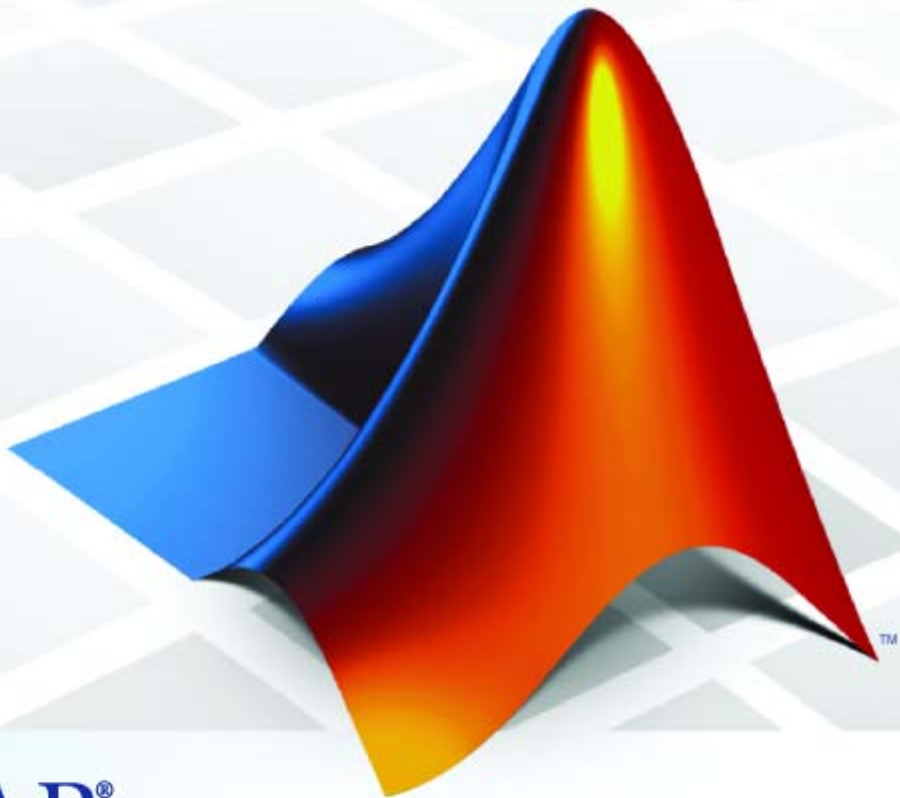


Embedded IDE Link™ 4

User's Guide

For Use with Texas Instruments' Code Composer Studio™



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Embedded IDE Link™ User's Guide

© COPYRIGHT 2002–2010 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

July 2002	Online only	New for Version 1.0 (Release 13)
October 2002	Online only	Revised for Version 1.1
May 2003	Online only	Revised for Version 1.2
September 2003	Online only	Revised for Version 1.3 (Release 13SP1+)
June 2004	Online only	Revised for Version 1.3.1 (Release 14)
October 2004	Online only	Revised for Version 1.3.2 (Release 14SP1)
December 2004	Online only	Revised for Version 1.4 (Release 14SP1+)
March 2005	Online only	Revised for Version 1.4.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.4.2 (Release 14SP3)
March 2006	Online only	Revised for Version 1.5 (Release 2006a)
April 2006	Online only	Revised for Version 2.0 (Release 2006a+)
September 2006	Online only	Revised for Version 2.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.0 (Release 2007a)
September 2007	Online only	Revised for Version 3.1 (Release 2007b)
March 2008	Online only	Revised for Version 3.2 (Release 2008a)
October 2008	Online only	Revised for Version 3.3 (Release 2008b)
March 2009	Online only	Revised for Version 3.4 (Release 2009a)
September 2009	Online only	Revised for Version 4.0 (Release 2009b)
March 2010	Online only	Revised for Version 4.1 (Release 2010a)

Getting Started

1

Product Overview	1-2
Automation Interface	1-3
Project Generator	1-4
Verification	1-5
Product Features Supported for Each Processor Family ..	1-5
Configuration Information	1-6
Verifying Your Code Composer Studio Installation	1-6
Software Requirements	1-8

Automation Interface

2

Getting Started with Automation Interface	2-2
Introducing the Automation Interface Tutorial	2-2
Selecting Your Processor	2-6
Creating and Querying Objects for CCS IDE	2-8
Loading Files into CCS	2-10
Working with Projects and Data	2-12
Closing the Links or Cleaning Up CCS IDE	2-18
Getting Started with RTDX	2-20
Introducing the Tutorial for Using RTDX	2-21
Creating the tics Objects	2-26
Configuring Communications Channels	2-29
Running the Application	2-31
Closing the Connections and Channels or Cleaning Up ...	2-38
Listing Functions	2-41

Constructing tics Objects	2-42
Example — Constructor for tics Objects	2-42
tics Properties and Property Values	2-44
Overloaded Functions for tics Objects	2-45
tics Object Properties	2-46
Quick Reference to tics Object Properties	2-46
Details About tics Object Properties	2-48
Managing Custom Data Types with the Data Type	
Manager	2-54
Adding Custom Type Definitions to MATLAB	2-56

Project Generator

3

Introducing Project Generator	3-2
Project Generation and Board Selection	3-3
Project Generator Tutorial	3-5
Creating the Model	3-6
Adding the Target Preferences Block to Your Model	3-6
Specify Configuration Parameters for Your Model	3-10
Model Reference	3-14
How Model Reference Works	3-14
Using Model Reference	3-15
Configuring processors to Use Model Reference	3-17

Exporting Filter Coefficients from FDATool

4

About FDATool	4-2
Preparing to Export Filter Coefficients to Code	
Composer Studio Projects	4-4
Features of a Filter	4-4
Selecting the Export Mode	4-5
Choosing the Export Data Type	4-6
Exporting Filter Coefficients to Your Code Composer	
Studio Project	4-9
Exporting Filter Coefficients from FDATool to the CCS IDE Editor	4-9
Reviewing ANSI C Header File Contents	4-12
Preventing Memory Corruption When You Export	
Coefficients to Processor Memory	4-15
Allocating Sufficient or Extra Memory for Filter Coefficients	4-15
Example: Using the Exported Header File to Allocate Extra Processor Memory	4-15
Replacing Existing Coefficients in Memory with Updated Coefficients	4-16
Example: Changing Filter Coefficients Stored on Your Processor	4-17

Block Reference

5

Block Library: idelinklib_ticcs	5-2
Block Library: idelinklib_common	5-3

6

Supported Processors

A

Supported Platforms	A-2
Product Features Supported by Each Processor or Family	A-2
Coemulation Support	A-3
Supported Processors and Simulators	A-3
Custom Board Support	A-4
 Supported Versions of Code Composer Studio	 A-5

Reported Limitations and Tips

B

Reported Issues	B-2
Demonstration Programs Do Not Run Properly Without Correct GEL Files	B-3
Error Accessing type Property of ticcs Object Having Size Greater Than 1	B-3
Changing Values of Local Variables Does Not Take Effect	B-4
Code Composer Studio Cannot Find a File After You Halt a Program	B-4
C54x XPC Register Can Be Modified Only Through the PC Register	B-6
Working with More Than One Installed Version of Code Composer Studio	B-6
Changing CCS Versions During a MATLAB Session	B-7
MATLAB Hangs When Code Composer Studio Cannot Find a Board	B-7
Using Mapped Drives	B-9

Uninstalling Code Composer Studio 3.3 Prevents Embedded IDE Link From Connecting	B-9
PostCodeGenCommand Commands Do Not Affect Embedded IDE Link Projects	B-10

Index

Getting Started

- “Product Overview” on page 1-2
- “Configuration Information” on page 1-6
- “Software Requirements” on page 1-8

Product Overview

In this section...
“Automation Interface” on page 1-3
“Project Generator” on page 1-4
“Verification” on page 1-5
“Product Features Supported for Each Processor Family” on page 1-5

Embedded IDE Link™ software enables you to use MATLAB® functions to communicate with Code Composer Studio™ software and with information stored in memory and registers on a processor. With the `ticcs` objects, you can transfer information to and from Code Composer Studio software and with the embedded objects you get information about data and functions stored in your signal processor memory and registers, as well as information about functions in your project.

Embedded IDE Link lets you build, test, and verify automatically generated code using MATLAB, Simulink®, Real-Time Workshop®, and the Code Composer Studio integrated development environment. Embedded IDE Link makes it easy to verify code executing within the Code Composer Studio software environment using a model in Simulink software. This processor-in-the-loop testing environment uses code automatically generated from Simulink models by Real-Time Workshop® Embedded Coder™ software. A wide range of Texas Instruments DSPs are supported:

- TI's C2000™
- TI's C5000™
- TI's C6000™

With Embedded IDE Link , you can use MATLAB software and Simulink software to interactively analyze, profile and debug processor-specific code execution behavior within CCS. In this way, Embedded IDE Link automates deployment of the complete embedded software application and makes it easy for you to assess possible differences between the model simulation and processor code execution results.

Embedded IDE Link consists of these components:

- Project Generator—add embedded framework code to the C code generated from Simulink models, and package as a complete IDE project
- Automation Interface—use functions in the MATLAB command window to access and manipulate data and files in the IDE and on the processor
- Verification—verify how your programs run on your processor

With Embedded IDE Link, you create objects that connect MATLAB software to Code Composer Studio software. For information about using objects, refer to “Software Requirements” on page 1-8.

Note Embedded IDE Link uses objects. You work with them the way you use all MATLAB objects. You can set and get their properties, and use their methods to change them or manipulate them and the IDE to which they refer.

The next sections describe briefly the components of Embedded IDE Link software.

Automation Interface

The automation interface component is a collection of methods that use the Code Composer Studio API to communicate between MATLAB software and Code Composer Studio. With the interface, you can do the following:

- Automate complex tasks in the development environment by writing MATLAB software scripts to communicate with the IDE, or debug and analyze interactively in a live MATLAB software session.
- Automate debugging by executing commands from the powerful Code Composer Studio software command language.
- Exchange data between MATLAB software and the processor running in Code Composer Studio software.
- Set breakpoints, step through code, set parameters and retrieve profiling reports.

- Automate project creation, including adding source files, include paths, and preprocessor defines.
- Configure batch building of projects.
- Debug projects and code.
- Execute API Library commands.

The automation interface provides an application program interface (API) between MATLAB software and Code Composer Studio. Using the API, you can create new projects, open projects, transfer data to and from memory on the processor, add files to projects, and debug your code.

Project Generator

The Project Generator component is a collection of methods that use the Code Composer Studio API to create projects in Code Composer Studio and generate code with Real-Time Workshop. With the interface, you can do the following:

- Automated project-based build process
Automatically create and build projects for code generated by Real-Time Workshop or Real-Time Workshop Embedded Coder.
- Customize code generation
Use Embedded IDE Link with any Real-Time Workshop system target file (STF) to generate processor-specific and optimized code.
- Customize the build process
- Automate code download and debugging
Rapidly and effortlessly debug generated code in the Code Composer Studio software debugger, using either the instruction set simulator or real hardware.
- Create and build CCS projects from Simulink software models. Project Generator uses Real-Time Workshop software or Real-Time Workshop Embedded Coder software to build projects that work with C2000™ software, C5000™ software, and C6000™ software processors.

- Highly customized code generation with the system target file `ccslink_ert.tlc` and `ccslink_grt.tlc` that enable you to use the Configuration Parameters in your model to customize your generated code.
- Automate the process of building and downloading your code to the processor, and running the process on your hardware.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded IDE Link combine to provide the following verification tools for you to apply as you develop your code:

Processor in the Loop Cosimulation

Use cosimulation techniques to verify generated code running in an instruction set simulator or real processor environment.

Execution Profiling

Gather execution profiling timing measurements with Code Composer Studio to establish the timing requirements of your algorithm. See “Profiling Code Execution in Real-Time”.

Product Features Supported for Each Processor Family

Within the collection of processors that Embedded IDE Link supports, some subcomponents of the product do not apply. For the complete list of which features work with each processor or family, refer to “Product Features Supported by Each Processor or Family” on page A-2.

Configuration Information

To determine whether Embedded IDE Link is installed on your system, type this command at the MATLAB software prompt.

```
ver
```

When you enter this command, MATLAB software displays a list of the installed products. Look for a line similar to the following:

```
Embedded IDE Link          Version 4.x    (Release Specifier)
```

To get a bit more information about the software, such as the functions provided and where to find demos and help, enter the following command at the prompt:

```
help ticcs
```

If you do not see the listing, or MATLAB software does not recognize the command, you need to install Embedded IDE Link. Without the software, you cannot use MATLAB software with the objects to communicate with CCS.

Note For up-to-date information about system requirements, see “Software Requirements” on page 1-8.

Verifying Your Code Composer Studio Installation

To verify that CCS is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

at the MATLAB software command line. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

```
Board Board          Proc Processor  Processor
Num  Name            Num  Name         Type
-----
```



```
1 C6xxx Simulator (Texas Instrum .0 6701 TMS320C6701
0 C6x13 DSK (Texas Instruments) 0 CPU TMS320C6x1x
```

If MATLAB software does not return information about any boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to ensure that it starts up successfully. For Embedded IDE Link to operate with CCS, the CCS IDE must be able to run on its own.

Embedded IDE Link uses objects to create:

- Connections to the Code Composer Studio Integrated Development Environment (CCS IDE)
- Connections to the RTDX™ (RTDX) interface. This object is a subset of the object that refers to the CCS IDE.

Concepts to know about the objects in this toolbox are covered in these sections:

- Constructing Objects
- Properties and Property Values
- Overloaded Functions for Links

Refer to MATLAB Classes and Objects in your MATLAB documentation for more details on object-oriented programming in MATLAB software.

Many of the objects use COM server features to create handles for working with the objects. Refer to your MATLAB documentation for more information about COM as used by MATLAB software.

Software Requirements

For detailed information about the software and hardware required to use Embedded IDE Link software, refer to the Embedded IDE Link system requirements areas on the MathWorks Web site:

- Requirements for Embedded IDE Link:
www.mathworks.com/products/ide-link/requirements.html
- Requirements for use with Code Composer Studio:
www.mathworks.com/products/ide-link/ti-adaptor.html

Automation Interface

- “Getting Started with Automation Interface” on page 2-2
- “Getting Started with RTDX” on page 2-20
- “Constructing ticcs Objects” on page 2-42
- “ticcs Properties and Property Values” on page 2-44
- “Overloaded Functions for ticcs Objects” on page 2-45
- “ticcs Object Properties” on page 2-46
- “Managing Custom Data Types with the Data Type Manager” on page 2-54

Getting Started with Automation Interface

In this section...
“Introducing the Automation Interface Tutorial” on page 2-2
“Selecting Your Processor” on page 2-6
“Creating and Querying Objects for CCS IDE” on page 2-8
“Loading Files into CCS” on page 2-10
“Working with Projects and Data” on page 2-12
“Closing the Links or Cleaning Up CCS IDE” on page 2-18

Introducing the Automation Interface Tutorial

Embedded IDE Link provides a connection between MATLAB software and a processor in CCS. You can use objects to control and manipulate a signal processing application using the computational power of MATLAB software. This approach can help you debug and develop your application. Another possible use for automation is creating MATLAB scripts that verify and test algorithms that run in their final implementation on your production processor.

Before using the functions available with the objects, you must select a processor to be your processor because any object you create is specific to a designated processor and a designated instance of CCS IDE. Selecting a processor is necessary for multiprocessor boards or multiple board configurations of CCS.

When you have one board with a single processor, the object defaults to the existing processor. For the objects, the simulator counts as a board; if you have both a board and a simulator that CCS recognizes, you must specify the processor explicitly.

To get you started using objects for CCS IDE software, Embedded IDE Link includes a tutorial that introduces you to working with data and files. As you work through this tutorial, you perform the following tasks that step you through creating and using objects for CCS IDE:

- 1 Select your processor.
- 2 Create and query objects to CCS IDE.
- 3 Use MATLAB software to load files into CCS IDE.
- 4 Work with your CCS IDE project from MATLAB software.
- 5 Close the connections you opened to CCS IDE.

The tutorial provides a working process (a *workflow*) for using Embedded IDE Link and your signal processing programs to develop programs for a range of Texas Instruments™ processors.

During this tutorial, you load and run a digital signal processing application on a processor you select. The tutorial demonstrates both writing to memory and reading from memory in the “Working with Projects and Data” on page 2-12” portion of the tutorial.

You can use the read and write methods, as described in this tutorial, to read and write data to and from your processor.

The tutorial covers the object methods and functions for Embedded IDE Link. The functions listed in the first table apply to CCS IDE independent of the objects — you do not need an object to use these functions. The methods listed in the second and third table requires a `ticcs` object that you use in the method syntax:

Functions for Working With Embedded IDE Link

The following functions do not require a `ticcs` object as an input argument:

Function	Description
<code>ccsboardinfo</code>	Return information about the boards that CCS IDE recognizes as installed on your PC.
<code>ticcs</code>	Construct an object to communicate with CCS IDE. When you construct the object you specify the processor board and processor.

Methods for Working with ticcs Objects

The methods in the following table require a ticcs object as an input argument:

Method	Description
	Return the address and page for an entry in the symbol table in CCS IDE.
display	Display the properties of an object to CCS IDE and RTDX.
halt	Terminate execution of a process running on the processor.
info	Return information about the processor or information about open RTDX channels.
info	Test whether your processor supports RTDX communications.
isrunning	Test whether the processor is executing a process.
read	Retrieve data from memory on the processor.
restart	Restore the program counter (PC) to the entry point for the current program.
run	Execute the program loaded on the processor.
visible	Set whether CCS IDE window is visible on the desktop while CCS IDE is running.
write	Write data to memory on the processor.

Methods for Embedded Objects

The methods in the following table enable you to manipulate programs and memory with an embedded object:

Method	Description
list	Return various information listings from Code Composer Studio software.
read	Read the information at the location accessed by an object into MATLAB software as numeric values. Demonstrated with a numeric, string, structure, and enumerated objects.
write	Write to the location referenced by an object. Demonstrated with numeric, string, structure, and enumerated objects.

Running Code Composer Studio Software on Your Desktop – Visibility

When you create a `ticcs` object, Embedded IDE Link starts CCS in the background.

When CCS IDE is running in the background, it does not appear on your desktop, in your task bar, or on the **Applications** page in the Task Manager. It does appear as a process, `cc_app.exe`, on the **Processes** tab in Microsoft® Windows Task Manager.

You can make the CCS IDE visible with the function `visible`. The function `invisible` returns the status of the IDE—whether it is visible on your desktop. To close the IDE when it is not visible and MATLAB software is not running, use the **Processes** tab in Microsoft Windows Task Manager and look for `cc_app.exe`.

If a link to CCS IDE exists when you close CCS, the application does not close. Microsoft Windows software moves it to the background (it becomes invisible).

Only after you clear all links to CCS IDE, or close MATLAB software, does closing CCS IDE unload the application. You can see if CCS IDE is running in the background by checking in the Microsoft Windows Task Manager. When CCS IDE is running, the entry `cc_app.exe` appears in the **Image Name** list on the **Processes** tab.

When you close MATLAB software while CCS IDE is not visible, MATLAB software closes CCS if it started the IDE. This happens because the operating system treats CCS as a subprocess in MATLAB software when CCS is not visible. Having MATLAB software close the invisible IDE when you close MATLAB software prevents CCS from remaining open. You do not need to close it using Microsoft Windows Task Manager.

If CCS IDE is not visible when you open MATLAB software, closing MATLAB software leaves CCS IDE running in an invisible state. MATLAB software leaves CCS IDE in the visibility and operating state in which it finds it.

Running the Interactive Tutorial

You have the option of running this tutorial from the MATLAB software command line or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB software, click `run ccstutorial`. This command opens the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next portion of the lesson. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial file by clicking `ccstutorial.m`.

