

Link for Code Composer Studio™ Development Tools

For Use with **MATLAB®**

- Computation
- Visualization
- Programming

User's Guide

Version 2



How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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.

Link for Code Composer Studio™ Development Tools

© COPYRIGHT 2002–2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

Other product or brand names are 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+)
Sept. 2006	Online only	Revised for Version 2.1 (Release 2006b)

Getting Started

1

What Is Link for Code Composer Studio?	1-2
Specific Link Features Supported For Each Board	
Family	1-2
Configuration Information	1-5
Requirements for Link for Code Composer Studio	1-8
Platform Requirements — Hardware and Operating	
System	1-8
Getting Started with Links	1-11
Introducing the Tutorial	1-11
Selecting Your Target	1-14
Creating and Querying Links for CCS IDE	1-15
Loading Files into CCS	1-18
Working with Links and Data	1-21
Working with Embedded Objects	1-26
Closing the Links or Cleaning Up CCS IDE	1-36
Getting Started with RTDX	1-38
Introducing the Tutorial for Using RTDX	1-38
Creating the Links	1-43
Configuring Communications Channels	1-46
Running the Application	1-48
Closing the Links or Cleaning Up	1-55
Listing the Functions for Links	1-59
Constructing Link Objects	1-61
Example — Constructor for Links	1-61
Properties and Property Values	1-63
Setting and Retrieving Property Values	1-63
Setting Property Values Directly at Construction	1-63

Setting Property Values with set	1-64
Retrieving Properties with get	1-65
Direct Property Referencing to Set and Get Values	1-67
Overloaded Functions for Links	1-68
Link Properties	1-69
Quick Reference to Link Properties	1-69
Details About the Link Properties	1-71

Link Objects

2

Introduction to Objects	2-3
Some Object-Oriented Programming Terms	2-5
About the Relationships Between Objects	2-9
Class Diagrams for the Link for Code Composer Studio ..	2-11
Numeric Objects — Their Methods and Properties	2-15
Properties of Numeric Objects	2-15
Methods of Numeric Objects	2-16
Bitfield Objects — Their Methods and Properties	2-18
Properties of Bitfield Objects	2-18
Methods of Bitfield Objects	2-20
Enum Objects — Their Methods and Properties	2-21
Properties of Enum Objects	2-21
Methods of Enum Objects	2-23
Pointer Objects — Their Methods and Properties	2-24
Properties of Pointer Objects	2-24
Methods of Pointer Objects	2-26
String Objects — Their Methods and Properties	2-27
Properties of String Objects	2-27
Methods of String Objects	2-29

Rnumeric Objects — Their Methods and Properties ..	2-30
Properties of Rnumeric Objects	2-30
Methods of Rnumeric Objects	2-32
Renum Objects — Their Methods and Properties	2-33
Properties of Renum Objects	2-33
Methods of Renum Objects	2-35
Rpointer Objects — Their Methods and Properties ...	2-36
Properties of Rpointer Objects	2-36
Methods of Rpointer Objects	2-38
Rstring Objects — Their Methods and Properties	2-39
Properties of Rstring Objects	2-39
Methods of Rstring Objects	2-41
Function Objects — Their Methods and Properties ...	2-42
Properties of Function Objects	2-42
Methods of Function Objects	2-44
Structure Objects — Their Methods and Properties ...	2-46
Properties of Structure Objects	2-46
Methods of Structure Objects	2-47
Working with Structure Objects	2-47
Type Objects — Their Methods and Properties	2-52
Properties of Type Objects	2-52
Methods of Type Objects	2-53
Constructing Objects That Access Bitfields	2-54
Creating function Objects	2-56
When to Use declare to Provide the Function	
Declaration	2-57
Differences Between Objects for Library Functions and C	
Functions	2-58
Examples of Creating Function Objects	2-59
Creating Type Objects	2-74
Working with Type Definitions in Projects	2-74

Tutorial — Using function Objects and Function Calls	2-77
Introducing the Tutorial	2-78
To Run the Hardware-In-The-Loop Tutorial	2-80
Select Your Target and Load the Tutorial Project	2-82
Initialize the Embedded C Variables and Use read and write	2-85
Use read, write, cast, and convert with Objects	2-90
Construct a function Object	2-94
Use Methods That Work with Function Objects	2-96
Construct Different Objects and Work with Them	2-102
Close The Tutorial and Clean Up	2-107
Managing Custom Data Types with the Data Type Manager	2-109
Adding Custom Type Definitions to MATLAB	2-111
Reference for the Properties of Embedded Objects ...	2-120
Property Reference Format and Contents	2-120
Functions	2-120

Using FDATool

3

Introducing FDATool	3-2
Guidelines on Exporting Filters from FDATool to CCS IDE	3-3
Selecting the Export Mode	3-4
Cautions Regarding Writing Directly to Memory	3-5
Variables and Memory Necessary for Filter Export	3-6
Selecting the Export Data Type	3-8
Tutorial — Exporting Filters from FDATool to CCS IDE	3-10
Descriptions of the Two Tutorial Tasks	3-10
Setting Up for the Tutorial	3-10
Task 1 — Export Filter by Generating a C Header File ...	3-10

Task 2 — Export Filter by Writing Directly to Target	
Memory	3-17

Functions — By Category

4

Operations on Links for CCS IDE	4-2
Operations on Links for RTDX	4-4
Manipulate Data	4-5
Hardware-in-the-Loop Processing	4-5

Functions — Alphabetical List

5

Supported Hardware

A

Introduction to Supported Platforms	A-2
Supported Hardware and Simulators	A-2
Link Features Supported by Each Processor or Family ...	A-3
OMAP Coemulation Support	A-5
Custom Hardware Support	A-5
 Supported Versions of Code Composer Studio	 A-7
 Continuing Issues in Link for Code Composer	
Studio	A-9
Function Call Support for Different Compiler Options ...	A-10
Function Calls on Functions That Use Global Variables ..	A-10

Demonstration Programs Do Not Run Properly Without Correct GEL Files	A-11
Issues Using USB-Based RTDX Emulators and the C6416 DSK and C6713 DSK	A-12
Error When Accessing type Property of cc dsp Object Having Size>1	A-13
Changing the represent Property of an Object	A-14
Changing Values of Local Variables Does Not Take Effect	A-15
Code Composer Studio Cannot Find a File After You Halt a Program	A-15
C54x XPC Register Can Be Modified Only Through the PC Register	A-17
Working with Multiple Installed Versions of Code Composer Studio	A-17
Changing CCS Versions During a MATLAB Session	A-18
createobj and address Return Inconsistent Page Information on C5xxx Targets	A-18
MATLAB Hangs When Code Composer Studio Cannot Find a Target	A-20
Different Read Techniques Appear to Return Different Values	A-22

Index

Getting Started

What Is Link for Code Composer Studio? (p. 1-2)	Introduces Link for Code Composer Studio — the capabilities and supported hardware
Configuration Information (p. 1-5)	Shows you how to determine whether you have Link for Code Composer Studio on your PC
Requirements for Link for Code Composer Studio (p. 1-8)	Describes the software and hardware requirements for running this product
Getting Started with Links (p. 1-11)	Guides you through the process of creating and using links and embedded objects
Getting Started with RTDX (p. 1-38)	Demonstrates one process for using RTDX to communicate with CCS IDE and transferring data between MATLAB® and CCS IDE
Constructing Link Objects (p. 1-61)	Shows you what a link object is and how to construct one
Properties and Property Values (p. 1-63)	Describes how to work with objects, their properties and property values
Overloaded Functions for Links (p. 1-68)	Explains what makes a function overloaded and where to get more information about the overloaded functions in this product
Link Properties (p. 1-69)	Describes the properties of link objects

What Is Link for Code Composer Studio?

Link for Code Composer Studio Development Tools lets you use MATLAB® functions to communicate with Code Composer Studio™ and with information stored in memory and registers on a target. With the links you can transfer information to and from Code Composer Studio 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.

Note Both the links and the embedded objects are objects, and you work with them in the same way you use all MATLAB objects. You can set and get their properties, and use their methods to change them or manipulate them.

With Link for Code Composer Studio, you create two kinds of objects:

- Links that connect MATLAB to Code Composer Studio. For information about using links, refer to “Requirements for Link for Code Composer Studio” on page 1-8.
- Embedded objects you create that provide access to data and functions in your project in Code Composer Studio and on your target. The link objects let you use the embedded objects to access your target. Refer to “Introduction to Objects” on page 2-3 for more information about using the embedded objects, their properties, and their methods.

Specific Link Features Supported For Each Board Family

Within the collection of hardware that Link for Code Composer Studio supports, some components of the link do not apply.

Link for Code Composer Studio provides four components that work with and use CCS IDE and TI Real-Time Data Exchange (RTDX™):

- Debug component— Lets you use objects to create links between CCS IDE and MATLAB. From the command window, you can run applications in CCS IDE, send to and receive data from target memory, and check the

processor status, as well as other functions such as starting and stopping applications running on your digital signal processors.

- **Data manipulation component**— Provides object methods and properties that let you access and manipulate information stored in memory and registers on digital signal processors, or in your Code Composer Studio project. From MATLAB you gather information from your project, work with the information in MATLAB, doing things like converting data types, creating function declarations, or changing values, and return the information to your project — all from the MATLAB command line.
- **Function call component**— Enables you to write scripts in MATLAB that exercise functions from your project on your target processor. From MATLAB, you can generate data, send the data to your target and use a C function in your program to manipulate the data on your hardware or simulator. Afterwards, you return the output to MATLAB so you can analyze the results.
 - **RTDX component**— Provides a communications pathway between MATLAB and digital signal processors installed on your PC. Using objects in the Link for Code Composer Studio, you open channels to processors on boards in your computer and send and retrieve data about the processors and executing applications, as well as send data to the processes for use and get data from the applications.

In the next table, each board family appears with headings that specify the support provided. The information here is general according to the processor family. For details about the support for processors within a family, such as the C24xx, refer to Appendix A, “Supported Hardware”.

Some Board Families Do Not Support Full Link for Code Composer Studio Functions

Board Family	Hardware/Simulators?	Debug Mode?	Data Manipulation	Function Call?	RTDX
C2xx	Yes/No	Yes	No	No	No
C54x	Yes/Yes	Yes	Yes	Yes	Yes
C55x	Yes/Yes	Yes	Yes	No	Yes

Some Board Families Do Not Support Full Link for Code Composer Studio Functions (Continued)

Board Family	Hardware/Simulators?	Debug Mode?	Data Manipulation	Function Call?	RTDX
C6x	Yes/Yes	Yes	Yes	Yes	Yes
TMS470Rxx	Yes/Yes	Yes	Yes	No	No

Debug mode includes those operations that CCS handles and that Link for Code Composer Studio enables you to use from MATLAB — a yes tells you that the listed hardware supports MATLAB interaction with CCS. Data manipulation support indicates that the board family supports using objects in MATLAB to work with symbol table entries in CCS. A yes in the Function call column means the board family supports using function objects to run functions on your target from MATLAB.

Configuration Information

To determine whether the Link for Code Composer Studio is installed on your system, type this command at the MATLAB prompt.

```
help ccslink
```

When you enter this command, MATLAB displays the contents of the product, the first few lines of which are shown here. What you see should look similar, although the product version numbers change.

```
Link for Code Composer Studio  
Version 1.3.2 (R14SP1) 18-Sep-2004  
  
Methods for Link for Code Composer Studio  
ccshelp/ccsdsp - Construct CCS object.  
...
```

If you do not see the listing, or MATLAB does not recognize the command, you need to install the Link for Code Composer Studio. Without the software, you cannot use MATLAB with the links to communicate with Code Composer Studio.

Note For up-to-date information about system requirements, refer to the system requirements page, available in the products area at the MathWorks Web site (<http://www.mathworks.com>).

To verify that CCS is installed on your machine, enter

```
ccsboardinfo
```

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

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum ...	0	6701	TMS320C6701
0	C6x11 DSK (Texas Instruments)	0	CPU	TMS320C6x1x

If MATLAB does not return information about any boards, revisit your CCS installation and setup in your CCS documentation.

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

The Link for Code Composer Studio Development Tools uses objects to create:

- Links to the Code Composer Studio Integrated Development Environment (CCS IDE)
- Links to the Real-Time Data Exchange (RTDX) interface. This link is a subset of the link to CCS IDE.

Concepts you need to know about the objects for linking in this toolbox are covered in these sections:

- “Constructing Link Objects” on page 1-61
- “Properties and Property Values” on page 1-63
- “Setting and Retrieving Property Values” on page 1-63
- “Setting Property Values Directly at Construction” on page 1-63
- “Setting Property Values with set” on page 1-64
- “Retrieving Properties with get” on page 1-65
- “Direct Property Referencing to Set and Get Values” on page 1-67
- “Overloaded Functions for Links” on page 1-68

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

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

Requirements for Link for Code Composer Studio

This section describes the hardware and software you need to run the Link for Code Composer Studio Development Tools on your Microsoft Windows PC.

Link for Code Composer Studio runs on Microsoft Windows XP and Windows 2000 platforms.

Platform Requirements – Hardware and Operating System

To run the Link for Code Composer Studio, your host PC must meet the following hardware configuration:

- Intel Pentium or Intel Pentium processor compatible PC
- 64 MB RAM (128 MB recommended)
- 20 MB hard disk space available after installing MATLAB
- Color monitor
- One full-length peripheral component interface (PCI) slot available to use the C6701 EVM internally in your PC
- Slots or connectors, such as USB or parallel ports, for connecting your hardware
- CD-ROM drive
- Microsoft Windows XP or Windows 2000

Refer to your documentation from The MathWorks for more information on installing the software required to support Link for Code Composer Studio, list in this table.

Prerequisites for Using Link for Code Composer Studio Software for Targeting

Installed Product	Additional Information
MATLAB 7	Core software from The MathWorks
Signal Processing Toolbox 6.2 or later	Software package for analyzing signals, processing signals, and developing algorithms
Simulink®	Simulation and Model-Based Design

For information about the software required to use the Link for Code Composer Studio Development Tools, refer to the Products area of the MathWorks Web site — <http://www.mathworks.com>.

Texas Instruments Software

In addition to the required software from The MathWorks, Link for Code Composer Studio requires that you install the Texas Instruments development tools and software listed in the following table.

Required TI Software for Link for Code Composer Studio

Installed Product	Additional Information
Code Composer Studio 3.1	<p>Texas Instruments integrated development environment (IDE) that provides code debugging and development tools (recommended).</p> <ul style="list-style-type: none"> • For C2000 • For C5000 • For C6000 • For OMAP <p>To use Link for Code Composer Studio with the Embedded Target for TI C6000</p>

Required TI Software for Link for Code Composer Studio (Continued)

Installed Product	Additional Information
	DSP v3.0, you must use Code Composer Studio 3.1.

In addition to the TI software, you need one or more boards or simulators that CCS supports in the Setup utility.

For up-to-date information about the software from The MathWorks you need to use the Link for Code Composer Studio, refer to the MathWorks Web site — <http://www.mathworks.com>. Check the Product area for the Link for Code Composer Studio.

Getting Started with Links

The Link for Code Composer Studio™ IDE (CCS IDE), a part of the Link for Code Composer Studio, provides a connection between MATLAB and a digital signal processor in Code Composer Studio. Using links provides a mechanism for you to control and manipulate a signal processing application using the computational power of MATLAB. This can help you while you debug and develop your application. Another possible use is for creating MATLAB scripts that you use to verify and test algorithms that run in their final implementation on your production processor target.

Before using the functions available with the link for CCS IDE, you must select a digital signal processor to be your target because any link you create is specific to a designated digital signal processor. Selecting a processor is only necessary for multiprocessor boards or multiple board configurations of Code Composer Studio. When you have only one board with a single processor, the link defaults to the existing processor. For the links, the simulator counts a board; if you have both a board and a simulator that CCS recognizes, you must specify the target explicitly.

Introducing the Tutorial

To get you started using links for CCS IDE software, the Link for Code Composer Studio includes an example script `ccstutorial.m`. As you follow along with this tutorial, you perform five tasks that step you through creating and using links for CCS IDE:

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

During this tutorial, you load and run a simple digital signal processing application on target processor you select. To help you understand how they work, the tutorial demonstrates both writing to memory and reading from

memory in the “Working with Links and Data” on page 1-21 portion of the tutorial.

Using the read and write functions gets a bit complicated. You can read and write a range of data types to and from your target. Seeing how the read and write functions work can help you when you need to do your work.

The tutorial covers the link functions listed below. The functions listed first apply to CCS IDE independent of the links — you do not need a link to use these functions. The functions listed next require a CCS IDE link in place before you can use the function syntax:

- Global functions for CCS IDE
 - `ccsboardinfo` — return information about the boards that CCS IDE recognizes as installed on your PC.
 - `boardprocel` — select the board to target. Although you can use this generally, the Link for Code Composer Studio provides it as an example of a user interface you can build and as a tool in the tutorial. We do not recommend that you use this to select your target. Use `ccsboardinfo` and `ccsdsp` to specify the target for your processing application
 - `ccsdsp` — construct a link to CCS IDE. When you construct the link you specify the target board and processor.
 - `clear` — remove a specific link to CCS IDE or remove all existing links.
- CCS IDE link functions
 - `address` — return the address and page for an entry in the symbol table in CCS IDE
 - `display` — display the properties of a link to CCS IDE and RTDX
 - `halt` — terminate execution of a process running on the processor
 - `info` — return information about the target processor or information about open RTDX channels
 - `isrunning` — test whether the target processor is executing a process
 - `isrtdxcapable` — test whether your target supports RTDX communications
 - `read` — retrieve data from memory on the target processor

- `restart` — restore the program counter (PC) to the entry point for the current program
- `run` — execute the program loaded on the target processor
- `visible` — set whether CCS IDE window is visible on the desktop while CCS IDE is running
- `write` — write data to memory on the target processor
- Link for Code Composer Studio functions for working with embedded objects
 - `cast` — create a new object with a different datatype (the `represent` property) from an object in Link for Code Composer Studio. Demonstrated with a numeric object.
 - `convert` — change the `represent` property for an object from one datatype to another. Demonstrated with a numeric object.
 - `createobj` — return an object in MATLAB that accesses embedded data. Demonstrated with structure, string, and numeric objects.
 - `getmember` — return an object that accesses a single field from a structure. Demonstrated with a structure object.
 - `goto` — position the program counter to the specified location in the project code.
 - `list` — return various information listings from Code Composer Studio.
 - `read` — read the information at the location accessed by an object into MATLAB as numeric values. Demonstrated with a numeric, string, structure, and enumerated objects.
 - `readnumeric` — return the numeric equivalent of data at the location accessed by an object. Demonstrated with an enumerated object.
 - `write` — write to the location referenced by an object. Demonstrated with numeric, string, structure, and enumerated objects.

Running the Interactive Tutorial

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

To run the tutorial in MATLAB, click run `ccstutorial`. Running the interactive tutorial in MATLAB puts you 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 M-file used here by clicking `ccstutorial.m`.

Selecting Your Target

Links for CCS IDE provides two tools for selecting a DSP 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 structure that you use when you want your script to select a target board without your help.

Note The board and processor you select in the tutorial remains the target throughout the tutorial.

1 To see a list of the boards and processors installed on your PC, type

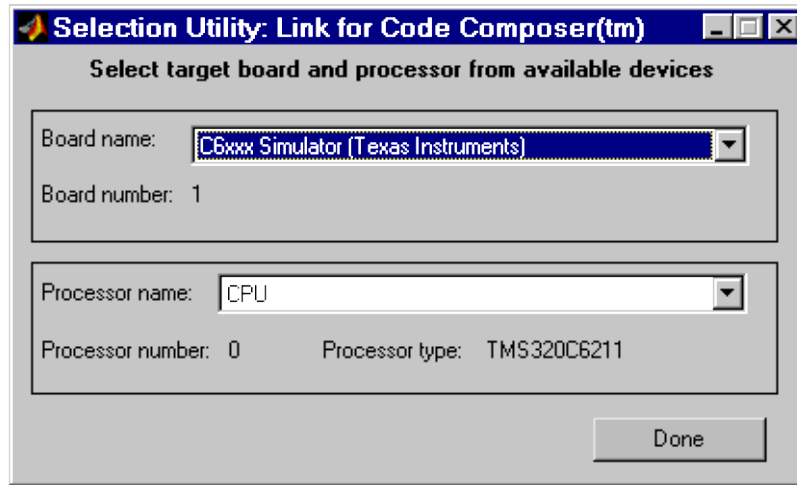
```
ccsboardinfo
```

MATLAB 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, `boardprocse1`, to select a target board, type

```
[boardnum,procnum] = boardprocse1
```


When you use `boardprocse1`, 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 target. When you create the link for CCS IDE in the next section of this tutorial, the selected board and processor become the target of the link.

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 Links for CCS IDE

In this tutorial section you create the connection between MATLAB and Code Composer Studio IDE. This connection, or link, is represented by a MATLAB object, which for this session you save as variable `cc`. You use function `ccsdsp` to create link objects. When you create links, `ccsdsp` input arguments let you define other link properties, such as the global timeout. Refer to the `ccsdsp` documentation for more information on these input arguments.

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

- 1** Create a link to your selected board and processor by typing

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

If you were to watch closely, and your machine is not too fast, you see Code Composer Studio appear briefly when you call `ccsdsp`. If CCS IDE was not running before you established the new link, CCS starts and gets placed in the background.

Note 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 show up as a process, `cc_app.exe`, on the **Processes** tab in Task Manager.

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

In most cases, you need to interact with Code Composer Studio while you develop your application, so the first link function we introduce, `visible`, controls the state of Code Composer Studio on your desktop. `visible` accepts Boolean inputs that make Code Composer Studio either visible on your desktop (input to `visible` ≥ 1) or invisible on your desktop (input to `visible` = 0). For the rest of this tutorial you need to interact with CCS IDE so we use `visible` to set the CCS IDE visibility to 1.

- 3** Now type `display(cc)` at the prompt to see the status information.

```
CCSDSP 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
```

The Link for Code Composer Studio provides three functions to read the status of a target board and processor:

- `info` — return a structure of testable target conditions
- `display` — print information about the target CPU
- `isrunning` — return the state (running or halted) of the CPU
- `isrtdxcapable` — return whether the target handle RTDX

4 Type `linkinfo = info(cc)`.

The `cc` link status information tells you about the target

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

5 Check to see if the target is running by entering

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

MATLAB responds by telling you that the processor is stopped

```
runstatus =
```

```
0
```

- 6** At last, check to see whether the target supports RTDX communications by entering

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

1

Loading Files into CCS

You have established the link to CCS IDE and to target. Using three functions you learned about the target, whether it was running, its type, and whether CCS IDE was visible. Now the target needs something to do.

In this tutorial section you load the executable code for the target CPU in CCS IDE. For this tutorial, the Link for Code Composer Studio includes a Code Composer Studio project file. With the following commands in the tutorial you locate the tutorial project file and load it into CCS IDE. The open function directs Code Composer Studio to load a project file or workspace file

Note Code Composer Studio has its own workspace and workspace files that are quite different from MATLAB workspace files and the MATLAB workspace. Remember to pay attention to both workspaces.

After you have executable code running on your target you can exchange data blocks with the target. This is the purpose of the links for CCS IDE:

- 1** To load the appropriate project file to your target, enter the following commands at the MATLAB prompt. `getDemoProject` is a specialized function for loading Link for CCS demo files. It is not supported as a standard Link for CCS function.

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

    isLibProj: 0
TemplateProject: 'C:\Temp\LinkForCCSDemos_v2.1\template\c6x\c64x.pjt'
DemoDir: 'C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp'
ProjectFile: 'C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp\ccstut.pjt'
```

```

ProgramFile: 'C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp\ccstut.out'
SrcFile: {'W:\bat\Adsphw\perfect\matlab\toolbox\ccslink\ccsdemos\ccstutorial\c6x\c64xp\ccstut.m'}
LibFile: ''
CmdFile: {'W:\bat\Adsphw\perfect\matlab\toolbox\ccslink\ccsdemos\shared\c6x\c64xp\ccstut.m'}
HdrFile: ''
BuildOpts: [1x1 struct]
ProjectAction: 'recreateProj-rebuildProg'
RebuildDemo: 1

```

```
demoPjt.ProjectFile
```

```
ans =
```

```
C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp\ccstut.pjt
```

```
demoPjt.DemoDir
```

```
ans =
```

```
C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp
```

Notice where the demo files are stored on your machine. In general, Link for CCS stores the demo project files in

```
LinkforCCS_vproduct_version
```

For example, if you are using version 2.1 of Link for CCS, the project demos are stored in `LinkforCCS_v2.1`. Link for CCS creates this directory to a location that you have write permissions on your machine. Usually, there are three possible locations where Link for CCS tries to create the demo directory, in the order shown.

- a** In the MATLAB work directory—`matlabroot\work`.
- b** If `matlabroot\work` is not available, Link for CCS uses a temporary directory on the C drive, such as `C:\temp\`.
- c** If Link for CCS cannot use the temp directory, you see dialog box that asks you to select a location to store the demos.

- 2** Next, build the target executable file in CCS IDE. Select **Project > Build** from the menu bar in CCS IDE.

You may get an error here 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. Use the path in the error message as an indicator of where to find the missing files.

- 3** Type `load(cc, 'projectname.out')` to load the target execution file, where *projectname* is the tutorial you chose, such as `ccstut_67x`.

- 4** You now have a loaded program file and associated symbol table. To determine the memory address of the global symbol `ddata`, type

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

    1.0e+009 *
    2.1475      0
```

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

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

- 5** To convert `ddata` to a hexadecimal string that contains the memory address and memory page, type

```
dec2hex(ddata)

MATLAB displays

ans =

    80000010
    00000000
```

where the memory page is 0x00000000 and the address is 0x80000010.

Working with Links and Data

With the target code loaded, you can use the Link for Code Composer Studio 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`. The Link for Code Composer Studio 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 access these processor memory arrays from MATLAB using the functions `read` and `write`.

The Link for Code Composer Studio provides three functions to control target execution — `run`, `halt`, and `restart`. To demonstrate these commands, use CCS IDE to add a breakpoint to line 64 of `cctut.c`. Line 64 is

```
printf("Link for Code Composer: Tutorial - Memory Modified by Matlab!\n");
```

For information about adding breakpoints to a file, refer to your *Code Composer Studio User's Guide* from Texas Instruments. Then proceed with the tutorial:

- 1 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

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

in the CCS IDE Stdout tab. Nothing prints in MATLAB.

- 2** Before you restart your program (currently stopped at line 64) you can change some of the values in memory. Perform one of the procedures listed below based on your target processor.

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

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

MATLAB responds with

```
ddatv =
```

```
16.3000 -2.1300 5.1000 11.8000
```

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

Now MATLAB 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 typing
`write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)...
sin(pi/4)]))`

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

- d** To change `idat`, type

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


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

- e Start the program running again by typing

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

Checking the Stdout tab in CCS IDE reveals that `ddat` and `idat` contain new values. Now we read those new values back into MATLAB.

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

```
ddatv =
```

```
3.1416 12.3000 0.3679 0.7071
```

`ddatv` does contain the values you wrote in step c.

- g Check that the change to `idatv` occurred by typing

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

MATLAB returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h Finally, use `restart` to reset the program counter for your program to the beginning. Type

```
restart(cc);
```

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

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

MATLAB responds with

```
ddatv =
```

```
16.3000 -2.1300 5.1000 11.8000
```

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

Now MATLAB 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 typing
`write(cc,address(cc,'ddat'),double([pi 12.3 exp(-1)...
sin(pi/4)]))`

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

- d** To change `idat`, type

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

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

- e** Now start the program running again by typing

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

Checking the Stdout tab in CCS IDE reveals that `ddat` and `idat` contain new values. Now read those new values back into MATLAB.

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

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

`ddatv` does contain the values you wrote in step c.

- g** Check that the change to `idatv` occurred by typing

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

MATLAB returns the new values for `idatv`.

```
idatv =
```

```
1 2 3 4
```

- h** Finally, use `restart` to reset the program counter for your program to the beginning. Type

```
restart(cc);
```

- 3** The Link for Code Composer Studio offers two more functions for reading and writing data to your target. 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. As before, the functions behave slightly differently depending on your target. Select the appropriate procedure for your target to demonstrate `regread` and `regwrite`.

C5xxx processor family — Most registers are memory-mapped and consequently are available using `read` and `write`. However, the PC register is not memory mapped. To access this register, you use the special pair of 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, type

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

To tell MATLAB what datatype you are reading, the string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

In response, MATLAB displays

```
ans =
```

```
33824
```

- b** To write a new value to the PC register, type

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

This time, binary as an input argument tells MATLAB to write the value to the target as an unsigned binary integer. Notice that you used hex2dec to convert the hexadecimal string to decimal.

- c** Check the PC contains the value you wrote.

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

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

- a** Retrieve the value in register A0 and store it in a variable in your MATLAB workspace. Type

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

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

- b** Retrieve the value in register B2 as an unsigned binary integer, by typing

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

- c** Now, 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 see a listing of the processor registers.

Working with Embedded Objects

Having direct access to the memory on your target DSP, as provided by the links in Link for Code Composer Studio, can be a powerful tool for helping you develop and troubleshoot your digital signal processing applications. But for programming in C, it is perhaps more valuable to be able to work with memory and data in ways that are consistent with the C variables embedded in your programs.

Link for Code Composer Studio implements just this sort of access and manipulation capability by using MATLAB objects (called embedded objects in this guide) that access and represent variables and data embedded in your project. Various functions that compose the Link for Code Composer Studio, such as `createobj`, `convert`, and `write`, help you create the embedded objects you use to work with your data in DSP memory and registers, and let you manipulate the data in MATLAB and in your code.

This portion of the tutorial introduces some of the functions and how to use them to access and manipulate them.

Function `list` generates a lot of information for you about an embedded variable in the symbol table. An even more useful function is `createobj` that creates a MATLAB object that represents a C variable in the symbol table in CCS. Working with the object that `createobj` returns, you can read the entire contents of a variable, or one or more elements of the variable when the variable is an array or structure.

From the beginning of this tutorial you have used the link object `cc` with all of the functions. `cc` represents the path to communicate with a particular processor in CCS. For the remainder of this tutorial you work with a variety of functions that use, not the link object `cc`, but other objects such as numeric or structure objects, that represent embedded objects in CCS. All of these new functions use the object names (handles) as the first input argument to the function (in just the way you used `cc`). When you create the object `cvar` in step 4 that follows, `cvar` represents the embedded variable `idat`.

To begin, restart the program and use `list` to get some information about a variable (an embedded object) in Code Composer Studio.

Using list

- 1 To restart the program in CCS, enter

```
restart(cc)
```

This resets the program counter to the beginning of your program.

- 2 To move the program counter (PC) to the beginning of main, which you should do before rerunning your program, enter

```
goto(cc,'main')
run(cc,'main')
```

Moving the PC to main ensures that the program initializes the embedded C variables.

- 3** Now, to get information about a variable in your program, use `list` with two input options — **'variable'** which defines the type of information to return, and **'idat'** which identifies the symbol itself.

```
idatlist = list(cc,'variable','idat')
```

`idat` is a global variable; the input keyword **variable** identifies it as one. Other keywords for `list` include **project**, **globalvar**, **function**, and **type**. Refer to `list` for more information about these options.

In your MATLAB workspace and window, you see a new structure named `idatlist`. If you use the MATLAB Workspace browser, double-click `idatlist` in the browser to see `idatlist`.

- 4** Rather than using `list` to get information about `idat`, create an object that represents `idat` in your MATLAB workspace by entering

```
cvar = createobj(cc,'idat')
```

which creates the new numeric object `cvar`.

```
cvar=createobj(cc,'idat')
```

```
NUMERIC Object stored in memory:
Symbol name           : idat
Address               : [ 44316 0]
Data type             : short
Word size             : 16 bits
Address units per value : 2 au
Representation        : signed
Size                  : [ 4 ]
Total address units   : 8 au
Array ordering        : row-major
Endianness            : little
```

You use `cvar`, through the numeric object properties and functions, to access and manipulate the embedded variable `idat`, both in your MATLAB workspace and in CCS if you write your changes back to CCS from your workspace.

Using read and write

- 5** Try the following functions to read and write `cvar`. Notice the way the return values change as you change the function syntax. Notice also that `write` actually changes the data in memory on the target, as you see from what comes back to MATLAB after the third read.

a `read(cvar)`

This form returns all of the entries in the embedded array `cvar` to your MATLAB workspace.

`ans =`

```

         1         508         647         7000

```

b `read(cvar,2)`

In contrast to the previous syntax, this one returns only the second element of `cvar` — 508.

c `write(cvar,4,7001)`

Using `write` to change the value stored in the fourth element of `cvar` to 7001.

d `write(cvar,1,'FFFF')`

Change the first element of `cvar` to -1, which is the decimal equivalent of 0xFFFF. When you entered FFFF as a string (enclosed in single quotation marks), `write` converts the string to its decimal equivalent and stores that at the target location in memory.

e `read(cvar)`

At last, read the embedded array `cvar` to see if your changes to the first and fourth elements really occurred (they did).

```
f read(cvar,[1 size(cvar)])
```

Finally, read the first and last elements of the embedded variable `cvar`.

Using `cast`, `convert`, and `size`

Each time you used `read`, the function took the raw values of `idat` stored in memory on your target and converted them to equivalent MATLAB numeric values. The way that `read` converts `idat` elements to numeric values is controlled by the properties of the object `cvar` which resulted from using `createobj` to create it. When you created `cvar`, the object that accesses the embedded variable `idat`, `createobj` assigned default property values to the properties of `cvar` that were appropriate for your target DSP architecture and for the C representation of variable `idat`.

In many cases, it may help you develop your program if you change the default conversion properties. Several of the object properties, such as `endianness`, `arrayorder`, and `size` respond to changes made using function `set`. To make more complex changes, use functions like `cast` and `convert` that adjust multiple object property values simultaneously.

In step 6 of this tutorial, you have the opportunity to use `cast`, `convert`, and `size` to modify `cvar` by changing property values. Unlike `read` and `write`, `cast`, `convert`, and `size` (and `set` mentioned earlier) do not affect the information stored on the target; they only change the properties of the object in MATLAB. Unless you write your changes back to your target, the changes you make in MATLAB stay in MATLAB.

- 6 To introduce changing the properties of `cvar` using `cast`, `convert`, and `size`, enter the following commands at the prompt. In this series of examples, you use `read` to view the changes each command makes to `cvar`.

- a `set(cvar,'size',[2])`

As a result of this function, `idat` gets resized to only the first two elements in the array.

- b `read(cvar)`

```
ans =
```

```
1 508
```


Returns only two values, not the full data set you saw in step 5a.

c `uintcvar = cast(cvar, 'unsigned short')`

`uintcvar` is a new object, a copy of `cvar` (and thus `idat`), but with the `datatype` property value of `unsigned short` instead of `double`. Notice that the actual values are not different — just the interpretation. Where `cvar` interprets the values in `idat` as doubles, `uintcvar` interprets the values in `idat` as unsigned integers with 16 bits each. Now when you use the object to read `idat`, the returned values from `idat` are interpreted differently.

d `read(uintcvar)`

e `convert(cvar, 'unsigned short')`

In contrast to `cast`, `convert` does not make a copy of `cvar`; it changes the `datatype` property of `cvar` to be `unsigned short`.

NUMERIC Object stored in memory:

Symbol name : `idat`

Address : [44316 0]

Data type : `unsigned short`

Word size : 16 bits

Address units per value : 2 au

Representation : `unsigned`

Size : [2]

Total address units : 4 au

Array ordering : `row-major`

Endianness : `little`

f `read(cvar)`

`ans =`

1 508

Remember that one of the first things you did in these examples was change the size of `cvar` to 2. You should see that reflected in the returned values. The values returned by `cvar` after you change the `datatype` property should match the values returned by `uintcvar` since the objects have the same properties.

One more thing to notice — the first value of `idat` is no longer -1, although you changed the value in step 5d. Recall that you changed the `datatype` to `unsigned short` for `cvar`, so the first element of `idat` that you set to -1 is now shown as the unsigned equivalent 1.

Using `getmember`

To this point you have worked with fairly simple data in memory on your target. However, with functions in Link for Code Composer Studio, you can manipulate more complex data like strings, structures, bitfields, enumerated data types, and pointers in a very similar way.

In the next, somewhat extended examples, the tutorial demonstrates some common functions for manipulating structures, strings, and enumerated data types on your target. Pay particular attention to function `getmember` which extracts a single specified field from a structure on your target as an object in MATLAB.

```
7 cvar = createobj(cc, 'myStruct')
```

Here you create a new object `cvar`, replacing the old `cvar`, that represents an embedded structure named `myStruct` on your target. When you loaded this tutorial program, one of the defined structures in the program was `myStruct`.

```
STRUCTURE Object stored in memory:
Symbol name           : myStruct
Address               : [ 44288 0]
Address units per value : 28 au
Size                  : [ 1 ]
Total Address Units   : 28 au
Array ordering        : row-major
Members                : 'iy', 'iz'
```

8 `read(cvar)`

```
ans =

    iy: [2x3 double]
    iz: 'MatlabLink'
```

Now you see the contents of `myStruct`, its fields and values.

Here's the definition of `myStruct` from `ccstut.c` in CCS.

```
struct TAG_myStruct {
    int iy[2][3];
    myEnum iz;
} myStruct = { {{1,2,3}},{4,-5,6}}, MatlabLink}
```

9 `write(cvar, 'iz', 'Simulink')`

After this command, you have updated the field `iz` in `myStruct` with the actual enumerated name `Simulink`. If you look into `ccstut.c`, you see that `iz` is an enumerated datatype. That feature comes into play in the next steps.

10 `cfield = getmember(cvar, 'iz')`

`cfield`, the object returned by `getmember`, represents the embedded variable `iz` in the project. Here's what `cfield` looks like in property form.

```
ENUM Object stored in memory:
Symbol name           : iz
Address               : [ 44312 0]
Word size            : 32 bits
Address units per value : 4 au
Representation       : signed
Size                 : [ 1 ]
Total address units  : 4 au
Array ordering       : row-major
Endianness           : little
Labels & values      : MATLAB=0, Simulink=1, SignalToolbox=2,
MatlabLink=3, EmbeddedTargetC6x=4
```

```
11 write(cfield,4)
```

```
12 read(cvar)
```

```
ans =  
  
    iy: [2x3 double]  
    iz: 'EmbeddedTargetC6x'
```

Your command `write(cfield,4)` replaced the string `MatlabLink` with the fourth value `EmbeddedTargetC6x`. That is an example of writing to an embedded variable by value.

```
13 cstring = createobj(cc,'myString')
```

`createobj` returns the object `cstring` that represents a C structure embedded in the project. When you leave off the closing semicolon (;) on the command, you see

```
STRING Object stored in memory:  
Symbol name           : myString  
Address               : [ 44360 0]  
Word size             : 8 bits  
Address units per value : 1 au  
Representation        : signed  
Size                  : [ 29 ]  
Total address units   : 29 au  
Array ordering        : row-major  
Endianness            : little  
Char conversion type  : ASCII
```

which provides details about `cstring`. Using `get` with `cstring` returns the same information, plus more, in a form listing the property names and property values of `cstring`.

```
14 read(cstring)
```

In response you see the contents of `cstring`

```
ans =
```

Treat me like an ASCII String

15 `write(cstring,7,'ME')`

This changes the seventh element of `MyString` to `ME`. When you reread `cstring`, `me` should be replaced by `ME`, so the string becomes

Treat ME like an ASCII String

as you see in the next example.

16 `read(cstring)`

`ans =`

Treat ME like an ASCII String

17 `write(cstring,1,127)`

`write` changes the contents of the first element of `MyString` to the ASCII character 127 — a nonprinting character.

18 `readnumeric(cstring)`

Using `readnumeric` with a string object returns the numeric equivalent of the characters in `MyString`, as shown here.

`ans =`

Columns 1 through 12

127 114 101 97 116 32 77 69 32 108 105 107

Columns 13 through 24

101 32 97 110 32 65 78 83 73 32 83 116

Columns 25 through 29

114 105 110 103 0

Closing the Links or Cleaning Up CCS IDE

Objects that you create in Link for Code Composer Studio have COM handles to CCS. Until you delete these handles, the CCS process (`cc_app.exe` in the Windows Task Manager) remains in memory. Closing MATLAB removes these COM handles automatically, but there may be times when it helps to delete the handles manually, without quitting MATLAB.

Use `clear` to remove objects from your MATLAB workspace and to delete any handles they contain. `clear all` deletes everything in your workspace. When you need to retain your MATLAB data while deleting objects and handles, use `clear objname`. Note that this applies both to objects you create with `ccdsp` and `createobj`. To clean up the objects created during the tutorial, the tutorial program executes the command

```
clear cvar cfield uintcvar
```

at the prompt.

One more bit of clean up that this tutorial does is to close the project in CCS with the command

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

Finally, to delete your link to Code Composer, use `clear cc`.

Note If a link to CCS IDE exists when you close Code Composer Studio, the application does not close. Windows moves it to the background (it becomes invisible). Only after you clear all links to CCS IDE, or close MATLAB, does closing CCS IDE unload the application. You can see if CCS IDE is running in the background by checking in the Windows Task Manager. When CCS IDE is running, the entry `cc_app.exe` appears in the **Image Name** list on the **Processes** page.

Your development tutorial using CCS IDE is done.

During the tutorial you

- 1 Selected your target.

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

In future development work with your signal processing applications you follow the same set of tasks. Thus the tutorial provided here gives you a working process for using the Link for Code Composer Studio and your signal processing programs to develop programs for a range of Texas Instruments digital signal processors. While the target may change, and the program will change, the essentials of the process remain the same, as do the functions you use to interact with the processor and CCS IDE.

Getting Started with RTDX

The Link for Code Composer Studio and the links for CCS IDE and RTDX speed and enhance your ability to develop and deploy digital signal processing systems on Texas Instruments digital signal processors. By using MATLAB and the Link for Code Composer Studio, 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 target. Across the link, 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 and at least one DSP development board. You can use the hardware simulator in CCS IDE to run this tutorial. Within the tutorial we use the TMS320C6711 DSK as the target board, with the C6711 DSP on the C6711 DSK as the target 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 to work 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. Once 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.

Link for Code Composer Studio and the links for RTDX and CCS IDE ease the job of taking algorithms from the model realm to the real world of the target digital signal processor on which the algorithm will run.

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

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

- From links for CCS IDE
 - `ccsdsp` — Create links to CCS IDE and RTDX.
 - `cd` — Change your CCS IDE working directory from MATLAB.
 - `open` — Load program files in CCS IDE.
 - `run` — Run processes on the target processor.
- From the RTDX class
 - `close` — Close the RTDX links between MATLAB and your target.
 - `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, `disp` (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 target.
- `isenabled` — Determine whether channels are enabled for RTDX communications.
- `isreadable` — Determine whether MATLAB can read the specified memory location.
- `iswritable` — Determine whether MATLAB can write to the target.
- `msgcount` — Find out how many messages are waiting in a channel queue.
- `open` — Open channels in RTDX.
- `readmat` — Read data matrices from the target into MATLAB as an array.
- `readmsg` — Read one or more messages from a channel.
- `writemsg` — Write messages to the target over a channel.

This tutorial provides the following procedure to show you how to use many of the functions in the links. By doing the steps listed, you can work through many of the operations yourself. As a bonus, the tutorial follows the general task flow for developing digital signal processing programs through testing with the links for RTDX.

Four tasks comprise this tutorial:

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

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

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

Notice that 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 target exceeds the rate at which MATLAB can capture the data.

- 3** Run your application on the target. At this stage you use MATLAB to investigate the results of your running process.

The previous tasks are common to all projects where you use RTDX to communicate with a target. While this step is also common to all development projects, the program used and the methods and details are up to you.

- 4** Close the links to the target and clean up the links and associated debris left over from your work.

Once again, all projects end with these tasks. Closing channels and cleaning up the memory and links you created ensures that CCS IDE, RTDX, and the Link for Code Composer Studio are ready for the next time you start development on a project.

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

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

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

Note To be able to open and enable channels over a link to RTDX, the program loaded on your target 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); % Target reads from this.  
    rtdx_CreateOutputChannel(ochan); % Target 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 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 target.

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, we use

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

to open and configure `ichan` for write mode. 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 follows

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

Creating the Links

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

Note Program `rtdxtutorial_6xevm.out` targets the C6711 DSK. We compiled, built, and linked the program as an executable to run on the C6711 digital signal processor. To use the tutorial without changes, target your C6711 DSK when you define properties `boardnum` and `procnum`.

Before continuing with this tutorial, you must load a valid GEL file to configure the EMIF registers of your target and perform any required processor initialization steps. Default GEL files provided by Code Composer Studio are stored in `..\cc\gel` in the folder where you installed Code Composer Studio. 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 target by creating a link to CCS IDE. Start by clearing all existing handles and setting `echo` on so you see functions in the M-file execute as the program runs:

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

`clear all` has the side effect of removing debugging breakpoints and resetting persistent variables since function breakpoints and persistent variables are cleared whenever the M-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 target board and processor by typing

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

boardnum defines which board the new link accesses. In this example, boardnum is 0. The Link for Code Composer Studio 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

```
ccsboardinfo
```

at the prompt, the Link for Code Composer Studio 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 target file, change the path for MATLAB to be able to find the file.

```
projname =  
  
C:\Temp\LinkForCCSDemos_v2.1\rtdxtutorial\c6x\c64xp\rtdxtut_sim.pjt  
  
outFile =  
  
C:\Temp\LinkForCCSDemos_v2.1\rtdxtutorial\c6x\c64xp\rtdxtut_sim.out  
  
target_dir = demoPjt.DemoDir  
  
target_dir =  
  
C:\Temp\LinkForCCSDemos_v2.1\rtdxtutorial\c6x\c64xp  
  
% Go to target directory
```

```
cd(cc,target_dir);cd(cc,tgt_dir); % Or cc.cd(tgt_dir)
dir(cc); % Or cc.dir
```

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

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

    isLibProj: 0
  TemplateProject: 'C:\Temp\LinkForCCSDemos_v2.1\template\c6x\c64x.pjt'
      DemoDir: 'C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp'
  ProjectFile: 'C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp\ccstut.pjt'
  ProgramFile: 'C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp\ccstut.out'
      SrcFile: {'W:\bat\Adsphw\perfect\matlab\toolbox\ccslink\ccsdemos\ccstutorial\
  LibFile: ''
      CmdFile: {'W:\bat\Adsphw\perfect\matlab\toolbox\ccslink\ccsdemos\shared\c6x\c6
  HdrFile: ''
      BuildOpts: [1x1 struct]
  ProjectAction: 'recreateProj-rebuildProg'
      RebuildDemo: 1

demoPjt.ProjectFile

ans =

C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp\ccstut.pjt

demoPjt.DemoDir

ans =

C:\Temp\LinkForCCSDemos_v2.1\ccstutorial\c6x\c64xp
```

Notice where the demo files are stored on your machine. In general, Link for CCS stores the demo project files in

```
LinkforCCS_vproduct_version
```

For example, if you are using version 2.1 of Link for CCS, the project demos are stored in `LinkforCCS_v2.1`. Link for CCS creates this directory to a location that you have write permissions on your machine. Usually, there are three possible locations where Link for CCS tries to create the demo directory, in the order shown.

- a** In the MATLAB work directory—`matlabroot\work`.
 - b** If `matlabroot\work` is not available, Link for CCS uses a temporary directory on the C drive, such as `C:\temp\`.
 - c** If Link for CCS cannot use the temp directory, you see dialog box that asks you to select a location to store the demos.
- 4** You have reset the directory 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 target 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 and your target.

Configuring Communications Channels

Communications channels to the target do not exist until you open and enable them through the Link for Code Composer Studio 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 target — `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 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 — read and write are defined as seen by the target. When you write data from MATLAB, you write to the channel that the target reads, `ichan` in this case. Conversely, when you read from the target, you read from `ochan`, the channel that the target writes to:

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

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

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

- 6** Reset the global timeout to 20 s to provide a little room for error. `ccsdsp` applies a default timeout value of 10 s. 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 20 s and that your link 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 target. You load the target, 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 the Link for Code Composer Studio 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 target for processing, then read data from the target after processing:

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

```
cc.restart
```

Restarting the target 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 so you can work in MATLAB while the program runs. Other options for the mode are

- 'runtohalt' — start to execute the program and wait to return control to MATLAB 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 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 target. Check that you can write to the target, then use `writemsg` to send the data. You do not need to type the if-test code shown.

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

We included the if-statement to simulate writing the data from within a MATLAB 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 target in a script or M-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 to print a message as it reads the data from the target by adding the function

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

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

When you write `indata` to the target, the following code running on the target 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` directory.

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

```
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 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. In the next few steps of this tutorial you read data from the `ochan` queue to different data formats within MATLAB.

