cpp;*.cc;*.cxx)
Asm Source Files (*.a*)
Linear Source Files (*.s*)
Object and Library Files (*.o*;*.l*)
Configuration File (*.cdb)
Documents (*.txt;*.doc;*.pdf)
Data Files (*.dat;*.m)
Linker Command File (*.cmd;*.lcf)
Drop-down list of file types

Step 2: In the Add Files to Project dialog box, specify a file to add. If the file does not exist in the current directory, browse to the correct location. Use the Files of Type drop-down list to set the type of files that appear in the File name field.

Note:

Do not try to manually add header/include files (*.h) to the project. These files are automatically added when the source files are scanned for dependencies as part of the build process.

Step 3: Click Open to add the specified file to your project.

The Project View (see Figure 4−1) is automatically updated when a file is added to the current project.

The project manager organizes files into folders for source files, include files, libraries, and DSP/BIOS configuration files. Source files that are generated by DSP/BIOS are placed in the Generated Files folder. Code Composer Studio IDE finds files by searching for project files in the following path order when building the program:

- ❑ The folder that contains the source file.

- ❑ The folders listed in the Include search path for the compiler or assembler options (from left to right).
- ❑ The folders listed in the definitions of the optional DSP_C_DIR (compiler) and DSP_A_DIR (assembler) environment variables (from left to right).

Removing a File

If you need to remove a file from the project, right-click on the file in the Project View and choose Remove from Project in the pop-up menu.

4.1.2 Project Configurations

A project configuration defines a set of project level build options. Options specified at this level apply to every file in the project.

Project configurations enable you to define build options for the different phases of program development. For example, you can define a Debug configuration to use while debugging your program and a Release configuration for building the finished product.

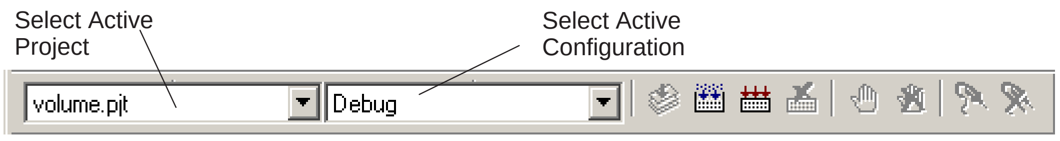
Each project is created with two default configurations: Debug and Release. Additional configurations can be defined. Whenever a project is created or an existing project is initially opened, the first configuration (in alphabetical order) is set to active and is preserved the Code Composer Studio workspace.

When you build your program, the output files generated by the software tools are placed in a configuration-specific subdirectory. For example, if you have created a project in the directory MyProject, the output files for the Debug configuration are placed in MyProject\Debug. Similarly, the output files for the Release configuration are placed in MyProject\Release.

Changing the Active Project Configuration

Click on the Select Active Configuration field in the Project toolbar and select a configuration from the drop-down list.

Figure 4−2. Changing Active Project Configuration

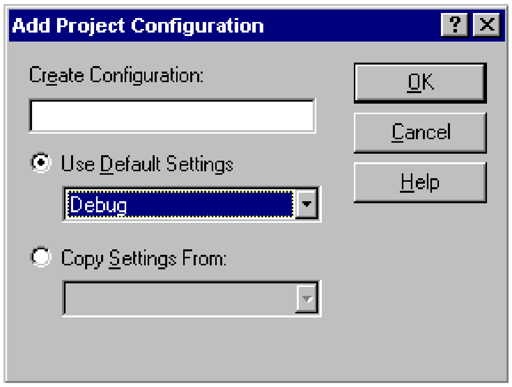


Adding a New Project Configuration

Step 1: Select Project→Configurations, or right-click on the project's filename in the Project View window and select Configurations.

Step 2: In the Project Configurations dialog box, click Add.

The Add Project Configuration window displays.


Add Project Configuration

Create Configuration:

● Use Default Settings: Debug

○ Copy Settings From:

Step 3: In the Add Project Configuration dialog box, specify the name of the new configuration in the Create Configuration field, and choose to Use Default Settings (build options) or Copy Settings from an existing configuration to populate your new configuration.

Step 4: Click OK to accept your selections and exit the Add Project Configuration dialog.

Step 5: Click Close to exit the Project Configurations dialog.

Step 6: Modify your new configuration using the build options dialog found in the Project menu.

4.1.3 Project Dependencies

The project dependencies tool provides an easy way to manage and build more complex projects. Project dependencies allow you to break a large project into multiple smaller projects and to subsequently create the final project using dependencies between projects. **Note:** Subprojects are always built first, since the main project depends on them.

Adding/Creating a sub−project

The three ways to create a subproject, or, to be more specific, to create a project dependency relationship are discussed in the following topics.

First Method: Drag−and−drop from the project view windows.

You can drop the sub-project to the target project icon or to the Dependent Projects icon under the target project. You can drag and drop from within the same project view window, or you can you drag and drop between project view windows of two Code Composer Studios running simultaneously.

Second Method: Drag−and−drop from Windows File Explorer.

Step 1: Choose the .pjt file from the project you want to be a subproject.

Step 2: Open Windows Explorer so that both Explorer and Code Composer Studio are visible at the same time.

Step 3: In Windows Explorer, select the .pjt file of the project you want to be a subproject.

Step 4: Drag this .pjt file to the Project Window of Code Composer Studio. A plus sign should appear on the .pjt file that you were dragging.

Step 5: Drop it into the Project Dependency folder of the main project.

Third Method: Use the Context Menu

In the project view, right-click on the Dependent Projects icon under a loaded project, select Add Dependent Projects. In the pop-up dialog, browse and select another project .pjt file. The selected .pjt file will be a sub-project of the loaded project. If the selected .pjt file is not yet loaded, it will be automatically loaded.

Project Dependencies Settings

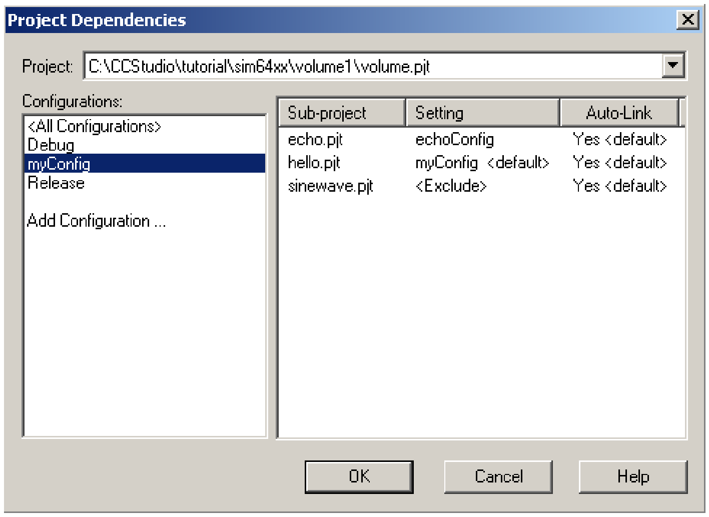
Sub-projects each have their own configuration settings. In addition, the main project has configuration settings for each sub-project. All of these settings can be accessed from the Project Dependencies dialogue. The dialogue can be accessed by the project menu as well as the context menu of the project. You simply click on Project Dependencies… to access the dialogue.

Modifying Project Configurations

In the Project Dependencies dialogue, it is possible to modify the subprojectsettings. As mentioned previously, the dialogue can be accessed by

Project→Project Dependencies.

As shown by Figure 4−3, you can choose to exclude certain subprojects from your configuration. In the example shown, the myConfig configuration for volume. pjt excludes sinewave.pjt from the build. In addition, you can also select a particular subproject configuration for this configuration. In myConfig, echo.pjt is built using a user-created configuration echoConfig rather than the default, myConfig subproject configuration.



Project: C:\CCStudio\tutorial\sim64xx\volume1\volume.pjt

Configurations:
<All Configurations>
Debug
myConfig
Release

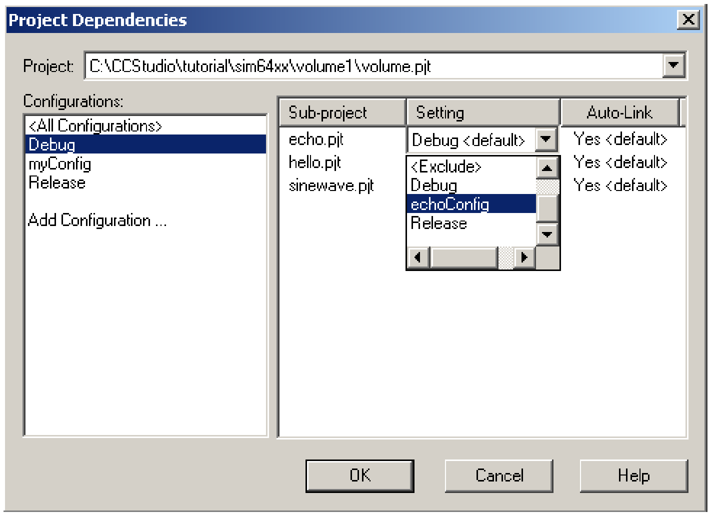
Add Configuration ...

Sub-project	Setting	Auto-Link
echo.pjt	echoConfig	Yes <default>
hello.pjt	myConfig <default>	Yes <default>
sinewave.pjt	<Exclude>	Yes <default>

Sub−project configurations

Each sub-project has its own set of build configurations. For each main project configuration, you can choose to build each sub-project using a particular configuration. To modify the sub-project setting, use the dropdown box besides the project (under the settings column). Take the example shown in

Figure 4−4, you can change the subproject echo.pjt configuration to globsConfig in the Debug configuration of volume.pjt.



Project: C:\CCStudio\tutorial\sim64xx\volume1\volume.pjt

Configurations:
<All Configurations>
Debug
myConfig
Release
Add Configuration ...

Sub-project	Setting	Auto-Link
echo.pjt	Debug <default>	Yes <default>
hello.pjt		Yes <default>
sinewave.pjt		Yes <default>

<Exclude>
Debug
echoConfig
Release

4.1.4 Makefiles

The Code Composer Studio IDE supports the use of external makefiles (*.mak) and an associated external make utility for project management and build process customization.

To enable the Code Composer Studio IDE to build a program using a makefile, a Code Composer Studio project must be created that wraps the makefile. After a Composer Composer Studio project is associated with the makefile, the project and its contents can be displayed in the Project View window and the Project→Build and Project→Rebuild All commands can be used to build the program.

Step 1: Double-click on the name of the makefile in the Project View window to open the file for editing.

Step 2: Modify your makefile build commands and options.

Special dialogs enable you to modify the makefile build commands and makefile options. The normal Code Composer Studio Build Options dialogs are not available when working with makefiles.

Multiple configurations can be created, each with its own build commands and options.

Limitations and Restrictions

Source files can be added to or removed from the project in the Project View. However, changes made in the Project View do not change the contents of the makefile. These source files do not affect the build process nor are they reflected in the contents of the makefile. Similarly, editing the makefile does not change the contents in the Project View. File-specific options for source files that are added in the Project View are disabled. The Project→Compile File command is also disabled. However, when the project is saved, the current state of the Project View is preserved.

Note:

Before using Code Composer Studio IDE commands to build your program using a makefile, it is necessary to set the necessary environment variables. To set environment variables, run the batch file

DosRun.bat

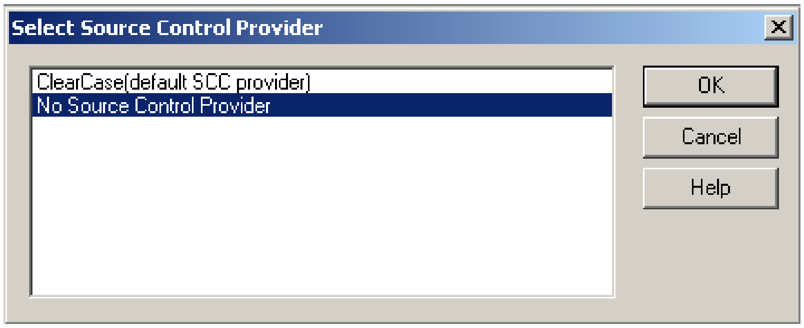
The batch file is located in the directory c:\CCStudio. If you installed Code Composer Studio IDE in a directory other than c:\ti, the batch file will be located in the directory you specified during installation.

4.1.5 Source Control Integration

The project manager enables you to connect your projects to a variety of source control providers. The Code Composer Studio IDE automatically detects compatible providers that are installed on your computer.

Step 1: From the Project menu, choose Source Control.

Step 2: From the Source Control submenu, choose Select Provider...


Select Source Control Provider
ClearCase(default SCC provider)
No Source Control Provider
OK
Cancel
Help

Step 3: Select the Source Control Provider that you want to use and press OK.

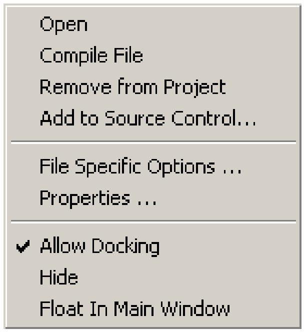
NOTE: If no source control providers are listed, please ensure that you have correctly installed the client software for the provider on your machine.

Step 4: Open one of your projects and select Add to Source Control from Project→Source Control.

Step 5: Add your source files to Source Control.

You can check files in and out of source control by selecting a file in the Project View window and right clicking on the file.

Figure 4−3. Source Control Pop-Up Menu



4.2 Configuring DSP/BIOS

The DSP/BIOS Configuration Tool enables developers to select and deselect kernel modules, and control a wide range of configurable parameters accessed by the DSP/BIOS kernel at run-time as shown in the figure below. A file of data tables generated by the tool ultimately becomes an input to the program linker.

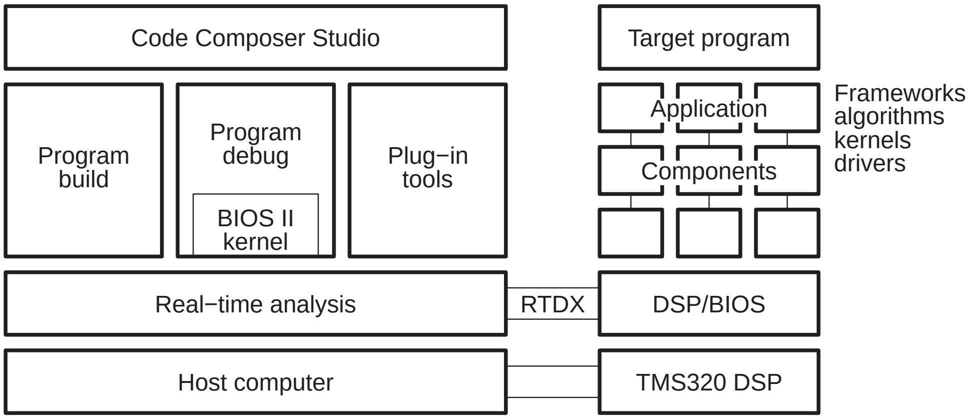
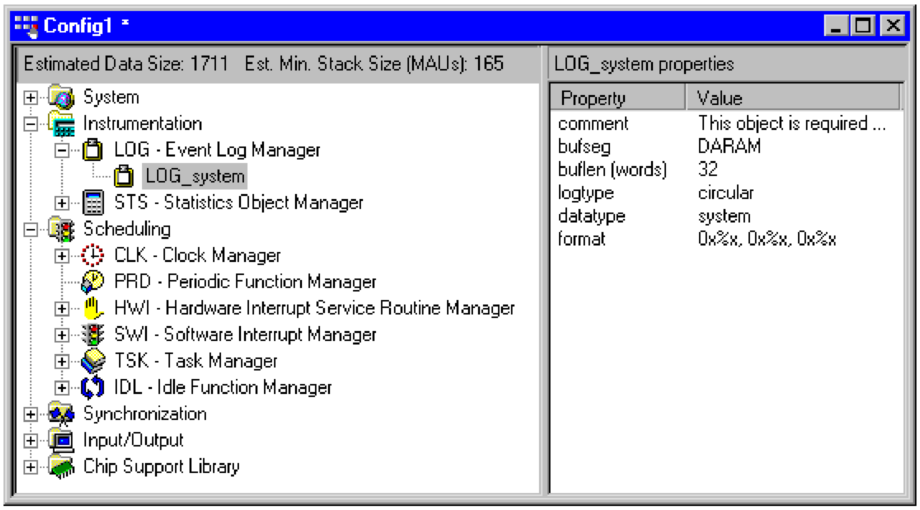


Figure 4−4. DSP/BIOS Configuration Window


Config1 *
Estimated Data Size: 1711 Est. Min. Stack Size (MAUs): 165
System
Instrumentation
LOG - Event Log Manager
LOG_system
STS - Statistics Object Manager
Scheduling
CLK - Clock Manager
PRD - Periodic Function Manager
HWI - Hardware Interrupt Service Routine Manager
SWI - Software Interrupt Manager
TSK - Task Manager
IDL - Idle Function Manager
Synchronization
Input/Output
Chip Support Library
LOG_system properties

Property	Value
comment	This object is required ...
bufseg	DARAM
buflen (words)	32
logtype	circular
datatype	system
format	0x%x, 0x%x, 0x%x

The DSP/BIOS Configuration Tool (see Figure 4−4) serves as a special-purpose visual editor for creating and assigning attributes to individual run-time kernel objects (threads, streams, etc.) used by the target application program in conjunction with DSP/BIOS API calls. The Configuration Tool provides developers the ability to statically declare and configure DSP/BIOS kernel objects during program development rather than during program execution. Declaring these kernel objects through the Configuration Tool produces static objects which exist for the duration of the program. DSP/BIOS kernel also allows dynamic creation and deletion for many of the kernel objects during program execution. However, dynamically created objects require additional code to support the dynamic operations. Statically declared objects minimize memory footprint since they do not include the additional create code.

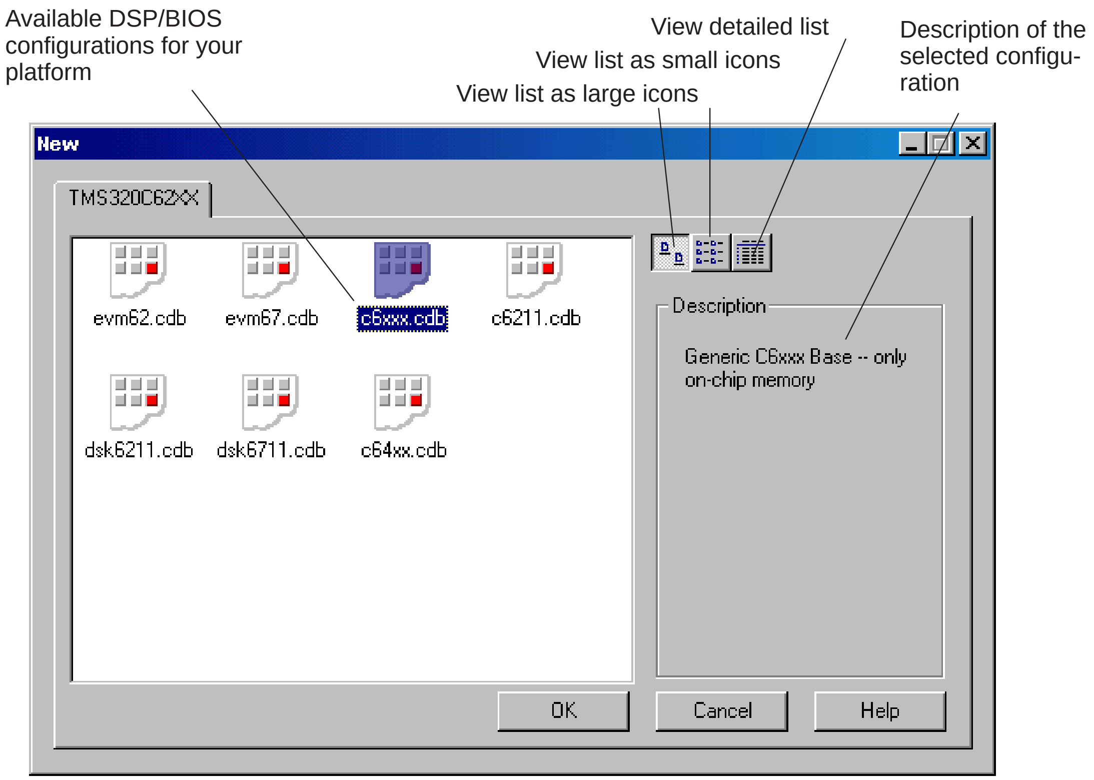
Another important benefit of static configuration is the potential for static program analysis by the DSP/BIOS Configuration Tool. In addition to minimizing the target memory footprint, the DSP/BIOS Configuration Tool provides the means for early detection of semantic errors through the validation of object attributes, prior to program execution. When the configuration tool is aware of all target program objects prior to execution, it can accurately compute and report such information as the total amount of data memory and stack storage required by the program.

Creating DSP/BIOS Configuration Files

To create DSP/BIOS configuration files:

Step 1: Within Code Composer Studio, choose File→New→DSP/BIOS Configuration.

The New Configuration window displays.

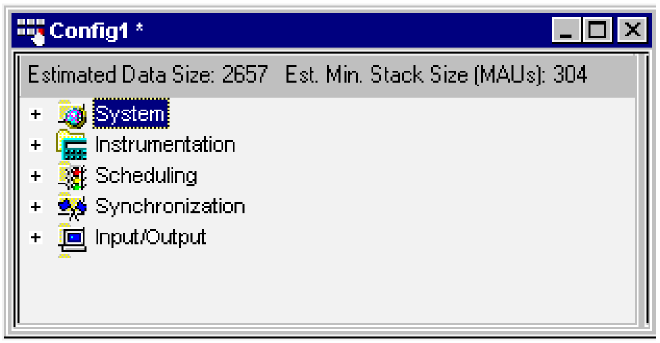


Step 2: Select a Configuration template.

If your board is not listed, you can create and add a custom template to this list.

Step 3: Click OK to create the new configuration.

The Configuration window displays.



Step 4: In the Configuration window, perform the following tasks as required by your application:

- Create objects to be used by the application.
- Name the objects.
- Set global properties for the application.
- Modify module manager properties.
- Modify object properties.
- Set priorities for software interrupts and tasks.

See Help→Contents→DSP/BIOS→DSP/BIOS API Modules for details on implementation of APIs.

Step 5: Save the configuration.

Step 6: Add the DSP/BIOS configuration file(s) to your project as described in the next procedure.

Adding DSP/BIOS Configuration Files to Your Project

After you save a DSP/BIOS configuration file, follow these steps to add files to your Code Composer Studio project.

Step 1: If it is not already open, use Project→Open to open the project with Code Composer Studio.

Step 2: Choose Project→Add Files to Project. In the Files of type box, select Configuration File (*.cdb). Select the .cdb file you saved and click Open.

Adding the .cdb file to a project automatically adds the following file to the Project View folders:

- program.cdb in the DSP/BIOS Config folder
- programcfg.s62 in the Generated Files folder
- programcfg_c.c in the Generated Files folder

Step 3: Choose Project→Add Files to Project again. In the Files of type box, select Linker Command File (*.cmd). Select the *cfg.cmd file the Configuration Tool generated when you saved the configuration file and click Open.

Step 4: If your project already contained a linker command file, you may want to modify the file or remove it from your project. It may duplicate or conflict with some of the linker commands in the file generated by DSP/BIOS.

Step 5: If your project includes the vectors.asm source file, right-click on the file and choose Remove from project in the shortcut menu. Hardware interrupt vectors are automatically defined in the configuration file.

Step 6: If your project includes the rts*xxxx*.lib file (where *xxxx* is your device or device’s generation), right−click on the file and choose Remove from project in the shortcut menu. This file is automatically included by the linker command file created from your configuration.