Selecting Your Processor

Links for CCS IDE provides two tools for selecting a board and processor in multiprocessor configurations. One is a command line tool called `ccsboardinfo` which prints a list of the available boards and processors. So that you can use this function in a script, `ccsboardinfo` can return a MATLAB software structure that you use when you want your script to select a board without your help.

Note The board and processor you select is used throughout the tutorial.

- 1 To see a list of the boards and processors installed on your PC, enter the following command at the MATLAB software prompt:

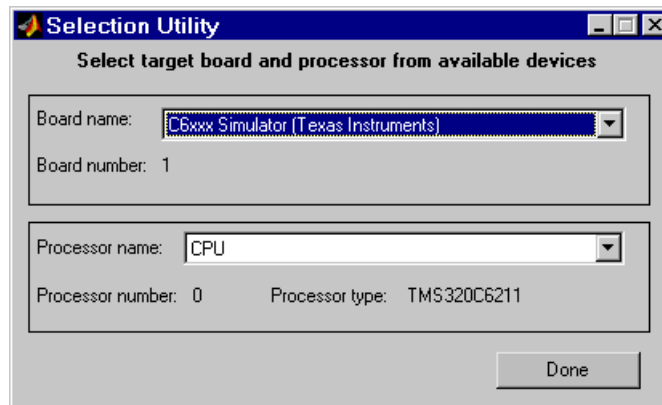
```
ccsboardinfo
```

MATLAB software returns a list that shows you all the boards and processors that CCS IDE recognizes as installed on your system.

- 2 To use the Selection Utility, `boardprocsel`, to select a board, enter

```
[boardnum,procnum] = boardprocsel
```

When you use `boardprocsel`, you see a dialog box similar to the following. Note that some entries vary depending on your board set.



- 3 Select a board name and processor name from the lists.

You are selecting a board and processor number that identifies your particular processor. When you create the object for CCS IDE in the next section of this tutorial, the selected board and processor become the processor of the object.

- 4 Click **Done** to accept your board and processor selection and close the dialog box.

boardnum and procnum now represent the **Board name** and **Processor name** you selected — boardnum = 1 and procnum = 0

Creating and Querying Objects for CCS IDE

In this tutorial section, you create the connection between MATLAB software and CCS IDE. This connection, or object, is a MATLAB software object that you save as variable `cc`.

You use function `ticcs` to create objects. When you create objects, `ticcs` input arguments let you define other object property values, such as the global timeout. Refer to the `ticcs` reference documentation for more information on these input arguments.

Use the generated object `cc` to direct actions to your processor. In the following tasks, `cc` appears in all function syntax that interact with CCS IDE and the processor:

- 1 Create an object that refers to your selected board and processor. Enter the following command at the prompt.

```
cc=ticcs('boardnum',boardnum,'procnum',procnum)
```

If you were to watch closely, and your machine is not too fast, you see Code Composer Studio software appear briefly when you call `ticcs`. If CCS IDE was not running before you created the new object, CCS starts and runs in the background.

- 2 Enter `visible(cc,1)` to force CCS IDE to be visible on your desktop.

Usually, you need to interact with Code Composer Studio software while you develop your application. The first function in this tutorial, `visible`, controls the state of CCS on your desktop. `visible` accepts Boolean inputs that make CCS either visible on your desktop (input to `visible` = 1) or invisible on your desktop (input to `visible` = 0). For this tutorial, use `visible` to set the CCS IDE visibility to 1.

- 3 Next, enter `display(cc)` at the prompt to see the status information.

```

TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

```

Embedded IDE Link provides three methods to read the status of a board and processor:

- `info` — Return a structure of testable board conditions.
- `display` — Print information about the processor.
- `isrunning` — Return the state (running or halted) of the processor.
- `info` — Return whether the hardware supports RTDX.

4 Type `linkinfo = info(cc)`.

The `cc` link status information provides information about the hardware as follows:

```

linkinfo =
  boardname: 'C6711 Device Simulator'
  procname: 'CPU_1'
  isbigendian: 0
    family: 320
    subfamily: 103
    revfamily: 11
  procestortype: 'simulator'
  revsilicon: 0
    timeout: 10

```

5 Check whether the processor is running by entering

```
runstatus = isrunning(cc)
```

MATLAB software responds, indicating that the processor is stopped, as follows:

```
runstatus =  
  
0
```

- 6** At last, verify that the processor supports RTDX communications by entering

```
usesrtdx = isrtdxcapable(cc)  
usesrtdx =  
  
1
```

Loading Files into CCS

You have established the connection to a processor and board. Using three methods you learned about the hardware, whether it was running, its type, and whether CCS IDE was visible. Next, the processor needs something to do.

In this part of the tutorial, you load the executable code for the processor CPU in CCS IDE. Embedded IDE Link includes a CCS project file. Through the next tasks in the tutorial, you locate the tutorial project file and load it into CCS IDE. The open method directs CCS to load a project file or workspace file.

Note CCS has workspace and workspace files that are different from the MATLAB workspace files and workspace. Remember to monitor both workspaces.

After you have executable code running on your processor, you can exchange data blocks with it. Exchanging data is the purpose of the objects provided by Embedded IDE Link software.

- 1** To load the appropriate project file to your processor, enter the following command at the MATLAB software prompt. `getdemo`project is a specialized function for loading Embedded IDE Link demo files. It is not supported as a standard Embedded IDE Link function.

```

demoPjt= getDemoProject(cc,'ccstutorial')

demoPjt.ProjectFile

ans =

C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\c6x\c67x\ccstut.pjt

demoPjt.DemoDir

ans =

C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\c6x\c67x

```

Your paths may be different if you use a different processor. Note where the software stored the demo files on your machine. In general, Embedded IDE Link software stores the demo project files in

```
EmbIDELinkCCDemos_v#.#
```

Embedded IDE Link creates this directory in a location where you have write permission. There are two locations where Embedded IDE Link software tries to create the demo directory, in the following order:

- a** In a temporary directory on your C drive, such as `C:\temp\`.
 - b** If Embedded IDE Link software cannot use the `temp` directory, you see a dialog box that asks you to select a location to store the demos.
- 2** Enter the following command at the MATLAB command prompt to build the processor executable file in CCS IDE.

```
build(cc,'all',20)
```

You may get an error related to one or more missing `.lib` files. If you installed CCS IDE in a directory other than the default installation directory, browse in your installation directory to find the missing file or files. Refer to the path in the error message as an indicator of where to find the missing files.

- 3** Change your working directory to the demo directory and enter `load(cc, 'projectname.out')` to load the processor execution file, where *projectname* is the tutorial you chose, such as `ccstut_67x`.

You have a loaded program file and associated symbol table to the IDE and processor.

- 4** To determine the memory address of the global symbol `ddat`, enter the following command at the prompt:

```
ddata = address(cc, 'ddat')  
ddata =
```

```
1.0e+009 *  
2.1475      0
```

Your values for `ddata` may be different depending on your processor.

Note The symbol table is available after you load the program file into the processor, not after you build a program file.

- 5** To convert `ddata` to a hexadecimal string that contains the memory address and memory page, enter the following command at the prompt:

```
dec2hex(ddata)
```

MATLAB software displays the following response, where the memory page is `0x00000000` and the address is `0x80000010`.

```
ans =  
80000010  
00000000
```

Working with Projects and Data

After you load the processor code, you can use Embedded IDE Link functions to examine and modify data values in the processor.

When you look at the source file listing in the CCS IDE Project view window, there should be a file named `ccstut.c`. Embedded IDE Link ships this file with the tutorial and includes it in the project.

`ccstut.c` has two global data arrays — `ddat` and `idat` — that you declare and initialize in the source code. You use the functions `read` and `write` to access these processor memory arrays from MATLAB software.

Embedded IDE Link provides three functions to control processor execution — `run`, `halt`, and `restart`.

- 1 To demonstrate these commands, use the following function to add a breakpoint to line 64 of `ccstut.c`.

```
insert(cc, 'ccstut.c', 64)
```

Line 64 is

```
printf("Embedded IDE Link: Tutorial - Memory Modified by Matlab!\n");
```

For information about adding breakpoints to a file, refer to `insert` in the online Help system. Then proceed with the tutorial.

- 2 To demonstrate the new functions, try the following functions.

```
halt(cc)                % Halt the processor.
restart(cc)             % Reset the PC to start of program.
run(cc, 'runtohalt', 30); % Wait for program execution to stop at
                        % breakpoint (timeout = 30 seconds).
```

When you switch to viewing CCS IDE, you see that your program stopped at the breakpoint you inserted on line 64, and the program printed the following messages in the CCS IDE **Stdout** tab. Nothing prints in the MATLAB command window:

```
Embedded IDE Link: Tutorial - Initialized Memory
Double Data array = 16.3 -2.13 5.1 11.8
Integer Data array = -1-508-647-7000 (call me anytime!)
```

- 3** Before you restart your program (currently stopped at line 64), change some values in memory. Perform one of the following procedures based on your processor.

C5xxx processor family — Enter the following functions to demonstrate the read and write functions.

- a** Enter `ddatv = read(cc,address(cc,'ddat'),'double',4)`.

MATLAB software responds with

```
ddatv =
```

```
16.3000 -2.1300 5.1000 11.8000
```

- b** Enter `idatv = read(cc,address(cc,'idat'),'int16',4)`.

Now MATLAB software responds

```
idatv =
```

```
-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(cc,address(cc,'idat'),'int8',4)
```

```
idatv =
```

```
1 0 -4 1
```

- c** You can change the values stored in `ddat` by entering `write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)... sin(pi/4)]))`

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

- d** To change `idat`, enter

```
write(cc,address(cc,'idat'),int32([1:4]))
```


Here you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

- e Start the program running again by entering the following command:

```
run(cc, 'runtohalt', 30);
```

The `Stdout` tab in CCS IDE reveals that `ddat` and `idat` contain new values. Next, read those new values back into MATLAB software.

- f Enter `ddatv = read(cc, address(cc, 'ddat'), 'double', 4)`.

```
ddatv =
```

```
3.1416 12.3000 0.3679 0.7071
```

`ddatv` contains the values you wrote in step c.

- g Verify that the change to `idatv` occurred by entering the following command at the prompt:

```
idatv = read(cc, address(cc, 'idat'), 'int16', 4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(cc);
```

C6xxx processor family — Enter the following commands to demonstrate the read and write functions.

- a Enter `ddatv = read(cc, address(cc, 'ddat'), 'double', 4)`.

MATLAB software responds with

```
ddatv =
```

```
16.3000 -2.1300 5.1000 11.8000
```

- b** Enter `idatv = read(cc,address(cc,'idat'),'int16',4)`.

MATLAB software responds

```
idatv =  
-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(cc,address(cc,'idat'),'int8',4)  
  
idatv =  
  
1 0 -4 1
```

- c** Change the values stored in `ddat` by entering `write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)... sin(pi/4)]))`

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

- d** To change `idat`, enter the following command:

```
write(cc,address(cc,'idat'),int32([1:4]))
```

In this command, you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

- e** Next, start the program running again by entering the following command:

```
run(cc,'runtohalt',30);
```

The **Stdout** tab in CCS IDE reveals that `ddat` and `idat` contain new values. Read those new values back into MATLAB software.

- f** Enter `ddatv = read(cc,address(cc,'ddat'),'double',4)`.

```
ddatv =  
  
3.1416 12.3000 0.3679 0.7071
```

Verify that `ddatv` contains the values you wrote in step c.

- g** Verify that the change to `idatv` occurred by entering the following command:

```
idatv = read(cc,address(cc,'idat'),'int32',4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h** Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(cc);
```

- 4** Embedded IDE Link offers more functions for reading and writing data to your processor. These functions let you read and write data to the processor registers: `regread` and `regwrite`. They let you change variable values on the processor in real time. The functions behave slightly differently depending on your processor. Select one of the following procedures to demonstrate `regread` and `regwrite` for your processor.

C5xxx processor family — Most registers are memory-mapped and available using `read` and `write`. However, the PC register is not memory mapped. To access this register, use the special functions — `regread` and `regwrite`. The following commands demonstrate how to use these functions to read and write to the PC register.

- a** To read the value stored in register PC, enter the following command at the prompt to indicate to MATLAB software the data type to read. The input string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

```
cc.regread('PC','binary')
```

MATLAB software displays

```
ans =
```

```
33824
```

- b** To write a new value to the PC register, enter the following command. This time, the `binary` input argument tells MATLAB software to write the value to the processor as an unsigned binary integer. Notice that you used `hex2dec` to convert the hexadecimal string to decimal.

```
cc.regwrite('PC',hex2dec('100'),'binary')
```

- c** Verify that the PC register contains the value you wrote.

```
cc.regread('PC','binary')
```

C6xxx processor family — `regread` and `regwrite` let you access the processor registers directly. Enter the following commands to get data into and out of the A0 and B2 registers on your processor.

- a** To retrieve the value in register A0 and store it in a variable in your MATLAB workspace. Enter the following command:

```
treg = cc.regread('A0','2scomp');
```

`treg` contains the two's complement representation of the value in A0.

- b** To retrieve the value in register B2 as an unsigned binary integer, enter the following command:

```
cc.regread('B2','binary');
```

- c** Next, enter the following command to use `regwrite` to put the value in `treg` into register A2.

```
cc.regwrite('A2',treg,'2scomp');
```

CCS IDE reports that A0, B2, and A2 have the values you expect. Select **View > CPU Registers > Core Registers** from the CCS IDE menu bar to list the processor registers.

Closing the Links or Cleaning Up CCS IDE

Objects that you create in Embedded IDE Link software have COM handles to CCS. Until you delete these handles, the CCS process (`cc_app.exe` in the Microsoft Windows Task Manager) remains in memory. Closing MATLAB software removes these COM handles, but there may be times when you want to delete the handles without closing the application.

Use `clear` to remove objects from your MATLAB workspace and to delete handles they contain. `clear all` deletes everything in your workspace. To retain your MATLAB software data while deleting objects and handles, use `clear objname`. This applies to IDE handle objects you created with `ticcs`. To remove the objects created during the tutorial, the tutorial program executes the following command at the prompt:

```
clear cvar cfield uintcvar
```

This tutorial also closes the project in CCS with the following command:

```
close(cc,projfile,'project')
```

To delete your link to CCS, enter `clear cc` at the prompt.

Your development tutorial using Code Composer Studio IDE is done.

During the tutorial you

- 1** Selected your processor.
- 2** Created and queried links to CCS IDE to get information about the link and the processor.
- 3** Used MATLAB software to load files into CCS IDE, and used MATLAB software to run that file.
- 4** Worked with your CCS IDE project from MATLAB software by reading and writing data to your processor, and changing the data from MATLAB software.
- 5** Created and used the embedded objects to manipulate data in a C-like way.
- 6** Closed the links you opened to CCS IDE.

Getting Started with RTDX

In this section...
“Introducing the Tutorial for Using RTDX” on page 2-21
“Creating the ticcs Objects” on page 2-26
“Configuring Communications Channels” on page 2-29
“Running the Application” on page 2-31
“Closing the Connections and Channels or Cleaning Up” on page 2-38
“Listing Functions” on page 2-41

Support for using RTDX with C5000 and C6000 processors will be removed in a future release.

Embedded IDE Link and the objects for CCS IDE and RTDX speed and enhance your ability to develop and deploy digital signal processing systems on Texas Instruments processors. By using MATLAB software and Embedded IDE Link, your MathWorks™ tools, CCS IDE and RTDX work together to help you test and analyze your processing algorithms in your MATLAB workspace.

In contrast to CCS IDE, using links for RTDX lets you interact with your process in real time while it's running on the processor. Across the connection between MATLAB software and CCS, you can:

- Send and retrieve data from memory on the processor
- Change the operating characteristics of the program
- Make changes to algorithms as needed without stopping the program or setting breakpoints in the code

Enabling real-time interaction lets you more easily see your process or algorithm in action, the results as they develop, and the way the process runs.

This tutorial assumes you have Texas Instruments' Code Composer Studio™ software and at least one DSP development board. You can use the hardware simulator in CCS IDE to run this tutorial. The tutorial uses the TMS320C6711 DSK as the board, with the C6711 DSP as the processor.

After you complete the tutorial, either in the demonstration form or by entering the functions along with this text, you are ready to begin using RTDX with your applications and hardware.

Introducing the Tutorial for Using RTDX

Digital signal processing development efforts begin with an idea for processing data; an application area, such as audio or wireless communications or multimedia computing; and a platform or hardware to host the signal processing. Usually these processing efforts involve applying strategies like signal filtering, compression, and transformation to change data content; or isolate features in data; or transfer data from one form to another or one place to another.

Developers create algorithms they need to accomplish the desired result. After they have the algorithms, they use models and DSP processor development tools to test their algorithms, to determine whether the processing achieves the goal, and whether the processing works on the proposed platform.

Embedded IDE Link and the links for RTDX and CCS IDE ease the job of taking algorithms from the model realm to the real world of the processor on which the algorithm runs.

RTDX and links for CCS IDE provide a communications pathway to manipulate data and processing programs on your processor. RTDX offers real-time data exchange in two directions between MATLAB software and your processor process. Data you send to the processor has little effect on the running process and plotting the data you retrieve from the processor lets you see how your algorithms are performing in real time.

To introduce the techniques and tools available in Embedded IDE Link for using RTDX, the following procedures use many of the methods in the link software to configure the processor, open and enable channels, send data to the processor, and clean up after you finish your testing. Among the functions covered are:

Functions From Objects for CCS IDE

Function	Description
ticcs	Create connections to CCS IDE and RTDX.
cd	Change your CCS IDE working directory from MATLAB software.
open	Load program files in CCS IDE.
run	Run processes on the processor.

Functions From the RTDX Class

Function	Description
close	Close the RTDX links between MATLAB software and your processor.
configure	Determine how many channel buffers to use and set the size of each buffer.
disable	Disable the RTDX links before you close them.
display	Return the properties of an object in formatted layout. When you omit the closing semicolon on a function, <code>disp</code> (a built-in function) provides the default display for the results of the operation.
enable	Enable open channels so you can use them to send and retrieve data from your processor.
isenabled	Determine whether channels are enabled for RTDX communications.

Function	Description
isreadable	Determine whether MATLAB software can read the specified memory location.
iswritable	Determine whether MATLAB software can write to the processor.
msgcount	Determine how many messages are waiting in a channel queue.
open	Open channels in RTDX.
readmat	Read data matrices from the processor into MATLAB software as an array.
readmsg	Read one or more messages from a channel.
writemsg	Write messages to the processor over a channel.

This tutorial provides the following workflow to show you how to use many of the functions in the links. By performing the steps provided, you work through many of the operations yourself. The tutorial follows the general task flow for developing digital signal processing programs through testing with the links for RTDX.

Within this set of tasks, numbers 1, 2, and 4 are fundamental to all development projects. Whenever you work with MATLAB software and objects for RTDX, you perform the functions and tasks outlined and presented in this tutorial. The differences lie in Task 3. Task 3 is the most important for using Embedded IDE Link to develop your processing system.

- 1** Create an RTDX link to your desired processor and load the program to the processor.

All projects begin this way. Without the links you cannot load your executable to the processor.

- 2** Configure channels to communicate with the processor.

Creating the links in Task 1 did not open communications to the processor. With the links in place, you open as many channels as you need to support the data transfer for your development work. This task includes configuring channel buffers to hold data when the data rate from the processor exceeds the rate at which MATLAB software can capture the data.

- 3** Run your application on the processor. You use MATLAB software to investigate the results of your running process.
- 4** Close the links to the processor and clean up the links and associated debris left over from your work.

Closing channels and cleaning up the memory and links you created ensures that CCS IDE, RTDX, and Embedded IDE Link are ready for the next time you start development on a project.

This tutorial uses an executable program named `rtdxtutorial_6xevm.out` as your application. When you use the RTDX and CCS IDE links to develop your own applications, replace `rtdxtutorial_6xevm.out` in Task 3 with the filename and path to your digital signal processing application.

You can view the tutorial file used here by clicking `rtdxtutorial`. To run this tutorial in MATLAB software, click run `rtdxtutorial`.