- 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 target you need to tell MATLAB 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 target 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** The Link for Code Composer Studio provides a function for reading messages into matrices—`readmat`. Use `readmat` to read a message into a 5-by-2 matrix in MATLAB.

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

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

Since a 5-by-2 matrix requires ten elements, MATLAB 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, 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 Links 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 target 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 left-over handles, channels, and links from your 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 four 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

- `clear` — remove all RTDX objects and handles associated with a CCS and RTDX link. When you finish a session with RTDX, `clear` removes all traces of the specified link, or all links when you use the 'all' option in the syntax. When you clear one or more links, they no longer exist and cannot be reopened or used. If you are ending your programming session and do not want to retain any of the channels or links you created, use `clear` to end the RTDX communications and links and release all channels and resources associated with existing CCS IDE and RTDX links. You do not need to use the `close` or `disable` functions first.

To load a new program to a processor on which you have a program running, and to which you have links, you must clear the existing links before you load the new program to the target.

- `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 target.
- `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 and the target, and end your session:

- 1 Begin the process of shutting down the target and RTDX by stopping the target 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 target.

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

If necessary, using `disable` with channel name and target 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 the Link for Code Composer Studio 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 may exist but do not get closed correctly when you end your session.

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

```
if cc.isvisible,
```

```
cc.visible(1);  
end
```

Note Visibility can cause problems. When CCS IDE is running invisibly on your desktop, meaning you set `visibility` to 0, do not use `clear all` to get rid of your links for CCS IDE and RTDX. Without a link to CCS IDE you cannot access CCS IDE to change the visibility setting, or unload the application. To close CCS IDE when you do not have an existing link, either create a new link to CCS IDE, or use Windows Task Manager to end the process `cc_app.exe`, or close MATLAB.

- 5** You have finished the work in this tutorial, type the following to close all your remaining links to CCS IDE and release all the associated resources.

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

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

In future development work with your signal processing applications you follow the same set of tasks. Thus the tutorial provided here gives you a working process for using the Link for Code Composer Studio and your signal processing programs to develop programs for a range of Texas Instruments digital signal processors. While the target may change, the essentials of the process remain the same.

Listing the Functions for Links

To review a complete list of functions that operate on links, either CCS IDE or RTDX, type either

```
help ccsdsp
help rtdx
```

at the command line. If you already have a link `cc`, you can use dot notation to return the methods for CCS IDE or RTDX by entering

```
cc.methods or cc.rtdx.methods
```

at the prompt. In either instance MATLAB 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, type:

```
help ccsdsp
```

```
CCDSP - Base constructor for the 'Link to Code Composer Studio(tm)'
Description of methods available for CCSDSP
-----
ACTIVATE  Set the active project, text file or build configuration
ADD       Add source file to a project
ANIMATE   Initiate a target execution with breakpoint animation
ADDRESS   Search the target's symbol table for an address
BUILD     Compile/Link to build a program file
CCSDSP    Constructor which establishes the link to CCS
CD        Change or query working directory of Code Composer Studio
CLOSE     Close Code Composer Studio project or text file
CREATEOBJ Creates objects for manipulating target values
DELETE    Delete a debug point from DSP memory
DIR       List files in Code Composer Studio working directory
DISP     Display information about the CCSDSP object
GOTO     Executes the target to the entry of a function
HALT     Immediately terminate execution of the DSP processor
INFO     Produce a list of information about the target processor
INSERT    Insert a debug point into DSP memory
ISREADABLE Query if a block of DSP memory is available for reading
ISRUNNING Query status of DSP execution
```

ISRTDXCAPABLE Query if DSP supports RTDX communications
ISVISIBLE Query visibility of Code Composer Studio application
ISWRITABLE Query if a block of DSP memory is available for writing
LIST Produces various lists of information from Code Composer
LOAD Loads a program file into the DSP processor
NEW Create a default project, text file or build configuration
OPEN Loads a workspace, project or program file
PROFILE Return measurements from any DSP/BIOS(tm) STS objects
READ Return a block of data from the memory of the DSP
REGREAD Return data storied in a DSP register
REGWRITE Modify the contents of a DSP register
RELOAD Reload most recently loaded program file
REMOVE Remove a file from a project
RESTART Return PC to the beginning of a target program
RUN Initiates execution of the DSP processor
SAVE Save Code Composer Studio project or text file
SYMBOL Returns the target's entire symbol table
VISIBLE Hide or reveal Code Composer Studio application window
WRITE Places a block of Matlab data into the memory of the DSP

Constructing Link Objects

When you create a link to CCS IDE using the `ccsdsp` command, you are creating a “link to CCS IDE and RTDX interface” object (called a link object for brevity from here on). The link object implementation relies on MATLAB object-oriented programming capabilities similar to the objects you find in the Filter Design and Control Systems Toolboxes.

The discussions in this section apply to the link objects in the Link for Code Composer Studio. For a discussion of the embedded objects that are also part of this product, refer to “Introduction to Objects” on page 2-3. Since both object types use the MATLAB programming techniques, the information about working with the links, such as how you get or set properties, or use methods, apply equally to the link objects and the embedded objects. Only their constructors, properties, and methods are different.

Like other MATLAB structures, objects (also called links; we use the terms interchangeably here) in the Link for Code Composer Studio Development Tools have predefined fields called object properties.

If you are new to objects, you might find the glossary section, “Some Object-Oriented Programming Terms ” on page 2-5, helpful to explain the terms used in this Guide.

You specify object property values by either

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

For examples of setting link properties, refer to “Setting Property Values with `set`” on page 1-64.

Example – Constructor for Links

The easiest way to create a link object is to use the function `ccsdsp` to create a link with the default properties. Create a link named `cc` to CCS IDE by typing

```
cc = ccsdsp
```

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

```
CCSDSP 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 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 type

```
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 link properties are described in Chapter 4, “Functions — By Category”, and in more detail in “Link Properties” on page 1-69. These properties are set to default values when you construct links.

Properties and Property Values

Links (or objects) in this Link for Code Composer Studio 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 target processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

Setting and Retrieving Property Values

You can set Link for Code Composer Studio for Texas Instruments DSP link property values either:

- Directly when you create the link—“Setting Property Values Directly at Construction” on page 1-63
- By using the set function with an existing link “Setting Property Values with set” on page 1-64

Retrieve CCS IDE link property values with the get function.

In addition, direct property referencing lets you either set or retrieve property values for links.

Setting Property Values Directly at Construction

To set property values directly when you construct a link, include the following pair of entries in the input argument list for the link construction function `ccsdsp`:

- A string for the property name to set followed by a comma. Enclose the string in single quotation marks as you do any string in MATLAB.
- The associated property value. Sometimes this value is also a string.

Include as many property names in the argument list for the object construction command as there are properties to set directly.

Example — Setting Link Property Values at Construction

Suppose you want to set the following link characteristics when you create a link to a DSP on a board in your computer:

- Link to the second DSP board installed on your computer.
- Connect to the first processor on the target board.
- Set the global timeout to 5 s. The default is 10 s.

Do this by typing

```
cc = ccsdsp('boardname',1,'procnum',0,'timeout',5);
```

boardname, procnum, and timeout properties are described in “Link Properties” on page 1-69, as are the other properties for links.

Note When you set link property values, the strings for property names and their values are not sensitive to the case of the string. In addition, you only need to type the shortest uniquely identifying string in each case. For example, you could have typed the above code as

```
cc = ccsdsp('board',1,'proc',0,'tim',5);
```

Setting Property Values with set

Once you construct a link, the set function lets you modify its property values.

You can use the set function to both

- Set link property values
- “Example — Displaying Properties Using set” on page 1-65

Example — Setting Link Property Values Using set

For example, set the timeout specification for the link cc from the previous section.

To do this, type

```
set(cc,'time',8);
```

Now use `get` to check that the desired changes have been made to `cc`.

```
get(cc)
```

```
ans =
```

```
      rtdx: [1x1 rtdx]
apiversion: [1 0]
  ccsappexe: []
  boardnum: 0
  procnum: 0
  timeout: 8
  page: 0
```

Notice that the display reflects the changes in the property values.

Example – Displaying Properties Using set

To display a listing of all of the properties associated with a link `cc` that you can set, enter

```
set(cc)
```

```
ans =
```

```
      rtdx: [1x1 rtdx]
apiversion: [1 0]
  ccsappexe: []
  boardnum: 0
  procnum: 0
  timeout: 10
  page: 0
```

Retrieving Properties with get

You can use the `get` command to

- Retrieve property values for an object

- Display a listing of the properties associated with an object and their associated property values

Example – Retrieving Link Property Values Using get

For example, to retrieve the value of the `apiversion` property for `cc`, and assign it to a variable, type

```
v = get(cc, 'apiversion')  
  
ans =  
  
    1     0
```

Note When you retrieve properties, the strings for property names and their values are not case-sensitive. In addition, you only need to type the shortest uniquely identifying string in each case. For example, you could have typed the above code as

```
v = get(cc, 'api');
```

Example – Displaying Link Property Values Using get

To list the properties of a link `cc`, and their values, enter

```
get(cc)  
  
ansrtdx: [1x1 rtdx]  
  
    rtdx: [1x1 rtdx]  
  apiversion: [1 0]  
  ccsappexe: []  
  boardnum: 0  
  procnum: 0  
  timeout: 10  
  page: 0
```

Direct Property Referencing to Set and Get Values

You can reference directly into a property for setting or retrieving property values using MATLAB structure-like referencing. Do this by using a period to index into an object property by name.

Example — Direct Property Referencing in Links

- 1 Create a link with default values.
- 2 Change its timeout and number of open channels.

```
cc = ccstdsp;  
cc.time = 6;  
cc.rtdx.numchannels = 4;
```

Notice that you do not have to type the full name of the timeout property name, and you can use lower case to refer to the property name.

To retrieve property values, you can use direct property referencing.

```
num = cc.rtdx.numchannels  
  
num =  
    4
```

Overloaded Functions for Links

Several functions in this Link for Code Composer Studio have the same name as functions in other MathWorks toolboxes or in MATLAB. 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 link objects (links). Once 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 target processor) directly to the variable name you assign to your link, without having to specify your link parameters again.

For a complete list of the functions that act on links, refer to the tables of functions in the function reference pages.

Link Properties

The Link for Code Composer Studio provides links to your target hardware so you can communicate with processors for which you are developing systems and algorithms. Each link comprises two objects — a CCS IDE object and an RTDX interface object. The link objects are not separable; the RTDX object is a subclass of the CCS IDE object. Each of the link objects has multiple properties. To configure the links for CCS IDE and RTDX, you set parameters that define details such as the desired target board, the target processor, the timeout period applied for communications operations, and a number of other values. Since the Link for Code Composer Studio uses objects to create the links, 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 links 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 users may find much of this handling of objects familiar. Objects, or links as we call them in the Link for Code Composer Studio, behave like objects in MATLAB and the other object-oriented toolboxes. For C++ programmers, this discussion of object-oriented programming is likely to be a review.

Quick Reference to Link Properties

The following table lists the properties for the links in the Link for Code Composer Studio. The second column tells you which object the property belongs to. Knowing which property belongs to each object in a link tells you how to access the property.

Property Name	Applies to Which Link?	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 the Link 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 10 s.)

Details of the properties associated with links 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 COM server creates the handles when you create links 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)
```

```
CCSDSP 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 target board referenced by a link for CCS IDE. When you create a link, you use `boardnum` to specify the board you are targeting. To get the value for `boardnum`, use `ccsboardinfo` or the CCS Setup utility from Texas Instruments. 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 `ccsdsp` to create a link, MATLAB 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 `get` or `display` to see the RTDX link properties.

```
cc=ccsdsp

CCSDSP 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 a link, you use procnum to specify the processor you are targeting. 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.

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 ccscdsp 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 the Link for Code Composer Studio, 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 target, 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 link, `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 `appversion` value for link `rx`.

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

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

Link Objects

These sections of the User's Guide introduce both the objects you use to work with the software and tutorial and reference pages for using the objects.

Introduction to Objects (p. 2-3)	Object classes that compose Link for Code Composer Studio
Numeric Objects — Their Methods and Properties (p. 2-15)	Reference numeric data in memory
Bitfield Objects — Their Methods and Properties (p. 2-18)	Objects that reference bitfield data in memory
Enum Objects — Their Methods and Properties (p. 2-21)	Objects that reference enumerated data in memory
Pointer Objects — Their Methods and Properties (p. 2-24)	Reference pointers in memory
String Objects — Their Methods and Properties (p. 2-27)	Introduces objects that reference strings in memory
Rnumeric Objects — Their Methods and Properties (p. 2-30)	Introduces objects that reference numeric data in registers
Renum Objects — Their Methods and Properties (p. 2-33)	Objects that reference enumerated data in registers
Rpointer Objects — Their Methods and Properties (p. 2-36)	Introduces objects that reference pointers in registers
Rstring Objects — Their Methods and Properties (p. 2-39)	Introduces objects that reference strings in registers

Function Objects — Their Methods and Properties (p. 2-42)	About objects that reference functions, either ANSIC or assembly (that have C prototypes), in your project
Structure Objects — Their Methods and Properties (p. 2-46)	Introduces objects that reference data structures
Type Objects — Their Methods and Properties (p. 2-52)	Objects that reference typedefs in your project source code
Constructing Objects That Access Bitfields (p. 2-54)	Introduces the concepts behind using bitfield objects
Creating function Objects (p. 2-56)	Provides an extensive introduction to function objects
Creating Type Objects (p. 2-74)	Shows some of the ways you use type objects in projects
Tutorial — Using function Objects and Function Calls (p. 2-77)	Run a tutorial that shows you how to work with function objects and use them in a hardware-in-the-loop fashion
Managing Custom Data Types with the Data Type Manager (p. 2-109)	Describes and demonstrates how you use custom data types that are part of your projects
Reference for the Properties of Embedded Objects (p. 2-120)	Provides a comprehensive property reference

Introduction to Objects

Within your Link for Code Composer Studio Development Tools software, the links and the objects use object-oriented programming techniques. Along with the link object you use to connect MATLAB to your target hardware, Link for Code Composer Studio provides many objects for creating, accessing (reading from and writing to), and manipulating (changing the contents of in MATLAB) all the symbols in the symbol table for a program loaded on your signal processor. Within the table, each object in the class name column provides access to objects as described.

Class Name	Inherits From	Description
bitfield	memoryobj class	Access the contents of a bitfield defined in your code
enum	numeric class	Contents of an enumerated data type stored in memory defined in your code
function	None	Contents of a function in your source code, or used in your project as a library function. Can also represent new functions you develop and add to your project.
numeric	memoryobj class	Access the contents of a numeric data type stored in memory defined in your code
pointer	numeric class	Contents of a pointer stored in a memory location on your target
renum	rnumeric class	Contents of an enumerated data type stored in a register on your target
rnumeric	registerobj class	Contents of register that contains a numeric data type
rpointer	rnumeric class	Contents of a pointer stored in a register on your target

Class Name	Inherits From	Description
rstring	rnumeric class	Contents of a string stored in a register on your target
string	numeric class	Contents of a string stored in a memory location on your target
structure	None	Contents of a structure stored in memory on your target
type	None	Typedefs stored in memory on your target after you add them to the type object

In the Inherits From column you see the name of another class. Classes that inherit from another class contain all the properties and methods of the Inherited From class as well as their own unique properties. Note that although object and class seem to be interchangeable, objects are instances of classes — the properties of a class are the properties of an instance of the class, an object. This guide treats the distinction fairly loosely, using object in most instances.

For example, the String object has the properties and methods of the Numeric class, and its own properties and methods.

By using the objects provided, you can modify and view any and all symbols from MATLAB.

Each of the objects has properties and methods specific to its use, although many of the objects use the same methods and properties, as you see in the next sections.

While you can use the Link for Code Composer Studio software without knowing about its object-oriented design and implementation, you might find the next sections about objects useful to gain a better understanding of the objects.

Some Object-Oriented Programming Terms

As an object-oriented software package, describing how to use the Link for Code Composer Studio requires discussing the objects, classes, properties, and methods you use to manipulate and access data. To ensure we use the same terms and understand them in the same way, this section provides definitions of some terms commonly used throughout the this guide.

For more information about objects and working with their properties and methods (or functions), refer to “Constructing Link Objects” on page 1-61.

Note Except for read and write, all functions that work with objects operate solely in your MATLAB workspace. They do not affect the data stored in memory, registers, functions, or structures on your signal processor and in CCS. Only read and write allow you to access and change information on your target or in your project in CCS.

Definitions of Object-Oriented Terms

Abstract class	A class without instances. Abstract classes expect that their concrete subclasses will add to their structure and behavior.
Aggregation	The part-of relationship between two objects. For example, a bicycle has wheels, so wheels are part of a bicycle. Note that the wheels can exist separately from the bicycle. Compare to composition.
Base class	The most general class in a class structure. Also called root classes, most applications or systems have more than one base class.
Behavior	How an object reacts to its methods. How the object state changes in response to one of its methods acting on it.

Class	A set of abstract objects that share a common structure and behavior. A class forms the prototype that defines the properties and methods common to all objects of the class. Types and classes are not quite the same, but are used interchangeably in this guide.
Class diagram	Used to show the existence of classes and their relationships. Class diagrams can represent part or all of the class structure of a system.
Composition	A relationship between objects where one part object exists only as part of the whole object. The parts live and die together. You create and destroy them as one.
Constructor	A function that creates an object and initializes its state. Constructors can also initialize the state without creating the object.
Container Class	A class whose instances are collections of other objects in the system. Also called a package.
Function	Same as method. Used in MATLAB for consistency with other functions. Functions and methods are not quite the same, but are used interchangeably.
Handle	A means to access any object that Link for Code Composer Studio creates. Used in this guide to refer to the object. Often the handle is the name you assign when you create the object. For example, cc is the object and handle when you create a link object.

Inheritance	A relationship between classes. One class shares the structure (properties) and behavior (methods) defined in one or more other classes. Subclasses inherit from one or more superclasses, typically augmenting the superclass with their own properties and methods.
Instance	Something you can operate on. Instance and object are synonyms and this guide uses them interchangeably. Instantiate is the verb form — to create an instance of a class or object.
Instantiation	To create an object — an instance of a class.
Method	An operation on an object, defined as part of the class of the object. We call this a function.
Object	Something you can operate on. Objects that are the same class share similar structure and behavior. An object is a collection of properties and methods. Some programming sources call properties “variables.” In all cases, an object is an instance of a class. Classes are abstract; objects are not.
Object Diagram	Shows the existence of objects and their relationships in the logical design of a system. Object diagrams can represent part or all of the class structure of a system.

Object-based Programming	<p>Programming style that organizes programs as cooperative collections of objects.</p> <p>Each object represents an instances of a type; where the types are members of an hierarchy, united through relationships that are not inheritance relationships. Compare to object-oriented programming.</p>
Object-oriented Programming	<p>Programming implementation that organizes programs as cooperative collections of objects.</p> <p>Each object represents an instance of some class, and the classes are members of an hierarchy of classes united through inheritance relationships. Compare to object-based programming.</p>
Property	<p>Part of an object — treated as a variable at times. Also called attribute, it is part of the structure that defines the state of an object.</p>
Subclass	<p>A class that inherits from one or more classes, called its superclasses.</p>
Superclass	<p>A class that other classes inherit from. The inheriting classes are called subclasses.</p>
State	<p>The accumulated results of the behavior of an object. At any time, the state of an object encompasses the properties of the object and the values for each of the properties.</p>
Structure	<p>The concrete representation of the state of an object.</p>

Determining an Object Class

After you create an object, use `whos` to determine the class for your new object (although you should know the class from the input argument you provided to `createobj`). Being able to query the class for an object is particularly important in this case because the constructor `createobj` determines the

class of the object created — you cannot specify the object class. Depending on the input symbol name you provide to `createobj`, the returned class changes. So you need to be able to determine the class. `whos` lets you do this.

If you use the MATLAB Workspace browser, your object appears in the list of the contents of your workspace, indicating the object type and class — just like `whos`.

Alternatively, using `createobj` or `ccsdsp` without the closing semicolon (;) at the end of the command directs MATLAB to display the properties of your new object in the MATLAB window when you create the object.

About the Relationships Between Objects

Link for Code Composer Studio uses objects exclusively to access and manipulate complex data structures and functions, among other programming constructs, in your project and code. Many of the objects inherit properties and functions, also called methods, from other objects. The class diagrams and tables presented in the next sections discuss and show the relationships between the objects that you create when you use `createobj`.

The Base Classes

Class Name	Description
Memoryobj	An abstract class. The numeric and bitfield classes inherit properties and methods from this class, making this a superclass. You cannot create an instance of this class. Subclasses of the memoryobj class always describe objects that reside in DSP memory on your target.
Registerobj	An abstract class. The rnumeric class inherits properties and methods from this class, making this a superclass. You cannot create an instance of this class. Subclasses of the registerobj class always describe objects that reside in DSP registers on your target.

The Subclasses

Class Name	Description
Numeric	A superclass from which the enum, pointer, and string subclasses inherit properties and methods. You can create an object of this class using <code>createobj</code> . Numeric inherits from the abstract class <code>memoryobj</code> .
Enum	A subclass of the numeric class. You can create an object of this class using <code>createobj</code> .
Pointer	A subclass of the numeric class. You can create an object of this class using <code>createobj</code> .
String	A subclass of the numeric class. You can create an object of this class using <code>createobj</code> .
Bitfield	A subclass of the <code>memoryobj</code> class. You can use <code>createobj</code> to make a bitfield object.
Rnumeric	A superclass from which the <code>renum</code> , <code>rpointer</code> , and <code>rstring</code> subclasses inherit properties and methods. You can create an object of this class using <code>createobj</code> . Rnumeric inherits from the abstract class <code>registerobj</code> .
Renum	A subclass of the <code>registerobj</code> class. You can create an object of this class using <code>createobj</code> .
Rpointer	A subclass of the <code>registerobj</code> class. You can create an object of this class using <code>createobj</code> .
Rstring	A subclass of the <code>registerobj</code> class. You can create an object of this class using <code>createobj</code> .

Other Classes

Class Name	Description
Function	A class containing information about a function in your project. <code>createobj</code> constructs this class directly.

Other Classes (Continued)

Class Name	Description
Structure	A class containing information about a structure in memory on your target. <code>createobj</code> constructs this class directly.
Type	A class containing information about C type definitions in the source code for your project. Type objects are composition objects to <code>ccsdsp</code> objects. When you create a <code>ccsdsp</code> object, it includes a type object.

Class Diagrams for the Link for Code Composer Studio

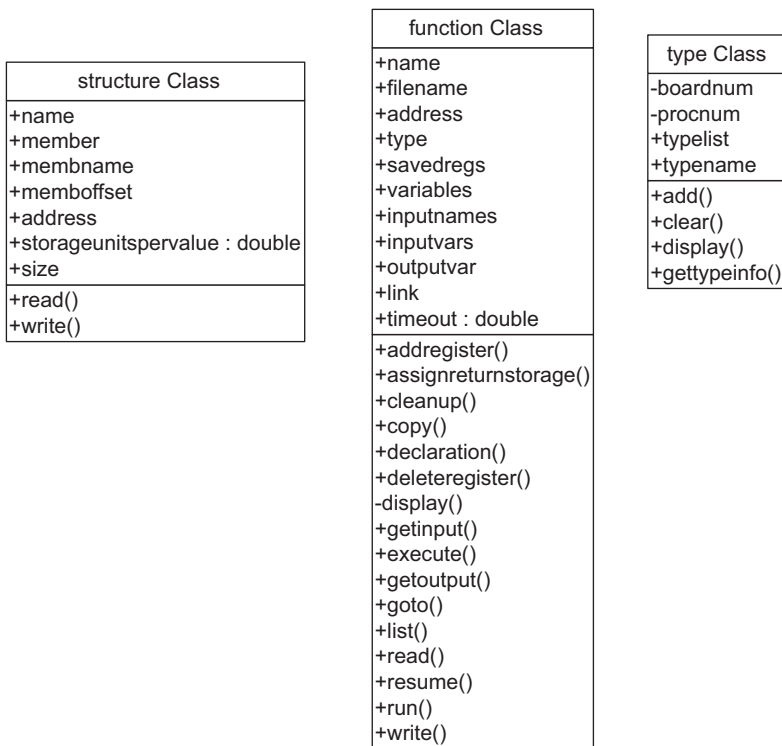
One of the most important features of object-oriented programming is the relationship between the classes that compose the system. Class relationships lend themselves to a graphical layout like a tree structure, where the structure of the tree shows clearly the super classes and subclasses, the base classes, and the other classes. In addition, the diagrams can show the properties and methods for each class, and where a subclass adds properties and methods to those it inherits from its superclass.

The following figures show the methods and properties of each class or object. For short descriptions about the properties for each class, refer to the tables in the following sections:

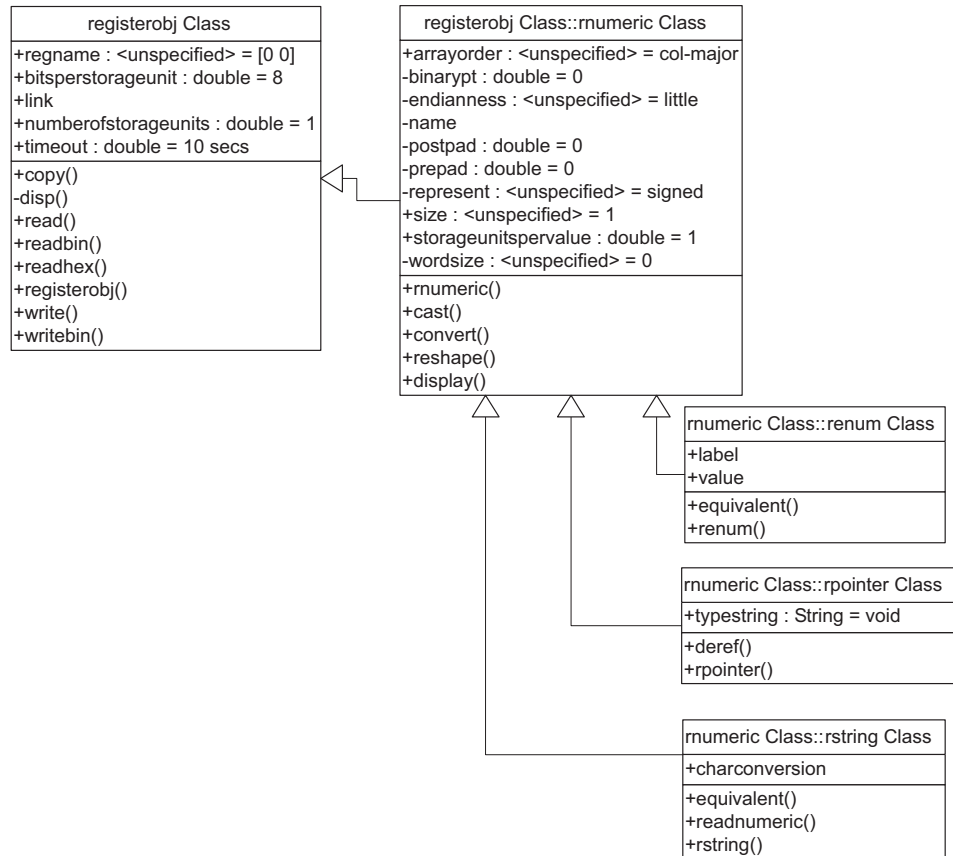
- “Numeric Objects — Their Methods and Properties” on page 2-15
- “Bitfield Objects — Their Methods and Properties” on page 2-18
- “Enum Objects — Their Methods and Properties” on page 2-21
- “Pointer Objects — Their Methods and Properties” on page 2-24
- “String Objects — Their Methods and Properties” on page 2-27
- “Rnumeric Objects — Their Methods and Properties” on page 2-30
- “Renum Objects — Their Methods and Properties” on page 2-33
- “Rpointer Objects — Their Methods and Properties” on page 2-36

- “Rstring Objects — Their Methods and Properties” on page 2-39
- “Function Objects — Their Methods and Properties” on page 2-42
- “Structure Objects — Their Methods and Properties” on page 2-46
- “Type Objects — Their Methods and Properties” on page 2-52

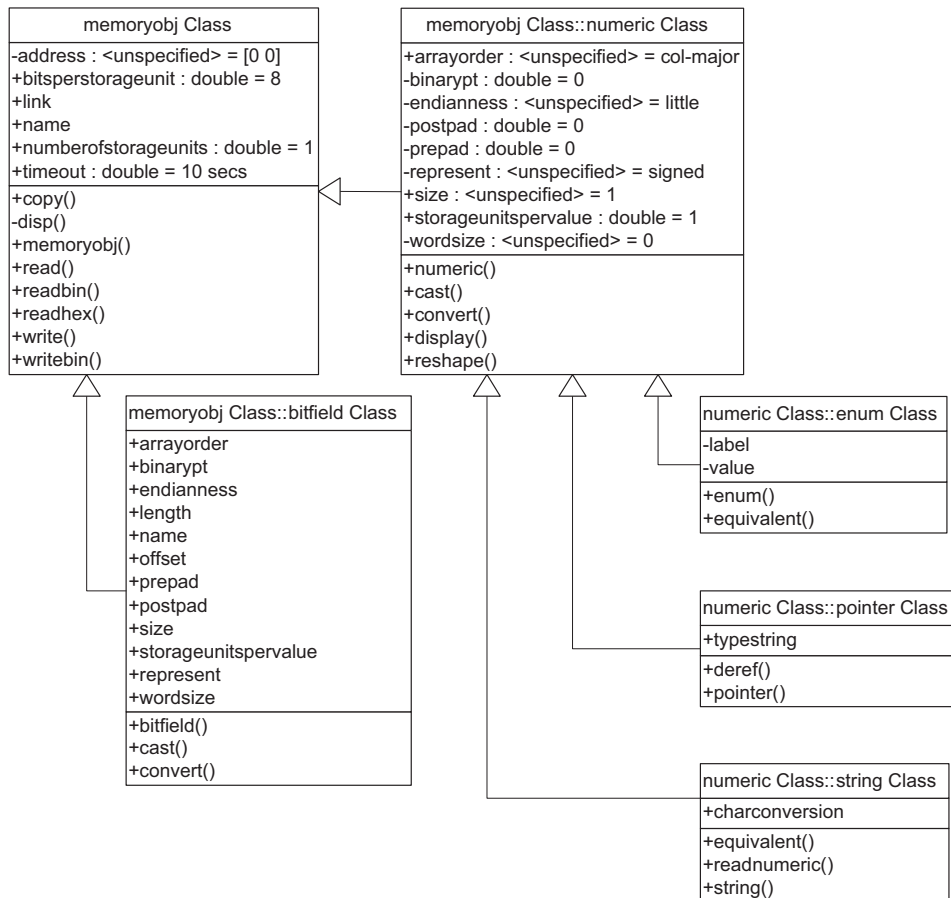
Detailed descriptions of the properties appear in the section “Reference for the Properties of Embedded Objects” on page 2-120.



Class Diagram of the Memory Class



Class Diagram of the Structure, Function, and Type Classes



Class Diagram of the Register Class

Numeric Objects — Their Methods and Properties

When you create an object that accesses a numeric symbol in your source code, the object constructor `createobj` returns a numeric object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the numeric class, numeric objects inherit properties and methods from the memory class.

Properties of Numeric Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>address</code>	<code>mxAarray</code>	<code>[0 0]</code>	Memory address of the symbol, in [Offset Page] format
<code>arrayorder</code>	<code>{col-major, row-major}</code>	<code>col-major</code>	Ordering of values when moving data from linear memory storage to N-D arrays in MATLAB
<code>bitsperstorageunit</code>	<code>double</code>	<code>8</code>	Bits per smallest addressable unit in the signal processor
<code>endianness</code>	<code>{little, big}</code>	<code>'little'</code>	Specifies whether the data is stored as little endian or big endian data
<code>link</code>	MATLAB handle	None	Object handle that identifies the object

Property Name	Property Type	Default Value	Description
name	string	None	Name of the embedded symbol in the symbol table
numberofstorageunits	double	1	Number of storage units needed to represent the object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the data type of the values
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory
storageunitspervalue	int	No default	Addressable units (au) per value in memory. Value depends on the processor. May be less than one when you use bit packing
timeout	double	10 seconds	Time-out period for link methods

Methods of Numeric Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the cast and convert methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
cast	Yes	Copies an object and change the data type for a value at the same time
convert	No	Changes the data type for a value
display	Yes	Display the properties of the numeric object
reshape	Yes	Changes the dimensions of the array that contains the data in MATLAB

Bitfield Objects – Their Methods and Properties

When you create an object that accesses a bitfield symbol in your source code, the object constructor `createobj` returns a struct object that includes the bitfield as members of the struct object. Bitfields are always parts of structures, so you create struct objects to access bitfields. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values, you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the class, `bitfield` objects inherit properties and methods from the numeric and `memoryobj` classes.

Properties of Bitfield Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>address</code>	<code>mxArray</code>	<code>[0 0]</code>	Memory address of the symbol, in [Offset Page] format
<code>arrayorder</code>	<code>{col-major, row-major}</code>	<code>row-major</code>	Ordering of values when moving data from linear memory storage to N-D arrays in MATLAB
<code>binarypt</code>	<code>int</code>	<code>0</code>	Location of the binary point for fractional data types
<code>bitsperstorageunit</code>	<code>double</code>	<code>8</code>	Bits per addressable unit in the signal processor
<code>endianness</code>	<code>{little, big}</code>	<code>little</code>	Specifies whether the data is stored as little endian or big endian data

Property Name	Property Type	Default Value	Description
length	int	0	Number of bits in the bitfield
link	MATLAB handle	None	Object handle that identifies the object
name	string	None	Name of the embedded symbol in the symbol table
numberofstorageunits	double	1	Number of memory units needed to represent the object
offset	int	0	Starting point of the bitfield in relation to bit 0
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the data type of the values
size	mxArray	1	Size of the array created in MATLAB from the data received from memory
storageunitspervalue	int	32	Addressable units (au) per value in memory. May be less than one when you use bit packing

Property Name	Property Type	Default Value	Description
timeout	double	10 seconds	Time-out period for link methods
wordsize	int	32	Number of bits in a word for the processor

Methods of Bitfield Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself.

Name	Overloaded?	Description
cast	Yes	Copies an object and change the data type for a value at the same time
convert	No	Changes the data type for a value
copy	Yes	Copies an existing object by creating a new pointer to the object
display	Yes	Displays the properties of the object
read	Yes	Returns the contents of the memory location specified by the symbol
write	Yes	Writes one or more values to the memory location

Enum Objects — Their Methods and Properties

When you create an object that accesses an enumerated symbol in your source code, the object constructor `createobj` returns an enum object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the enum class, enum objects inherit properties and methods from the numeric and memoryobj classes.

Properties of Enum Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>address</code>	<code>mxAarray</code>	<code>[0 0]</code>	Memory address of the symbol, in [Offset Page] format
<code>arrayorder</code>	<code>{col-major, row-major}</code>	<code>row-major</code>	Ordering of values when moving data from linear memory storage to N-D arrays in MATLAB
<code>bitsperstorageunit</code>	<code>double</code>	<code>8</code>	Bits per smallest addressable unit in the signal processor
<code>endianness</code>	<code>{little, big}</code>	<code>little</code>	Specifies whether the data is stored as little endian or big endian data
<code>label</code>	<code>mxAarray</code>	<code>N/A</code>	Lists the enumerated labels for the object

Property Name	Property Type	Default Value	Description
link	MATLAB handle	None	Object handle that identifies the object
name	string	None	Name of the embedded symbol in the symbol table
numberofstorageunits	double	1	Number of memory units needed to represent the object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values.
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the data type of the values
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory
storageunitspervalue	int	32	Addressable units (au) per value in memory. May be less than one when you use bit packing
timeout	double	10 seconds	Time-out period for link methods
value	mxArray	0	Contains a vector of the enumerated type

Methods of Enum Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>display</code>	Yes	Displays the properties of the object
<code>equivalent</code>	No	Returns the equivalent string or numeric value based on the input argument

Pointer Objects – Their Methods and Properties

When you create an object that accesses a pointer symbol in your source code, the object constructor `createobj` returns a pointer object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the pointer class, pointer objects inherit properties and methods from the numeric and memory classes.

Properties of Pointer Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>address</code>	<code>mxAarray</code>	<code>[0 0]</code>	Memory address of the symbol, in [Offset Page] format
<code>arrayorder</code>	<code>{ 'col-major' 'row-major' }</code>	<code>row-major</code>	Describes the ordering of the data moved from linear memory storage to n-dimensional arrays
<code>binarypt</code>	<code>int</code>	<code>0</code>	Locates binary point needed to interpret the value
<code>bitsperstorageunit</code>	<code>double</code>	<code>8</code>	Bits per smallest addressable unit in the signal processor
<code>endianness</code>	<code>character</code>	<code>little</code>	Specifies whether the data is stored as little endian or big endian data

Property Name	Property Type	Default Value	Description
name	mxArray	None	Name of the embedded symbol in the symbol table
numberofstorageunits	double	1	Number of memory units needed to represent the object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the representation of the values in the object
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory
storageunitspervalue	double	1	Addressable units per memory value in memory on the DSP
typestring	string	void	Specifies the type of data the pointer points to
wordsize	int	0	Valid bits per value (read-only)

Methods of Pointer Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>deref</code>	No	Returns the data to which the specified pointer points

String Objects — Their Methods and Properties

When you create an object that accesses a string symbol in your source code, the object constructor `createobj` returns a string object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the string class, string objects inherit properties and methods from the numeric and memory classes.

Properties of String Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>name</code>	<code>string</code>	None	Name of the embedded symbol in the symbol table
<code>address</code>	<code>mxArray</code>	[0 0]	Memory address of the symbol, in [Offset Page] format
<code>arrayorder</code>	{col-major, row-major}	row-major	Ordering of values when moving data from linear memory storage to N-D arrays in MATLAB
<code>bitsperstorageunit</code>	<code>double</code>	8	Bits per smallest addressable unit in the signal processor

Property Name	Property Type	Default Value	Description
charconversion	mxArray	ASCII	Conversion type of the characters in the object
endianness	{little, big}	little	Specifies whether the data is stored as little endian or big endian data
link	MATLAB handle	None	Object handle that identifies the object
numberofstorageunits	double	1	Number of units needed to represent the object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the representation of the values in the object
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory

Property Name	Property Type	Default Value	Description
storageunitspervalue	int	32	Addressable units (au) per value in memory. May be less than one when you use bit packing
timeout	double	10 seconds	Time-out period for link methods
wordsize	int	0	Valid bits per value (read-only)

Methods of String Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>equivalent</code>	Yes	Returns the equivalent numeric value for the input string
<code>readnumeric</code>	Yes	Returns the data in memory to MATLAB as numeric equivalent of the values on the target

Rnumeric Objects – Their Methods and Properties

When you create an object that accesses a numeric symbol stored in a register in your source code, the object constructor `createobj` returns an `rnumeric` object. `createobj` uses the information in your project to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the `rnumeric` class, `rnumeric` objects inherit properties and methods from the `register` class.

Classes that inherit from the `registerobj` base class always access data that resides in registers on the target, not in memory locations.

Properties of Rnumeric Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>arrayorder</code>	{ 'col-major' 'row-major' }	row-major	Describes the ordering of the data moved from linear memory storage to n-dimensional arrays
<code>binarypt</code>	int	0	Locates binary point needed to interpret fractional data types
<code>bitsperstorageunit</code>	double	8	Bits per smallest register unit in the signal processor
<code>link</code>	MATLAB handle	None	Object handle that identifies the object

Property Name	Property Type	Default Value	Description
name	string	None	Name of the register symbol in the symbol table
numberofstorageunits	double	1	Number of register units needed to represent the register object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
regname	mxArray	None	Name of the register on the signal processor
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the representation of the values in the object, such as numeric or string
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory
storageunitspervalue	double	1	Register units per register value in memory on the DSP

Property Name	Property Type	Default Value	Description
timeout	double	10 seconds	Time-out period for link methods
wordsize	int	0	Valid bits per value (read-only)

Methods of Rnumeric Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>cast</code>	No	Changes the data type of the input argument to another data type
<code>convert</code>	No	Converts the current data type to the specified data type
<code>display</code>		Displays the properties of the object
<code>read</code>	Yes	Returns the contents of the register location specified by the symbol
<code>reshape</code>	No	Reshapes the object in MATLAB
<code>write</code>	Yes	Writes one or more values to the register location

Renum Objects — Their Methods and Properties

When you create an object that accesses an enumerated symbol stored in a register in your source code, the object constructor `createobj` returns a `renum` object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the `renum` class, `renum` objects inherit properties and methods from the `rnumeric` and `register` classes.

Classes that inherit from the `registerobj` base class always access data that resides in registers on the target, not in memory locations.

Properties of Renum Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>arrayorder</code>	{ 'col-major' 'row-major' }	row-major	Describes the ordering of the data moved from linear memory storage to n-dimensional arrays
<code>binarypt</code>	int	0	Locates binary point needed to interpret fractional data types
<code>bitsperstorageunit</code>	double	8	Bits per smallest register unit in the signal processor
<code>label</code>	mxArray	N/A	Lists the enumerated labels for the object

Property Name	Property Type	Default Value	Description
link	MATLAB handle	None	Object handle that identifies the object
name	string	None	Name of the register symbol in the symbol table
numberofstorageunits	double	1	Number of register units needed to represent the register object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
regname	mxArray	None	Name of the register on the signal processor
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the representation of the values in the object, such as numeric or string
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory

Property Name	Property Type	Default Value	Description
storageunitspervalue	double	1	Register units per register value in memory on the DSP
timeout	double	10 seconds	Time-out period for link methods
value	mxArray	0	Contains a vector of the enumerated type
wordsize	int	0	Valid bits per value (read-only)

Methods of Renum Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>equivalent</code>	No	Returns the equivalent string or numeric
<code>read</code>	Yes	Returns the data from the register on the target
<code>write</code>	Yes	Writes one or more values to the register location

Rpointer Objects – Their Methods and Properties

When you create an object that accesses a pointer symbol stored in a register in your source code, the object constructor `createobj` returns an `rpointer` object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the `rpointer` class, `rpointer` objects inherit properties and methods from the `rnumeric` and `register` classes.

Classes that inherit from the `registerobj` base class always access data that resides in registers on the target, not in memory locations.

Properties of Rpointer Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>arrayorder</code>	{'col-major' 'row-major'}	row-major	Describes the ordering of the data moved from linear memory storage to n-dimensional arrays
<code>binarypt</code>	int	0	Locates binary point needed to interpret fractional data types
<code>bitsperstorageunit</code>	double	8	Bits per smallest register unit in the signal processor

Property Name	Property Type	Default Value	Description
link	MATLAB handle	None	Object handle that identifies the object
name	string	None	Name of the register symbol in the symbol table
numberofstorageunits	double	1	Number of register units needed to represent the register object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
regname	mxArray	None	Name of the register on the signal processor
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the representation of the values in the object, such as numeric or string
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory

Property Name	Property Type	Default Value	Description
storageunitspervalue	double	1	Register units per register value in memory on the DSP
timeout	double	10 seconds	Time-out period for link methods
typestring	string	void	Specifies the type of data the pointer points to
wordsize	int	0	Valid bits per value (read-only)

Methods of Rpointer Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the cast and convert methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
deref	No	Returns the data to which the specified pointer points
read	Yes	Returns the contents of the register location specified by the symbol
write	Yes	Writes one or more values to the register location

Rstring Objects — Their Methods and Properties

When you create an object that accesses a string symbol stored in a register in your source code, the object constructor `createobj` returns an `rstring` object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

To add to the properties of the `rstring` class, `rstring` objects inherit properties and methods from the `rnumeric` and `register` classes.

Classes that inherit from the `registerobj` base class always access data that resides in registers on the target, not in memory locations.

Properties of Rstring Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>arrayorder</code>	{ 'col-major' 'row-major' }	row-major	Describes the ordering of the data moved from linear memory storage to n-dimensional arrays
<code>binarypt</code>	int	0	Locates binary point needed to interpret fractional data types
<code>bitsperstorageunit</code>	double	8	Bits per smallest register unit in the signal processor
<code>charconversion</code>	mxArray	ASCII	Conversion character set applied for the characters in the referent string

Property Name	Property Type	Default Value	Description
link	MATLAB handle	None	Object handle that identifies the object
name	string	None	Name of the register symbol in the symbol table
numberofstorageunits	double	1	Number of register units needed to represent the register object
postpad	int	0	Number of bits of padding added at the end of the memory buffer. Added bits are ignored in final numeric values
prepad	int	0	Number of bits of padding added at the beginning of the memory buffer. Added bits are ignored in final numeric values
regname	mxArray	None	Name of the register on the signal processor
represent	{signed, unsigned, float, fract, ufract}	signed	Reports the representation of the values in the object, such as numeric or string
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory
storageunitspervalue	double	1	Register units per register value in memory on the DSP
timeout	double	10 seconds	Time-out period for link methods
wordsize	int	0	Valid bits per value (read-only)

Methods of Rstring Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>equivalent</code>	Yes	Returns the equivalent numeric value for the input string
<code>readnumeric</code>	Yes	Returns the data in memory as a numeric array in MATLAB
<code>write</code>	Yes	Writes data to memory on the target
<code>writebin</code>	Yes	Write datas to memory on the target as binary data – 0s and 1s

Function Objects – Their Methods and Properties

When you create an object that accesses a function declared in your source code, or a library function in your project, the object constructor `createobj` returns a function object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values, you find all the information about the function, so that MATLAB understands how to handle the function in your MATLAB workspace and how to run the function on your target.

Unlike memory and register objects, function objects do not inherit properties from a parent class.

Properties of Function Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
address	mxArray	0	Returns the starting memory address for the function
filename	string	None	Reports the name of the file that contains the function
inputnames	mxArray	ASCII	Lists the name of the input arguments for the function

Property Name	Property Type	Default Value	Description
inputvars	MATLAB handle	None	Handles to the objects that access each input argument to the function. Created when you create the function object
link	MATLAB handle	None	Identifies the name of the link object you used to create the associated embedded object
name	string	None	Name of the register symbol in the symbol table
outputvar	MATLAB handle	None	Handles to the object that accesses the output argument from the function. Created when you create the function object
savedregs	mxArray	ASCII	Lists the names of the processor registers that are saved during processing. Contents of saved registers are preserved after you run a function or program
timeout	double	10 s	Specifies how long MATLAB waits for calls to the function to complete their work

Property Name	Property Type	Default Value	Description
type	string	ASCII	Specifies the function return type
variables	mxArray	ASCII	Lists the names of variables in the function

Methods of Function Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself

Name	Overloaded?	Description
<code>addregister</code>	No	Adds registers to the saved register list
<code>cleanup</code>	No	Restore CCS to the state it was in before you ran a function. Restores the register contents to their previous state as well
<code>copy</code>	Yes	Makes a copy of the function object
<code>declare</code>	No	Provides a function declaration to MATLAB
<code>deleteregister</code>	No	Removes a register you added to the saved register list
<code>display</code>	Yes	Returns the properties of the function
<code>execute</code>	No	Runs a function or program
<code>getinput</code>	No	Gets information about one or more input arguments for a function
<code>getoutput</code>	No	Gets information about the output argument for a function

Name	Overloaded?	Description
goto	No	Positions the cursor in CCS IDE to the start of the specified function. This method does not initialize the function
list	Yes	Returns information about one or more variables in your function
read	Yes	Reads a value from memory on the processor
resume	Yes	Restarts execution of a paused or stopped process
run	Yes	Runs a program or function. Similar to execute
write	Yes	Write to the processor memory

Structure Objects – Their Methods and Properties

When you create an object that accesses a structure symbol declared in your source code, the object constructor `createobj` returns a structure object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values you find all the information about the symbol, so that MATLAB understands how to handle the symbol in your MATLAB workspace.

Like memory and register class objects, structure objects do not inherit properties from a parent class.

Properties of Structure Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>address</code>	<code>mxArray</code>	None	Address of the function
<code>arrayorder</code>	<code>{col-major, row-major}</code>	row-major	Ordering of values when moving data from linear memory storage to N-D arrays in MATLAB
<code>filename</code>	<code>mxArray</code>	None	Name of the file that contains the function
<code>member</code>	cell array	None	Object that contains a list of the structure members
<code>membname</code>	cell array	None	Object that contains the names of the members of the structure
<code>memboffset</code>	<code>int</code>	0	Offset of the member from the starting address of the structure

Property Name	Property Type	Default Value	Description
name	string	None	Name of the C or assembly function
numberofstorageunits	double	1	Number of memory units needed to represent the object
size	mxArray	1	Specifies the size of the array created in MATLAB from the data received from memory
storageunitspervalue	double	1	Memory units per value in memory on the DSP

Methods of Structure Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>copy</code>	Yes	Returns a copy of the object
<code>display</code>	Yes	Returns information about the object
<code>getmember</code>	No	Returns an object that accesses one member of a structure
<code>read</code>	Yes	Reads a structure from the symbol table
<code>write</code>	Yes	Writes changes or values to the structure in memory

Working with Structure Objects

structure objects present some unexpected behavior when you try to access the elements referred to by the structure object. Consider the following example that creates a structure object and accesses the members.

Suppose we have a structure variable in CCS:

```
creal32_T mw_output[512];
```

In MATLAB, create an object to `mw_output`. The size (512) is correctly propagated to the structure object:

```
a = createobj(cc, 'mw_output');
```

STRUCTURE Object stored in memory:

```
Symbol name      : mw_output
Address          : [ 2147652624 0]
Address units per value : 8 au
Size             : [ 512 ]
Total Address Units : 4096 au
Array ordering   : row-major
Members          : 're', 'im'
```

If you now look at a member `re` of the structure the size is not 512 any more. It returns a size of 1.