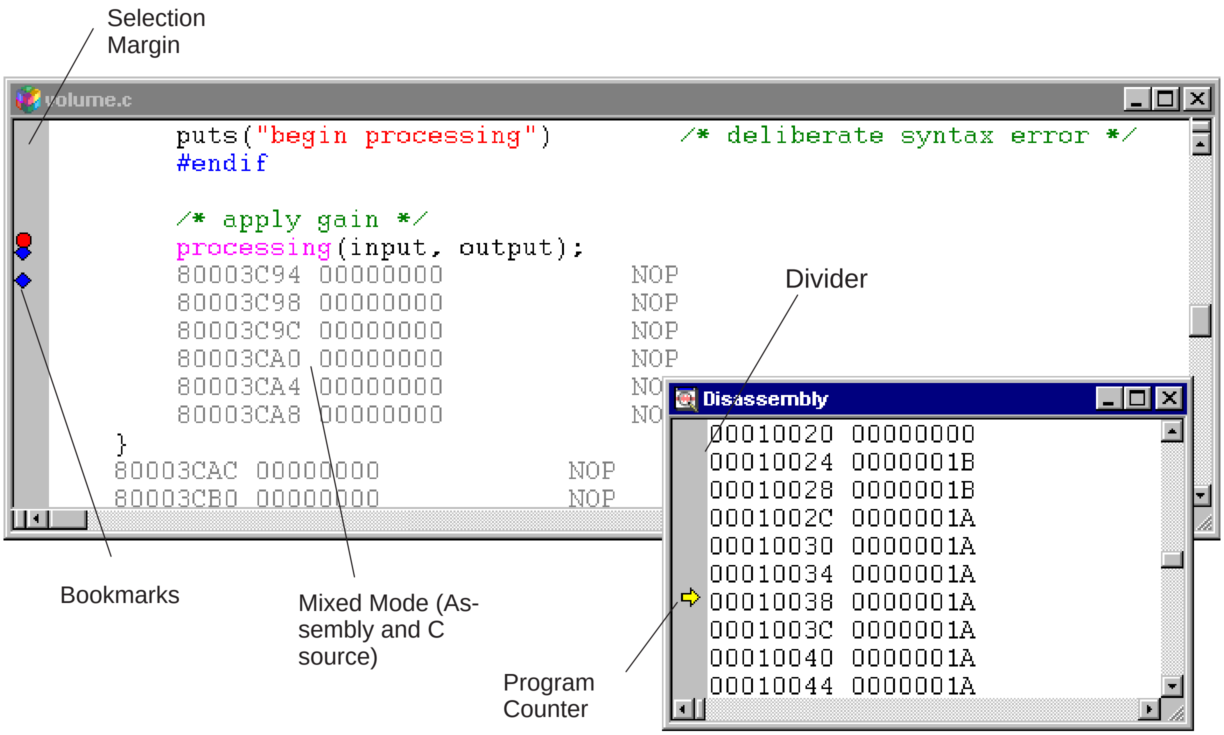
These steps can be used whenever you want to convert an existing program to one that can call DSP/BIOS API functions.

4.3 Editor

4.3.1 Using the Code Composer Studio Editor

Double-click on the *filename*.c file in the Project View to display the source code in the right half of the Code Composer Studio window.

Figure 4−5. View Source Code



```
        puts("begin processing")          /* deliberate syntax error */
        #endif

        /* apply gain */
        processing(input, output);
        80003C94  00000000             NOP
        80003C98  00000000             NOP
        80003C9C  00000000             NOP
        80003CA0  00000000             NOP
        80003CA4  00000000             NO
        80003CA8  00000000             NO
    }
    80003CAC  00000000             NOP
    80003CB0  00000000             NOP
```

```
00010020  00000000
00010024  0000001B
00010028  0000001B
0001002C  0000001A
00010030  0000001A
00010034  0000001A
00010038  0000001A
0001003C  0000001A
00010040  0000001A
00010044  0000001A
```

- ❑ **Selection Margin.** By default, a Selection Margin is displayed on the left-hand side of integrated editor and Disassembly windows. Colored icons in the Selection Margin indicate that a breakpoint (red) or Probe Point (blue) is set at this location. A yellow arrow identifies the location of the Program Counter (PC).

  TIP: The Selection Margin can be resized by dragging the divider.

- ❑ **Keywords.** The integrated editor features keyword highlighting. Keywords, comments, strings, assembler directives, and GEL commands are highlighted in different colors.

  TIP: In addition, new sets of keywords can be created, or the default keyword sets can be customized and saved in keyword files (*.kwd).

- ❑ **Keyboard Shortcuts.** The default keyboard shortcuts can be changed and new keyboard shortcuts can be created for any editing or debugging commands that can be invoked from a document window. Keyboard shortcuts can be modified through the customize dialog box in the Options menu.
- ❑ **Bookmarks.** Use bookmarks to find and maintain key locations within your source files. A bookmark can be set on any line of any source file.

Editor Properties

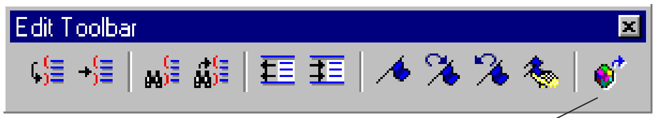
Customize the editor properties to suit your personal preferences. For example, you can choose to open files as read-only, or always save your source code before starting a build, or change the way keywords appear in the editor. You can customize the editor by going to Options→Customize→Editor properties.

4.3.2 Using an External Editor

The Code Composer Studio IDE supports the use of an external (third-party) text editor in place of the default integrated editor. After an external editor is configured and enabled, it is launched whenever a new blank document is created or an existing file is opened. You can configure an external editor by selecting Options→Customize→Editor Properties.

An external editor can only be used to edit files. The integrated editor must be used to debug your program.

Figure 4−6. External Editor Icon



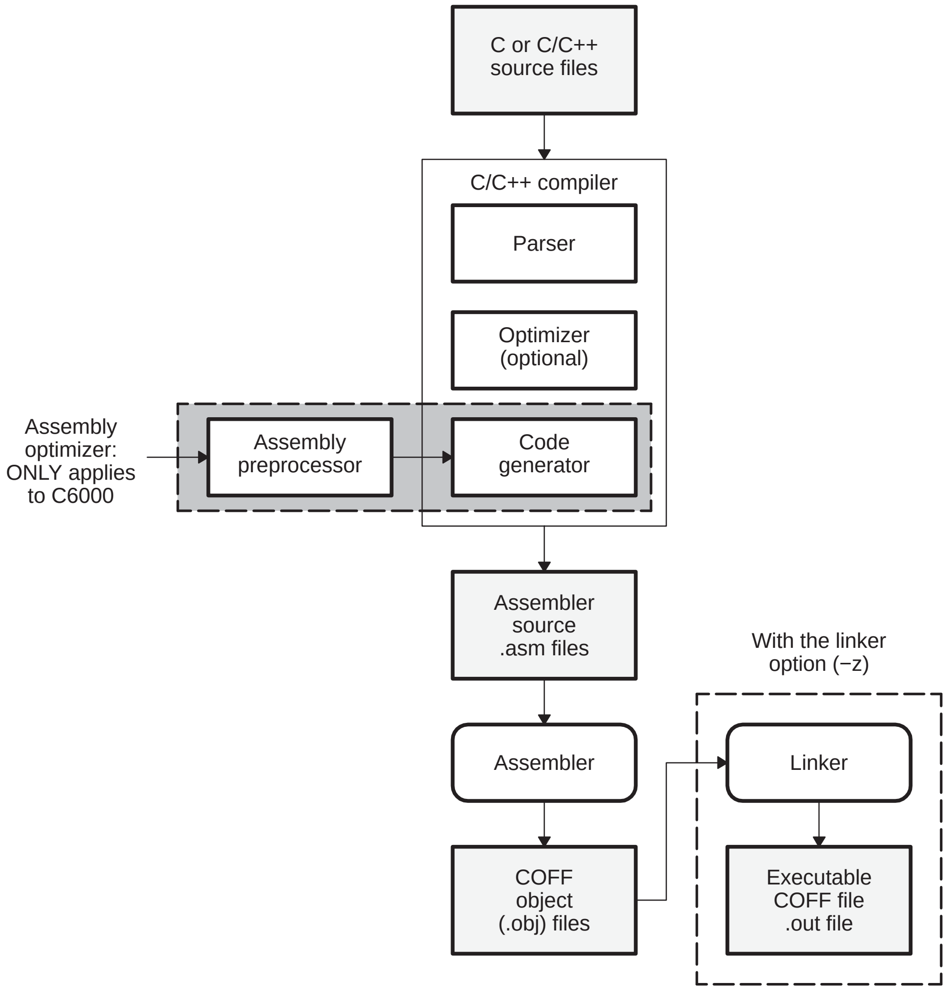
External Editor icon: toggle between an external editor and the Code Composer Studio integrated editor

4.4 Code Generation Tools

4.4.1 Code Development Flow

Code generation tools include an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities. The figure below shows you how these tools and utilities work together when you generate code.

Figure 4−7. Code Development Flow



4.4.2 Project Build Options

A graphical interface is provided for using the code generation tools.

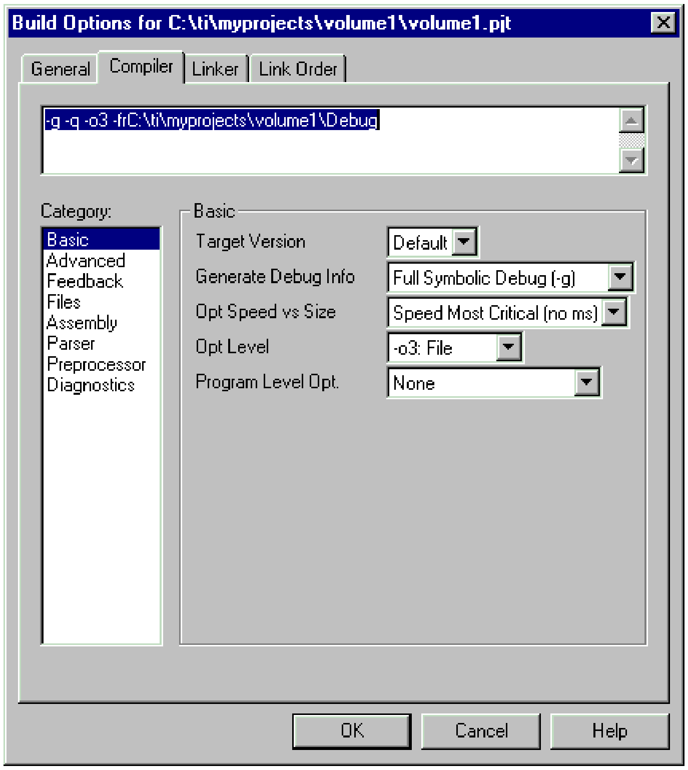
A Code Composer Studio project keeps track of all information needed to build a target program or library. A project records:

❏ Filenames of source code and object libraries
❏ Compiler, assembler, and linker options
❏ Include file dependencies

When you build a project, the appropriate code generation tools are invoked to compile, assemble, and/or link your program.

The compiler, assembler, and linker options can be specified within the Build Options dialog box (Figure 4−8). Nearly all command line options are represented within this dialog box. Options that are not represented can be specified by typing the option directly into the editable text box that appears at the top of the dialog box.

Figure 4−8. Build Options Dialog Box



Build Options for C:\ti\myprojects\volume1\volume1.pjt

-g -q -o3 -frC:\ti\myprojects\volume1\Debug

Category: Basic, Advanced, Feedback, Files, Assembly, Parser, Preprocessor, Diagnostics

Basic
Target Version: Default
Generate Debug Info: Full Symbolic Debug (-g)
Opt Speed vs Size: Speed Most Critical (no ms)
Opt Level: -o3: File
Program Level Opt.: None

You can set the compiler and linker options that are used during the build process.

Your build options can be set at two different levels, depending on how frequently or in what configuration they are needed. First, you can define a set of project-level options that apply to all files in your project. Then, you can optimize your program by defining file-specific options for individual source code files.

TIP: For options that are commonly used together, you can set project-level configurations, rather than setting the same individual options repeatedly. You can also look for this information in the online help and tutorial.

Setting Project Level Options

Step 1: Select Project→Build Options.

Step 2: In the Build Options Dialog Box, select the appropriate tab.

Step 3: Select the options to be used when building your program.

Step 4: Click OK to accept your selections.

Setting File-Specific Options

Step 1: Right-click on the name of the source file in the Project View window and select File Specific Options from the context menu.

Step 2: Select the options to be used when compiling this file.

Step 3: Click OK to accept your selections.

4.4.3 Compiler Overview

The C and C++ compilers (for C5000™ and C6000™) are full-featured optimizing compilers that translate standard ANSI C programs into assembly language source. The following subsections describe the key features of the compilers.

Interfacing with Code Composer Studio

The following features pertain to interfacing with the compiler:

❑ **Compiler shell program**

  The compiler tools include a shell program that you use to compile, assembly optimize, assemble, and link programs in a single step. For more information, see the *About the Shell Program* section in the *Optimizing Compiler User's Guide* appropriate for your device.

❑ **Flexible assembly language interface**

The compiler has straightforward calling conventions, so you can write assembly and C functions that call each other. For more information, see Chapter 8, *Run-Time Environment,* in the *Optimizing Compiler User's Guide* appropriate for your device.

4.4.4 Assembly Language Development Tools

The following is a list of the assembly language development tools:

❑ **Assembler.** The assembler translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF).

❑ **Archiver.** The archiver allows you to collect a group of files into a single archive file called a *library*. Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules.

❑ **Linker.** The linker combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input.

❑ **Absolute Lister.** The absolute lister accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations.

❑ **Cross-reference Lister.** The cross-reference lister uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files.

❑ **Hex-conversion Utility.** The hex-conversion utility converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. You can download the converted file to an EPROM programmer.

❑ With the **TMS320C54x** device, the **mnemonic-to-algebraic translator utility** converts assembly language source files. The utility accepts an assembly language source file containing mnemonic instructions. It converts the mnemonic instructions to algebraic instructions, producing an assembly language source file containing algebraic instructions.

4.4.5 Assembler Overview

The assembler translates assembly language source files into machine language object files. These files are in common object file format (COFF).

The two-pass assembler does the following:

- ❑ Processes the source statements in a text file to produce a relocatable object file
- ❑ Produces a source listing (if requested) and provides you with control over this listing
- ❑ Allows you to segment your code into sections and maintains a section program counter (SPC) for each section of object code
- ❑ Defines and references global symbols and appends a cross-reference listing to the source listing (if requested)
- ❑ Assembles conditional blocks
- ❑ Supports macros, allowing you to define macros inline or in a library

4.4.6 Linker Overview

The linker allows you to configure system memory by allocating output sections efficiently into the memory map. As the linker combines object files, it performs the following tasks:

- ❑ Allocates sections into the target system's configured memory
- ❑ Relocates symbols and sections to assign them to final addresses
- ❑ Resolves undefined external references between input files

The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory module that you design. Two powerful directives, MEMORY and SECTIONS, allow you to:

- ❑ Allocate sections into specific areas of memory
- ❑ Combine object file sections
- ❑ Define or redefine global symbols at link time

4.4.6.1 Text-Based Linker

The text linker combines object files into a single executable COFF object module. Linker directives in a linker command file allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. For more information on the TI linker, see the Code Generation Tools online help.

4.4.7 C/C++ Development Tools

The following is a list of the C/C++ development tools:

- **C/C++ Compiler.** The C/C++ compiler accepts C/C++ source code and produces assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are parts of the compiler.
  - The shell program enables you to compile, assemble, and link source modules in one step. If any input file has a .sa extension, the shell program invokes the assembly optimizer.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist utility interweaves C/C++ source statements with assembly language output.
- **Assembly Optimizer (C6000™ only).** The assembly optimizer allows you to write linear assembly code without being concerned with the pipeline structure or with assigning registers. It accepts assembly code that has not been register-allocated and is unscheduled. The assembly optimizer assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly that takes advantage of software pipelining.
- **Library-build Utility.** You can use the library-build utility to build your own customized run-time-support library. Standard run-time-support library functions are provided as source code in rts.src and rstcpp.src. The object code for the run-time-support functions is compiled for little-endian mode versus big-endian mode and C code versus C++ code into standard libraries.

  The **run-time-support libraries** contain the ANSI standard run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler.

- **C++ Name Demangling Utility.** The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function’s signature in its link-level name. The process of encoding the signature into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files or linker output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

4.5 Building Your Code Composer Studio Project

4.5.1 From Code Composer Studio

To build and run a program, follow these steps:

TIP: You can use the supplied “timake.exe” utility to build a project from the DOS shell.

Step 1: Choose Project→Rebuild All or click the Rebuild All toolbar button. All the files in your project are recompiled, reassembled, and relinked. Messages about this process are shown in a frame at the bottom of the window.

Step 2: By default, the .out file is built into a debug directory located under your current project folder. To change this location, select a different one from the toolbar.

Step 3: Choose File→Load Program.
Select the program you just rebuilt, and click Open.
The program is loaded onto the target DSP and opens a Dis-Assembly window that shows the disassembled instructions that make up the program. (Notice that a tabbed area at the bottom of the window is automatically opened. It shows the output that the program sends to stdout.)

Step 4: Choose View→Mixed Source/ASM.
This allows you to simultaneously view your c source and the resulting assembly code .

Step 5: Click on an assembly instruction in the mixed-mode window. (Click on the actual instruction, not the address of the instruction or the fields passed to the instruction.)

Press the F1 key. The Code Composer Studio IDE searches for help on that instruction. This is a good way to get help on an unfamiliar assembly instruction.

Step 6: Choose Debug→Go Main to begin execution from the main function. The execution halts at Main.

Step 7: Choose Debug→Run or click the Run toolbar button to run the program.

Step 8: Choose Debug→Halt to quit running the program.

4.5.2 External Make

Code Composer Studio supports the use of external makefiles (*.mak) and an associated external make utility for project management and build process customization.

To build a program using a makefile, a project must be created that wraps the makefile. After a project is associated with the makefile, the project and its contents can be displayed in the Project View window and the Project→Build and Project→Rebuild All commands can be used to build the program.

Step 1: Double-click on the name of the makefile in the Project View window to open the file for editing.

Step 2: Modify your makefile build commands and options.

Special dialogs enable you to modify the makefile build commands and makefile options. The normal Build Options dialogs are not available when working with makefiles.

Multiple configurations can be created, each with its own build commands and options.

Limitations and Restrictions

Source files can be added to or removed from the project in the Project View. However, changes made in the Project View do not change the contents of the makefile. These source files do not affect the build process nor are they reflected in the contents of the makefile. Similarly, editing the makefile does not change the contents in the Project View. File-specific options for source files that are added in the Project View are disabled. The Project→Compile File command is also disabled. However, when the project is saved, the current state of the Project View is preserved.

4.5.3 Command Line

Using the timake utility from the command line

The timake.exe utility located in the <install dir>\cc\bin directory provides a way to build projects (*.pjt) outside of the Code Composer Studio environment from a command prompt. This utility can be used to accomplish batch builds.

To invoke the timake Utility:

Step 1: Open a DOS Command prompt

Step 2: Set up the necessary environment variables by running the batch file Dos-Run.bat. This batch file must be run before using timake. If you installed the Code Composer Studio product in c:\CCStudio, the batch file is located at:

C:\CCStudio\DosRun.bat

Step 3: Run the timake utility.

Please refer to the online help topic Using the timake Utility for more information.

Makefiles

In addition to the option of using external makefiles within the Code Composer Studio IDE, you can also export standard Code Composer Studio project file (*.pjt) to a standard makefile that can be built from the command line using any standard make utility. Code Composer Studio comes with a standard make utility (gmake) that can be run after running the DosRun.bat file.

To Export a Code Composer Studio Project to a Standard Makefile

Step 1: Make the desired project active by selecting the project name from theSelect Active Project dropdown list on the Project toolbar.

Step 2: Select Project Export to Makefile.

Step 3: In the Exporting <filename>.pjt dialog box, specify the configurations to export, the default configuration, the host operating system for your make utility, and the file name for the standard makefile.

Step 4: Click OK to accept yor selections and generate a standard makefule.

Pleae refer to the online help topic, Exporting a Project to a Makefile.

4.6 Available Foundation Software

4.6.1 DSP/BIOS

DSP/BIOS™ is a scalable real-time kernel, designed specifically for the TMS320C5000™ , TMS320C2000™, and TMS320C6000™ DSP platforms. DSP/BIOS enables you to develop and deploy sophisticated applications more quickly than with traditional DSP software methodologies and eliminates the need to develop and maintain custom operating systems or control loops. Because multithreading enables real-time applications to be cleanly partitioned, an application using DSP/BIOS is easier to maintain and new functions can be added without disrupting real-time respnse. DSP/BIOS provides standardized APIs across C2000, C5000, and C6000 DSP platforms to support rapid application migration.

4.6.2 CSL

The Chip Support Library(CSL) provides C-program functions to configure and control on-chip peripherals. It is intended to simplify the process of running algorithms in a real system. The goal is peripheral ease of use, shortened development time, portability, hardware abstraction, and a small level of standardization and compatibility among devices.

4.6.2.1 Benefits of CSL

The CSL benefits you in the following ways:

❑ **Standard Protocol to Program Peripherals**

CSL provides a higher-level programming interface for each on-chip peripheral. This includes data types and macros to define peripheral register configuration, and functions to implement the various operations of each peripheral.

❑ **Basic Resource Management**

Basic resource management is provided through the use of open and close functions for many of the peripherals. This is especially helpful for peripherals that support multiple channels.

❑ **Symbol Peripheral Descriptions**

As a side benefit to the creation of CSL, a complete symbolic description of all peripheral registers and register fields has been created. It is suggested that you use the higher-level protocols described in the first two bullets, as these are less device specific, making it easier to migrate your code to newer versions of DSPs.

4.6.3 BSL

The TMS320C6000 DSK Board Support Library (BSL) is a set of C-language application programming interfaces (APIs) used to configure and control all on-board devices, which is intended to make it easier for developers to get algorithms up and running in a real system. The BSL consists of discrete modules that are built and archived into a library file. Each module represents an individual API and is referred to simply as an API module. The module granularity is constructed such that each device is covered by a single API module except the I/O Port module, which is divided into two APU modules: LED and DIP.

4.6.3.1 Benefits of BSL

Some of the advantages offered by the BSL include: device ease of use, a level of compatibility between devices, shortened development time, portability, some standardization, and hardare abstraction. In general, BSL makes it easier for you to get yoru algorithms up and running in the shortest length of time.

4.6.4 DSPLIB

The DSP Library (DSPLIB) includes many C-callable, assembly-optimized, general-purpose signal-processing and image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP and image/video processing functions, DSPLIB and IMGLIB can significantly shorten your application development time.

For more information on DSPLIB see:

- ❏ *TMS320C54x DSP Library Programmer's Reference* (SPRU518)
- ❏ *TMS320C55x DSP Library Programmer's Reference* (SPRU422)
- ❏ *TMS320C62x DSP Library Programmer's Reference* (SPRU402)
- ❏ *TMS320C64x DSP Library Programmer's Reference* (SPRU565)

4.6.4.1 Benefits of DSPLIB

DSPLIB includes commonly used routines. Source code is provided that allows you to modify functions to match your specific needs.

Features include:

- ❑ Optimized assembly code routines
- ❑ C and linear assembly source code
- ❑ C-callable routines fully compatible with the TI Optimizing C compiler
- ❑ Benchmarks (cycles and code size)
- ❑ Tested against reference C model

4.6.4.2 DSPLIB Functions Overview

DSPLIB provides a collection of C-callable high performance routines that can serve as key enablers for a wide range of signal and image/video processing applications. These functions are representative of the high performance capabilities of the C5000 and C6000 DSPs.

The routines contained in the DSPLIB are organized into the following functional categories**:**

- ❑ Adaptive filtering
- ❑ Correlation
- ❑ FFT
- ❑ Filtering and convolution
- ❑ Math
- ❑ Matrix functions
- ❑ Miscellaneous

4.6.5 IMGLIB

The Image/Video Processing Library (IMGLIB) includes many C-callable, assembly-optimized, general-purpose signal-processing and image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP and image/video processing functions, DSPLIB and IMGLIB can significantly shorten your application development time.

For more information on IMGLIB see:

- *TMS32C55x Imaging/Video Processing Library Programmer's Reference Guide* (SPRU037)
- *TMS320C62x Image/Video Processing Library Programmer's Reference* (SPRU400)
- *TMS320C64x Image/Video Processing Library Programmer's Reference* (SPRU023).

4.6.5.1 Benefits of IMGLIB

IMGLIB includes commonly used routines. Source code is provided that allows you to modify functions to match your specific needs.

Features include:

- Optimized assembly code routines
- C and linear assembly source code
- C-callable routines fully compatible with the TI Optimizing C compiler
- Benchmarks (cycles and code size)
- Tested against reference C model

4.6.5.2 IMGLIB Functions Overview

IMGLIB provides a collection of C-callable high performance routines that can serve as key enablers for a wide range of signal and image/video processing applications. These functions are representative of the high performance capabilities of the C6000 DSPs.