Note To be able to open and enable channels over a link to RTDX, the program loaded on your processor must include functions or code that define the channels.

Your C source code might look something like this to create two channels, one to write and one to read.

```
    rtdx_CreateInputChannel(ichan); % processor reads from this.  
    rtdx_CreateOutputChannel(ochan); % processor writes to this.
```

These are the entries we use in `int16.c` (the source code that generates `rtdxtutorial_6xevm.out`) to create the read and write channels.

If you are working with a model in Simulink software and using code generation, use the To Rtdx and From Rtdx blocks in your model to add the RTDX communications channels to your model and to the executable code on your processor.

One more note about this tutorial. Throughout the code we use both the dot notation (direct property referencing) to access functions and link properties and the function form.

For example, use the following command to open and configure `ichan` for write mode.

```
cc.rtdx.open('ichan','w');
```

You could use an equivalent syntax, the function form, that does not use direct property referencing.

```
open(cc.rtdx,'ichan','w');
```

Or, use

```
open(rx,'ichan','w');
```

if you created an alias `rx` to the RTDX portion of `cc`, as shown by the following command:

```
rx = cc.rtdx;
```

Creating the ticcs Objects

With your processing model converted to an executable suitable for your desired processor, you are ready to use the objects to test and run your model on your processor. Embedded IDE Link and the objects do not distinguish the source of the executable — whether you used Embedded IDE Link and Real-Time Workshop, CCS IDE, or some other development tool to program and compile your model to an executable does not affect the object connections. So long as your `..out` file is acceptable to the processor you select, Embedded IDE Link provides the connection to the processor.

Note Program `rtdxtutorial_6xevm.out` uses the C6711. The executable is compiled, built, and linked to run on the C6711 processor. To use the tutorial without changes, specify your C6711 when you define the object properties `boardnum` and `procnum`.

Before continuing with this tutorial, you must load a valid GEL file to configure the EMIF registers of your processor and perform any required processor initialization steps. Default GEL files provided by CCS are stored in `..\cc\gel` in the folder where you installed CCS software. Select **File** > **Load GEL** in CCS IDE to load the default GEL file that matches your processor family, such as `init6x0x.gel` for the C6x0x processor family, and your configuration.

Begin the process of getting your model onto the processor by creating a an object that refers to CCS IDE. Start by clearing all existing handles and setting echo on so you see functions execute as the program runs:

```
1 clear all; echo on;
```

`clear all` has the side effect of removing debugging breakpoints and resetting persistent variables because function breakpoints and persistent variables are cleared whenever the MATLAB file changes or is cleared. Breakpoints within your executable remain after `clear`. Clearing the MATLAB workspace does not affect your executable.

2 Now construct the link to your board and processor by entering

```
cc=tics('boardnum',0);
```

`boardnum` defines which board the new link accesses. In this example, `boardnum` is 0. Embedded IDE Link connects the link to the first, and in this case only, processor on the board. To find the `boardnum` and `procnum` values for the boards and simulators on your system, use `ccsboardinfo`. When you enter the following command at the prompt

```
ccsboardinfo
```

Embedded IDE Link returns a list like the following one that identifies the boards and processors in your computer.

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Inst...	0	CPU	TMS320C6211
0	C6701 EVM (Texas Instruments)	0	CPU_1	TMS320C6701

- 3 To open and load the processor file, change the path for MATLAB software to be able to find the file.

```
projname =
```

```
C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\c6x\c64xp\rtdx_tut_sim.pjt
```

```
outFile =
```

```
C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\c6x\c64xp\rtdx_tut_sim.out
```

```
processor_dir = demoPjt.DemoDir
```

```
processor_dir =
```

```
C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\c6x\c64xp
```

```
% Go to processor directory
cd(cc,processor_dir);cd(cc,tgt_dir); % Or cc.cd(tgt_dir)
dir(cc); % Or cc.dir
```

To load the appropriate project file to your processor, enter the following commands at the MATLAB software prompt. `getDemoProject` is a specialized function for loading Embedded IDE Link demo files. It is not supported as a standard Embedded IDE Link function.

```
demoPjt = getDemoProject(cc,'ccstutorial');

demoPjt.ProjectFile

ans =

C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\c6x\c64xp\ccstut.pjt

demoPjt.DemoDir

ans =

C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\c6x\c64xp
```

Notice where the demo files are stored on your machine. In general, Embedded IDE Link software stores the demo project files in

```
EmbIDELinkCCDemos_v#.#
```

For example, if you are using version 4.1 of Embedded IDE Link software, the project demos are stored in `EmbIDELinkCCDemos_v4.1\`. Embedded IDE Link software creates this folder in a location on your machine where you have write permission. Usually, there are two locations where Embedded IDE Link software tries to create the demo folder, in the order shown.

- a** In a temporary folder on the C drive, such as `C:\temp\`.
- b** If Embedded IDE Link software cannot use the `temp` folder, you see a dialog box that asks you to select a location to store the demos.

- 4 You have reset the folder path to find the tutorial file. Now open the .out file that matches your processor type, such as rtdxtutorial_c67x.out or rtdxtutorial_c64x.out.

```
cc.open('rtdxtutorial_67x.out')
```

Because open is overloaded for the CCS IDE and RTDX links, this may seem a bit strange. In this syntax, open loads your executable file onto the processor identified by cc. Later in this tutorial, you use open with a different syntax to open channels in RTDX.

In the next section, you use the new link to open and enable communications between MATLAB software and your processor.

Configuring Communications Channels

Communications channels to the processor do not exist until you open and enable them through Embedded IDE Link and CCS IDE. Opening channels consists of opening and configuring each channel for reading or writing, and enabling the channels.

In the open function, you provide the channel names as strings for the channel name property. The channel name you use is not random. The channel name string must match a channel defined in the executable file. If you specify a string that does not identify an existing channel in the executable, the open operation fails.

In this tutorial, two channels exist on the processor — ichan and ochan. Although the channels are named ichan for input channel and ochan for output channel, neither channel is configured for input or output until you configure them from MATLAB software or CCS IDE. You could configure ichan as the output channel and ochan as the input channel. The links would work just the same. For simplicity, the tutorial configures ichan for input and ochan for output. One more note—reading and writing are defined as seen by the processor. When you write data from MATLAB software, you write to the channel that the processor reads, ichan in this case. Conversely, when you read from the processor, you read from ochan, the channel that the processor writes to:

- 1** Configure buffers in RTDX to store the data until MATLAB software can read it into your workspace. Often, MATLAB software cannot read data as quickly as the processor can write it to the channel.

```
cc.rtdx.configure(1024,4); % define 4 channels of 1024 bytes each
```

Channel buffers are optional. Adding them provides a measure of insurance that data gets from your processor to MATLAB software without getting lost.

- 2** Define one of the channels as a write channel. Use 'ichan' for the channel name and 'w' for the mode. Either 'w' or 'r' fits here, for write or read.

```
cc.rtdx.open('ichan','w');
```

- 3** Now enable the channel you opened.

```
cc.rtdx.enable('ichan');
```

- 4** Repeat steps 2 and 3 to prepare a read channel.

```
cc.rtdx.open('ochan','r');  
cc.rtdx.enable('ochan');
```

- 5** To use the new channels, enable RTDX by entering

```
cc.rtdx.enable;
```

You could do this step before you configure the channels — the order does not matter.

- 6** Reset the global time-out to 20s to provide a little room for error. `ticcs` applies a default timeout value of 10s. In some cases this may not be enough.

```
cc.rtdx.get('timeout')  
ans =  
    10  
cc.rtdx.set('timeout', 20); % Reset timeout = 20 seconds
```


- 7 Check that the `timeout` property value is now 20s and that your object has the correct configuration for the rest of the tutorial.

```
cc.rtdx
```

```
RTDX Object:
```

```
API version:      1.0
Default timeout:  20.00 secs
Open channels:    2
```

Running the Application

To this point you have been doing housekeeping functions that are common to any application you run on the processor. You load the processor, configure the communications, and set up other properties you need.

In this tutorial task, you use a specific application to demonstrate a few of the functions available in Embedded IDE Link that let you experiment with your application while you develop your prototype. To demonstrate the link for RTDX `readmat`, `readmsg`, and `writemsg` functions, you write data to your processor for processing, then read data from the processor after processing:

- 1 Restart the program you loaded on the processor. `restart` ensures the program counter (PC) is at the beginning of the executable code on the processor.

```
cc.restart
```

Restarting the processor does not start the program executing. You use `run` to start program execution.

- 2 Type `cc.run('run');`

Using `'run'` for the run mode tells the processor to continue to execute the loaded program continuously until it receives a halt directive. In this mode, control returns to MATLAB software so you can work in MATLAB software while the program runs. Other options for the mode are

- `'runtohalt'` — start to execute the program and wait to return control to MATLAB software until the process reaches a breakpoint or execution terminates.

- 'tohalt' — change the state of a running processor to 'runtohalt' and wait to return until the program halts. Use tohalt mode to stop the running processor cleanly.

- 3** Type the following functions to enable the write channel and verify that the enable takes effect.

```
cc.rtdx.enable('ichan');  
cc.rtdx.isenabled('ichan')
```

If MATLAB software responds `ans = 0` your channel is not enabled and you cannot proceed with the tutorial. Try to enable the channel again and verify the status.

- 4** Write some data to the processor. Check that you can write to the processor, then use `writemsg` to send the data. You do not need to enter the if-test code shown.

```
if cc.rtdx.iswritable('ichan'), % Used in a script application.  
    disp('writing to processor...') % Optional to display progress.  
    indata=1:10  
    cc.rtdx.writemsg('ichan', int16(indata))  
end % Used in scripts for channel testing.
```

The `if` statement simulates writing the data from within a MATLAB software script. The script uses `iswritable` to check that the input channel is functioning. If `iswritable` returns 0 the script would skip the write and exit the program, or respond in some way. When you are writing or reading data to your processor in a script or MATLAB file, checking the status of the channels can help you avoid errors during execution.

As your application runs you may find it helpful to display progress messages. In this case, the program directed MATLAB software to print a message as it reads the data from the processor by adding the function

```
disp('writing to processor...')
```

Function `cc.rtdx.writemsg('ichan', int16(indata))` results in 20 messages stored on the processor. Here's how.

When you write `indata` to the processor, the following code running on the processor takes your input data from `ichan`, adds one to the values and copies the data to memory:

```
while ( !RTDX_isInputEnabled(&ichan) )

/* wait for channel enable from MATLAB */
RTDX_read( &ichan, recvd, sizeof(recvd) );
puts("\n\n Read Completed ");

for (j=1; j<=20; j++) {
    for (i=0; i<MAX; i++) {
        recvd[i] +=1;
    }
    while ( !RTDX_isOutputEnabled(&ochan) )
        { /* wait for channel enable from MATLAB */ }
    RTDX_write( &ochan, recvd, sizeof(recvd) );
    while ( RTDX_writing != NULL )
        { /* wait for data xfer INTERRUPT DRIVEN for C6000 */ }
}
```

Program `int16_rtdx.c` contains this source code. You can find the file in a folder in the `..\tidemos\rtdxtutorial` folder.

- 5** Type the following to check the number of available messages to read from the processor.

```
num_of_msgs = cc.rtdx.msgcount('ochan');
```

`num_of_msgs` should be zero. Using this process to check the amount of data can make your reads more reliable by letting you or your program know how much data to expect.

- 6** Type the following to verify that your read channel `ochan` is enabled for communications.

```
cc.rtdx.isenabled('ochan')
```

You should get back `ans = 0` — you have not enabled the channel yet.

- 7** Now enable and verify `'ochan'`.

```
cc.rtdx.enable('ochan');  
cc.rtdx.isenabled('ochan')
```

To show that ochan is ready, MATLAB software responds `ans = 1`. If not, try enabling ochan again.

8 Type

```
pause(5);
```

The `pause` function gives the processor extra time to process the data in `indata` and transfer the data to the buffer you configured for ochan.

9 Repeat the check for the number of messages in the queue. There should be 20 messages available in the buffer.

```
num_of_msgs = cc.rtdx.msgcount('ochan')
```

With `num_of_msgs = 20`, you could use a looping structure to read the messages from the queue in to MATLAB software. In the next few steps of this tutorial you read data from the ochan queue to different data formats within MATLAB software.

10 Read one message from the queue into variable `outdata`.

```
outdata = cc.rtdx.readmsg('ochan','int16')
```

```
outdata =  
     2     3     4     5     6     7     8     9    10    11
```

Notice the `'int16'` represent option. When you read data from your processor you need to tell MATLAB software the data type you are reading. You wrote the data in step 4 as 16-bit integers so you use the same data type here.

While performing reads and writes, your process continues to run. You did not need to stop the processor to get the data or send the data, unlike using most debuggers and breakpoints in your code. You placed your data in memory across an RTDX channel, the processor used the data, and you read the data from memory across an RTDX channel, without stopping the processor.

- 11** You can read data into cell arrays, rather than into simple double-precision variables. Use the following function to read three messages to cell array `outdata`, an array of three, 1-by-10 vectors. Each message is a 1-by-10 vector stored on the processor.

```
outdata = cc.rtdx.readmsg('ochan','int16',3)

outdata =
 [1x10 int16] [1x10 int16] [1x10 int16]
```

- 12** Cell array `outdata` contains three messages. Look at the second message, or matrix, in `outdata` by using dereferencing with the array.

```
outdata{1,2}

outdata =
     4     5     6     7     8     9    10    11    12    13
```

- 13** Read two messages from the processor into two 2-by-5 matrices in your MATLAB workspace.

```
outdata = cc.rtdx.readmsg('ochan','int16',[2 5],2)

outdata =
 [2x5 int16] [2x5 int16]
```

To specify the number of messages to read and the data format in your workspace, you used the `siz` and `nummsgs` options set to `[2 5]` and `2`.

- 14** You can look at both matrices in `outdata` by dereferencing the cell array again.

```
outdata{1,:}

ans =
     6     8    10    12    14
     7     9    11    13    15
ans =
     7     9    11    13    15
     8    10    12    14    16
```

- 15** For a change, read a message from the queue into a column vector.

```
outdata = cc.rtdx.readmsg('ochan','int16',[10 1])
```

```
outdata =  
      8  
      9  
     10  
     11  
     12  
     13  
     14  
     15  
     16  
     17
```

- 16** Embedded IDE Link provides a function for reading messages into matrices—`readmat`. Use `readmat` to read a message into a 5-by-2 matrix in MATLAB software.

```
outdata = readmat(cc.rtdx,'ochan','int16',[5 2])
```

```
outdata =  
      9  14  
     10  15  
     11  16  
     12  17  
     13  18
```

Because a 5-by-2 matrix requires ten elements, MATLAB software reads one message into `outdata` to fill the matrix.

- 17** To check your progress, see how many messages remain in the queue. You have read eight messages from the queue so 12 should remain.

```
num_of_msgs = cc.rtdx.msgcount('ochan')
```

```
num_of_msgs =  
      12
```

- 18** To demonstrate the connection between messages and a matrix in MATLAB software, read data from 'ochan' to fill a 4-by-5 matrix in your workspace.

```
outdata = cc.rtdx.readmat('ochan','int16',[4 5])
```

```
outdata =
    10    14    18    13    17
    11    15    19    14    18
    12    16    11    15    19
    13    17    12    16    20
```

Filling the matrix required two messages worth of data.

- 19** To verify that the last step used two messages, recheck the message count. You should find 10 messages waiting in the queue.

```
num_of_msgs = cc.rtdx.msgcount('ochan')
```

- 20** Continuing with matrix reads, fill a 10-by-5 matrix (50 matrix elements or five messages).

```
outdata = cc.rtdx.readmat('ochan','int16',[10 5])
```

```
outdata =
    12    13    14    15    16
    13    14    15    16    17
    14    15    16    17    18
    15    16    14    18    19
    16    17    18    19    20
    17    18    19    20    21
    18    19    20    21    22
    19    20    21    22    23
    20    21    22    23    24
    21    22    23    24    25
```

- 21** Recheck the number of messages in the queue to see that five remain.
- 22** flush lets you remove messages from the queue without reading them. Data in the message you remove is lost. Use flush to remove the next message in the read queue. Then check the waiting message count.

```
cc.rtdx.flush('ochan',1)
num_of_msgs = cc.rtdx.msgcount('ochan')

num_of_msgs =

4
```

- 23** Empty the remaining messages from the queue and verify that the queue is empty.

```
cc.rtdx.flush('ochan','all')
```

With the `all` option, `flush` discards all messages in the `ochan` queue.

Closing the Connections and Channels or Cleaning Up

One of the most important programmatic processes you should do in every RTDX session is to clean up at the end. Cleaning up includes stopping your processor, disabling the RTDX channels you enabled, disabling RTDX and closing your open channels. Performing this series of tasks ensures that future processes avoid trouble caused by unexpected interactions with remaining handles, channels, and links from earlier development work.

Best practices suggest that you include the following tasks (or an appropriate subset that meets your development needs) in your development scripts and programs.

We use several functions in this section; each has a purpose — the operational details in the following list explain how and why we use each one. They are

- `close` — close the specified RTDX channel. To use the channel again, you must open and enable the channel. Compare `close` to `disable`. `close('rtdx')` closes the communications provided by RTDX. After you close RTDX, you cannot communicate with your processor.
- `disable` — remove RTDX communications from the specified channel, but does not remove the channel, or link. Disabling channels may be useful when you do not want to see the data that is being fed to the channel, but you may want to read the channel later. By enabling the channel later, you have access to the data entering the channel buffer. Note that data that entered the channel while it was disabled is lost.

- `halt` — stop a running processor. You may still have one or more messages in the host buffer.

Use the following procedure to shut down communications between MATLAB software and the processor, and end your session:

- 1 Begin the process of shutting down the processor and RTDX by stopping the processor. Type the following functions at the prompt.

```
if (isrunning(cc)) % Use this test in scripts.
    cc.halt;        % Halt the processor.
end                % Done.
```

Your processor may already be stopped at this point. In a script, you might put the function in an `if`-statement as we have done here. Consider this test to be a safety check. No harm comes to the processor if it is already stopped when you tell it to stop. When you direct a stopped processor to `halt`, the function returns immediately.

- 2 You have stopped the processor. Now disable the RTDX channels you opened to communicate with the processor.

```
cc.rtdx.disable('all');
```

If necessary, using `disable` with channel name and processor identifier input arguments lets you disable only the channel you choose. When you have more than one board or processor, you may find disabling selected channels meets your needs.

When you finish your RTDX communications session, disable RTDX to ensure that Embedded IDE Link releases your open channels before you close them.

```
cc.rtdx.disable;
```

- 3 Use one or all of the following function syntaxes to close your open channels. Either close selected channels by using the channel name in the function, or use the `all` option to close all open channels.
 - `cc.rtdx.close('ichan')` to close your input channel in this tutorial.
 - `cc.rtdx.close('ochan')` to close your output channel in the tutorial.

- `cc.rtdx.close('all')` to close all of your open RTDX channels, regardless of whether they are part of this tutorial.

Consider using the `all` option with the `close` function when you finish your RTDX work. Closing channels reduces unforeseen problems caused by channel objects that exist but do not get closed correctly when you end your session.

- 4 When you created your RTDX object (`cc = ticcs('boardnum',1)`) at the beginning of this tutorial, the `ticcs` function opened CCS IDE and set the visibility to 0. To avoid problems that occur when you close the interface to RTDX with CCS visibility set to 0, make CCS IDE visible on your desktop. The following `if` statement checks the CCS IDE visibility and changes it if needed.

```
if cc.isvisible,  
    cc.visible(1);  
end
```

Visibility can cause problems. When CCS IDE is running invisibly on your desktop, do not use `clear all` to remove your links for CCS IDE and RTDX. Without a `ticcs` object that references the CCS IDE you cannot access CCS IDE to change the visibility setting, or close the application. To close CCS IDE when you do not have an existing object, either create a new object to access the CCS IDE, or use Microsoft Windows Task Manager to end the process `cc_app.exe`, or close the MATLAB software.