```
a.member.re
```

NUMERIC Object stored in memory:

```
Symbol name      : re
Address          : [ 2147652624 0]
Data type        : float
Word size        : 32 bits
Address units per value : 4 au
Representation    : float
Size             : [ 1 ]
Total address units : 4 au
Array ordering   : row-major
Endianness       : little
```

Now look at another member. Again, the size is 1.

```
a.member.im
```

NUMERIC Object stored in memory:

```
Symbol name      : im
```

```

Address           : [ 2147652628 0]
Data type        : float
Word size        : 32 bits
Address units per value : 4 au
Representation    : float
Size             : [ 1 ]
Total address units : 4 au
Array ordering    : row-major
Endianness       : little

```

The size of the members cannot reflect the size of the structure object.

If you have

```

struct tag {
    int a;
    int b;
} mystruct[10];

```

In memory, the values are arranged in the following way:

memory1: mystruct[0] -> member a value

memory2: mystruct[0] -> member b value

memory3: mystruct[1] -> member a value

memory4: mystruct[1] -> member b value

...

memory19: mystruct[9] -> member a value

memory20: mystruct[9] -> member b value

Therefore, when you do the following functions:

```

structobj = createobj(cc, 'mystruct')
aobj = structobj.member.a;

```

and `aobj.size` is same as `structobj.size`, you will be reading the wrong set of values, as shown by the example that shows the structure memory values and layout.

Setting the value of `aobj` to 1 provides a safe way to allow you to access the members of a structure.

The most reliable way to access a structure objects members is to read the structure into MATLAB and reference the members there. In the next code example you see how to do this.

```
x = createobj(cc,'m')

STRUCTURE Object stored in memory:
Symbol name           : m
Address               : [ 13544 0]
Address units per value : 12 au
Size                  : [ 2 ]
Total Address Units   : 12 au
Array ordering        : row-major
Members               : 'a', 'b'

out = read(x)

out =

1x2 struct array with fields:
    a
    b

out(1)

ans =

    a: 3
    b: 1.5000

out(2)

ans =
```



```
a: 12648430  
b: 1.7724e-038
```

```
read(x, 'a')
```

```
ans =
```

```
3    12648430
```

Use the read function to return the value of a for the first element in the structure.

```
read(x, 1, 'a')
```

```
ans =
```

```
3
```

```
read(x, 2, 'a')
```

```
ans =
```

```
12648430
```

Type Objects – Their Methods and Properties

When you create an object that accesses a typedef declared in your source code, the object constructor `createobj` returns a type object. `createobj` uses the information in your source code to set the properties of the object appropriately to match the code. Within the properties and their values, you find all the information about the declaration, so that MATLAB understands how to handle the typedef in your MATLAB workspace and how to read and write the typedef on your target.

Like `memoryobj` and `registerobj` class objects, type class objects do not inherit properties from a parent class. Unlike the other objects in Link for Code Composer Studio, type objects only exist as members of `ccsdsp` objects. You cannot directly create a type object using `createobj`. When you delete the `ccsdsp` object, you delete the type object as well. This relationship is called composition in the standard object modeling language (UML). Instead, when you call `createobj`, the resulting object includes by composition a type object, with the object properties set to their default values.

Properties of Type Objects

Object properties can include both properties that the object inherits from its superclass, if any, and some properties that are unique to the class itself. For this reason, many objects in Link for Code Composer Studio share common properties; as you use the objects you will become familiar with the common and special properties for each.

Property Name	Property Type	Default Value	Description
<code>typelist</code>	cell array	None	List of the typedef equivalents in the object. This list relates the typedef name to its equivalent data type, either a native data type or a custom type definition. Equivalent types follow the order of the names in <code>typename</code>

Property Name	Property Type	Default Value	Description
typename	string	None	Names of the typedef entries in the object
timeout	integer	30 s	Local time-out value applied to type class operations

Methods of Type Objects

Like properties, methods for objects may come from the superclass or derive only from the class itself. For example, the `cast` and `convert` methods do not appear in all objects; listing them here indicates that the object does not inherit these methods but provides them itself.

Name	Overloaded?	Description
<code>add</code>	No	Adds a new type definition to the type object in MATLAB
<code>clear</code>	Yes	Removes an existing type declaration from your type object
<code>display</code>	Yes	Displays the properties of a type object
<code>gettypeinfo</code>	No	Returns information about a type declaration in your type object

Constructing Objects That Access Bitfields

Since bitfield objects do not stand by themselves, but only as parts of struct objects, you work with bitfields by starting with a struct object. You create an object that accesses the structure that uses the bitfield. With the struct object now in your workspace, use `getmember` to create objects that access the elements of the structure. For example, in the next code offerings, we create a structure that contains a bitfield, then access the bitfield elements to be able to read and write to them.

Here is the target structure definition

```
struct {
  int b_2 : 1;
  unsigned int b_22 : 22;
  unsigned int b_10 : 3;
} bit_field = { 0, 689, 4};
```

Create the struct object.

```
bit_field=createobj(cc,'bit_field')
```

Use `bit_field` and `getmember` to construct objects for the components in the bit field.

```
b_2=getmember(bit_field,'b_2')
```

BITFIELD Object stored in memory:

```
Symbol name      : b_2
Address          : [ 2147501596 0]
Wordsize        : 32 bits
Address units per value : 4 au
Representation   : signed
Size            : [ 1 ]
Total address units : 4 au
Array ordering   : row-major
Endianness      : little
Length (bits)   : 1
Offset (bits)   : 0
```

```
b_22=bfield.member.b_22 % Alternate syntax for accessing members
```

```
BITFIELD Object stored in memory:
```

```
Symbol name      : b_22  
Address          : [ 2147501596 0]  
Wordsize        : 32 bits  
Address units per value : 4 au  
Representation   : unsigned  
Size            : [ 1 ]  
Total address units : 4 au  
Array ordering   : row-major  
Endianness       : little  
Length (bits)    : 22  
Offset (bits)    : 1
```

Creating function Objects

Like the other objects in Link for Code Composer Studio, you use `createobj` to construct objects that access the functions in your program and project in CCS. However, unlike many of the other objects, constructing function objects has some peculiarities with which you must be familiar.

The Link for Code Composer Studio function objects support two kinds of program functions:

- Functions that you write in ANSI C
- Functions that you write in Assembly but that have C function prototypes, such as library functions

A number of classes of functions that are allowed in your program are not supported by function objects:

- Assembly language functions that do not have C prototypes
- Functions where the number of input arguments changes
- Functions written in non-ANSI C language

For the unsupported function types, you cannot create function objects that access them and you cannot work with them with Link for Code Composer Studio.

In general, Link for Code Composer Studio provides three related ways to create function objects, all of which use `createobj` as a starting point.

- 1 Use `createobj` with the function name in the syntax. For example, to create an object that accesses `func_name`, use

```
ff = createobj(cc,func_name)
```

which creates the function object `ff` that accesses `func_name`. This syntax tells MATLAB to try to locate the function declaration string in your project. When it finds the required declaration, `createobj` generates the objects and information, such as function object property values, that enable MATLAB to run `func_name`. Note that searching your project for

the function declaration may take some time, depending on projects you have open in CCS and the communications speed between your PC host and the target.

If MATLAB cannot find the function declaration for `func_name`, one of the next two approaches works to create the necessary function object.

Note An important note about creating function objects and `createobj`. Even when MATLAB cannot find your specified function by name, it creates the function object `ff`, although populated with default values for the properties. Method 3 below takes advantage of this fact of object creation.

- 2 Pass the function declaration string in the calling syntax for `createobj`. When you use this method, MATLAB skips the search for the function prototype and creates the function object from your input string. Here is one way to do it, using the `createobj` optional keywords `function` and `funcdecl`.

```
ff = createobj(cc,func_name,'function','funcdecl',declaration_string)
```

- 3 When the function object exists already, but it does not have full property values associated with it, pass the function declaration string to the function object with `declare`, and the keyword `decl`.

```
declare(ff,'decl','declaration_string')
```

When to Use `declare` to Provide the Function Declaration

Some types of functions in your project require that you explicitly provide the function declaration to MATLAB. In the following types of functions, MATLAB cannot determine the function declaration from CCS:

- Functions that you write in assembly, but you provide C declaration strings for them
- Functions in a CCS project that you compile without enabling symbolic debugging
- Projects where you load the COFF file but not the project

- Instances where something in your function declaration, such as a non-ANSI C keyword, causes `createobj` to fail to read the declaration fully

Using `declare` to send the declaration string corrects each of the above situations so you can use MATLAB to run the function on your target.

To help you see what this means, here is one example that uses `declare`. Note that you cannot run this example code without modifications.

In your project:

```
#define NumDefinedQualifier extern
NumDefinedQualifier void foo(void)
```

In MATLAB:

```
ff = createobj(ff, 'foo')
```

generates a warning that MATLAB could not read the function declaration for `foo`. Try either of the following to overcome the error:

```
declare(ff, 'decl', 'void foo(void)')
```

or:

```
declare(ff, 'decl', 'extern void foo(void)')
```

Differences Between Objects for Library Functions and C Functions

To run functions on your target, MATLAB needs a range of information about the function you are running. `function` objects in Link for Code Composer Studio provide the information MATLAB needs. When you create an object that accesses a function, the properties of the new `function` object contain all the information MATLAB requires to be able to run the function. Unfortunately, this is not true for all functions — `function` objects that access library functions do not contain the same function prototype that C `function` objects contain when you create them. When you try to create a `function` object to access a library function, MATLAB returns a warning message that it created the object you requested but could not set all the properties of the object.

Library Functions

Library functions are functions that are not compiled when you build your project. They represent precompiled functions that you call from your C source code and the compiler does not know about the functions beyond their locations. Examples of library functions include those functions in the C standard library, or functions in other standard libraries. Another example of library functions are functions written in assembly but accompanied by C prototypes (the TI run-time libraries fall into this category). In CCS IDE, you find library functions listed in the `Libraries` directory in your project directory tree.

Functions written in non-ANSI C or functions written in another language like Assembly that do not have C prototypes; or functions that have varying numbers of input arguments, do not work with the function objects in Link for Code Composer Studio.

Because library functions are not part of the compile and build process for projects in CCS, the information about library function declarations, or prototypes, is not available to MATLAB from the symbol table in CCS. To overcome this problem, Link for Code Composer Studio includes a method named `declare` that lets you provide the declaration for a library function from the MATLAB command line. For more about using `declare` to enter function prototype strings in to MATLAB, refer to the reference page for `declare`, and to the tutorial about using functions in “Tutorial — Using function Objects and Function Calls” on page 2-77.

Examples of Creating Function Objects

The following sections cover situations you may encounter when you create function objects:

- Run a C function.
- Run a library function.
- Run a function that includes a custom data type.
- Run code generated by Real-Time Workshop.
- Run a function that has input vectors.

Unless you have project code that supports the functions used here you cannot run these examples. They are for instruction only.

These examples refer to four functions — `sin_taylor`, `dotprod`, `adotprod`, and `cdotprod`. Here is the code for each one.

- Function `sin_taylor` is a C function.

```
/*-----*
 * Taylor Series expansion of sin function - Fixed Point
 * Limitations: input range: -pi <x <pi;
 *
 * Input Datatype is:
 *   Q2.13 (or MATLAB sfix16_En13), scale factor = 2^13
 * Output Datatype is:
 *   Q1.14 (or MATLAB sfix16_En14), scale factor = 2^14
 *
 * Taylor Expansion of sin function (first 4 terms)
 *   sin(x) =(approx) x[1 - (x^2/6)*[1 + (x^2/20)*[ 1 - (x^2/42)]]]
 *-----*/
#define SFIX32_EN26_VAL_1    67108864 // Integer equivalent of
1.0 in Q5.26
#define SFIX32_EN28_VAL_1    268435456 // Integer equivalent of
1.0 in Q3.28
#define SFIX32_EN30_VAL_1    1073741824 // Integer equivalent of
1.0 in Q1.30

/* Global buffers */
short ibuf[63];
short obuf[63];

short sin_taylor(short x)
{

// Define 16/32 bit local variables depending on processor
#if INT_MAX == 0x7FFFFFFF
int acc,a1,a2,a3,xpow;
#elif LONG_MAX == 0x7FFFFFFF
```

```

long acc, a1, a2, a3, xpow;
#endif

xpow = x*x;    // x^2  sfix32_En26

a1 = xpow/42;  // x^2/42  sfix32_En26
a2 = xpow/20;  // x^2/20  sfix32_En26
a3 = xpow/6;   // x^2/6   sfix32_En26

acc = SFIX32_EN26_VAL_1 - a1;
acc >>= 11;
acc *= (a2>>11);

acc = SFIX32_EN30_VAL_1 - acc;
acc >>= 14;
acc *= (a3>>14);

acc = SFIX32_EN28_VAL_1 - acc;
acc >>= 11;
acc *= x;

return (acc>>16);
}

```

- Function `dotprod` is a library function and has only a prototype, no source code.

```
int dotprod (short *x, short *y, int nx);
```

- Function `adotprod`

```

/* Global buffers */
short a[] = {1, 2, 3, 4, 5};
short b[] = {1, 2, 3, 4, 5};

int adotprod(short x[4], short y[4])
{
    int sum;
    int i;

```

```
    sum = 0;
    for(i=0;i<4;i++) {
        sum += ( x[i]*y[i] );
    }
    return sum;
}
```

- Function cdotprod

```
/* Global buffers */
short a[] = {1, 2, 3,4,5};
short b[] = {1, 2, 3, 4,5};

/* Typedef info */
typedef int INT;
typedef short SHORT;

/*
   Function cdotprod returns the dot product of
   two integer arrays (datatype=short).
   Inputs:
       x, y - pointer to an array of shorts
       n    - size of array pointed to by x and y
*/
INT cdotprod(SHORT x[], SHORT y[], INT n)
{
    int sum;
    int i;
    sum = 0;
    for(i=0;i<n;i++) {
        sum += ( x[i]*y[i] );
    }
    return sum;
}
```

Run a Standard C Function

In this example, we run function `sin_taylor` that computes the value for the sine of an input value. This function accepts one input, `x` (using data type short), and returns a short data type result.

To get the correct values, the input data must be converted to Q16.13 format before passing to the function. After execution, the output value must be converted from Q16.14 to decimal representation.

Create a `ccsdsp` link:

```
cc = ccstdsp;  
reset(cc);  
pause(1); % Wait for hardware reset to complete before proceeding
```

Run to start of main — ensures that global variables are initialized:

```
run(cc, 'main', 1000);
```

Create a function object for `sin_taylor`:

```
ff = createobj(cc, 'sin_taylor')  
inputdata = 0.5; % input value to be used
```

Set value of input `x`:

```
x_obj = getinput(ff, 'x');  
write(x_obj, inputdata * 2^13);
```

Run the function:

```
outputdata = run(ff);
```

Run a Library Function

For a library function, you pass the declaration string explicitly through `declare`.

This example runs the function `dotprod` that computes the dot product of two arrays. This function requires three inputs:

- `x` — a pointer to a vector of short data type values
- `y` — a pointer to a vector of short data type values
- `n` — the size of `x` and `y` vectors

We use the global variable `a` for input `x`, `b` for input `y`, and `4` for input `nx` (since `a` and `b` are four element vectors). The function returns a short.

Create a `ccsdsp` link and object:

```
cc = ccsdsp;  
reset(cc);  
pause(1); % Wait for hardware reset to complete before proceeding
```

Run to start of `main` to ensure that you initialize the global variables:

```
run(cc, 'main', 1000);  
a_addr = address(cc, 'a'); % Global buffer for 'x'  
b_addr = address(cc, 'b'); % Global buffer for 'y'
```

Create the function object for the library function `dotprod`:

```
ff = createobj(cc, 'dotprod')
```

The previous step yields an incomplete function object `ff` because library functions always require that you provide the function declaration explicitly, as follows:

```
declare(ff, 'decl', 'int dotprod (short *x, short *y, int nx)')
```

Set the value for the input parameter `x`:

```
x_obj = getinput(ff, 'x');  
write(x_obj, a_addr(1));  
xRef_obj = deref(x_obj);  
reshape(xRef_obj, 4);  
x_inputval = read(xRef_obj) % Verify 'y' referent value
```

Set the value for `y`, the second input parameter:

```
y_obj = getinput(ff, 'y');
```

```

write(y_obj,b_addr(1));
yRef_obj = deref(y_obj);
reshape(yRef_obj,4);
y_inputval = read(yRef_obj) % Verify 'y' referent value

```

Pass the value for nx to the function:

```

nx_obj = getinput(ff,'nx');
write(nx_obj,4);
nx_inputval = read(nx_obj) % Verify 'nx' value

```

Now run the function:

```
run(ff);
```

Run a Function That Has a Custom Type Definition in the Prototype

Having custom data types in your function declaration can cause problems when you run the functions from MATLAB.

Case 1 – Running a Function That Has a Typedef in the Function Prototype. This example runs the function `cdotprod` that computes the dot product of two matrices. This function requires three inputs:

- `x` — a pointer to a vector of short data type values
- `y` — a pointer to a vector of short data type values
- `n` — the size of `x` and `y` vectors

Both `n` and the return argument are defined as data type `INT`, a custom data type defined in the source code.

We use the global variable `a` for input `x`, `b` for input `y`, and 4 for input `n` (since `a` and `b` are four-element vectors). The function returns a short.

Create a `ccsdsp` link:

```

cc = ccsdsp;
reset(cc);
pause(1); % Wait for hardware reset to complete before proceeding

```

Run to start of main to ensure that CCS initializes all of the global variables before you create your function object for `cdotprod`:

```
run(cc, 'main', 1000);  
a_addr = address(cc, 'a'); % Global buffer for x  
b_addr = address(cc, 'b'); % Global buffer for y
```

Create a function object for the library function `cdotprod`:

```
ff = createobj(cc, 'cdotprod')
```

The previous call to `createobj` yields an incomplete function object because the function declaration includes an unresolved typedef — the type `INT`. To resolve this error, add the custom data type `INT` to the type object and use `declare` to pass the function declaration to MATLAB:

```
add(cc.type, 'INT', 'int'); % Earlier warning that data type  
                           % INT cannot be resolved  
declare(ff, 'decl', 'INT cdotprod (short x[], short y[], INT n)')
```

Set values for the inputs `x`, `y`, and `n`, and run the function, passing the input values in the run syntax. Input `x` is a pointer so pass an address. Input `y` is a pointer as well, so pass another address. Input `n` is an integer that specifies the size of `x` and `y`:

```
run(ff, 'x', a_addr(1), 'y', b_addr(1), 'n', 4);
```

Case 2 – A Second Approach to Solving the Typedef Problem. We now run the function `cdotprod`, which computes the dot product of two matrices. This function accepts three inputs:

- `x` — a pointer to a vector of short data type values
- `y` — a pointer to a vector of short data type values
- `n` — the size of `x` and `y` vectors

We are going to use the global variable `a` for input `x`, `b` for input `y`, and 4 for input `n` (since `a` and `b` are four element vectors). The function returns a short.

Create `ccsdsp` link:


```
cc = ccstdsp;
reset(cc);
pause(1); % Wait for hardware reset to complete before proceeding
```

Run to start of main to ensure that CCS initializes all of the global variables before you create your function object for `cdotprod`:

```
run(cc,'main',1000);
a_addr = address(cc,'a'); % Global buffer for 'x'
b_addr = address(cc,'b'); % Global buffer for 'y'
```

Create function object for library function `cdotprod`:

```
ff = createobj(cc,'cdotprod')
```

Again `createobj` generates an incomplete function object because of the unresolved data type `INT` in the function declaration. In this case, fix the problem by adding the custom data type `INT` to the type object and create the object `ff` again, instead of using `declare` to pass the function declaration to MATLAB:

```
add(cc.type,'INT','int'); % Warning only mentioned that type INT
                        % cannot be resolved
ff = createobj(cc,'cdotprod')
```

Set values for the inputs `x`, `y`, and `n`, and run the function, passing the input values in the run syntax. Input `x` is a pointer so pass an address. Input `y` is a pointer as well, so pass another address. Input `n` is an integer that specifies the size of `x` and `y`:

```
run(ff,'x',a_addr(1),'y',b_addr(1),'n',4);
```

Case 3 – A Third Approach to Solving the Typedef Problem. Once more we are going to run the function `cdotprod` which computes the dot product of two matrices. This function accepts three inputs:

- `x` — a pointer to a vector of short data values
- `y` — a pointer to a vector of short data values
- `n` — the size of `x` and `y` vectors

We are going to use the global variable `a` for input `x`, `b` for input `y`, and `4` for input `n` (since `a` and `b` are four element vectors). `cdotprod` returns a short.

Create `ccsdsp` link:

```
cc = ccsdsp;  
reset(cc);  
pause(1); % wait for hardware reset to complete before proceeding
```

Run to start of main, ensuring that CCS initializes all of the global variables before you create the function object that accesses `cdotprod`:

```
run(cc, 'main', 1000);  
a_addr = address(cc, 'a'); % Global buffer for x  
b_addr = address(cc, 'b'); % Global buffer for y
```

Create a function object for the library function `cdotprod`:

```
ff = createobj(cc, 'cdotprod')
```

This attempt to create a new function object `ff` results in an incomplete function object because MATLAB could not resolve the data type `INT` in the function declaration. In this approach to overcoming the unresolved type error, use `declare` to pass to MATLAB a version of the `cdotprod` function declaration that does not include the offending type `INT` — you do not need to add the typedef to the type object:

```
declare(ff, 'decl', 'int cdotprod (short x[], short y[], short n)')
```

Notice that the data types for the return argument and for `n` now specify `int`. Set values for the inputs `x`, `y`, and `n`, and run the function, passing the input values in the `run` syntax. Input `x` is a pointer so pass an address. Input `y` is a pointer as well, so pass another address. Input `n` is an integer that specifies the size of `x` and `y`:

```
run(ff, 'x', a_addr(1), 'y', b_addr(1), 'n', 4);
```

Run a Function Generated by Real-Time Workshop

We run the function 'mwdsp_fir_df_dd', which applies a filter to a noisy input signal. This function accepts nine input parameters and returns the filtered signal in the input argument *y*.

Create a `ccsdsp` link:

```
cc = ccsdsp;
reset(cc);
pause(1); % Wait for hardware reset to finish before proceeding
```

Now run the Real-Time Workshop generated code from the beginning to `MdlOutputs`. You run from program start until `MdlOutputs` to ensure that all of the code configuration processes get done — the CCS initializes all of the variables in program. In the case of generated code, running to `main` is not sufficient to ensure that all the variable get initialized:

```
run(cc, 'runtofunc', MdlOutputs);
```

After running to `MdlOutputs`, you create the function object — pass the function declaration to avoid MATLAB returning an error when you create the function object. Due to the complexity of this function declaration, we have assigned the string to a variable `decl`. We use the variable in the `createobj` syntax:

```
decl = ['MWDSP_IDECL void MWDSP_FIR_DF_DD(const real_T *u,...
real_T *y, real_T * const mem_base,int_T *mem_offset,...
const int_T numDelays, const int_T sampsPerChan,...
const int_T numChans, const real_T * const b,...
const boolean_T one_fpf)'];
ff = createobj(cc, 'MWDSP_FIR_DF_DD', 'function', 'funcdecl', decl);
```

Examine the function declaration above. This declaration causes MATLAB to fail to create the fully populated function object `ff` because the `MWDSP_IDECL` macro at the beginning of the string. MATLAB cannot recognize this string. Since the information in `MWDSP_IDECL` is not relevant to creating the function object, you can remove this from the declaration string:

```
decl = ['void MWDSP_FIR_DF_DD(const real_T *u,...
real_T *y, real_T * const mem_base,int_T *mem_offset,...
const int_T numDelays, const int_T sampsPerChan,...
```

```
const int_T numChans, const real_T * const b,...  
const boolean_T one_fpf)'];  
ff = createobj(cc,'MWDSP_FIR_DF_DD','function','funcdecl',decl);
```

Now function object `ff` has all the information MATLAB needs.

Note You may not always be able to remove offending entries in a declaration string, as we did with the macro `MWDSP_IDECL`. Often you can try your declaration and see if it works. If not, use `add` to include typedefs in the type object when MATLAB complains about a data type, or try removing the problem portion of the declaration string if the function does not require the troublesome text.

With the function object in your MATLAB workspace, create objects for the inputs to `MWDSP_FIR_DF_DD`:

Create an object for `rtB`:

```
rtBobj = createobj(cc,'rtB');
```

Get the relevant `rtB` member objects:

```
SumObj = getmember(rtBobj,'Sum');  
% Store Output of MWDSP_FIR_DF_DD in FilObj  
FilObj = getmember(rtBobj,'Digital_Lowpass_Fil');
```

Next, create an object for `rtDWork`

```
rtDWorkObj = createobj(cc,'rtDWork');
```

and again get the relevant member objects:

```
Fil_FILT_STATES = getmember(rtDWorkObj,...  
'Digital_Lowpass_Fil_FILT_STATES');  
DF_INDX = getmember(rtDWorkObj,...  
'Digital_Lowpass_Fil_FILT_STATES');
```

Create one last object for `filterCoeffs`:

```
filterCoeffsObj = createobj(cc, 'filterCoeffs');
```

To run the function, you need to provide the input values:

```
u = SumObj.address(1); % Input 1
y = FilObj.address(1); % Input 2
mem_base = Fil_FILT_STATES.address(1); % Input 3
mem_offset = DF_INDX.address(1); % Input 4
numDelays = 65; % Input 5
sampsPerChan = 256; % Input 6
numChans = 1; % Input 7
b = filterCoeffsObj.address(1); % Input 8
one_fpf = 1; % Input 9
```

Run the function, providing the input argument values in input value/input name pairs, such as 3, membase and 6, sampPerChan:

```
run(ff, 1, u, 2, y, 3, mem_base, 4, mem_offset, 5, numDelays, 6, ...
    sampsPerChan, 7, numChans, 8, b, 9, one_fpf)
```

Run a Function That Has Vector Inputs

This example shows how to run a function that accepts vector inputs.

We are going to run the function `adotprod` that computes the dot product of two matrices. `adotprod` accepts two inputs:

- `x` — a four-element vector of short data type values
- `y` — a four-element vector of short data type values

The compiler converts the vector inputs into pointers to the vectors. We use the global variable `a` for input `x` and `b` for input `y`. The function returns a short.

Create a `ccsdsp` link:

```
cc = ccsdsp;
reset(cc);
pause(1); % Wait for hardware reset to complete before proceeding.
```

Run to the start of main to ensure that the global variables are initialized:

```
run(cc,'main',1000);
a_addr = address(cc,'a'); % Global buffer for 'x'
b_addr = address(cc,'b'); % Global buffer for 'y'
```

Create a function object ff to access adotprod:

```
ff = createobj(cc,'adotprod')
```

The function prototype for adotprod is

```
int adotprod(short x[4], short y[4])
```

adotprod requires as input two vector arrays x and y. The compiler requires that you pass the addresses of x[4] and y[4], not the actual vectors x and y. So instead of writing a data vector to input object x_obj and y_obj, you provide the addresses of existing four-element vectors:

```
display('INPUT VALUE ''x'':')
x_obj = getinput(ff,'x') % Note that this is a pointer to a vector
                        % of shorts
display('INPUT VALUE ''y'':')
y_obj = getinput(ff,'y') % note that this is a pointer to a vector
                        %of shorts
```

Set values of inputs x and y and run the function. Pass addresses to x and y since both are pointers to other data:

```
write(x_obj,a_addr(1))
write(y_obj,b_addr(1))
x_inputval = read(reshape(deref(x_obj),4));
y_inputval = read(reshape(deref(y_obj),4));
```

In contrast to using pointers, using the following commands to write data to x and y does not give you the expected result — the compiler cannot determine where to put array [1:4]:

```
write(x_obj,[1:4]);
write(y_obj,[1:4]);
```

Now run your function:

```
run(ff);
```

The preceding examples present a few of the wide variety of functions and conditions you may encounter when you construct function objects.

Creating Type Objects

Type objects are unique among the objects in Link for Code Composer Studio because you cannot use `createobj` to create a type object directly. Each time you create a `ccsdsp` object `objectname`, the new object contains an empty type object, called `objectname.type`. You may note that this looks very much like a property of the object `objectname`. It is, however, an object: it has properties and methods that let you manipulate it from MATLAB.

When you create a type object, the object constructor add the following DSP/BIOS data types to the `namelist` property:

BIOS Data Type	Equivalent C Data Type
Void	void
Float	float
Double	double
Long	long
Int	int
Short	short
Char	char

Link for Code Composer Studio ignores certain CCS keywords when you create type objects: `interrupt`, `near`, `far`, `cregister`, and `volatile`. These keywords have no meaning in the MATLAB workspace.

Working with Type Definitions in Projects

Type definitions (`typedefs`) in your C source code present a special problem in Link for Code Composer Studio. While you can use any valid `typedef` in the C programs you use in your project, MATLAB cannot read your custom data types from the project in CCS without your help. You must supply each `typedef` to MATLAB explicitly. There is no way for MATLAB to interpret existing `typedefs` in your CCS project.

In particular, until you tell MATLAB about the `typedefs` you use in your project, you cannot use your `typedefs` when you create objects that access

functions whose prototypes include the typedefs as either input or output arguments. Unless MATLAB recognizes your custom data types, you get an error when you try to create the object or use `declare` to specify the function prototype in MATLAB.

To tell MATLAB about your custom data types, you use `add` to add the type definitions to a `ccsdsp` object that accesses your project in CCS.

To Add a Type Definition to an Existing `ccsdsp` Object

Adding a new type definition to a `ccsdsp` object entails using `add` to include the new data type in the type object associated with your `ccsdsp` object. Follow this example to see how you add a typedef to your type object. At the end of the example, you use your new typedef in a function declaration.

1 Create a `ccsdsp` object:

```
mylink = ccsdsp;
```

2 Look at the properties of `mylink`, and the associated type object `mylink.type`:

```
get(mylink)

rtdx: [1x1 rtdx]
  apiversion: [1 2]
  ccsappexe: 'D:\Applications\ti\cc\bin\'
  boardnum: 0
  procnum: 0
  type: [1x1 type]
  timeout: 10
  page: 0

get(mylink.type)
  typename: {'Void' 'Float' 'Double' 'Long' 'Int' 'Short' 'Char'}
  typelist: {1x7 cell}
  timeout: 10
```

`typename` contains the default set of defined types. `typelist` contains seven cell arrays of the form `[1x1 struct]`. You can verify this by issuing the command

```
mylist.type.typelist
```

- 3** Now add a new type definition to the type object. For now add a typedef `mytype` which uses the `uint32` data type:

```
add(mylink.type,'mytype','uint32')
```

```
ans =
```

```
    type: 'uint32'  
    size: 1  
    uclass: 'numeric'
```

```
mylink.type.typename
```

```
ans =
```

```
Columns 1 through 7
```

```
    'Void'    'Float'    'Double'    'Long'    'Int'    'Short'    'Char'
```

```
Column 8
```

```
    'mytype'
```

`typelist` now contains eight 1-by-7 cell arrays, one additional one for the new type `mytype`.

With MATLAB informed about your custom data type `mytype`, you could use the `typedef` in a function declaration, such as the following command where `ff` is an object that accesses the function `myfunction`:

```
declare(ff,'decl','void myfunction(short x* int32 y* float z mytype m)')
```

Tutorial — Using function Objects and Function Calls

The Link for Code Composer Studio Development Tools provides a connection between MATLAB and a digital signal processor in CCS. Using objects with the links provides a mechanism for you to control and manipulate a signal processing application using the computational power of MATLAB. This can help you debug and develop your application. Another use for links and objects is creating MATLAB scripts that you use to verify and test algorithms by running the algorithms on your potential target during development.

The Link for Code Composer Studio provides hardware-in-the-loop (HIL) functionality that enables you to verify your signal processing (DSP) application implementation, within the context of a system design, by simulating in MATLAB components that you did not implement on the digital signal processor. You may want to verify your implementation of an FIR filter, for example, on your processor while simulating your input data and processing your output data in MATLAB. The performance of your closed-loop system design may be assessed with the real-world constraints of your hardware (the processor) and software (DSP implementation).

In this tutorial, you perform these operations from MATLAB.

- Call digital signal processing functions
- Get the function signature information, such as
 - Input argument names and types
 - Function return type
 - Starting address
- Specify the values for each input argument
- Run the function
- Read the returned value(s)

This tutorial assumes you are relatively familiar with the Link for Code Composer Studio. If not, we suggest that you run `csstutorial` first to give you a better understanding of what the Link for Code Composer Studio does. Run `ccstutorial` by typing `ccstutorial` at the command prompt in MATLAB.

Before using the function object available with the Link for Code Composer Studio, you must select a digital signal processor to be your target because your objects require a link to refer to. Selecting a processor is only necessary for multiprocessor boards or multiple board configurations of CCS. When you have only one board with a single processor, the link defaults to the existing processor. For the links, the simulator counts as a board; if you have both a board and a simulator that CCS recognizes, you must specify the target explicitly.

Introducing the Tutorial

To get you started using function objects in your CCS IDE software, the Link for Code Composer Studio includes an example script `hiltutorial.m`. As you follow along with this tutorial, you perform tasks that step you through creating and using function objects in MATLAB and in your projects:

- 1 Selecting your target.
- 2 Creating and querying links to CCS IDE.
- 3 Constructing and using various objects, such as numeric and string objects.
- 4 Creating and using function objects to access functions in your project.
- 5 Closing the links you opened to CCS IDE.

For this tutorial, you load and run a sample DSP application on a target processor you select. To help you understand how objects work, the tutorial also demonstrates writing to memory and reading from memory and registers.

Using the data manipulation functions gets a bit complicated. MATLAB supports only double-precision values for calculations, but you can convert and cast a range of data types to and from other data types. Seeing how the functions work with many of the different objects can help you when you are doing your work.

The tutorial covers the objects and methods listed below. The functions listed first apply to CCS IDE independent of the links — you do not need a link to use these functions. The functions and methods listed next require a CCS IDE link in place before you can use the function syntax. Finally, the last set

of entries use the function object, using the methods that apply to working with function objects in MATLAB and your project:

Global Functions for CCS IDE – No Link Required

- `ccsboardinfo` — return information about the boards that CCS IDE recognizes as installed on your PC.
- `boardprocel` — select the board to target. Although you can use this generally, the Link for Code Composer Studio provides it as an example of a user interface you can build and as a tool in the tutorial. We do not recommend that you use this to select your target. Use `ccsboardinfo` and `ccsdsp` to specify the target for your processing application
- `ccsdsp` — construct a link to CCS IDE. When you construct the link you specify the target board and processor.
- `clear` — remove a specific link to CCS IDE or remove all existing links.

Link for Code Composer Studio Functions for Working with Embedded Objects – Uses Links

- `cast` — create a new object with a different data type (the `represent` property) from an object in the Link for Code Composer Studio. Demonstrated with a numeric object.
- `convert` — change the `represent` property for an object from one data type to another. Demonstrated with a numeric object.
- `createobj` — return an object in MATLAB that accesses embedded data. Demonstrated with structure, string, and numeric objects.
- `getmember` — return an object that accesses a single field from a structure. Demonstrated with a structure object.
- `goto` — position your cursor in CCS to the specified function in the project code.
- `list` — return various information listings from CCS.
- `read` — read the information at the location accessed by an object into MATLAB as numeric values. Demonstrated with numeric, string, structure, and enum objects.

- `readnumeric` — return the numeric equivalent of data at the location accessed by an object. Demonstrated with an enum object.
- `write` — write to the location referenced by an object. Demonstrated with numeric, string, structure, and enum objects.

Link for Code Composer Studio Functions for Working with Embedded Functions – Uses Function Objects

- `createobj` — construct a function object that accesses a function in your project in CCS.
- `run` — run a function on your target.
- `copy` — copy an existing function.
- `declare` — create a new function for your project from MATLAB. With this method you create the function prototype, configuring the input and output arguments, among other things.
- `getinput` — create the input arguments for your new function.
- `getoutput` — create the output arguments for your new function.

Running the Interactive Tutorial

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

To run the tutorial in MATLAB, click `run ccstutorial`. Running the interactive tutorial in MATLAB puts you 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. To view the tutorial M-file, click `tutorial`.

To Run the Hardware-In-The-Loop Tutorial

Running the tutorial consists of completing the following tasks that cover setting up and running a project in CCS and interacting with the project from MATLAB, as well as running functions from MATLAB on your target hardware. In order, the tasks are

- 1 Select your target and establish the link between MATLAB and the target. These operations, or some variant of them, are the first things you do to work between MATLAB and CCS.
- 2 Load the tutorial project.
- 3 Initialize the embedded C variables, then construct and work with an embedded object.
- 4 Use `read`, `write`, `cast`, and `convert` to manipulate a few variables. Within this section you learn that `read`, `write`, `cast`, and `convert` behave differently depending on the object you are using.
- 5 Construct a function object and run the function from MATLAB.
- 6 Exercise various methods that work with function objects, such as `copy`.
- 7 Construct other embedded objects and work with them, such as pointer objects, enum objects, and type objects.
- 8 Close the tutorial and clean up the lingering objects, handles, and assorted variables left over. Close CCS as well.

Note To run this tutorial, you must have either a C54xx or C6xxx processor and board, or be using one of the C54 or C6x simulators in CCS. Link for Code Composer Studio does not support function calls for this tutorial on other TI processors.

This is a rather long and complicated tutorial, because the embedded object concepts are somewhat less straightforward and more numerous than the `ccsdsp` or `rt dx` objects and concepts.

Stopping and Saving the Tutorial Program

If you stop in the middle of the tutorial, save your workspace so you can reload the tutorial program (`hiltut.board.out`, where `board` is the numeric designation of your target, such as `6x11` or `54xx`) directly into CCS and continue later. To save your workspace before you close MATLAB, select **File > Save Workspace As** from the MATLAB menu bar. To start the

tutorial again, reload the MATLAB workspace you stored and build and load the .out file to start the tutorial.

Select Your Target and Load the Tutorial Project

You start by selecting your target DSP using a tool called `boardprocse1`. Then you create a link between MATLAB and CCS. The link is represented by a MATLAB object which you save in variable `cc`.

Note You use the digital signal processor that you select in the GUI for the rest of this tutorial. For single processor installations of CCS, click **OK** to continue. When you click **OK**, `boardprocse1` assigns the board and processor identification information to the output arguments `boardnum` and `procnum`.

1 Start the board selection tool by entering

```
[board,processor] = boardprocse1
```

Follow the instructions on the dialog box to select your target processor.

2 Use the board and processor variables to construct a `ccsdsp` object named `cc`:

```
cc=ccsdsp(boardnum,board,procnum,processor)
```

Now that you have established the connection between MATLAB and your target (the link), the target processor needs something to do. Your next step is to create or load executable code for the target DSP with CCS.

For this tutorial, we created a CCS project file and board-specific executables and included them with MATLAB. In this tutorial section, you load the included executable directly; if the load fails (perhaps because you selected a different board or processor target), you build the included project to compile and generate the executable for your target.

The following functions locate the function call tutorial project and load it into CCS. Loading the project uses `open`, and directs CCS to load the project files or a program file.

- 1** Start by gathering some information about the `cc` object you constructed. Enter the following function calls to learn more about `cc` and to assign MATLAB variables to the values of some properties of `cc`:

```
linkinfo = info(cc);
familycpu = linkinfo.subfamily;
revisioncpu = linkinfo.revfamily;
board = GetDemoProp(familycpu,revisioncpu,hiltutorial);
```

- 2** Now locate the project file for the tutorial and assign the path to a variable:

```
projfile = fullfile(matlabroot,toolbox,ccslink,ccsdemos,...
hiltutorial,board.hiltut.projname);
```

- 3** For convenience, assign the path to the project file to a variable (you use it in a later step):

```
projpath = fileparts(projfile);
```

- 4** Now open the project file, using the link `cc`:

```
open(cc,projfile); % Open project file
```

- 5** To make your CCS working directory the same as your project directory, use `cd`:

```
cd(cc,projpath); % Change working directory of CCS
```

Changing the working directory ensures that CCS finds all the project files and stores changes in the same area as well.

- 6** When you created `cc`, the process opened CCS with the visibility set to 0 — not visible. You are going to need to see the source files and variables for the tutorial, so set the visibility for CCS to 1:

```
visible(cc,1)
```

- 7** Finally, open the tutorial source file, activate it, and bring it to the foreground in CCS:

```
open(cc,'hiltut.c' ,'text');
activate(cc,'hiltut.c','text');
```

Notice that the tutorial project is loaded in CCS. Examine the files in CCS that compose this project. The main source file the project uses is `hiltut.c` (the same for all targets), accompanied by a linker command file (`*.cmd`) and a vector table source file (`*.asm`) that will be different depending on the DSP family you are using. Also review the variables and functions in the file `hiltut.c` — you manipulate them from MATLAB later in the tutorial. Throughout the remaining tutorial, we call these variables and functions *embedded* objects or variables. Before you build this project, try to load the included executable program file `hiltut.loadfile`.

- 8** Use `load` to load the target executable file to the target by entering the following command:

```
load(cc,board.hiltut.loadfile,30)
```

Possibly the load failed. This might happen when the load file was created for a different target than the one you are using. When this happens, try rebuilding the executable and then loading it before proceeding.

- 9** To create the executable file for the project from the source files, use `build`. Before proceeding to build the file, you should set up the build options for the build process, just as you would in CCS. From MATLAB, use `setbuilddopt` with `cc` to provide the build options and needed configuration:

```
cc.setbuilddopt(compiler,board.hiltut.Cbuilddopt);
```

- 10** To avoid overwriting the existing executable file, redirect the output program file to a temporary directory on your system:

```
eval([cc.setbuilddopt(Linker , -c -o 'tempdir...  
board.hiltut.loadfile' -x)]);
```

In the `eval` syntax, notice that you use the dot notation to access the members of the structure or object `cc`. Using this notation to access properties of an object is common in MATLAB operations.

- 11** Everything is ready for you to build your project. Use the following command to start the build:

```
build(cc,all,60);
```

Depending on your configuration, building a project can be slow and the default time-out value may not be long enough. Therefore, an explicit 60 second time-out is supplied as an input argument with the build syntax. Wait for the build operation to complete and press Enter before proceeding.

- 12** You have built the executable. Now load the program to your target with this code. Note that you are loading the program from your temporary directory and you provide an explicit time-out value of 40 s:

```
load(cc,['tempdir','board.hiltut.loadfile'],40)
```

Again, this load might fail for a number of reasons. One might be that your target DSP needs different linker command (*.cmd) and vector table source (*.asm) files. If so, attempt to rebuild the executable with the appropriate files and then load it from the CCS IDE. After you load the executable successfully, continue the tutorial.

- 13** To make sure the working directory is correct for the rest of the tutorial, reset it to the project path from step 5:

```
cd(cc,projpath); % Restore CCS working directory
```

Initialize the Embedded C Variables and Use read and write

Direct access to DSP memory is powerful, but for C programmers it can be more convenient to manipulate memory in ways more like working with the defined C variables. Link for Code Composer Studio implements this approach by using MATLAB objects as representations of embedded entities (entries in the symbol table for your project).

This section of the tutorial starts by investigating data values in the program and manipulating them using embedded objects. For that you apply the method `list` with variable `idat`, which queries CCS for information about the variable. `idat` is a global C variable in the tutorial program `hiltut.c`.

- 1** Enter the following code to ensure that the embedded C variables in your project are initialized. In this tutorial, `main` contains all the variables required for the project. Otherwise the methods for accessing variables

outside of main do not work because they are not initialized by running to main:

```
run(cc, 'main')
```

When you look at the project in CCS, you see that the program is running — CCS shows the CPU as running.

- 2** Function `list` provides one way to gather information about the embedded variables and functions in your project. Use the following `list` examples to explore the tutorial program. In each example, you could assign the return structure to an output argument by including an argument on the left side of the `list` syntax:

```
list(cc, 'function')
Warning: NAME 'ASM$' is an invalid ML structure fieldname. The
dollar ($) character is replaced by 'DOLLAR'.
```

```
ans =
```

```
    ASMDOLLAR: [1x1 struct]
    fir_filter: [1x1 struct]
         main: [1x1 struct]
    sin_taylor: [1x1 struct]
sin_taylor_vect: [1x1 struct]
```

This syntax, with the **function** keyword, returns a structure that contains the names of all the functions in your project:

```
list(cc, 'variable')
ans =

    coeff: [1x1 struct]
    ddat: [1x1 struct]
    din: [1x1 struct]
    dout: [1x1 struct]
    ibuf: [1x1 struct]
    idat: [1x1 struct]
myString: [1x1 struct]
myStruct: [1x1 struct]
    nbuf: [1x1 struct]
```

```

ncoeff: [1x1 struct]
obuf: [1x1 struct]
data: [1x1 struct]
min: [1x1 struct]
result: [1x1 struct]

```

Switching to the **variable** keyword returns a structure that contains the names of all the variable defined in the tutorial:

```

list(cc, 'type')
ans =

TAG_myStruct: [1x1 struct]
TAG_myEnum: [1x1 struct]

```

The last keyword, **type**, returns all the data types defined in the program, in a structure in your workspace.

- 3** To focus on just one variable, the next code example returns the information about one variable, named `idat`. Again, the results come back in structure form. In this case, you use an output argument to store the structure:

```

listI = list(cc, 'variable', 'idat')
listI =

idat: [1x1 struct]

```

`idat` is a global variable, as you see from the structure contents.

- 4** Now take a look at the `idat` element in structure `listI`:

```

listI.idat
ans =

name: 'idat'
isglobal: 0
address: [17468 0]
size: [2 3]
bitsize: 16
type: 'short'

```

`list` generates quite a lot of information about the embedded `idat` variable. However, an even more useful method is `createobj`, which constructs a MATLAB object to represent the C variable — in this case `idat`. The object you construct using `createobj` acquires the properties of the C variable. Applying the object returned by `createobj`, you can directly read the entire variable or access individual elements of the variable, such as the elements of an array for array variables.

To this point in the tutorial, you have applied all the methods to the original `cc` object that you created with `ccsdsp`. The `cc` object represents communication with a particular digital signal processor in CCS.

For the remainder of this tutorial, you apply methods to many different objects. In typical object-oriented programming fashion, the action performed by a method depends on its object. The relevant or target object is always the first input argument passed to the method. For example, in the following section `cvar` is an object representing the embedded `idat` variable.

- 1** Use `createobj` to construct a MATLAB object that accesses the embedded variable `idat`. By assigning the return value to the variable `cvar`, you have a handle in MATLAB that represents access to `idat` on your DSP target:

```
cvar = createobj(cc,idat)
NUMERIC Object stored in memory:
Symbol name      : idat
Address         : [ 17468 0]
Datatype        : unsigned short
Wordsize        : 16 bits
Address units per value : 2 au
Representation   : unsigned
Size            : [ 2 ]
Total address units : 4 au
Array ordering   : row-major
Endianness      : little
```

- 2** Now you use `cvar` to get information about `idat`, or to manipulate the way MATLAB interprets `idat` in your workspace. Try the code examples below to see how some of the data manipulation methods work:

```
get(cvar,'size') % Size of cvar should be 2-by-3 as defined in
```

```

                                % our DSP application.

ans =

      2      3

read(cvar)    % Reads the entire embedded matrix into the MATLAB
              % workspace.

ans =

      -1      508      647
      7000      8      619

readhex(cvar) % Reads cvar in hex.

ans =

      'FFFF'      '1FC'      '287'
      '1B58'      '8'      '26B'

readbin(cvar) % Reads cvar in binary.

ans =

      '1111111111111111'      '0000000111111100'      '0000001010000111'
      '0001101101011000'      '0000000000001000'      '0000001001101011'

```

The previous read examples return the entire `idat` matrix to your MATLAB workspace. You can read and write selected elements of `idat` by indexing into it. Being able to read or write with objects is easier and more powerful than reading and writing to raw DSP memory, or manually figuring out the right address offsets for your data arrays.

- 3** In the next code examples, you use indexing to return specific elements of embedded variable `idat`, as accessed by `cvar`. Note the `write` method for changing the contents of `cvar` from MATLAB:

```
read(cvar,[2 1])    % Read element specified by column 2, row 1
```

```
ans =  
  
    7000  
  
write(cvar,[2 1], -7000)    % Modifies 7000 to -7000.  
read(cvar)  
  
ans =  
  
    -1    508    647  
 -7000     8    619
```

Use read, write, cast, and convert with Objects

The previous read operations with `cvar` took raw memory values and converted them into equivalent MATLAB numeric values. The conversion that gets applied is controlled by the properties of `idat`, which were initially configured in `createobj` to settings appropriate for your DSP architecture and C representation. In some cases, changing these default conversion properties can help your development process. Several properties of the object, such as `endianness`, `arrayorder`, and `size`, can be directly modified using `set`. Methods such as `convert` and `cast`, which adjust multiple object properties simultaneously, enable you to make more complex changes from MATLAB.

- 1 To introduce the idea of changing the representation in MATLAB of an object, try the following `set` function on `cvar`, which changes the way MATLAB interprets `idat`. After the change, check that `cvar` is indeed smaller:

```
set(cvar,'size',[2]) % Reduce size of 'idat' to first 2 elements.  
read(cvar)  
  
ans =  
  
    -1    508
```

- 2 Now change the data type of `cvar` using `cast`:

```
uicvar = cast(cvar,'unsigned short')
```

NUMERIC Object stored in memory:


```

Symbol name      : idat
Address         : [ 17468 0]
Datatype       : unsigned short
Wordsize       : 16 bits
Address units per value : 2 au
Representation  : unsigned
Size           : [ 2 ]
Total address units : 4 au
Array ordering  : row-major
Endianness     : little

```

Using `cast` in this way changes the representation of `cvar` from double precision to unsigned short. As a result, MATLAB interprets the first value in `cvar` as the unsigned equivalent of -1, as shown when you read the new `uicvar` object. And do note that `uicvar` is a new object, not an alias or handle to `cvar`, but fully independent of `cvar`.