The set of software routines included in the IMGLIB (available for the C6000 platform only) are organized into three different functional categories as follows:

- Image/video compression and decompression
- Image Analysis
- Picture filtering/format conversions

4.6.6 XDAIS Components

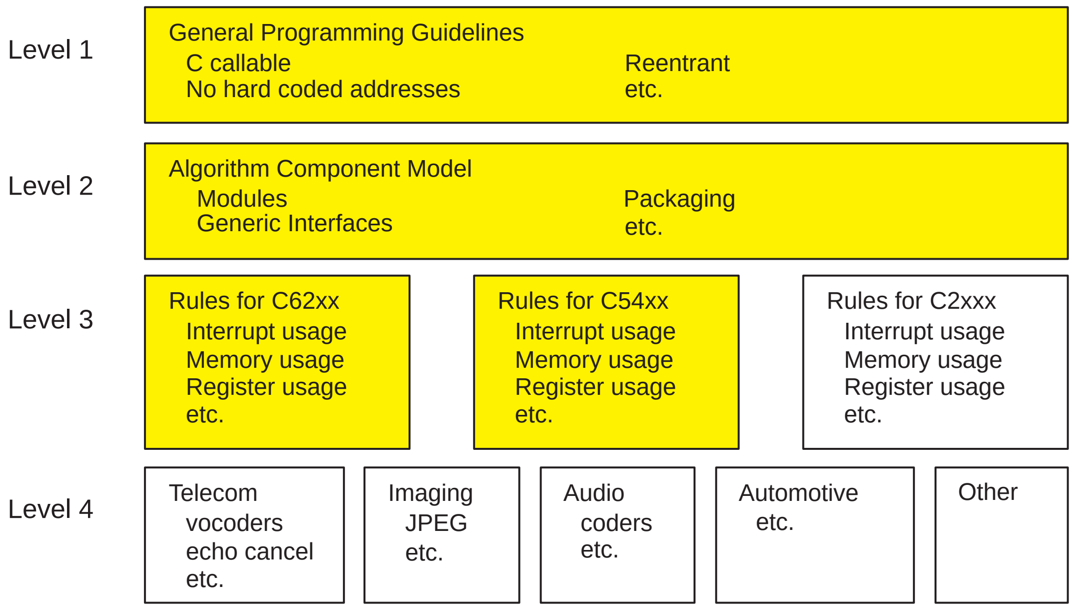
DSPs are programmed in a mix of C and assembly language, directly access hardware peripherals, and for performance reasons, almost always have little or no standard operating system support. Unlike general-purpose embedded microprocessors, DSPs are designed to run sophisticated signal processing algorithms and heuristics, but because of the lack of consistent standards, it is not possible to use an algorithm in more than one system without significant reengineering. Since reuse of DSP algorithms is so labor intensive, the time-to-market for a new DSP-based product is measured in years rather than months.

The TMS320 DSP Algorithm Standard (known as XDAIS) defines a set of requirements for DSP algorithms that, if followed, allow system integrators to quickly assemble production-quality systems from one or more such algorithms.

Scope of XDAIS

The TMS320 DSP Algorithm Standard defines three levels of guidelines.

Figure 4−9. TMS320 DSP Algorithm Standard Elements



Level 1 contains programming guidelines that apply to all algorithms on all DSP architectures regardless of application area. Almost all recently developed software modules follow these common sense guidelines already, so this level just formalizes them.

Level 2 contains rules and guidelines that enable all algorithms to operate harmoniously within a single system. Conventions are established for an algorithm's use of data memory and names for external identifiers, as well as simple rules for how algorithms are packaged.

Level 3 contains the guidelines for specific families of DSPs. Today, there are no agreed-upon guidelines for algorithms with regard to the use of processor resources. These guidelines will provide guidance on the dos and don'ts for the various architectures. There is always the possibility that deviations from these guidelines will occur, but then the algorithm vendor can explicitly draw attention to the deviation in the relevant documentation or module headers.

The shaded boxes in Figure 4−9 represent the areas that are covered in this version of the specification.

Level 4 contains the various vertical markets. Due to the inherently different nature of each of these businesses, it seems appropriate for the stakeholders in each of these markets to define the interfaces for groups of algorithms based on the vertical market. If each unique algorithm were specified with an interface, the standard would never be able to keep up and thus not be effective. It is important to note that at this level, any algorithm that conforms to the rules defined in the top three levels is considered eXpressDSP-compliant.

Rules and Guidelines

The TMS320 DSP Algorithm Standard specifies both rules and guidelines. Rules **must** be followed in order for software to be eXpressDSP-compliant. Guidelines, on the other hand, are strongly suggested recommendations that should be obeyed, but are not required, in order for software to be eXpressDSP-compliant.

Requirements of the Standard

The required elements of XDAIS:

❑ Algorithms from multiple vendors can be integrated into a single system.

❑ Algorithms are framework-agnostic. That is, the same algorithm can be efficiently used in virtually any application or framework.

❑ Algorithms can be deployed in purely static as well as dynamic run-time environments.

- ❑ Algorithms can be distributed in binary form.
- ❑ Integration of algorithms does not require recompilation of the client application; reconfiguration and relinking may be required however.

Goals of the Standard

The goals of XDAIS are:

- ❑ Easy to adhere to the standard
- ❑ Possible to verify conformance to standard
- ❑ Enable system integrators to easily migrate between TI DSPs
- ❑ Enable host tools to simplify a system integrator's tasks, including configuration, performance modeling, standard conformance, and debugging.
- ❑ Incur little or no "overhead" for static systems

4.6.7 Reference Frameworks

Reference Frameworks for eXpressDSP™ software are provided as starterware for applications that use DSP/BIOS™ and the TMS320™ DSP Algorithm Standard. Developers first select the Reference Framework that best approximates their system and its future needs, and then adapt the framework and populate it with eXpressDSP-compliant algorithms. As common elements such as device drivers, memory management, and channel encapsulation are already pre-configured in the frameworks, developers can focus on their sys

Reference Frameworks contain design-ready, reusable, C language source code for TMS320C5000™ and TMS320C6000™ digital signal processors (DSPs). Developers can build on top of the framework, confident that the underlying pieces are robust and appropriate for the characteristics of the target application.

Reference Frameworks software and documentation are available for download from the TI website. It is not included in the Code Composer Studio installation.

Figure 4−10 shows the elements that make up a Reference Framework on the target DSP.

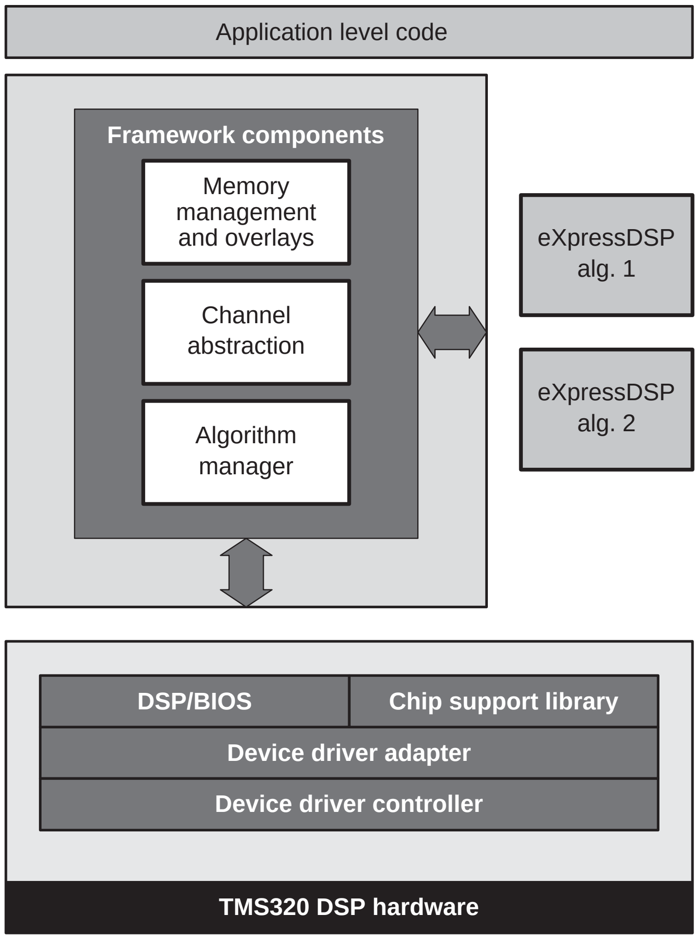


Figure 4−10. Elements of a Reference Framework

Here are the elements, starting from the bottom up:

- **Device controller and device adapter.** The device drivers used in reference frameworks are based on a standard driver model, which provides device adapters and specifies a standard device controller interface. If you have unique external hardware, the device controller is likely to need customization, but the device adapter probably needs little or no modification.
- **Chip Support Library (CSL).** The device controller uses chip support library modules to support peripheral hardware.
- **DSP/BIOS.** This extensible software kernel is a good example of how each reference framework leverages different amounts of the eXpressDSP

infrastructure, depending on its needs. The low-end RF1 framework uses relatively few DSP/BIOS modules. In addition to providing an obvious footprint savings, reducing the number of modules helps clarify design choices for a designer who may not fully appreciate the ramifications of module selections.

- **Framework Components.** These elements provide overall system resource management, such as channel abstraction. Every reference framework needs some kind of channel management; but, design optimizations can be made based on knowledge of the number of channels likely to be in use. For simple systems with 1 to 3 channels, channel scheduling is handled with the low-overhead DSP/BIOS HWI and IDL modules. For larger numbers of channels, it is wiser to use the SWI module, although it comes with some extra footprint. For large systems with channels that change dynamically, the TSK module is most appropriate. In each reference framework, design decisions such as these let designers get systems running both more quickly and in the way most suited to the system.

  Similar to channel managers are the algorithm managers, which manage some number of eXpressDSP-compliant algorithms. Other framework components are modules that handle memory overlay schemes—a critical technique in most memory constrained systems. Simply starting with the appropriate framework simplifies many choices that need to be made when developing from scratch.

- **eXpressDSP-compliant Algorithms.** Each algorithm follows the rules and guidelines of the *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352). To be compliant with the standard, algorithms must not directly access any hardware peripherals and must implement standard resource management interfaces known as IALG (for memory management) and optionally, IDMA (for DMA resource management). In the examples that TI provides, the algorithms are generally simple ones like finite impulse response (FIR) filters and volume controllers. It is relatively easy to substitute more significant eXpressDSP-compliant algorithms for the TI-provided ones. It is this substitution that starts the process of making a generic reference framework application-specific.

- **Application-level code.** The last step is to modify the application-level code. This code applies unique and value-added application-specific knowledge, allowing for real product differentiation. Clearly the application code required for a single-channel MP3 player is quite different than that required for a digital hearing aid.

4.7 Automation (for Project Management)

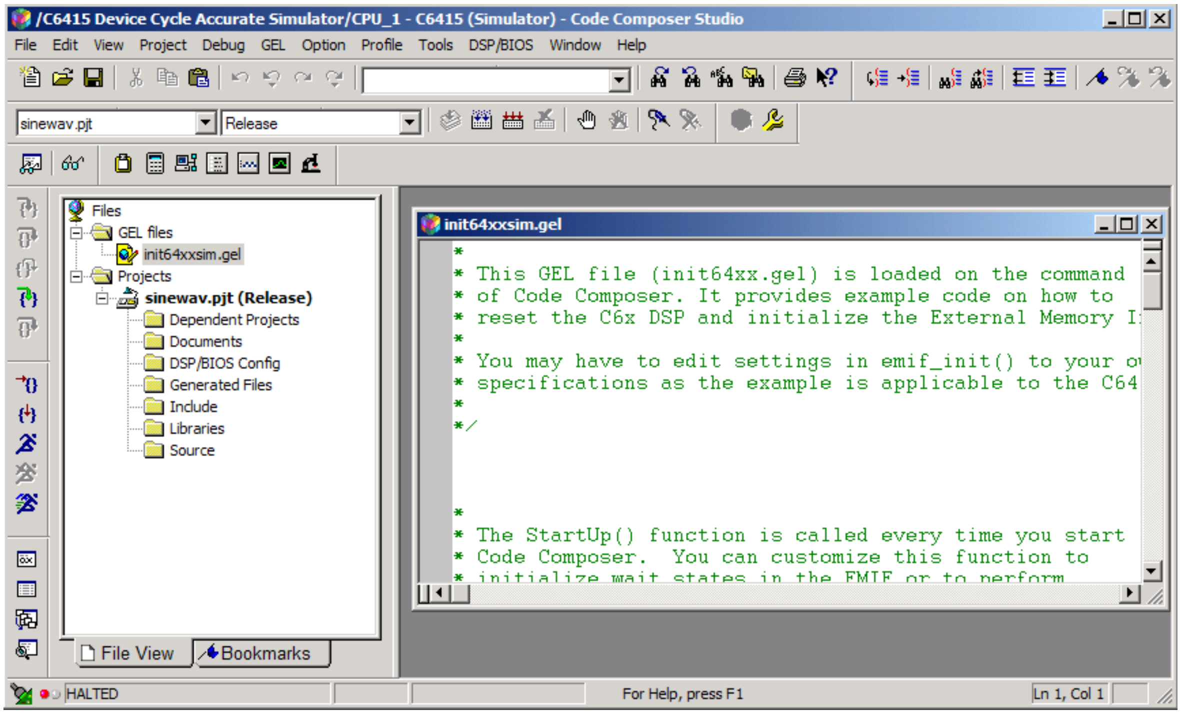
4.7.1 Using General Extension Language (GEL)

The General Extension Language (GEL) is an interpretive language, similar to C, that lets you create functions to extend Code Composer Studio IDE's usefulness. You create your GEL functions by using the GEL grammar, and then by loading them into Code Comoser Studio IDE. There is subset of GEL functions that may be used to automate building of projects with CCStudio environment. Custom GEL menus may be created to automatically open and build a project.

Figure 4−11. GEL script to open volume project.

```
/*
 *  Copyright 1998 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments
Incorporated.
 *  Restricted rights to use, duplicate or disclose this
code are
 *  granted through contract.
 */
/*
 *  ======== PrjOpen.gel ========
 *  Simple gel file to demonstrate project managment
capabilities of GEL
 */

menuitem ”MyProjects”

hotmenu OpenVolume()
{
        // Open Volume tutorial example

GEL_ProjectLoad(”C:\\ti\\tutorial\\sim55xx\\volume1\\volum
e.pjt”);

        // Set currently active configuration to debug

GEL_ProjectSetActiveConfig(”c:\\ti\\tutorial\\sim55xx\\vol
ume1\\volume.pjt”,
”Debug”);

        // Build the project.
        GEL_ProjectBuild();
}
```

Figure 4−12. Custom GEL menu to open a project.



```
*
* This GEL file (init64xx.gel) is loaded on the command
* of Code Composer. It provides example code on how to
* reset the C6x DSP and initialize the External Memory I
*
* You may have to edit settings in emif_init() to your o
* specifications as the example is applicable to the C64
*
*/



*
* The StartUp() function is called every time you start
* Code Composer.  You can customize this function to
* initialize wait states in the EMIF or to perform
```

4.7.2 Scripting Utility

The Scripting Utility is a set of Code Composer Studio commands that are integrated into a VB or perl scripting languages. You may utilize full capabilities of a scripting language such as perl or VB and combine it with automation tasks that need to be performed in Code Composer Studio. The Scripting utility may be used to configure a test scenario, open and build a corresponding project and load it for execution. There are a number of scripting commands that may be used to build and manage projects.

The Scripting Utility is an add-on capability available through Update Advisor.

Chapter 5

Debug

This chapter applies to all platforms using Code Composer Studio™ IDE. *However, not all devices have access to all of the tools discussed in this chapter. For a complete listing of the tools available to you, see the online help and online documentation provided with the Code Composer Studio IDE.*

The Code Composer Studio IDE comes with a number of tools that help you debug your programs. This chapter discusses these tools and shows you how to use them.

Topic **Page**

5.1 Setting Up Your Environment for Debug

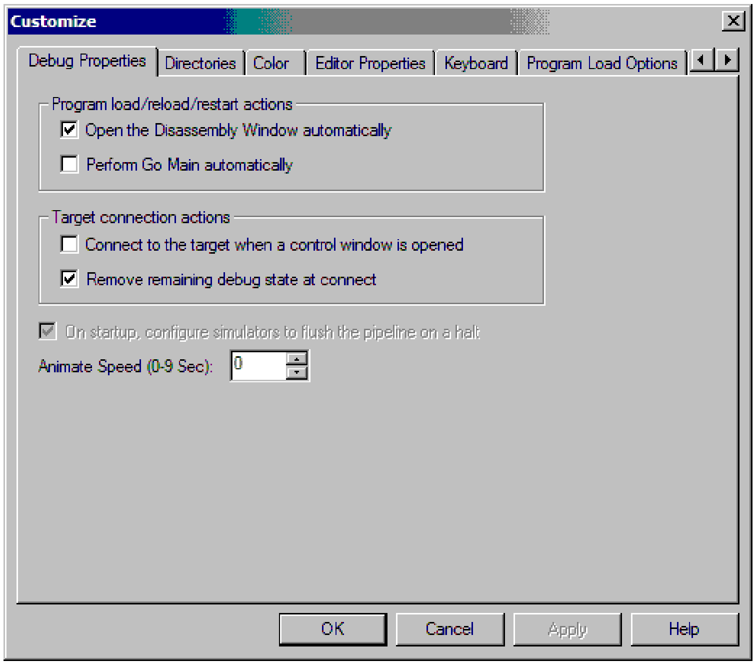
Before you can successfully debug an application, the environment must be configured. The following sections tell how this can be accomplished.

5.1.1 Setting Custom Debug Options

Several debugging options are customizable within Code Composer Studio. You can configure these options to help with the debug process or to suit your desired preferences.

5.1.1.1 Debug Properties Tab

This debug properties dialog is on the Customize Tab under Options→Customize→Debug. It allows you to turn off certain default behaviors when debugging.


Customize

Debug Properties | Directories | Color | Editor Properties | Keyboard | Program Load Options

Program load/reload/restart actions
- [✓] Open the Disassembly Window automatically
- [ ] Perform Go Main automatically

Target connection actions
- [ ] Connect to the target when a control window is opened
- [✓] Remove remaining debug state at connect

- [✓] On startup, configure simulators to flush the pipeline on a halt

Animate Speed (0-9 Sec): 0

Program load/reload/restart actions

Open the Disassembly Window automatically: Unchecking this will prevent the Disassembly Window from appearing after a program is loaded. This option is enabled by default.

Perform Go Main Automatically. Enabling this option will instruct the debugger to automatically run to the symbol ‘main’ for the application loaded. This option is disabled by default.

Target Connection actions

Connect to the Target When a Control Window is Opened: Control Window simply refers to the entire IDE interface for Code Composer Studio. You can have multiple instances of this open when running PDM+. You can turn this off when you are experiencing target connection problems or don't need the actual target to be connected (i.e., when writing source code, etc.). This option is disabled by default.

Remove Remaining Debug State at Connect. When Code Composer Studio disconnects from the target, it typically tries to remove breakpoints by default. If for some reason there are errors when trying to remove breakpoints, Code Composer Studio will try again to remove breakpoints when reconnecting to the target. However, this second attempt to remove breakpoints can potentially put some targets into a bad state. For this reason, users are urged to disable this second attempt to remove breakpoints when reconnecting. To disable this second attempt, uncheck this option.

Animation Speed. Animation Speed is the minimum time (in seconds) between breakpoints. Program execution does not resume until the minimum time has expired since the previous breakpoint. See section 5.2.1 for more details.

5.1.1.2 Directories

To open up the source file where the application is halted, the debugger needs its location. The debugger includes source path information for:

❑ All the files in a project that is open in the project view

❑ Files in the current directory

❑ Paths specified by the user to Code Composer Studio

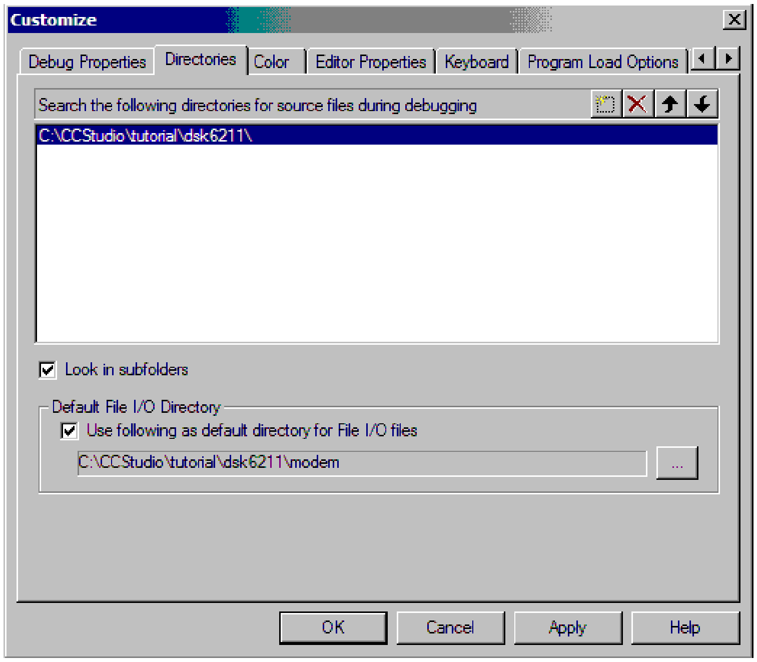
The paths specified to Code Composer Studio by the user are empty by default. Thus, if the application being debugged has used source files that are not in the two former options, then the path to these files need to be specified. If not, the debugger will not be able to automatically open the file when execution halts at a location that references that source file (but will prompt the user to manually find it). A common example of source files not being part of the open project is when including libraries in a build.

The Directories dialog box enables you to specify additional search paths that the debugger uses to find the source files included in a project.

To Specify Search Path Directories

Step 1: Select Option→Customize.

Step 2: In the Customize dialog box, select the Directories tab. Use the scroll arrows at the top of the dialog box to locate the tab.



Search the following directories for source files during debugging

C:\CCStudio\tutorial\dsk6211\

☑ Look in subfolders

Default File I/O Directory

☑ Use following as default directory for File I/O files

C:\CCStudio\tutorial\dsk6211\modem

The Directories dialog offers the following options.

- **Directories.** The Directories list displays the defined search path. The debugger searches the listed directories in order from top to bottom.

  If two files have the same name and are located in different directories, the file located in the directory that appears highest in the Directories list takes precedence.

- **New.** To add a new directory to the Directories list, click New. Enter the full path or browse […] to the appropriate directory. By default, the new directory is added to the bottom of the list.

- **Delete.** Select a directory in the Directories list, then click Delete to remove that directory from the list.

- **Up.** Select a directory in the Directories list, then click Up to move that directory higher in the list.

- **Down.** Select a directory in the Directories list, then click Down to move that directory lower in the list.
- **Look in subfolders.** You can instruct the debugger to search in the subfolders of the listed paths by enabling the Look in subfolders option.
- **Default File I/O directory.** In addition to setting source file directories, you can now set a default directory for File I/O files. Simply enable the Default File I/O directory option and then use the browse button to find the path you wish to select as the default directory.

Step 3: Click OK to exit the Customize dialog box.

5.1.1.3 Program Load Options

The Program Load dialog box enables you to select actions that will occur automatically when you load a program or load symbols.

To Set Program Load Options

Step 1: Select Option→Customize.

Step 2: In the Customize dialog box, select the Program Load Options tab.

The Program Load Options dialog offers the following options.

- **Perform verification after Program Load.** By default, this checkbox is selected. This means that Code Composer Studio will verify (by reading back selected memory) that the program was loaded correctly. If you remove the check from this option, this verification will not be performed.
- **Load Program After Build.** When this option is selected, the executable is loaded immediately upon building the project. This ensures that the target contains the most up-to-date symbolic information generated after a build.
- **Open Dependent Projects When Loading Projects.** By default, if your program has subprojects upon which a main project is dependent, all the subprojects are opened along with the main project. If this option is disabled, then the subprojects will not be opened.
- **Do Not Scan Dependencies When Loading Projects.** To determine which files must be compiled during an incremental build, the project must maintain a list of include file dependencies

for each source file. A dependency tree is created whenever you build a project. To create the dependency tree, all the source files in the project list are recursively scanned for #include, .include, and .copy directives, and each included file name is added to the project list. By default, when a project is opened, all files in the project are scanned for dependencies. If this option is disabled, it will not automatically scan for dependencies upon opening a project. This may result in quicker times for opening a project.

- **Do Not Set CIO Breakpoint At Load.** By default, if your program has been linked using a TI runtime library (rts*.lib), a C I/O breakpoint (C$$IO$$) is set when the program is loaded. This option enables you to choose not to set the C I/O breakpoint.

  The C I/O breakpoint is necessary for the normal operation of C I/O library functions such as printf and scanf. The C I/O breakpoint is not needed if your program does not execute CIO functions.