- 5 You have finished the work in this tutorial. Enter the following commands to close your remaining references to CCS IDE and release the associated resources.

```
clear ('all'); % Calls the link destructors to remove all links.  
echo off
```

`clear all` without the parentheses removes all variables from your MATLAB workspace.

You have completed the tutorial using RTDX. During the tutorial you

- 1 Opened connections to CCS IDE and RTDX and used those connections to load an executable program to your processor.

- 2** Configured a pair of channels so you could transfer data to and from your processor.
- 3** Ran the executable on the processor, sending data to the processor for processing and retrieving the results.
- 4** Stopped the executing program and closed the links to CCS IDE and RTDX.

This tutorial provides a working process for using Embedded IDE Link and your signal processing programs to develop programs for a range of Texas Instruments processors. While the processor may change, the essentials of the process remain the same.

Listing Functions

To review a complete list of functions and methods that operate with `ticcs` objects, either CCS IDE or RTDX, enter either of the following commands at the prompt.

```
help ticcs
help rtdx
```

If you already have a `ticcs` object `cc`, you can use dot notation to return the methods for CCS IDE or RTDX by entering one of the following commands at the prompt:

- `cc.methods`
- `cc.rtdx.methods`

In either instance MATLAB software returns a list of the available functions for the specified link type, including both public and private functions. For example, to see the functions (methods) for links to CCS IDE, enter:

```
help ticcs
```

Constructing ticcs Objects

When you create a connection to CCS IDE using the `ticcs` command, you are creating a “`ticcs` object for accessing the CCS IDE and RTDX interface”. The `ticcs` object implementation relies on MATLAB software object-oriented programming capabilities.

The discussions in this section apply to the `ticcs` objects in Embedded IDE Link.

Like other MATLAB software structures, objects in Embedded IDE Link have predefined fields called object properties.

You specify object property values by one of the following methods:

- Setting the property values when you create the `ticcs` object
- Creating an object with default property values, and changing some or all of these property values later

For examples of setting `ticcs` object properties, refer to `ticcs`.

Example – Constructor for ticcs Objects

The easiest way to create an object is to use the function `ticcs` to create an object with the default properties. Create an object named `cc` to refer to CCS IDE by entering

```
cc = ticcs
```

MATLAB software responds with a list of the properties of the object `cc` you created along with the associated default property values.

```
ticcs object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs
```

```
RTDX channels      : 0
```

Inspecting the output reveals two objects listed—a CCS IDE object and an RTDX object. CCS IDE and RTDX objects cannot be created separately. By design they maintain a member class relationship; the RTDX object is a class, a member of the CCS object class. In this example, `cc` is an instance of the class CCS. If you enter

```
rx = cc.rtdx
```

`rx` is a handle to the RTDX portion of the CCS object. As an alias, `rx` replaces `cc.rtdx` in functions such as `readmat` or `writemsg` that use the RTDX communications features of the CCS link. Typing `rx` at the command line now produces

```
rx
```

```
RTDX channels      : 0
```

The object properties are described in “Function Reference”, and in more detail in ticcs Object Properties. These properties are set to default values when you construct objects.

tics Properties and Property Values

Objects in Embedded IDE Link software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. And a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

For more information about using objects and properties, refer to “Using Objects” in *MATLAB Programming Fundamentals*.

Overloaded Functions for ticcs Objects

Several functions in this Embedded IDE Link have the same name as functions in other MathWorks toolboxes or in MATLAB software. These behave similarly to their original counterparts, but you apply these functions directly to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for `ticcs` objects. After you specify your link by assigning values to its properties, you can apply the functions in this toolbox (such as `readmat` for using RTDX to read an array of data from the processor) directly to the variable name you assign to your object, without specifying your object parameters again.

For a complete list of the functions that act on `ticcs` objects, refer to the “Function Reference”.

ticcs Object Properties

In this section...

“Quick Reference to ticcs Object Properties” on page 2-46

“Details About ticcs Object Properties” on page 2-48

Embedded IDE Link provides an interface to your processor hardware so you can communicate with processors for which you are developing systems and algorithms. Each ticcs object comprises two objects—a CCS IDE object and an RTDX interface object. The objects are not separable; the RTDX object is a subclass of the CCS IDE object. Each of the objects has multiple properties. To configure the interface objects for CCS IDE and RTDX, you set parameters that define details such as the desired board, the processor, the timeout period applied for communications operations, and a number of other values. Because Embedded IDE Link uses objects to create the interface, the parameters you set are called properties and you treat them as properties when you set them, retrieve them, or modify them.

This section details the properties for the ticcs objects for CCS IDE and RTDX. First the section provides tables of the properties, for quick reference. Following the tables, the section offers in-depth descriptions of each property, its name and use, and whether you can set and get the property value associated with the property. Descriptions include a few examples of the property in use.

MATLAB software users may find much of this handling of objects familiar. Objects in Embedded IDE Link, behave like objects in MATLAB software and the other object-oriented toolboxes. For C++ programmers, discussion of object-oriented programming is likely to be a review.

Quick Reference to ticcs Object Properties

The following table lists the properties for the ticcs objects in Embedded IDE Link. The second column tells you which object the property belongs to. Knowing which property belongs to each object in a ticcs object tells you how to access the property.

Property Name	Applies to Which Connection?	User Settable?	Description
apiversion	CCS IDE	No	Reports the version number of your CCS API.
boardnum	CCS IDE	Yes/initially	Specifies the index number of a board that CCS IDE recognizes.
ccsappexe	CCS IDE	No	Specifies the path to the CCS IDE executable. Read-only property.
numchannels	RTDX	No	Contains the number of open RTDX channels for a specific link.
page	CCS IDE	Yes/default	Stores the default memory page for reads and writes.
procnum	CCS IDE	Yes/at start only	Stores the number CCS Setup Utility assigns to the processor.
rtdx	RTDX	No	Specifies RTDX in a syntax.
rtdxchannel	RTDX	No	A string. Identifies the RTDX channel for a link.
timeout	CCS IDE	Yes/default	Contains the global timeout setting for the link.
version	RTDX	No	Reports the version of your RTDX software.

Some properties are read only — you cannot set the property value. Other properties you can change at all times. If the entry in the User Settable column is “Yes/initially”, you can set the property value only when you create the link. Thereafter it is read only.

Details About `ticc` Object Properties

To use the links for CCS IDE and RTDX interface you set values for:

- `boardnum` — specify the board with which the link communicates.
- `procnum` — specify the processor on the board. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — specify the global timeout value. (Optional. Default is 10s.)

Details of the properties associated with connections to CCS IDE and RTDX interface appear in the following sections, listed in alphabetical order by property name.

Many of these properties are object linking and embedding (OLE) handles. The MATLAB software COM server creates the handles when you create objects for CCS IDE and RTDX. You can manipulate the OLE handles using `get`, `set`, and `invoke` to work directly with the COM interface with which the handles interact.

apiversion

Property `apiversion` contains a string that reports the version of the application program interface (API) for CCS IDE that you are using when you create a link. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `apiversion` property value for a link. This example shows the `apiversion` value for link `cc`.

```
display(cc)
```

```
TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0
```

Note that the API version is not the same as the CCS IDE version.

boardnum

Property `boardnum` identifies the board referenced by a link for CCS IDE. When you create a link, you use `boardnum` to specify the board you are using. To get the value for `boardnum`, use `ccsboardinfo` or the CCS Setup utility from Texas Instruments software. The CCS Setup utility assigns the number for each board installed on your system.

ccsappexe

Property `ccsappexe` contains the path to the CCS IDE executable file `cc_app.exe`. When you use `ticcs` to create a link, MATLAB software determines the path to the CCS IDE executable and stores the path in this property. This is a read-only property. You cannot set it.

numchannels

Property `numchannels` reports the number of open RTDX communications channels for an RTDX link. Each time you open a channel for a link, `numchannels` increments by one. For new links `numchannels` is zero until you open a channel for the link.

To see the value for `numchannels` create a link to CCS IDE. Then open a channel to RTDX. Use `display` to see the RTDX link properties.

```
cc=ticcs
```

```
TICCS Object:
```

```
API version      : 1.0
Processor type   : C67
Processor name   : CPU
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs

RTDX channels    : 0
```

```
rx=cc.rtdx

    RTDX channels      : 0

open(rx,'ichan','r','ochan','w');

get(cc.rtdx)

ans =

    numChannels: 2
           Rtdx: [1x1 COM ]
    RtdxChannel: {' ' ' '}
           procType: 103
           timeout: 10
```

page

Property `page` contains the default value CCS IDE uses when the user does not specify the `page` input argument in the syntax for a function that access memory.

procnum

Property `procnum` identifies the processor referenced by a link for CCS IDE. When you create an object, you use `procnum` to specify the processor you are using. The CCS Setup Utility assigns a number to each processor installed on each board. To determine the value of `procnum` for a processor, use `ccsboardinfo` or the CCS Setup utility from Texas Instruments software.

To identify a processor, you need both the `boardnum` and `procnum` values. For boards with one processor, `procnum` equals zero. CCS IDE numbers the processors on multiprocessor boards sequentially from 0 to the number of processors. For example, on a board with four processors, the processors are numbered 0, 1, 2, and 3.

rtdx

Property `rtdx` is a subclass of the `ticcs` link and represents the RTDX portion of a link for CCS IDE. As shown in the example, `rtdx` has properties of its own that you can set, such as `timeout`, and that report various states of the link.

```
get(cc.rtdx)

ans =

    version: 1
  numChannels: 0
      Rtdx: [1x1 COM ]
RtdxChannel: {'' [] ''}
   procType: 103
     timeout: 10
```

In addition, you can create an alias to the `rtdx` portion of a link, as shown in this code example.

```
rx=cc.rtdx

RTDX channels      : 0
```

Now you can use `rx` with the functions in Embedded IDE Link, such as `get` or `set`. If you have two open channels, the display looks like the following

```
get(rx)

ans =

    numChannels: 2
      Rtdx: [1x1 COM ]
RtdxChannel: {2x3 cell}
   procType: 98
     timeout: 10
```

when the processor is from the C62 family.

rtdxchannel

Property `rtdxchannel`, along with `numchannels` and `proctype`, is a read-only property for the RTDX portion of a link for CCS IDE. To see the value of this property, use `get` with the link. Neither `set` nor `invoke` work with `rtdxchannel`.

`rtdxchannel` is a cell array that contains the channel name, handle, and mode for each open channel for the link. For each open channel, `rtdxchannel` contains three fields, as follows:

<code>.rtdxchannel{i,1}</code>	Channel name of the <i>i</i> th-channel, <i>i</i> from 1 to the number of open channels
<code>.rtdxchannel{i,2}</code>	Handle for the <i>i</i> th-channel
<code>.rtdxchannel{i,3}</code>	Mode of the <i>i</i> th-channel, either 'r' for read or 'w' for write

With four open channels, `rtdxchannel` contains four channel elements and three fields for each channel element.

timeout

Property `timeout` specifies how long CCS IDE waits for any process to finish. Two `timeout` periods can exist — one global, one local. You set the global `timeout` when you create a link for CCS IDE. The default global `timeout` value 10 s. However, when you use functions to read or write data to CCS IDE or your processor, you can set a local `timeout` that overrides the global value. If you do not set a specific `timeout` value in a read or write process syntax, the global `timeout` value applies to the operation. Refer to the help for the read and write functions for the syntax to set the local `timeout` value for an operation.

version

Property `version` reports the version number of your RTDX software. When you create a `ticcs` object, `version` contains a string that reports the version of the RTDX application that you are using. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `version` property value for a link. This example shows the `apiversion` value for object `rx`.

```
get(rx) % rx is an alias for cc.rtdx.
```

```
ans =
```

```
    version: 1  
 numChannels: 0  
      Rtdx: [1x1 COM ]  
RtdxChannel: {'' [] ''}  
   procType: 103  
   timeout: 10
```

Managing Custom Data Types with the Data Type Manager

Using custom data types, called typedefs (using the C keyword `typedef`), is one of the complications you encounter when you use hardware-in-the-loop (HIL) to run a function in your project from MATLAB. Because MATLAB does not recognize custom type definitions you use in your projects, it cannot interpret data that you define in your project code with the `typedef` keyword, or use as arguments in your function prototype (declaration).

To allow you to use functions that include custom type definitions in function calls, Embedded IDE Link offers the Data Type Manager (DTM), a tool for defining custom type definitions to MATLAB. Using options in the DTM, you define one or more custom data types for a project and use them in the project. Or you define your custom data types and save them to use in many projects. This second feature is particularly useful when you use the same custom data types in many projects. Rather than redefining your custom types for each new project or function, you reload the types from an earlier project to use them again.

As programmers, usually you use typedefs for one or more of a few reasons:

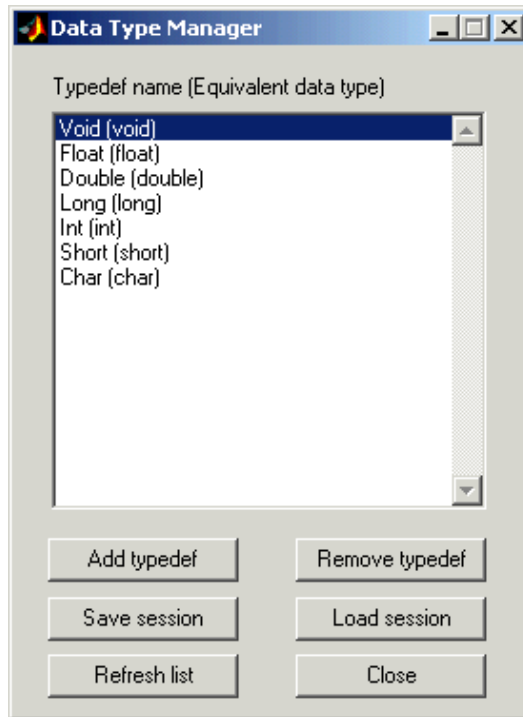
- Make your code more accessible by providing more information about the variable(s)
- Create a Boolean data type that C does not provide
- Define structures in your programs
- Define nonstandard data types

The DTM lets you define all of these things in the MATLAB context so your C function that uses typedefs works with your MATLAB command line functions. For reference information about the DTM, go to [datatypemanager](#).

Entering

```
datatypemanager(cc)
```


at the MATLAB command line opens the DTM, with the Data Type Manager dialog box shown here:



When the DTM opens, a variety of information and options displays in the Data Type Manager dialog box:

- **Typedef name (Equivalent data type)** — provides a list of default data types. When you create a typedef, you see it added to this list.

The lowercase versions of the data types appear because MATLAB does not recognize the initial capital versions automatically. In the data type list the project data type with the initial capital letter is mapped to the lowercase MATLAB data type.

- **Add typedef** — opens the **Add Typedef** dialog box so you can add one or more typedefs to your object. Your added typedef appears on the **Typedef name (Equivalent data type)** list and is added to your ticcs object. Also,

when you pass the `cc` object to the DTM, and then add a `typedef`, the command

```
cc.type
```

returns the list of data types in the `type` property of your `cc` object, including the `typedefs` you added.

- **Remove typedef** — removes a selected `typedef` from the **Typedef name (Equivalent data type)** list.
- **Load session** — loads a previously saved session so you can use the `typedefs` you defined earlier without reentering them.
- **Refresh list** — updates the list in **Typedefs name (Equivalent data type)**. Refreshing the list ensures the contents are current. If you changed your project data type content or loaded a new project, this updates the type definitions in the DTM.
- **Close** — closes the DTM and prompts you to save the session information. This is the only way to save your work in this dialog box. Saving the session creates a MATLAB file you can reload into the DTM later.

Adding Custom Type Definitions to MATLAB

Every custom type definition in your project must appear on the **Typedef name (Equivalent data type)** list for MATLAB to understand the data types involved. To add entries the list, use the **Add typedef** option to identify your type definition with a data type that MATLAB recognizes. When you click **Add typedef**, the **List of Known Data Types** dialog box opens, displaying the data types currently recognized by MATLAB. To make finding a specific type easier, the known data types are grouped into categories:

- MATLAB types
- TI C types
- TI fixed point types
- Struct, union, enum types
- Other (e.g. pointers, `typedefs`)

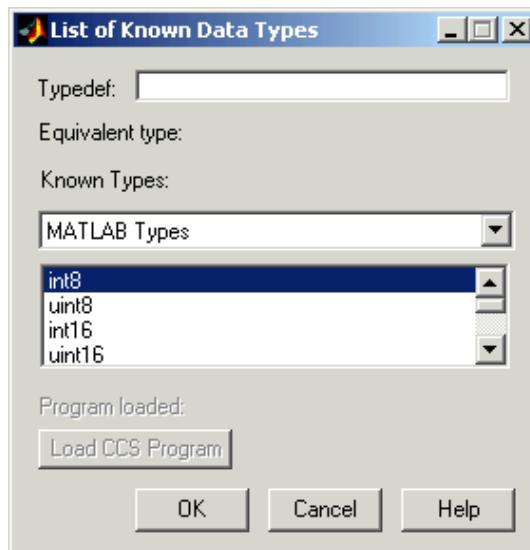
Each custom type definition added in the DTM becomes part of the `ticcs` object passed to the DTM in `datatypemanager(objectname)`. The list of data types in the object, both default and custom, is available by entering

```
objectname.type
```

at the command prompt.

The same list appears in the DTM on the **Typedef name (Equivalent data type)**

MATLAB uses the type definitions when you run a function residing on your processor from MATLAB.



To Add a Typedef to MATLAB

You use the DTM to add typedefs for MATLAB to recognize, such as:

- Typedefs that use a MATLAB data type in the type definition
- Typedefs that use an enumerated or union data type in the type definition
- Typedefs that use a structure in the type definition

- Typedefs that use pointers or typedefs in the type definition

To define custom data types that use structs, enums, or unions from a project, the project must be loaded on the processor before you add the custom type definitions. Either load the project and .out file before you start the DTM, or use the **Load Program** option in the DTM to load the .out file.

Note After a successful load process, you see the name of the file you loaded in **Loaded program**. Otherwise, you get an error message that the load failed.

Only programs that you load from this dialog box appear in **Program loaded**. Programs that are already loaded on your processor do not appear there because MATLAB cannot determine what program you have loaded.

You need to know the custom definitions you used so you can add them in the DTM. Use the options for list to verify whether you loaded a .out file on the processor.

Create an object and load a program.

- 1 Create a ticcs object.

```
cc=ticcs;
```

- 2 Load a program on your processor. For example, the MATLAB command

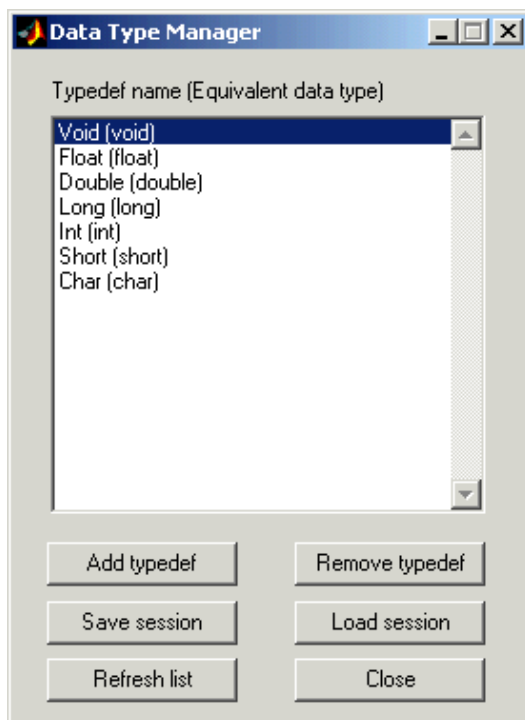
```
load(cc, 'c6711dskwdnoisf_c6000_rtwD\c6711dskwdnoisf.out');
```

loads the executable file from the model c6711dskwdnois.mdl on the processor.