```

read(uicvar)
ans =

        65535         508

```

In the next step you meet the method `convert`, which changes the data type of the specified object, rather than creating a new object with the new data type.

- 3** For the second data type conversion method, use `convert` with `cvar` to change the data type for `idat` in MATLAB:

```

convert(cvar,'unsigned short')

NUMERIC Object stored in memory:
Symbol name      : idat
Address         : [ 17468 0]
Datatype       : unsigned short
Wordsize       : 16 bits
Address units per value : 2 au
Representation  : unsigned
Size           : [ 2 ]
Total address units : 4 au

```

```
Array ordering      : row-major
Endianness         : little
```

```
read(cvar)
ans =

        65535        508
```

Note that the embedded object `cvar` has the new data type and size; it is not a new embedded object. Writing this version of `cvar` back to the DSP memory would cause `idat` to take on the new data type definition.

Embedded DSP variables such as strings, structures, bitfields, enumerated types, and pointers can be manipulated in exactly the same way. The following operations demonstrate manipulations on structures, strings and enumerated types. In particular, note the method `getmember`, which extracts one field from a structure as a new MATLAB object in your workspace.

- 4** To demonstrate `getmember`, you need an embedded object that accesses a structure in memory. In the following code, you replace your current `cvar` object with one that represents a structure named `myStruct`, an embedded C structure in the symbol table for the tutorial program:

```
cvar = createobj(cc, 'myStruct')
STRUCTURE Object stored in memory:
Symbol name          : myStruct
Address              : [ 17440 0]
Address units per value : 28 au
Size                 : [ 1 ]
Total Address Units   : 28 au
Array ordering       : row-major
Members              : 'iy', 'iz'

read(cvar)
ans =

iy: [2x3 double]
iz: 'MatlabLink'
```

myStruct is a fairly complex structure containing a variety of data types, including enumerated data and strings. Since you use the elements of myStruct in the next steps, carefully review it so you see what it contains and how.

- 5** In this step you read, write, and manipulate the elements of myStruct. As you enter each command, try to determine what you expect to get back from MATLAB. Notice that we ask you to perform read operations between other operations. read lets you see the changes you make in DSP memory when you write variables to CCS, not just in MATLAB:

```
write(cvar,'iz', 'Simulink')
cfield = getmember(cvar,'iz') % Extract iz field from cvar

ENUM Object stored in memory:
Symbol name      : iz
Address          : [ 17464 0]
Wordsize        : 32 bits
Address units per value : 4 au
Representation   : signed
Size            : [ 1 ]
Total address units : 4 au
Array ordering   : row-major
Endianness      : little
Labels & values  : MATLAB=0, Simulink=1, SignalToolbox=2,
                  MatlabLink=3, EmbeddedTargetC6x=4

write(cfield,4) % Write to same cvar enumerated variable by value
read(cvar)
ans =

    iy: [2x3 double]
    iz: 'EmbeddedTargetC6x'

cstring = createobj(cc,'myString') % cstring represents an
                                     % embedded C structure

STRING Object stored in memory:
Symbol name      : myString
Address          : [ 17512 0]
Wordsize        : 8 bits
```

```
Address units per value : 1 au
Representation          : signed
Size                   : [ 29 ]
Total address units    : 29 au
Array ordering         : row-major
Endianness             : little
Char Conversion Type   : ASCII
read(cstring)

ans =

Treat me like an ANSI String

write(cstring,7,'ME')
read(cstring)
ans =

Treat ME like an ANSI String

write(cstring,1,127) % Set first location to numeric value 127
                    % (nonprinting ASCII character)
readnumeric(cstring) % Read cstring as equivalent numeric values
ans =

Columns 1 through 13

    127    114    101    97    116    32    77    69    32    108    105    107    101

Columns 14 through 26

    32    97    110    32    65    78    83    73    32    83    116    114    105

Columns 27 through 29

    110    103     0
```

Construct a function Object

In step 12 you performed a number of operations on `myStruct` in your workspace, and between MATLAB and CCS.

Manipulating embedded data is useful, but eventually you must contend with embedded functions, not just variables. To facilitate your debugging and verification work, the Link for Code Composer Studio provides objects for accessing embedded functions directly from MATLAB. This permits you to execute any C-callable function on your target from MATLAB for hardware-in-the-loop functionality.

The first step in running embedded functions from MATLAB is to make function objects by applying the (now familiar) `createobj` on `cc`. Just like variables, use `list` to retrieve information about functions that you access.

The following steps create an object `listI` that you use to access the embedded function `sin_taylor`.

- 1 Get information about an embedded function, then create an object to access the function. Your target function is `sin_taylor`:

```
listI =

    sin_taylor: [1x1 struct]

listI.sin_taylor

ans =

    name: 'sin_taylor'
 filename: 'hiltut.c'
  uclass: 'function'
islibfunc: 0
 address: [1x1 struct]
 linepos: [86 116]
 funcvar: {'a1' 'a2' 'a3' 'acc' 'x' 'xpow'}

cfunc = cc.createobj('sin_taylor')    % Create function object

FUNCTION Object
Function name      : sin_taylor
File found        : hiltut.c
```

```
Start address      : [12328 0]
All variables     : a1, a2, a3, acc, x, xpow
Input variables   : x
Return type       : short
```

At this point, you are ready to run function object `listI`.

Embedded function `sin_taylor` computes a fixed-point sine function using four terms of the Taylor series representation. Let's use your new object `cfunc` to verify the embedded function. From the information returned by `list`, you know that the input fixed-point data format is `Q2.13` and the output is `Q1.14`.

- 2** To run `sin_taylor`, you provide a number between $(-\pi)$ and (π) to use for the sine calculation. Enter a value as shown in this code:

```
userval = pi/2; % Use any value between -pi and pi.
```

- 3** Now run `sin_taylor` using `userval` and the `cfunc` object:

```
sintf = run(cfunc, 'x', (userval*2^13)/2^14));
```

The numeric values in the command provide scaling for the binary point in `userval` to prevent the output (`sintf`) from saturating in `Q1.14` format.

The returned values from the MATLAB `sin` function and `sin_taylor` should match quite closely.

Use Methods That Work with Function Objects

In some cases you may find it useful to alter function object properties that were initialized to reflect your DSP source code. Several function object properties, like `returntype`, `saveregs`, and `timeout`, can be set using `set`. For applying other complex properties, Link for Code Composer Studio offers the `cast` and `convert` methods.

At times you might like to change the properties of an object while keeping the original object unchanged, and, if the object is a function, apply the new properties to a copy of the function. The method `copy` does just that. In the following steps of the tutorial, you create a copy of `cfunc` and use the copy for program debugging purposes.

- 1 Create the copy of your cfunc object, and get the properties for it:

```

cfunc_copy = copy(cfunc)
FUNCTION Object
  Function name      : sin_taylor
  File found        : hiltut.c
  Start address     : [12328 0]
  All variables     : a1, a2, a3, acc, x, xpow
  Input variables   : x
  Return type       : short

getprop(cfunc_copy, 'outputvar') % Get the function return type

NUMERIC Object stored in register(s):
  Symbol name       :
  Register          : A4
  Datatype          : Unknown
  Wordsize          : 16 bits
  Register units per value : 1 ru
  Representation    : signed
  Bit padding (post) : 16
  Size              : [ 1 ]
  Total register units : 1 ru
  Array ordering    : row-major

```

As you review the information returned by `getprop`, notice the difference in the `wordsize` property between `cfunc` and `cfunc_copy`.

- 2 With the copy of `cfunc` in your workspace, convert the output data type to `int8` from Q1.14. Recall that `int8` is both a MATLAB data type and a C native data type:

```
convert(cfunc_copy.outputvar, 'int8')
```

Property `outputvar` holds the data type specification for the returned value from `sin_taylor`.

- 3 Entering the following command at the prompt

```
int8_OUT = run(cfunc_copy, 'x', (userval*2^13)/2^14)
```

executes the copy of the `sin_taylor` function that you modified to have the output data type `int8` instead of the original output data type.

Function calls support different types of DSP variables, such as strings, structures, bitfields, enumerated types, and pointers. In the next examples, you create an object that accesses `sin_taylor_vect`, a vectorized version of `sin_taylor`.

To prepare to run `sin_taylor_vect`, you create input and output buffer objects, each containing 10 memory locations; you supply the start addresses of both buffers to the function object; and you run the function from MATLAB with the `run` method. With vectors needed for its input and output, `sin_taylor_vect` uses buffers to store the data in both directions. As a function that used one input value and returned one output value, `sin_taylor` did not require buffers.

4 Enter the following commands to construct objects that access `sin_taylor_vect` and input and output buffers:

```
cfunc_vec = cc.createobj('sin_taylor_vect') % Yet another object
ibufobj = createobj(cc,'ibuf'); % Create input buffer object
obufobj = createobj(cc,'obuf'); % Create output buffer object
```

5 With the buffer objects in place, make the input data vector and write the data to your input buffer:

```
inputdata = [-pi:0.1:pi]; % Input data to write to the DSP target
write(ibufobj,int16(inputdata*2^13)); % Write data to buffer with
                                     % scaling
write(obufobj,int16(zeros(1,63))); % Set output buffer to zeros
read(ibufobj) % (optional)          % Verify data initialization
ans =
```

Columns 1 through 6

```
-25735      -24916      -24097      -23278      -22459      -21639
```

Columns 7 through 12

```
-20820      -20001      -19182      -18363      -17543      -16724
```


Columns 13 through 18

-15905	-15086	-14267	-13447	-12628	-11809
--------	--------	--------	--------	--------	--------

Columns 19 through 24

-10990	-10171	-9351	-8532	-7713	-6894
--------	--------	-------	-------	-------	-------

Columns 25 through 30

-6075	-5255	-4436	-3617	-2798	-1979
-------	-------	-------	-------	-------	-------

Columns 31 through 36

-1159	-340	478	1297	2116	2936
-------	------	-----	------	------	------

Columns 37 through 42

3755	4574	5393	6212	7032	7851
------	------	------	------	------	------

Columns 43 through 48

8670	9489	10308	11128	11947	12766
------	------	-------	-------	-------	-------

Columns 49 through 54

13585	14404	15224	16043	16862	17681
-------	-------	-------	-------	-------	-------

Columns 55 through 60

18500	19320	20139	20958	21777	22596
-------	-------	-------	-------	-------	-------

Columns 61 through 63

23416	24235	25054
-------	-------	-------

read(obufobj) % (optional) Should be zeros

ans =

Columns 1 through 13

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0
Columns 14 through 26
0 0 0 0 0 0 0 0 0 0 0 0 0 0
Columns 27 through 39
0 0 0 0 0 0 0 0 0 0 0 0 0 0
Columns 40 through 52
0 0 0 0 0 0 0 0 0 0 0 0 0 0
Columns 53 through 63
0 0 0 0 0 0 0 0 0 0 0
```

- 6** You've done all the preparation — now run `sin_taylor_vect`. Remember that the object you named `cfunc_vec` accesses `sin_taylor_vect`:

```
outputAddress = run(cfunc_vec, 'x', ibufobj.address(1), 'y', ...
obufobj.address(1), 'npts', 63);
```

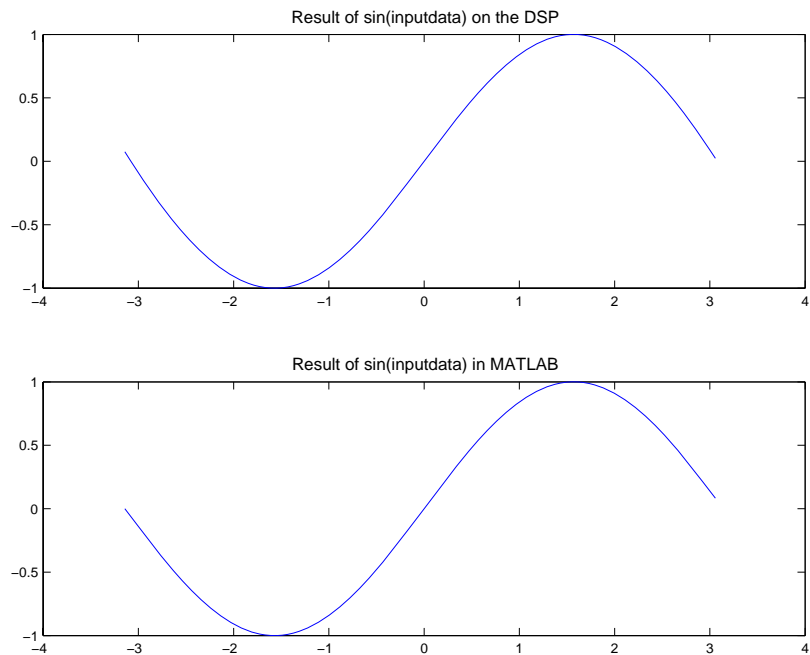
Notice how the input and output parameters correspond to the prototype of the function. Also notice how input parameters are supplied in pairs — parameter name/parameter value.

- 7** You need to use a new method, `deref`, to read the output data buffer. The value in object property `outputvar` is a pointer. To get to the actual data, you dereference the pointer (just as you do in C, since you are working in C). The next code does the dereferencing for you:

```
outputdataAddress = deref(cfunc_vec.outputvar);
outputdataAddress.size = 63; % Need to read the next 63
                           % addresses (obufobj)
outputdata = read(outputdataAddress)/2^14; % Get output scaling
                                           %for binary point
```

- 8** If you are interested in seeing what you have done, the following code plots the results from running `sin_taylor_vect` on your input data set. Comparing the output from the MATLAB `sin` function and `sin_taylor_vect` gives you an idea of how your algorithm performs on your DSP target:

```
subplot(2,1,1)
plot(inputdata,outputdata)
title('Result of sin(inputdata) on the DSP')
a = gca;
set(get(a,'title'),'fontsize',10);
set(a,'fontsize',8);
set(a,'fontweight','light') ;
subplot(2,1,2)
plot(inputdata,sin(inputdata))
title('Result of sin(inputdata) in MATLAB')
b = gca;
set(get(b,'title'),'fontsize',10);
set(b,'fontsize',8);
set(b,'fontweight','light')
```



Among other things, this example plotting technique might be handy for other plotting tasks.

Construct Different Objects and Work with Them

In the previous tutorial section, you created an object that accessed a C function and ran the function in your project from MATLAB. The Link for Code Composer Studio also supports calling library functions — those functions in your project that are precompiled and callable directly from your C program — in your project from MATLAB. Library functions do not build when you build your project in CCS or from MATLAB and therefore do not have the function prototype available that compiled functions provide and that MATLAB needs to get the information about the function.

The difference between using function objects with library functions and regular C functions is you must use the method `declare` with library functions

to provide the function declaration for the object to MATLAB. Because CCS cannot provide full information about library functions, MATLAB gets its library function information from your `declare` operation. The `declare` method accepts C declaration strings for any functions.

In this part of the tutorial, you create an object to access the `fir_filter` filter library function, using `declare` to supply the function declaration to MATLAB. Then you use the object to run the function in CCS and on your target. To introduce the concepts needed to work with typedefs you might have defined in your projects, you use `add` in this process to define some typedefs in MATLAB to include in your `cc` object. Your additional typedefs remain available as long as the `cc` object exists for this project.

- 1 Start this section by creating and plotting the frequency response for a lowpass FIR filter in MATLAB. Use `fir1` from the Signal Processing Toolbox to create the FIR filter. Later in this section you compare the results of filtering with this filter to the results of filtering with an FIR filter function (`fir_filter`) on your target — they should match closely, within the differences caused by the filter coefficients being stored on the target with lower precision:

```
n = 10;
wb1 = 0.3;
bcoeff = fir1(n,wb1);
[sco sw]=freqz(bcoeff,1);
scodb = 20*log10(abs(sco));
swdb = sw./pi;
h = figure;
plot(swdb,scodb);
hold on; grid on; % Save the figure to add another later.
nfrm = 128;
cscaling = 2^15;
ncoeff = length(bcoeff);
```

To plot the filter magnitude response, you could have used the Filter Visualization Tool (FVTool), as shown here:

```
fvtool(bcoeff,1);
```

Using FVTool gives you access to a full range of analyses for your lowpass filter. Plotting the magnitude response in the more conventional way allows you to compare the results of running the same FIR filter on your target that you do later in this tutorial.

- 2** Now create handles to three filter parameters in CCS — `coeff` (filter coefficients), `nbuf` (input buffer), and `ncoeff` (number of filter coefficients; equal to `[filter order + 1]`):

```
coeff = createobj(cc, 'coeff');  
nbuf = createobj(cc, 'nbuf');  
ncoeff = createobj(cc, 'ncoeff');
```

- 3** You need input and output objects so create them:

```
din = createobj(cc, 'din');  
dout = createobj(cc, 'dout');
```

- 4** To run the filter function, you create and scale input data for the function to process. The following code creates an input data set with scaling:

```
datain = randn(nfrm, 1);  
glim = max([abs(max(datain)) abs(min(datain))]));  
dscale = 2^15 / (glim * 0.99);  
idin = int16(dscale * datain);
```

- 5** Provide data to your target to initialize the filtering function (`fir_filter`) by writing the required input data and filter specifications to the target:

```
write(coeff, int16(cscaling.*bcoeff));  
write(din, idin);  
write(ncoeff, n);  
write(nbuf, nfrm);
```

After you have initialized your input data and written the data to the target, you are ready to run the library function `fir_filter` in the tutorial project.

- 6** First create an object to access `fir_filter`:

```
ff = createobj(cc, 'fir_filter'); % Expect a warning message
```

Recall from earlier comments in this tutorial that library functions behave slightly differently from compiled C code functions. When you try to create a function object to access a library function, you get a warning message telling you to use `declare` to supply the function declaration. For library functions you supply the function declaration to MATLAB using the `declare` method. In spite of the warning message, MATLAB creates `ff` with default property values.

- 7** Use `declare` to provide the function declaration for `fir_filter` to MATLAB:

```
declare(ff,'decl','short fir_filter (short *x, short *h,...
short *r,short **dbuffer, unsigned short nh, unsigned short nx)');
```

- 8** Add a custom type definition (C typedef) `INT16` to the type definitions in `cc`. Use `list` to see the available type definitions:

```
add(cc.type,'INT16','int16');
list(cc.type) % Display existing defined types. Includes INT16.
Defined types : Void, Float, Double, Long, Int, Short, Char, INT16
```

- 9** Running the function requires one more object — a pointer to a buffer. Use `createobj` to create the object that accesses `dbptr` in the symbol table for your project:

```
dbptr = createobj(cc,'dbptr');
```

- 10** Now run `fir_filter` from MATLAB. Position the program counter to the beginning of the function, set the input argument values `x`, `r`, `h`, `nh`, `n`, `nr`, and `nfrm`, and run `ff`:

```
goto(ff,'x',din.address(1),'h',coeff.address(1),...
'r',dout.address(1),'nh',n,'nr',nfrm);
execute(ff); %
```

You took advantage of the ability to use `goto` to both position the PC and set values for the `fir_filter` function input arguments. This feature can be convenient for developing and testing algorithms with function call work.

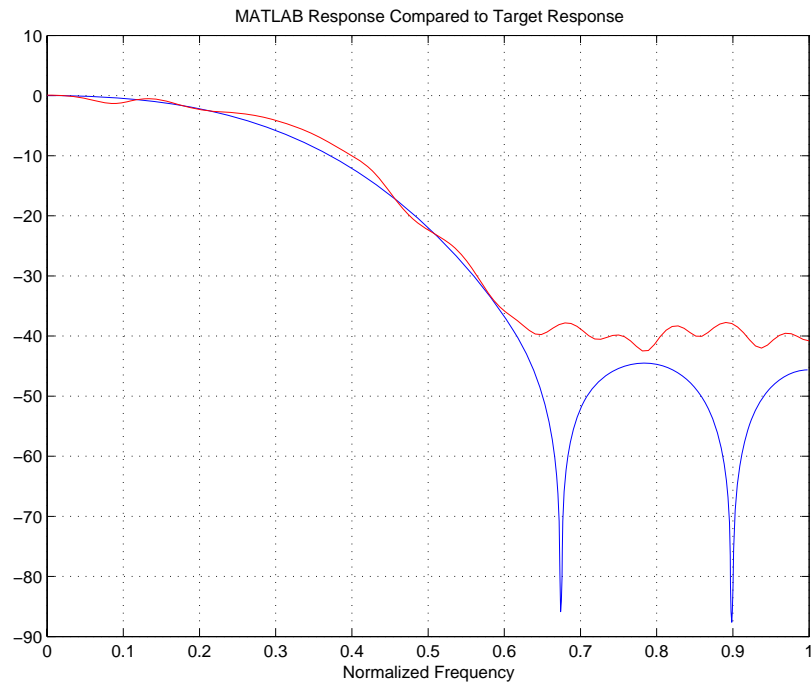
- 11** After the filtering process finishes, use `read` to get the results back from CCS to MATLAB:

```
idout = read(dout);
```

- 12** Plot `idout` to see the magnitude response of the FIR filter on your target:

```
[sout wsd]= pwelch(double(idout));  
sin = pwelch(double(idin));  
runningsum = (sout./sin);  
wplotdb = 10*log10(runningsum/1);  
wsdn = wsd/pi;  
plot(wsdn,wplotdb,'r');  
title('Target Generated Filter Response');
```

Compare this response to the magnitude response from FVTool you created earlier. Your target stores the filter coefficients slightly differently from MATLAB, so the results are not identical — the filters are not quite the same.



Close The Tutorial and Clean Up

Finally, the objects created during this tutorial have COM handles to CCS. Until you delete these handles, the CCS process remains in memory. Closing MATLAB removes these handles, but in some cases you may find it useful to delete them without closing MATLAB. Use `clear` to remove objects from your MATLAB workspace and delete the handles that objects contain. `clear all` deletes everything in your workspace. To retain your MATLAB workspace contents while removing specific objects, use `clear` on the objects to remove, such as those derived from your `ccsdsp` object, including all embedded objects returned by `createobj`.

In addition the tutorial performs a `close` operation to remove the tutorial project from CCS.

- 1 First close your project file from MATLAB by entering

```
close(cc,'projfile','project') % Clean-up CCS by closing the...  
                                %project file
```

2 To remove the objects you created during the tutorial, enter

```
clear cc cvar cfield uicvar cstring ibufobj obufobj cfunc  
cfunc_vec cfunc_copy
```

at the command line.

If you do not care about keeping other variables and objects that were in your MATLAB workspace when you started this tutorial, use `clear all` to remove everything from your workspace — objects, variables, and more — in one operation.

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. Since 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, Link for Code Composer Studio 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:

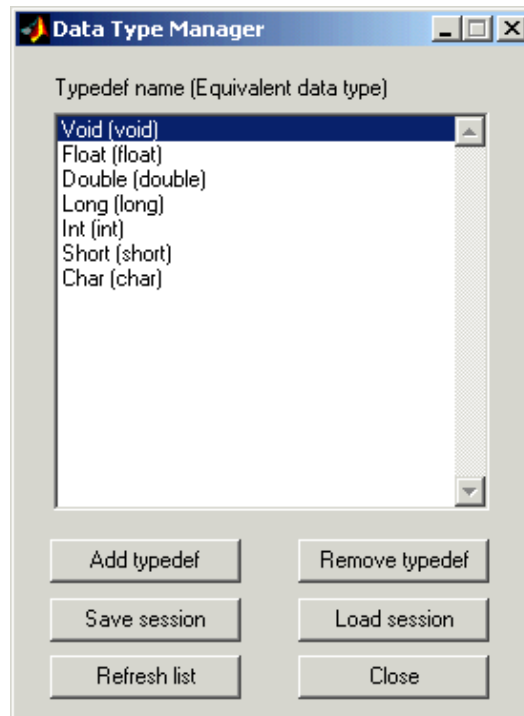
- Make your code more accessible by providing more information about the variable(s)
- Create a Boolean data type that C does not provide
- Define structures in your programs
- Define nonstandard data types

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 ccscsp 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 an M-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 to 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)

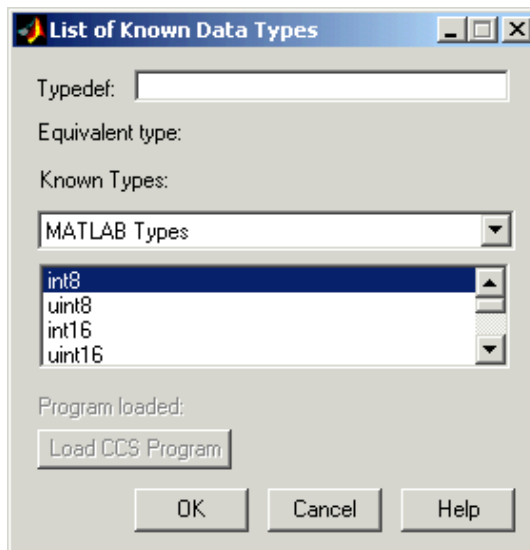
Each custom type definition added in the DTM becomes part of the `ccsdsp` object passed to the DTM in `datatype manager(objectname)`. The list of data types in the object, both default and custom, is available by entering

```
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 target 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 target 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 When the load process works, 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 target do not appear in the **Loaded program** option. 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 target.

Follow the example procedure to add type definitions to your project. To go directly to a specific typedef example, click one of these links:

- “Add a MATLAB type definition” on page 2-115
- “Add an enumerated type definition” on page 2-116
- “Add a structure typedef” on page 2-117

Create an object and load a program.

- 1 Create a `ccsdsp` object.

```
cc=ccsdsp;
```

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

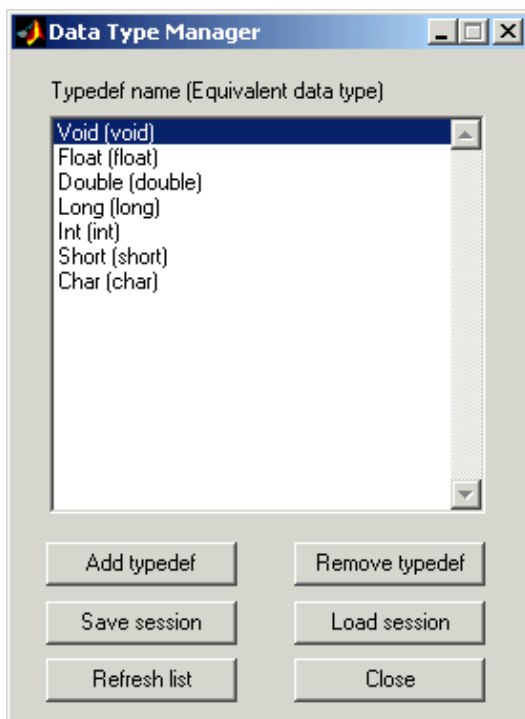
```
load(cc, 'c6711dskwdnoisf_c6000_rtwD\c6711dskwdnoisf.out');
```

loads the executable file from the model `c6711dskwdnois.mdl` on the target.

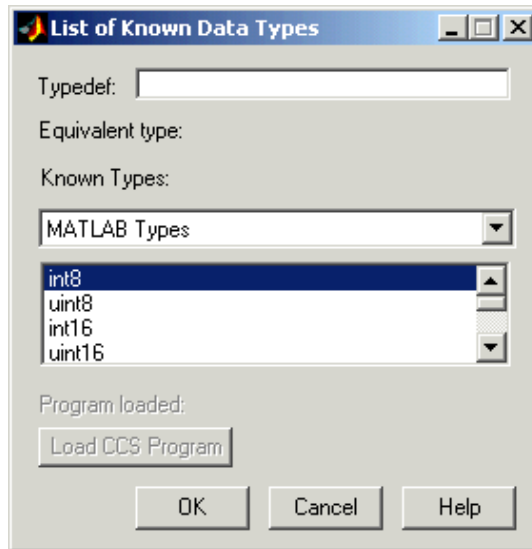
- 3** Start the DTM with the object you created.

```
datatypemanager(cc);
```

The DTM starts, showing the default data types.



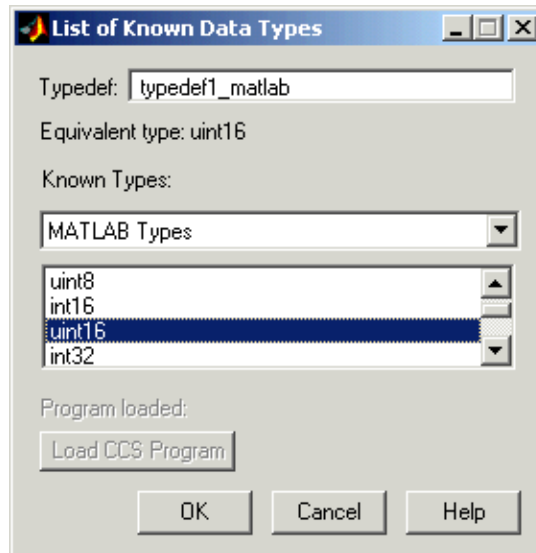
- 4 Click **Add typedef** to add your first custom data type. The List of Known Data Types dialog box appears as shown.



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.

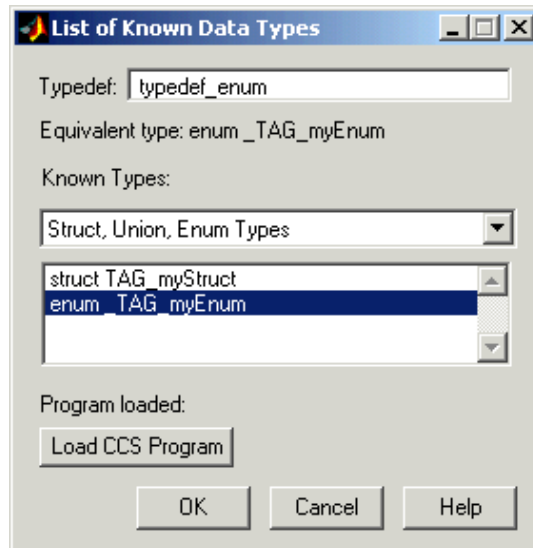


- 7 Click **OK** to close the dialog box and add the new type definition to the **Typedef name** list.

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.

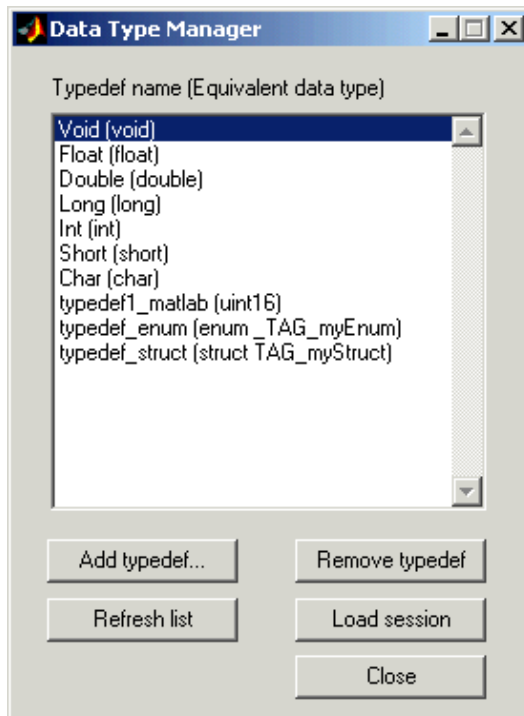


- Click **OK** to close the dialog box and add `typedef_enum` to your defined types that MATLAB recognizes.

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 can interpret the data correctly. For example, MATLAB interprets the `typedef1_matlab` data type as `uint16`.

Clicking **Close** in the DTM prompts you to save your session. Saving the session creates an M-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 M-file is a function definition, where the name of the function is the filename of the session you saved. In the stored M-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 M-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.

An interesting note — all the operations you performed adding and removing data types in the DTM during the session are stored in the generated M-file that you save. This has the effect of storing mistakes you made while creating or removing type definitions. One consequence of storing mistakes is when you load your saved session into the DTM, you see the same error messages you saw, if any, when you created the data types in the session. You might find this disconcerting.

Reference for the Properties of Embedded Objects

This section presents details of the properties that apply to the embedded objects in Link for Code Composer Studio. The reference information contained can help you learn about using the links and objects.

Property Reference Format and Contents

Ordered alphabetically by property name, references include

- Property name heading
- Description
- Property characteristics, including
 - Data type
 - Default value
 - Read/Write status
- Range of valid property values
- One or more examples using the property
- Referrals to related properties where appropriate

Some reference pages do not include all the features listed; in particular, some pages may not provide examples or the range of valid property values or referrals.

Functions

address

Description. Reports the starting address of the symbol the object references — either a memory address or a register name. In some cases the address is in [Offset Page] format when the processor supports memory pages and the address is a location in memory.

Characteristics. Either a numeric value (for memory locations) or alphanumeric value (for register locations), this is a writable value.

If you change the offset and page values for the property, the object points to a different location in memory. Changing the address property does not affect the location of the symbol.

Range. Covers the entire range of addresses available on the target.

apiversion

Description. Contains a string that defines the version of the CCS application program interface (API) being used by the link object.

Characteristics. A string value. The first entry in the square brackets is the major version number and the second entry is the minor revision number. You cannot set this value — it is read only.

Range. Any ASCII characters that make up the name and version number of the API.

Examples. Create a link object and use `get` to review the object properties. For this object, the `API version` returns 1.2 and `apiversion` is [1 2]. The API version may not be the same as the version of CCS:

```
cc=ccsdsp

CCSDSP Object:
  API version      : 1.2
  Processor type   : TMS320C6711
  Processor name   : CPU_1
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

get(cc)
      rtdx: [1x1 rtdx]
```

```
apiversion: [1 2]
ccsappexe: 'D:\Applications\ti\cc\bin\'
boardnum: 0
procnum: 0
timeout: 10
page: 0
```

arrayorder

Description. Specifies the manner in which the object interprets data stored linearly in memory, whether as rows or columns of an array.

Characteristics. A string with one of two possible values — row-major (C style interpretation) or col-major (normal MATLAB ordering).

Range. Allowed strings are row-major and column-major.

Examples. When you have nine values in memory, such as 1, 2,..., 9, the arrayorder property value determines how to build an array from the values:

- In row-major order, the values form the 3-by-3 array by filling the array row by row and left to right:

```
1 2 3
4 5 6
7 8 9
```

- In column-major order, the values form the 3-by-3 array by filling the array column by column and top to bottom:

```
1 4 7
2 5 8
3 6 9
```

You can increase the number of array dimensions without limit.

binarypt

Description. Specifies the location of the binary point in a value. To interpret the actual value of a value in memory, you need both the data type and binary point to convert correctly from the binary or hexadecimal representation to decimal. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted. Since the object uses double-precision representation, the word size and binary point form the basis for simulating fixed-point values.

Characteristics. A positive or negative integer.

Range. `binarypt` ranges from 0 to the word size. You can use negative binary point locations and binary point locations larger than the word size, to the limit of double-precision representation.

Referrals. See also `wordsize`.

bitsperstorageunit

Description. Reports the smallest number of bits per address location (addressable unit) on the target. Memory locations and registers may have different values on a target. Different processors can use different values as well.

Characteristics. An integer.

Range. Depends on the target processor. Usually 8, 16, or 32 bits.

Referrals. See also `numberofstorageunits` and `storageunitspvalue`.

boardnum

Description. Specifies the target board or simulator with which the link object communicates.

Characteristics. An integer. This is a read-only value determined when you create link objects and select your target.

Range. Integer values ranging from 0 for the first board up to the number of boards that CCS recognizes configured on your machine. Note that both simulators and hardware count as boards.

ccsappexe

Description. Reports the full directory path to the CCS executable.

Characteristics. A string that shows the path to your CCS installation. You cannot change this string except by moving your CCS storage location.

Examples. If your CCS installation is in a folder called Applications on your D: drive, you might see a string such as

```
'D:\Applications\ti\cc\bin\ '
```

for the `ccsappexe` property value when you use the command

```
cc.ccsappexe
```

at the MATLAB prompt.

charconversion

Description. Specify the character set that read and write use to interpret data in memory or when transferring data to target memory. When you set the `charconversion` property, you are telling read or write to interpret the data, either in MATLAB or on the target, as though they represent values in the specified character set.

For `read`, `charconversion` tells MATLAB to return the values from memory as characters from the specified data set. For `write`, `charconversion` tells MATLAB to write the data to target memory as the numeric equivalents of the specified character set. Recall that all data in memory is numeric. `charconversion` defines how the numeric values in memory become characters in MATLAB. And how characters in MATLAB become numeric values on the target.

Characteristics. This is a string and should be entered as a string in single quotation marks.

Range. The only valid entry for charconversion is `ascii`.

endianness

Description. Specifies whether to interpret the bit pattern in memory in little-endian or big-endian format. Big-endian format assumes the least significant bit (LSB) is last in a word that spans more than one addressable unit in memory; little-endian assumes the LSB is first in a word that spans multiple addressable units.

Characteristics. Property values are strings, either `little` or `big`. You can change the state within the object, which changes the way MATLAB interprets the bits stored in memory on your target.

Range. You have two options for endianness — `little` or `big`.

Examples. When you have a variable in memory, such as `ddat` from the link object tutorial, creating a numeric object to access `ddat` shows you whether `ddat` is big endian or little endian:

```
ddat = createobj(cc,'ddat')

NUMERIC Object
  Symbol Name      : ddat
  Address          : [ 40072 0]
  Wordsize        : 64 bits
  Address Units per value : 8 AU
  Representation   : float
  Binary point position : 0
  Size            : [ 4 ]
  Total address units : 32 AU
  Array ordering   : row-major
  Endianness      : little

get(ddat)
           address: [40072 0]
```

```
bitsperstorageunit: 8
numberofstorageunits: 32
    link: [1x1 ccsdsp]
    timeout: 10
    name: 'ddat'
    wordsize: 64
storageunitspervalue: 8
    size: 4
endianness: 'little'
arrayorder: 'row-major'
    prepad: 0
    postpad: 0
represent: 'float'
binarypt: 0
```

filename

Description. Specifies the name of the file in the project that contains the function declaration. When you create an object that accesses a function, MATLAB returns the name of the file in `filename`. When the target function is a library function, `filename` is empty.

Characteristics. A string that contains the full path name to a file.

Range. Any valid filename and directory path.

inputnames

Description. Defines and contains the names of input arguments to a function in your project. For library functions, `inputnames` is empty until you use `declare` or `getinput` to define the input arguments for the function.

Characteristics. A character string in the form of an `mxAarray`.

Range. Any valid C variable name string.

inputvars

Description. The objects that represent each input argument to a function when you create a function object to access a specific function. When you create a new function object, MATLAB creates appropriate objects to access each input argument to the function.

Characteristics. An object that represents the input argument type, such as numeric or pointer. These are handles to objects.

Range. Any valid object in Link for Code Composer Studio.

label

Description. Contains the names of the fields in an enumerated object or memory location.

Characteristics. ASCII characters of any type. Contains as many strings as there are enumerated entries, entered as a cell array of strings.

Examples. Using the `cfield` object created in the link tutorial (run `ccstutorial` at the MATLAB prompt), you see the following when you display the object:

```
cfield

ENUM Object
  Symbol Name      : iz
  Address          : [ 40056 0]
  Wordsize        : 32 bits
  Address Units per value : 4 AU
  Representation   : signed
  Binary point position : 0
  Size            : [ 1 ]
  Total address units : 4 AU
  Array ordering   : row-major
  Endianness      : little
  Labels & values  : MATLAB=0, Simulink=1, SignalToolbox=2,
                   MATLABLink=3, EmbeddedTargetC6x=4
```

The labels are MATLAB, Simulink, SignalToolbox, MATLABLink, and EmbeddedTargetC6x. In this case, label is {1x5 cell}.

Referrals. See also property value.

link

Description. Specifies the link object you used when you created the embedded object.

Characteristics. A 1-by-1 array containing the name of the link object associated with the symbol table that holds the symbol.

Examples. In the tutorial, you created a numeric object named `uicvar`, using `cast` with the numeric object `cvar`. To create `cvar`, you used link object `cc` to determine the symbol table and project or target. When you view the properties of `uicvar`, you see the property `link` listing the link object as `ccsdsp`:

```
get(uicvar)
           address: [40060 0]
           bitsperstorageunit: 8
           numberofstorageunits: 4
           link: [1x1 ccsdsp]
           timeout: 10
           name: 'idat'
           wordsize: 16
           storageunitspvalue: 2
           size: 2
           endianness: 'little'
           arrayorder: 'row-major'
           prepad: 0
           postpad: 0
           represent: 'unsigned'
           binarypt: 0
```

Delving more deeply into the property `link` reveals the properties of the link object:

```
uicvar.link
```

```

CCSDSP Object:
  API version      : 1.2
  Processor type   : TMS320C6711
  Processor name   : CPU_1
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

```

Referrals. See also `createobj`.

member

Description. This identifies a MATLAB structure that holds the entry for each C member in the structure accessed by the object.

Characteristics. A MATLAB array containing

- Array type
- Array dimensions
- Data associated with this array
- If numeric, whether the variable is real or complex
- If a structure or object, the number of fields and field names

Examples. If you have a structure in DSP memory declared like the following structure

```

struct TAG_myStruct {
  int iy[2][3];
  myEnum iz;
} myStruct = { {{1,2,3},{4,-5,6}}, MatlabLink};

```

the member property of an object that accesses `myStruct`, might look like

```
get(cvar)
```

```
        name: 'myStruct'  
        member: [1x1 ccs.containerobj]  
        memname: {'iy' 'iz'}  
        memboffset: [0 24]  
        address: [40032 0]  
storageunitspervalue: 28  
        size: 1  
numberofstorageunits: 28  
        arrayorder: 'row-major'
```

where `member` is a 1-by-1 MATLAB array with a handle to the object that contains it named `ccs.containerobj`.

memname

Description. Contains the names of the fields in a structure or union accessed by a structure object.

Characteristics. `memname` is one or more strings providing the names of the structure fields, formatted as a cell array.

Range. Strings in `memname` contain any valid ASCII characters that might be found in a C structure field.

Examples. In CCS, if you had the following structure in your project code

```
struct tag {  
    int _a;  
    int B;  
    int b;  
} var;
```

you could create a structure object, `var`, that accesses the structure. Using `get` with `var` lets you review the names of the fields in the structure by looking at the `memname` property for `var`:

```
var = createobj(cc, 'var')  
get(var, 'memname')  
'a' 'B' 'b'
```


memoffset

Description. While this is not directly useful to you, the values in the vector specify how far, in memory in addressable units, each field in a structure is from the starting address for the structure.

Characteristics. Any numeric or alphanumeric value that represents a valid address or register location on the target. The vector contains one element for each field in the structure, representing the offset to that field in memory.

Range. A vector containing M element, where M is the number of fields in the structure. The second element in the vector is the offset to the second field in the structure, the third element in the vector is the offset to the third field, and so on until the final element is the offset to the final field. The first element in the memoffset vector is always 0, since this represents the offset to the first element in the structure, which is where the structure begins.

Examples. When you are working with structure objects, the property memoffset tells you how far one structure field is from another in memory:

```
cvar = createobj(cc,'myStruct')
```

```
STRUCTURE Object:
```

```
Symbol Name           : myStruct
Address               : [ 40032 0]
Address Units per value : 28 AU
Size                 : [ 1 ]
Total Address Units   : 28 AU
Array ordering        : row-major
Members              : 'iy', 'iz'
```

```
read(cvar)
```

```
ans =
```

```
    iy: [2x3 double]
    iz: 'MatlabLink'
get(cvar)
           name: 'myStruct'
           member: [1x1 ccs.containerobj]
```

```
        memname: {'iy' 'iz'}
        memboffset: [0 24]
        address: [40032 0]
storageunitspervalue: 28
        size: 1
numberofstorageunits: 28
        arrayorder: 'row-major'
```

From the property `memoffset`, you see that member `iz` of `myStruct` is 24 addresses from member `iy`, and from the start of the structure.

name

Description. Provides the name of the symbol or embedded object (mostly they are the same thing) to which the object refers. Contains the name of the function when the embedded object is a function.

Characteristics. ASCII character string that composes a valid C variable name.

Range. Any valid C variable name that occurs in your project.

numberofstorageunits

Description. Reports the number of smallest addressable units necessary to represent the symbol to which the object refers.

Characteristics. Reported in addressable units. Property `bitsperstorageunit` tells you how many bits are in each addressable unit — the smallest value supported by the processor. Combined with property `numberofstorageunits`, you can determine the storage used by the symbol.

Range. Any number of addressable units up to the limit of memory on the target.

numchannels

Description. Reports the number of RTDX communications channels configured for the RTDX link. Includes both read and write channels and does not depend on whether the channels are enabled.

Examples. As you did if you followed the RTDX tutorial in “Getting Started with RTDX” on page 1-38, create a link object, and then open two RTDX channels for the link:

```
cc=ccsdsp('boardnum',boardNum,'procnum',procNum)
```

```
CCSDSP Object:
```

```
API version      : 1.2
Processor type   : TMS320C6711
Processor name   : CPU_1
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs
```

```
RTDX channels    : 0
```

```
cc.rtdx.configure(1024,4);
```

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

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

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

```
get(cc,'rtdx')
```

```
RTDX Object:
```

```
Default timeout : 15.00 secs
Open channels    : 2
```

Ch Name	Mode
-- ----	----
1 ichan	write

2 ochan read

Where the listing for the RTDX object shows two open channels, this is the numChannels property value.

offset

Description. Specifies the starting position of the bitfield relative to bit 0 of the address. For A value of zero indicates that the bitfield begins at bit 0.

Characteristics. offset is an integer specifying a number of bits. The default value is zero.

outputvar

Description. An object created by Link for Code Composer Studio that represents the output argument from a function.

Characteristics. A handle to an object.

Range. A handle to any valid object in Link for Code Composer Studio.

page

Description. When you get the properties of an object, the address comes back in the format [address page]. In the address field for your object, page specifies which memory page contains the symbol address. For processors that do not use pages in memory, such as the C6701, the page value is always 0.

Characteristics. An integer that specifies the memory page for an address in memory.

Range. From 0 to the maximum number of memory pages supported by the processor.

Examples. Given a symbol in memory named ddat, after you create an object to access ddat, you can get the properties for the object and see the address format:

```
ddat=createobj(cc,'ddat')
```

```
NUMERIC Object
Symbol Name      : ddat
Address          : [ 40072 0]
Wordsize        : 64 bits
Address Units per value : 8 AU
Representation   : float
Binary point position : 0
Size            : [ 4 ]
Total address units : 32 AU
Array ordering   : row-major
Endianness      : little
```

Notice that the memory page value is 0 — the second value in the address field [40072 0] in the example. Since this example targets a C6701 digital signal processor, the page property value is always zero — the C6701 processor does not support memory pages.

postpad

Description. Reports the number of bits of padding required at the end of the memory buffer to fill the buffer. Determining the final numeric value stored in memory ignores the added bits.

Characteristics. Double-precision value that specifies the number of added bits.

prepad

Description. Reports the number of bits of padding required at the beginning of the memory buffer to fill the buffer. Determining the final numeric value stored in memory ignores the added bits.

Characteristics. Double-precision value that specifies the number of added bits.

procnum

Description. The number assigned by CCS to the processor on the board or simulator. When the board contains more than one processor, CCS assigns a number to each processor, numbering from 0 for the first processor on the first board. For example, when you have two recognized boards, and the second has two processors, the first processor on the first board is `procnum=0`, and the first and second processors on the second board are `procnum=1` and `procnum=2`. This is also a property used when you create a new link to CCS IDE.

Range. From 0 for one processor to $N-1$, where N is the number of processors that CCS recognizes as installed and configured on your machine.

Description. Contains the name of the register as used by the target. Note that this is not the same as a CPU register on the target.

Characteristics. `regname` is a MATLAB array with no initial value nor a default value.

Range. Any valid register used by your target.

represent

Description. Contains a string that specifies the data type for the accessed symbol. Memory locations consist of bits and bytes. The property value for `represent` specifies to MATLAB how to interpret the data stored in memory on the target.

You can change property `represent` to change the access format. For example, if an object has property `represent = float` and you change it to `represent = signed`

```
set(obj, 'represent', 'signed')
```

the data will be read as a signed integer. In addition, the data will be written as a signed integer.

Note Take care when you change the `represent` property of an object to `float`. Change this property only if the word size for the object is at least 32 bits.

For example, if the specified object is a 16-bit integer whose property `represent = signed`, `represent` cannot be changed to `float`. For the data to be accessed as a floating point number, it should be at least 32 bits in length.

Characteristics. A string that defines the data type for the variable — one of the following strings applies:

- `float` — IEEE floating point representation, either 32 or 64 bits
- `fract` — fractional fixed-point data
- `signed` — two's complement signed integers
- `ufract` — unsigned fractional fixed-point data
- `unsigned` — unsigned two's complement integer data

Range. While MATLAB recognizes many different data types, C and the TI processors are somewhat different. The tables provided here show the valid data types (from property `datatype`) and the strings that appear for them as the `represent` property value.

datatype Property String	represent Property Value
'double'	'float'
'single'	'float'
'int32'	'signed'
'int16'	'signed'
'int8'	'signed'
'uint32'	'unsigned'
'uint16'	'binary'
'uint8'	'binary'

datatype Property String	represent Property Value
'long double'	'float'
'float'	'float'
'long'	'signed'
'int'	'signed'
'char'	'signed'
'unsigned long'	'signed'
'unsigned int'	'unsigned'
'unsigned char'	'binary'
'Q0.15'	'fract'
'Q0.31'	'fract'

Various TI processors restrict the sizes of the data types used by objects in Link for Code Composer Studio. Shown in the next table, the processor families restrict the valid word sizes for the listed data types.

represent Property Value	C54 Processor Word Size Limits	C6x Processor Word Size Limits
'float'	32, 64 bits	32,64 bits
'signed'	16, 32 bits	8, 16, 32, 40, 64 bits
'unsigned'	16, 32 bits	8, 16, 32, 40, 64 bits
'binary'	16, 32 bits	8, 16, 32, 40,64 bits
'fract'	16, 32 bits	8, 16, 32, 40, 64 bits

Using the properties of the objects, you change the word size by changing the value of the `storageunitspervalue` property of the object. Note that you cannot change the `bitsperstorageunit` property value which depends on the processor and whether the object represents a memory location or a register.

Referrals. See also `cast`, `convert`.

rtdx

Description. Specifies whether the link object has RTDX channels included in the link. When the link has open RTDX channels, this property contains a structure of cell arrays that detail the information about the channels — the number of channels and the names of the channels.

Characteristics. Empty or an array of cell arrays containing strings and values.

Examples. When you create a link, the default state is not to have RTDX channels and the property `rtdx` is empty, as you see here:

```
cc=ccsdsp('boardnum',boardNum,'procnum',procNum)
```

```
CCSDSP Object:
```

```
API version      : 1.2
Processor type   : TMS320C6711
Processor name   : CPU_1
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs
```

```
RTDX channels    : 0
```

Now, configure and open two RTDX channels to the target:

```
cc.rtdx.configure(1024,4);
```

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

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

After creating the channels, displaying the link shows that the `rtdx` property is no longer empty. It contains the names and number of channels available, and the channel mode, either read or write:

```
get(cc)
      rtdx: [1x1 rtdx]
      apiversion: [1 2]
```

```
ccsappexe: 'D:\Applications\ti\cc\bin\  
boardnum: 0  
procnum: 0  
timeout: 10  
page: 0
```

```
get(cc, 'rtdx')
```

```
RTDX Object:  
Default timeout : 15.00 secs  
Open channels   : 2
```

Ch Name	Mode
1 ichan	write
2 ochan	read

Referrals. See also `ccdsp`, `enable`, `open`.

rtdxchannel

Description. Provides the names of open RTDX channels for the link.

Characteristics. Alphanumeric strings using ASCII characters that define the channel names.

Range. From 0 to the number of defined and open channels in your project.

size

Description. Defines the number of dimensions for the numeric array that is accessed by the numeric object. The `size` property provides the same information that function `size` provides in MATLAB.

Characteristics. `size` is a vector having as many elements as the number of dimensions in the symbol represented by the object. Each element in the vector reports the number of entries in that dimension.

Range. `size` can be a scalar greater than or equal to one, or a vector of integers, each greater than or equal to one.

Examples. When you have a variable declaration in your code like

```
int x[3] [2] = {(1,2), (3,4), (5,6)};
```

the `size` property tells you about `x` if you create an object that accesses `x`.