  When C I/O code is loaded in RAM, Code Composer Studio sets a software breakpoint. However, when C I/O code is loaded in ROM, Code Composer Studio uses a hardware breakpoint. Since most processors support only a small number of hardware breakpoints, using even one can have a significant impact when debugging.

  Note: You can also avoid using the hardware breakpoint when C I/O code is loaded in ROM by embedding a breakpoint in your code and renaming the label C$$IO$$ to C$$IOE$$ to indicate that this is an embedded breakpoint.

- **Do Not Set End of Program Breakpoint At Load.** By default, if your program has been linked using a TI runtime library (rts*.lib), an End of Program breakpoint (C$$EXIT) is set when the program is loaded. This option allows you to choose not to set the End of Program breakpoint.

  The End of Program breakpoint is used to halt the processor when your program exits following completion. The End of Program breakpoint is not needed if your program executes an infinite loop.

  When End of Program code is loaded in RAM, Code Composer Studio sets a software breakpoint. However, when End of Program code is loaded in ROM, Code Composer Studio uses a hardware breakpoint. Since most processors support only a small number of hardware breakpoints, using even one can have a significant impact when debugging.

Note: You can also avoid using the hardware breakpoint when End of Program code is loaded in ROM by embedding a breakpoint in your code and renaming the label C$$EXIT to C$$EXITE$$ to indicate that this is an embedded breakpoint.

- **Disable All Breakpoints When Loading New Programs.** Enabling this option will remove all existing breakpoints before loading a new program.

Step 3: Click OK.

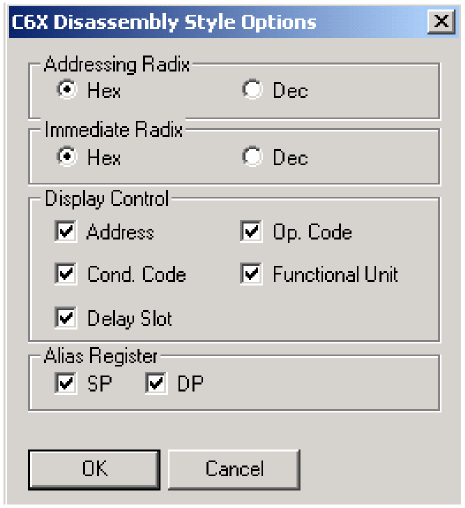
5.1.1.4 Disassembly Style

Several options are available for changing the way you view information in the Disassembly window. The Disassembly Style Options dialog box allows you to input specific viewing options for your debugging session.

To Set Disassembly Style Options

Step 1: Select Option→Disassembly Style, or right−click in the Disassembly window and select Properties→Disassembly Options.

Step 2: Enter your choices in the Disassembly Style Options dialog box.


C6X Disassembly Style Options

Addressing Radix
[✓ Hex] Dec

Immediate Radix
[✓ Hex] Dec

Display Control
☑ Address ☑ Op. Code
☑ Cond. Code ☑ Functional Unit
☑ Delay Slot

Alias Register
☑ SP ☑ DP

Step 3: Click OK.
The contents of the Disassembly window are immediately updated with the new style.

5.1.1.5 Default File I/O Directory

In addition to setting source file directories, you can now set a default directory for File I/O files. Simply enable the Default File I/O Directory option and then use the browse button to find the path you wish to select as the default directory.

5.1.2 Simulation

Debugging on simulation may require additional steps in order to configure the simulator to behave closer to the actual HW target.

5.1.2.1 Memory Mapping

The memory map tells the debugger which areas of memory it can and cannot access. Memory maps vary depending on the application.

When a memory map is defined and memory mapping is enabled, the debugger checks every memory access against the memory map. The debugger will not attempt to access an area of memory that is protected by the memory map

When the debugger compares memory accesses against the memory map, it performs this checking in software, not hardware. The debugger cannot prevent your program from attempting to access nonexistent memory.

Memory Mapping with Simulation

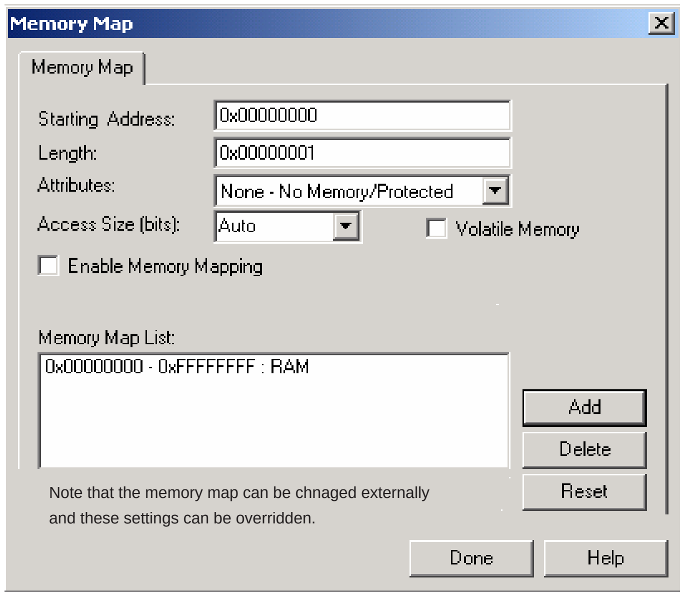
The simulator utilizes pre-defined memory map ranges to allow the most generic representation of valid memory settings for DSP targets being simulated. The memory map settings can be altered to some degree; however, this is not a recommended practice as simulator performance may be affected if extensive changes to valid memory ranges are employed.

Using the Debugger

The memory map can be defined interactively while using the debugger. This can be inconvenient because, in most cases, you will set up up one memory map before you begin debugging, and will then use this memory map for all other debugging sessions.

To Add a New Memory Map Range

Step 1: Select Option→Memory Map.



Memory Map

Starting Address: 0x00000000

Length: 0x00000001

Attributes: None - No Memory/Protected

Access Size (bits): Auto

☐ Volatile Memory

☐ Enable Memory Mapping

Memory Map List:

0x00000000 - 0xFFFFFFFF : RAM

Note that the memory map can be chnaged externally and these settings can be overridden.

Step 2: If your actual or simulated target memory configuration supports multiple pages, the Memory Map dialog box contains a separate tab for each type of memory page (e.g., Program, Data, and I/O). Select the appropriate tab for the type of memory that you want to modify. Tabs do not appear for processors that have only one memory page.

The Memory Map dialog offers the following options.

- **Enable Memory Mapping.** Make sure that the Enable Memory Mapping checkbox is checked. Otherwise, all addressable memory (RAM) on your target is assumed to be valid by the debugger.
- **Starting Address.** Enter the start address of the new memory range in the Starting Address input field.
- **Length.** Enter the length of the new memory range in the Length input field.
- **Attributes**. Select the read/write characteristics of the new memory range in the Attributes field.

- **Access Size (bits).** Specify the access size for your target processor. You can select an access size from the drop−down list, or you can type a value in the Access Size field.

  It is not necessary to specify a size for processors that support only one access size.
- **Volatile Memory.** Normally, a write access consists of Read, Modify, and Write operations. When the Volatile Memory option is set on a segment of memory, any write access to that memory is completed by using only a Write operation.
- **Memory Map List.** Displays the list of memory−mapped ranges.
- **Add.** Click the Add button to add the new memory range to the Memory Map List.
- **Delete.** In the Memory Map List, select the memory map range that you want to delete and click the Delete button.

  You can also delete an existing memory map range by changing the Attributes field to “None − No Memory/Protected”. This means you can neither read nor write to this memory location.
- **Reset.** Resets the default values in the Memory Map List.

Step 3: Click Done to accept your selections.

The debugger allows you to enter a new memory range that overlaps existing ones. The new range is assumed to be valid, and the overlapped range’s attributes are changed accordingly.

After you have defined a memory map, you may wish to modify its read/write attributes. You can do this by defining a new memory map (with the same Starting Address and Length) and clicking the Add button. The debugger overwrites the existing attributes with the new ones.

Using GEL

The memory map can also be defined using the general extension language (GEL) built-in functions. GEL provides a complete set of memory-mapping functions. The easiest method of implementing a memory map is to put the memory−mapping functions in a GEL text file and execute the GEL file at start up.

When you first invoke Code Composer Studio, the memory map is turned off. You can access any memory location; the memory map does not interfere. If you invoke Code Composer Studio with an optional GEL filename specified as a parameter, the Gel file is automatically loaded. If the file contains the GEL function StartUp(), the GEL functions in the file are executed. You can specify GEL mapping functions in this file to automatically define the memory mapping requirements for your environment.

Use the following GEL functions to define your memory map:

GEL_MapAdd()	Memory map add
GEL_MapDelete()	Memory map delete
GEL_MapOn()	Enable memory map
GEL_MapOff()	Disable memory map
GEL_MapReset()	Reset memory map

The GEL_MapAdd() function defines a valid memory range and identifies the read/write characteristics of the memory range. The following is a sample of a GEL file that can be used to define two blocks of length 0xF000 that are both readable and writeable:

```
StartUp()
{
  GEL_MapOn();
  GEL_MapReset();
  GEL_MapAdd(0, 0, 0xF000, 1, 1);
  GEL_MapAdd(0, 1, 0xF000, 1, 1);
}
```

When you have set up your memory map, you can use the Option→Memory Map command to view it.

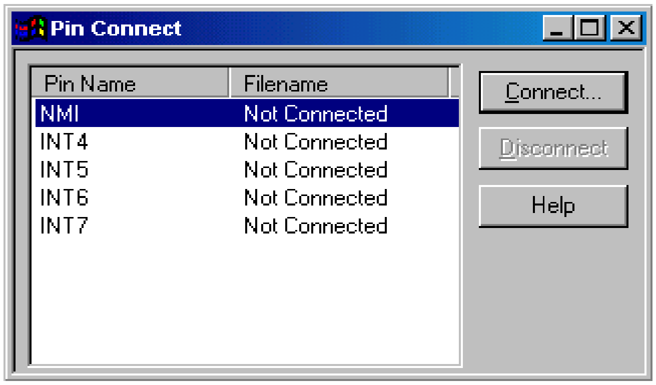
5.1.2.2 Pin Connect

The Pin Connect tool enables you to specify the interval at which selected external interrupts occur.

To simulate external interrupts:

Step 1: Create a data file that specifies interrupt intervals.

Step 2: Start the Pin Connect tool. From the Tools menu, choose Pin Connect.



Pin Name	Filename
NMI	Not Connected
INT4	Not Connected
INT5	Not Connected
INT6	Not Connected
INT7	Not Connected

Step 3: Select the Pin name and click Connect.

Step 4: Load your program.

Step 5: Run your program.

For detailed information on the Pin Connect tool, see the Pin Connect topics provided in the online help: Help→Contents→Pin Connect.

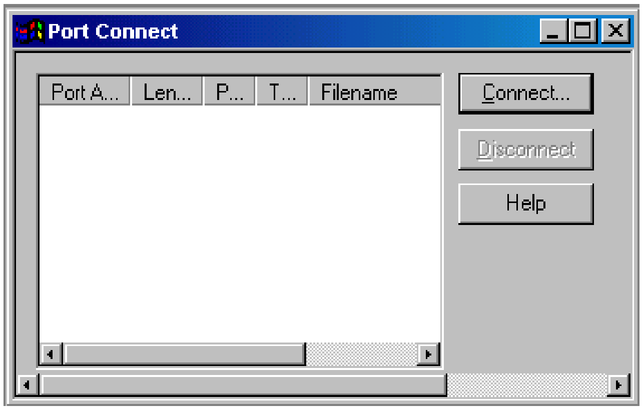
5.1.2.3 Port Connect

You can use the Port Connect tool to access a file through a memory address. Then, by connecting to the memory (port) address, you can read data in from a file, and/or write data out to a file.

To connect a memory (port) address to a data file, follow these steps:

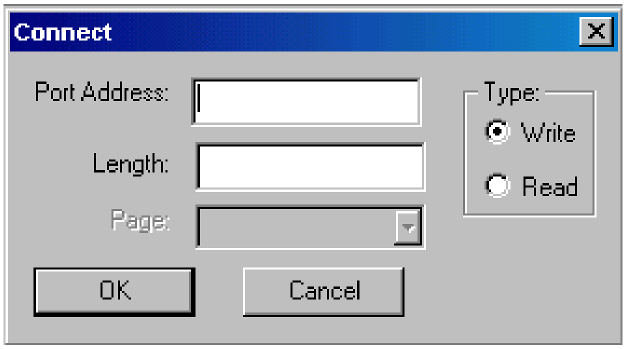
Step 1: From the Tools menu, select Port Connect.

This action displays the Port Connect window and starts the Port Connect tool.



Step 2: Click the Connect button.

This action opens the Connect dialog box.



Step 3: In the Port Address field, enter the memory address.

This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with 0x; otherwise, it is treated as a decimal address.

Step 4: In the Length field, enter the length of the memory range.

The length can be any C expression.

Step 5: In the Page field (C5000 only), choose type of memory (program or data) that the address occupies:

To identify this page. . .	use this value.
Program memory	Prog
Data memory	Data
I/O space	I/O

Step 6: In the Type field, select the Write or Read radio button, depending on whether you want to read data from a file or write data to a file.

Step 7: Click OK.

This action displays the Open Port File window.

Step 8: Select the data file to which you want to connect and click Open.

Step 9: Select the No Rewind feature to prohibit the file from being rewinded when the end-of-file (EOF) is reached. For read accesses made after EOF, the value 0xFFFFFFFF is read and the file pointer is kept unchanged.

The file is accessed during an assembly language read or write of the associated memory address. Any memory address can be connected to a file. A maximum of one input and one output file can be connected to a single memory address; multiple addresses can be connected to a single file.

For detailed information on the Port Connect tool, see the Port Connect topics provided in the online help: Help→Contents→Port Connect.

5.1.3 Program Load

The COFF file (*.out) produced by building your program must be loaded onto the actual or simulated target board prior to execution.

Program code and data are downloaded onto the target at the addresses specified in the COFF file.

Symbols are loaded into a symbol table maintained by the debugger on the host. The symbols are loaded at the code and data addresses specified in the COFF file.

A COFF file can be loaded by selecting File→Load Program...from the main menu and then using the Load Program dialog box to select the desired COFF file.

5.1.3.1 Loading Symbols Only

It is useful to load only symbol information when working in a debugging environment where the debugger cannot or need not load the object code, such as when the code is in ROM.

Symbols can be loaded by selecting File→Load Symbols→Load Symbols Only… from the main menu and then using the Load Symbols dialog box to select the desired COFF file.

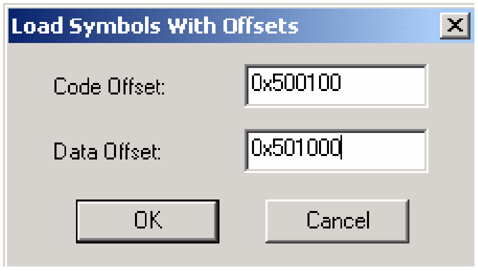
The debugger deletes any previously loaded symbols from the symbol table maintained on the host. The symbols in the symbol file are then loaded into the symbol table. Symbols are loaded at the code and data addresses specified in the symbol file.

This command does not modify memory or set the program entry point.

You can also specify a code offset and a data offset that the debugger will apply to every symbol in the specified symbol file.

For example, suppose you have a symbol file for an executable that contains code addresses starting at 0x100 and data addresses starting at 0x1000. However, in the program loaded on the target the corresponding code starts at 0x500100 and the data is located at 0x501000.

To specify the code and data offset, select File→Load Symbols→Load Symbols with Offsets… from the main menu and then use the Load Symbols dialog box to select the desired COFF file. Once a COFF file is selected, an additional Load Symbols with Offsets dialog box will appear for the user to enter the actual addresses where code and data start.



Code Offset: 0x500100

Data Offset: 0x501000

The debugger automatically offsets every symbol in that symbol file by the given value.

Adding Symbols Only

Symbol information can also be appended to the existing symbol table. This command differs from the Load Symbol command in that it does not clear the existing symbol table before loading the new symbols.

The steps for adding symbol information, with (File→Load Symbols→Load Symbols with Offsets…) or without offsets (File→Load Symbols→Load Symbols Only…), is similar to the steps outlined above for loading symbols.

5.2 Basic Debugging

Several components are available and many times necessary for basic debugging in the Code Composer Studio IDE. The chart below provides a list of the icons used for debugging; they will be used throughout this chapter.

Step into (source mode)

Step over (source mode)

Step out (source and assembly mode)

Step into (assembly mode)

Step out (assembly mode)

Sync run

Sync halt

Animate

Toggle breakpoint

Toggle probe point

Expression

5.2.1 Running/Stepping

Both source and assembly stepping are available only when the execution has been halted. By accessing a mixed Source/ASM mode through View→Mixed Source/ASM, you can view both source and assembly code simultaneously.

There are three types of stepping.

1) Step-Into executes one single statement and then execution is halted.
2) Step-Over executes the function and then halts after the function returns.
3) Step-Out executes the current subroutine and then returns to the calling function. Execution is then halted after returning to the calling function.

5.2.1.1 Source Stepping

Source stepping steps through the lines of code displayed on your source editor.

5.2.1.2 Assembly Stepping

Assembly stepping steps through the lines of instructions that can be displayed on your dis-assembly window.

Run

The following commands allow you to run the program.

- **Main** − You can begin your debugging at main by Debug→Go Main. This action will take your execution to your main function.
- **Run** − After execution has been halted, you can continue to run by pressing the run button.
- **Run to Cursor** − If you want the program to run to a specific location, you can simply place the cursor at the said location and then pressing this button .
- **Set PC to Cursor** − You can also set the program counter to a certain location by placing the cursor at the location and then pressing the button .
- **Animate** − This action runs the program until a breakpoint is encountered. At the breakpoint, execution stops and all windows not connected to any probe points are updated. Program execution then resumes until the next breakpoint. You can animate execution by pressing the button.

Note: Animate speed can be modified under the options menu, Customize... section.

- **Halt** − Lastly, you can halt execution at any time by pressing the halt button .

5.2.1.3 Multiprocessor Broadcast Commands (PDM+ only)

When using the Parallel Debug Manager (PDM), all run/step commands are broadcast to all target processors in the current group. If the device driver supports synchronous operation, each of the following commands is synchronized to start at the same time on each processor.

- ❑ You can use Step into to single step all processors that are not already running.
- ❑ You can use Step Over to execute a step over on all processors that are not already running.
- ❑ If all the processors are inside a subroutine, you can use Step Out to execute the step-out command on all the processors that are not already running.
- ❑ Sync Run sends a global run command to all processors that are not already running.
- ❑ Sync Halt halts all processors simultaneously.
- ❑ Animate starts animating all the processors that are not already running.

5.2.2 Breakpoints

Breakpoints are essential components of any debugging session.

Breakpoints stop the execution of the program. While the program is stopped, you can examine the state of the program, examine or modify variables, examine the call stack, etc. Breakpoints can be set on a line of source code in an Editor window or a disassembled instruction in the Disassembly window. After a breakpoint is set, it can be enabled or disabled.

For breakpoints set on source lines it is necessary that there be an associated line of dissassembly code. When compiler optimization is turned on, many source lines cannot have breakpoints set. To see allowable lines, use mixed mode in the editor window.

Note:

Code Composer Studio IDE tries to relocate a breakpoint to a valid line in your source window and places a breakpoint icon in the selection margin beside the line on which it locates the breakpoint. If an allowable line cannot be determined, it reports an error in the message window.

Note:

Code Composer Studio briefly halts the target whenever it reaches a probe point. Therefore, the target application may not meet real-time deadlines if you are using probe points. At this stage of development, you are testing the algorithm. Later, you can analyze real-time behavior using RTDX and DSP/BIOS.

Software Breakpoints

Breakpoints can be set in any Disassembly window or document window containing C/C++ source code. There is no limit to the number of software breakpoints that can be set, provided they are set at writable memory locations (RAM). Software breakpoints operate by modifying the target program to add a breakpoint instruction at the desired location.

The fastest way to set a breakpoint is to simply double-click on the desired line of code.

Step 1: In a document window or Disassembly window, move the cursor over the line where you want to place a breakpoint.

Step 2: Double-click in the Selection Margin immediately preceding the line when you are in a document window.

In a Disassembly window, double-click on the desired line.

A breakpoint icon in the Selection Margin indicates that a breakpoint has been set at the desired location.

The Toggle Breakpoint command and the Toggle Breakpoint button also enable you to quickly set and clear breakpoints.

Step 1: In a document window or Disassembly window, put the cursor in the line where you want to set the breakpoint.

Step 2: Right-click and select Toggle Breakpoint, or click its icon on the Project toolbar.

Hardware Breakpoints

Hardware breakpoints differ from software breakpoints in that they do not modify the target program; they use hardware resources available on the chip. Hardware breakpoints are useful for setting breakpoints in ROM memory or breaking on memory accesses instead of instruction acquisitions. A breakpoint can be set for a particular memory read, memory write, or memory read or write. Memory access breakpoints are not shown in the source or memory windows.

Hardware breakpoints can also have a count, which determines the number of times a location is encountered before a breakpoint is generated. If the count is 1, a breakpoint is generated every time. Hardware breakpoints cannot be implemented on a simulated target.

To set a hardware breakpoint:

Step 1: Select Debug→Breakpoints. The Break/Probe Points dialog box appears with the Breakpoints tab selected.

Step 2: In the Breakpoint type field, choose “H/W Break at location” for instruction acquisition breakpoints or choose “Break on <bus> <Read|Write|R/W>” at location for a memory access breakpoint.

Step 3: Enter the program or memory location where you want to set the breakpoint. Use one of the following methods:

- For an absolute address, you can enter any valid C expression, the name of a C function, or a symbol name.
- Enter a breakpoint location based on your C source file. This is convenient when you do not know where the C instruction is located in the executable. The format for entering in a location based on the C source file is as follows: fileName line lineNumber.

Step 4: Enter the number of times the location is hit before a breakpoint is generated, in the Count field. Set the count to 1 if you wish to break every time.

Step 5: Click the Add button to create a new breakpoint. This causes a new breakpoint to be created and enabled.

Step 6: Click OK.

5.2.3 Probe Points

In this section, you add a probe point, which reads data from a file on your PC. Probe Points are a useful tool for algorithm development. You can use probe points to:

- ❑ Transfer input data from a file on the host PC to a buffer on the target for use by the algorithm.
- ❑ Transfer output data from a buffer on the target to a file on the host PC for analysis.
- ❑ Update a window, such as a graph, with data.

More Information About Probe Points

Probe points are similar to breakpoints in that they both halt the target to perform their action. However, probe points differ from breakpoints in the following ways:

- ❑ Probe Points halt the target momentarily, perform a single action, and resume target execution.
- ❑ Breakpoints halt the CPU until execution is manually resumed and cause all open windows to be updated.
- ❑ Probe points permit automatic file input or output to be performed; breakpoints do not.

This section shows how to use a probe point to transfer the contents of a PC file to the target for use as test data. It also uses a breakpoint to update all the open windows when the Probe Point is reached.

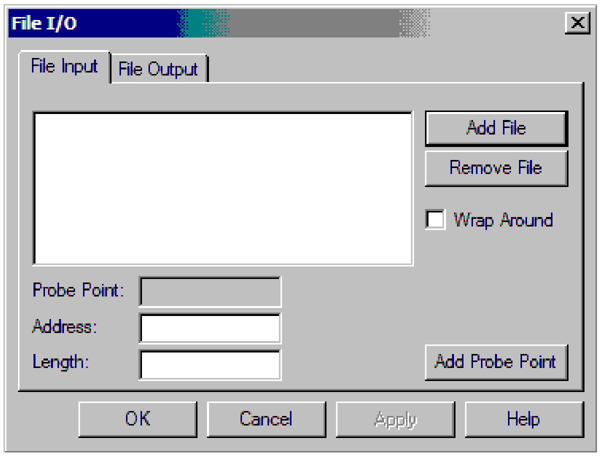
Step 1: Choose File→Load Program. Select *filename*.out, and click Open.

Step 2: Double-click on the *filename*.c file in the Project View.

Step 3: Put your cursor in a line of the main function to which you want to add a probe point.

Step 4: Click the Toggle Probe Point toolbar button.

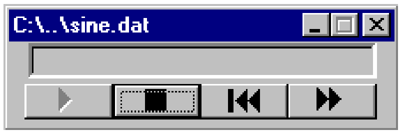
Step 5: From the File menu, choose File I/O. The File I/O dialog appears so that you can select input and output files.