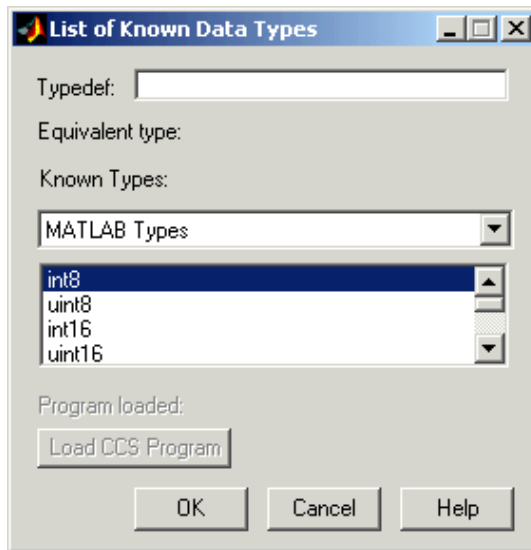
- 3 Start the DTM with the object you created.

```
datatypemanager(cc);
```

The DTM starts, showing the default data types.



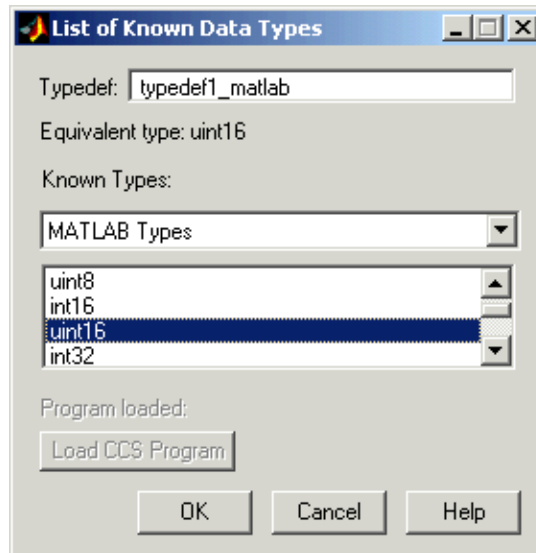
- 4 Click **Add typedef** to add your first custom data type. The List of Known Data Types dialog box appears as shown.



Add a MATLAB type definition.

- 5 In **Typedef**, enter the name of the typedef as you defined it in your code. For this example, use `typedef1_matlab`.

- 6 Select an appropriate MATLAB data type from the MATLAB Types in **Known Types**. `uint16` is the choice. Choose the data type that best represents the data type in your code.

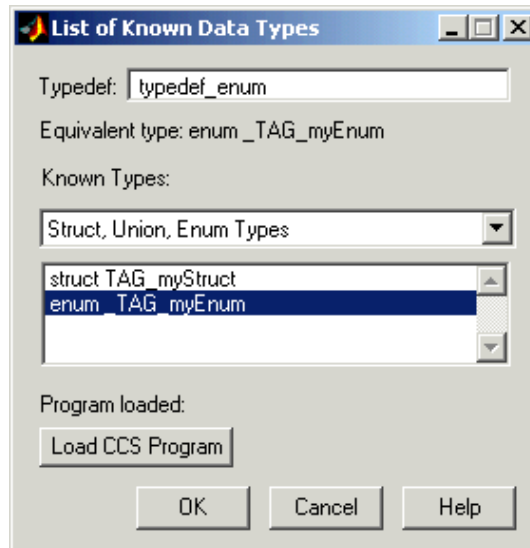


- 7 Click **OK** to close the dialog box and add the new type definition to the **Typedef name** list.

Add an enumerated type definition.

- 8 Click **Add Typedef**.
- 9 From the **Known Types** list, select Struct, Enum, Union Types.
- 10 To define your type definition, give it a name in **Typedef**, such as `typedef_enum`

- From the Struct, Enum, Union Types list, select the appropriate enumerated data type to use with `typedef_enum`. The `enum_TAG_myEnum` choice fills the enumerated type chosen.

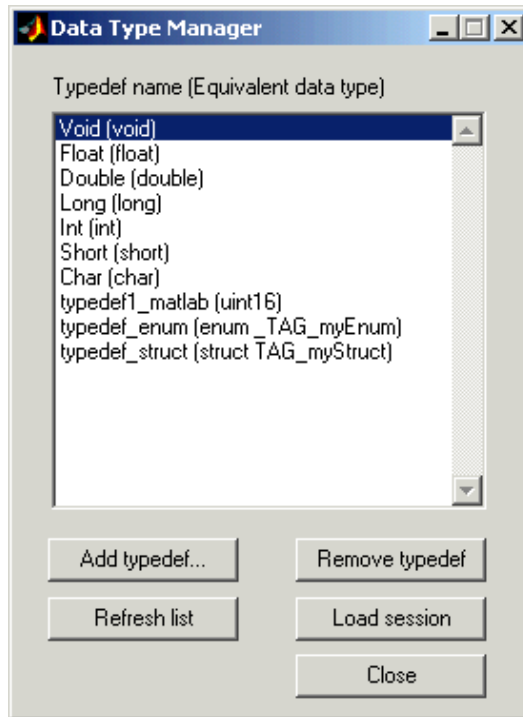


- Click **OK** to close the dialog box and add `typedef_enum` to your defined types that MATLAB software recognizes.

Add a structure typedef.

- Click **Add Typedef**.
- From the **Known Types** list, select Struct, Enum, Union Types.
- To define your type definition, give it a name in **Typedef**, such as `typedef_struct`.
- From the Struct, Enum, Union Types list, select the appropriate enumerated data type to use with `typedef_struct`. This example uses `struct_TAG_myStruct`.
- Click **OK** to close the dialog box and add the new data type to the list.

After you close the dialog box, the **Typedef name** list in the Data Type Manager looks like this.



To check the data types in the `cc` object, enter

```
cc.type
```

which returns

```
Defined types      : Void, Float, Double, Long, Int, Short, Char,  
typedef1_matlab, typedef_enum, typedef_struct
```

If your function declaration uses any of the types listed by `cc.type`, MATLAB software can interpret the data correctly. For example, MATLAB software interprets the `typedef1_matlab` data type as `uint16`.

Clicking **Close** in the DTM prompts you to save your session. Saving the session creates a MATLAB file that contains operations that create your final list of data types, identical to the data types in the **Typedef name** list.

The first line of the MATLAB file is a function definition, where the name of the function is the filename of the session you saved. In the stored MATLAB file, you find a function that includes add and remove operations that replicate the add and remove `typedef` operations you used to create the list of known data types in the DTM. For each time you added a `typedef` in the DTM, the MATLAB file contains an add command that adds the new type definition to the `type` property of the `cc` object. When you removed a data type, you created an equivalent `clear` command that removes the specified data type from the `type` property of the `cc` object.

All the operations you performed adding and removing data types in the DTM during the session are stored in the generated MATLAB file that you save, including mistakes you made while creating or removing type definitions. When you load your saved session into the DTM, you see the same error messages you saw, during the session. Keep in mind that you have already corrected these errors.

Project Generator

- “Introducing Project Generator” on page 3-2
- “Project Generation and Board Selection” on page 3-3
- “Project Generator Tutorial” on page 3-5
- “Model Reference” on page 3-14

Introducing Project Generator

Project generator provides the following features for developing project and generating code:

- Support automated project building for Texas Instruments' Code Composer Studio software that lets you create projects from code generated by Real-Time Workshop and Real-Time Workshop Embedded Coder products. The project automatically populates CCS projects in the CCS development environment.
- Configure code generation using model configuration parameters and processor preferences block options
- Select from two system target files to generate code specific to your processor
- Configure project build process
- Automatically download and run your generated projects on your processor

Note You cannot generate code for C6000 processors in big-endian mode. Code generation supports only little-endian processor data byte order.

Project Generation and Board Selection

Project Generator uses `ticcs` objects to connect to the IDE. Each time you build a model to generate a project, the build process starts by issuing the `ticcs` method, as shown here:

```
cc=ticcs('boardnum',boardnum,'procnum',procnum)
```

The software attempts to connect to the board (`boardnum`) and processor (`procnum`) associated with the **Board name** and **Processor number** parameters in the Target Preferences block in the model.

The result of the `ticcs` method changes, depending on the boards you configured in CCS. The following table describes how the software selects the board to connect to in your board configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio or Embedded IDE Link software not installed.	Returns an error message asking you to verify that you installed both Code Composer Studio and Embedded IDE Link properly.
Code Composer Studio software does not have any configured boards.	Returns an error message that the software could not find any boards in your configuration. Use Setup Code Composer Studio™ to configure at least one board.
Code Composer Studio software has one configured board.	Attaches to the board regardless of the name of the board supplied in the Target Preferences block. You see a warning message telling you which board the software selected.
Code Composer Studio software has one board configured that does not match the board name in the Target Preferences block. ^(*)	Returns a warning message that the software could not find the board specified in the block and connected to the board listed in the warning message. The software connects to the first board in your CCS configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio has more than one board configured. The board name specified in the Target Preferences block is one of the configured boards.	Connects to the specified board.
Code Composer Studio has more than one board configured. The board name specified in the Target Preferences block is not one of the configured boards. ^(*)	<p>Returns a message asking you to select a board from the list of configured boards. You have two choices:</p> <ul style="list-style-type: none"> • Select a board to use for project generation, and click OK. Your selection does not change the board specified in the Target Preferences block. The software connects to the selected board. • Click Abort to stop the project build and code generation process. The software does not connect to the IDE or board.

^(*)You may encounter the situation where you do not have the correct board configured in CCS because of one of the following conditions:

- You changed your board configuration after you added the Target Preferences block to a model and saved the model. When you reopen the model, the board specified in **Board name** in the block is no longer in your configuration.
- You are working with a model from a source whose board configuration is not the same as yours. The model includes a Target Preferences block.

Use `csboardinfo` at the MATLAB prompt to verify or review your configured boards.

Project Generator Tutorial

In this section...
“Creating the Model” on page 3-6
“Adding the Target Preferences Block to Your Model” on page 3-6
“Specify Configuration Parameters for Your Model” on page 3-10

In this tutorial you will use the Embedded IDE Link software to:

- Build a model.
- Generate a project from the model.
- Build the project and run the binary on a processor.

Note The model demonstrates project generation. You cannot not build and run the model on your processor without additional blocks.

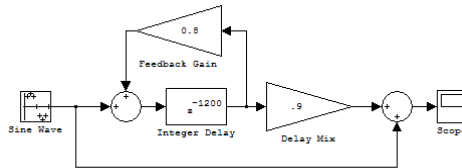
To generate a project from a model, complete the following tasks:

- 1** Create a model application.
- 2** Add a Target Preferences block from the Embedded IDE Link library to your model.
- 3** In the Target Preferences block, verify and set the block parameters for your hardware or simulator.
- 4** Set the configuration parameters for your model, including
 - Solver parameters such as simulation start and solver options
 - Real-Time Workshop software options such as processor configuration and processor compiler selection
- 5** Generate your project.
- 6** Review your project in CCS.

Creating the Model

To create the model for audio reverberation, follow these steps:

- 1 Start Simulink software.
- 2 Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.
- 3 Use Simulink blocks and Signal Processing Blockset™ blocks to create the following model.



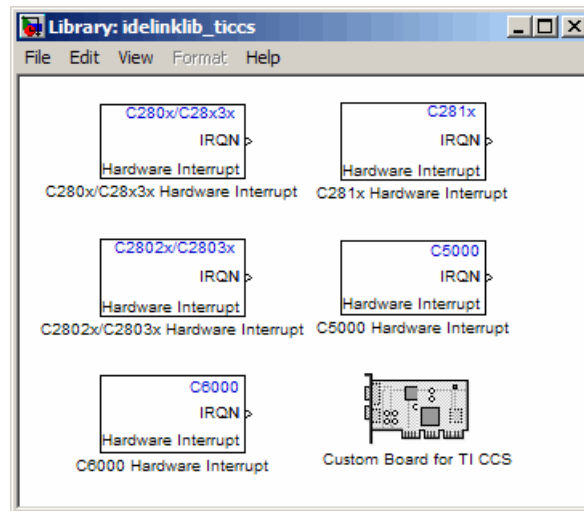
Look for the Integer Delay block in the Discrete library of Simulink blocks and the Gain block in the Commonly Used Blocks library. Do not add the Custom Board for TI CCS block at this time.

- 4 Save your model with a suitable name before continuing.

Adding the Target Preferences Block to Your Model

So that you can configure your model to work with TI processors, Embedded IDE Link supplies a Target Preferences/Custom Board block for Texas Instruments processors.

Entering `idelinklib_ticcs` at the MATLAB software prompt opens the block library. This block library is included in Embedded IDE Link `idelinklib` blockset in the Simulink Library browser.



Adding a Target Preferences block to a model triggers a dialog box that asks about your model configuration settings. The message tells you that the model configuration parameters will be set to default values based on the processor specified in the block parameters. To set the parameters automatically, click **Yes**. Clicking **No** dismisses the dialog box and does not set the parameters.

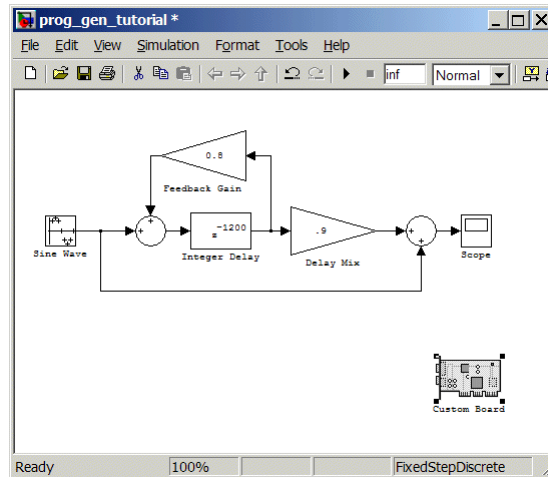
When you click **Yes**, the software sets the system target file to `ccslink_grt.tlc` or `ccslink_ert.tlc` and sets the hardware options and product-specific parameters in the model to default values. If you open the model Configuration Parameters, you see the Embedded IDE Link pane option on the select tree.

Clicking **No** prevents the software from setting the system target file and the product specific options. When you open the model Configuration Parameters for your model, you do not see the Embedded IDE Link pane option on the select tree. To enable the options, select the `ccslink_ert.tlc` or `ccslink_grt.tlc` system target file from the System Target File list in the Real-Time Workshop pane options.

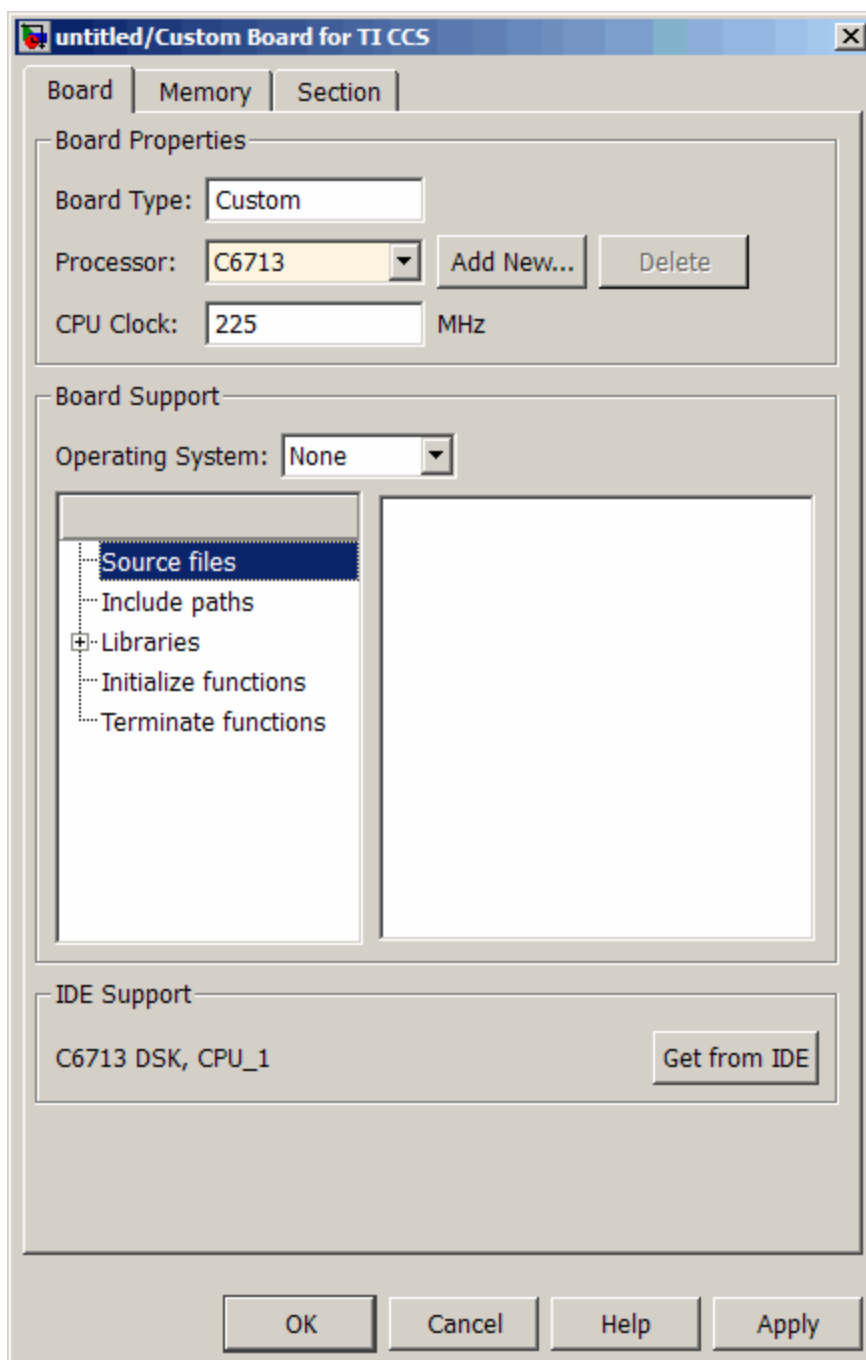
To add the Target Preferences block to your model, follow these steps:

- 1 Double-click Embedded IDE Link in the Simulink Library browser to open the `idelinklib` blockset.

- 2 Select **Supported IDEs > Texas Instruments Code Composer Studio** block library.
- 3 Drag and drop the Custom Board for TI CCS block to your model as shown in the following model window figure.



- 4 Double-click the Custom Board for TI CCS block in the model to open the block dialog box.



- 5 In the Block dialog box, select your processor from the **Processor** list.
- 6 Verify the **CPU clock** value and, if you are using a simulator, select **Simulator**.
- 7 Verify the settings on the **Memory** and **Sections** tabs to be sure they are correct for the processor you selected.
- 8 Click **OK** to close the Target Preferences dialog box.

You have completed the model. Now configure the model configuration parameters to generate a project in CCS IDE from your model.

Specify Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog box in Simulink software.

Setting Solver Parameters

After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

- 1 Open the Configuration Parameters dialog box and set the appropriate options on the **Solver** category for your model and for Embedded IDE Link.
 - Set **Start time** to 0.0 and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to `inf` for completeness.
 - Under **Solver options**, select the **fixed-step** and **discrete** settings from the lists
 - Set the **Fixed step size** to **Auto** and the **Tasking Mode** to **Single Tasking**

Note Generated code does not honor Simulink software stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, add a Stop Simulation block in your model.

When you use PIL, you can set the **Solver options** to any selection from the **Type** and **Solver** lists.

Ignore the Data Import/Export, Diagnostics, and Optimization categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Real-Time Workshop Code Generation Parameters

To configure Real-Time Workshop software to use the correct processor files and to compile and run your model executable file, set the options in the Real-Time Workshop category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the code generation options for your DSP:

- 1 Select Real-Time Workshop on the **Select** tree.
- 2 In Target selection, use the **Browse** button to set **System target file** to `ccslink_grt.tlc`.