```
x = createobj(cc, 'x');
```

```
get(x, 'size')
```

```
ans =
```

```
[3 2]
```

so `x` represents a 3-by-2 array having six elements.

savedregisters

Description. Contains the list of registers whose contents are saved during function processing. The list of registers is different for each processor, and you can change the registers on the `savedregisters` list using `addregister` and `deleteregister`. Note that you cannot delete the default registers for a processor. You can delete only registers that you add.

Characteristics. An `mxAarray` that contains the names of all registers on the target that are preserved during processing.

Examples. For the C54x family of signal processors, the default saved registers are

AR1, AR6, AR7, and SP. Register SP is not required to be saved by the processor but Link for Code Composer Studio requires that the contents of SP be saved.

storageunitspervalue

Description. Describes how many storage units — addressable (AU) and register (RU) — make up the accessed symbol.

Characteristics. Given in addressable units (AU or RU), storageunitspervalue is an integer.

Range. storageunitspervalue is an integer equal to or greater than one, up to the limit of your target processor. This can have a value less than one in the case of packing of the bits in the symbol.

Examples. From the Function Call tutorial (“Tutorial — Using function Objects and Function Calls” on page 2-77, object cfield returns the following properties when you create an object to provide access to the myStruct member iz:

```
cfield = getmember(cvar,'iz') % Extract object from structure

ENUM Object
Symbol Name           : iz
Address               : [ 40056 0]
Wordsize              : 32 bits
Address Units per value : 4 AU
Representation        : signed
Binary point position : 0
Size                  : [ 1 ]
Total address units   : 4 AU
Array ordering         : row-major
Endianness            : little
Labels & values        : MATLAB=0, Simulink=1, SignalToolbox=2,
                        MatlabLink=3, EmbeddedTargetC6x=4

get(cfield)
      address: [40056 0]
      bitsperstorageunit: 8
      numberofstorageunits: 4
              link: [1x1 ccstdsp]
      timeout: 10
              name: 'iz'
```

```

        wordsize: 32
storageunitspervalue: 4
        size: 1
        endianness: 'little'
        arrayorder: 'row-major'
        prepad: 0
        postpad: 0
        represent: 'signed'
        binarypt: 0
        label: {1x5 cell}
        value: [0 1 2 3 4]

```

Requiring 4 addressable units (storage units) with 8 bits per storage unit (property `bitsperstorageunit = 8`) and a size of 1, `cfield` requires 32 bits of storage space in memory.

timeout

Description. Specifies how long Link for Code Composer Studio waits for an operation to complete, or at least to return a status of complete. In some cases, operations continue after the time-out expires, since the time period depends on the status of the operation, not the actual completion.

Characteristics. A value in seconds.

Range. A value greater than zero. 10 s is the default value. The time-out period for build is 1000 s.

Examples. In this example, the time-out period is 10 seconds for the new object:

```
cc=ccsdsp('boardnum',boardNum,'procnum',procNum)
```

CCSDSP Object:

```

API version      : 1.2
Processor type   : TMS320C6711
Processor name   : CPU_1
Running?        : No
Board number     : 0

```

```
Processor number : 0
Default timeout  : 10.00 secs

RTDX channels    : 0
```

type

Description. Specifies the return type for a function in your project.

Characteristics. A string that contains a valid return type, such as a data type or void.

typelist

Description. Lists the type entries in a type object. When you construct a `ccsdsp` object `cc`, it contains a type object `cc.type` with default entries `void`, `float`, `double`, `long`, `int`, `short`, and `char`. After you add your typedefs to the type object, `typelist` contains a listing of the types in the object.

Characteristics. An array of alphanumeric strings. The default entries in `typelist` are `void`, `float`, `double`, `long`, `int`, `short`, and `char`.

typename

Description. Lists the type names in a type object. When you construct a `ccsdsp` object `cc`, it contains a default type object `cc.type`. After you add your typedefs to the type object, `typename` contains a list of the names of the types in the object.

Characteristics. An `mxAarray` of alphanumeric strings.

Examples. Add a type definition to a `cc` object. You add your typedef to the type object that is part of the `ccsdsp` object:

```
cc=ccsdsp;
add(cc.type, 'mytypedef', 'uint32')
```

```
ans =
  type: 'uint32'
  size: 1
  uclass: 'numeric'
```

```
cc.type
```

```
Defined types : Void, Float, Double, Long, Int, Short, Char,
mytypedef
```

typestring

Description. Describes the data type of the referent for the pointer object accesses. `typestring` returns the data type for the referent as well as an asterisk to indicate that the symbol is a pointer.

Examples. For a pointer object that points to a floating-point symbol, the property value for `typestring` is `float *`. For a pointer to an integer, the value is `int *`.

value

Description. Reports the values associated with labels in an enumerated object.

Characteristics. Numbers, one or more, configured as a vector depending on the number of entries.

Examples. Using the enumerated data type variable `myEnum` from the link tutorial, create an object that accesses the labels and values for the enumerated data variable `iz`:

```
cvar = createobj(cc, 'myStruct')
```

```
STRUCTURE Object:
  Symbol Name      : myStruct
```

```
Address           : [ 40032 0]
Address Units per value : 28 AU
Size              : [ 1 ]
Total Address Units   : 28 AU
Array ordering       : row-major
Members             : 'iy', 'iz'
```

```
cfield = getmember(cvar,'iz')
```

```
ENUM Object
Symbol Name       : iz
Address           : [ 40056 0]
Wordsize          : 32 bits
Address Units per value : 4 AU
Representation     : signed
Binary point position : 0
Size              : [ 1 ]
Total address units : 4 AU
Array ordering     : row-major
Endianness         : little
Labels & values    : MATLAB=0, Simulink=1, SignalToolbox=2,
                   MatlabLink=3, EmbeddedTargetC6x=4
```

The values for `iz` are 0, 1, 2, 3, and 4. In the value property, the values show up as `[0 1 2 3 4]`, a vector whose elements are the values.

wordsize

Description. Specifies the word size for the target processor, and the referenced symbol.

Characteristics. Depends on the processor architecture. Because this is fixed on the processor, it is read only, set when you create an embedded object.

Range. For most processors, the word size can be from 8 to 64 bits, usually 8, 16, or 32.

Using FDATool

Introducing FDATool (p. 3-2)

Introduces the Filter Design and Analysis Tool

Guidelines on Exporting Filters from FDATool to CCS IDE (p. 3-3)

Things to think about when you export filters to Code Composer Studio

Tutorial — Exporting Filters from FDATool to CCS IDE (p. 3-10)

Takes you through the process of exporting a filter from FDATool to Code Composer Studio

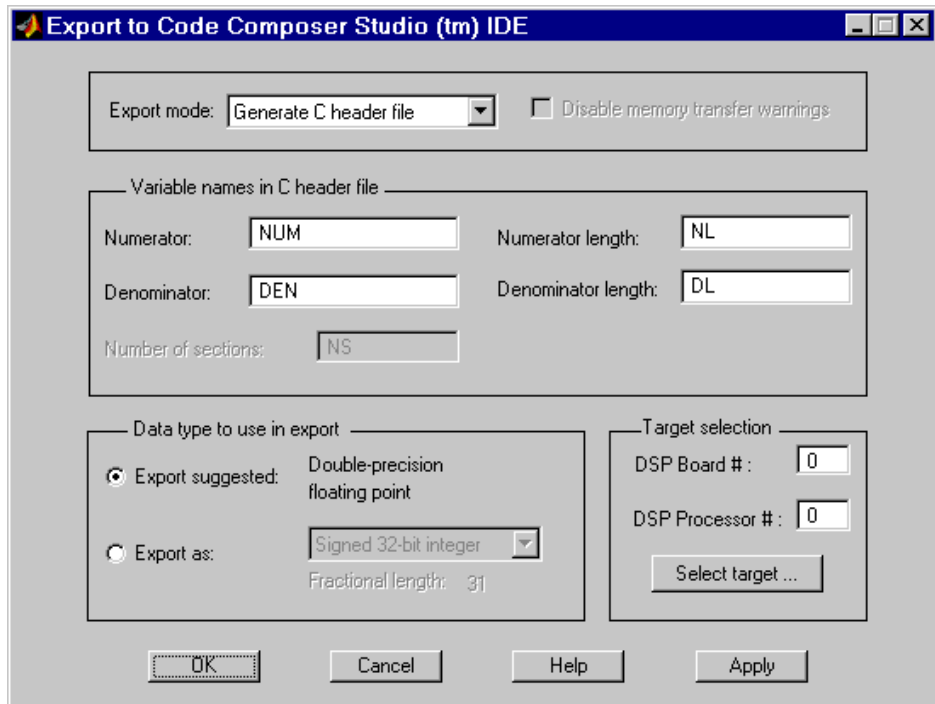
Introducing FDATool

The Filter Design and Analysis Tool (“FDATool: A Filter Design and Analysis GUI”) in the Signal Processing Toolbox is a powerful user interface for designing and analyzing filters. FDATool enables you to design digital FIR or IIR filters by setting filter performance specifications, by importing filters from your MATLAB workspace, or by directly specifying filter coefficients. FDATool also provides tools for analyzing filters, such as magnitude and phase response plots and pole-zero plots.

Once you have designed a filter in FDATool, you can export it to Code Composer Studio Integrated Development Environment (CCS IDE) to test the filter on a target digital signal processor (DSP). Using FDATool with CCS IDE takes filter design to the next level — enabling you to test your filter on a target DSP, tune and optimize the filter in FDATool, and test your redesigned filter on the target.

Guidelines on Exporting Filters from FDATool to CCS IDE

You can export filters from FDATool to CCS IDE by generating a C header file, or by writing the filter coefficients directly to target memory. To export a filter from FDATool to CCS IDE, use the Export to Code Composer Studio(tm) IDE dialog box (shortened to Export to CCS IDE dialog box in this section). Open the dialog box from the FDATool **Targets** menu.



For guidelines on exporting filters using the Export to CCS IDE dialog box, refer to the following sections:

- “Selecting the Export Mode” on page 3-4
- “Cautions Regarding Writing Directly to Memory” on page 3-5
- “Variables and Memory Necessary for Filter Export” on page 3-6
- “Selecting the Export Data Type” on page 3-8

For instructions on using the Export to CCS IDE dialog box, refer to “Tutorial — Exporting Filters from FDATool to CCS IDE” on page 3-10.

Selecting the Export Mode

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

Export Mode	When to Use	Suggested Use
Generate C header file	You have not yet allocated memory in your target DSP for the filter coefficients to export. (For a sample generated header file, see “Contents of the C Header File Generated in Task 1” on page 3-16.)	Create a program file from the generated C header file. Loading this program file into your target allocates static memory locations in the target, and exports your filter coefficients to these memory locations. You may want to edit the header file so that the program file allocates extra target memory, providing you more freedom to change your filter. See “Allocating Sufficient or Extra Memory for Filter Coefficients” on page 3-5 in the next section.
Write directly to memory	You have already allocated memory in your target DSP for the filter coefficients to export.	Tune your filter coefficients in FDATool, and write the updated filter coefficients directly to the allocated target memory. Refer to the next section “Cautions Regarding Writing Directly to Memory” on page 3-5.

Cautions Regarding Writing Directly to Memory

When you write filter coefficients directly to target memory, you need to allocate sufficient memory for the coefficients, and proceed with caution when you update your filter coefficients in target memory.

Allocating Sufficient or Extra Memory for Filter Coefficients

When you export filter coefficients directly to target memory, the filter coefficients overwrite as many memory locations as they need. The export process does not check whether you allocated sufficient memory for your filter coefficients. You must allocate enough memory for your filter coefficients or you may get unexpected results. To ensure you allocate enough target memory for your filter, export the filter by generating a C header file, as described in “Tutorial — Exporting Filters from FDATool to CCS IDE” on page 3-10.

You can allocate extra memory by editing the generated C header file, and then loading the associated program file into your target as described in the tutorial in “Step 8 — Export the Filter by Generating a Program File” on page 3-15. Allocating extra memory provides more freedom for changing a filter and overwriting its previous version stored in target memory. Even after you allocate extra memory, you should still proceed with caution when overwriting old filter coefficients with updated coefficients as discussed in the next section.

Overwriting Old Filter Coefficients with Updated Coefficients

When you tune a filter to overwrite its previous version in target memory, carefully consider changes that increase the memory required to store the filter coefficients, or that alter the export data type.

Do Not Tune a Filter’s Export Data Type. Never tune a filter by changing its data type, because the allocated memory expects the data type of the first version of the filter that you exported. Overwriting a filter with a filter that has a different data type usually yields unexpected results.

Be Wary of Filter Changes that Increase Memory Required to Store Filter Coefficients.

If you do not allocate extra memory when exporting the first version of your filter, do not tune the filter in ways that increase the memory required to store its coefficients. For instance, you should not increase the order of the filter. When you overwrite your original filter with one of a higher order, the updated filter may overwrite data in memory locations that you did not intend for storing filter coefficients. Even if you do allocate extra memory for your filter coefficients, be cautious about making changes that increase the memory required to store the coefficients. Examples of such changes include

- Changing an FIR filter to an IIR filter
- Increasing the filter order
- Increasing the number of filter sections

Variables and Memory Necessary for Filter Export

When you export a filter by generating a C header file, the header file stores the filter coefficients in filter coefficient variables. You must name these variables in the Export to CCS IDE dialog box. Variable names cannot be reserved words of the C programming language, such as `if`. By generating a program file from the C header file and loading the program file into your target, the filter coefficient variables in the header file appear in the target application symbol table.

When you export a filter by writing directly to target memory, the target stores the filter coefficients in memory locations. These memory locations correspond to filter coefficient variables in the target application symbol table. To export directly to target memory, you specify these variables in the Export to CCS IDE dialog box.

The necessary filter coefficient variables depend on the structure of your filter. The Export to CCS IDE dialog box provides you with the following parameters to specify or name the necessary filter coefficient variables. The dialog box activates only the parameters you need to set; the others become invisible or inactive.

Parameters for Specifying Filter Coefficient Variables	Description
Numerator	Numerator filter coefficients
Numerator length	Number of numerator filter coefficients
Denominator	Denominator filter coefficients
Denominator length	Number of denominator filter coefficients
Lattice coeffs	Lattice coefficients
Lattice coeffs length	Number of lattice coefficients
Ladder coeffs	Ladder coefficients
Ladder coeffs length	Number of ladder coefficients
Number of sections	Number of filter sections (parameter is inactive if your filter has only one section)

In the following table, x marks indicate the parameters to set for each filter structure.

Parameters for Naming Filter Coefficient Variables

Filter Structures	Numerator	Numerator length	Denominator	Denominator length	Lattice coeffs	Lattice coeffs length	Ladder coeffs	Ladder coeffs length	Number of sections
df1	x	x	x	x					x
df1t	x	x	x	x					x
df2	x	x	x	x					x
df2t	x	x	x	x					x
fir	x	x							x
firt	x	x							x
latticearma					x	x	x	x	x
latticeama					x	x			x

Selecting the Export Data Type

When you export a filter, the export process suggests the export data type that best preserves the performance of your filter. Use the suggested export data type by selecting **Export suggested** in the Export to CCS IDE dialog box. The data types that you can export are

- Signed integer (8-, 16-, or 32-bit)
- Unsigned integer (8-, 16-, or 32-bit)
- Double-precision floating point

- Single-precision floating point

Recommended Procedure for Selecting Export Data Type

By adhering to the following procedure when you set the export data type of your filter, the exported filter coefficients closely match the coefficients of the filter you designed in FDATool.

Step 1 — Set the Numerical Precision of Your Filter in FDATool. Set the numerical precision of your filter in FDATool by using the Quantized Filter pane, available when you install the Filter Design Toolbox. If you do not have the Filter Design Toolbox, your filters in FDATool have the default precision — double-precision floating point.

Step 2 — Select an Export Data Type in the Export to CCS IDE Dialog.

Use the export data type indicated by the **Export suggested** parameter in the Export to CCS IDE dialog box. Refer to the following note.

Though Step 2 insists you use the **Export suggested** parameter, you may find it useful to select the **Export as** option and select an export data type other than the one suggested. However, if you deviate from the suggested data type, the exported filter coefficients can be very different from the coefficients of the filter you designed in FDATool.

Note When you design your filter using an unsupported data type, the Export to Code Composer Studio(tm) IDE dialog box rounds the word length up to the next supported data type, and maintains the specified difference between the word length and fraction length. For example, for a filter with 14-bit word length and an 11-bit fraction length, the **Export suggested** parameter sets the export data type to a signed 16-bit integer with a 13-bit fraction length.

Tutorial — Exporting Filters from FDATool to CCS IDE

This tutorial shows you how to export filters from FDATool to CCS IDE with the **Export to Code Composer Studio IDE** dialog box. The tutorial covers exporting filters by generating C header files, and by writing filter coefficients directly to the target memory. Also see the previous section, “Guidelines on Exporting Filters from FDATool to CCS IDE” on page 3-3.

Descriptions of the Two Tutorial Tasks

“Task 1 — Export Filter by Generating a C Header File” on page 3-10 — You should complete this task before starting Task 2. Exporting a filter by generating a C header file not only exports your filter; it also ensures that you allocate enough target memory for the exported filter coefficients.

“Task 2 — Export Filter by Writing Directly to Target Memory” on page 3-17 — You should complete Task 1 before starting this task to ensure you allocate enough target memory for the filter coefficients to export. Exporting directly to target memory is useful when you want to repeatedly tune your filter in FDATool, and then export the updated filter coefficients directly to the allocated target memory.

Setting Up for the Tutorial

To complete this tutorial, you must install both the Signal Processing Toolbox and the Embedded Target for the TI TMS320C6000 DSP Platform (shortened to Embedded Target for TI C6000 DSP in this section). You do not need to open CCS IDE before starting the tutorial.

Task 1 — Export Filter by Generating a C Header File

In Task 1, you export a filter by generating a C header file. The generated C header file defines global arrays of filter coefficients that correspond to static memory locations in the final target program. By generating a program file from the C header file and loading the program file into your target, not only do you export your filter, but you ensure that you allocated enough memory for the exported filter coefficients. In Task 2, you write filter coefficients directly to the memory allocated in Task 1. You should complete Task 1 before starting Task 2.

Step 1 — Open FDATool

Open FDATool by entering `fdatool` in the MATLAB command window.

```
fdatool    % Open FDATool
```

FDATool opens with a default lowpass equiripple FIR filter displayed. The filter that you export in this tutorial is an IIR filter (to match the tutorial, design an IIR filter by changing **Design Method** from FIR to IIR.).

The screenshot displays the MATLAB Filter Design Tool (FDATool) interface. The top-left panel, 'Current Filter Information', shows the following details:

- Structure: Direct-Form II, Second-Order Sections
- Order: 31
- Sections: 16
- Stable: Yes
- Source: Designed

The top-right panel, 'Magnitude Response (dB)', shows a plot of Magnitude (dB) versus Frequency (kHz). The magnitude is 0 dB up to approximately 10 kHz, then rolls off to -2500 dB at 25 kHz.

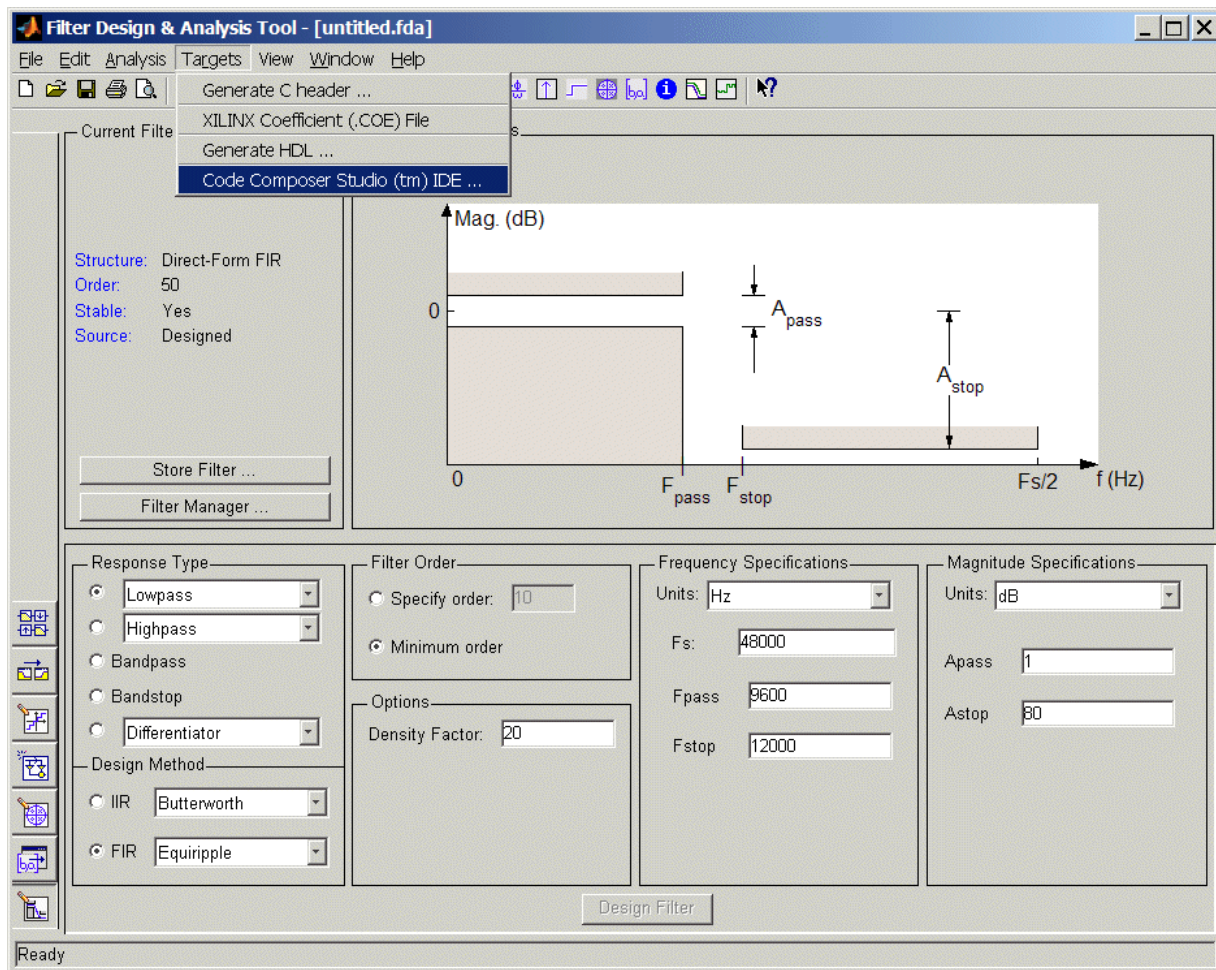
The bottom section contains design parameters:

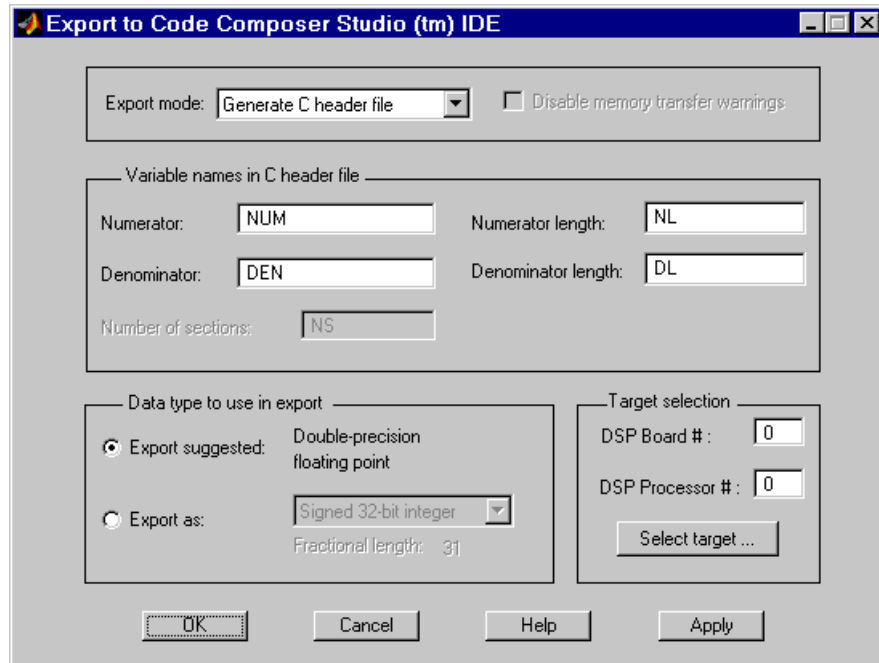
- Response Type:** Lowpass (selected)
- Filter Order:** Specify order: 10 (selected), Minimum order (selected)
- Options:** Match exactly: stopband (selected)
- Frequency Specifications:** Units: Hz (selected), Fs: 48000, Fpass: 9600, Fstop: 12000
- Magnitude Specifications:** Units: dB (selected), Apass: 1, Astop: 80
- Design Method:** IIR Butterworth (selected), FIR Equiripple (unselected)

The 'Design Filter' button is visible at the bottom center. The status bar at the bottom left shows 'Designing Filter ... Done'.

Step 2 – Open the Export to Code Composer Studio(™) IDE Dialog Box

Open the Export to Code Composer Studio(™) IDE dialog box by selecting **Targets > Code Composer Studio(™) IDE** from the FDATool menu bar.





Step 3 — Set the Export Mode

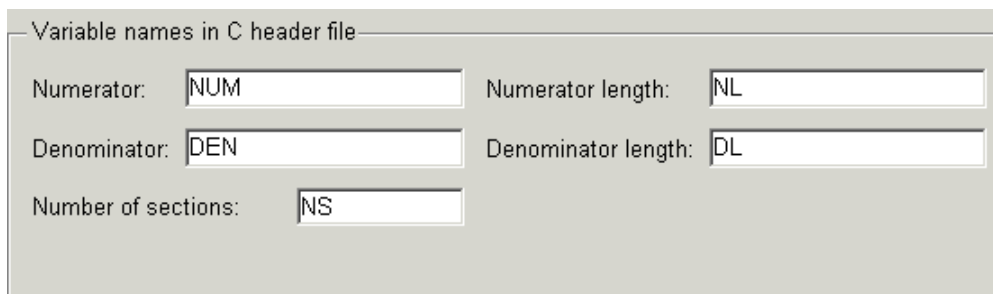
Set **Export mode** to Generate C header file.



Step 4 — Name the Filter Coefficient Variables

You must name the variables that store the filter coefficients in the generated C header file by setting the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters. (These correspond to the four variables for the numerator filter coefficients, denominator filter coefficients, number of numerator coefficients, and number of denominator coefficients.) For this tutorial, use the default variable names, NUM, DEN, NL, and DL. Use NS for the number of filter sections variable name, as provided by the dialog box.

The generated C header file will define global arrays, NUM, DEN, NL, and DL, that correspond to static memory locations containing the filter coefficients in the final target program.



Variable names in C header file

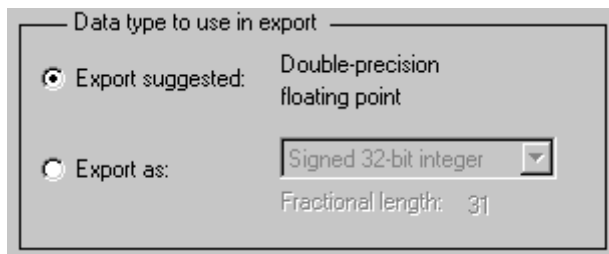
Numerator: Numerator length:

Denominator: Denominator length:

Number of sections:

Step 5 – Select a Data Type

Use the suggested data type to export your filter coefficients by selecting the **Export suggested** parameter.



Data type to use in export

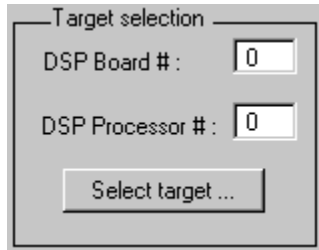
Export suggested: Double-precision floating point

Export as:

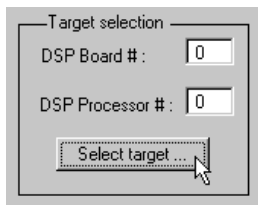
Fractional length: 31

Step 6 – Select a Target

If you know the board number and processor number of your target DSP, select your target by setting the **DSP Board #** and **DSP Processor #** values.



Alternatively, click **Select target...**, which opens the Selection Utility: Link for Code Composer(tm) dialog box. Select the board name and processor name of the DSP target and click **Done**. This automatically sets the **DSP Board #** and **DSP Processor #** values in the Export to Code Composer Studio(tm) IDE dialog box.



Step 7 — Generate the C Header File

Click **Generate** to generate the C header file. FDATool prompts you for a file name to save the generated header file with the .h extension, and a location to store the file. In addition, this opens the generated C header file in CCS IDE. (CCS IDE will be opened for you if you did not have it open.)

Step 8 — Export the Filter by Generating a Program File

Add the generated C header file to an appropriate project, generate a program file, and load the program file into your target DSP. The program file allocates static memory locations in the target, and writes the filter coefficients to these locations. Refer to the following note.

By completing steps 1 through 8, you allocated target memory for the filter coefficients and exported the coefficients to these memory locations. Now you can tune the filter in FDATool, then export the updated filter coefficients

directly to the allocated memory locations as described in Task 2 of this tutorial.

Note You may want to edit the generated C header file so the associated program file allocates extra target memory. This allows you to change your filter and export the new filter coefficients directly to the allocated memory without worrying about whether there is enough memory. For example, in the following header file, you could modify `const real64_T NUM[47] = {...}` to `real64_T NUM[256] = {...}` to allow NUM to store up to 256 numerator filter coefficients rather than 47.

Contents of the C Header File Generated in Task 1

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 *
 * Generated by MATLAB(R) 7.0.1 and the Signal Processing Toolbox 6.2.1.
 *
 * Generated on: 16-Sep-2004 14:57:57
 *
 */

/*
 * Discrete-Time FIR Filter (real)
 * -----
 * Filter Structure   : Direct-Form FIR
 * Filter Length     : 51
 * Stable             : Yes
 * Linear Phase      : Yes (Type 1)
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * M:\perfect\matlab\extern\include\tmwtypes.h
 */
const int BL = 51;
```



```

const real64_T B[51] = {
    -0.0009190982084683, -0.002717696026596, -0.002486952759832, 0.003661438383507,
    0.01365092523066, 0.01735116590109, 0.007665306190422, -0.006554718869642,
    -0.007696784037065, 0.006105459421394, 0.01387391574864, 0.0003508617282909,
    -0.01690892543669, -0.008905642749159, 0.01744112950085, 0.02074504452761,
    -0.01229649425194, -0.03424086590958, -0.001034529605572, 0.0477903055208,
    0.02736303791485, -0.05937951883105, -0.08230702592923, 0.06718690943287,
    0.3100151770903, 0.4300478803435, 0.3100151770903, 0.06718690943287,
    -0.08230702592923, -0.05937951883105, 0.02736303791485, 0.0477903055208,
    -0.001034529605572, -0.03424086590958, -0.01229649425194, 0.02074504452761,
    0.01744112950085, -0.008905642749159, -0.01690892543669, 0.0003508617282909,
    0.01387391574864, 0.006105459421394, -0.007696784037065, -0.006554718869642,
    0.007665306190422, 0.01735116590109, 0.01365092523066, 0.003661438383507,
    -0.002486952759832, -0.002717696026596, -0.0009190982084683
};

```

Task 2 — Export Filter by Writing Directly to Target Memory

In Task 2 you export a filter by writing the filter coefficients directly to target memory. Before starting this task, you must allocate enough target memory for the filter coefficients by completing “Task 1 — Export Filter by Generating a C Header File” on page 3-10. After you allocate enough target memory, you can tune your filter in FDATool and export the updated filter coefficients directly to the allocated target memory by following the steps in this task. For important guidelines on writing directly to target memory, refer to “Cautions Regarding Writing Directly to Memory” on page 3-5.

Step 9 — Tune Your Filter in FDATool


Tune your filter coefficients in FDATool to improve its performance. Set the numerical precision of your filter by using the Quantized Filter pane in FDATool, available when you install the Filter Design Toolbox. If you do not have the Filter Design Toolbox, your filters in FDATool have the default precision, double-precision floating point.

If you have the Export to Code Composer Studio(tm) IDE dialog box open from Task 1, the dialog box automatically updates itself as you tune the filter in FDATool. If you closed the dialog box, reopen it as described in “Step 2 — Open the Export to Code Composer Studio(tm) IDE Dialog Box” on page 3-12.

Note If you allocated exactly enough memory for the filter coefficients in Task 1, you should not tune your filter such that it requires more memory than did the original filter (by increasing the filter order, for example). If you need more memory for your updated filter, allocate extra memory by editing the generated C header file from Task 1 (as described in the previous note), generating a program file, and loading the program file into your target.

Step 10 – Set the Export Mode

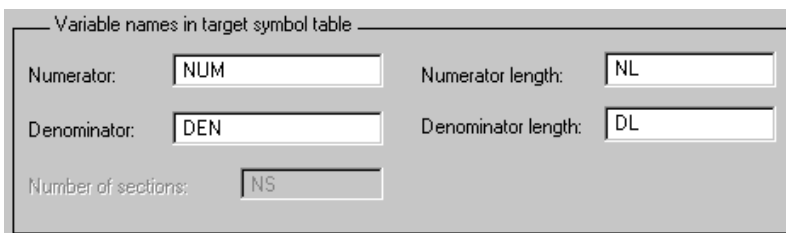
Set **Export mode** to **Write directly to memory**. Clear the parameter **Disable memory transfer warnings** so that you get a warning if your target does not support the export data type.



Export mode: Write directly to memory Disable memory transfer warnings

Step 11 – Input Filter Variable Names

To write to the memory allocated in Task 1, enter the names of the variables in the target symbol table corresponding to the allocated memory. These names are the same as the names of the filter coefficient variables in the C header file from Task 1: NUM, DEN, NL, and DL. You do not need to type these names in, since they are the default setting of the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters. (These parameters correspond to the memory locations that store the numerator filter coefficients, denominator filter coefficients, number of numerator coefficients, and number of denominator coefficients.)



Variable names in target symbol table

Numerator: NUM Numerator length: NL
Denominator: DEN Denominator length: DL
Number of sections: NS

Step 12 — Set All Other Parameters for Export as in Task 1

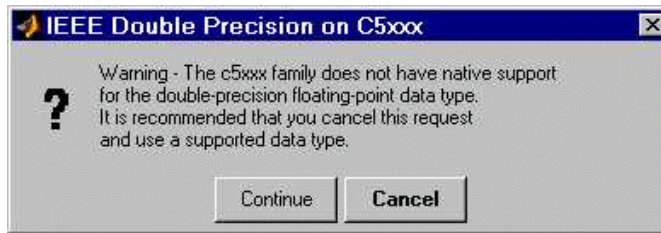
Select an export data type and indicate your target DSP as in Steps 5 and 6 of Task 1.

Step 13 — Load the Program File

Load the program file associated with your target into CCS IDE to activate the target symbol table. The program file must contain the global variables you entered in Step 11.

Step 14 — Export by Writing Directly to Target Memory

Click **Apply** to export your filter. Before the filter export begins, a warning dialog box appears if your target does not support the export data type. You can choose to continue to export the filter, or to cancel the export. To prevent this warning dialog box from appearing, select the parameter **Disable memory transfer warnings** in Step 10.



Step 15 — Continue Optimizing Filter Performance

Continue to optimize filter performance by retuning your filter in FDATool and exporting the updated filter coefficients directly to target memory. Since you already set up the export process to write to specific memory locations, you can click **Apply** to export updated coefficients to these same memory locations.

When the Export to Code Composer Studio (tm) IDE dialog box is open, it automatically updates as you tune your filter in FDATool, and preserves the parameter settings from Steps 10 through 13. The dialog box stays open as long as you do not click **Cancel** or **OK**. Keep the dialog box open when exporting multiple times to the same memory locations so you do not have to repeat Steps 10 through 13, and can just click **Apply**.

Where to Find More Information

For more information on exporting filters from FDATool to CCS IDE, refer to “Guidelines on Exporting Filters from FDATool to CCS IDE” on page 3-3, which contains the following sections:

- “Selecting the Export Mode” on page 3-4
- “Cautions Regarding Writing Directly to Memory” on page 3-5
- “Variables and Memory Necessary for Filter Export” on page 3-6
- “Selecting the Export Data Type” on page 3-8

To learn to use FDATool, refer to the section “Filter Design and Analysis Tool” in the Signal Processing Toolbox documentation.

Also refer to the reference pages for the following Link for Code Composer Studio functions:

- `address`
- `ccsdsp`
- `write`

Functions — By Category

Operations on Links for CCS IDE
(p. 4-2)

Work with links for CCS IDE

Operations on Links for RTDX
(p. 4-4)

Work with links to RTDX

Manipulate Data (p. 4-5)

Manipulate data on target from
MATLAB

Hardware-in-the-Loop Processing
(p. 4-5)

Work with hardware in loop

Operations on Links for CCS IDE

activate	Make specified project, file, or build configuration active in CCS IDE
add	Add files or new typedef to active project in CCS
address	Address and page for entry in symbol table in CCS IDE
animate	Run application on target processor to breakpoint
build	Build active project in CCS IDE
ccsboardinfo	Information about boards and simulators known to CCS IDE
ccsdsp	Create link to CCS IDE
cd	Change CCS IDE working directory
clear	Remove links to CCS IDE and RTDX interface, or clear type entries in type objects
close	Close CCS IDE files or RTDX channel
delete	Remove debug points in addresses or source files in CCS
dir	List files in current CCS IDE working directory
display	Display properties of link to CCS IDE or RTDX link
get	Access object properties
halt	Terminate execution of process running on target
info	Information about target processor
insert	Add debug point to source file or address in CCS

<code>isreadable</code>	Determine whether MATLAB can read specified memory block
<code>isrtdxcapable</code>	Determine whether target processor supports RTDX
<code>isrunning</code>	Determine whether target processor is executing process
<code>isvisible</code>	Determine whether CCS IDE is running
<code>iswritable</code>	Determine whether MATLAB can write to specified memory block
<code>load</code>	Transfer program file (*.out, *.obj) to target processor in active project
<code>new</code>	Create and open text file, project, or build configuration in CCS IDE
<code>open</code>	Open channel to target processor or load file into CCS IDE
<code>profile</code>	Profiling information from executing code that includes DSP/BIOS
<code>read</code>	Data from memory on target processor or in CCS
<code>regread</code>	Value from target processor register
<code>regwrite</code>	Write data values to registers on target processor
<code>reload</code>	Reload most recent program file to target signal processor
<code>remove</code>	Remove file from active CCS IDE project
<code>reset</code>	Reset target processor
<code>restart</code>	Restore program counter to entry point for current program
<code>run</code>	Execute program loaded on target processor

set	Set properties of links to CCS IDE and RTDX interface
symbol	Program symbol table from CCS IDE
visible	Set whether CCS IDE window is visible while CCS runs
write	Write data to memory on target processor

Operations on Links for RTDX

close	Close CCS IDE files or RTDX channel
configure	Define size and number of RTDX channel buffers
disable	Disable RTDX interface, specified channel, or all RTDX channels
display	Display properties of link to CCS IDE or RTDX link
enable	Enable RTDX interface, specified channel, or all RTDX channels
flush	Flush data or messages from specified RTDX channel(s)
get	Access object properties
isenabled	Determine whether RTDX link is enabled for communications
msgcount	Number of messages in read-enabled channel queue
open	Open channel to target processor or load file into CCS IDE
readmat	Matrix of data from RTDX channel

<code>readmsg</code>	Read messages from specified RTDX channel
<code>set</code>	Set properties of links to CCS IDE and RTDX interface
<code>writemsg</code>	Write messages to specified RTDX channel

Manipulate Data

<code>cast</code>	Change data type of object in Link for CCS
<code>createobj</code>	Create MATLAB objects representing embedded data or functions in program on target
<code>deref</code>	Object that accesses object pointed to by pointer object
<code>read</code>	Data from memory on target processor or in CCS
<code>write</code>	Write data to memory on target processor

Hardware-in-the-Loop Processing

<code>createobj</code>	Create MATLAB objects representing embedded data or functions in program on target
<code>declare</code>	Define C function declaration in MATLAB for CCS application
<code>getinput</code>	Specified input argument object from function object

<code>getoutput</code>	Access output from function object
<code>goto</code>	Position program counter to specified location in project code
<code>read</code>	Data from memory on target processor or in CCS
<code>resume</code>	Resume execution of stopped or paused function
<code>run</code>	Execute program loaded on target processor
<code>write</code>	Write data to memory on target processor

Functions — Alphabetical List

activate

Purpose Make specified project, file, or build configuration active in CCS IDE

Syntax `activate(cc, 'objectname', 'type')`

Description `activate(cc, 'objectname', 'type')` makes the object specified by `objectname` and `type` the active document window or project in CCS IDE. While you must include the link `cc`, it does not identify the project or file you make active. `activate` accepts one of three strings for `type`

String	Description
'project'	Makes an existing project in CCS IDE active (current). You must include the <code>.pjt</code> extension in <code>objectname</code> .
'text'	Makes the specified text file in CCS IDE the active document window. Include the file extension in <code>objectname</code> when you specify the file.
'buildcfg'	Makes the specified build configuration in CCS IDE active. Note that build configuration is similar to project configuration.

To specify the project file, text file, or build configuration, `objectname` must contain the full project name with the `.pjt` extension, or the full path name and extension for the text file.

When you activate a build configuration, `activate` applies to the active project in CCS IDE. If the build configuration you specify in `activate` does not exist in the active project, MATLAB returns an error that the specified configuration does not exist in the project. Fix this error by using `activate` to make the correct project active, then use `activate` again to select the desired build configuration.

Examples Create two projects in CCS IDE and use `activate` to change the active project, build configuration, and document window.

```
cc=ccsdsp;
```

```
visible(cc,1)
```

Now make two projects in CCS IDE.

```
new(cc,'myproject1.pjt','project')
new(cc,'myproject2.pjt')
```

In CCS IDE, `myproject2` is now the active project, since you just created it. With two projects in CCS IDE, add a new build configuration to the second project.

```
new(cc,'Testcfg','buildcfg')
```

If you switch to CCS IDE, you see `myproject2.pjt` in bold lettering in the project view, signaling it is the active project. When you check the active configuration list, you see three build configurations—Debug, Release, and Testcfg. Currently, Testcfg is the active build configuration in `myproject2`.

Finally, add a text file to `myproject1` and make it the active document window in CCS IDE. In this case, you add the source file for the ADC block.

```
activate(cc,'myproject1.pjt','project') % Makes myproject1 the active project.
add(cc,'c6711dsk_adc.c')
activate(cc,'c6711dsk_adc.c','text')
```

See Also

`build`, `new`, `remove`

add

Purpose Add files or new typedef to active project in CCS

Syntax

```
add(cc, 'filename')  
info = add(cc.type, 'typedefname', 'datatype')
```

Description Use add when you have an existing file to add to your active project in CCS. You can have more than one CCS IDE open at the same time, such as C5000 and C6000 instances. cc identifies which CCS IDE instance gets the file, and it identifies your board or target. Note that cc does not identify your project in CCS — it identifies only your target hardware or simulator. add puts the file specified by filename in the active project in CCS. Files you add must exist and be one of the supported file types shown in the next table.