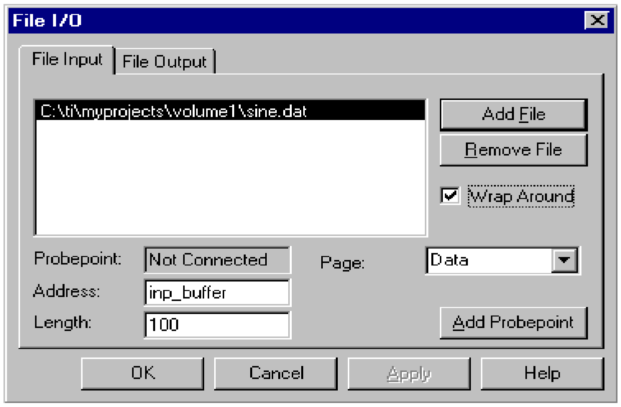
Step 6: In the File Input tab, click Add File.

Step 7: Browse to your project folder, select *filename*.dat and click Open.

A control window for the *filename*.dat file appears. When you run the program, you can use this window to start, stop, rewind, or fast forward within the data file.



Step 8: In the File I/O dialog, change the Address and the Length values. Also, put a check mark in the Wrap Around box.



C:\ti\myprojects\volume1\sine.dat

☑ Wrap Around

Probepoint: Not Connected

Page: Data

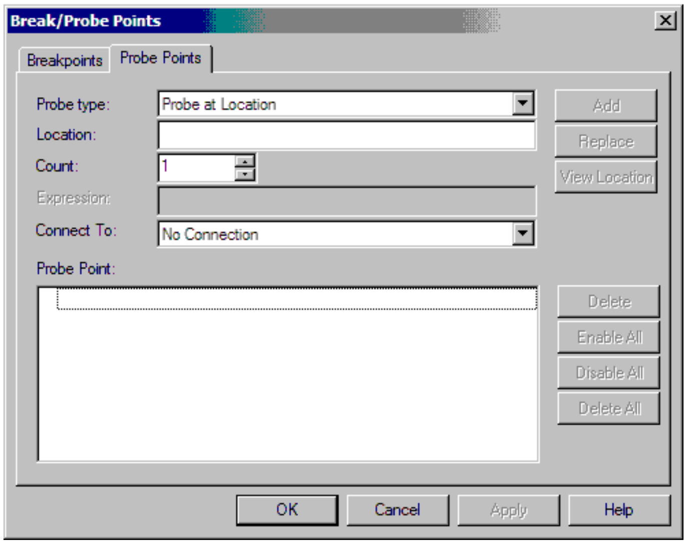
Address: inp_buffer

Length: 100

Add Probepoint

- The Address field specifies where the data from the file is to be placed.
- The Length field specifies how many samples from the data file are read each time the Probe Point is reached.
- The Wrap Around option enables the data to start being read from the beginning of the file when it reaches the end of the file. This allows the data file to be treated as a continuous stream of data.

Step 9: Click Add Probe Point. The Probe Points tab of the Break/Probe Points dialog appears.



Probe type: Probe at Location

Location:

Count: 1

Expression:

Connect To: No Connection

Probe Point:

Step 10: In the Probe Point list, highlight a line.

Step 11: In the Connect To field, click the down arrow and select a file from the list.

Step 12: Click Replace. The Probe Point list changes to show that this Probe Point is connected to the sine.dat file.

Step 13: Click OK. The File I/O dialog shows that the file is now connected to a Probe Point.

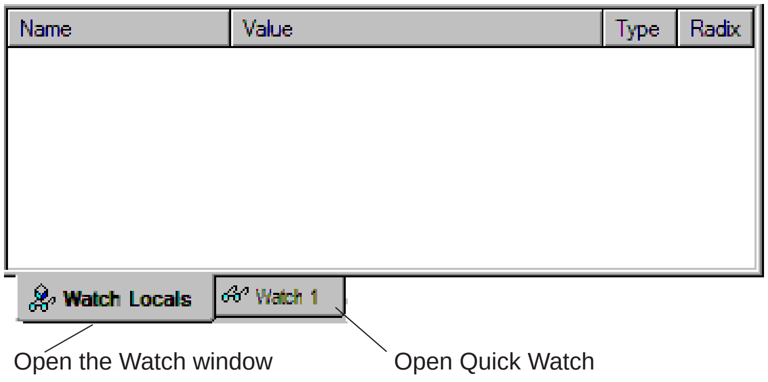
Step 14: Click OK to close the File I/O dialog.

5.2.4 Watch Window

When debugging a program, it is often helpful to understand how the value of a variable changes during program execution. The Watch window allows you to monitor the values of local and global variables and C/C++ expressions.

To open the Watch window:

Step 1: Select View→Watch Window, or click the Watch Window button on the Watch toolbar.



The Watch window, contains two tabs labeled: Watch Locals and Watch.

❑ In the Watch Locals tab, the debugger automatically displays the Name, Value, and Type of the variables that are local to the currently executing function.

❑ In the Watch tab, the debugger displays the Name, Value, and Type of the local and global variables and expressions that you specify.

❑ For detailed information on the Watch Window, see the Watch Window topics provided in the online help: Help→Contents→Watch Window.

❑ When you are developing and testing programs, you often need to check the value of a variable during program execution. You use breakpoints and the Watch Window to view such values.

Step 2: Choose File→Load Program.

Step 3: Double−click on the *filename*.c file in the Project View.

Step 4: Put your cursor in a line that allows breakpoints.

Step 5: Click the Toggle Breakpoint toolbar button or press F9.

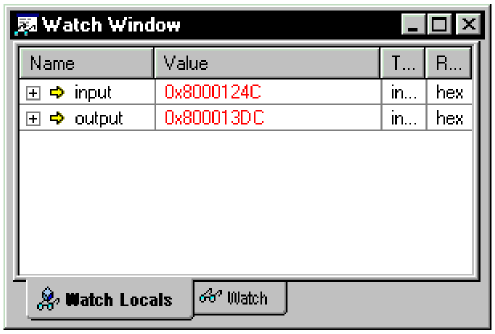
The selection margin indicates that a breakpoint has been set (red icon). If you disable the selection margin (Options→Customize) the line is highlighted in magenta.

Step 6: Choose View→Watch Window.

A separate area in the lower−right corner of the window appears. At run time, this area shows the values of watched variables.
By default, the Locals tab is selected and displays variables that are local to the function being executed.

Step 7: If not at main, choose Debug→Go Main.

Step 8: Choose Debug→Run, or press F5, or press the Icon.

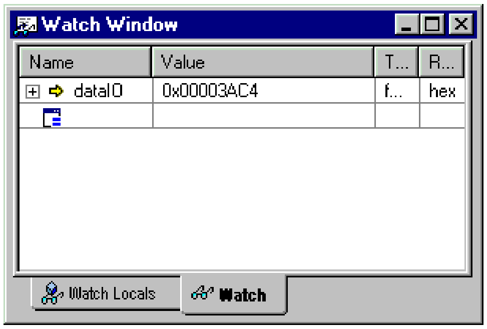


Name	Value	T...	R...
input	0x8000124C	in...	hex
output	0x800013DC	in...	hex

Step 9: Select the Watch tab.

Step 10: Click on the Expression icon in the Name column and type the name of the variable to watch.

Step 11: Click on the white space in the watch window to save the change. The value should immediately appear, similar to this example.



Name	Value	T...	R...
dataIO	0x00003AC4	f...	hex

Step 12: Click the Step Over toolbar button or press F10 to step over the call to your watched variable.

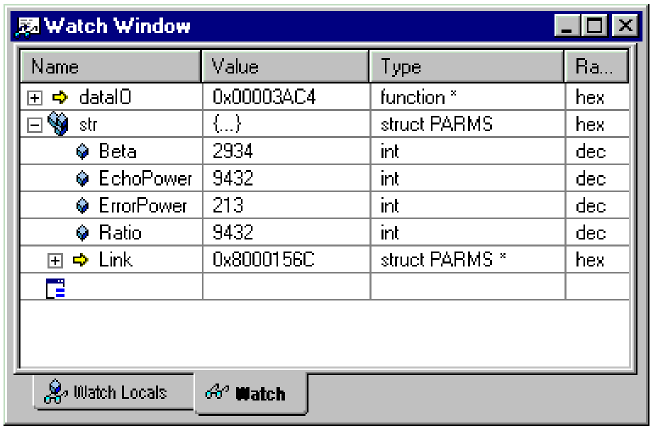
In addition to watching the value of a simple variable, you can watch the values of the elements of a structure.

Step 1: Select the Watch tab.

Step 2: Click on the Expression icon in the Name column and type the name of the expression to watch.

Step 3: Click on the white space in the watch window to save the change.

Step 4: Click once on the + sign. The line expands to list all the elements of the structure and their values. (The address shown for Link may vary.)



Name	Value	Type	Ra...
dataIO	0x00003AC4	function *	hex
str	{...}	struct PARMS	hex
Beta	2934	int	dec
EchoPower	9432	int	dec
ErrorPower	213	int	dec
Ratio	9432	int	dec
Link	0x8000156C	struct PARMS *	hex

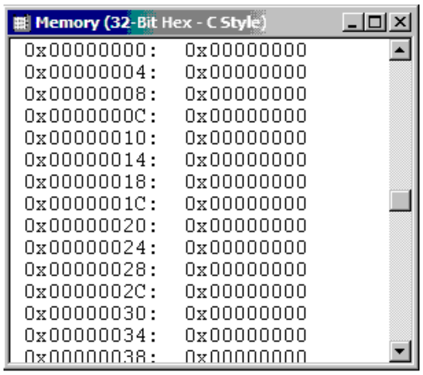
Step 5: Double-click on the Value of any element in the structure to edit the value for that element.

Step 6: Change the value of a variable.

Notice that the value changes in the Watch Window. The value also changes color to red, indicating that you have changed it manually.

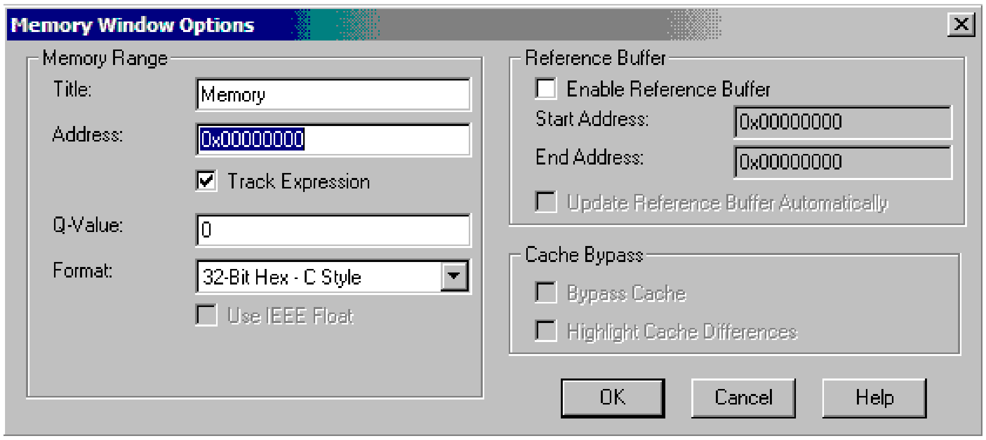
5.2.5 Memory Window

The Memory window allows you to view the contents of memory starting at a specified address. Options enable you to format the Memory window display. You can also edit the contents of a selected memory location.



```
0x00000000:  0x00000000
0x00000004:  0x00000000
0x00000008:  0x00000000
0x0000000C:  0x00000000
0x00000010:  0x00000000
0x00000014:  0x00000000
0x00000018:  0x00000000
0x0000001C:  0x00000000
0x00000020:  0x00000000
0x00000024:  0x00000000
0x00000028:  0x00000000
0x0000002C:  0x00000000
0x00000030:  0x00000000
0x00000034:  0x00000000
0x00000038:  0x00000000
```

The Memory Window Options dialog box allows you to specify various characteristics of the Memory window.



Title: Memory
Address: 0x00000000
[✓] Track Expression
Q-Value: 0
Format: 32-Bit Hex - C Style
[ ] Use IEEE Float
[ ] Enable Reference Buffer
Start Address: 0x00000000
End Address: 0x00000000
[ ] Update Reference Buffer Automatically
[ ] Bypass Cache
[ ] Highlight Cache Differences

The Memory Window Options dialog offers the following options:

- **Title.** Enter a meaningful name for the Memory window. When the Memory window is displayed, the name appears in the title bar. This feature is especially useful when multiple Memory windows are displayed.

- **Address.** Enter the starting address of the memory location you want to view.
- **Q Value.** You can display integers using a Q value. This value is used to represent integer values as more precise binary values. A decimal point is inserted in the binary value; its offset from the least significant bit (LSB) is determined by the Q value as follows:

  New_integer_value = integer / (2^(Q value))

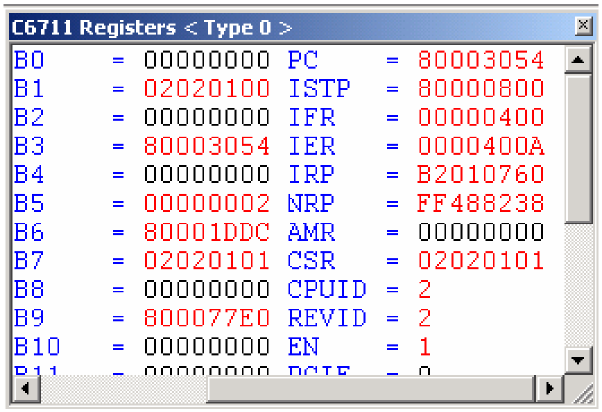
  A Q value of xx indicates a signed 2s complement integer whose decimal point is displaced xx places from the least significant bit (LSB).
- **Format.** From the drop−down list, select the format of the memory display. More information on the different formats can be found in the on−line help.
- **Enable Reference Buffer.** Save a snapshot of a specified area of memory that can be used for later comparison.
- **Start Address.** Enter the starting address of the memory locations you want to save in the Reference Buffer. This field only becomes active when Enable Reference Buffer is selected.
- **End Address.** Enter the ending address of the memory locations you want to save in the Reference Buffer. This field only becomes active when Enable Reference Buffer is selected.
- **Update Reference Buffer Automatically.** Select this checkbox to automatically overwrite the contents of the Reference Buffer with the current contents of memory at the specified range of addresses. When this option is selected, the Reference Buffer is updated whenever the Memory window is refreshed (for example, Refresh Window is selected, a breakpoint is hit, or execution on the target is halted). If this checkbox is not selected, the contents of the Reference Buffer are not changed. This option only becomes active when Enable Reference Buffer is selected.
- **Bypass Cache Differences.** This option forces the memory to always read memory contents from physical memory. Normally, if a memory’s contents were in cache, the returnedmemory value would display the value from cache, not from physicalmemory. If this option is turned on, Code Composer Studio will ignore or bypass the cached memory contents.
- **Highlight Cache Differences.** This option highlights the value of memory locations when the cached value and physical value

differ. It is also possible to use colors tohighlight the cache difference. Go to Options→Customize→Color and select the Cache Bypass Differences option under the Screen element dropdown box.

Please refer to the online section on the Memory Window for more detailed information.

5.2.6 Register Window

The Register window enables you to view and edit the contents of various registers on the target.



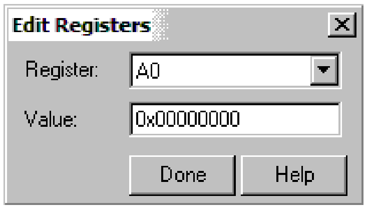
```
B0      = 00000000 PC     = 80003054
B1      = 02020100 ISTP   = 80000800
B2      = 00000000 IFR    = 00000400
B3      = 80003054 IER    = 0000400A
B4      = 00000000 IRP    = B2010760
B5      = 00000002 NRP    = FF488238
B6      = 80001DDC AMR    = 00000000
B7      = 02020101 CSR    = 02020101
B8      = 00000000 CPUID  = 2
B9      = 800077E0 REVID  = 2
B10     = 00000000 EN     = 1
```

To access the Register Window:

❑ Select View→Registers and select the register set that you would like to view/edit

To Edit the Contents of a Register:

❑ Select Edit→Edit Register, or from the Register window, double-click a register, or right-click in the Register window and select Edit Register.



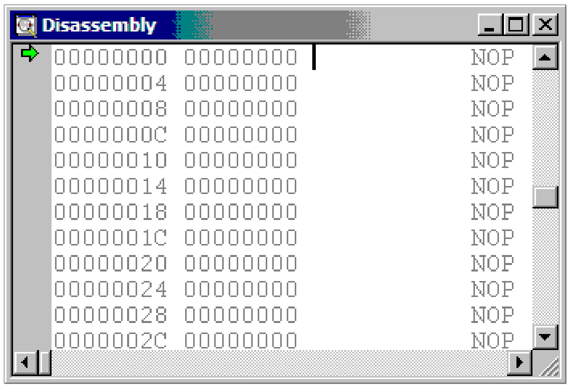
Register: A0

Value: 0x00000000

5.2.7 Disassembly/Mixed Mode

Disassembly

When you load a program onto your actual or simulated target, the debugger automatically opens a Disassembly window.



```
00000000 00000000      NOP
00000004 00000000      NOP
00000008 00000000      NOP
0000000C 00000000      NOP
00000010 00000000      NOP
00000014 00000000      NOP
00000018 00000000      NOP
0000001C 00000000      NOP
00000020 00000000      NOP
00000024 00000000      NOP
00000028 00000000      NOP
0000002C 00000000      NOP
```

The Disassembly window displays disassembled instructions and symbolic information needed for debugging. Disassembly reverses the assembly process and allows the contents of memory to be displayed as assembly language code. Symbolic information consists of symbols and strings of alphanumeric characters that represent addresses or values on the target.

As you step through your program using the stepping commands, the PC advances to the appropriate instruction. For the sections of your program code that are written in C, you can choose to view mixed C source and assembly code.

Mixed Mode

In addition to viewing disassembled instructions in the Disassembly window, the debugger enables you to view your C source code interleaved with disassembled code, allowing you to toggle between source mode and mixed mode.

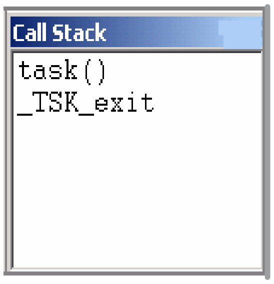
To change your selection, toggle View→Mixed Source/ASM; or right-click in the source window and select Mixed Mode or Source Mode, depending on your current selection.

5.2.8 Call Stack

Use the Call Stack window to examine the function calls that led to the current location in the program that you are debugging.

To Display the Call Stack:

Step 1: Select View→Call Stack, or click the View Stack button on the Debug toolbar.



```
task()
_TSK_exit
```

Step 2: Double-click on a function listed in the Call Stack window. The source code containing that function is displayed in a document window. The cursor is set to the current line within the desired function.

Once you select a function in the Call Stack window, you can observe local variables that are within the scope of that function.

The call stack only works with C programs. Calling functions are determined by walking through the linked list of frame pointers on the runtime stack. Your program must have a stack section and a main function; otherwise, the call stack displays the message: C source is not available. Also note that the Call Stack Window displays only the first 100 lines of output. Any amount of lines over 100 will be omitted from the display.

5.2.9 Symbol Browser

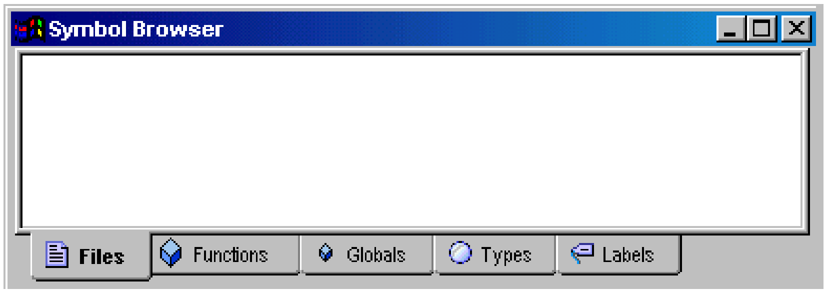
The Symbol Browser window (Figure 5−1) displays five tabbed windows:

❑ All associated files;
❑ functions;
❑ global variables;
❑ types; and
❑ labels of a loaded COFF output file (*.out).

Each tabbed window contains nodes representing various symbols. A plus sign (+) preceding a node indicates that the node can be further expanded. To expand the node, simply click the + sign. A minus sign (−) precedes an expanded node. Click the − sign to hide the contents of that node.

To open the Symbol Browser window, select Tools→Symbol Browser.

Figure 5−1. Symbol Browser Window



For detailed information on the Symbol Browser tool, see the Symbol Browser topics provided in the online help: Help→Contents→Symbol Browser.

5.2.10 Command Window

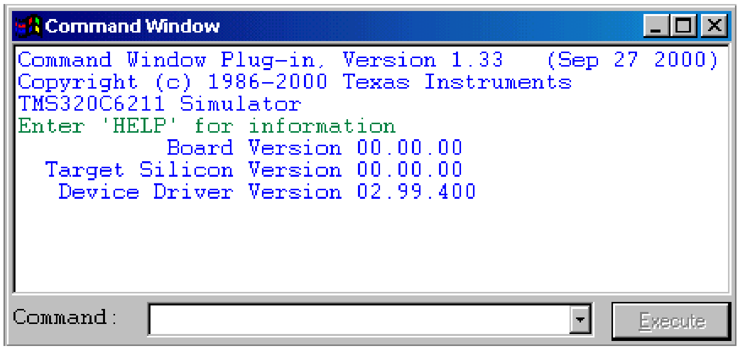
The Command Window enables you to specify commands to the debugger using the TI Debugger command syntax.

Many of the commands accept C expressions as parameters. This allows the instruction set to be relatively small, yet powerful. Because C expressions can have side effects (that is, the evaluation of some types of expressions can affect existing values) you can use the same command to display or change a value.

To open the Command Window:

Select Tools→Command Window from the menu bar.

Figure 5−2. Command Window



```
Command Window Plug-in, Version 1.33   (Sep 27 2000)
Copyright (c) 1986-2000 Texas Instruments
TMS320C6211 Simulator
Enter 'HELP' for information
               Board Version 00.00.00
  Target Silicon Version 00.00.00
   Device Driver Version 02.99.400
```

For detailed information on the Command Window, see the Command Window topics provided in the online help: Help→Contents→Command Window.

5.3 Advanced Debugging Features

5.3.1 Thread Level Debugging

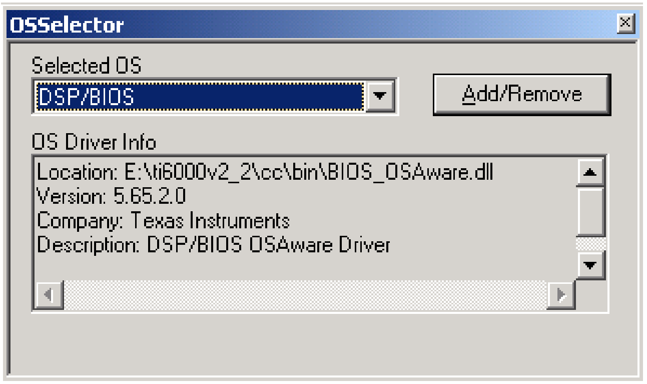
Thread level debugging means that the debugger halts when the thread it is associated with reaches it. Target execution commands such as stepping, halting, and running can also be specific to a selected thread. Also, information displayed by certain debug windows can pertain to the thread it is related to – such as variable information displayed in the watch window, or values of context-saved registers for non−executing threads.

To enable Thread Level Debugging:

Step 1: Specify the target OS to the debugger. This can be done by going to Tools→OS Selector to launch the OSSelector dialog box.

Note: DSP/BIOS has been configured as the default OS upon installation of Code Composer Studio. If the target OS is DSP/BIOS then the user can skip to step 4.

Step 2: Select the target OS from a list of currently installed drivers in the drop down menu list. Note that DSP/BIOS is configured as the default OS. If the OS of choice is not available then it must be added to the list by selecting 'Add/Remove' and then browsing for the OS TLI driver provided by Code Composer Studio or a third−party vendor depending on the OS.