Setting Embedded IDE Link Parameters

To configure Real-Time Workshop software to use the correct code generation options and to compile and run your model executable file, set the options in the Embedded IDE Link category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the code generation options for your processor:

- 1 From the **Select** tree, choose Embedded IDE Link to specify code generation options that apply to your processor.
- 2 Set the following options in the pane under **Project options**:
 - **Project options** should be Custom.

- Set **Compiler options string** and **Linker options string** should be blank.
- 3** Under **Link Automation**, verify that **Export IDE link handle to base workspace** is selected and provide a name for the handle in **IDE handle name** (optional).
 - 4** Set the following **Runtime** options:
 - **Build action:** `Build_and_execute`.
 - **Interrupt overrun notification method:** `None`.

You have configured the Real-Time Workshop software options that let you generate a project for you processor. You may have noticed that you did not configure a few categories on the **Select** tree, such as **Comments**, **Symbols**, and **Optimization**.

For your new model, the default values for the options in these categories are correct. For other models you develop, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code. Refer to your Simulink and Real-Time Workshop documentation for more information about setting the configuration parameters.

Building Your Project

After you set the configuration parameters and configure Real-Time Workshop software to create the files you need, you direct the build process to create your project:

- 1** Press **OK** to close the Configuration Parameters dialog box.
- 2** Click **Ctrl+B** to generate your project into CCS IDE.

When you click **Build** with `Create_project` selected for **Build action**, the automatic build process starts CCS IDE, populates a new project in the development environment, builds the project, loads the binary on the processor, and runs it.

- 3** To stop processor execution, use the **Halt** option in CCS or enter `cc.halt` at the MATLAB command prompt. (Where “cc” is the IDE handle name you specified previously in **Configuration Parameters**.)

Model Reference

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Real-Time Workshop documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models — Blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Real-Time Workshop documentation in the online Help system.

Model Reference in Simulation

When you simulate the top model, Real-Time Workshop software detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates

an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink software rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these settings through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation

Real-Time Workshop software requires executables to generate code from models. If you have not simulated your model at least once, Real-Time Workshop software creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, Real-Time Workshop software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

Using Model Reference

With few limitations or restrictions, Embedded IDE Link provides full support for generating code from models that use model reference.

Build Action Setting

The most important requirement for using model reference with the TI's processors is that you must set the **Build action** (go to **Configuration Parameters > Embedded IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action

- 1 Open your model.
- 2 Select **Simulation > Configuration Parameters** from the model menus.

The Configuration Parameters dialog box opens.

- 3** From the **Select** tree, choose **Embedded IDE Link**.
- 4** In the right pane, under **Runtime**, select set **Archive_library** from the **Build action** list.

If your top model uses a reference model that does not have the build action set to **Archive_library**, the build process automatically changes the build action to **Archive_library** and issues a warning about the change.

As a result of selecting the **Archive_library** setting, other options are disabled:

- DSP/BIOS is disabled for all referenced models. Only the top model supports DSP/BIOS operation.
- **Interrupt overrun notification method**, **Export IDE link handle to the base workspace**, and **System stack size** are disabled for the referenced models.

Target Preferences Blocks in Reference Models

Each referenced model and the top model must include a Target Preferences block for the correct processor. You must configure all the Target Preferences blocks for the same processor.

To obtain information about which compiler to use and which archiver to use to build the referenced models, the referenced models require Target Preferences blocks. Without them, the compile and archive processes does not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations

Model reference with Embedded IDE Link does not allow you to use certain blocks or S-functions in reference models:

- No blocks from the C62x DSP Library (in c6000lib) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (in c6000lib) (because these are noninlined S-functions)
- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Target Support Package™ or Target Support Package block library

Configuring processors to Use Model Reference

processors that you plan to use in Model Referencing must meet some general requirements.

- A model reference compatible processor must be derived from the ERT or GRT processors.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation processor.
- The External mode option is not supported in model reference Real-Time Workshop software processor builds. Embedded IDE Link does not support External mode. If you select this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in Supporting Shared Utility Directories in the Build Process in the Real-Time Workshop documentation.

To use an existing processor, or a new processor, with Model Reference, you set the `ModelReferenceCompliant` flag for the processor. For information on how to set this option, refer to `ModelReferenceCompliant` in the online Help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference processor, use the following command to set the `ModelReferenceCompliant` flag to On:

```
set_param(bdroot, 'ModelReferenceCompliant', 'on')
```

Models that you develop with versions 2.4 and later of Embedded IDE Link automatically include the model reference capability. You do not need to set the flag.

Exporting Filter Coefficients from FDATool

- “About FDATool” on page 4-2
- “Preparing to Export Filter Coefficients to Code Composer Studio Projects” on page 4-4
- “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 4-9
- “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 4-15

About FDATool

Signal Processing Toolbox™ software provides the Filter Design and Analysis tool (FDATool) that lets you design a filter and then export the filter coefficients to a matching filter implemented in a CCS project.

Using FDATool with CCS IDE enables you to:

- Design your filter in FDATool
- Use CCS to test your filter on a processor
- Redesign and optimize the filter in FDATool
- Test your redesigned filter on the processor

For instructions on using FDATool, refer to the section “Filter Design and Analysis Tool” in the Signal Processing Toolbox documentation.

Procedures in this chapter demonstrate how to use the FDATool export options to export filter coefficients to CCS. Using these procedures, you can perform the following tasks:

- Export filter coefficients from FDATool in a header file—“Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 4-9
- Export filter coefficients from FDATool to processor memory—“Replacing Existing Coefficients in Memory with Updated Coefficients” on page 4-16

Caution As a best practice, export coefficients in a header file for the most reliable results. Exporting coefficients directly to processor memory can generate unexpected results or corrupt memory.

Also see the reference pages for the following Embedded IDE Link functions. These primary functions allow you use to access variables and write them to processor memory from the MATLAB Command window.

- — Return the address of a symbol so you can read or write to it.

- `ticc` — Create a connection between MATLAB software and CCS IDE so you can work with the project in CCS from the MATLAB Command window.
- `write` — Write data to memory on the processor.

Preparing to Export Filter Coefficients to Code Composer Studio Projects

In this section...

“Features of a Filter” on page 4-4

“Selecting the Export Mode” on page 4-5

“Choosing the Export Data Type” on page 4-6

Features of a Filter

When you create a filter in FDATool, the filter includes defining features identified in the following table.

Defining Feature	Description
Structure	Structure defines how the elements of a digital filter—gains, adders/subtractors, and delays—combine to form the filter. See the Signal Processing Toolbox documentation in the Online Help system for more information about filter structures.
Design Method	Defines the mathematical algorithm used to determine the filter response, length, and coefficients.
Response Type and Specifications	Defines the filter passband shape, such as lowpass or bandpass, and the specifications for the passband.
Coefficients	Defines how the filter structure responds at each stage of the filter process.
Data Type	Defines how to represent the filter coefficients and the resulting filtered output. Whether your filter uses floating-point or fixed-point coefficients affects the filter response and output data values.

When you export your filter, FDATool exports only the number of and value of the filter coefficients and the data type used to define the coefficients.

Selecting the Export Mode

You can export a filter by generating an ANSI® C header file, or by writing the filter coefficients directly to processor memory. The following table summarizes when and how to use the export modes.

To...	Use Export Mode...	When to Use	Suggested Use
Add filter coefficients to a project in CCS	C header file	You implemented a filter algorithm in your program, but you did not allocate memory on your processor for the filter coefficients.	<ul style="list-style-type: none"> • Add the generated ANSI C header file to an appropriate project. Building and loading this project into your processor allocates static memory locations on the processor and writes your filter coefficients to those locations. • Edit the file so the header file allocates extra processor memory and then add the header file to your project. Refer to “Allocating Sufficient or Extra Memory for Filter Coefficients” on page 4-15 in the next section. <p>(For a sample generated header file, refer to “Reviewing ANSI C Header File Contents” on page 4-12.)</p>
Modify the filter coefficients in an embedded application loaded on a processor	Write directly to memory	You loaded a program on your processor. The program allocated space in your processor memory to store the filter coefficients.	<ul style="list-style-type: none"> • Optimize your filter design in FDATool. Then, • Write the updated filter coefficients directly to the allocated processor memory. Refer to section “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 4-15 for more information.

Choosing the Export Data Type

The export process provides two ways you can specify the data type to use to represent the filter coefficients. Select one of the options shown in the following table when you export your filter.

Specify Data Type for Export	Description
Export suggested	Uses the data type that FDATool suggests to preserve the fidelity of the filter coefficients and the performance of your filter in the project
Export as	Lets you specify the data type to use to export the filter coefficients

FDATool exports filter coefficients that use the following data types directly without modifications:

- Signed integer (8, 16, or 32 bits)
- Unsigned integer (8, 16, or 32 bits)
- Double-precision floating point (64 bits)
- Single-precision floating point (32 bits)

Filters in FDATool in the Signal Processing Toolbox software use double-precision floating point. You cannot change the data type.

If you have installed Filter Design Toolbox™ software, you can use the filter quantization options in FDATool to set the word and fraction lengths that represent your filter coefficients. For information about using the quantization options, refer to Filter Design and Analysis Tool in the Filter Design Toolbox documentation in the Online help system.

If your filter uses one of the supported data types, **Export suggested** specifies that data type.

If your filter does not use one of the supported data types, FDATool converts the unsupported data type to one of the supported types and then suggests that data type. For more information about how FDATool determines the data

type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 4-7.

Follow these best-practice guidelines when you implement your filter algorithm in source code and design your filter in FDATool:

- Implement your filter using one of the data types FDATool exports without modifications.
- Design your filter in FDATool using the data type you used to implement your filter.

To Choose the Export Data Type

When you export your filter, follow this procedure to select the export data type to ensure the exported filter coefficients closely match the coefficients of your filter in FDATool.

- 1** In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog box.
- 2** Perform one of the following actions:
 - Select **Export suggested** to export the coefficients in the suggested data type.
 - Select **Export as** and choose the data type your filter requires from the list.

Caution If you select **Export as**, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

How FDATool Determines the Export Suggested Data Type

By default, FDATool represents filter coefficients as double-precision floating-point data. When you export your filter coefficients, FDATool suggests the same data type.

If you set custom word and fraction lengths to represent your filter coefficients, the export process suggests a data type to maintain the best fidelity for the filter.

The export process converts your custom word and fraction lengths to a suggested export data type, using the following rules:

- Round the word length up to the nearest larger supported data type. For example, round an 18-bit word length up to 32 bits.
- Set the fraction length to maintain the same difference between the word and fraction length in the new data type as applies in the custom data type.

For example, if you specify a fixed-point data type with word length of 14 bits and fraction length of 11 bits, the export process suggests an integer data type with word length of 16 bits and fraction length of 13 bits, retaining the 3 bit difference.

Exporting Filter Coefficients to Your Code Composer Studio Project

In this section...

“Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 4-9

“Reviewing ANSI C Header File Contents” on page 4-12

Exporting Filter Coefficients from FDATool to the CCS IDE Editor

In this section, you export filter coefficients to a project by generating an ANSI C header file that contains the coefficients. The header file defines global arrays for the filter coefficients. When you compile and link the project to which you added the header file, the linker allocates the global arrays in static memory locations in processor memory.

Loading the executable file into your processor allocates enough memory to store the exported filter coefficients in processor memory and writes the coefficients to the allocated memory.

Use the following steps to export filter coefficients from FDATool to the CCS IDE text editor.

- 1 Start FDATool by entering `fdatool` at the MATLAB command prompt.

```
fdatool    % Starts FDATool.
```

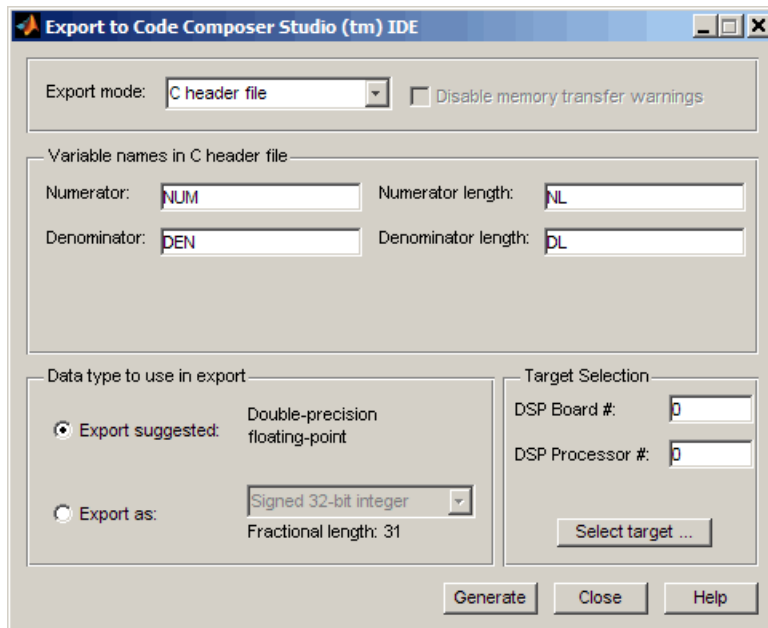
- 2 Design a filter with the same structure, length, design method, specifications, and data type you implemented in your source code filter algorithm.

The following figure shows a Direct-form II IIR filter example that uses second-order sections.

- 3 Click **Store Filter** to store your filter design. Storing the filter allows you to recall the design to modify it.

- 4 To export the filter coefficients, select **Targets > Code Composer Studio IDE** from the FDATool menu bar.

The Export to Code Composer Studio IDE dialog box opens, as shown in the following figure.



- 5 Set **Export mode** to C header file.



- 6 In **Variable names in C header file**, enter variable names for the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters where the coefficients will be stored.

The dialog box shows only the variables you need to export to define your filter.

Note You cannot use reserved ANSI C programming keywords, such as `if` or `int` as variable names, or include invalid characters such as spaces or semicolons (`;`).

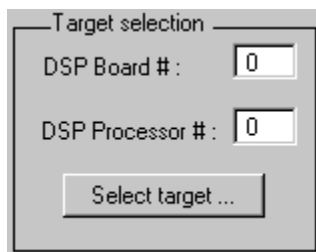
- 7** In **Data type to use in export**, select **Export suggested** to accept the recommended export data type. FDATool suggests a data type that retains filter coefficient fidelity.

You may find it useful to select the **Export as** option and select an export data type other than the one suggested.

Caution If you deviate from the suggested data type, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

For more information about how FDATool decides which data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 4-7.

- 8** If you know the board number and processor number of your DSP, enter **DSP Board #** and **DSP Processor #** values to identify your board.



The image shows a dialog box titled "Target selection". It contains two input fields: "DSP Board #" and "DSP Processor #", both with the value "0" entered. Below these fields is a button labeled "Select target ...".

When you have only one board or simulator, Embedded IDE Link software sets **DSP Board #** and **DSP Processor #** values for your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility: Embedded IDE Link dialog box.
 - From the list of boards and list of processors, select the board name and processor name to use.
 - Click **Done** to set the **DSP Board #** and **DSP Processor #** values.
- 9 Click **Generate** to generate the ANSI header file. FDATool prompts you for a file name and location to save the generated header file.

The default location to save the file is your MATLAB working folder. The default file name is `fdacoeffs.h`.

- 10 Click **OK** to export the header file to the CCS editor.

If CCS IDE is not open, this step starts the IDE.

The export process does not add the file to your active project in the IDE.

- 11 Drag your generated header file into the project that implements the filter.
- 12 Add a `#include` statement to your project source code to include the new header file when you build your project.
- 13 Generate a `.out` file and load the file into your processor. Loading the file allocates locations in static memory on the processor and writes the filter coefficients to those locations.

To see an example header file, refer to “Reviewing ANSI C Header File Contents” on page 4-12.

Reviewing ANSI C Header File Contents

The following program listing shows the exported header (`.h`) file that FDATool generates. This example shows a direct-form II filter that uses five second-order sections. The filter is stable and has linear phase.

Comments in the file describe the filter structure, number of sections, stability, and the phase of the filter. Source code shows the filter coefficients and variables associated with the filter design, such as the numerator length and the data type used to represent the coefficients.


```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 *
 * Generated by MATLAB(R) 7.8 and the Signal Processing Toolbox 6.11.
 *
 * Generated on: xx-xxx-xxxx 14:24:45
 *
 */

/*
 * Discrete-Time IIR Filter (real)
 * -----
 * Filter Structure   : Direct-Form II, Second-Order Sections
 * Number of Sections : 5
 * Stable             : Yes
 * Linear Phase       : No
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"

/*
 * Expected path to tmwtypes.h
 * $MATLABROOT\extern\include\tmwtypes.h
 */
#define MWSPT_NSEC 11
const int NL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const real164_T NUM[MWSPT_NSEC][3] = {
    {
        0.802536131462,          0,          0
    },
    {
        0.2642710234701,    0.5285420469403,    0.2642710234701
    },
    {
        1,          0,          0
    },
    {
        0.1743690465012,    0.3487380930024,    0.1743690465012
    },
};
```

```
    {
      0.2436793028081,  0.4873586056161,  0.2436793028081
    },
    {
      1, 0, 0
    },
    {
      0.3768793219093,  0.7537586438185,  0.3768793219093
    },
    {
      1, 0, 0
    }
};
const int DL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const rea164_T DEN[MWSPT_NSEC][3] = {
  {
    1, 0, 0
  },
  {
    1, -0.1842138030775,  0.1775781189277
  },
  {
    1, 0, 0
  },
  {
    1, -0.2160098642842,  0.3808329528195
  },
  {
    1, 0, 0
  }
};
```

Preventing Memory Corruption When You Export Coefficients to Processor Memory

In this section...

“Allocating Sufficient or Extra Memory for Filter Coefficients” on page 4-15

“Example: Using the Exported Header File to Allocate Extra Processor Memory” on page 4-15

“Replacing Existing Coefficients in Memory with Updated Coefficients” on page 4-16

“Example: Changing Filter Coefficients Stored on Your Processor” on page 4-17

Allocating Sufficient or Extra Memory for Filter Coefficients

You can allocate extra memory by editing the generated ANSI C header file. You can then load the associated program file into your processor as described in “Example: Using the Exported Header File to Allocate Extra Processor Memory” on page 4-15. Extra memory lets you change filter coefficients and overwrite existing coefficients stored in processor memory more easily.

To prevent problems when you update filter coefficients in a project, , such as writing coefficients to unintended memory locations, use the `C header file export mode` option in FDATool to update filter coefficients in your program.

Example: Using the Exported Header File to Allocate Extra Processor Memory

You can edit the generated header file so the linked program file allocates extra processor memory. By allocating extra memory, you avoid the problem of insufficient memory when you export new coefficients directly to allocated memory.

For example, changing the following command in the header file:

```
const real64_T NUM[47] = {...}
```

to

```
real164_T NUM[256] = {...}
```

allocates enough memory for NUM to store up to 256 numerator filter coefficients rather than 47.

Exporting the header file to CCS IDE does not add the filter to your project. To incorporate the filter coefficients from the header file, add a `#include` statement:

```
#include "headerfilename.h"
```

Refer to “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 4-9 for information about generating a header file to export filter coefficients.

When you export filter coefficients directly to processor memory, the export process writes coefficients to as many memory locations as they need. The write process does not perform bounds checking. To ensure you write to the correct locations, and have enough memory for your filter coefficients, plan memory allocation carefully.

Replacing Existing Coefficients in Memory with Updated Coefficients

When you redesign a filter and export new coefficients to replace existing coefficients in memory, verify the following conditions for your new design:

- Your redesign did not increase the memory required to store the coefficients beyond the allocated memory.

Changes that increase the memory required to store the filter coefficients include the following redesigns:

- Increasing the filter order
 - Changing the number of sections in the filter
 - Changing the numerical precision (changing the export data type)
- Your changes did not change the export data type.

Caution Identify changes that require additional memory to store the coefficients before you begin your export. Otherwise, exporting the new filter coefficients may overwrite data in memory locations you did not allocate for storing coefficients. Also, exporting filter coefficients to memory after you change the filter order, structure, design algorithm, or data type can yield unexpected results and corrupt memory.