When you add files, CCS puts the files in the appropriate folder in the project, such as putting source files with the .c extension in the Source folder and adding .lib files to the Libraries folder. You cannot change the destination folder in your CCS project. Using add is identical to selecting **Project > Add Files to Project...** in CCS IDE.

To specify the file to add, filename must be the full path name to the file, unless your file is in your CCS working directory or in a directory on your MATLAB path. The Link for Code Composer Studio searches for files first in your CCS IDE working directory, then in directories on your MATLAB path.

You can add the following file types to a project through add.

File Types and Extensions Supported by add and CCS IDE

File Type	Extensions Supported	CCS Project Folder
C/C++ source files	.c, .cpp, .cc, .ccx, .sa	Source
Assembly source files	.a*, .s* (excluding .sa, refer to C/C++ source files)	Source

File Type	Extensions Supported	CCS Project Folder
Object and library files	.o*, .lib	Libraries
Linker command file	.cmd	Project Name
DSP/BIOS file	.cdb*	DSP/BIOS Config
Visual Linker Recipe	rcp	Replaces the .cmd file, or goes under Project Name

Use `activate` to change your active project in CCS IDE or switch to the CCS IDE and change the active directory within CCS.

`info = add(cc.type, 'typedefname', 'datatype')` adds the new type definition `typedefname` to the type class in `cc`. Return value `info` contains the information about your custom data type. Your new data type `typedefname` has type `datatype`. As long as the `cc` object exists, the information about your new typedef exists as well. Including the left side argument is an option. Omitting the left side argument does not prevent `add` from making additions to the type objects.

Examples

Create a new project and to it add a source file and a build configuration. To do this task from MATLAB, use `new` to make your project in CCS IDE, then use `add` to put the required files into your new project.

```
cc=ccsdsp
```

```
CCSDSP Object:
```

```
API version      : 1.2
Processor type   : TMS320C64127
Processor name   : CPU_1
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs
```

add

```
RTDX channels    : 0

cc.visible(1) % Optional. Makes CCS IDE visible on your desktop.
new(cc,'myproject','project');

% Now add a C source file

add(cc,'c6711dsk_adc.c'); % Adds the source file for the ADC block.
```

In CCS IDE, `c6711dsk_adc.c` shows up in `myproject`, in the Source folder. Now add a new build configuration to `myproject`. After you add the new configuration, you can see it on the configurations list in CCS IDE, along with the usual Debug and Release configurations:

```
new(cc,'Testcfg','buildcfg')
```

Adding a new type definition to the type object is straightforward:

```
cc=ccsdsp;
info = add(cc.type, 'mynew typedef','int32');
info =

    type: 'int32'
    size: 1
    uclass: 'numeric'

cc.type

Defined types : Void, Float, Double, Long, Int, Short, Char, mynewtypedef
```

See Also

`activate`, `cd`, `open`, `remove`

Purpose Append registers to list of saved registers in property savedregs of function objects

Syntax `addregister(ff,regname)`
`addregister(ff,regnamelist)`

Description `addregister(ff,regname)` adds register `regname` to the list of registers that get preserved or reverted when a function is finished running. `ff` identifies the program function to which the register applies. You can add any register to the saved registers list.

Note `addregister` is the only way to add registers to the saved registers (`savedregs`) listing.

To remove a register from the list, use `deleteregister`.

When you issue the `createobj` call to create a handle to a function, the compiler creates the default list of saved registers. When you execute the function, the compiler saves the registers in the list, runs its process, and after completing its process, restores the saved registers to their initial state using the contents of the saved registers.

After a function generates a result, the execution process returns the saved registers to their initial states and values. When you add a register to the saved registers list, the added register is restored and saved with the other registers in the list.

For each processor family, the default list of saved registers changes, as shown in these sections. The default lists include registers that the compiler saves and that MATLAB requires for Link for Code Composer Studio to operate correctly.

Default Saved Registers for C28x Processors

AL, AH, AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7, XAR0, XAR1, XAR2, XAR3, XAR4,XAR5, XAR6, XAR7, SP, T, TL, PL, PH, DP

Default Saved Registers for C54x Processors

AR1, AR6, AR7, and SP (required by MATLAB, not the compiler)

Default Saved Registers for C55x Processors

T0, T1, T2, T3, TRN0, TRN1, AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7, XAR0, XAR1, XAR2, XAR3, XAR4, XAR5, XAR6, XAR7, AC0, AC1, AC2, AC3

Default Saved Registers for C62x and C67x Processors

A0, A2, A6, A7, A8, A9. Also B0, B1, B2, B4, B5, B6, B7, B8, B9. To support MATLAB requirements, B15 (the stack pointer) gets saved as well.

Registers A3, A4, A5, and B3 — your function must preserve these but they are not needed for reading function output.

Default Saved Registers for C64x Processors

A0, A2, A6, A7, A8, A9, A16, A17, A19, A19, A20, A21, A22, A23, A24, A25, A26, A27, A28, A29, A30, A31. Also B0, B1, B2, B4, B5, B6, B7, B8, B9, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31. To support MATLAB requirements, B15 (the stack pointer) gets saved as well.

Register B15 — not required by the compiler, but is required by MATLAB and is saved.

Registers A3, A4, and A5 — function must preserve these but they are needed for reading function output.

Default Saved Registers for R1x and R2x Processors

R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15

`addregister(ff,reglist)` appends the register names in `reglist` to the list of save-on-call (SOC) registers `ff.savedregs` that get preserved when a task is finished. `ff` identifies the function to which the register applies. `reglist` is a cell array that contains the names of registers on your processor that must be preserved during the changes that occur

during operation. You can also enter one register name in `reglist` as a string.

When you add entries to the existing SOC list, follow these recommendations:

- Add the entries before you execute any `write`, `run`, `goto`, or `execute` commands
- Add the entries immediately after you create the function object you are planning to use

These considerations ensure that the register values saved are the original register values.

See Also

`deleteregister`

address

Purpose Address and page for entry in symbol table in CCS IDE

Syntax `a = address(cc, 'symbolstring')`

Description `a = address(cc, 'symbolstring')` returns the address and page values for the symbol identified by 'symbolstring' in CCS IDE. `address` returns the symbol from the most recently loaded program in CCS IDE. In some instances this might not be the program loaded on the target to which `cc` is linked. By returning the address and page values as a structure, your programs can use the values directly. If you provide an output argument, the output `a` contains the 1-by-2 vector of [address page]. For `address` to work, `symbolstring` must represent a valid entry in the symbol table. To ensure that `address` returns information for the correct symbol, use the proper case when you enter `symbolstring` because symbol names are case-sensitive; 'symbolstring' is not the same as 'Symbolstring'.

If `address` does not find a symbol table entry that matches `symbolstring`, the first cell of `a` is returned empty. Notice that this function returns only the first matching symbol in the symbol table. The output argument is a cell array where each row in `a` presents the symbol name and address in the table. Each returned symbol address comprises a two element vector with the symbol page as the second element. For example, this table shows a few possible elements of `a`, and their interpretation.

a Array Element	Contents of the Specified Element
<code>a{1}</code>	String reflecting the symbol name. If <code>address</code> found a symbol that matches <code>symbolstring</code> , this is the same as <code>symbolstring</code> . Otherwise this is empty.
<code>a{2}(1)</code>	Address or value of symbol entry.
<code>a{2}(2)</code>	Memory page value. For TI C6xxx processors, the page is 0.

Examples

After you load a program to your target, `address` lets you read and write to specific entries in the symbol table for the program. For example, the following function reads the value of symbol `'ddat'` from the symbol table in CCS IDE.

```
ddatv = read(cc,address(cc,'ddat'),'double',4)
```

`ddat` is an entry in the current symbol table. `address` searches for the string `ddat` and returns a value when it finds a match. `read` returns `ddat` to MATLAB as a double-precision value as specified by the string `'double'`.

To change values in the symbol table, use `address` with `write`:

```
write(cc.address(cc,'ddat'),double([pi 12.3 exp(-1)...  
sin(pi/4)]))
```

After executing this write operation, `ddat` contains double-precision values for π , 12.3, e^{-1} , and $\sin(\pi/4)$. Use `read` to check the contents of `ddat`:

```
ddatv = read(cc,address(cc,'ddat'),'double',4)
```

MATLAB returns

```
ddatv =  
  
3.1416    12.3    0.3679    0.7071
```

See Also

`load`, `read`, `symbol`, `write`

animate

Purpose Run application on target processor to breakpoint

Syntax `animate(cc)`

Description `animate(cc)` starts the target application, which runs until it encounters a breakpoint in the code. At the breakpoint, application execution halts and CCS Debugger returns data to CCS IDE to update all windows that are not connected to probe points. After updating the display, the application resumes execution and runs until it encounters another breakpoint. The run-break-resume process continues until you stop the application from MATLAB with the `halt` function or from CCS IDE.

When you are running scripts or files in MATLAB, you might find that `animate` provides a useful way to update the CCS IDE with information as your script or program runs.

Using `animate` with multiprocessor boards

When you use `animate` with a target object `cc` that comprises more than one processor, such as an OMAP target, the method applies to each processor in your `cc` object. This causes each processor to run a loaded program just as it does for the single processor case.

See Also `halt`, `restart`, `run`

Purpose

Build active project in CCS IDE

Syntax

```
build(cc,timeout)
build(cc)
build(cc,'all',timeout)
build(cc,'all')
[result,numwarns]=build(...)
```

Description

`build(cc,timeout)` incrementally rebuilds your active project in CCS IDE. In an incremental build:

- Files that you have changed since your last project build process get rebuilt or recompiled.
- Source files rebuild when the time stamp on the source file is later than the time stamp on the object file created by the last build.
- Files whose time stamps have not changed do not rebuild or recompile.

This incremental build is identical to the incremental build in CCS IDE, available from the CCS IDE toolbar.

After building the files, CCS IDE relinks the files to create the program file with the `.out` extension. To determine whether to relink the output file, CCS IDE compares the time stamp on the output file to the time stamp on each object file. It relinks the output when an object file time stamp is later than the output file time stamp.

To reduce the compile and build time, CCS IDE keeps a build information file for each project. CCS IDE uses this file to determine which file needs to be rebuilt or relinked during the incremental build. After each build, CCS IDE updates the build information file.

Note CCS IDE opens a *Save As* dialog box when the requested project build overwrites any files in the project. You must respond to the dialog box before CCS IDE continues the build. The dialog box may not be visible when it opens and CCS IDE, MATLAB, and other applications can appear to be frozen until you respond to the dialog box. It may be hidden by open windows on your desktop.

To limit the time that `build` spends performing the build, the optional argument `timeout` stops the process after `timeout` seconds. `timeout` defines the number of seconds allowed to complete the required compile, build, and link operation. If the build process exceeds the time-out period, `build` returns an error in MATLAB. Generally, `build` causes the processor to initiate a restart even when the period specified by `timeout` passes. Exceeding the allotted time for the operation usually indicates that confirmation that the build was finished was not received before the time-out period passed. If you omit the `timeout` option in the syntax, `build` defaults to the global time-out defined in `cc`.

`build(cc)` is the same as `build(cc, timeout)` except that when you omit the `timeout` option, `build` defaults to the time-out for `build`, 1000 s. This time-out value overrides the default time-out setting for `cc`.

`build(cc, 'all', timeout)` completely rebuilds all of the files in the active project. This full build is identical to selecting **Project > Rebuild All** from the CCS menubar. After rebuilding all files in the project, `build` performs the link operation to create a new program file.

To limit the time that `build` spends performing the build, optional argument `timeout` stops the process after `timeout` seconds. `timeout` defines the number of seconds allowed to complete the required compile, build, and link operation.

If the build process exceeds the time-out period, `build` returns an error in MATLAB. Generally, `build` causes the processor to initiate a restart even when the period specified by `timeout` passes. Exceeding the allotted time for the operation usually indicates that confirmation that the build was finished was not received before the time-out period

passed. If you omit the `timeout` option in the syntax, `build` defaults to the global time-out defined in `cc`.

`build(cc, 'all')` is the same as `build(cc, 'all', timeout)` except that when you omit the `timeout` option, `build` defaults to the time-out set for `build` only, 1000 s.

`[result, numwarns]=build(...)` returns two output values that report the results of the build operation. For a successful build, the output arguments are the following:

- `result` equal to 1 for the build
- `numwarns` reports the number of build warnings that occurred during the build.

When the build is not successful, `build` displays an error and a message that contains the build string in the MATLAB Command Window.

Examples

To demonstrate building a project from MATLAB, use CCS IDE to load a project from the TI tutorials. For this example, open the project file `volume.pjt` from the Tutorial folder where you installed CCS IDE. (You can open any project you have for this example.)

Now use `build` to build the project:

```
cc=ccsdsp
```

```
CCSDSP Object:
```

```
API version      : 1.2
Processor type   : TMS320C64127
Processor name   : CPU_1
Running?        : No
Board number     : 0
Processor number : 0
Default timeout  : 10.00 secs
```

```
RTDX channels    : 0
```

build

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

You just completed a full build of the project in CCS IDE. On the Build pane in CCS IDE, you see the record of the build process and the results. Now, make a change to a file in the project in CCS IDE and save the file. Then rebuild the project with an incremental build.

```
build(cc, 20)
```

When you look at the Build pane in CCS IDE, the log shows that the build only occurred on the file or files that you changed and saved.

See Also

activate, isrunning, open

Purpose Change data type of object in Link for CCS

Syntax
`objname2 = cast(objname,datatype)`
`objname2 = cast(objname,datatype,size)`

Description `objname2 = cast(objname,datatype)` returns `objname2`, a copy of `objname`, whose `represent`, `storagepervalue`, and `wordsize` properties are changed so `objname2` supports the data type specified by `datatype`. Input argument `datatype` can be any supported data type. After the `cast` operation, read or write operations apply the appropriate data conversion to implement on your target the data type specified by the `represent`, `storagepervalue`, and `wordsize` properties of `objname2`.

The following data types work as input arguments to `cast`.

datatype String	represent Property Value
'double'	'float'
'single'	'float'
'int32'	'signed'
'int16'	'signed'
'int8'	'signed'
'uint32'	'unsigned'
'uint16'	'unsigned'
'uint8'	'unsigned'
'long double'	'float'
'float'	'float'
'long'	'signed'
'int'	'signed'
'char'	'signed'/'unsigned'
'unsigned long'	'signed'

datatype String	represent Property Value
'unsigned int'	'unsigned'
'unsigned char'	'unsigned'
'Q0.15'	'fract'
'Q0.31'	'fract'

Note pointer and rpointer objects respond differently when you use cast. Refer to “Using cast with pointer and rpointer Objects” on page 5-19 for more information about the supported data types for pointer or rpointer objects.

Various TI processors restrict the sizes of the data types used by objects in Link for Code Composer Studio. Shown in the next table, the processor families restrict the valid word sizes for the listed data types.

represent Property Value	C5x Processor Word Size Limits	C6x Processor Word Size Limits
'float'	32, 64 bits	32,64 bits
'signed'	16, 24, 32, 40, 48, 56, 64 bits	8, 16, 24, 32, 40, 48, 56, 64 bits
'unsigned'	16, 24, 32, 40, 48, 56, 64 bits	8, 16, 24, 32, 40, 48, 56, 64 bits
'binary'	16, 24, 32, 40, 48, 56, 64 bits	8, 16, 24, 32, 40, 48, 56, 64 bits
'fract'		

Using the properties of the objects, you change the word size by changing the value of the storageunitspervalue property of the object. Note that you cannot change the bitsperstorageunit property value,

which depends on the processor and whether the object represents a memory location or a register.

`cast` applies to any object that has the `represent`, `storagepvalue`, and `wordsize` properties. `function`, `ccsdsp`, and `rtdx` objects do not use the `represent` property and do not support `cast`.

A note — you could change the properties for `objname2` directly with `set` when you work with less common data types. Generally, we recommend you use `cast` to change the data type for an object, and consider `convert` as well.

`objname2 = cast(objname, datatype, size)` returns `objname2`, a copy of `objname`, with the specified data type for the `represent`, `storagepvalue`, and `wordsize` properties, and the `size` property value set to `size`. For bitfield objects, `size` is always 1.

Using cast with pointer and rpointer Objects

Working with pointer objects and register pointer (`rpointer`) objects is slightly different from using `cast` with numeric objects.

When you cast a pointer object, the results depend on the data type you specify to cast to in the syntax:

- When you specify a valid pointer type for your new pointer or `rpointer` object, `cast` creates the new pointer or `rpointer` object as a pointer type. Valid pointer data types are `datatype *` — you include the asterisk to indicate this is a pointer.
- When you specify a nonpointer data type for your new object, `cast` creates a new object that is no longer a pointer and does not access the referent that the original object accessed. Trying to cast to a nonpointer data type causes an error in MATLAB. Data types that do not support pointers are
 - All C native data types without the asterisk that indicates this is a pointer
 - `enum` (enumerated)
 - `string`

- struct

Examples

If your project includes the variables used in the three examples that follow, try them out to see `cast` at work. Without the specified variables, the examples do not run — read the examples to see the input and output from `cast`.

Cast the Data Type from int16 to Q0.31

After you create a `ccsdsp` object, use `cast` to recast a variable from data type `int16` to `Q0.31`.

Create the `int16` indirectly since you cannot create handles to `int16` data types in one step:

```
g_int16=createobj(cc,'g_float')
convert(g_int16,'int16')
cast(g_int16,'Q0.31')
```

Cast the Data Type from signed char to Q0.15

After you create a `ccsdsp` object, use `cast` to recast a variable from data type `signed char` to `Q0.15`.

Create the `unsigned char` from a `signed char` and cast from there to `Q0.15`:

```
g_uchar=createobj(cc,'g_schar')
cast(g_uchar,'Q0.15')
```

Cast the Data Type from double to uint32

After you create a `ccsdsp` object, use `cast` to recast a variable from data type `double` to `uint32`.

Create the `double` data type variable and cast it to a `uint32`:

```
g_double=createobj(cc,'double')
cast(g_double,'uint32')
```

See Also

`convert`

Purpose Information about boards and simulators known to CCS IDE

Syntax `ccsboardinfo`
`boards = ccsboardinfo`

Description `ccsboardinfo` returns configuration information about each board and processor installed and recognized by CCS. When you issue the function, `ccsboardinfo` returns the following information about each board or simulator.

Installed Board Configuration Data	Configuration Item Name	Description
Board number	boardnum	The number that CCS assigns to the board or simulator. Board numbering starts at 0 for the first board. This is also a property used when you create a new link to CCS IDE.
Board name	boardname	The name assigned to the board or simulator. Usually, the name is the board model name, such as TMS320C67xx evaluation module. If you are using a simulator, the name tells you which processor the simulator matches, such as C67xx simulator. If you renamed the board during setup, your assigned name appears here.

Installed Board Configuration Data	Configuration Item Name	Description
Processor number	procnum	The number assigned by CCS to the processor on the board or simulator. When the board contains more than one processor, CCS assigns a number to each processor, numbering from 0 for the first processor on the first board. For example, when you have two recognized boards, and the second has two processors, the first processor on the first board is procnum=0, and the first and second processors on the second board are procnum=1 and procnum=2. This is also a property used when you create a new link to CCS IDE.
Processor name	procname	Provides the name of the processor. Usually the name is CPU, unless you assign a different name.
Processor type	proctype	Gives the processor model, such as TMS320C6x1x for the C6xxx series processors.

Each row in the table that you see displayed represents one digital signal processor, either on a board or simulator. As a consequence, you use the information in the table in the function `ccsdsp` to target a selected board in your PC.

`boards = ccsboardinfo` returns the configuration information about your installed boards in a slightly different manner. Rather than returning the table containing the information, you get a listing of the board names and numbers, where each board has an associated structure named `proc` that contains the information about each processor on the board. For example

```
boards = ccsboardinfo
```


returns

```
boards =  
  
    name: 'C6xxx Simulator (Texas Instruments)'  
    number: 0  
    proc: [1x1 struct]
```

where the structure proc contains the processor information for the C6xxx simulator board:

```
boards.proc  
  
ans =  
  
    name: 'CPU'  
    number: 0  
    type: 'TMS320C6200'
```

Reviewing the output from both function syntaxes shows that the configuration information is the same.

When you combine this syntax with the dot notation used to access the elements in a structure, the result is a way to determine which board to connect to when you construct a link to CCS IDE. For example, when you are creating a link to a board in your PC, the dot notation provides the means to set the target board by issuing the command with the boardnum and procnum properties set to the entries in the structure boards. For example, when you enter

```
boards = ccsboardinfo;
```

`boards(1).name` returns the name of your second installed board and `boards(1).proc(2).name` returns the name of the second processor on the second board. To create a link to the second processor on the second board, use

```
cc = cc dsp('boardnum', boards(1).number, 'procnum', ...
```

```
boards(1).proc(2).name);
```

Examples

On a PC with both a simulator and a DSP Starter Kit (DSK) board installed,

```
ccsboardinfo
```

returns something similar to the following table. Your display may differ slightly based on what you called your boards when you configured them in CCS Setup Utility:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum ...	0	CPU	TMS320C6200
0	DSK (Texas Instruments)	0	CPU_3	TMS320C6x1x

When you have one or more boards that have multiple CPUs, `ccsboardinfo` returns the following table, or one similar to it:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	C6xxx Simulator (Texas Instrum ...	0	CPU	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	1	CPU_Primary	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	0	CPU_Secondary	TMS320C6200
0	C64xx Simulator (Texas Instru...	0	CPU	TMS320C64xx

In this example, board number 1 returns two defined CPUs: `CPU_Primary` and `CPU_Secondary`. Note that the `C6xxx` does not in fact have two CPUs; we defined a second CPU for this example.

To demonstrate the syntax `boards = ccsboardinfo`, this example assumes a PC with two boards installed, one of which has three CPUs.

Enter

```
ccsboardinfo
```

at the MATLAB prompt. You get

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum ...	0	CPU	TMS320C6211
0	C6211 DSK (Texas Instruments)	2	CPU_3	TMS320C6x1x
0	C6211 DSK (Texas Instruments)	1	CPU_4_1	TMS320C6x1x
0	C6211 DSK (Texas Instruments)	0	CPU_4_2	TMS320C6x1x

Now enter

```
boards = ccsboardinfo
```

MATLAB returns

```
boards=
2x1 struct array with fields
    name
    number
    proc
```

showing that you have two boards in your PC.

Use the dot notation to determine the names of the boards:

```
boards.name
```

returns

```
ans=
C6xxx Simulator (Texas Instruments)

ans=
C6211 DSK (Texas Instruments)
```

To identify the processors on each board, again use the dot notation to access the processor information. You have two boards (numbered 0 and

1). Board 0 has three CPUs defined for it. To determine the type of the second processor on board 0 (the board whose boardnum = 0), enter

```
boards(2).proc(1)
```

which returns

```
ans=  
  name: 'CPU_3'  
  number: 1  
  type: 'TMS320C6x1x'
```

Recall that

```
boards(2).proc
```

gives you this information about the board

```
ans=  
3x1 struct array with fields:  
  name  
  number  
  type
```

indicating that this board has three processors (the 3x1 array).

The dot notation is useful for accessing the contents of a structure when you create a link to CCS IDE. When you use `ccsdsp` to create your CCS link, you can use the dot notation to tell CCS IDE which processor you are targeting.

```
cc = ccsdsp('boardnum',boards(1).proc(1))
```

See Also

`info`, `ccsdsp`

Purpose

Create link to CCS IDE

Syntax

```
cc = ccsdsp
cc = ccsdsp('propertyname','propertyvalue',...)
```

Description

`cc = ccsdsp` returns a handle (or object or link) in `cc` that MATLAB uses to communicate with the default processor. In the case of no input arguments, `ccsdsp` constructs the object with default values for all properties. CCS IDE handles the communications between MATLAB and the target CPU. When you use the function, `ccsdsp` starts CCS IDE if it is not running. If `ccsdsp` opened an instance of the CCS IDE when you issued the `ccsdsp` function, CCS IDE becomes invisible after Link for Code Composer Studio creates the new object.

Note When `ccsdsp` creates the link `cc`, it sets the working directory for CCS IDE to be the same as your MATLAB working directory. This can have consequences when you create files or projects in CCS IDE, or save files and projects.

Each link to CCS IDE you create comprises two objects — a `ccsdsp` object and an `rtdx` object — that include the following properties.

Object	Property Name	Property	Default	Description
ccsdsp	'apiversion'	API version	N/A	Defines the API version used to create the link
	'proctype'	Processor Type	N/A	Specifies the kind of processor on the target board
	'procname'	Processor Name	CPU	Name given to the processor on the board to which this object links
	'status'	Running	No	Status of the program currently loaded on the processor
	'boardnum'	Board Number	0	Number that CCS assigns to the board. Used to identify the board
	'procnum'	Processor number	0	Number the CCS assigns to a processor on a board
	'timeout'	Default time-out	10.0 s	Specifies how long MATLAB waits for a response from CCS after issuing a request. This also applies when you try to construct a ccsdsp object. The create process waits for this time-out period for the connection to the target to complete. If the time-out period expires, you get an error message that the connection to the target failed and MATLAB could not create the ccsdsp object.

Object	Property Name	Property	Default	Description
rt dx	'timeout'	Time-out	10.0 s	Specifies how long CCS waits for a response from the processor after requesting data
	'numchannels'	Number of open channels	0	The number of open channels using this link
type	type	Defined types in the object	Void, Float, Double, Long, Int, Short, Char	List of the C types in the project cc accesses. Use add to include your C type definitions to the list

`cc = ccsdsp('propertyname','propertyvalue',...)` returns a handle in `cc` that MATLAB uses to communicate with the specified processor. CCS handles the communications between MATLAB and the target CPU.

MATLAB treats input parameters to `ccsdsp` as property definitions. Each property definition consists of a property name/property value pair.

Two properties of the `ccsdsp` handle are read only after you create the handle:

- 'boardnum' — the identifier for the installed board selected from the active boards recognized by CCS. If you have one board, use the default property value 0 to access the board.
- 'procnum' — the identifier for the processor on the board defined by `boardnum`. On boards with more than one processor, use this value to specify the target processor on the board. On boards with one processor, use the default property value 0 to specify the processor.

Given these two properties, the most common forms of the `ccsdsp` method are

```
cc = ccsdsp('boardnum',value)
cc = ccsdsp('boardnum',value,'procnum',value)
cc = ccsdsp(...,timeout)
```

which specify the target board, and processor in the second example, as the target.

The third example adds the `timeout` input argument to allow you to change how long MATLAB waits for the connection to the target or the response to a command to return completed.

You do not need to specify the `boardnum` and `procnum` properties when you have one board with one processor installed. The default property values refer correctly to the processor on the board.

Note Simulators count as boards. If you defined both boards and simulators in CCS IDE, specify the `boardnum` and `procnum` properties to connect to specific boards or simulators. Use `ccsboardinfo` to determine the values for the `boardnum` and `procnum` properties of your boards and simulators.

Because these properties are read only after you create the handle, you must set these property values as input arguments when you use `ccsdsp`. You cannot change these values after the handle exists. After you create the handle, use the `get` function to retrieve the `boardnum` and `procnum` property values.

Using `ccsdsp` with multiple processor targets

When you create `ccsdsp` objects that access targets that contain more than one processor, such as the OMAP1510 platform, `ccsdsp` behaves a little differently.

For each of the `ccsdsp` syntaxes above, the result of the method changes in the multiple processor case, as follows.


```

cc = ccsdsp
cc = ccsdsp('propertyname',propertyvalue)
cc = ccsdsp('propertyname',propertyvalue,'propertyname',...
propertyvalue)

```

In the case where you do not specify a board or processor:

```

cc = ccsdsp
Array of CCSDSP Objects:
  API version           : 1.2
  Board name           : OMAP 3.0 Platform Simulator [Texas
Instruments]
  Board number         : 0
  Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
  Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)

```

Where you choose to identify your target as an input argument to `ccsdsp`, for example, when your target board contains two processors:

```

cc = ccsdsp('boardnum',2)
Array of CCSDSP Objects:
  API version           : 1.2
  Board name           : OMAP 3.0 Platform Simulator [Texas Instruments]
  Board number         : 2
  Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
  Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)

```

`cc` returns a two element object handle with `cc(1)` corresponding to the first processor and `cc(2)` corresponding to the second.

You can include both the board number and the processor number in the `ccsdsp` syntax, as shown here:

```

cc = ccsdsp('boardnum',2,'procnum',[0 1])
Array of CCSDSP Objects:
  API version           : 1.2
  Board name           : OMAP 3.0 Platform Simulator [Texas
Instruments]

```

```
Board number           : 2
Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)
```

Enter `procnum` as either a single processor on the board (a single value in the input arguments to specify one processor) or a vector of processor numbers, as shown in the example, to select two or more processors.

Support Co-Emulation

Co-emulation, defined by Texas Instruments to mean simultaneous debugging of two or more CPUs, allows you to coordinate your debugging efforts between two or more processors within one device. Efficient development with OMAP™ hardware requires co-emulation support. Instead of creating one `cc` object when you issue the command

```
cc = ccsdsp
```

or your target that has multiple processors, the resulting `cc` object comprises a vector of `cc` objects `cc(1)`, `cc(2)`, and so on, each of which accesses one processor on your target device, say an OMAP1510. When your target has one processor, `cc` is a single object. With a multiprocessor target, the `cc` object returns the new vector of objects. For example, for board 2 with two processors,

```
cc = ccsdsp
```

returns the following information about the board and processors:

```
cc = ccsdsp('boardnum',2)
Array of CCSDSP Objects:
API version           : 1.2
Board name            : OMAP 3.0 Platform Simulator [Texas
Instruments]
Board number          : 2
Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)
```

Checking the existing boards shows that board 2 does have two processors:

```
ccsboardinfo
```

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	OMAP 3.0 Platform Simulator [T ...	0	MPU	TMS470R2x
2	OMAP 3.0 Platform Simulator [T ...	1	DSP	TMS320C550
1	MGS3 Simulator [Texas Instruments]	0	CPU	TMS320C5500
0	ARM925 Simulator [Texas Instru ...	0	CPU	TMS470R2x

Examples

On a system with three boards, where the third board has one processor and the first and second boards have two processors each, the function

```
cc = ccsdsp('boardnum',1,'procnum',0);
```

returns a handle to the first processor on the second board. Similarly, the function

```
cc = ccsdsp('boardnum',0,'procnum',1);
```

returns a handle to the second processor on the first board.

To access the processor on the third board, use

```
cc = ccsdsp('boardnum',2);
```

which sets the default property value `procnum=0` to connect to the processor on the third board.

When you use `get` with the object, MATLAB returns the properties for the object, and the property values.

```
cc = ccsdsp
CCSDSP Object:
  API version      : 1.2
  Processor type   : TMS320C6711
```

```
Processor name   : CPU_1
Running?        : No
Board number    : 1
Processor number : 0
Default timeout : 10.00 secs
```

```
RTDX channels   : 0
```

```
get(cc)
```

```
    rtdx: [1x1 rtdx]
  apiversion: [1 2]
  ccsappexe: 'D:\Applications\ti\cc\bin\'
  boardnum: 1
  procnum: 0
    type: [1x1 type]
  timeout: 10
  page: 0
```

```
cc.type    % Returns information about the type object
```

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

See Also

get, ccsboardinfo, createobj, set

Purpose

Change CCS IDE working directory

Syntax

```
cd(cc, 'directory')  
wd = cd(c, 'directory')  
cd(cc, pwd)
```

Description

`cd(cc, 'directory')` changes the CCS IDE working directory to the directory identified by the string `dir`. For the change to take effect, `dir` must refer to an existing directory. You can give the directory string either as a relative pathname or an absolute pathname including the drive letter. CCS IDE applies relative pathnames from the current working directory.

`wd = cd(c, 'directory')` returns the current CCS IDE working directory in `wd`.

Using `cc` to change the CCS IDE working directory does not affect your MATLAB working directory or any MATLAB paths. Use the following function syntax to set your CCS IDE working directory to match your MATLAB working directory.

`cd(cc, pwd)` where `pwd` calls the MATLAB function `pwd` that shows your present MATLAB working directory and changes your current CCS IDE working directory to match the pathname returned by `pwd`.

Examples

When you open a project in CCS IDE, the folder containing the project becomes the current working folder in CCS IDE. Try opening the tutorial project `volume.mak` in CCS IDE. `volume.mak` is in the tutorial files from CCS IDE. When you check the working directory for CCS IDE in MATLAB, you see something like the following result

```
wd=cd(cc)  
  
wd =  
  
D:\ticcs\c6000\tutorial\volume1
```

where the drive letter D may be different based on where you installed CCS IDE.

Now check your MATLAB working directory:

```
pwd  
  
ans =  
  
J:\bin\win32
```

Your CCS IDE and MATLAB working directories are not the same. To make the directories the same, use the `cd(cc,pwd)` syntax:

```
cd(cc,pwd) % Set CCS IDE to use your MATLAB working directory.  
pwd % Check your MATLAB working directory.  
  
ans =  
  
J:\bin\win32  
  
cd(cc) % Check your CCS IDE working directory.  
  
ans =  
  
J:\bin\win32
```

You have set CCS IDE and MATLAB to use the same working directory.

See Also

`dir`, `load`, `open`

Purpose Execute C or General Extension Language (GEL) expressions on target

Syntax

```
result = cexpr(cc, 'expression', timeout)
result = cexpr(cc, 'expression')
```

Description `result = cexpr(cc, 'expression', timeout)` executes the specified expression on the target processor referred to by `cc` and returns a result. If your program includes data in complex data structures and arrays, `cexpr` offers one way to access the data.

To run `cexpr` on your target, you must load a program to the processor. Your target processor does not need to be running the loaded program to execute `cexpr`. In operation `cexpr` is equivalent to using the *CCS Command Line* dialog box. Refer to your CCS documentation for more information about using the command line in CCS.

When you place single quotation marks around the `expression` argument, MATLAB ignores the enclosed string, passing it to your target. The target processor evaluates the expression and returns the result to MATLAB. Any part of the `expression` argument that is not in single quotation marks gets evaluated by MATLAB and sent to the target processor along with the quoted portion. Using single quotation marks, you can combine MATLAB, GEL, and C expressions within one `cexpr` command so that MATLAB sets a value on the target. The target uses the value and returns the result to your MATLAB workspace. Refer to “Examples” for a code example that mixes C and MATLAB functions in one command.

After you execute the function, MATLAB waits `timeout` seconds for CCS to confirm successful completion of the operation. If the wait exceeds `timeout` seconds, MATLAB returns an error. Often, the time-out error means the confirmation was delayed but the operation succeeded.

Enter `expression` as a string in single quotation marks defining either a C expression, a GEL command, or a combination of both C and GEL. CCS defines the syntax for `expression` as either

- A string with C syntax, whose variables reside in the local scope of the target processor

- A routine mapped to GEL functions defined in the current CCS project

`result = cexpr(cc, 'expression')` is the same as the preceding syntax except the timeout value defaults to the global time-out in `cc`. Use `get(cc)` to determine the global time-out value.

When you use `cexpr`, a few points can help you work effectively:

- `cexpr` returns a result in MATLAB when you use a C statement as the expression argument. In the first example syntax in “Examples.” `result = cexpr(cc, 'x.a')`, MATLAB returns `result =` the value of `x.a` on the target. In more concrete form, the syntax `result = cexpr(cc, 'x.b=10')` sets `x.b` to 10 on the target and returns `result = 10` to your MATLAB workspace.
- When your expression arguments are GEL functions, `cexpr` does not return results to MATLAB.
- Combining C and MATLAB expressions requires that you use single quotation marks around the C expressions to isolate them from the MATLAB interpreter. MATLAB performs the functions it understands and then passes the rest to the target for evaluation. The target returns the result to MATLAB.
- Pay attention to the scope of the program you are accessing. Only variables within the current scope of the program in CCS and on the target respond to `cexpr`. To access variables using `cexpr`, the variables must be either global or within the current scope. When you try to read or write to a variable outside the current scope, MATLAB returns errors like the following:

```
??? EvalC: identifier not found: variablename.  
??? EvalC: line(1), unexpected token: variablename.
```

Generally, variables within the program main are available without extra effort. To get to variables defined locally in subprograms, use breakpoints and the `runtohalt` input option in `run` to set your program to the right scope, then use `cexpr` to return the information.

For more information on GEL and GEL files, refer to your CCS documentation.

Examples

cexpr covers a broad range of uses. To introduce some of the possibilities, the following examples use both the C expression and GEL expression forms. Because executing the examples requires that specific variables and functions exist on the target, you cannot execute the code shown.

cexpr Syntax	Description
<pre>result = cexpr(cc, 'x.a')</pre>	<p>Returns the value of field <code>a</code> in structure <code>x</code> stored on your target. For this example, expression is <code>x.a</code> and <code>result</code> contains the value stored in <code>x.a</code> on the target.</p>
<pre>result = cexpr(cc, 'StartUp()')</pre>	<p>Executes the GEL function <code>StartUp</code> on the target processor. expression is <code>'StartUp'</code>, a function in the GEL file that loads each time you start CCS. Note that GEL function names are case sensitive — <code>StartUp</code> is not the same as <code>startup</code>. In this example, <code>result</code> is <code>NULL</code> or empty because GEL functions do not generate return values. Do not use an output argument with GEL expressions as input arguments.</p>

cexpr

cexpr Syntax	Description
<pre>result = cexpr(cc, 'x.b = 10')</pre>	Sets and returns the value of the field <code>b</code> in structure <code>x</code> . Here the assignment statement in single quotation marks replaces expression. <code>x.b</code> must be a structure in memory on your target and in the current program scope. After execution, <code>result</code> contains the value 10 returned from the target.
<pre>result =cexpr(cc,['x.c[2] =' int2str(z)])</pre>	Sets the value of <code>x.c[2]</code> to the string represented by integer <code>z</code> . In MATLAB, <code>result</code> contains the value stored in <code>x.c[2]</code> as returned from the target. Notice that the C expression is in single quotation marks, and the MATLAB <code>int2str</code> is not. Using single quotation marks directs MATLAB to ignore the C string that applies to the target processor and to evaluate <code>int2str</code> .

A note about the final example — the variable `z` must be in your MATLAB workspace for `int2str` to work. In contrast, `x.c[2]` defines a value on your target, not in MATLAB.

See Also

`address`, `read`, `write`

Purpose	Restore CCS to previous state before running function
Syntax	<code>cleanup(ff)</code>
Description	<code>cleanup(ff)</code> returns CCS to the state it was in before running or executing the function accessed by <code>ff</code> . After cleanup, the saved registers for your program are restored to their state before you ran <code>ff</code> . Using <code>cleanup</code> is entirely optional after <code>run</code> or <code>execute</code> .
See Also	<code>execute</code> , <code>run</code>

clear

Purpose Remove links to CCS IDE and RTDX interface, or clear type entries in type objects

Syntax

```
clear(cc)
clear('all')
clear(cc.type, 'all')
clear(cc.type, typedefname)
```

Description `clear(cc)` clears the link associated with `cc`. This is the last step in any development effort that uses links. Clear links that you no longer need for your work to avoid unforeseen problems. Calling `clear` executes the object destructors that delete the link object and all associated memory and resources.

`clear('all')` clears all existing links to CCS IDE and RTDX interface. This is the final step in any development process that uses links. Clear links that you no longer need for your work to avoid unforeseen problems. Calling `clear` with the `'all'` option executes the object destructors to delete all the link objects and all associated memory and resources.

Note If a link exists when you close CCS IDE, the application does not close. Microsoft Windows moves it to the background (it becomes invisible). Only after you clear all open links to CCS IDE, or close MATLAB, does closing CCS IDE actually close the application. You can check to see if CCS IDE is running by checking the Microsoft Windows Task Manager.

`clear(cc.type, 'all')` clears all user-defined type entries in the type object `obj`.

`clear(cc.type, typedefname)` clears the information on the specified user-defined type entry `typedefname` in the type object `obj`.

See Also `add`, `ccsdsp`, `close`, `disable`, `gettypeinfo`

Purpose

Close CCS IDE files or RTDX channel

Syntax

```
close(cc, 'filename', 'type')
close(rx, 'channel1', 'channel2', ...)
close(rx, 'channel')
```

Description

`close(cc, 'filename', 'type')` closes the file in CCS IDE identified by `filename` of type `'type'`. `type` identifies the type of file to close. This can be either project files when you use `'project'` for the `type` option, or text files when you use `'text'` for the `type` option. To close a specific file in CCS IDE, `filename` must match exactly the name of the file to close. If you replace `filename` with `'all'`, `close` terminates every open file whose `type` matches the `type` option. File types recognized by `close` include these extensions.

type String	Affected files
<code>'project'</code>	Project files with the <code>.pjt</code> extension.
<code>'text'</code>	All files with these extensions — <code>.a*</code> , <code>.c</code> , <code>.cc</code> , <code>.ccx</code> , <code>.cdb</code> , <code>.cmd</code> , <code>.cpp</code> , <code>.lib</code> , <code>.o*</code> , <code>.rcp</code> , and <code>.s*</code> . Note that <code>'text'</code> does not close <code>.cfg</code> files.

When you replace `filename` with the null entry `[]`, `close` shuts the current active file window in CCS IDE. When you specify `'project'` for the `type` option, it closes the active project.

Note `close` does not save files before shutting them. Closing files can result in lost data if you have changed the files since you last saved them. Use `save` to ensure that your changes are preserved before you close files that are open.

`close(rx, 'channel1', 'channel2', ...)` closes the channels specified by the strings `channel1`, `channel2`, and so on as defined in `rx`.

close

`close(rx, 'channel')` closes the specified channel. When you set channel to **'all'**, this function closes all the open channels associated with `rx`.

To avoid conflicts, do not name channels “all” or “ALL.”

Examples

Using close with Files and Projects

To clarify the different `close` options, here are six commands that close open files or projects in CCS IDE.

Command	Result
<code>close(cc, 'all', 'project')</code>	Close all open projects in CCS IDE.
<code>close(cc, 'my.pjt', 'project')</code>	Close the project <code>my.pjt</code> .
<code>close(cc, [], 'project')</code>	Close the active project.
<code>close(cc, 'all', 'text')</code>	Close all open text files. This includes source file, libraries, command files, and others.
<code>close(cc, 'my_source.cpp', 'text')</code>	Close the text file <code>my_source.cpp</code> .
<code>close(cc, [], 'text')</code>	Close the active file window.

Using close with RTDX

When you plan to use RTDX to communicate with a target, you open and enable channels to the board and processor. For example, to communicate with the processor on your installed board, you use `open` to set up a channel, as follows:

```
cc = cc dsp('boardnum',1,'procnum',0)
rx=cc.rtdx % Create an alias to the RTDX portion of this link.
open(rx,'ichan','w') % Open a channel for write access.
enable(rx,'ichan') % Enable the open channel for use.
```

After you finish using the open channel, you must close it to avoid difficulties later on.

```
close(rx, 'ichan')
```

Or to close all open channels, you could use

```
close(rx, 'all')
```

See Also

disable, open

configure

Purpose Define size and number of RTDX channel buffers

Syntax `configure(rx,length,num)`

Description `configure(rx,length,num)` sets the size of each main (host) buffer, and the number of buffers associated with `rx`. Input argument `length` is the size in bytes of each channel buffer and `num` is the number of channel buffers to create.

Main buffers must be at least 1024 bytes, with the maximum defined by the largest message. On 16-bit processors, the main buffer must be four bytes larger than the largest message. On 32-bit processors, set the buffer to be eight bytes larger than the largest message. By default, `configure` creates four, 1024-byte buffers. Independent of the value of `num`, CCS IDE allocates one buffer for each processor.

Use CCS to check the number of buffers and the length of each one.

Examples Create a default link to CCS and configure six main buffers of 4096 bytes each for the link.

```
cc=ccsdsp           % Create the CCS link with default values.
```

```
CCSDSP 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        % Create an alias to the rtdx portion.
```

```
RTDX channels    : 0
```



```
configure(rx,4096,6) % Use the alias rx to configure the length  
                    % and number of buffers.
```

After you configure the buffers, use the RTDX tools in CCS IDE to verify the buffers.

See Also

readmat, readmsg, write, writemsg

convert

Purpose Change object represent property from one data type to another

Syntax
`convert(objname,datatype)`
`convert(objname,datatype,size)`

Description `convert(objname,datatype)` returns objname with the represent property changed to the data type specified by datatype. Input argument datatype can be any supported data type. After you change the data type specified in represent, read or write operations apply the appropriate data conversion to implement on the target the data type specified by the represent property.

Note pointer and rpointer objects respond differently when you use convert. Refer to “Using convert with pointer and rpointer Objects” on page 5-50 for more information about the supported data types for pointer or rpointer objects and how convert behaves with different data types.

The following data types work as input arguments to convert.

datatype String	represent Property Value
'double'	'float'
'single'	'float'
'int32'	'signed'
'int16'	'signed'
'int8'	'signed'
'uint32'	'unsigned'
'uint16'	'unsigned'
'uint8'	'unsigned'
'long double'	'float'

datatype String	represent Property Value
'float'	'float'
'long'	'signed'
'int'	'signed'
'char'	'signed' / 'unsigned'
'unsigned long'	'signed'
'unsigned int'	'unsigned'
'unsigned char'	'unsigned'
'Q0.15'	'fract'
'Q0.31'	'fract'

Various TI processors restrict the sizes of the data types used by objects in Link for Code Composer Studio. Shown in the next table, the processor families restrict the valid word sizes for the listed data types.

Word size limits for supported processors

represent Property Value	C2x	C54	C55	C6x
'float'	32 bits	32 bits	32 bits	32, 64 bits
'signed'	16, 32 bits	16, 32 bits	16, 32, 40, 64 bits	8, 16, 32, 40, 64 bits
'unsigned'	16, 32 bits	16, 32 bits	16, 32, 40, 64 bits	8, 16, 32, 40, 64 bits
'fract'	16, 32bits	16, 32 bits	16, 32	16, 32 bits

Using the properties of the objects, you change the word size by changing the value of the `storageunitspvalue` property of the object. Note that you cannot change the `bitsperstorageunit` property value,

which depends on the processor and whether the object represents a memory location or a register.

Pointer objects, both data and numeric, usually use fewer than 32 bits, such as 22 or 23 bits, but are incorporated in 32-bit words.

`convert` applies to any object that has the `represent` property. `function`, `ccsdsp`, and `rtidx` objects do not use the `represent` property and do not support `convert`.

`convert(objname, datatype, size)` returns `objname` with the specified data type for the `represent` property, and the `size` property value set to `size`.

Using `convert` with pointer and `rpointer` Objects

Note `convert` does not support pointers to `void`, that is, pointers of the form `void *`. Before you convert a pointer to `void`, change the pointer to a valid data type, such as `int *` or `char *`.

When you convert a pointer object, the results depend on the data type you specify to convert to in the syntax:

- When you specify a valid pointer type for your converted pointer or `rpointer` object, `convert` changes the data type of the pointer and it remains a pointer.
- When you specify a nonpointer data type for your converted object, `convert` changes the `referent` or `regstring` properties of your pointer object, changing the data type of the referent (the value the pointer refers to) and your object is no longer a pointer. Therefore, use `convert` to change pointer or `rpointer` objects to nonpointer objects. You can convert to any data type, such as:
 - All C native data types without the asterisk that indicates this is a pointer
 - `enum` (enumerated)

- `string`
- `struct`

Working with pointer objects and register pointer (`rpointer`) objects is slightly different from using `convert` with numeric objects.

See Also

`cast`

copy

Purpose Copy object

Syntax `objname2 = copy(objname)`

Description `objname2 = copy(objname)` returns `objname2`, which is a copy of the input object specified by `objname`. All objects in the Link for Code Composer Studio support the `copy` function. Note that `objname2` is independent of the original; it is not an alias to the original `objname`. When you change a property of `objname2`, you are not changing the same property in `objname`.

See Also `createobj`

Purpose

Create MATLAB objects representing embedded data or functions in program on target

Syntax

```
objname = createobj(cc, 'symbolname');  
objname = createobj(cc, 'symbolname', 'option');  
objname = createobj(cc, 'functionname', 'function', 'funcdecl', ...  
    'function_declaration_string');  
objname = createobj(cc, functionname, 'function', 'allocate', ...  
    {'input', value1, 'input2', value2, ...});
```

Description

`objname = createobj(cc, 'symbolname')` makes an object in your MATLAB workspace named `objname`. Your new object contains information about the program symbol defined by `symbolname`. To use `createobj` successfully, you must have loaded a `.out` file to your target in CCS, and the symbol must be in the current symbol table in CCS.