Selected OS: DSP/BIOS

OS Driver Info

```
Location: E:\ti6000v2_2\cc\bin\BIOS_OSAware.dll
Version: 5.65.2.0
Company: Texas Instruments
Description: DSP/BIOS OSAware Driver
```

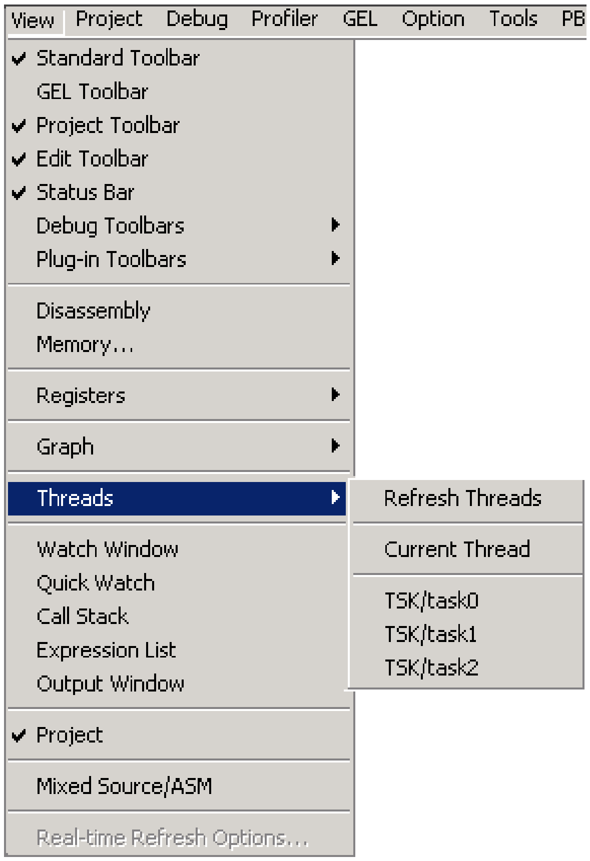
Step 3: Close the OS Selector dialog box

Step 4: Select File→Load Program and select the program you wish to debug

Step 5: Select Debug→Go Main. Wait until the application reaches 'main'

Step 6: Select Debug→Enable Thread Level Debugging…

Thread Level Debugging has now been enabled for the debug session. The developer can now access the options under View→Threads.



The Threads menu will contain an alphabetical list of all tasks and SWI's in the system. The list can include both statically and dynamically created threads and the list can change depending on the creation and destruction of threads in the system. The list is automatically updated whenever the target is halted. The Refresh Threads option will also refresh the list of threads. Selecting a listed thread for the first time will launch a Thread Control Window (TCW). The subsequent selections of the threads in the list will select the Thread Control Window of the thread. This is an easy way to switch between multiple Thread Control Windows. There is also an option in the menu called Current Thread, which will allow for easy access to the current running thread.

The Thread Control Window is the main interface for debugging a particular thread. It has the same look and feel as the Main Control Window; however, it contains context information for the thread associated with a thread control window. Debug commands and information are only in context of that thread.

More information on Task Control Windows and Task Level Debugging in general can be found in the online help.

5.3.2 Advanced Event Triggering (AET)

Advanced Event Triggering (AET) is supported by a group of tools that makes hardware analysis easier than before. AET uses Event Analysis and Event Sequencer to simplify hardware analysis.

Event Analysis uses a simple interface to help you configure common hardware debug tasks called jobs. You can easily set breakpoints, action points, and counters by using a right-click menu and performing a simple drag-and-drop. You can access Event Analysis from the tools menu, or by right-clicking in a source file.

Event Sequencer allows you to look for conditions that you specify in your target program and initiates specific actions when these conditions are detected. While the CPU is halted, you define the conditions and actions, then run your target program. The sequence program then looks for the condition you specified and performs the action you requested.

Event Analysis

The following jobs can be performed using Event Analysis:

- Setting Breakpoints
  - Hardware Breakpoint
  - Hardware Breakpoint With Count
  - Chained Breakpoint
  - Global Hardware Breakpoint

- Setting Action/Watch Points
  - Data Actionpoint
  - Program Actionpoint
  - Watchpoint
  - Watchpoint With Data

- Setting Counters
  - Data Access Counter
  - Profile Counter
  - Watchdog Timer
  - Generic Counter

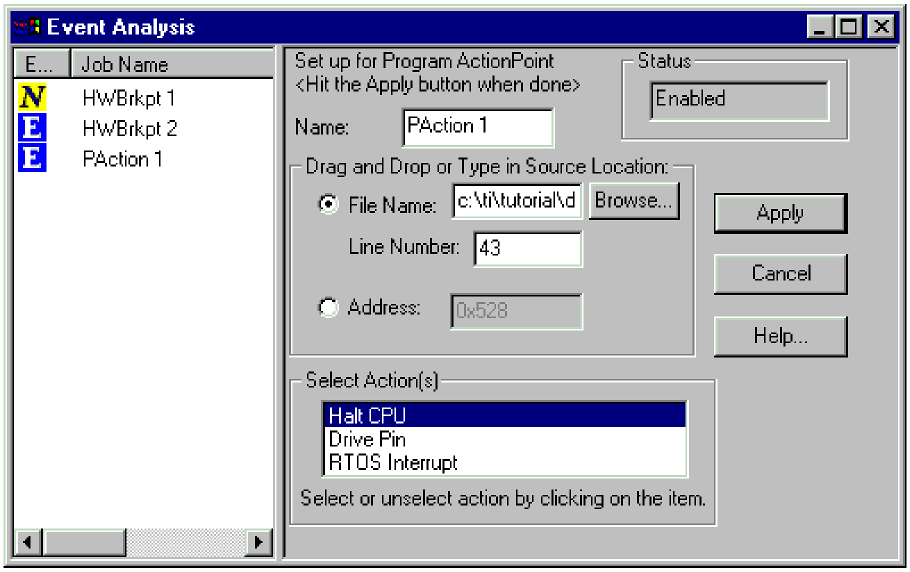
- ❏ Other
  - ■ Benchmark to Here
  - ■ Emulation Pin Configuration

For detailed information on the Event Analysis tool, see the Event Analysis topics provided in the online help: Help→Contents→Advanced Event Triggering.

To configure a job using the Event Analysis Tool, Code Composer Studio IDE must be configured for a target processor that contains on-chip analysis features. You can use Event Analysis by selecting it from the Tools menu or by right-clicking in a source file. Once you configure a job, it is enabled and will perform analysis when you run code on your target. For information about how to enable or disable a job that is already configured, see the Advance Event Triggering online help.

Step 1: Select Tools→Advanced Event Triggering→Event Analysis.

The Event Analysis window displays.



E...	Job Name
N	HWBrkpt 1
E	HWBrkpt 2
E	PAction 1

Set up for Program ActionPoint
<Hit the Apply button when done>

Name: PAction 1

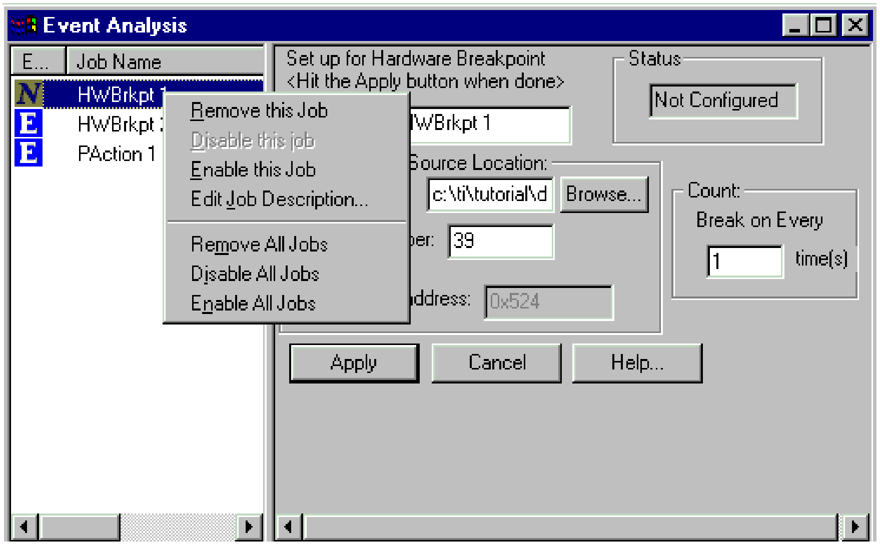
Status: Enabled

Drag and Drop or Type in Source Location:
[✓ File Name: c:\ti\tutorial\d]
Line Number: 43
Address: 0x528

Select Action(s)
[✓ Halt CPU]
Drive Pin
RTOS Interrupt
Select or unselect action by clicking on the item.

Step 2: Right-click in the Event Analysis Window and choose Event Triggering→Job Type→Job.

The job menu is dynamically built and dependent on the target configuration. If a job is not supported on your target, the job is grayed out.



Step 3: Type your information in the Job dialog box.

Step 4: Click Apply to program the job and save your changes.

Event Sequencer

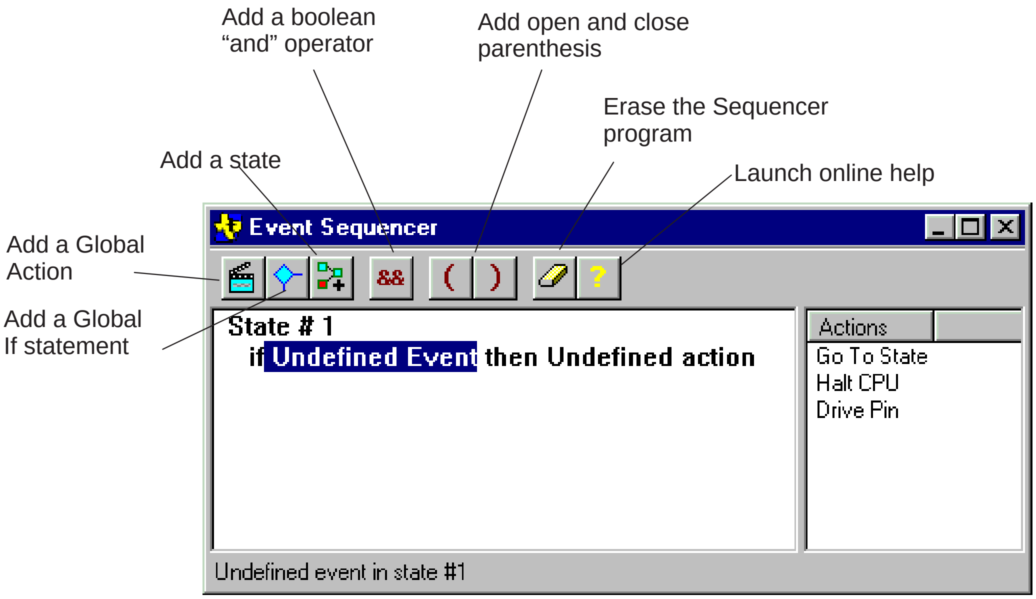
The Event Sequencer allows you to look for conditions that you specify in your target program and initiates specific actions when these conditions are detected. While the CPU is halted, you define the actions, then run your target program. The sequencer program then looks for the condition that you specified and performs the action you requested.

To use the Event Sequencer, Code Composer Studio IDE must be configured for a target processor that contains on-chip analysis features. You can use the Event Sequencer by selecting it from the Tools menu. Once you create an Event Sequencer program, it is enabled and performs analysis when you run code on your target. For information on creating an Event Sequencer program, see the Advanced Event Triggering online help.

To enable the Event Sequencer:

Step 1: Select Tools→Advanced Event Triggering→Event Sequencer.

The Event Sequencer window displays.


Add a boolean "and" operator
Add open and close parenthesis
Erase the Sequencer program
Add a state
Launch online help
Add a Global Action
Add a Global If statement
Event Sequencer
Actions
Go To State
Halt CPU
Drive Pin

State # 1
if Undefined Event then Undefined action

Undefined event in state #1

Step 2: Right-click in the Event Sequencer window or use the Event Sequencer toolbar buttons to create a sequencer program.

5.4 Real-Time Debugging

Traditional debugging approaches (Stop Mode) require that programmers completely halt their system, which stops all threads and prevents interrupts from being handled. Stop Mode can exclusively be used for debug as long as the system/application does not have any *real-time* constraints. However there will be times when the developer will want a better gauge of their application's real-world system behavior. Code Composer Studio offers several options to help with this effort.

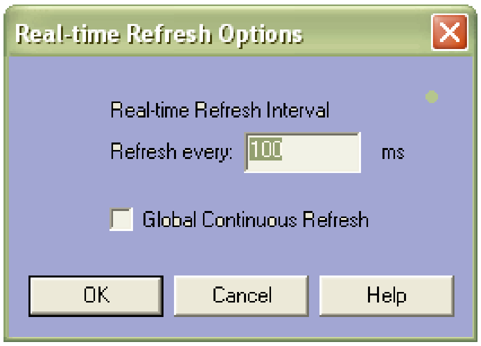
5.4.1 Real−Time Mode

Real-Time Mode debug support provides a better gauge of real−world system behavior by enabling programmers to halt and examine the application while allowing user specified *time critical* interrupts to be handled. You can suspend program execution in multiple locations, which allows you to break within one time−critical interrupt while still servicing others.

Enabling Real-Time Mode Debug:

Step 1: Select Debug→Real-Time Mode. The status bar at the bottom of the Code Composer Studio window now indicates POLITE REALTIME.

Step 2: Configure the real−time refresh options by selecting View→Real-Time Refresh Options…



Real-time Refresh Interval

Refresh every: 100 ms

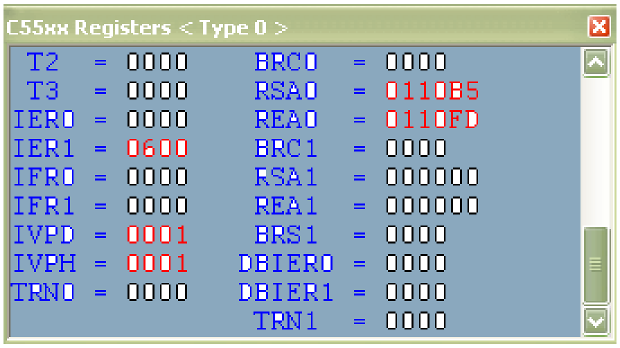
Global Continuous Refresh

Configure the options if desired. The first option will specify how often the Watch Window is updated. Checking the Global Continuous Refresh checkbox will continuously refresh all windows that display target data, including memory/graph/watch windows. To continuously update only a certain window, uncheck this box and select Continuous Refresh from the window's context menu.

Step 3: Click OK to close the dialog box.

Step 4: Select View→Registers→CPU Registers to open the CPU Register Display.

The DIER is used to designate a single, a specific subset, or all interrupts that are user selected via the IER as Real-Time (time-critical) interrupts. DIER mirrors the architecturally specified Interrupt Enable Register (IER)


C55xx Registers < Type 0 >

```
 T2   = 0000      BRC0   = 0000
 T3   = 0000      RSA0   = 0110B5
IER0  = 0000      REA0   = 0110FD
IER1  = 0600      BRC1   = 0000
IFR0  = 0000      RSA1   = 000000
IFR1  = 0000      REA1   = 000000
IVPD  = 0001      BRS1   = 0000
IVPH  = 0001     DBIER0  = 0000
TRN0  = 0000     DBIER1  = 0000
                  TRN1   = 0000
```

Step 5: Right-click on a register and select Edit Register… and enter the new register value that will specify which interrupts to designate interrupts as real−time interrupts:

Step 6: Click Done to close the Edit Registers dialog box.

Step 7: Code Composer Studio has been configured for Real-Time Mode debug.

Rude Real−Time Mode

High priority interrupts, or other sections of code can be extremely time-critical, and the number of cycles taken to execute them must be kept at a minimum or to an exact number. This means debug actions (both execution control and register/memory accesses) may need to be prohibited in some code areas or targeted at a specific machine state. In real-time mode, by default the processor runs in ‘polite’ mode by absence of privileges, i.e., debug actions will respect the appropriate delaying of the action and not intrude in the debug sensitive windows.

However, debug commands (both execution control and register/memory access) can fail if they are not able to find a window that is not marked debug action-sensitive. In order to have the debugger gain control, you must change real−time debug from ‘polite’ to ‘rude’ mode. In rude real-time mode, the possession of privileges allows a debug action to override any protection that may prevent debug access and be executed successfully without delay. Also, the user can not debug critical code regions until they are switched into rude real−time mode.

To enable Rude Real-Time mode, perform one of the following:

- ❑ Select Perform a Rude Retry from the display window when a debug command failed.
- ❑ Select Enable Rude Real-Time Mode from Debug menu when Real-Time is turned on.

When Rude Real-Time is enabled, the status bar at the bottom of the main program window displays RUDE REALTIME. To disable Rude Real-Time, deselect the Enable Rude Real-Time Mode item in the Debug menu. The status bar now reads POLITE REALTIME.

If Rude Real-Time is enabled and you halt the CPU, there is a good chance that the CPU will halt even when debug accesses are blocked, which might be within a time-critical ISR. This prevents the CPU from completing that ISR in the appropriate amount of time, as the CPU cannot do anything until you respond to the breakpoint. To prevent this problem, you must switch back to Polite real-time mode by deselecting Enable Rude Real-Time Mode.

Please refer to the online help section Debugging→Real Time Debugging for more detailed information on Real-Time Mode.

5.4.2 Real-Time Data Exchange (RTDX)

DSP/BIOS Real-Time Analysis (RTA) facilities utilize the Real-Time Data Exchange (RTDX) link to obtain and monitor target data in real-time. You can utilize the RTDX link to create your own customized interfaces to the DSP target by using the RTDX API Library.

Real-time data exchange (RTDX) allows system developers to transfer data between a host computer and target devices without interfering with the target application. This bi-directional communication path provides for data collection by the host as well as host interaction with the running target application. The data collected from the target may be analyzed and visualized on the host. Application parameters may be adjusted using host tools, without stopping the application. RTDX also enables host systems to provide data stimulation to the target application and algorithms.

RTDX consists of both target and host components. A small RTDX software library runs on the target application. The target application makes function calls to this library’s API in order to pass data to or from it. This library makes use of a scan-based emulator to move data to or from the host platform via a JTAG interface. Data transfer to the host occurs in real-time while the target application is running.

On the host platform, an RTDX Host Library operates in conjunction with Code Composer Studio IDE. Data visualization and analysis tools communicate with RTDX through COM APIs to obtain the target data and/or to send data to the DSP application.

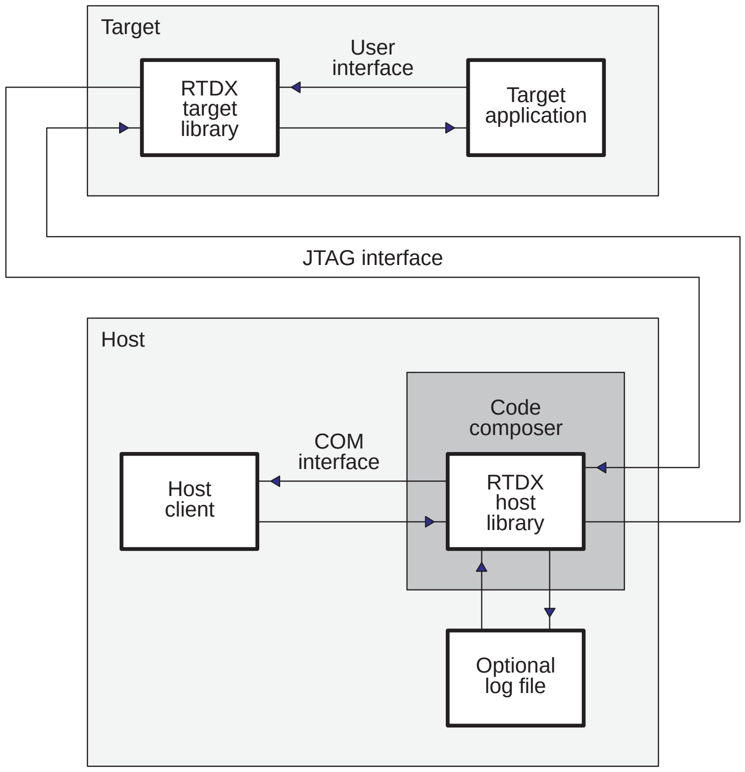
The host library supports two modes of receiving data from a target application: continuous and non-continuous. In Continuous mode, the data is simply buffered by the RTDX Host Library and is not written to a log file. Continuous mode should be used when the developer wants to continuously obtain and display the data from a target application, and does not need to store the data in a log file. In Non Continuous mode, data is written to a log file on the host. This mode should be used when developers want to capture a finite amount of data and record it in a log file.

For details on using RTDX, see the Help→Contents→RTDX or Help→Tutorial→RTDX Tutorial.

RTDX Data Flow

RTDX forms a two-way data pipe between a target application and a host client. This data pipe consists of a combination of hardware and software components as shown below.

Figure 5−3. RTDX Data Flow



Configuring RTDX Graphically

The RTDX tools allow you to configure RTDX graphically, set up RTDX channels, and run diagnostics on RTDX. These tools allow you to enhance RTDX functionality when transmitting data.

RTDX has three menu options:

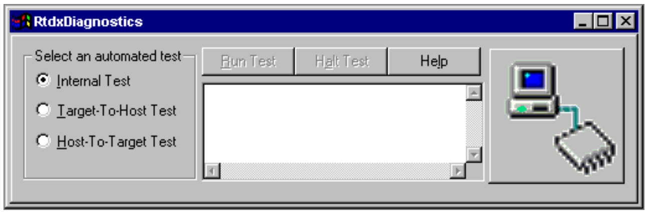
- ❑ Diagnostics Control
- ❑ Configuration Control
- ❑ Channel Viewer Control

Diagnostics Control

RTDX provides the RTDX Diagnostics Control to verify that RTDX is working correctly on your system. The diagnostic tests test the basic functionality of target−to−host transmission and host−to−target transmission.

To open the RTDX Diagnostics Control, select Tools→RTDX→Diagnostics Control.

Figure 5−4. RTDX Diagnostics Window



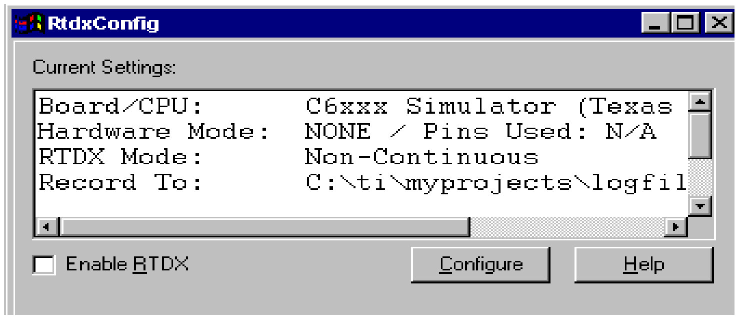
Configuration Control

Configuration Control is the main RTDX window. It allows you to do the following:

- ❑ View the current RTDX configuration settings
- ❑ Enable or disable RTDX
- ❑ Access the RTDX Configuration Control Properties page to reconfigure RTDX and select port configuration settings

To open the RTDX Configuration Control, select Tools→RTDX→Configuration Control.

Figure 5−5. RTDX Config Window



```
Board/CPU:        C6xxx Simulator (Texas
Hardware Mode:    NONE / Pins Used: N/A
RTDX Mode:        Non-Continuous
Record To:        C:\ti\myprojects\logfil
```

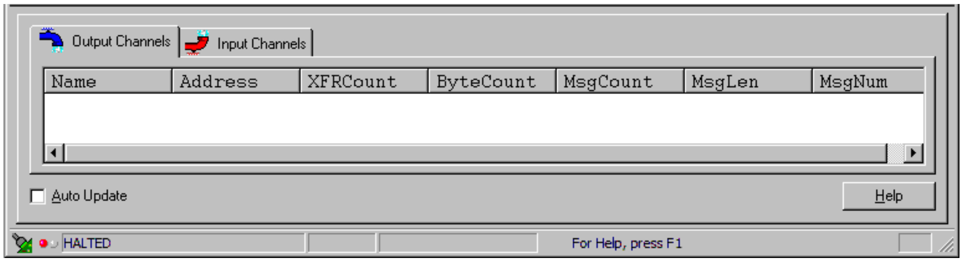
Channel Viewer Control

The RTDX Channel Viewer Control is an Active X control that automatically detects target-declared channels and adds them to the viewable list. The RTDX Channel Viewer Control also allows you to:

- ❑ Remove a target-declared channel from the viewable list
- ❑ Re-add a target−declared channel to the viewable list
- ❑ Enable or disable a channel that is in the list

To open the RTDX Channel Viewer Control in Code Composer Studio, select Tools→RTDX→Channel Viewer Control. The Channel Viewer Control window displays.

Figure 5−6. RTDX Channel Viewer Window



Click on the Input and Output Channels tabs to display a list of those channels. Both the Output and Input Channels windows allow you to view, delete, and re-add channels.