Changing the filter design algorithm in FDATool, such as changing from Butterworth to Maximally Flat, often changes the number of filter coefficients (the filter order), the number of sections, or both. Also, the coefficients from the new design algorithm may not perform properly with your source code filter implementation.

If you change the design algorithm, verify that your filter structure and length are the same after you redesign your filter, and that the coefficients will perform properly with the filter you implemented.

If you change the number of sections or the filter order, your filter will not perform properly unless your filter algorithm implementation accommodates the changes.

Example: Changing Filter Coefficients Stored on Your Processor

This example writes filter coefficients to processor memory to replace the existing coefficients. To perform this process, you need the names of the variables in which your project stores the filter data.

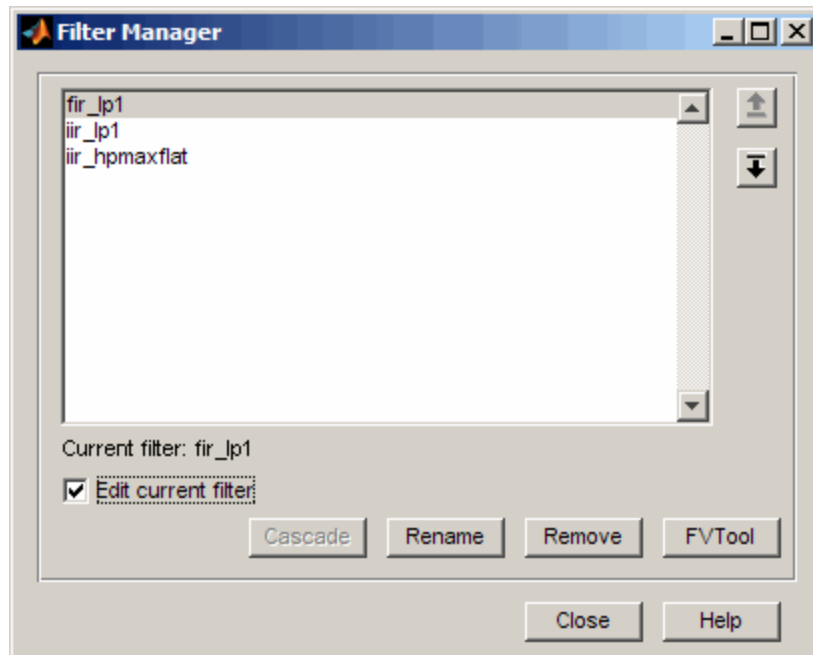
Before you export coefficients directly to memory, verify that your project allocated enough memory for the new filter coefficients. If your project allocated enough memory, you can modify your filter in FDATool and then follow the steps in this example to export the updated filter coefficients to the allocated memory.

If your new filter requires additional memory space, use a C header file to allocate memory on the processor and export the new coefficients as described in “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 4-9.

For important guidelines on writing directly to processor memory, refer to “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 4-15.

Follow these steps to export filter coefficients from FDATool directly to memory on your processor.

- 1 Load the program file that contains your filter into CCS IDE to activate the program symbol table. The symbol must contain the global variables you use to store the filter coefficients and length parameters.
- 2 Start FDATool.
- 3 Click **Filter Manager** to open the Filter Manager dialog box, shown in the following figure.



- 4 Highlight the filter to modify on the list of filters, and select **Edit current filter**. The highlighted filter appears in FDATool for you to change.

If you did not store your filter from a previous session, design the filter in FDATool and continue.

- 5 Click **Close** to dismiss the Filter Manager dialog box.
- 6 Adjust the filter specifications in FDATool to modify its performance.
- 7 In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog box.

Keep the export dialog box open while you work. When you do so, the contents update as you change the filter in FDATool.

Tip Click **Generate** to export coefficients to the same processor memory location multiple times without reentering variable names.

- 8 In the Export to Code Composer Studio dialog box:
 - Set **Export mode** to Write directly to memory
 - Clear **Disable memory transfer warnings** to get a warning if your processor does not support the export data type.
- 9 In **Variable names in target symbol table**, enter the names of the variables in the processor symbol table that correspond to the memory allocated for the parameters, such as **Numerator** and **Denominator**. Your names must match the names of the filter coefficient variables in your program.

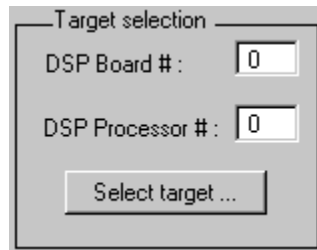
Variable names in target symbol table

Numerator:	<input type="text" value="NUM"/>	Numerator length:	<input type="text" value="NL"/>
Denominator:	<input type="text" value="DEN"/>	Denominator length:	<input type="text" value="DL"/>
Number of sections:	<input type="text" value="NS"/>		

- 10 Select **Export suggested** to accept the recommended export data type.

For more information about how FDATool determines the data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 4-7.

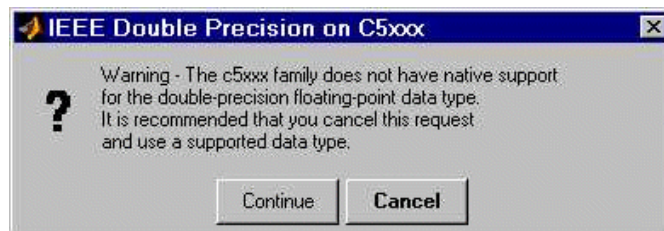
- 11 If you know the board number and processor number of your DSP, enter **DSP Board #** and **DSP Processor #** values to identify your board.



Note When you have only one board or simulator, Embedded IDE Link sets **DSP Board #** and **DSP Processor #** to your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility: Embedded IDE Link dialog box.
 - Select the board name and processor name to use from the list of boards.
- 12 Click **Generate** to export your filter. If your processor does not support the data type you export, you see a warning similar to the following message.



You can continue to export the filter, or cancel the export process. To prevent this warning dialog box from appearing, select **Disable memory transfer warnings** in the Export to Code Composer Studio IDE dialog box.

- 13** (Optional) Continue to optimize filter performance by modifying your filter in FDATool and then export the updated filter coefficients directly to processor memory.
- 14** When you finish testing your filter, return to FDATool, and click **Store filter** to save your changes.

Block Reference

Block Library: idelinklib_ticcs

C280x/C2802x/C2803x/C28x3x
Hardware Interrupt

Interrupt Service Routine to
handle hardware interrupt on
C280x/C28x3x processors

C281x Hardware Interrupt

Interrupt Service Routine to handle
hardware interrupt

C5000/C6000 Hardware Interrupt

Interrupt Service Routine to handle
hardware interrupt on C5000 and
C6000 processors

Block Library: idelinklib_common

Blocks — Alphabetical List

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

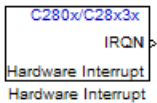
Purpose

Interrupt Service Routine to handle hardware interrupt on C280x/C28x3x processors

Library

Embedded IDE Link for TI Code Composer Studio (idelinklib_ticcs)

Description



For many systems, an execution scheduling model based on a timer interrupt is not sufficient to ensure a real-time response to external events. The C280x/C28x3x Hardware Interrupt block addresses this problem by allowing asynchronous processing of interrupts triggered by events managed by other blocks in the C280x/C28x3x DSP Chip Support Library.

The following C280x/C28x3x blocks that can generate an interrupt for asynchronous processing are available in Target Support Package.

- C280x ADC
- C280x eCAN Receive
- C280x SCI Receive
- C280x SCI Transmit
- C280x SPI Receive
- C280x SPI Transmit

Only one Hardware Interrupt block can be used in a model. To handle multiple interrupts, place a Demux block at the output of the Hardware Interrupt block to direct function calls to the appropriate function-call subsystems.

Vectorized Output

The output of this block is a function call. The size of the function call line equals the number of interrupts the block is set to handle. Each interrupt is represented by four parameters shown on the dialog box of the block. These parameters are a set of four vectors of equal length. Each interrupt is represented by one element from each parameter (four elements total), one from the same position in each of these vectors.

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

Each interrupt is described by:

- CPU interrupt numbers
- PIE interrupt numbers
- Task priorities
- Preemption flags

So one interrupt is described by a CPU interrupt number, a PIE interrupt number, a task priority, and a preemption flag.

The CPU and PIE interrupt numbers together uniquely specify a single interrupt for a single peripheral or peripheral module. Locate the “PIE MUXed Peripheral Interrupt Vector Table” in the following Texas Instruments documents for your processor:

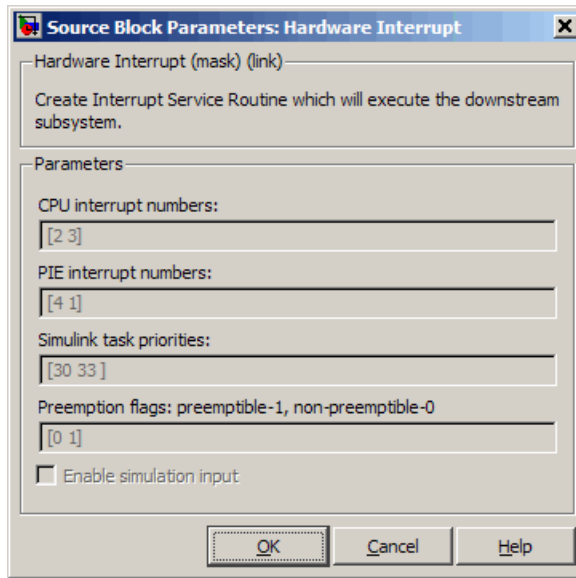
Processor	Literature Number at ti.com
280x and 28044	SPRU712
C2833x	SPRUFB0
C2802x	SPRUFN3
C2803x	SPRUGL8

The task priority indicates the relative importance tasks associated with the asynchronous interrupts. If an interrupt triggers a higher-priority task while a lower-priority task is running, the execution of the lower-priority task will be suspended while the higher-priority task is executed. The lowest value represents the highest priority. The default priority value of the base rate task is 40, so the priority value for each asynchronously triggered task must be less than 40 for these tasks to suspend the base rate task.

The preemption flag determines whether a given interrupt is preemptable. Preemption overrides prioritization, such that a preemptable task of higher priority can be preempted by a non-preemptable task of lower priority.

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

Dialog Box



CPU interrupt numbers

Enter a vector of CPU interrupt numbers for the interrupts you want to process asynchronously.

See the table of C280x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

PIE interrupt numbers

Enter a vector of PIE interrupt numbers for the interrupts you want to process asynchronously.

See the table of C280x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

Simulink task priorities

Enter a vector of task priorities for the interrupts you want to process asynchronously.

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

See the discussion of this block's "Vectorized Output" on page 6-2 for an explanation of task priorities.

Preemption flags

Enter a vector of preemption flags for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 6-2 for an explanation of preemption flags.

Enable simulation input

Select this check box if you want to be able to test asynchronous interrupt processing in the context of your Simulink software model.

Note Select this check box to enable you to test asynchronous interrupt processing behavior in Simulink software.

References

Detailed information about interrupt processing is in *TMS320x280x DSP System Control and Interrupts Reference Guide*, Literature Number SPRU712B, available at the Texas Instruments Web site.

See Also

The following links refer to block reference pages that require the Target Support Package software.

C280x/C2802x/C2803x/C28x3x Software Interrupt Trigger, Idle Task

C281x Hardware Interrupt

Purpose Interrupt Service Routine to handle hardware interrupt

Library Embedded IDE Link for TI Code Composer Studio (idelinklib_ticcs)

Description



For many systems, an execution scheduling model based on a timer interrupt is not sufficient to ensure a real-time response to external events. The C281x Hardware Interrupt block addresses this problem by allowing for the asynchronous processing of interrupts triggered by events managed by other blocks in the C281x DSP Chip Support Library.

The following C281x blocks that can generate an interrupt for asynchronous processing are available from Target Support Package:

- C281x ADC
- C281x CAP
- C281x eCAN Receive
- C281x Timer
- C281x SCI Receive
- C281x SCI Transmit
- C281x SPI Receive
- C281x SPI Transmit

Only one Hardware Interrupt block can be used in a model. To handle multiple interrupts, place a Demux block at the output of the Hardware Interrupt block to direct function calls to the appropriate function-call subsystems.

Vectorized Output

The output of this block is a function call. The size of the function call line equals the number of interrupts the block is set to handle. Each interrupt is represented by four parameters shown on the dialog box of the block. These parameters are a set of four vectors of equal length.

Each interrupt is represented by one element from each parameter (four elements total), one from the same position in each of these vectors.

Each interrupt is described by:

- CPU interrupt numbers
- PIE interrupt numbers
- Task priorities
- Preemption flags

So one interrupt is described by a CPU interrupt number, a PIE interrupt number, a task priority, and a preemption flag.

The CPU and PIE interrupt numbers together uniquely specify a single interrupt for a single peripheral or peripheral module. The following table maps CPU and PIE interrupt numbers to these peripheral interrupts.

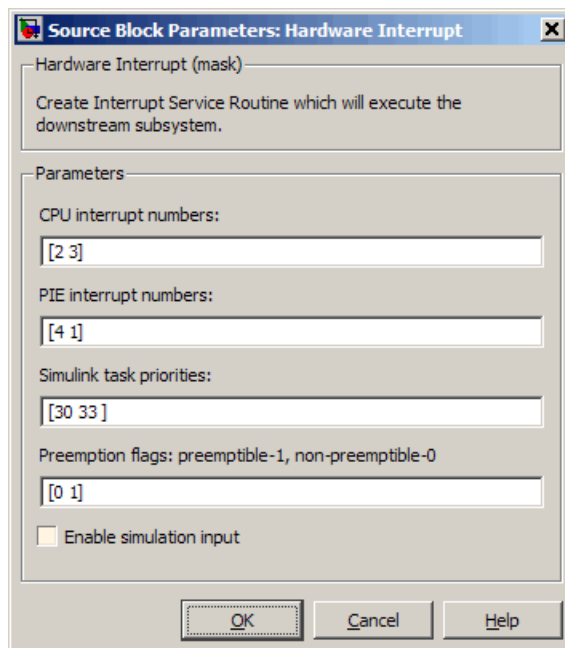
C281x Peripheral Interrupt Vector Values

Row numbers = CPU values / Column numbers = PIE values								
	8	7	6	5	4	3	2	1
1	WAKEINT (LPM/WMD)	TINT0 (TIMER 0)	ADCINT (ADC)	XINT2	XINT1	Reserved	PDPINTB (EV-B)	PDPINTA (EV-A)
2	Reserved	T1OFINT (EV-A)	T1UFINT (EV-A)	T1CINT (EV-A)	T1PINT (EV-A)	CMP3INT (EV-A)	CMP2INT (EV-A)	CMP1INT (EV-A)
3	Reserved	CAPINT3 (EV-A)	CAPINT2 (EV-A)	CAPINT1 (EV-A)	T2OFINT (EV-A)	T2UFINT (EV-A)	T2CINT (EV-A)	T2PINT (EV-A)
4	Reserved	T3OFINT (EV-B)	T3UFINT (EV-B)	T3CINT (EV-B)	T3PINT (EV-B)	CMP6INT (EV-B)	CMP5INT (EV-B)	CMP4INT (EV-B)
5	Reserved	CAPINT6 (EV-B)	CAPINT5 (EV-B)	CAPINT4 (EV-B)	T4OFINT (EV-B)	T4UFINT (EV-B)	T4CINT (EV-B)	T4PINT (EV-B)
6	Reserved	Reserved	MXINT (McBSP)	MRINT (McBSP)	Reserved	Reserved	SPITXINTA (SPI)	SPIRXINTA (SPI)
7	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
8	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
9	Reserved	Reserved	ECAN1INT (CAN)	ECAN0INT (CAN)	SCITXINTB (SCI-B)	SCIRXINTB (SCI-B)	SCITXINTA (SCI-A)	SCIRXINTA (SCI-A)
10	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
11	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
12	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

The task priority indicates the relative importance tasks associated with the asynchronous interrupts. If an interrupt triggers a higher-priority task while a lower-priority task is running, the execution of the lower-priority task will be suspended while the higher-priority task is executed. The lowest value represents the highest priority. Note that the default priority value of the base rate task is 40, so the priority value for each asynchronously triggered task must be less than 40 for these tasks to actually cause the suspension of the base rate task.

The preemption flag determines whether a given interrupt is preemptable or not. Preemption overrides prioritization, such that a preemptable task of higher priority can be preempted by a non-preemptable task of lower priority.

Dialog Box



C281x Hardware Interrupt

CPU interrupt numbers

Enter a vector of CPU interrupt numbers for the interrupts you want to process asynchronously.

See the table of C281x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

PIE interrupt numbers

Enter a vector of PIE interrupt numbers for the interrupts you want to process asynchronously.

See the table of C281x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

Simulink task priorities

Enter a vector of task priorities for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 6-6 for an explanation of task priorities.

Preemption flags

Enter a vector of preemption flags for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 6-6 for an explanation of preemption flags.

Enable simulation input

Select this check box if you want to be able to test asynchronous interrupt processing in the context of your Simulink software model.

Note Use this check box to enable you to test asynchronous interrupt processing behavior in Simulink software.

References

Detailed information interrupt processing is in *TMS320x281x DSP System Control and Interrupts Reference Guide*, Literature Number SPRU078C, available at the Texas Instruments Web site.

See Also

The following links to block reference pages require that Target Support Package is installed.

C281x Software Interrupt Trigger, C281x Timer, Idle Task

C5000/C6000 Hardware Interrupt

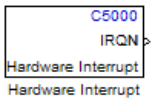
Purpose

Interrupt Service Routine to handle hardware interrupt on C5000 and C6000 processors

Library

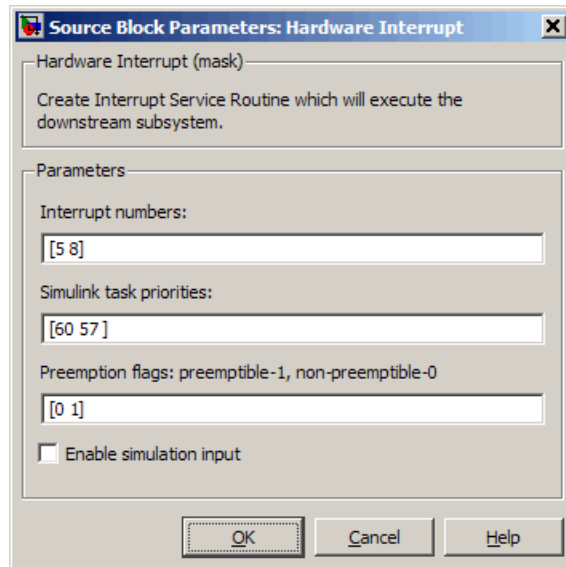
Embedded IDE Link for TI Code Composer Studio (idelinklib_ticcs)

Description



Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from the this block or a Task block connected to this block.

Dialog Box



Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The following table provides the valid range for C5xxx and C6xxx processors:

C5000/C6000 Hardware Interrupt

Processor Family	Valid Interrupt Numbers
C5xxx	2, 3, 5-21, 23
C6xxx	4-15

The width of the block output signal corresponds to the number of interrupt numbers specified here. Combined with the **Simulink task priorities** that you enter and the preemption flag you enter for each interrupt, these three values define how the code and processor handle interrupts during asynchronous scheduler operations.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink software task priority specifies the Simulink priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Simulink task priority values are required to generate the proper rate transition code (refer to Rate Transitions and Asynchronous Blocks). The task priority values are also required to ensure absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. Typically, you assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags preemptable – 1, non-preemptable – 0

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of

C5000/C6000 Hardware Interrupt

the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 5 in **Interrupt numbers** is not preemptible and the priority 8 interrupt can be preempted.

Enable simulation input

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Supported Processors

This appendix provides the details about the processors, simulators, and software that work with Embedded IDE Link.