`symbolname` can be any variable name or function name. By default, the embedded variable object returned accesses a variable within the current program scope.

Depending on the variable type and the storage used (register, memory, structure, function) for the variable, `createobj` generates an object that is one of the following kinds of objects:

- Memory object — access any symbol that resides in DSP memory
- Register object — access any symbols that reside in DSP registers
- Structure object — container class that accesses any symbol stored as a C struct type or C union type
- Function object — access any callable C function or assembly function that has a C prototype

Memory Objects

You do not create memory objects directly. Rather, you use `createobj` to make objects that are derived classes of memory objects:

- Numeric class objects — objects that access primitive data type variables, such as floats, ints, and shorts.

Numeric class objects also have derived classes:

- Pointer class objects — objects that access pointer data types (unsigned integers)
 - Enum class objects — objects that access enumerated data types (integers)
 - String class objects — objects that access string data types (characters)
- Bitfield class objects — objects that access bitfield data types

Register Objects

Like memory objects, you cannot instantiate a register object directly. Using `createobj`, you create a derived class object that accesses variables stored in registers on the processor.

- Rnumeric class objects — objects that access primitive data type variables, such as floats, ints, and shorts

Rnumeric class objects have derived classes just like numeric objects:

- Rpointer class objects — objects that access pointer data types (unsigned integers)
- Renum class objects — objects that access enumerated data types (integers)
- Rstring class objects — objects that access string data types (characters)

It should be clear that register objects differ from memory objects only in the kind of data storage they access — registers versus memory locations. Otherwise, many of the properties and methods of the two object classes are the same.

Structure Objects

Acting as a container class, structure objects hold either memory objects or register objects, as defined in the descriptions of both objects. Unlike memory or register objects, you create structure objects directly when you use `createobj` to access a C struct or C union data type variable.

Function Objects

When you create an object that accesses a C function in your program, `createobj` returns a function object, whose properties and methods provide information about and the ability to manipulate the target function. Your target can be any function in your code, whether a library function, a subprogram in your code, or a function you create from the MATLAB command line.

To create objects for local variables, the program counter (PC) must be located within the function that contains the local variable of interest. Note also the static variables for which you are creating objects must be within the current scope as well.

To increase the accuracy of the information about global symbols in your project, use `run`, as shown here, to position the PC to the start of main in your application in CCS.

```
run(cc, 'main')
```

Note that `symbolname` can be the name of a function in your target code. Thus, `symbolname` can refer to data or a function present on the target.

`symbolname` can be either a static variable or a global variable.

`objname = createobj(cc, 'symbolname', 'option')` lets you declare more information about `symbolname`, such as whether it represents a static or global variable. Use one of the following strings to declare the type for `symbolname` in `option`:

- `static` — declares that `symbolname` refers to a static variable in your code.

- `local` — declares the symbol to be a local variable in your code.
- `global` — declares that `symbolname` refers to a global variable in your code.
- `function` — declares that `symbolname` refers to a function in your code. Refer to the next syntax for more information about this optional keyword.

`objname = createobj(cc, 'functionname', 'function', 'funcdecl', ... 'function_declaration_string)` creates a function object `objname` that accesses the function defined by `function_declaration_string`. Use the optional keywords **function** and **funcdecl** to specify that you are creating a function object, and the declaration string follows. This syntax is required to create function objects that access library functions, unless you use `declare` with an existing function object to provide the function declaration to MATLAB.

Function Object Details

Working with function objects is more complicated than working with the other object classes. A number of limitations and considerations apply when you create objects that access functions in your project.

`createobj` works without modification for the following kinds of functions:

- Functions you write in C.
- Functions you write in assembly but for which you provide C prototypes. One example of this kind is library functions that you call from your C programs in your project.

Using `createobj` to construct an object that accesses a function of the kind listed causes MATLAB to search for the function declaration string in your project. When MATLAB finds the prototype, it uses the declaration to create the information it needs to be able to run the function from MATLAB, including

- Objects that access the input parameters for the function

- Objects that access the output parameter for the function
- Storage locations and addresses for the function

If MATLAB does not find the function, it creates the function object anyway, without the information it needs to run the function, and returns an error.

To respond to the error and provide MATLAB the information it needs, use `declare` to provide the declaration string to MATLAB.

You cannot create function objects for these kinds of functions:

- Assembly functions that do not have C prototypes
- Functions where the number of input arguments changes
- Functions that include non-ANSI C code

When you create a function object to access one of the above unsupported kinds, MATLAB returns an error that it could not find the function declaration.

Allocating Memory For Function Objects

To allocate memory buffers for function objects that you create, use

```
objname = createobj(cc,functionname,'function','allocate',...
    {'input',value1,'input2',value2,...});
```

which lets you set aside memory for each function input, called `input1`, `input2`, and so on in the syntax. `createobj` assigns `value1` and `value2` to `input1` and `input2`. **allocate** used here as a keyword specifies that this `createobj` syntax should perform memory allocation. So, to create memory buffers and assign values (12, 8, and 15) to three input variables for a function named `filter`, use the following syntax for `createobj`:

```
objname = createobj(cc,'filter','function','allocate',...
    {'input1',12,'input2',8,'input3',15});
```

Using Library Functions

Library functions present a special case of functions for Link for Code Composer Studio. `createobj` cannot find function declaration strings for library functions that you use in your project. While `createobj` does create the function object, it does not populate the function object with the information that enables MATLAB to run the target function. For library functions you must use `declare` to define explicitly the function declaration for objects that access library functions. Or, when you create the function object, use the syntax

```
objname = createobj(cc, 'functionname', 'function', 'funcdecl', ...  
    'function_declaration_string');
```

that passes the declaration string to MATLAB at creation time.

Examples

The following examples cover many situations you may encounter when you create function objects:

- Run a C function.
- Run a library function.
- Run a function that includes a custom data type.
- Run code generated by the Real-Time Workshop.
- Run a function that uses input vectors.

Unless you have project code that supports the functions used here you cannot run these examples. They are for inspection only.

These examples refer to four functions — `sin_taylor`, `dotprod`, `adotprod`, and `cdotprod`. Here is the code for `sin_taylor`.

```
/*-----*  
 * Taylor Series expansion of sin function - Fixed Point  
 * Limitations: input range: -pi <x <pi;  
 *  
 * Input Datatype is:
```

```

* Q2.13 (or MATLAB sfixed16_En13), scale factor = 2^13
* Output Datatype is:
* Q1.14 (or MATLAB sfixed16_En14), scale factor = 2^14
*
* Taylor Expansion of sin function (first 4 terms)
* sin(x) =(approx) x[1 - (x^2/6)*[1 + (x^2/20)*[ 1 - (x^2/42)]]]

*-----*/
#define SFIX32_EN26_VAL_1    67108864 // Integer equivalent of
1.0 in Q5.26
#define SFIX32_EN28_VAL_1    268435456 // Integer equivalent of
1.0 in Q3.28
#define SFIX32_EN30_VAL_1    1073741824 // Integer equivalent of
1.0 in Q1.30

short sin_taylor(short x)
{

// Define 16/32 bit local variables depending on processor
#if INT_MAX == 0x7FFFFFFF
int acc,a1,a2,a3,xpow;
#elif LONG_MAX == 0x7FFFFFFF
long acc,a1,a2,a3,xpow;
#endif

xpow = x*x; // x^2 sfixed32_En26

a1 = xpow/42; // x^2/42 sfixed32_En26
a2 = xpow/20; // x^2/20 sfixed32_En26
a3 = xpow/6; // x^2/6 sfixed32_En26

acc = SFIX32_EN26_VAL_1 - a1;
acc >>= 11;
acc *= (a2>>11);

acc = SFIX32_EN30_VAL_1 - acc;

```

```
    acc >>= 14;
    acc *= (a3>>14);

    acc = SFIX32_EN28_VAL_1 - acc;
    acc >>= 11;
    acc *= x;

    return (acc>>16);
}
```

Run a Standard C Function

In this example, we run function `sin_taylor` that computes the value for the sine of an input value. This function accepts one input, `x` (using data type `short`), and returns a `short`.

To get the correct values, the input data must be converted to Q16.13 format before passing to the function. After execution, the output value must be converted from Q16.14 to decimal representation.

Create a `ccsdsp` link:

```
cc = ccsdsp;
reset(cc);
pause(1); % Wait for hardware reset to complete before proceeding.
```

Run to start of main to ensure that your global variables are initialized:

```
run(cc, 'main', 1000);
```

Create a function object for `sin_taylor`:

```
ff = createobj(cc, 'sin_taylor')
inputdata = 0.5; % Input value to be used
```

Set value of input `x`:

```
x_obj = getinput(ff, 'x');
write(x_obj, inputdata* 2^13);
```

Run the function:

```
outputdata = run(ff);
```

Run a Library Function

For a library function, you pass the declaration string explicitly through `declare`.

This example runs the function `dotprod` that computes the dot product of two arrays. This function requires three inputs:

- `x` — a pointer to a vector of shorts
- `y` — a pointer to a vector of shorts
- `n` — the size of `x` and `y` vectors

We use the global variables `a` for input `x`, `b` for input `y`, and 4 for input `nx` (since `a` and `b` are four-element vectors). The function returns a short.

Create a `ccsdsp` link:

```
cc = ccsdsp;  
reset(cc);  
pause(1); % Wait for hardware reset to complete before proceeding
```

Run to start of `main` to ensure that you initialize the global variables:

```
run(cc, 'main', 1000);  
a_addr = address(cc, 'a'); % Global buffer for 'x'.  
b_addr = address(cc, 'b'); % Global buffer for 'y'.
```

Create the function object for the library function `dotprod`:

```
ff = createobj(cc, 'dotprod')
```

The previous step yields an incomplete function object `ff` because library functions always require that you provide the function declaration explicitly, as follows:

```
declare(ff,'decl','int dotprod (short *x, short *y, int nx)')
```

Set the value for the input parameter `x`:

```
x_obj = getinput(ff,'x');
write(x_obj,a_addr(1));
xRef_obj = deref(x_obj);
reshape(xRef_obj,4);
x_inputval = read(xRef_obj) % Verify 'y' referent value.
```

Set the value for `y`, the second input parameter:

```
y_obj = getinput(ff,'y');
write(y_obj,b_addr(1));
yRef_obj = deref(y_obj);
reshape(yRef_obj,4);
y_inputval = read(yRef_obj) % Verify 'y' referent value.
```

Pass the value for `nx` to the function:

```
nx_obj = getinput(ff,'nx');
write(nx_obj,4);
nx_inputval = read(nx_obj) % Verify 'nx' value.
```

Now run the function:

```
run(ff);
```

Run a Function That Has a Typedef in the Prototype

Having custom data types in your function declaration can cause problems when you run the functions from MATLAB.

Case 1 — Running a Function That Has a Typedef in the Function Prototype

This example runs the function `cdotprod` that computes the dot product of two matrices. This function requires three inputs:

- `x` — a pointer to a vector of shorts
- `y` — a pointer to a vector of shorts
- `n` — the size of `x` and `y` vectors

Both `n` and the return argument are defined as data type `INT`, a custom data type defined in the source code.

We use the global variables `a` for input `x`, `b` for input `y`, and `4` for input `n` (since `a` and `b` are four-element vectors). The function returns a short.

Create a `ccsdsp` link:

```
cc = ccsdsp;  
reset(cc);  
pause(1); % Wait for hardware reset to complete before proceeding
```

Run to start of `main` to ensure that `CCS` initializes all of the global variables:

```
run(cc, 'main', 1000);  
a_addr = address(cc, 'a'); % Global buffer for x.  
b_addr = address(cc, 'b'); % Global buffer for y.
```

Create a function object for the library function `cdotprod`:

```
ff = createobj(cc, 'cdotprod')
```

The previous call to `createobj` yields an incomplete function object because the function declaration includes an unresolved typedef — the type `INT`. To resolve this error, add the custom data type `INT` to the type object and use `declare` to pass the function declaration to `MATLAB`:

```
add(cc.type,'INT','int'); % A warning mentions that data type
                        % INT cannot be resolved.
declare(ff,'decl','INT cdotprod (short x[], short y[], INT n)')
```

Set values for the inputs *x*, *y*, and *n*, and run the function, passing the input values in the run syntax. Input *x* is a pointer so pass an address. Input *y* is a pointer as well, so pass another address. Input *n* is an integer that specifies the size of *x* and *y*:

```
run(ff,'x',a_addr(1),'y',b_addr(1),'n',4);
```

Case 2 – A Second Approach to Solving the Typedef Problem

We are going to run the function `cdotprod`, which computes the dot product of two matrices. This function accepts three inputs:

- *x* — a pointer to a vector of shorts
- *y* — a pointer to a vector of shorts
- *n* — the size of *x* and *y* vectors

We use the global variable *a* for input *x*, *b* for input *y*, and 4 for input *n* (since *a* and *b* are four-element vectors). The function returns a short.

Create `ccsdsp` link:

```
cc = ccsdsp;
reset(cc);
Pause(1); % Wait for hardware reset to complete before proceeding.
```

Run to start of main to ensure that CCS initializes all of the global variables:

```
run(cc,'main',1000);
a_addr = address(cc,'a'); % Global buffer for 'x'.
b_addr = address(cc,'b'); % Global buffer for 'y'.
```

Create function object for library function `cdotprod`:

```
ff = createobj(cc, 'cdotprod')
```

Again `createobj` generates an incomplete function object because of the unresolved data type `INT` in the function declaration. In this case, fix the problem by adding the custom data type `INT` to the type object and create the object `ff` again, instead of using `declare` to pass the function declaration to MATLAB:

```
add(cc.type, 'INT', 'int'); % Warning mentioned that data type
                           % INT cannot be resolved.
ff = createobj(cc, 'cdotprod')
```

Set values for the inputs `x`, `y`, and `n`, and run the function, passing the input values in the run syntax. Input `x` is a pointer so pass an address. Input `y` is a pointer as well, so pass another address. Input `n` is an integer that specifies the size of `x` and `y`:

```
run(ff, 'x', a_addr(1), 'y', b_addr(1), 'n', 4);
```

Case 3 — A Third Approach to Solving the Typedef Problem

Once more we are going to run the function `cdotprod`, which computes the dot product of two matrices. This function accepts three inputs:

- `x` — a pointer to a vector of shorts
- `y` — a pointer to a vector of shorts
- `n` — the size of `x` and `y` vectors

We use the global variable `a` for input `x`, `b` for input `y`, and 4 for input `n` (since `a` and `b` are four-element vectors). `cdotprod` returns a short.

Create `ccsdsp` link:

```
cc = ccsdsp;
reset(cc);
pause(1); % Wait for hardware reset to complete before proceeding.
```

Run to start of main to ensure that CCS initializes all of the global variables:

```
run(cc,'main',1000);  
a_addr = address(cc,'a'); % Global buffer for x.  
b_addr = address(cc,'b'); % Global buffer for y.
```

Create a function object for the library function `cdotprod`:

```
ff = createobj(cc,'cdotprod')
```

This attempt to create a new function object `ff` results in an incomplete function object because MATLAB could not resolve the data type `INT` in the function declaration. In this approach to overcoming the unresolved type error, use `declare` to pass to MATLAB a version of the `cdotprod` function declaration that does not include the offending type `INT` — you do not need to add the `typedef` to the type object:

```
declare(ff,'decl','int cdotprod (short x[], short y[], short n)')
```

Notice that the data types for the return argument and for `n` now specify `int`, Set values for the inputs `x`, `y`, and `n`, and run the function, passing the input values in the run syntax. Input `x` is a pointer so pass an address. Input `y` is a pointer as well, so pass another address. Input `n` is an integer that specifies the size of `x` and `y`:

```
run(ff,'x',a_addr(1),'y',b_addr(1),'n',4);
```

Run a Function Generated by Real-Time Workshop

We are going to run the function `'mwdsp_fir_df_dd'` which applies a filter to a noisy input signal. This function accepts nine input parameters and returns the filtered signal in the input argument `y`.

Create a `ccsdsp` link:

```
cc = ccsdsp;  
reset(cc);  
pause(1); % Wait for hardware reset to complete before proceeding.
```

Now run the Real-Time Workshop generated code from the beginning to MdlOutputs. You run from program start until MdlOutputs to ensure that all of the code configuration processes get done:

```
run(cc,'runtofunc',MdlOutputs);
```

After running to MdlOutputs, you create the function object — pass the function declaration to avoid MATLAB returning an error when you create the function object. Due to the complexity of this function declaration, we have assigned the string to a variable decl. We use the variable in the createobj syntax.

```
decl = ['MWDSP_IDECL void MWDSP_FIR_DF_DD(const real_T *u,...  
real_T *y, real_T * const mem_base,int_T *mem_offset,...  
const int_T numDelays, const int_T sampsPerChan,...  
const int_T numChans, const real_T * const b,...  
const boolean_T one_fpf)'];  
ff = createobj(cc,'MWDSP_FIR_DF_DD','function','funcdecl',decl);
```

Examine the function declaration above. This declaration causes MATLAB to fail to create the fully populated function object ff because of the MWDSP_IDECL macro at the beginning of the string. MATLAB cannot recognize this string. Since the information in MWDSP_IDECL is not relevant to creating the function object, you can remove this from the declaration string:

```
decl = ['void MWDSP_FIR_DF_DD(const real_T *u,...  
real_T *y, real_T * const mem_base,int_T *mem_offset,...  
const int_T numDelays, const int_T sampsPerChan,...  
const int_T numChans, const real_T * const b,...  
const boolean_T one_fpf)'];  
ff = createobj(cc,'MWDSP_FIR_DF_DD','function','funcdecl',decl);
```

Now function object ff has all the information MATLAB needs.

Note You may not always be able to remove offending entries in a declaration string, as we did with the macro `MWDSP_IDECL`. Often you can try your declaration and see if it works. If not, use `add` to include typedefs in the type object when MATLAB complains about a data type, or try removing the problem portion of the declaration string if the function does not require the troublesome text.

With the function object in your MATLAB workspace, create objects for the inputs to `MWDSP_FIR_DF_DD`:

Create an object for `rtB`:

```
rtBobj = createobj(cc, 'rtB');
```

Get the relevant `rtB` member objects:

```
SumObj = getmember(rtBobj, 'Sum');  
% Store Output of MWDSP_FIR_DF_DD in FilObj  
FilObj = getmember(rtBobj, 'Digital_Lowpass_Fil');
```

Next, create an object for `rtDWork`:

```
rtDWorkObj = createobj(cc, 'rtDWork');
```

and get the relevant member objects:

```
Fil_FILT_STATES = getmember(rtDWorkObj, ...  
    'Digital_Lowpass_Fil_FILT_STATES');  
DF_INDX = getmember(rtDWorkObj, ...  
    'Digital_Lowpass_Fil_FILT_STATES');
```

Create one last object for `filterCoeffs`:

```
filterCoeffsObj = createobj(cc, 'filterCoeffs');
```

To run the function, you need to provide the input values:

```

u = SumObj.address(1); % Input 1.
y = FilObj.address(1); % Input 2.
mem_base = Fil_FILT_STATES.address(1); % Input 3.
mem_offset = DF_INDX.address(1); % Input 4.
numDelays = 65; % Input 5.
sampsPerChan = 256; % Input 6.
numChans = 1; % Input 7.
b = filterCoeffsObj.address(1); % Input 8.
one_fpf = 1; % Input 9.

```

Run the function, providing the input argument values in input value/input name pairs, such as 3,membase and 6,sampPerChan:

```

run(ff,1,u,2,y,3,mem_base,4,mem_offset,5,numDelays,6,...
sampsPerChan,7,numChans,8,b,9,one_fpf)

```

Run a Function That Has Vector Inputs

This example shows how to run a function that accepts vector inputs.

We are going to run the function `adotprod` that computes the dot product of two matrices. `adotprod` accepts two inputs,

- `x` — a four-element vector of shorts
- `y` — a four-element vector of shorts

The compiler converts the vector inputs into pointers to the vectors. We use the global variable `a` for input `x` and `b` for input `y`. The function returns a short.

Create a `ccsdsp` link:

```

cc = ccsdsp;
reset(cc);
pause(1); % Wait for hardware reset to complete before proceeding.

```

Run to start of `main` to ensure that CCS initializes all of the global variables:

createobj

```
run(cc,'main',1000);
a_addr = address(cc,'a'); % Global buffer for 'x'.
b_addr = address(cc,'b'); % Global buffer for 'y'.
```

Create a function object ff to access adotprod:

```
ff = createobj(cc,'adotprod')
```

The function prototype for adotprod is

```
int adotprod(short x[4], short y[4])
```

adotprod requires as input two vector arrays *x* and *y*. The compiler requires that you pass the addresses of *x*[4] and *y*[4], not the actual vectors *x* and *y*. So instead of writing a data vector to input object *x_obj* and *y_obj*, you provide the addresses of existing four-element vectors:

```
display('INPUT VALUE 'x':')
x_obj = getinput(ff,'x') % Note that this is a pointer to a vector
                        % of shorts
display('INPUT VALUE 'y':')
y_obj = getinput(ff,'y') % Note that this is a pointer to a vector
                        % of shorts
```

Set value of inputs *x* and *y* and run the function. Pass addresses to *x* and *y* since both are pointers to other data:

```
write(x_obj,a_addr(1))
write(y_obj,b_addr(1))
x_inputval = read(reshape(deref(x_obj),4));
y_inputval = read(reshape(deref(y_obj),4));
```

Using the following commands to write data to *x* and *y* does not give you the expected result — the compiler cannot determine where to put array [1:4]:

```
write(x_obj,[1:4]);
write(y_obj,[1:4]);
```


Now run your function:

```
run(ff);
```

The preceding examples present a few of the wide variety of functions and conditions you may encounter when you construct function objects.

See Also

copy, ccscdsp, declare

datatypemanager

Purpose Open Data Type Manager (DTM) to identify custom data types to MATLAB from a CCS project

Syntax `datatypemanager(cc)`
`cc2 = datatypemanager(cc)`

Description `datatypemanager(cc)` opens the Data Type Manager (DTM) with data type information about the project to which `cc` refers. With the type manager open, you can add type definitions (typedefs) from your project to MATLAB so it can interpret them. You add your typedefs because MATLAB cannot determine or understand typedefs in your function prototypes remotely across the interface to Code Composer Studio.

Each custom type definition in your prototype must appear on the **Typedef name (Equivalent data type)** list before you can use the typedef from MATLAB with a function object.

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, it appears added to this list.
- **Add typedef** — opens the **Add Typedef** dialog box so you can add one or more typedefs to your project. Your added typedef appears on the **Typedef name (Equivalent data type)** list. Also, when you pass the `cc` object to the DTM, and then add a typedef, the command

```
cc.type
```

returns a list of the data types in the 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 **Typedef 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 an M-file you can reload into the DTM later.

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

In the stored M-file, you find a function that includes the add and remove operations you used to create the list of data types in the DTM. For each time you added a typedef in the DTM, the M-file contains an add command that adds the new type definition to the `cc.type` property of the object. When you remove a data type, you see an equivalent `clear` command that removes a data type from the `cc.type` object.

Note An interesting note — all of your operations that add and remove data types in the DTM during a session are stored in the generated M-file that you save. Saving the operations has the effect of storing any mistakes you make while creating or removing type definitions. One consequence of storing mistakes is that when you load your saved session into the DTM, you see the same error messages you saw when you created the data types in the session. You might find this a little disconcerting.

The first line of the M-file is a function definition, where the name of the function is the filename of the session you saved.

`cc2 = datatypemanager(cc)` returns the `cc2` `ccsdsp` object while it opens the DTM. `cc2` represents an alias to `cc`. Objects `cc` and `cc2` are not independent objects. When you change a property of either `cc` or `cc2`, the corresponding property in the other object changes as well.

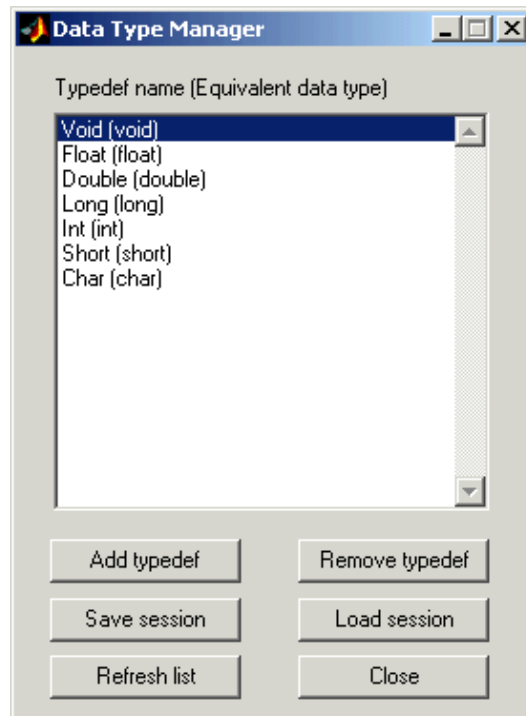
Data Type Manager

When you create objects that access functions in a project, MATLAB can recognize most data types that you use in your project. However, if the functions use one or more custom type definitions, MATLAB cannot recognize the data type and cannot work with the function. To overcome this problem, the Data Type Manager provides the capability to define your typedefs to MATLAB.

Entering

```
datatypemanager(cc)
```

at the MATLAB prompt opens the DTM.



Before you add a type definition, the **Typedef name (Equivalent data type)** list shows a number of data types already defined:

- Void(void) — void return argument for a function
- Float(float) — float data type used in a function input or return argument
- Double(double) — double data type used in a function input or return argument
- Long(long) — long data type used in a function input or return argument
- Int(int) — int data type used in a function input or return argument

datatypemanager

- Short (short) — short data type used in a function input or return argument
- Char (char) — character data type used in a function input or return argument

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

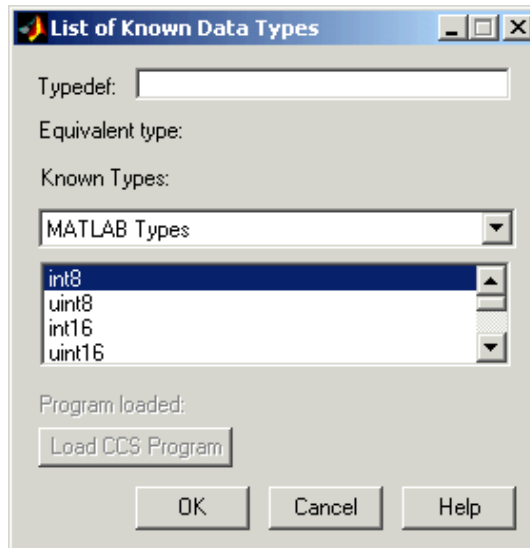
Although not recommended, you can use mixed case typedef names, so long as the equivalent data type uses lowercase. In particular, typedefs that refer to other typedefs should resolve to a data type in lowercase.

Adding a type definition adds the new data type to the list of typedefs.

Remove any existing or new type definitions with the **Remove typedef** option.

Add Typedef Dialog Box

Clicking **Add typedef** in the DTM opens the List of Known Data Types dialog box. As shown in this figure, you add your custom type definitions here.

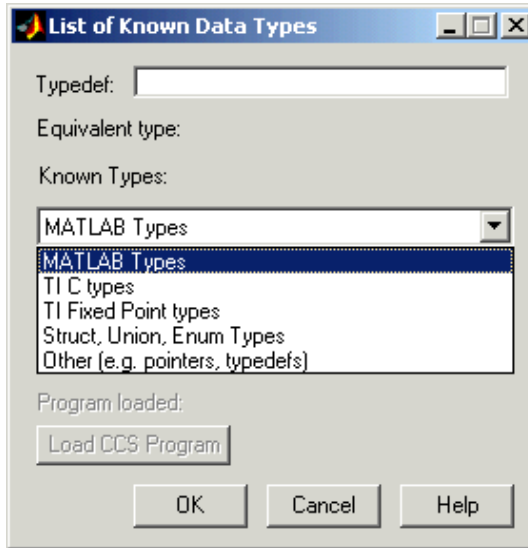


When you have used custom type definitions in your program or project, you must specify what they mean to MATLAB. The **Typedef** option lets you enter the name of the typedef in your program and select an equivalent type from the **Known Types** list. By defining your type definitions in this dialog box, you enable MATLAB can understand and work with them when you return the data to the MATLAB workspace or send data from the workspace to your project.

After you define each typedef, the **Equivalent type** option shows you the type you specified for each type definition, either when you enter it in the **Typedef** field or select it from the **Known Types** list.

datatypemanager

Options in this dialog box let you review the data types you are using or that are available in your projects. By selecting different data type categories from the **Known Types** list, you can see all of the supported data types.



From the list of known data types, choose one of the following data type categories:

- MATLAB Types

Data Type	Description
int8	8-bit integer data
uint8	unsigned 8-bit integer data
int16	16-bit integer data
uint16	unsigned 16-bit integer data
int32	32-bit integer data

Data Type	Description
uint32	unsigned 32-bit integer data
int64	64-bit integer data
uint64	unsigned 64-bit integer data
single	32-bit IEEE floating-point data
double	64-bit IEEE floating-point data

- TI C Types

Data Type	Description (For C6000 Compiler)
char	8-bit character data with a sign bit
unsigned char	8-bit character data
signed char	8-bit character data
short	16-bit numeric data
unsigned short	unsigned 16-bit numeric data
signed short	16-bit numeric data with sign designation
int	32-bit integer numeric data
unsigned int	32-bit integer numerics without sign information
signed int	32-bit integer numerics with sign information
long	40-bit data with sign bit. Note that this is not the same as int.
unsigned long	40-bit data without information about the sign of the number
signed long	40-bit data without information about the sign of the number represented
float	32-bit numeric data

Data Type	Description (For C6000 Compiler)
double	64-bit numeric data
long double	On the C2xxx and C5xxx – 32-bit IEEE floating-point data On the C6xxx – 64-bit IEEE floating-point data

Numbers of bits change depending on the processor and compiler. For more information about TI data types and specific processors or compilers, refer to your compiler documentation from TI.

- TI Fixed Point Types

Data Type	Description
Q0.15	Numeric data with 16-bit word length and 15-bit fraction length
Q0.31	32-bit word length numeric data with fraction length of 31 bits

- Struct, Union, Enum types

If the program you load on the processor includes one or more of struct, union, or enum data types, the type definitions show up on this list. Until you load a program on the processor, this list is empty and trying to access the list generates an error message.

Load a program, if you have not already done so, by clicking **Load CCS Program** and selecting a .out file to load on your processor.

- When the load process works, 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 you already loaded on your target do not appear in the **Loaded program** option. MATLAB cannot determine what program you have loaded.

- Others such as pointers and typedefs

Like `struct`, `union`, and `enum` data types, the `Others` list is empty until you define one or more typedefs. Unlike the `Struct`, `Union`, `Enum` types list, loading a program does not populate this list with typedefs from the program. You must define them explicitly in this dialog box.

Custom type definitions can refer to other typedefs in your project. Nesting typedefs works once you have defined the necessary custom types. To create a typedef that uses another typedef, define the nested (inner) definition, and then define the outer definition as a pointer to the nested definition. Refer to “Examples” on page 5-2 to see this in operation.

Program loaded — tells you the name of the program loaded on the processor, if you loaded the program from this dialog box. If not, **Program loaded** does not report the program name.

Load CCS Program — opens the **Load Program** dialog box so you can select and load a `.out` file to your processor.

Examples

This set of examples, show how to create custom type definitions with the DTM. Each example shows the **List of Known Data Types** dialog box with the selections or entries needed to create the typedef.

Start the examples by creating a `ccsdsp` object:

```
cc=ccsdsp;
```

Now start the DTM with the `cc` object. So far you have not loaded a file on the target.

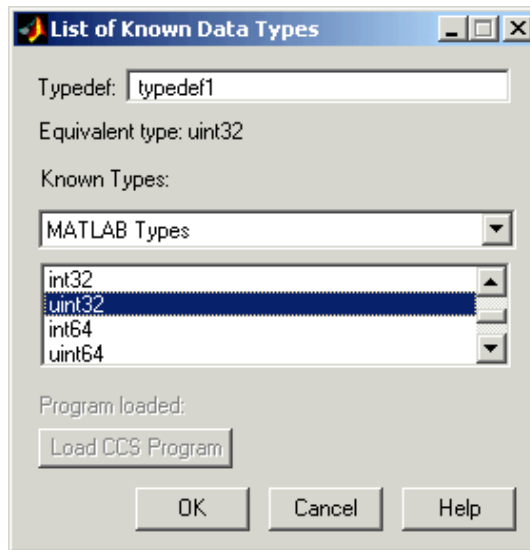
```
datatypemanager(cc);
```

With the DTM open, you can create a few custom data types.

datatypemanager

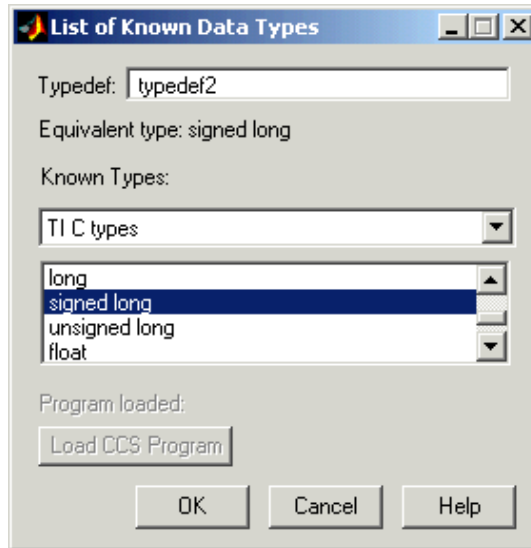
First example

Create a typedef (typedef1) that uses a MATLAB data type. typedef1 uses the equivalent data type uint32.



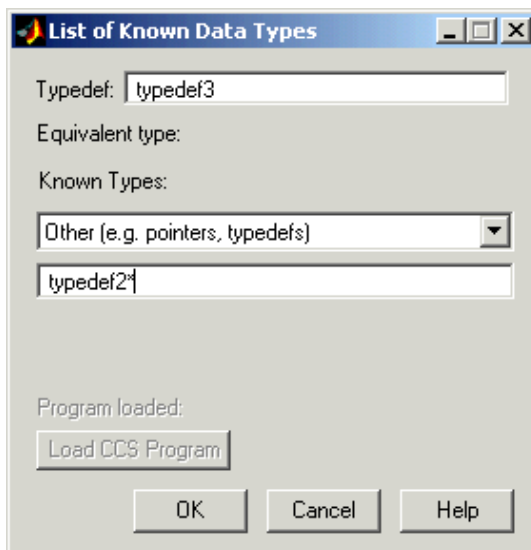
Second example

Create a second typedef (typedef2) that uses one of the TI C data types. Define typedef2 to use the signed long data type.



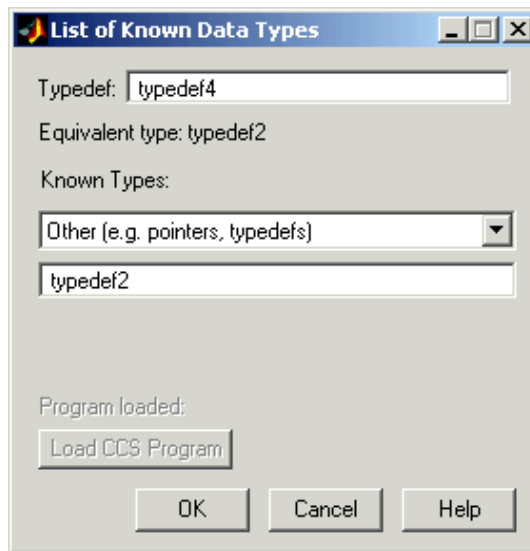
Third example

Create a typedef (typedef3) that refers to another typedef (typedef2).
Call this a nested typedef.



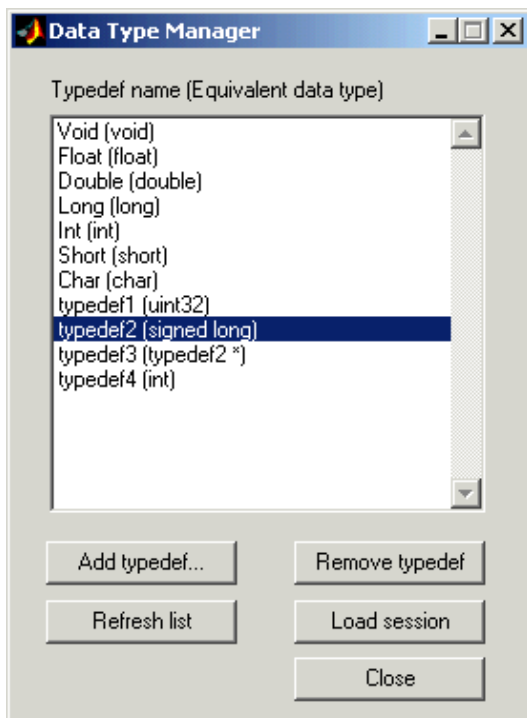
Notice that the referenced typedef, typedef2, is entered as a pointer (indicated by the added asterisk). Using the pointer form lets MATLAB recognize the data type that typedef2 represents. If you do not use the pointer, MATLAB converts typedef3 to a default value equivalent data type, in this case, int.

The next figure shows typedef4 created to use typedef2 rather than typedef2* for a nested typedef. Under **Equivalent type**, typedef4 has an equivalent data type of typedef2, as specified. But, when you look at the list of known data types in the Data Type Manager dialog box, you see that typedef4 maps to int, not typedef2, or eventually signed long.



Here is the DTM after you create all the example custom data types. Take note of typedef4 in this listing. You see typedef4 defaults to an equivalent data type int, where typedef3, also a nested type definition, retains the equivalent data type you assigned. Now you are ready to use a function that includes your custom type definitions in your hardware-in-the-loop development work.

datatypemanager



See Also `createobj`

Purpose	Define C function declaration in MATLAB for CCS application
Syntax	<pre>declare(objname, 'filetype', 'filename') declare(objname, 'decl', 'funcdeclaration')</pre>
Description	<p>When <code>createobj</code> cannot construct a function object to access a function, either because MATLAB could not find the function declaration for the function, or could not create the function object properties, use <code>declare</code> to pass the function declaration to MATLAB.</p> <p><code>declare(objname, 'filetype', 'filename')</code> passes your function declaration string to <code>objname</code> by providing the path to the file specified in <code>filename</code>. To set the type of file you are providing, input argument <code>filetype</code> can be one of three strings:</p> <ul style="list-style-type: none">• 'filename' — specifies that <code>filename</code> contains the path and filename for your header file that contains the function declaration• 'file' — same as <code>filename</code>• 'header' — specifies that <code>filename</code> is the path and name of a header file that contains the function declaration <p>When <code>declare</code> cannot find the declaration string because the specified header file or file is not available, use the next syntax to provide the complete declaration string explicitly.</p> <p><code>declare(objname, 'decl', 'funcdeclaration')</code> passes the declaration string in <code>funcdeclaration</code> to <code>objname</code>. To tell MATLAB that you are passing a declaration string, add the keyword decl, indicating that the next argument is the function declaration string. When you use <code>declare</code> to add a function declaration to <code>objname</code>, <code>declare</code> reads the input variables and return type for the declaration from <code>funcdeclaration</code> and populates the properties <code>inputvars</code>, <code>inputnames</code>, and <code>outputvar</code> of <code>objname</code>. When <code>declare</code> successfully determines the input and output variables, <code>objname</code> contains the updated property values.</p>

declare

Examples

The following code passes the function declaration for `cdotprod` to MATLAB and updates the properties of `ff` to match the declaration:

```
declare(ff,'decl','int cdotprod (short x[], short y[], short n)')
```

In the case of a very complex function declaration, assign the declaration string to a variable and pass the variable in the `declare` syntax:

```
declstring=['int cdotprod (short x[], short y[], short n)']  
declare(ff,'decl',declstring)
```

See Also

`createobj`, `execute`, `getinput`, `getoutput`, `goto`, `resume`, `run`

Purpose

Remove debug points in addresses or source files in CCS

Syntax

```
delete(cc,addr,'type')
delete(cc,addr,'type',timeout)
delete(cc,addr)
delete(cc,filename,line,'type')
delete(cc,filename,line,'type',timeout)
delete(cc,filename,line)
delete(cc,'all')
delete(cc,'all','break',timeout)
```

Description

`delete(cc,addr,'type')` removes a debug point located at the memory address identified by `addr` for your target digital signal processor. Object `cc` identifies which target has the debug point to delete. CCS provides several types of debug points specified by `type`. To learn more about the behavior of the various debugging points refer to your CCS documentation. Options for `type` include the following to remove breakpoints and probe points:

- **'break'** — removes a breakpoint. This is the default.
- **' '** — same as **'break'**.
- **'probe'** — removes a probe point.

When you use it, `delete` operates in *blocking* mode, meaning that after you issue the `delete` command, you do not regain control in MATLAB until the `delete` operation is completed successfully — you are blocked from further processing. `delete` waits for the period defined by either `timeout` or `cc.timeout`. If the `delete` operation does not get completed within the specified time period, `delete` returns an error and control.

Unlike deleting `break` and `probe` points in CCS, you cannot enter `addr` as a C function name, valid C expression, or a symbol name.

When the `type` you specify does not match the debug point type at the selected location, or no debug point exists, Link for Code Composer Studio returns an error reporting that it could not find the specified debugging point.

delete

`delete(cc,addr,'type',timeout)` adds the optional input parameter `timeout` that determines how long Link for CCS waits for a response to the request to delete a breakpoint. If the response is not received before the time-out period expires, the deletion process fails with a time-out error. The `timeout` input argument is valid only when you are deleting a breakpoint. When you omit the `timeout` argument, `delete` uses the default value defined by `cc.timeout`

`delete(cc,addr)` is the same as the previous syntax except the function defaults to '**break**' for removing a breakpoint.

`delete(cc,filename,line,'type')` lets you specify the line from which you are removing the debug point. Argument `line` specifies the line number in the source file `file` in CCS. `line`, in decimal notation, defines the line number of the debugging point to remove. To identify the source file, argument `filename` contains the name of the file in CCS, entered as a string in single quotation marks. Do not include the path to the file. `delete` ignores the path information. `type` accepts one of two strings — **break** or **probe** — as defined previously. When the type of debugging point you specify with the `type` string does not match the debug point type at the specified location, or no debug point exists, Link for Code Composer Studio returns an error that it could not find the debug point.

`delete(cc,filename,line,'type',timeout)` adds the optional input parameter `timeout` that determines how long Link for CCS waits for a response to the request to delete a breakpoint. If the response is not received before the time-out period expires, the deletion process fails with a time-out error. The `timeout` input argument is valid only when you are deleting a breakpoint. When you omit the `timeout` argument, `delete` uses the default value defined by `cc.timeout`

`delete(cc,filename,line)` defaults to 'break' to remove a breakpoint.

`delete(cc,'all')` removes all valid breakpoints in the project source files. This does not remove probe points and it does not remove invalid breakpoints..

`delete(cc,'all','break',timeout)` removes all of the valid breakpoints in the project source files. This command does not remove

probe points and it does not remove invalid breakpoints. Note that you can use the optional input parameter `timeout` that determines how long Link for CCS waits for a response to the request to delete all of the debug points. If the response is not received before the time-out period expires, the deletion process fails with a time-out error. When you omit the `timeout` argument, `delete` uses the default value defined by `cc.timeout`.

See Also

`address`, `insert`, `run`

deleteregister

Purpose Remove registers from list of saved registers in savedregs property of function objects

Syntax `deleteregister(ff,regname)`
`deleteregister(ff,reglist)`

Description `deleteregister(ff,regname)` removes register `regname` from the list of registers that get preserved or reverted when a function is finished running. `ff` identifies the program function to which the register applies. You can delete any register you added from the saved registers list. You cannot delete registers that are on the default list of saved registers — the must save registers.

When you issue the `createobj` call to create a handle to a function, the compiler creates the default list of saved registers. When you execute the function, the compiler saves the registers in the list, runs its process, and after completing its process, restores the saved registers to their initial state using the contents of the saved registers.

After a function generates a result, the execution process returns the saved registers to their initial states and values. When you delete a register you added to the saved registers list, the deleted register is not restored or saved with other registers in the list.

For each processor family, the default list of saved registers changes, as shown in these sections. The default lists include registers that the compiler saves and that MATLAB requires for Link for Code Composer Studio to operate correctly.

Default Saved Registers for C28x Processors

AL, AH, AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7, XAR0, XAR1, XAR2, XAR3, XAR4,XAR5, XAR6, XAR7, SP, T, TL, PL, PH, DP

Default Saved Registers for C54x Processors

AR1, AR6, AR7, and SP (required by MATLAB, not the compiler)

Default Saved Registers for C55x Processors

T0, T1, T2, T3, TRN0, TRN1, AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7, XAR0, XAR1, XAR2, XAR3, XAR4, XAR5, XAR6, XAR7, AC0, AC1, AC2, AC3

Default Saved Registers for C62x and C67x Processors

A0, A2, A6, A7, A8, A9. Also B0, B1, B2, B4, B5, B6, B7, B8, B9. To support MATLAB requirements, B15 (the stack pointer) gets saved as well.

Registers A3, A4, A5, and B3 — your function must preserve these but they are not needed for reading function output.

Default Saved Registers for C64x Processors

A0, A2, A6, A7, A8, A9, A16, A17, A19, A19, A20, A21, A22, A23, A24, A25, A26, A27, A28, A29, A30, A31. Also B0, B1, B2, B4, B5, B6, B7, B8, B9, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31. To support MATLAB requirements, B15 (the stack pointer) gets saved as well.

Register B15 — not required by the compiler, but is required by MATLAB and is saved.

Registers A3, A4, and A5 — function must preserve these but they are needed for reading function output.

Default Saved Registers for R1x and R2x Processors

R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15

`deleteregister(ff,reglist)` deletes the register names in `reglist` from the list of registers that get preserved when a task is finished. `ff` identifies the function to which the register applies. `reglist` is a cell array that contains the names of registers to remove from the saved registers collection.

See Also

`addrregister`

deref

Purpose Object that accesses object pointed to by pointer object

Syntax
`objname2 = deref(objname)`
`objname2 = deref(objname, index)`

Description `objname2 = deref(objname)` creates `objname2`, an object that accesses the target of `objname`, which is either a pointer or `rpointer` object. `deref` does exactly what the dereferencing operator `*` does in C. Pointer and `rpointer` objects support using function `deref`.
`objname2 = deref(objname, index)` selects one member, specified by `index`, of an array of pointers. `objname2` accesses only the single array member that `index` specifies.

See Also `createobj`, `read`, `write`

Purpose List files in current CCS IDE working directory

Syntax `dir(cc)`

Description `dir(cc)` lists the files and directories in the current CCS IDE working directory. This does not reflect your MATLAB working directory or change the working directory.

Use `cd` to change your CCS IDE working directory.

See Also `cd`, `open`

disable

Purpose Disable RTDX interface, specified channel, or all RTDX channels

Syntax

```
disable(rx, 'channel')
disable(rx, 'all')
disable(rx)
```

Description `disable(rx, 'channel')` disables the open channel specified by the string channel, for rx. Input argument rx represents the RTDX portion of the associated link to CCS IDE.

`disable(rx, 'all')` disables all the open channels associated with rx.

`disable(rx)` disables the RTDX interface for rx.

Important Requirements for Using disable

On the target side, `disable` depends on RTDX to disable channels or the interface. You must meet the following requirements to use `disable`:

- 1** The target must be running a program when you use `disable` for channels or the RTDX interface.
- 2** You must have the enabled the RTDX interface.
- 3** Your target program must be polling periodically for `disable` to work.

Examples When you have opened and used channels to communicate with a target processor, you should disable the channels and RTDX before ending your session. Use `disable` to switch off open channels and disable RTDX, as follows:

```
disable(cc.rtdx, 'all') % Disable all open RTDX channels.
disable(cc.rtdx)       % Disable RTDX interface.
```

See Also `close`, `enable`, `open`

Purpose Display properties of link to CCS IDE or RTDX link

Syntax

```
display(cc)
display(rx)
display(objectname)
display(cc.type)
```

Description This function is similar to omitting the closing semicolon from an expression on the command line, except that `display` does not display the variable name. `display` provides a formatted list of the property names and property values for a link to CCS IDE. To return the configuration data, `display` calls the function `disp`. To return a list of object properties, listed by the actual property names, use `get` with the object.

`display(cc)` returns the information about the `cc` object, listing the properties and values assigned to `cc`.

`display(rx)` returns the information about the `rtdx` object that is part of a `cc` object, listing the properties and values assigned to `cc.rtdx`.

`display(objectname)` returns the properties and property values for `objectname`. This syntax supports all objects except `cc`, `rtdx`, and `cc.type`.

`display(cc.type)` returns the properties and property values for the `cc.type` object. Note that the properties associate with the `cc` object.