Checking the Auto-Update feature enables you to automatically update information for all channels without refreshing the display. If you are not using the Auto Update feature, from the right-click menu, select Refresh to update information for all channels.

Note: For the RTDX Channel View Control to receive extended channel information for a specific channel, a RTDX client must have the channel of interest open.

Transmit a Single Integer to the Host

The basic function of RTDX is to transmit a single integer to the host. The following steps provide an overview of the process of sending data from the target to the host and from the host to the target. For specific commands and details on transmitting different types of data, see the Help→Contents→RTDX or Help→Tutorial→RTDX Tutorial.

To send data from your target application to the host:

Step 1: Prepare your target application to capture real-time data.

This involves inserting specific RTDX syntax into your application code to allow real-time data transfer from the target to the host. Although the process for preparing a target application is the same for all data types, different data types require different function calls for data transfer. Therefore, sending an integer to the host requires you to add a function call that is specific to only transmitting a single integer as compared to sending an array of integers to the host.

Step 2: Prepare your host client to process the data.

This involves instantiating one RTDX object for each desired channel, opening a channel for the objects specified, and calling any other desired functions.

Step 3: Start Code Composer Studio IDE.

Step 4: Load your target application onto the TI processor.

Step 5: Enable RTDX: Tools→RTDX→Configuration Control.

The Configuration Control window displays.

Step 6: Run your target application to capture real-time data and send it to the RTDX Host Library.

Step 7: Run your host client to process the data.

For details on using RTDX, see the Help→Contents→RTDX or Help→Tutorial→RTDX Tutorial.

Transmit Data from the Host to the Target

A client application can send data to the target application by writing data to the target. Data sent from the client application to the target is first buffered in the RTDX Host Library. The data remains in the RTDX Host Library until a request for data arrives from the target. Once the RTDX Host Library has enough data to satisfy the request, it writes the data to the target without interfering with the target application.

The state of the buffer is returned into the variable buffer state. A positive value indicates the number of bytes the RTDX Host Library has buffered, which the target has not yet requested. A negative value indicates the number of bytes that the target has requested, which the RTDX Host Library has not yet satisfied.

To send data from a host client to your target application:

Step 1: Prepare your target application to receive data.

This involves writing a simple RTDX target application that reads data from the host client.

Step 2: Prepare your host client to send data.

This involves instantiating one RTDX object for each desired channel, opening a channel for the objects specified, and calling any other desired functions.

Step 3: Start Code Composer Studio IDE.

Step 4: Load your target application onto the TI processor.

Step 5: Enable RTDX: Tools→RTDX→Configuration Control.

The Configuration Control window displays.

Step 6: Run your target application.

Step 7: Run your host client.

For details on using RTDX, see the Help→Contents→RTDX or Help→Tutorial→RTDX Tutorial.

5.5 Automation (for Debug)

5.5.1 Using the General Extension Language (GEL)

As mentioned earlier, GEL scripts can be used to create custom GEL menus and automate steps in Code Composer Studio. In section 4.7.1, we saw examples of using built-in GEL functions to automate steps related to project management. There are also many built-in GEL functions that can be used to automate steps during the debug process. Activities such as: set breakpoints, add variables to the Watch Window, begin execution, halt execution, and set up File I.O are just some of the actions that can be run from a GEL script.

5.5.2 Scripting Utility for Debug

The scripting utility as mentioned in section 4.7.2 also has comands that can greatly aid in automating many debug steps. You can also refer to the online help that comes with the Scripting Utility.

5.6 Target Reset

There will be times when it is necessary to perform a reset of the target or the emulator. Commands to perform these resets are integrated in the Code Composer Studio IDE. The availability to perform these reset commands are dependent on whether the IDE is connected to the target or not. Please refer to section 3.1.3 (Connect/Disconnect) for more information on connecting or disconnecting the target.

5.6.1 Reset Target

Target reset initializes the contents of all registers to their power-up state, and halts execution of the program. If the target board does not respond to this command and you are using a kernel-based device driver, the CPU kernel may be corrupt. In this case, you must reload the kernel.

The simulator initializes the contents of all registers to their power−up state, according to target simulation specifications.

To reset the target processor, select Debug→Reset CPU.

Note: Connection must be established with the target for the Debug→Reset CPU option to be available.

5.6.2 Emulator Reset

Some processors require putting the processor into its functional run state before a hard reset will work. In this case, the only way to force the processor back into this functional run state is to reset the emulator. An emulator reset will pull the TRST pin active, forcing the device to the functional run mode.

The Reset Emulator option becomes enabled whenever Code Composer Studio is disconnected from the target. To reset the emulator, choose Debug Menu→Reset Emulator. Upon running Reset Emulator, the hardware is left in a free running state and you can now manually reset the target hardware by pressing the reset button or by selecting Debug→Reset CPU. Note that this is not true on ARM.

Chapter 6

Analyze/Tune

Providing value to customers is often the main objective for a DSP developer. A valuable customer experience can stem from an efficient application. To create such an application, a developer may focus on performance, power, code size, or cost. Managing tradeoffs between these factors is a vital part of a DSP developer's role.

Application Code Analysis is the process of gathering and interpreting data about the factors that influence an application's efficiency. Application Code Tuning is the modification of code to improve its efficiency. DSP developers can analyze and tune their application as often as necessary to meet the efficiency goals defined by their customers, application, and hardware.

Code Composer Studio provides various tools to help developers analyze and tune their applications.

Topic **Page**

6.1 Appliction Code Analysis

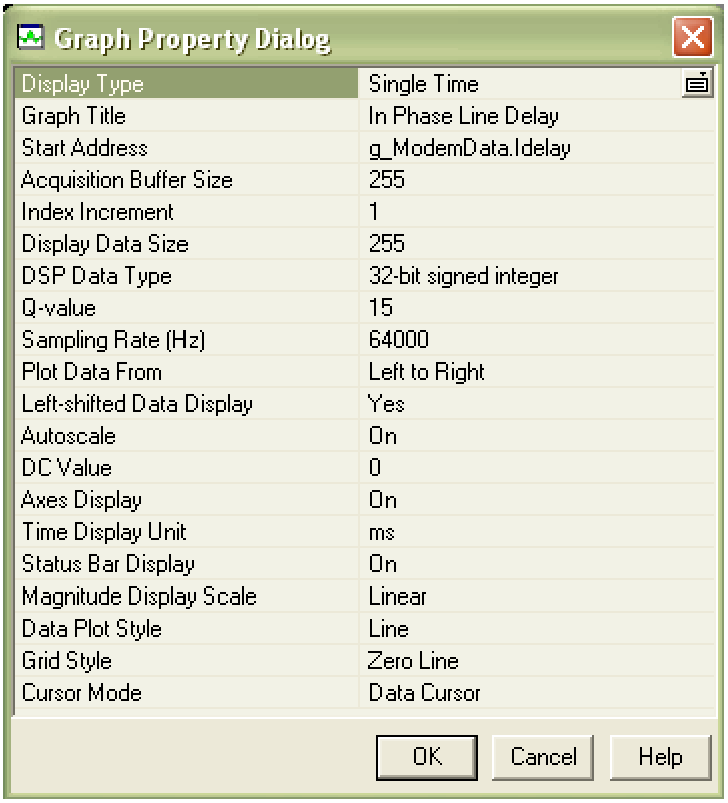
An analysis of application code can reveal many opportunities to improve efficiency, especially when knowing where to look. The tools offered by Code Composer Studio have been designed to gather important data and usefully present it to aid the DSP developer in the tuning process.

6.1.1 Data Visualization

Data visualization is useful when developing applications for communications, wireless, image processing, as well as general applications.

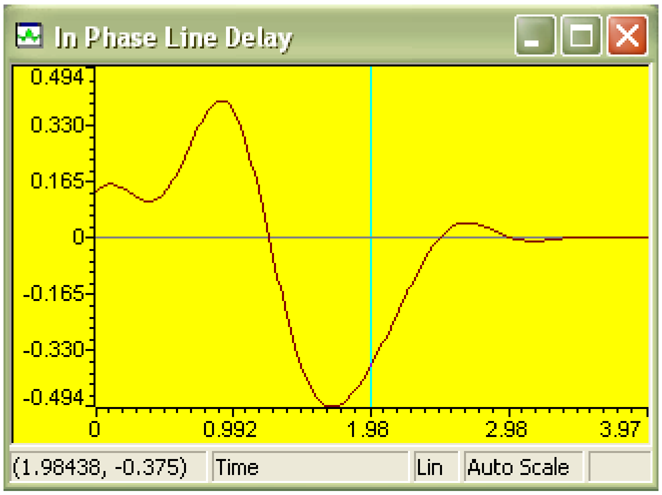
There are a variety of ways in Code Composer Studio to graph data processed by your program. The following classes of graphs are available: time/frequency, constellation diagram, eye diagram, and image.

All the graphs can be accessed by going to View→Graph and selecting the desired graph and specifying the graph properties in the graph properties box that appears. In this example below, a Single Time (Time/Frequency) graph is being configured:



Display Type	Single Time
Graph Title	In Phase Line Delay
Start Address	g_ModemData.Idelay
Acquisition Buffer Size	255
Index Increment	1
Display Data Size	255
DSP Data Type	32-bit signed integer
Q-value	15
Sampling Rate (Hz)	64000
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	ms
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

Once the properties are configured, hitting the OK button will open a graph window and plot the data specified in the graph properties.



Please refer to the online help under Graphing Windows for more detailed information on this topic.

6.1.2 Simulator Analysis

The Simulator Analysis tool reports the occurrence of particular system events so you can accurately monitor and measure the performance of your program.

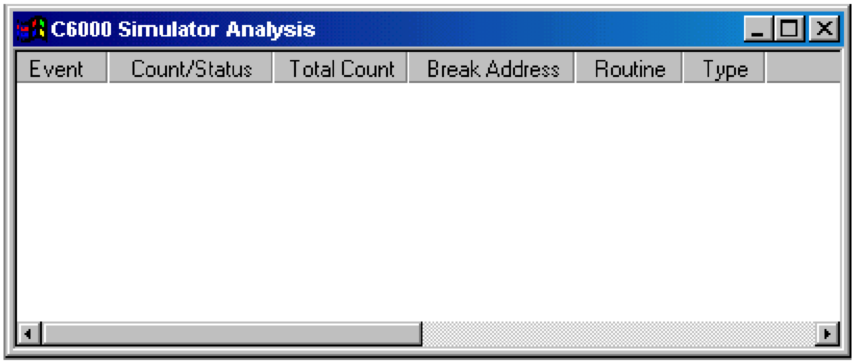
User Options:

- Enable/disable analysis
- Count the occurrence of selected events
- Halt execution whenever a selected event occurs
- Delete count or break events
- Create a log file
- Reset event counter

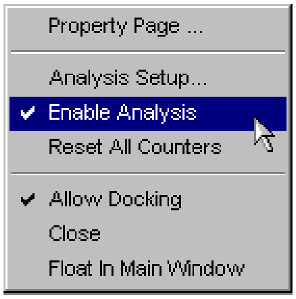
To use the Simulator Analysis tool:

Step 1: Load your program.

Step 2: Start the analysis tool. Select Tools→Simulator Analysis for your device.



Step 3: Right-click in the Simulator Analysis window and then select Enable analysis.



Step 4: Specify your analysis parameters (count events or break events).

Step 5: Run or step through your program.

Step 6: Analyze the output of the analysis tool.

For detailed information on the Simulator Analysis tool, see the Simulator Analysis topics provided in the online help: Help→Contents→Simulator Analysis.

6.1.3 Emulator Analysis

The Emulator Analysis tool allows you to set up, monitor, and count events and hardware breakpoints.

To start the Emulator Analysis tool:

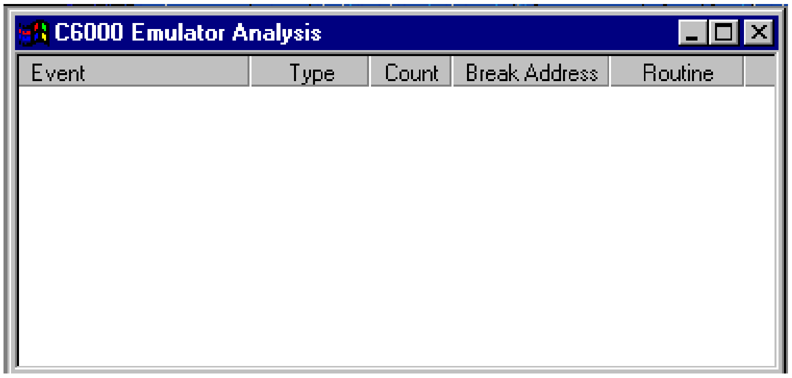
Step 1: Load your program.

Step 2: Select Tools→Emulator Analysis for your device from the menu bar.

The Emulator Analysis window (Figure 6−1) contains the following information:

This column. . .	displays. . .
Event	the event name.
Type	whether the event is a break or count event.
Count	the number of times the event occurred before the program halted.
Break Address	the address at which the break event occurred.
Routine	the routine in which the break event occurred.

Figure 6−1. Emulator Analysis Window



Note:

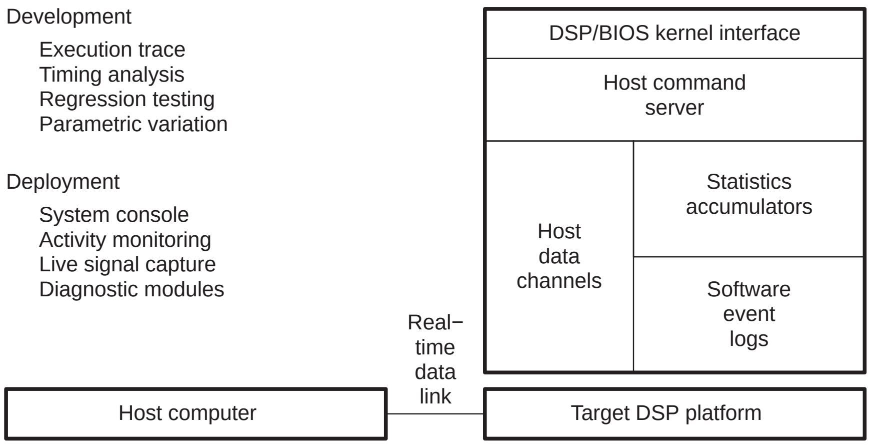
You cannot use the analysis features while you are using the profiling clock.

For detailed information on the Emulator Analysis tool, see the Emulator Analysis topics provided in the online help: Help→Contents→Emulator Analysis.

6.1.4 BIOS Real-Time Analysis (RTA) Tools

The DSP/BIOS Real-Time Analysis (RTA) features, shown in Figure 6−2, provide developers and integrators unique visibility into their application by allowing them to probe, trace, and monitor a DSP application during its course of execution. These utilities, in fact, piggyback upon the same physical JTAG connection already employed by the debugger, and utilize this connection as a low-speed (albeit real-time) communication link between the target and host.

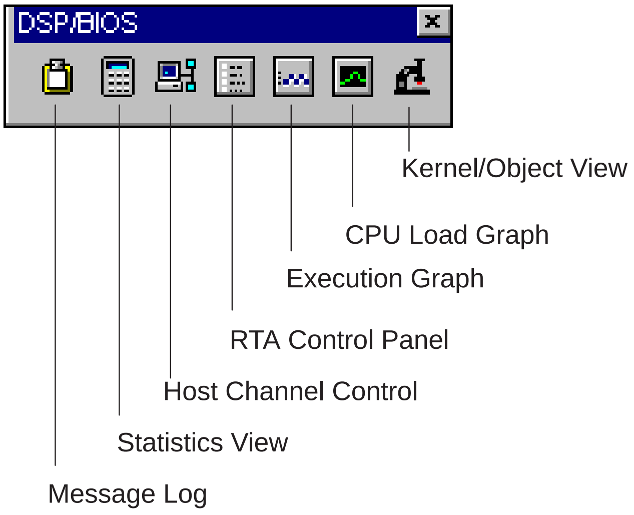
Figure 6−2. Real-Time Capture and Analysis



DSP/BIOS RTA requires the presence of the DSP/BIOS kernel within the target system. In addition to providing run-time services to the application, DSP/BIOS kernel provides support for real-time communication with the host through the physical link. By simply structuring an application around the DSP/BIOS APIs and statically created objects that furnish basic multitasking and I/O support, developers automatically instrument the target for capturing and uploading the real-time information that drives the visual analysis tools inside CCStudio IDE. Supplementary APIs and objects allow explicit information capture under target program control as well. From the perspective of its hosted utilities, DSP/BIOS affords several broad capabilities for real-time program analysis.

The DSP/BIOS Real-Time Analysis tools can be accessed through the DSP/BIOS toolbar.

Figure 6−3. DSP/BIOS Toolbar



❑ **Message Log:** Displays time-ordered sequences of events written to kernel log objects by independent real-time threads. This is useful for tracing the overall flow of control in the program. There are two ways in which the target program logs events:

1) Explicitly, through DSP/BIOS API calls. For example, this can be done through “LOG_printf(&trace, “hello world!”);”, where “trace” is the name of the log object.

2) Implicitly, by the underlying kernel when threads become ready, dispatched, and terminated. An example of this would be log events in the “Execution Graph Details.”

You can output the log to a file by right-clicking in the Message Log window and selecting Property Page.

❑ **Statistics View:** Displays summary statistics amassed in kernel accumulator objects, reflecting dynamic program elements ranging from simple counters and time-varying data values, to elapsed processing intervals of independent threads. The target program accumulates statistics explicitly through DSP/BIOS API calls or implicitly by the kernel, when scheduling threads for execution or performing I/O operations. You can change settings such as units for the statistics by right-clicking in the Statistics View window and selecting Property Page.

- **Host Channel Control:** Displays host channels defined by your program. You can use this window to bind files to these channels, start the data transfer over a channel, and monitor the amount of data transferred. Binding kernel I/O objects to host files provides the target program with standard data streams for deterministic testing of algorithms. Other real−time target data streams managed with kernel I/O objects can be tapped and captured on-the-fly to host files for subsequent analysis.

- **RTA Control Panel:** Controls the real-time trace and statistics accumulation in target programs. In effect, this allows developers to control the degree of visibility into the real-time program execution. By default, all types of tracing are enabled. You must check the “Global host enable” check box in order for any of the tracing types to be enabled. Your program can also change the settings in this window. The RTA Control Panel checks for any programmatic changes at the rate set for the RTA Control Panel in the Property Page. In the Property Page, you can also change refresh rates for any RTA tool, such as the Execution Graph.

- **Execution Graph:** Displays the execution of threads in real−time. Through the execution graph you can see the timing and the order in which threads are executed. Thick blue lines indicate the thread that is currently running, that is, the thread using the CPU. More information about different lines in the graph can be accessed by right-clicking in the Execution Graph window and selecting Legend. If you display the Execution Graph Details in a Message Log window, you can double-click on a box (a segment of a line) in the Execution Graph to see details about that event in text form. You can also hide threads in the graph by right-clicking in the Execution Graph window and selecting Property Page.

- **CPU Load Graph:** Displays a graph of the target CPU processing load. The most recent CPU load is shown in the lower−left corner and the highest CPU load reached so far is shown in the lower−right corner. The CPU load is defined as the amount of time not spent performing the low-priority task that runs when no other thread needs to run. Thus, the CPU load includes any time required to transfer data from the target to the host and to perform additional background tasks. The CPU load is averaged over the polling rate period. The longer the polling period, the more likely it is that short spikes in the CPU load is not shown in the graph. To set the polling rate, open the RTA Control Panel window and right-click in the window. Select Property Page, and in the Host Refresh Rates tab, set the polling rate with the Statistics View / CPU Load Graph slider and click OK.

- ❑ **Kernel/Object View:** Displays the configuration, state, and status of the DSP/BIOS objects currently running on the target. This tool shows both the dynamic and statically configured objects that exist on the target. You can right-click in the window and select Save Server Data to save the current data.

Note:

When used in tandem with the Code Composer Studio IDE standard debugger during software development, the DSP/BIOS real-time analysis tools provide critical visibility into target program behavior at exactly those intervals where the debugger offers little or no insight − during program execution. Even after the debugger halts the program and assumes control of the target, information already captured through DSP/BIOS can provide invaluable insights into the sequence of events that led up to the current point of execution.

Later in the software development cycle, regular debuggers become ineffective for attacking more subtle problems arising from time-dependent interaction of program components. The DSP/BIOS real-time analysis tools subsume an expanded role as the software counterpart of the hardware logic analyzer.

This dimension of DSP/BIOS becomes even more pronounced after software development concludes. The embedded DSP/BIOS kernel and its companion host analysis tools combine to form the necessary foundation for a new generation of manufacturing test and field diagnostic tools. These tools will be capable of interacting with application programs in operative production systems through the existing JTAG infrastructure.

The overhead cost of using DSP/BIOS is minimal, therefore instrumentation can be left in to enable field diagnostics, so that developers can capture and analyze the actual data that caused the failures.

6.1.5 Code Coverage and Multi-Event Profiler Tool

The Code Coverage and Multi-event Profiler tool provides two distinct capabilities:

- ❑ Code coverage provides visualization of source line coverage to facilitate developers in constructing tests to ensure adequate coverage of their code.
- ❑ Multi-event profiling provides function profile data collected over multiple events of interest – all in a single simulation run of the application. Events

include CPU cycles, instructions executed, pipeline stalls, cache hits, misses and so on. This tool helps identify hotspots, and possible factors affecting performance.

See the *Code Coverage and Multi-event Profiler User's Guide* (SPRU624) for further details.

6.2 Application Code Tuning

The tuning process begins where the analysis stage ends. When application code analysis is complete, the DSP developer should have identified inefficient code. The tuning process consists of determining whether inefficient code can be improved, setting efficiency objectives, and attempting to meet those goals by modifying code. Code Composer Studio links several tools together into an organized method of tuning as well as providing a single point for analyzing tuning progress. The Tuning tool suite is a cohesive set of tuning tools, logically formed to attack the key areas in which DSP developers need high efficiency.

6.2.1 Optimization Dashboard

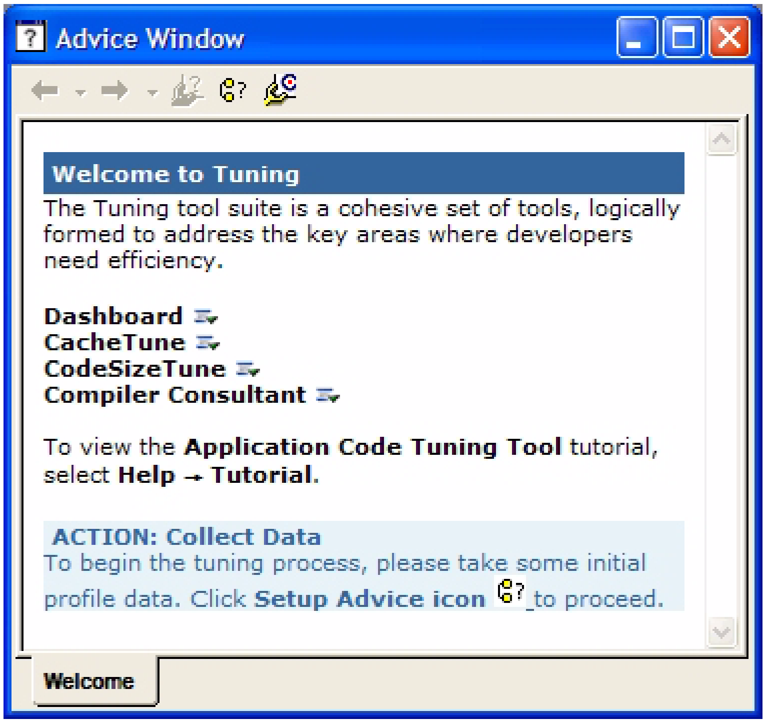
The Dashboard is a central focal point for the tuning process. It displays build− and run-time profile data suggestions for which tuning tool to use, and it can launch each of the tools. The Dashboard is the main interface during the tuning phase of the development cycle.

6.2.1.1 Advice Window

The Advice window is an area of the Dashboard that displays tuning information. It guides the user through the tuning process, explaining and detailing the appropriate steps to take, and displaying helpful information for using tools, links to other tools and documentation, and important messages.

The Advice window should be consulted when first using a tool, or to determine the appropriate action to take at any point in the tuning procedure.

Code Composer Studio initially starts in Debug layout. To open the Advice window, switch to Tuning layout by clicking the tuning fork icon on the toolbar. Alternatively, the user can choose Advice from the Profile→Tuning menu item. The Advice window will open at the left of the Code Composer Studio screen and display the following Welcome information:



Welcome to Tuning

The Tuning tool suite is a cohesive set of tools, logically formed to address the key areas where developers need efficiency.