- “Supported Platforms” on page A-2
- “Supported Versions of Code Composer Studio” on page A-5

Supported Platforms

In this section...
“Product Features Supported by Each Processor or Family” on page A-2
“Coemulation Support” on page A-3
“Supported Processors and Simulators” on page A-3
“Custom Board Support” on page A-4

This appendix lists the processors and simulators that work with the latest released version of Embedded IDE Link. Generally, the product supports boards and simulators from a given processor family. In some cases, only the simulators work, as noted in the tables in the next sections.

Product Features Supported by Each Processor or Family

The following table indicates which Embedded IDE Link features are available by processor family.

Features by Processor Family

Automation Interface Component			Project Generator Component	Verification	
Processor Family	Debug Mode	RTDX	Code Generation	Processor-in-the-Loop	Real-Time Execution Profiling
C28xx	Yes	Yes	Yes	Yes	Yes
C54xx	Yes	No	No	No	No
C55xx	Yes	Yes	Yes	Yes	Yes
C62xx	Yes	No	Yes	Yes	Yes
C64x and C64x+	Yes	No	Yes	Yes	Yes

Features by Processor Family (Continued)

Automation Interface Component			Project Generator Component	Verification	
Processor Family	Debug Mode	RTDX	Code Generation	Processor-in-the-Loop	Real-Time Execution Profiling
C67x and C67x+	Yes	No	Yes	Yes	Yes
DM64x	Yes	No	Yes	Yes	Yes
DM643x	Yes	No	Yes	Yes	Yes
TMS470R1x	Yes	No	No	No	No
TMS470R2x	Yes	No	No	No	No

Coemulation Support

An added feature for OMAP processors is coemulation for the two processors that comprise the OMAP. Embedded IDE Link supports coemulation or direct multiprocessor support for the TMS470R2x (TI-enhanced ARM925) and TMS320C55x DSP in OMAP 1510 and OMAP 5910.

Supported Processors and Simulators

Embedded IDE Link for has been tested on the following processors and boards produced by TI and others.

- TMS320C2000
 - Simulators (C28x)
 - C2808 eZdsp, C2812 eZdsp, C2833x Floating-Point Processors
- TMS320C5000
 - Simulators (C54x, C55x)
 - C5510 DSK
 - C5416 DSK, C5402 DSK

- TMS320C6000
 - Simulators (C62x, C64x, C67x)
 - C6713 DSK, C6711 DSK, C6701 EVM
 - C6416 DSK
 - DM64x
 - DM643x
 - C6211 DSK
- OMAP
 - OMAP 1510
 - OMAP 5910
- TMS470Rxx
 - Boards and simulators based on the TMS470R1x processor
 - Boards and simulators based on the TMS470R2x processor

Custom Board Support

You can use Embedded IDE Link with your custom board if:

- It uses one or more of the supported processors in the preceding list or if it is in the **Processor** list of the Target Preferences/Custom Board block for your processor family.
- You are able to use Code Composer Studio IDE to interact with your board/processor combination.

you should be able to use Embedded IDE Link with your hardware.

Supported Versions of Code Composer Studio

The following table lists versions of Embedded IDE Link and the versions of Code Composer Studio they support.

Embedded IDE Link Version	MATLAB Release	Supported Version of Code Composer Studio
4.0	R2009b	CCS 3.3 for C64x+, C6000, C5000, C2000, OMAP processors (tested on CCS 3.3 SR10)
3.4	R2009a	CCS 3.3 for C64x+, C6000, C5000, C2000, OMAP processors
3.3	R2008b	Only CCS 3.3 with DSP/BIOS 5.32.01 or 5.32.05 (not 5.32.00) (C64x+, C6000, C5000, C2000, OMAP) CCS 3.3 SR7 has a bug and is not supported
3.2	R2008a	Only CCS 3.3 with DSP/BIOS 5.3 (not 5.32.00)
3.1	R2007b	Only CCS 3.3 with DSP/BIOS 5.3
3.0	R2007a	<ul style="list-style-type: none"> • CCS 3.2 for C64x+ processors • CCS 3.1 for C2000, C5000, C6000, and OMAP processors
2.1	R2006b	<ul style="list-style-type: none"> • CCS 3.2 for C64x+ processors • CCS 3.1 for C2000, C5000, C6000, and OMAP processors
2.0	R2006a+	CCS 3.1 for C2000, C5000, C6000, and OMAP processors
1.5	R2006a	CCS 3.1 for C2000, C5000, C6000, and OMAP processors
1.4.2	R14SP3	<ul style="list-style-type: none"> • CCS 3.0 for C6000 processors • CCS 2.2 for C2000, C5000, C6000, and OMAP processors

Embedded IDE Link Version	MATLAB Release	Supported Version of Code Composer Studio
1.4.1	R14SP2	<ul style="list-style-type: none">• CCS 3.0 for C6000 processors• CCS 2.2 for C2000, C5000, C6000, and OMAP processors
1.4	R14SP1+	<ul style="list-style-type: none">• CCS 3.0 for C6000 processors• CCS 2.2 for C2000, C5000, C6000, and OMAP processors
1.3.2	R14SP1	<ul style="list-style-type: none">• CCS 2.2 for C2000, C5000, C6000, and OMAP processors• CCS 2.12 for C2000, C5000, C6000, and OMAP processors
1.3.1	R14	<ul style="list-style-type: none">• CCS 2.2 for C2000, C5000, C6000, and OMAP processors• CCS 2.12 for C2000, C5000, C6000, and OMAP processors
1.3	R13SP1+	CCS 2.12 for C2000, C5000, C6000, and OMAP processors

Reported Limitations and Tips

Reported Issues

In this section...

“Demonstration Programs Do Not Run Properly Without Correct GEL Files” on page B-3

“Error Accessing type Property of ticcs Object Having Size Greater Than 1” on page B-3

“Changing Values of Local Variables Does Not Take Effect” on page B-4

“Code Composer Studio Cannot Find a File After You Halt a Program” on page B-4

“C54x XPC Register Can Be Modified Only Through the PC Register” on page B-6

“Working with More Than One Installed Version of Code Composer Studio” on page B-6

“Changing CCS Versions During a MATLAB Session” on page B-7

“MATLAB Hangs When Code Composer Studio Cannot Find a Board” on page B-7

“Using Mapped Drives” on page B-9

“Uninstalling Code Composer Studio 3.3 Prevents Embedded IDE Link From Connecting” on page B-9

“PostCodeGenCommand Commands Do Not Affect Embedded IDE Link Projects” on page B-10

Some long-standing issues affect the Embedded IDE Link product. When you are using `ticcs` objects and the software methods to work with Code Composer Studio and supported hardware or simulators, recall the information in this section.

The latest issues in the list appear at the bottom. HIL refers to “hardware in the loop,” also called processor in the loop (PIL) here and in other applications, and sometimes referred to as function calls.

Demonstration Programs Do Not Run Properly Without Correct GEL Files

To run the Embedded IDE Link demos, you must load the appropriate GEL files before you run the demos. For some boards, the demos run fine with the default CCS GEL file. Some boards need to run device-specific GEL files for the demos to work correctly.

Here are demos and boards which require specific GEL files.

- Board: C5416 DSK
Demos: `rtdxtutorial`, `rtdx1msdemo`
Emulator: XDS-510
GEL file to load: `c5416_dsk.gel`

In general, if a demo does not run correctly with the default GEL file, try using a device-specific GEL file by defining the file in the CCS Setup Utility.

Error Accessing type Property of ticcs Object Having Size Greater Than 1

When `cc` is a `ticcs` object consisting of an array of single `ticcs` objects such that

```
cc
Array of TICCS Objects:
  API version : 1.2
  Board name  : C54x Simulator (Texas Instruments)
  Board number : 0
  Processor 0 (element 1) : TMS320C5407 (CPU, Not Running)
  Processor 0 (element 2) : TMS320C5407 (CPU, Not Running)
```

you cannot use `cc` to access the type object. The example syntaxes below generate errors.

- `cc.type`
- `add(cc.type, 'mytypedef', 'int')`

To access `type` without the error, reference the individual elements of `cc` as follows:

- `cc(1).type`
- `add(cc(2).type, 'mytypedef', 'int')`

Changing Values of Local Variables Does Not Take Effect

If you halt the execution of your program on your DSP and modify a local variable's value, the new value may not be acknowledged by the compiler. If you continue to run your program, the compiler uses the original value of the variable.

This problem happens only with local variables. When you write to the local variable via the Code Composer Studio Watch Window or via a MATLAB object, you are writing into the variable's absolute location (register or address in memory).

However, within the processor function, the compiler sometimes saves the local variable's values in an intermediate location, such as in another register or to the stack. That intermediate location cannot be determined or changed/updated with a new value during execution. Thus the compiler uses the old, unchanged variable value from the intermediate location.

Code Composer Studio Cannot Find a File After You Halt a Program

When you halt a running program on your processor, Code Composer Studio may display a dialog box that says it cannot find a source code file or a library file.

When you halt a program, CCS tries to display the source code associated with the current program counter. If the program stops in a system library like the runtime library, DSP/BIOS, or the board support library, it cannot find the source code for debug. You can either find the source code to debug it or select the **Don't show this message again** checkbox to ignore messages like this in the future.

For more information about how CCS responds to the halt, refer the online Help for CCS. In the online help system, use the search engine to search for the keywords “Troubleshooting” and “Support.” The following information comes from the online help for CCS, starting with the error message:

File Not Found

The debugger is unable to locate the source file necessary to enable source-level debugging for this program.

To specify the location of the source file

- 1** Click **Yes**. The Open dialog box appears.
- 2** In the Open dialog box, specify the location and name of the source file then click **Open**.

The next section provides more details about file paths.

Defining a Search Path for Source Files

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

To Specify Search Path Directories

- 1** Select **Option > Customize**.
- 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.

The Directories dialog box 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 click **browse [...]** to navigate 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.

3 Click **OK** to close the **Customize** dialog box and save your changes.

C54x XPC Register Can Be Modified Only Through the PC Register

You cannot modify the XPC register value directly using `regwrite` to write into the register. When you are using extended program addressing in C54x, you can modify the XPC register by using `regwrite` to write a 23-bit data value in the PC register. Along with the 16-bit PC register, this operation also modifies the 7-bit XPC register that is used for extended program addressing. On the C54x, the PC register is 23 bits (7 bits in XPC + 16 bits in PC).

You can then read the XPC register value using `regread`.

Working with More Than One Installed Version of Code Composer Studio

When you have more than one version of Code Composer Studio installed on your machine, you cannot select which CCS version MATLAB Embedded IDE Link attaches to when you create a `ticcs` object. If, for example, you have both CCS for C5000 and CCS for C6000 versions installed, you cannot choose to connect to the C6000 version rather than the C5000 version.

When you issue the command

```
cc = ticcs
```

Embedded IDE Link starts the CCS version you last used. If you last used your C5000 version, the `cc` object accesses the C5000 version.

Workaround

To make your `ticcs` object access the correct processor:

- 1** Start and close the appropriate CCS version before you create the `ticcs` object in MATLAB.
- 2** Create the `ticcs` object using the `boardnum` and `procnum` properties to select your processor, if needed.

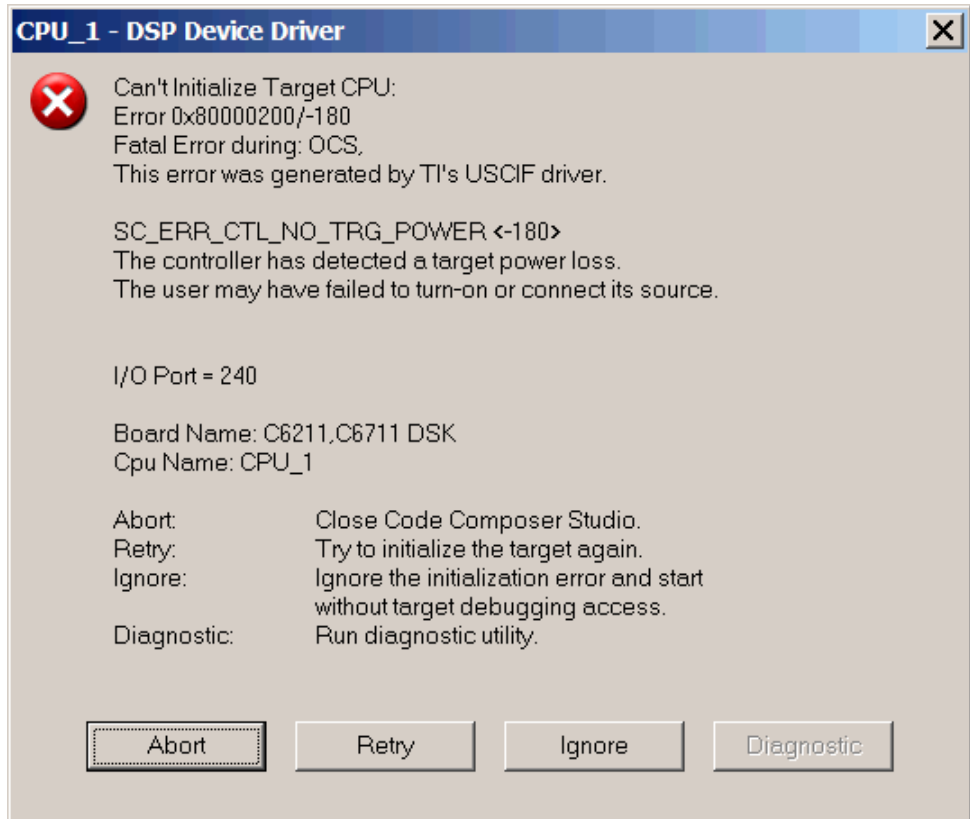
Recall that `ccsboardinfo` returns the `boardnum` and `procnum` values for the processors that CCS recognizes.

Changing CCS Versions During a MATLAB Session

You can use only one version of CCS in a single MATLAB session. Embedded IDE Link does not support using multiple versions of CCS in a MATLAB session. To use another CCS version, exit MATLAB software and restart it. Then create your links to the new version of CCS.

MATLAB Hangs When Code Composer Studio Cannot Find a Board

In MATLAB software, when you create a `ticcs` object, the construction process for the object automatically starts CCS. If CCS cannot find a processor that is connected to your PC, you see a message from CCS like the following DSP Device Driver dialog box that indicates CCS could not initialize the processor.



Four options let you decide how to respond to the failure:

- **Abort** — Closes CCS and suspends control for about 30 seconds. If you used MATLAB software functions to open CCS, such as when you create a `ticcs` object, the system returns control to MATLAB command window after a considerable delay, and issues this warning:

```
??? Unable to establish connection with Code Composer Studio.
```

- **Ignore** — Starts CCS without connecting to any processor. In the CCS IDE you see a status message that says `EMULATOR DISCONNECTED` in the status area of the IDE. If you used MATLAB to start CCS, you get control immediately and Embedded IDE Link creates the `ticcs` object. Because

CCS is not connected to a processor, you cannot use the object to perform processor operations from MATLAB, such as loading or running programs.

- **Retry** — CCS tries again to initialize the processor. If CCS continues not to find your hardware processor, the same DSP Device Driver dialog box reappears. This process continues until either CCS finds the processor or you choose one of the other options to respond to the warning.

One more option, **Diagnostic**, lets you enter diagnostic mode if it is enabled. Usually, **Diagnostic** is not available for you to use.

Using Mapped Drives

Limitations in Code Composer Studio do not allow you to load programs after you set your CCS working directory to a read-only mapped drive. When you set the CCS working directory to a mapped drive for which you do not have write permissions, you cannot load programs from any location. Load operations fail with an Application Error dialog box.

The following combination of commands does not work:

- 1 `cd(cc,'mapped_drive_letter')` % Change CCS working directory to read-only mapped drive.
- 2 `load(cc,'program_file')` % Loading any program fails.

Uninstalling Code Composer Studio 3.3 Prevents Embedded IDE Link From Connecting

Description On a machine where CCS V3.3 and CCS V3.1 are installed, uninstalling V3.3 makes V3.1 unusable from MATLAB. This is because the CCS V3.3 uninstaller leaves stale registry entries in the Windows Registry that prevent MATLAB from connecting to CCS V3.1.

Texas Instruments has documented this uninstall problem and the solution on their Web site at <http://www-k.ext.ti.com/SRV5/CGI-BIN/WEBCGI.EXE/,?St=76,E=0000000000008373418>

Updated information on this issue may also be available from the Bug Reports section of www.mathworks.com at <http://www.mathworks.com/support/bugreports/379676>

PostCodeGenCommand Commands Do Not Affect Embedded IDE Link Projects

PostCodeGenCommand commands, such as the addCompileFlags and addLinkFlags functions in the Real-Time Workshop BuildInfo API do not affect code generated by Embedded IDE Link.

Use the 'Compiler options string' and 'Linker options string' parameters in the **Configuration Parameters > Embedded IDE Link** pane instead. You can also automate this process using a model callback to SET_PARAM the 'CompilerOptionsStr' and 'LinkerOptionsStr' parameters.

A

- apiversion 2-48
- Archive_library 3-15

B

- block limitations using model reference 3-16
- boardnum 2-49
- boards, selecting 3-3

C

- C280x/C28x3x hardware interrupt block 6-2
- C280x/C28x3x Hardware Interrupt block 6-2
- c281x hardware interrupt block 6-6
- C6000 model reference 3-14
- CCS IDE objects
 - tutorial about using 2-2
- ccsappexe 2-49
- Code Composer Studio
 - MATLAB API 1-3
- custom data types 2-54
- custom type definitions 2-54

D

- Data Type Manager 2-54
- data types
 - managing 2-54

E

- Embedded IDE Link™
 - listing link functions 2-41
- export filters to CCS IDE from FDATool 4-1
 - select the export data type 4-6
 - set the Export mode option 4-5

F

- FDATool. *See* export filters to CCS IDE from FDATool
- functions
 - overloading 2-45

H

- Hardware Interrupt block 6-12

I

- import filter coefficients from FDATool.. *See* FDATool

L

- link properties
 - about 2-46 2-48
 - apiversion 2-48
 - boardnum 2-49
 - ccsappexe 2-49
 - numchannels 2-49
 - page 2-50
 - procnum 2-50
 - quick reference table 2-46
 - rtdx 2-51
 - rtdxchannel 2-52
 - timeout 2-52
 - version 2-52
- link properties, details about 2-48
- links
 - closing CCS IDE 2-18
 - closing RTDX 2-38
 - communications for RTDX 2-29
 - creating links for RTDX 2-26
 - details 2-48
 - introducing the tutorial for using links for RTDX 2-21
 - loading files into CCS IDE 2-10

- quick reference 2-46
- running applications using RTDX 2-31
- tutorial about using links for RTDX 2-20
- working with your processor 2-12

M

- managing data types 2-54
- model reference 3-14
 - about 3-14
 - Archive_library 3-15
 - block limitations 3-16
 - modelreferencecompliant flag 3-17
 - setting build action 3-15
 - Target Preferences blocks 3-16
 - using 3-15
- modelreferencecompliant flag 3-17

N

- numchannels 2-49

O

- object
 - ticcs 2-42
- objects
 - creating objects for CCS IDE 2-8
 - introducing the objects for CCS IDE
 - tutorial 2-2
 - selecting processors for CCS IDE 2-6
 - tutorial about using Automation Interface for CCS IDE 2-2
- overloading 2-45

P

- page 2-50

- procnum 2-50
- project generation
 - selecting the board 3-3
- properties
 - link properties 2-46

R

- rt dx 2-51
- RTDX links
 - tutorial about using 2-20
- rt dxchannel 2-52

S

- selecting boards 3-3

T

- Target Preferences blocks in referenced models 3-16
- ticcs 2-42
- timeout 2-52
- tutorials
 - links for RTDX 2-20
 - objects for CCS 2-2
- typedefs 2-56
 - about 2-54
 - adding 2-56
 - managing 2-56
 - removing 2-56

V

- version 2-52