Dashboard
CacheTune
CodeSizeTune
Compiler Consultant

To view the **Application Code Tuning Tool** tutorial, select **Help → Tutorial.**

ACTION: Collect Data
To begin the tuning process, please take some initial profile data. Click **Setup Advice icon** to proceed.

At the top of the Advice Window there is a toolbar that provides buttons for Internet−style navigation of advice pages as well as buttons for opening the main advice pages. Click on the arrows in the toolbar to navigate back and forth through the history of advice pages for that tab. Below the pane that contains the advice pages, there are one or more tabs. These tabs let several advice pages remain open, allowing DSP developers carry out more than one task at once. Click on the tabs at the bottom of the Advice Window to switch between open pages. To close the active tab, right-click on the Advice Window and choose Close Active Tab from the pop−up menu.

The Welcome advice page contains links to descriptions of each of the major tuning tools. At the bottom of the Welcome advice page, there is a blue box containing suggestions for the next step in the tuning process. These blue Action boxes are found throughout the Advice Window pages.

When navigating through the Advice Window pages, red warning messages may appear. These messages often serve as helpful reminders while tuning but may contain valuable solutions if problems are encountered along the way.

When you have launched a tool, such as CodeSizeTune, an advice tab for that tool appears at the bottom of the Advice window. The page contains information about the tool, including its function, when it should be used, and how to apply the tool effectively for optimum tuning efficiency.

If the user follows the information presented on each page, the Advice Window can be a useful handbook when getting started with Application Code Tuning.

6.2.1.2 Profile Setup

As the Advice Window indicates, Code Composer Studio must know what code elements the DSP developer is interested in tuning before tuning begins. This can be accomplished using Profile Setup. The Profile Setup window should be used at the beginning of the Tuning process to specify the data to be collected and the requisite sections of code.

The Profile Setup window can be opened using the Advice Window. In the blue Action box at the bottom of the Welcome page, click the link to open the Setup Advice page. The first Action box will contain a link to open the Profile Setup window. Profile Setup can also be launched from the main menu topic Profile→Setup.

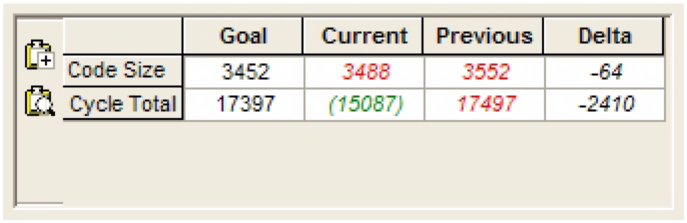
To configure the collected data, use the Activities tab of the Profile Setup window. Check off each desired activity corresponding to the element of the application that is being tuned. For a more detailed explanation of each activity, click on the activity and view its description in the window below.

The Ranges tab is used to tune specific sections of code. Functions, loops, and arbitrary segments of code can be added to the Ranges tab for collection of tuning information. Code Composer Studio must be notified when to stop data collection by using the Control tab to add exit points to the code. When an exit point is reached, Code Composer Studio will stop collecting tuning information. The Control can also be used to add Halt and Resume Collection points to the code in order to isolate different sections of code. The Custom tab can be used to collect custom data, such as cache hits or CPU idle cycles.

6.2.1.3 Goals Window

Tuning an application is about setting and reaching efficiency goals, so Code Composer Studio provides a method of recording numerical goals and tracking progress towards reaching them.

The Goals Window displays a summary of application data, including values for the size of the code and number of cycles, that can be updated each time the code is run. It also compares this data to the data for the last run, as well as the goals that the DSP developer has defined.



	Goal	Current	Previous	Delta
Code Size	3452	3488	3552	-64
Cycle Total	17397	(15087)	17497	-2410

1) To open the Goals Window, first open the Advice Window.
2) On the Advice Window toolbar, click on the View General Tuning Advice icon at the top of the window to open the General tab.
3) In the blue Action box, click on the Launch the Goals Window icon. The Goals Window will open on the left side of the screen.

If an application has been loaded and profiling has been set up, the Goals Window can be populated with data simply by running the application. To record objectives for tuning, click in the Goals column, type in the goal and press Enter. If a goal has been reached, the data will be displayed in green and in parentheses. Otherwise, the data will appear in red. When the application is restarted and run, the Current values in the Goals Window will move to the Previous column and the difference will be displayed in the Delta column. The Goals Window also allows a DSP developer to save or view the contents in a log at any time, by using the logging icons at the left side of the window.

6.2.1.4 Profile Viewer

The Profile Viewer displays collected data during the Application Code Tuning process. It consists of a spreadsheet-like grid with each row corresponding to the elements of code selected in the Ranges tab of the Profile Setup window. The columns in the grid store the collected data for each profiled section of code, as selected in the Activities and Custom tabs of the Profile Setup window.

The Profile Viewer provides a single location for the display of all collected information during the tuning process. Ranges can be sorted by different data values. Data sets displayed in the Profile Viewer can be saved, restored, and compared with other data sets.

The Profile Viewer can be used to pinpoint sections of code that require the most tuning. For instance, to determine which function results in the most cache stalls, the cache stall data in the Profile Viewer can be sorted from largest to smallest. Sections of the function can then be profiled to determine exactly what code is generating cache stalls.

To open the Profile Viewer, navigate to the Setup tab in the Advice Window. At the bottom of the Setup Advice page, click on the Profile Data Viewer link to display the Profile Viewer in the lower portion of the screen. Alternatively, Profile Viewer can be launched from the main menu topic Profile→Viewer. If tuning has been set up using the Profile Setup window, running the application will display data in the Profile Viewer. The view can be customized by dragging and dropping rows and columns. The data can be saved and restored using the Profile Viewer buttons, and it can be sorted by double-clicking on the title of a column. In addition, several Profile Viewers can be opened simultaneously.

6.2.2 Compiler Consultant

The Compiler Consultant Tool analyzes your application and makes recommendations for changes to optimize the compiler cycle time. The tool displays two types of information: Compile Time Loop Information and Run Time Loop Information. Compile Time Loop Information is created by the compiler. Run Time Loop Information is data gathered by profiling your application. Each time you compile or build your code, Consultant will analyze the code and create suggestions for different optimization techniques to improve code efficiency. You then have the option of implementing the advice and building the project again.

When you analyze Compiler Consultant information, sort information by:

Estimated Cycles Per Iteration if you want to view Compile Time Loop Information.

cycle.CPU:Excl.Total if you are analyzing Run Time Loop Information.

This sorting will bring the rows to the top of the Profile Viewer that consume the most CPU cycles and which should gain the most performance benefit by tuning.

You can then work on tuning one loop at a time. Double−clicking on the Advice Types entry for any loop row will bring up the full advice for that loop in the Consultant tab of the Advice window.

After you have applied the advice to fix individual loops, it is useful to hide that row in the Profile Viewer window. Hiding rows reduces the amount of information present in the Profile Viewer window. Rows can always be unhidden.

To find out more about using the Compiler Consultant, see Consultant in the online help under Application Code Tuning.

6.2.3 CodeSizeTune (CST)

CodeSizeTune (CST) is a tool that enables you to easily optimize the trade-off between code size and cycle count for your application. Using a variety of profiling configurations, CodeSizeTune will profile your application, collect data on individual functions, and determine the best combinations of compiler options. CST will then produce a graph of these function-specific option sets, allowing you to graphically choose the configuration that best fits your needs.

Previous users of Code Composer Studio will recognize CST as a replacement for the Profile-Based Compiler (PBC).

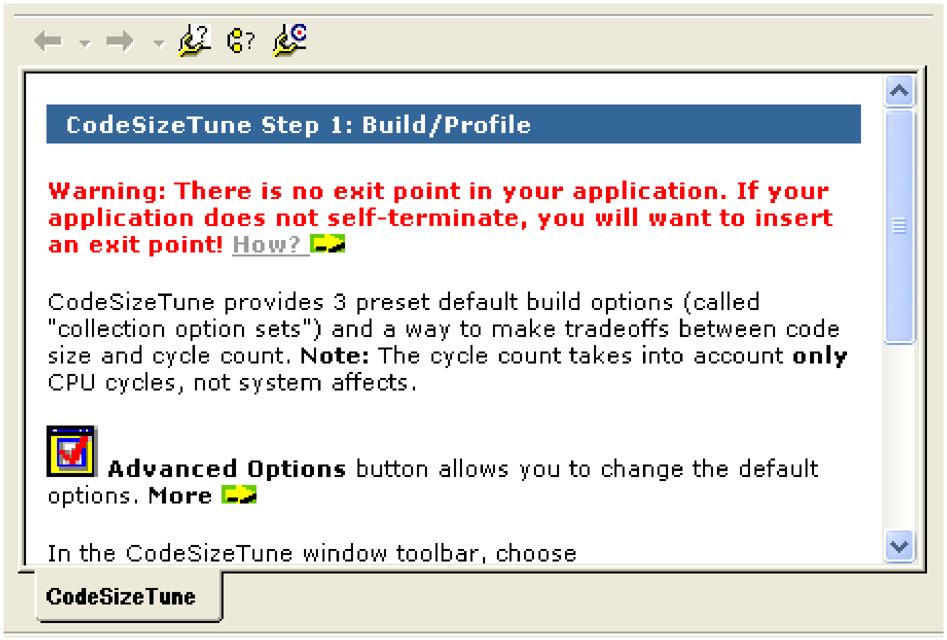
1) How do I begin? CST starts with your debugged application. Check out CodeSizeTune in the online help under Application Code Tuning to make sure your application meets the criteria for CST profiling.

2) CST will use several profile collection options sets to build and profile your application. Using the profile information it gains on how each function performed under several different profile collection options, CST pieces together collection options that contain compiler options at the function level. To find out more about how to build and profile, see Build & Profile under CodeSizeTune in the online help.

3) The best profile collection options sets are then plotted on a two−dimensional graph of code size vs. performance, which allows you to graphically select the optimum combination of size and speed to meet your system needs. For more on selecting a desired collection options set, see the topic Select Desired Speed and Code Size under CodeSizeTune in the online help.

4) Finally, you will save your selected collection options set to the Code Composer Studio project. See Save Settings and Close under CodeSizeTune in the online help.

The Advice window guide you though the steps of the process. When you launch CodeSizeTune, the CodeSizeTune tab of the advice window will be displayed automatically. See CodeSizeTune Advice Window in the online help for more information.



CodeSizeTune Step 1: Build/Profile

Warning: There is no exit point in your application. If your application does not self-terminate, you will want to insert an exit point! How?

CodeSizeTune provides 3 preset default build options (called "collection option sets") and a way to make tradeoffs between code size and cycle count. **Note:** The cycle count takes into account **only** CPU cycles, not system affects.

Advanced Options button allows you to change the default options. **More**

In the CodeSizeTune window toolbar, choose

6.2.4 Cache Tune

The Cache Tune tool provides a graphical visualization of cache accesses over a set amount of time. This tool is highly effective at highlighting non-optimal cache usage (due to factors such as conflicting code placement, inefficient data access patterns, etc.). All the memory accesses are color−coded by type. Various filters, panning, and zoom features facilitate quick drill−down to view specific areas. This visual/temporal view of cache accesses enables quick identification of problem areas, such as areas related to conflict, capacity, or compulsory misses. Using this tool, developers can significantly optimize cache efficiency, thereby reducing the cycles consumed in the memory subsystem. All of these features help the user to greatly improve the cache efficiency of the overall application.

The Tuning→CacheTune menu item launches the Cache Tune tool showing the latest cache traces. There are three kinds of cache trace files:

❑ Program Cache trace

❑ Data Cache trace

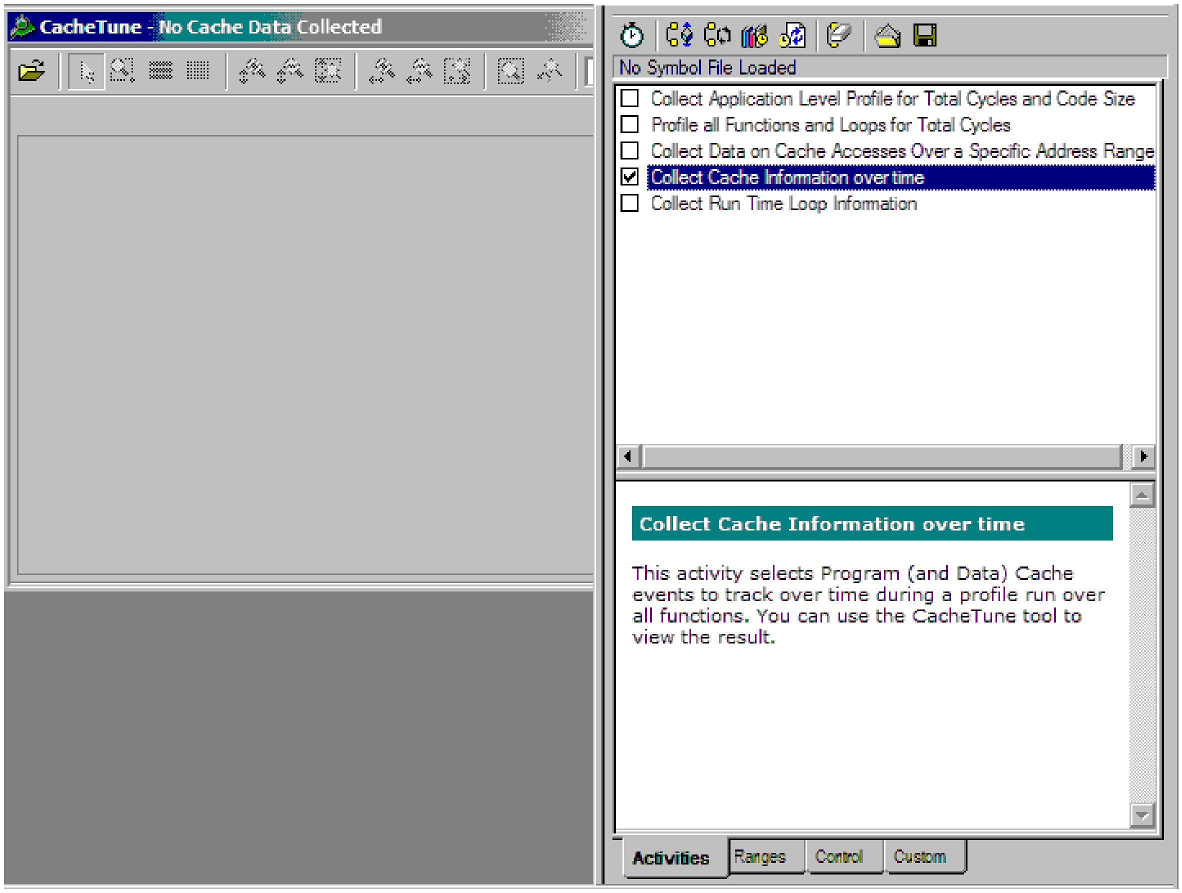
❑ Cross Cache trace

The data cache trace tab is displayed by default. If no cache traces have been collected, then the graph is empty.

Once the tool is launched, you can view other cache data files by opening a saved dataset.

Datasets can be opened by clicking the Open dataset button, pressing its hotkey, or clicking the Load dataset item in the pop-up menu.

See the *Cache Analysis User’s Guide* (SPRU575) accessed from the TI website for further information on the Cache Analysis tool.



- [ ] Collect Application Level Profile for Total Cycles and Code Size
- [ ] Profile all Functions and Loops for Total Cycles
- [ ] Collect Data on Cache Accesses Over a Specific Address Range
- [✓] Collect Cache Information over time
- [ ] Collect Run Time Loop Information

Collect Cache Information over time

This activity selects Program (and Data) Cache events to track over time during a profile run over all functions. You can use the CacheTune tool to view the result.

Chapter 7

Additional Tools, Help, and Tips

This chapter gives information on how to find additional help for documentation, updates, and with customizing your Code Composer Studio installaton.

Topic **Page**

7.1 Component Manager

Note:

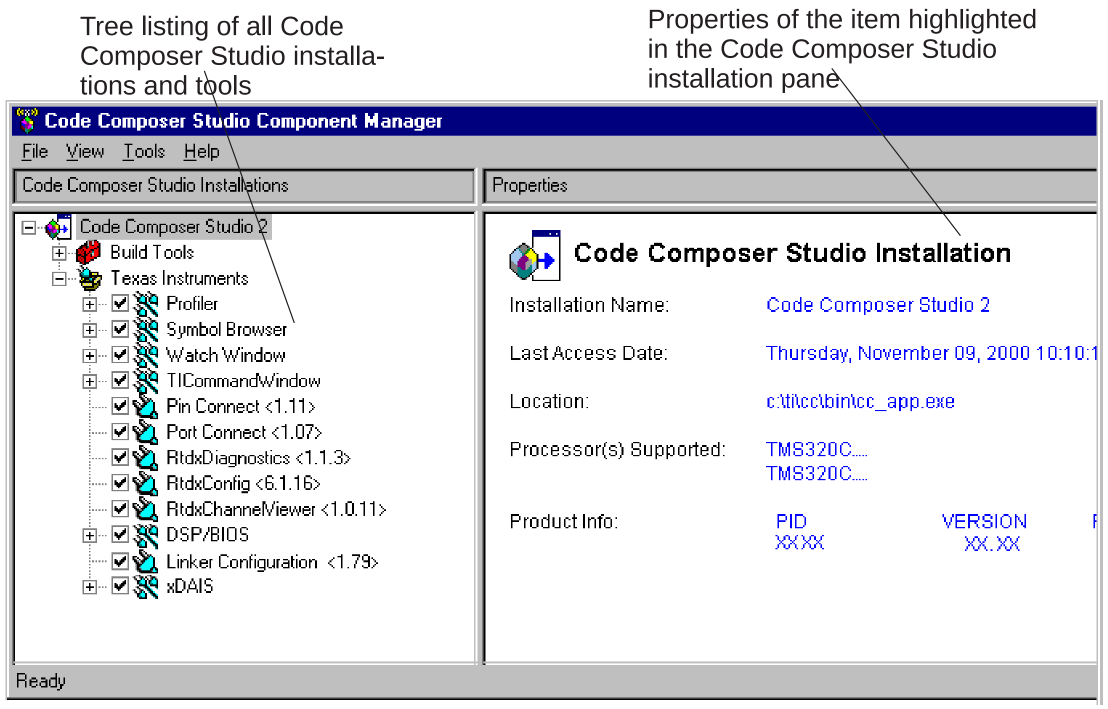
The Component Manager is an advanced tool used primarily to customize or modify your installation. Use this tool only to resolve component interaction in a custom or multiple installation environment.

Multiple installations of the Code Composer Studio IDE can share installed tools. The Component Manager provides an interface for handling multiple versions of tools with these multiple installations.

The Component Manager window displays a listing of all installations, build tools, Texas Instruments plug-in tools, and third-party plug-in tools. When a node is selected in the tree (the left pane of the Component Manager), its properties are displayed in the Properties pane (the right pane) (see Figure 7−1).

With the Component Manager, you can enable or disable tools for a particular Code Composer Studio installation. This functionality allows you to create a custom combination of tools contained within the IDE. The Component Manager also allows you to access the Update Advisor to download the most recent version of the tools from the web.

Figure 7−1. Component Manager


Tree listing of all Code Composer Studio installations and tools
Properties of the item highlighted in the Code Composer Studio installation pane
Code Composer Studio Component Manager
File View Tools Help
Code Composer Studio Installations
Properties
Ready

- Code Composer Studio 2
  - Build Tools
  - Texas Instruments
    - ✓ Profiler
    - ✓ Symbol Browser
    - ✓ Watch Window
    - ✓ TICommandWindow
    - ✓ Pin Connect <1.11>
    - ✓ Port Connect <1.07>
    - ✓ RtdxDiagnostics <1.1.3>
    - ✓ RtdxConfig <6.1.16>
    - ✓ RtdxChannelViewer <1.0.11>
    - ✓ DSP/BIOS
    - ✓ Linker Configuration <1.79>
    - ✓ xDAIS

Code Composer Studio Installation

Installation Name:	Code Composer Studio 2
Last Access Date:	Thursday, November 09, 2000 10:10:1
Location:	c:\ti\cc\bin\cc_app.exe
Processor(s) Supported:	TMS320C.... TMS320C....
Product Info:	PID XX.XX VERSION XX.XX

7.1.1 Opening Component Manager

To open the Component Manager:

Step 1: From the Help menu in the Code Composer Studio IDE, select About.

The About Code Composer Studio dialog box appears.

Step 2: In the About dialog box, click the Component Manager button.

The Component Manager window displays.

7.1.2 Multiple Versions of the Code Composer Studio IDE

The following is a list of requirements for maintaining multiple versions of the Code Composer Studio IDE and related tools:

- ❑ To keep more than one version of the Code Composer Studio IDE or a related tool, you must install each version in a different directory.
- ❑ If you install an additional version of the Code Composer Studio IDE, or an additional version of a tool, in the same directory as its previous installation, the original installation will be overwritten.
- ❑ You cannot enable multiple versions of the same tool within one installation.

7.2 Update Advisor

The Update Advisor allows you to download updated versions of the Code Composer Studio IDE and related tools. The Update Advisor accesses the Available Updates web site. This site displays a list of patches, drivers, and tools available for downloading.

To use the Update Advisor, you must have Internet access and a browser installed on your machine. See the Code Composer Studio IDE Quick Start for complete system requirements.

Note:

You must be registered with TI&ME before you can access the Available Updates web site.

7.2.1 Registering Update Advisor

If you did not register your product during installation, you can access the online registration form from the Code Composer Studio help menu: Help→CCS on the Web→Register.

Important! The first time you use Update Advisor, your browser may display the TI&ME web page. To register, follow the directions displayed on the page.

You must register online and have a valid subscription plan in place to receive downloads through Update Advisor. You receive a 90 day free subscription service with the Code Composer Studio product. At the end of this period, you must purchase an annual subscription service. Annual subscriptions are only available for the full product.

7.2.2 Checking for Tool Updates

In the Code Composer Studio IDE, select Help→Check for Updates→Update Advisor.

If you are already registered with TI&ME, and have accepted the cookie necessary for automatic log-in, your browser will go directly to the Available Updates web site.

To query the Available Updates web site, the Update Advisor passes certain information from your machine:

- ❑ Code Composer Studio product registration number
- ❑ Code Composer Studio installation version
- ❑ a text description of the installed product
- ❑ the list of installed plug-ins

The Available Updates web site will then list any updates appropriate for your Code Composer Studio installation.

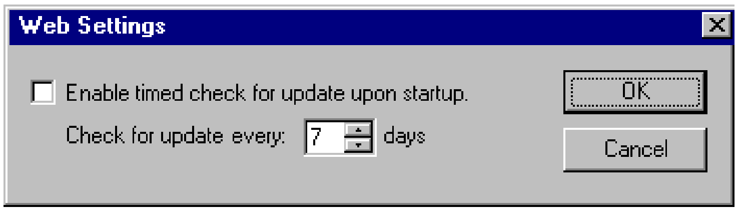
You have the opportunity to just download the updates, or to download and install them immediately.

You can also configure the Update Advisor to automatically check for updates.

7.2.3 Automatically Checking for Tool Updates

You may check for tool updates at any time, or you can configure the Update Advisor to automatically check for updates.

Step 1: Select Help→Update Advisor→Setting. The Web Settings dialog box appears:



Enable timed check for update upon startup.

Check for update every: 7 days

Step 2: In the Check for Update field, specify how often the Update Advisor should check the Available Updates web site.

Step 3: To enable the automatic update feature, click the checkbox to the left of the “Enable timed check for update upon startup” field.

When this field is enabled, the Update Advisor automatically checks for web updates according to the schedule specified in step 2.

Step 4: Click OK to save your changes and close the dialog box.

7.2.4 Uninstalling the Updates

Any installed update can be uninstalled to restore the previous version of the Code Composer Studio IDE.

Note that only the previous version of a tool can be restored. If you install one update for a tool, and then install a second update for the same tool, the first update can be restored. The original version of the tool cannot be restored, even if you uninstall both the second update and the first update.

7.3 Additional Help

A multitude of help tools are available to answer your questions. You can access Help→Contents to guide you through certain topics step by step; traverse online help sites that provide the most current help topics; or, peruse user manuals which are pdf files that provide information on specific features or processes.

Also, you can access the Update Advisor to get the newest features through Help→Update Advisor.

7.3.1 Code Composer Studio Online Help

The Online Help provides links to the tutorials, multimedia demos, user manuals, application reports, and a website (www.dspvillage.com) where you can obtain information regarding the software. Simply click on Help and follow the links provided.

Index