The following example illustrates the default display for a link to CCS IDE:

```
cc = ccsdsp;

display(cc)
CCSDSP Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
```

display

```
Board number      : 0
Processor number  : 0
Default timeout   : 10.00 secs

RTDX channels     : 0
```

Using display with multiprocessor targets

To support target boards that contain more than one processor, `display` behaves slightly differently when `cc` accesses multiprocessor boards.

The syntax

```
display(cc)
```

returns information about all of the members of the object. When the target has multiple processors, the information returned includes the details of all of the available processors on the target.

Examples

Try this example to see the `display` for an RTDX link to a target processor:

```
cc = cc dsp;
rx=(cc.rtdx)    % Assign the RTDX portion of cc to rx.

RTDX channels   : 0

display(rx)

RTDX channels   : 0
```

See Also

`get`, `set`

Purpose	Enable RTDX interface, specified channel, or all RTDX channels
Syntax	<pre>enable(rx, 'channel') enable(rx, 'all') enable(rx)</pre>
Description	<p><code>enable(rx, 'channel')</code> enables the open channel specified by the string <code>channel</code>, for RTDX link <code>rx</code>. The input argument <code>rx</code> represents the RTDX portion of the associated link to CCS IDE.</p> <p><code>enable(rx, 'all')</code> enables all the open channels associated with <code>rx</code>.</p> <p><code>enable(rx)</code> enables the RTDX interface for <code>rx</code>.</p> <p>Important Requirements for Using enable</p> <p>On the target side, <code>enable</code> depends on RTDX to enable channels. Therefore the you must meet the following requirements to use <code>enable</code>:</p> <ol style="list-style-type: none">1 The target must be running a program when you enable the RTDX interface. When the target is not running, the state defaults to disabled.2 You must enable the RTDX interface before you enable individual channels.3 Channels must be open before you can enable them.4 Your target program must be polling periodically for <code>enable</code> to work.5 Using code in the program running on the target to enable channels overrides the default disabled state of the channels.
Examples	<p>To use channels to RTDX, you must both open and enable the channels:</p> <pre>cc = ccstdsp; % Create a new link. enable(cc.rtdx) % Enable the RTDX interface. open(cc.rtdx, 'inputchannel', 'w') % Open a channel for sending % data to the target processor.</pre>

enable

```
enable(cc.rtdx,'inputchannel') % Enable the channel so you can use  
% it.
```

See Also disable, open

Purpose

Equivalent string or numeric value for input argument

Syntax

```
value = equivalent(objname,input)
```

Description

value = equivalent(objname,input) returns value as either

- The decimal numeric equivalent of input when input is a string
- The string equivalent value of input when input is a numeric

input can be a single value, a single string, an array of values or strings, or a cell array of values or strings.

Numeric objects, string objects, rstring objects, and enum objects all support equivalent.

The conversion process depends on the setting of the charconversion property of the object and applies only to string and rstring objects. Currently, the only property value allowed for charconversion is 'ASCII' indicating that strings are treated as ASCII characters and numeric values get converted to the ASCII equivalents.

See Also

cast, convert

execute

Purpose Execute function on target through CCS

Syntax
`output_val = execute(ff)`
`output_val = execute(ff,input1,value1,...,inputn,valuen)`

Description `output_val = execute(ff)` runs the function specified by handle `ff` on your target hardware. When you do not specify values for the inputs to the function, `execute` uses the values stored in property `inputvars` for the arguments. The function runs until the end of the function, or until it reaches a breakpoint. After executing the function, the execution process puts the return value in the assigned location in property `outputvar` of `ff`. From MATLAB, use `read` to check the result stored in `outputvar`. In this form, `output_val` holds the return value from executing the function.

Before you use `execute` to run a function, use `goto` to position the program counter to the beginning of the function. `execute` assumes that you have completed this step; it does not search for the function. Execution starts from the program counter location and continues to the end of the function or an intervening breakpoint.

`output_val = execute(ff,input1,value1,...,inputn,valuen)` runs the function identified by `ff`, first writing the input values assigned by the `inputn/valuen` pairs to `inputvars`. Arguments `input1`, `input2`,...,`inputn` must be strings. `input1` through `inputn` can be either the names of the input arguments, or the number of the input argument in the argument list, such as 1 for the first argument, 2 for the second, up to `n` for the `n`th argument on the list. In this form, `output_val` holds the return value from executing the function. You must call `goto` before using this syntax, or `execute` fails.

See Also `goto`, `run`, `write`

Purpose

Flush data or messages from specified RTDX channel(s)

Syntax

```
flush(rx,channel,num,timeout)
flush(rx,channel,num)
flush(rx,channel,[],timeout)
flush(rx,channel)
flush(rx,'all')
```

Description

`flush(rx,channel,num,timeout)` removes `num` oldest data messages from the RTDX channel queue specified by `channel` in `rx`. To determine how long to wait for the function to complete, `flush` uses `timeout` (in seconds) rather than the global time-out period stored in `rx`. `flush` applies the time-out processing when it flushes the last message in the channel queue, since the flush function performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,channel,num)` removes the `num` oldest messages from the RTDX channel queue in `rx` specified by the string `channel`. `flush` uses the global time-out period stored in `rx` to determine how long to wait for the process to complete. Compare this to the previous syntax that specifies the time-out period. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,channel,[],timeout)` removes all data messages from the RTDX channel queue specified by `channel` in `rx`. To determine how long to wait for the function to complete, `flush` uses `timeout` (in seconds) rather than the global time-out period stored in `rx`. `flush` applies the time-out processing when it flushes the last message in the channel queue, since `flush` performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,channel)` removes all pending data messages from the RTDX channel queue specified by `channel` in `rx`. Unlike the preceding syntax options, you use this statement to remove messages for both read-configured and write-configured channels.

flush

`flush(rx, 'all')` removes all data messages from all RTDX channel queues.

When you use `flush` with a write-configured RTDX channel, Link for Code Composer Studio sends all the messages in the write queue to the target. For read-configured channels, `flush` removes one or more messages from the queue depending on the input argument `num` you supply and disposes of them.

Examples

To demonstrate `flush`, this example writes data to the target over the input channel, then uses `flush` to remove a message from the read queue for the output channel:

```
cc = ccdsp;
rx = cc.rtdx;
open(rx, 'ichan', 'w');
enable(rx, 'ichan');
open(rx, 'ochan', 'r');
enable(rx, 'ochan');
indata = 1:10;
writemsg(rx, 'ichan', int16(indata));
flush(rx, 'ochan', 1);
```

Now flush the remaining messages from the read channel:

```
flush(rx, 'ochan', 'all');
```

See Also

`enable`, `open`

Purpose

Access object properties

Syntax

```
get(cc, 'propertyname')
get(cc)
v = get(cc, 'propertyname')
get(rx, 'propertyname')
get(rx)
v = get(rx)
get(objname, 'propertyname')
get(objname)
v = get(objname)
```

Description

`get(cc, 'propertyname')` returns the property value associated with `propertyname` for link `cc`.

`get(cc)` returns all the properties and property values identified by the link `cc`.

`v = get(cc, 'propertyname')` returns a structure `v` whose field names are the link `cc` property names and whose values are the current values of the corresponding properties. `cc` must be a link. If you do not specify an output argument, MATLAB displays the information on the screen.

`get(rx, 'propertyname')` returns the property value associated with `propertyname` for link `rx`.

`get(rx)` returns all the properties and property values identified by the link `rx`.

`v = get(rx)` returns a structure `v` whose field names are the link `rx` property names and whose values are the current values of the corresponding properties. `rx` must be a link. If you do not specify an output argument, MATLAB displays the information on the screen.

`get(objname, 'propertyname')` returns the property value associated with `propertyname` for `objname`.

`get(objname)` returns all the properties and property values identified by `objname`.

`v = get(objname)` returns a structure `v` whose field names are the `objname` property names and whose values are the current values of the corresponding properties. `objname` must be an object in your MATLAB workspace. If you do not specify an output argument, MATLAB displays the information on the screen.

Examples

After you create a link for CCS IDE and RTDX, `get` provides a way to review the properties of the link.

```
cc=ccsdsp
```

```
CCSDSP 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
```

```
get(cc)
```

```
ans =
```

```
    app: [1x1 activex]
 dspboards: [1x1 activex]
  dspboard: [1x1 activex]
 dsptasks: [1x1 activex]
  dsptask: [1x1 activex]
  dspuser: [1x1 activex]
    rtdx: [1x1 rtdx]
apiversion: [1 0]
ccsappexe: 'D:\ticcs\cc\bin\cc_app.exe'
  boardnum: 0
   procnum: 0
```

```
        timeout: 10
        page: 0

v=get(cc)

v =

        app: [1x1 activex]
        dspboards: [1x1 activex]
        dspboard: [1x1 activex]
        dsptasks: [1x1 activex]
        dsptask: [1x1 activex]
        dspuser: [1x1 activex]
        rtdx: [1x1 rtdx]
    apiversion: [1 0]
    ccsappexe: 'D:\ticcs\cc\bin\cc_app.exe'
    boardnum: 0
    procnum: 0
    timeout: 10
    page: 0

v.app

ans =
    activex object: 1-by-1

v.rtdx

    RTDX channels      : 0
```

RTDX links work in the same way. Create an alias rx to the RTDX portion of cc, then use the alias with get:

```
rx=cc.rtdx

    RTDX channels      : 0
```

get

```
get(rx)

ans =

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

v=get(rx)

v =

    numChannels: 0
         Rtdx: [1x1 activex]
    RtdxChannel: {'' [] ''}
         procType: 103
         timeout: 10
v.timeout

ans =

    10

v.procType

ans =

    103
```

See Also

set

Purpose Specified input argument object from function object

Syntax `inputobj = getinput(ff,input_name)`

Description `inputobj = getinput(ff,input_name)` returns the input object that accesses `input_name`. Enter `input_name` in single quotation marks since it is a string.

Note After you execute a function, the information returned by `getinput` may not be the same as the information returned before you run the method.

This occurs because the compiler uses stack and register locations as temporary storage and may overwrite the contents of either the stack or registers during execution. In particular, when your function stores the function return value in one of the input variables, the compiler overwrites the value of the input with the output value. Refer to “Examples” to see this in use.

Examples Use `getinput` to see the properties of an input object in a function object:

```
sin_t=createobj(cc,'sin_taylor')
```

```
FUNCTION Object
```

```
Function name      : sin_taylor
File found        : hiltut.c
Start address     : [12328 0]
All variables     : a1, a2, a3, acc, x, xpow
Input variables   : x
Return type       : short
```

```
sin_t.inputvars
```

```
ans =
```

getinput

```
x: [1x1 ccs.rnumeric]
```

```
x_inobj=getinput(sin_t,'x')
```

```
NUMERIC Object stored in register(s):  
Symbol name      : x  
Register         : A4  
Datatype         : Unknown  
Wordsize         : 16 bits  
Register units per value : 1 ru  
Representation   : signed  
Bit padding (post) : 16  
Size             : [ 1 ]  
Total register units : 1 ru  
Array ordering   : row-major
```

```
x_inobj
```

```
NUMERIC Object stored in register(s):  
Symbol name      : x  
Register         : A4  
Datatype         : Unknown  
Wordsize         : 16 bits  
Register units per value : 1 ru  
Representation   : signed  
Bit padding (post) : 16  
Size             : [ 1 ]  
Total register units : 1 ru  
Array ordering   : row-major
```

Demonstrate that the information from `getinput` may change after executing a function.

In your CCS project:

```
void f12q15(double *x, short *r,int nx); % r is where the output  
                                         % is stored
```


Now, in MATLAB, here is the code that demonstrates `getinput` changing.

```
% Create function class

cc = ccdsp;
ff = createobj(cc,'f12q15')

% Create objects that will be used as inputs to f12q15

input_x = createobj(cc,'input_x') % Global variable--an array of
                                   % doubles
write(input_x,[0.1 2.5 8.0]) % Write data into input_x

input_r = createobj(cc,'input_r') % Global variable--an array of
                                   % shorts

% Get input objects and assign values

xobj = getinput(ff,'x')
write(xobj,input_x.address)

robj = getinput(ff,'r')
write(robj,input_r.address) % Also means 'set the result to point
                             % to the location of input_r'

nxobj = getinput(ff,'nx')
write(nxobj,3)

% Run the function

run(ff)

% Read the result
```

getinput

```
output_err = read( deref( robj ) ) % Returns the wrong result
                                     % because robj now holds a
                                     % different value
```

```
output_correct = read( input_r ) )
```

Gives the correct result because the address of `input_r` did not change.

See Also

`createobj`, `getoutput`

Purpose Object that accesses one structure member

Syntax
`objname2 = getmember(objname, membername)`
`objname2 = getmember(objname, index, membername)`

Description `objname2 = getmember(objname, membername)` returns the object `objname2` that represents `membername`, a member of the structure that `objname` accesses. `membername` must be a string and `objname` must represent a structure in memory. Once you create `objname2`, it becomes the object you use to read and write `membername`. Along with `createobj`, these are the only functions that create objects in the product.

The class of `objname2` depends on the data type of `membername` — numeric structure members return numeric objects, enumerated members return enum objects, pointers return pointer objects, and so on:

`objname2 = getmember(objname, index, membername)`

Examples Suppose you have declared a structure in your source code called `testdeepstr`, using code like this:

```
struct testdeepstr {
    int x_int;
    struct mystructa x_str;
    struct mystructa z_str[2];
} str_recur;
```

Now, `getmember` creates objects that directly access members of `str_recur`:

```
str_recur=createobj(cc, 'str_recur')
```

STRUCTURE Object:

```
Symbol Name           : str_recur
Address                : [ 2147500816 0]
Address Units per value : 224 AU
Size                   : [ 1 ]
Total Address Units    : 224 AU
```

getmember

```
Array ordering      : row-major
Members            : 'x_int', 'x_str', 'z_str'
```

```
x_str=getmember(structttest, 'x_str')
```

STRUCTURE Object:

```
Symbol Name        : x_str
Address            : [ 2147500824 0]
Address Units per value : 72 AU
Size               : [ 1 ]
Total Address Units : 72 AU
Array ordering     : row-major
Members           : 's_int', 'a_int', 's_double', 'a_char'
```

Even when the structure member is itself a structure, `getmember` provides access directly to the nested structure, or to members within the nested structure:

```
s_double=getmember(nestx_str, 's_double')
```

NUMERIC Object

```
Symbol Name        : s_double
Address            : [ 2147500872 0]
Wordsize          : 64 bits
Address Units per value : 8 AU
Representation     : float
Binary point position : 0
Size               : [ 1 ]
Total address units : 8 AU
Array ordering     : row-major
Endianness        : little
```

Numeric object `s_double` is now your handle to write to or read from member `s_double`:

```
read(s_double)
```

```
ans =  
  
-1.4938e+059  
  
write(s_double,2)  
read(s_double)
```

```
ans =  
  
2
```

See Also createobj, read, write

getoutput

Purpose Access output from function object

Syntax `out_obj = getoutput(ff)`

Description `out_obj = getoutput(ff)` returns in `out_obj` the object that accesses the return from `ff`. The input argument `ff` must be a function object constructed either by `createobj` or a combination of `createobj` and `declare`. To return any value, `ff` must be a fully populated function object, with all the required input and output objects.

Examples Use `getoutput` to see the properties of the output object in a function object:

```
sin_t=createobj(cc,'sin_taylor')
```

```
FUNCTION Object
```

```
Function name      : sin_taylor
File found         : hiltut.c
Start address      : [12328 0]
All variables      : a1, a2, a3, acc, x, xpow
Input variables    : x
Return type        : short
```

```
getoutput(sin_t)
```

```
NUMERIC Object stored in register(s):
```

```
Symbol name       :
Register          : A4
Datatype          : Unknown
Wordsize          : 16 bits
Register units per value : 1 ru
Representation     : signed
Bit padding (post) : 16
Size              : [ 1 ]
Total register units : 1 ru
Array ordering     : row-major
```

Note that you do not need the output variable name in `getoutput`. Since there can only be one output object (one output variable) you do not need to specify which object to display.

See Also

`createobj`, `getinput`

gettypeinfo

Purpose Information about existing type definition in type object

Syntax `gettypeinfo(cc.type, 'typename')`

Description `gettypeinfo(cc.type, 'typename')` returns all the available information about the user-defined data type `typename` in the type object `cc.type`.

Examples Here is what happens when you use `gettypeinfo` to learn about a type in the type class:

```
cc.type
```

```
Defined types      : Void, Float, Double, Long, Int, Short, Char,  
mynewtypedef
```

```
gettypeinfo(cc.type, 'Double')
```

```
ans =
```

```
    type: 'double'  
    size: 1  
    uclass: 'numeric'
```

One important note — type names are case sensitive. `double` and `Double` are not the same.

See Also `add`, `clear`, `createobj`

Purpose

Position program counter to specified location in project code

Syntax

```
goto(cc, 'functionname')  
goto(ff)  
goto(ff, 'input1', value1, ..., 'inputn', valuen)
```

Description

`goto(cc, 'functionname')` opens the source file in CCS that contains `functionname` and positions the cursor at the beginning of `functionname`. Using `goto` can help you locate and work with a file that contains a specific function without searching through all the files.

`goto(ff)` positions the program counter to the beginning of the function accessed by `ff`. Using `goto` in this syntax prepares the function to be executed but does not place any information in the registers associated with the function. Before you use this form of `goto`, you can pass the necessary values for the function input arguments into the appropriate registers and stack locations. You can do this whether the function has input parameters or not.

In the following sections, you see the registers and memory locations on each processor that are affected by preparing to run the function.

C2800 Family Input Argument Storage Allocation

C2800 processors interpret and store input argument data in a way quite different from the other TI processors.

The processor first checks the sizes of the function input arguments. After determining which inputs are 32-bit, pointers, and 16-bit arguments, the processors starts to allocate storage for the data.

Having sorted the input arguments by data size and type, the processor starts to allocate storage by handling the 32-bit arguments. The processor places the first 32-bit input argument (either long or float data types) into the accumulator, registers AH and AL. Other 32-bit input arguments, if any, get stored on the stack.

Next come the pointer input arguments. The first and second pointer input arguments go to registers XAR4 and XAR5. If the function

prototype uses more than two pointers as input arguments, the remaining pointers go on the stack.

Finally, the processor treats the 16-bit input arguments. Where 16-bit arguments (`ints`) go depends on the number and kind of other input arguments to the function. The first four 16-bit inputs go into `AH`, `AL`, `XAR4`, and `XAR5`, in that order, if the registers are available.

But recall that 32-bit inputs go into `AL` and `AH`, and pointers go into `XAR4` and `XAR5`. So, 16-bit input arguments go into any empty location among `AL`, `AH`, `XAR4`, and `XAR5`. Remaining 16-bit arguments go on the stack.

To make this a bit more clear, this short example uses five input arguments to function `function`. Input arguments `a` and `c` are 32-bit arguments, `b` is a pointer, and `d` and `e` are 16-bit arguments. For a function like this one

```
void function(a,b,c,d,e)
```

the compiler allocates the input arguments as shown here, in order.

- 1** `a` goes into register `AH`. It is the first 32-bit input argument.
- 2** `c` goes into register `AL`. It is the second 32-bit input argument.
- 3** `b`, the first pointer, goes into `XAR4`
- 4** `d`, the first 16-bit argument, goes into `XAR5`
- 5** `e`, the second 16-bit argument, goes on the stack, since `AH`, `AL`, `XAR4`, and `XAR5` are full.

For this example, additional input arguments, if there were any, would go on the stack.

C5400 Family Input Argument Storage Allocation

Argument	Register	For Long Arguments	Description
value1	A	A	First input value to function
value2 and higher	Stack	Stack	All input arguments after the tenth argument get placed on the stack
Returned Argument	A	A	Returned argument

C6000 Family Input Argument Storage Allocation

Argument	Register	For Long Arguments	Description
value1	A4	A5:A4	First input value to function
value2	B4	B5:B4	Second input value to function
value3	A6	A7:A6	Third input value to function
value4	B6	B7:B6	Fourth input value to function
value5	A8	A9:A8	Fifth input value to function
value6	B8	B9:B8	Sixth input value to function

Argument	Register	For Long Arguments	Description
value7	A10	A11:A10	Seventh input value to function
value8	B10	B11:B10	Eighth input value to function
value9	A12	A13:A12	Ninth input value to function
value10	B12	B13:B12	Tenth input value to function
value11 and higher	Stack	Stack	All input arguments after the tenth argument get placed on the stack.
Pointer to returned structure	A3	N/A	Pointer
Return address register	B3	N/A	Address of register
Returned argument	A4	A5:A4	Returned argument
Data page pointer (DP)	B14	N/A	Specifies the data page. Always 1 for the C6000 processor family.
Frame Pointer (FP)	A15	N/A	Specifies the frame pointer location
Stack Pointer (SP)	B15	N/A	Specifies the stack pointer location

`goto(ff, 'input1', value1, ..., 'inputn', valuen)` positions the PC to the beginning of the function accessed by `ff`, and sets the function input arguments `input1` through `inputn` to the values `value1` through

valuen, as provided in the goto syntax. The order of the input names and values is not important; it does not need to match the order of the input arguments in the function prototype or declaration. input1 through inputn can be either the names of the input arguments, or the number of the input argument in the argument list, such as 1 for the first argument, 2 for the second, up to n for the nth argument on the list.

Note goto must be followed by execute.

See Also

delete, execute, insert, run

halt

Purpose Terminate execution of process running on target

Syntax `halt(cc,timeout)`
`halt(cc)`

Description `halt(cc,timeout)` immediately stops program execution by the processor. After the processor stops, `halt` returns to the host. `timeout` defines, in seconds, how long the host waits for the target processor to stop running. To resume processing after you halt the processor, use `run`. Also, the `read(cc,'pc')` function can determine the memory address where the processor stopped after you use `halt`.

`timeout` defines the maximum time the routine waits for the processor to stop. If the processor does not stop within the specified timeout period, the routine returns with a timeout error.

`halt(cc)` immediately stops program execution by the processor. After the processor stops, `halt` returns to the host. In this syntax, the timeout period defaults to the global timeout period specified in `cc`. Use `get(cc)` to determine the global timeout period.

Using halt with multiple processor target boards

When you issue a `halt` from the command line, it applies to every processor that the `cc` object represents. Thus `halt` stops every running processor for the object. We call this process broadcasting the method.

Examples

Use one of the provided demonstration programs to show how `halt` works. From the CCS IDE demonstration programs, load and run `volume.out`.

At the MATLAB prompt create a link to CCS IDE

```
cc = ccsdsp
```

Check whether the program `volume.out` is running on the processor.

```
isrunning(cc)
```

```
ans =  
    1  
  
cc.isrunning % Alternate syntax for checking the run status.  
  
ans =  
    1  
halt(cc) % Stop the running application on the processor.  
isrunning(cc)  
  
ans =  
    0
```

Issuing the halt stopped the process on the target. Checking in CCS IDE shows that the process has stopped.

See Also

ccsdsp, isrunning, run

info

Purpose Information about target processor

Syntax `info = info(cc)`
`info = info(rx)`

Description `info = info(cc)` returns the property names and property values associated with the processor targeted by `cc`. `info` is a structure containing the following information elements and values:

Structure Element	Data Type	Description
<code>info.procname</code>	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with <code>cc</code> .
<code>info.isbigendian</code>	Boolean	Value describing the byte ordering used by the target processor. When the processor is big-endian, this value is 1. Little-endian processors return 0.
<code>info.family</code>	Integer	Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors.

Structure Element	Data Type	Description
<code>info.subfamily</code>	Decimal	Decimal representation of the hexadecimal identification value that TI assigns to the processor to identify the processor subfamily. IDs range from 0x000 to 0x3822. Use <code>dec2hex</code> to convert the value in <code>info.subfamily</code> to standard notation. For example <pre>dec2hex(info.subfamily)</pre> produces '67' when the processor is a member of the 67xx processor family.
<code>info.timeout</code>	Integer	Default timeout value MATLAB uses when transferring data to and from CCS. All functions that use a timeout value have an optional <code>timeout</code> input argument. When you omit the optional argument, MATLAB uses this default value – 10s.

`info = info(rx)` returns `info` as a cell array containing the names of your open RTDX channels.

Using info with multiprocessor boards

Method `info` works with targets that have more than one processor by returning the information for each processor accessed by the `cc` object you created with `ccsdsp`. The structure of information returned is identical to the single processor case, for every included processor.

Examples

On a PC with a simulator configured in CCS IDE, `info` returns the configuration for the processor being simulated:

```
info(cc)

ans =

    procname: 'CPU'
    isbigendian: 0
```

```
family: 320
subfamily: 103
timeout: 10
```

This example simulates the TMS320C6211 processor running in little-endian mode. When you use CCS Setup Utility to change the processor from little-endian to big-endian, `info` shows the change.

```
info(cc)

ans =

    procname: 'CPU'
  isbigendian: 1
    family: 320
  subfamily: 103
    timeout: 10
```

If you have two open channels, `chan1` and `chan2`,

```
info = info(rx)
```

returns

```
info =
'chan1'
'chan2'
```

where `info` is a cell array. You can dereference the entries in `info` to manipulate the channels. For example, you can close a channel by dereferencing the channel in `info` in the `close` function syntax.

```
close(rx.info{1,1})
```

See Also

`ccsdsp`, `dec2hex`, `get`, `set`

Purpose

Add debug point to source file or address in CCS

Syntax

```
insert(cc,addr,'type')
insert(cc,addr,'type',timeout)
insert(cc,addr)
insert(cc,filename,line,'type')
insert(cc,filename,line,'type',timeout)
insert(cc,filename,line)
```

Description

`insert(cc,addr,'type')` adds a debug point located at the memory address identified by `addr` for your target digital signal processor. The link `cc` identifies which target has the debug point to insert. CCS provides several types of debug points specified by `type`. Options for `type` include the following strings to define Breakpoints, Probe Points, and Profile points:

- `'break'` — add a breakpoint. It defines a point at which program execution stops.
- `' '` — same as `'break'`.
- `'probe'` — add a Probe Point that updates a CCS window during program execution. When CCS connects your probe point to a window, the window gets updated only when the executing program reaches the Probe Point.
- `'profile'` — add a point in an executing program at which CCS gathers statistics about events that occurred since encountering the previous profile point, or from the start of your program.

When you use it, `insert` operates in *blocking* mode, meaning that after you issue the `insert` command, you do not regain control in MATLAB until the `insert` breakpoint operation is completed successfully — you are blocked from further processing. `insert` waits for the period defined by either `timeout` or `cc.timeout`. If the `insert` operation does not get completed within the specified time period, `insert` returns an error and control.

When you use the line input argument to insert a breakpoint on a specified line, `line` must represent a valid line. If `line` does not specify a valid line, `insert` returns an error and does not insert the breakpoint.

Enter `addr` as a hexadecimal address, not as a C function name, valid C expression, or a symbol name.

To learn more about the behavior of the various debugging points refer to your CCS documentation.

`insert(cc, addr, 'type', timeout)` adds the optional input parameter `timeout` that determines how long Link for CCS waits for a response to a request to insert a breakpoint. If the response is not received before the time-out period expires, the insertion process fails with a time-out error. Adding the `timeout` input argument is valid only when you are inserting a breakpoint. When you omit the `timeout` argument, `insert` uses the default value defined by `cc.timeout`

`insert(cc, addr)` is the same as the previous syntax except the `type` string defaults to 'break' for inserting a Breakpoint.

`insert(cc, filename, line, 'type')` lets you specify the line where you are inserting the debug point. `line`, in decimal notation, specifies the line number in `filename` in CCS where you are adding the debug point. To identify the source file, `filename` contains the name of the file in CCS, entered as a string in single quotation marks. Do not include the path to the file. `insert` ignores the file path information if you add it to `filename`. `type` accepts one of three strings — `break`, `probe`, or `profile` — as defined previously. When the line or file you specified does not exist, the Link for Code Composer Studio returns an error explaining that it could not insert the debug point.

`insert(cc, filename, line, 'type', timeout)` adds the optional input parameter `timeout` that determines how long Link for CCS waits for a response to a request to insert a breakpoint. If the response is not received before the time-out period expires, the insertion process fails with a time-out error. Adding the `timeout` input argument is valid only when you are inserting a breakpoint. When you omit the `timeout`

`insert(cc,filename,line)` defaults to type 'break' to insert a breakpoint.

Example

Open a project in CCS IDE, such as `volume.pjt` in the tutorial folder where you installed CCS IDE. Although you can do this from CCS IDE, use the Link for Code Composer Studio functions to open the project and activate the appropriate source file where you add the breakpoint. Remember to load the program file `volume.out` so you can access symbols and their addresses.

```
cd (cc,'c:\ti\tutorial\sim62xx\volume1') % Default install;
wd=cd(cc);

wd =

c:\ti\tutorial\sim62xx\volume1

open(cc,'volume.pjt');

build(cc, 30);
```

Now add a breakpoint and a probe point.

```
insert(cc,15424,'break') % Adds a breakpoint at symbol "main"
insert(cc,'volume.c',47,'probe') % Adds a probe point on line 47
```

Switch to CCS IDE and open `volume.c`. Note the blue diamond and red circle in the left margin of the `volume.c` listing. Red circles indicate Breakpoints and blue diamonds indicate Probe Points.

Use `symbol` to return a structure listing the symbols and their addresses for the current program file. `symbol` returns a structure that contains all the symbols. To display all the symbols with addresses, use a loop construct like the following:

```
for k=1:length(s),disp(k),disp(s(k)),end
```

where structure `s` holds the symbols and addresses.

insert

See Also

address, delete, run

Purpose Determine whether RTDX link is enabled for communications

Syntax `isenabled(rx, 'channel')`
`isenabled(rx)`

Description `isenabled(rx, 'channel')` returns `ans=1` when the RTDX channel specified by string 'channel' is enabled for read or write communications. When 'channel' has not been enabled, `isenabled` returns `ans=0`.

`isenabled(rx)` returns `ans=1` when RTDX has been enabled, independent of any channel. When you have not enabled RTDX you get `ans=0` back.

Important requirements for using isenabled

On the target side, `isenabled` depends on RTDX to determine and report the RTDX status. Therefore the you must meet the following requirements to use `isenabled`.

- 1** The target must be running a program when you query the RTDX interface.
- 2** You must enable the RTDX interface before you check the status of individual channels or the interface.
- 3** Your target program must be polling periodically for `isenabled` to work.

Note For `isenabled` to return reliable results, your target must be running a loaded program. When the target is not running, `isenabled` returns a status that may not represent the true state of the link or RTDX.

Examples With a program loaded on your target, you can determine whether RTDX channels are ready for use. Restart your program to be sure it is

isenabled

running. The target must be running for `isenabled` to work, as well as for `enabled` to work. In this example, we created a link `cc` to begin.

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

MATLAB returns 1 indicating that your channel 'ichan' is enabled for RTDX communications. To determine the mode for the channel, use `cc.rtdx` to display the properties of link `cc.rtdx`.

See Also

`clear`, `disable`, `enable`

Purpose

Determine whether MATLAB can read specified memory block

Syntax

```
isreadable(cc,address,'datatype',count)
isreadable(cc,address,'datatype')
isreadable(rx,'channel')
```

Description

`isreadable(cc,address,'datatype',count)` returns 1 if the processor referred to by `cc` can read the memory block defined by the `address`, `count`, and `datatype` input arguments. When the processor cannot read any portion of the specified memory block, `isreadable` returns 0. You use the same memory block specification for this function as you use for the `read` function.

The data block being tested begins at the memory location defined by `address`. `count` determines the number of values to be read. `datatype` defines the format of data stored in the memory block. `isreadable` uses the `datatype` string to determine the number of bytes to read per stored value. For details about each input parameter, read the following descriptions.

`address` — `isreadable` uses `address` to define the beginning of the memory block to read. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the target processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector `[location, page]`, a string, or a decimal value.

When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ccsdsp` sets the page to 0 at creation if you omit the page property as an input argument. For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
'1F'	String	Location is 31 decimal on the page referred to by cc (page)
10	Decimal	Address is 10 decimal on the page referred to by cc (page)
[18,1]	Vector	Address location 10 decimal on memory page 1 (cc (page) = 1)

To specify the address in hexadecimal format, enter the address property value as a string. `isreadable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that when you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc (page)`.

`count` — a numeric scalar or vector that defines the number of datatype values to test for being readable. To assure parallel structure with `read`, `count` can be a vector to define multidimensional data blocks. This function always tests a block of data whose size is the product of the dimensions of the input vector.

`datatype` — a string that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the `datatype` you specify. `datatype` determines how many bytes to check for each memory value. `isreadable` supports the following data types:

datatype String	Number of Bytes/Value	Description
'double'		Double-precision floating point values
'int8'		Signed 8-bit integers
'int16'		Signed 16-bit integers
'int32'		Signed 32-bit integers
'single'		Single-precision floating point data
'uint8'		Unsigned 8-bit integers
'uint16'		Unsigned 16-bit integers
'uint32'		Unsigned 32-bit integers

Like the `iswritable`, `write`, and `read` functions, `isreadable` checks for valid address values. Illegal address values would be any address space larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`isreadable(cc, address, 'datatype')` returns 1 if the processor referred to by `cc` can read the memory block defined by the `address`, and `datatype` input arguments. When the processor cannot read any portion of the specified memory block, `isreadable` returns 0. Notice that you use the same memory block specification for this function as you use for the `read` function. The data block being tested begins at the memory location defined by `address`. When you omit the count option, count defaults to one.

`isreadable(rx, 'channel')` returns a 1 when the RTDX channel specified by the string `channel`, associated with link `rx`, is configured for read operation. When `channel` is not configured for reading, `isreadable` returns 0.

isreadable

Like the `iswritable`, `read`, and `write` functions, `isreadable` checks for valid address values. Illegal address values are address spaces larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

Note `isreadable` relies on the memory map option in CCS IDE. If you did not properly define the memory map for the processor in CCS IDE, `isreadable` does not produce useful results. Refer to your Code Composer Studio documentation for more information about configuring memory maps.

Examples

When you write scripts to run models in MATLAB and CCS IDE, the `isreadable` function is very useful. Use `isreadable` to check that the channel from which you are reading is configured properly.

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

% Define read and write channels to the target linked by cc.
open(rx, 'ichannel', 'r');
open(rx, 'ochannel', 'w');
enable(rx, 'ochannel');
enable(rx, 'ichannel');

isreadable(rx, 'ochannel')
ans=
    0
isreadable(rx, 'ichannel')
ans=
    1
```

Now that your script knows that it can read from `ichannel`, it proceeds to read messages as required.

See Also `hex2dec`, `iswritable`, `read`

isrtdxcapable

Purpose Determine whether target processor supports RTDX

Syntax `b=isrtdxcapable(cc)`

Description `b=isrtdxcapable(cc)` returns `b=1` when the target processor referenced by link `cc` supports Real-Time Data Exchange (RTDX). When the target does not support RTDX, `isrtdxcapable` returns `b=0`.

Using `isrtdxcapable` with multiprocessor boards

When your target board contains more than one processor, `isrtdxcapable` checks each processor on the target, as defined by the `cc` object, and returns the RTDX capability for each processor on the board. In the returned variable `b`, you find a vector that contains the information for each accessed processor.

Examples Create a link to your C6711 DSK. Test to see if the processor on the board supports RTDX. It should.

```
cc=ccsdsp; %Assumes you have one board and it is the C6711 DSK.
b=isrtdxcapable(cc)
b =
    1
```

Purpose Determine whether target processor is executing process

Syntax `isrunning(cc)`

Description `isrunning(cc)` returns 1 when the target processor is executing a program. When the processor is halted, `isrunning` returns 0.

Using `isrunning` with multiprocessor boards

When your target board contains more than one processor, `isrunning` checks each processor on the target, as defined by the `cc` object, and returns the state for each processor on the board. In the returned variable `b`, you find a vector that contains the information for each accessed processor.

By providing a return variable, as shown here,

```
b = isrunning(cc)
```

`b` contains a vector that holds the information about the state of all processors accessed by `cc`.

Examples

`isrunning` lets you determine whether the target processor is running. After you load a program to the target, use `isrunning` to be sure the program is running before you enable RTDX channels.

```
cc = ccsdsp;

isrunning(cc)

ans =

    0
% Load a program to the target.

run(cc)
isrunning(cc)
```

isrunning

```
ans =  
    1  
  
halt(cc)  
isrunning(cc)  
  
ans =  
    0
```

See Also halt, restart, run

Purpose Determine whether CCS IDE is running

Syntax `isvisible(cc)`

Description `isvisible(cc)` determines whether CCS IDE is running on the desktop and the window is open. If CCS IDE window is open, `isvisible` returns 1. Otherwise, the result is 0 indicating that CCS IDE is either not running or is running in the background.

Examples Test to see if CCS IDE is running. Start by launching CCS IDE. Then open MATLAB. At the prompt, enter

```
cc=ccsdsp
```

```
CCSDSP 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 Object:
```

```
Timeout: 10.00 secs
Number of open channels: 0
```

MATLAB creates a link to CCS IDE and leaves CCS IDE visible on your desktop.

```
isvisible(cc)
```

```
ans =
```

```
1
```

Now, change the visibility state to 0, or invisible, and check the state.

isvisible

```
visible(cc,0)
isvisible(cc)

ans =

     0
```

Notice that CCS IDE is not visible on your desktop. Recall that MATLAB did not open CCS IDE. When you close MATLAB with CCS IDE in this invisible state, CCS IDE remains running in the background. To close it, do one of the following.

- Open MATLAB. Create a new link to CCS IDE. Use the new link to make CCS IDE visible. Close CCS IDE.
- Open Windows Task Manager. Click **Processes**. Find and highlight `cc_app.exe`. Click **End Task**.

See Also

`info`, `visible`

Purpose	Determine whether MATLAB can write to specified memory block
Syntax	<pre>iswritable(cc,address,'datatype',count) iswritable(cc,address,'datatype')</pre>
Description	<p><code>iswritable(cc,address,'datatype',count)</code> returns 1 if MATLAB can write to the memory block defined by the <code>address</code>, <code>count</code>, and <code>datatype</code> input arguments on the processor referred to by <code>cc</code>. When the processor cannot write to any portion of the specified memory block, <code>iswritable</code> returns 0. You use the same memory block specification for this function as you use for the <code>write</code> function.</p> <p>The data block being tested begins at the memory location defined by <code>address</code>. <code>count</code> determines the number of values to write. <code>datatype</code> defines the format of data stored in the memory block. <code>iswritable</code> uses the <code>datatype</code> parameter to determine the number of bytes to write per stored value. For details about each input parameter, read the following descriptions.</p> <p><code>address</code> — <code>iswritable</code> uses <code>address</code> to define the beginning of the memory block to write to. You provide values for <code>address</code> as either decimal or hexadecimal representations of a memory location in the target processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector [<i>location</i>, <i>page</i>], a string, or a decimal value. When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, <code>ccsdsp</code> sets the page to 0 at creation if you omit the page property as an input argument.</p> <p>For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.</p>

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Location is 31 decimal on the page referred to by cc (page)
10	Decimal	Address is 10 decimal on the page referred to by cc (page)
[18,1]	Vector	Address location 10 decimal on memory page 1 (cc (page) = 1)

To specify the address in hexadecimal format, enter the address property value as a string. `iswritable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that when you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc (page)`.

`count` — a numeric scalar or vector that defines the number of `datatype` values to test for being writable. To assure parallel structure with `write`, `count` can be a vector to define multidimensional data blocks. This function always tests a block of data whose size is the total number of elements in matrix specified by the input vector. If `count` is the vector `[10 10 10]`

```
iswritable(cc,31,[10 10 10])
```

`iswritable` writes 1000 values (10*10*10) to the target processor. For a two-dimensional matrix defined with `count` as

```
iswritable(cc,31,[5 6])
```

`iswritable` writes 30 values to the processor.

`datatype` — a string that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the specified `datatype`. `datatype` determines how many bytes to check for each memory value. `iswritable` supports the following data types:

datatype String	Description
<code>double</code>	Double-precision floating point values
<code>int8</code>	Signed 8-bit integers
<code>int16</code>	Signed 16-bit integers
<code>int32</code>	Signed 32-bit integers
<code>single</code>	Single-precision floating point data
<code>uint8</code>	Unsigned 8-bit integers
<code>uint16</code>	Unsigned 16-bit integers
<code>uint32</code>	Unsigned 32-bit integers

`iswritable(cc, address, 'datatype')` returns 1 if the processor referred to by `cc` can write to the memory block defined by the `address`, and `count` input arguments. When the processor cannot write any portion of the specified memory block, `iswritable` returns 0. Notice that you use the same memory block specification for this function as you use for the `write` function. The data block tested begins at the memory location defined by `address`. When you omit the `count` option, `count` defaults to one.

Note `iswritable` relies on the memory map option in CCS IDE. If you did not properly define the memory map for the processor in CCS IDE, this function does not produce useful results. Refer to your Code Composer Studio documentation for more information on configuring memory maps.

iswritable

Like the `isreadable`, `read`, and `write` functions, `iswritable` checks for valid address values. Illegal address values would be any address space larger than the available space for the processor – 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

Examples

When you write scripts to run models in MATLAB and CCS IDE, the `iswritable` function is very useful. Use `iswritable` to check that the channel to which you are writing to is indeed configured properly.

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

% Define read and write channels to the target linked by cc.
open(rx, 'ichannel', 'r');
open(rx, 'ochannel', 'w');
enable(rx, 'ochannel');
enable(rx, 'ichannel');

iswritable(rx, 'ochannel')
ans=
    1
iswritable(rx, 'ichannel')
ans=
    0
```

Now that your script knows that it can write to 'ichanne'1, it proceeds to write messages as required.

See Also

`hex2dec`, `iswritable`, `read`

Purpose Information listings from CCS

Syntax

```
list(ff,varname)
infolist = list(cc,'type')
infolist = list(cc,'type',typename)
```

Description `list(ff,varname)` lists the local variables associated with the function accessed by function object `ff`. Compare to `list(cc,'variable','varname')` which works the same way to return variables from link object `cc`.

Note `list` does not recognize or return information about variables that you declare in your code but that are not used or initialized.

Some restrictions apply when you use `list` with function objects. `list` generates an error in the following circumstances:

- When `varname` is not a valid input argument for the function accessed by `ff`

For example, if your function declaration is

```
int foo(int a)
```

but you request information about input argument `b`, which is not defined

```
list(ff,'b')
```

MATLAB returns an error.

- When `varname` is the same as a variable assigned by MATLAB. Usually this happens when you use `declare` to pass a function declaration to MATLAB and the declaration string does not match the declaration for `ff` as determined when you created `ff`.

In an example that demonstrates this problem, the function declaration has a name for the first input, `a`. In the `declare` call, the declaration string does not provide a name for the first input, just a data type, `int`. When you issue the `declare` call, MATLAB names the first input `ML_Input1`. If you try to use `list` to get information about the input named `ML_Input`, `list` returns an error. Here is the code, starting with the function declaration in your code:

```
int foo(int a) % Function declaration in your source code
declare(ff,'decl','int foo(int)')
% MATLAB generates a warning that it has assigned the name
% ML_Input to the first input argument
list(ff,'ML_Input') % list returns an error for this call
```

- When `varname` does not match the input name in the function declaration provided in your source code, as compared to the declaration string you used in a `declare` operation.

Assume your source code includes a function declaration for `foo`:

```
int foo(int a);
```

Now pass a declaration for `foo` to MATLAB:

```
declare(ff,'decl','int foo(int b)')
```

MATLAB issues a warning that the input names do not match. When you use `list` on the input argument `b`,

```
list(ff,'b')
```

`list` returns an error.

- When `varname` is an input to a library function. `list` always fails in this case. It does not matter whether you use `declare` to provide the declaration string for the library function.

Note When you call `list` for a variable in a function object `list(ff, varname)` the address field may contain an incorrect address if the program counter is not within the scope of the function that includes `varname` when you call `list`.

`infolist = list(cc, type)` reads information about your Code Composer Studio session and returns it in `infolist`. Different types of information and return formats are possible depending on the input arguments you supply to the `list` function call. The `type` argument specifies which information listing to return. To determine the information that `list` returns, use one of the following as the `type` parameter string:

- **project** — Tell `list` to return information about the current project in CCS.
- **variable** — Tell `list` to return information about one or more embedded variables.
- **globalvar** — Tell `list` to return information about one or more global embedded variables.
- **function** — Tell `list` to return details about one or more functions in your project.
- **type** — Tell `list` to return information about one or more defined data types, including `struct`, `enum`, and `union`. C data type typedefs are excluded from the list of data types.

Note, the `list` function returns dynamic Code Composer information that can be altered by the user. Returned listings represent snapshots of the current Code Composer studio configuration only. Be aware that earlier copies of `infolist` might contain stale information.

Also, `list` may report incorrect information when you make changes to variables from MATLAB. To report variable information, `list` uses the CCS API, which only knows about variables in CCS. Your changes from

MATLAB, such as changing the data type of a variable, do not appear through the API and `list`. For example, the following operations return incorrect or old data information from `list`.

Suppose your original prototype is

```
unsigned short tgtFunction7(signed short signedShortArray1[]);
```

After creating the function object `fcnObj`, perform a `declare` operation with this string to change the declaration:

```
unsigned short tgtFunction7(unsigned short signedShortArray1[]);
```

Now try using `list` to return information about `signedShortArray1`.

```
list(fcnObj, 'signedShortArray1')  
  
address: [3442 1]  
location: [1x66 char]  
size: 1  
type: 'short *'  
bitsize: 16  
reftype: 'short'  
referent: [1x1 struct]  
member_pts_to_same_struct: 0  
name: 'signedShortArray1'
```

The `type` field reports the original data type `short`.

You get this is because `list` uses the CCS API to query information about any particular variable. As far as the API is concerned, the first input variable is a `short*`. Changing the declaration does not change anything.

`infolist = list(cc, 'project')` returns a vector of structures containing project information in the format shown here when you specify option type as **project**.

infolist Structure Element	Description
<code>infolist(1).name</code>	Project file name (with path).
<code>infolist(1).type</code>	Project type — <code>project</code> , <code>projlib</code> , or <code>project</code> , refer to <code>new</code>
<code>infolist(1).targettype</code>	String description of target CPU
<code>infolist(1).srcfiles</code>	Vector of structures that describes project source files. Each structure contains the name and path for each source file — <code>infolist(1).srcfiles.name</code>
<code>infolist(1).buildcfg</code>	Vector of structures that describe build configurations, each with the following entries: <ul style="list-style-type: none"> • <code>infolist(1).buildcfg.name</code> — the build configuration name • <code>infolist(1).buildcfg.outpath</code> — the default directory for storing the build output.
<code>infolist(2)....</code>	...
<code>infolist(n)....</code>	...

`infolist = list(cc, 'variable')` returns a structure of structures that contains information on all local variables within scope. The list also includes information on all global variables. Note, however, that if a local variable has the same symbol name as a global variable, `list` returns the information about the local variable.

`infolist = list(cc, 'variable', varname)` returns information about the specified variable `varname`.

`infolist = list(cc, 'variable', varnamelist)` returns information about variables in a list specified by `varnamelist`. The information

list

returned in each structure follows the format below when you specify option type as **variable**:

infolist Structure Element	Description
<code>infolist.varname(1).name</code>	Symbol name
<code>infolist.varname(1).isglobal</code>	Indicates whether symbol is global or local
<code>infolist.varname(1).location</code>	Information about the location of the symbol
<code>infolist.varname(1).size</code>	Size per dimension
<code>infolist.varname(1).uclass</code>	ccsdsp object class that matches the type of this symbol
<code>infolist.varname(1).bitsize</code>	Size in bits. More information is added to the structure depending on the symbol type.
<code>infolist.(varname1).type</code>	data type of symbol
<code>infolist.varname(2)....</code>	...
<code>infolist.varname(n)....</code>	...

`list` uses the variable name as the fieldname to refer to the structure information for the variable.

`infolist = list(cc, 'globalvar')` returns a structure that contains information on all global variables.

`infolist = list(cc, 'globalvar', varname)` returns a structure that contains information on the specified global variable.

`infolist = list(cc, 'globalvar', varnamelist)` returns a structure that contains information on global variables in the list. The returned information follows the same format as the syntax `infolist = list(cc, 'variable', ...)`.

`infolist = list(cc, 'function')` returns a structure that contains information on all functions in the embedded program.

`infolist = list(cc, 'function', functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = list(cc, 'function', functionnamelist)` returns a structure that contains information on the specified functions in `functionnamelist`. The returned information follows the format below when you specify option type as **function**:

infolist Structure Element	Description
<code>infolist.functionname(1).name</code>	Function name
<code>infolist.functionname(1).filename</code>	Name of file where function is defined
<code>infolist.functionname(1).address</code>	Relevant address information such as start address and end address
<code>infolist.functionname(1).funcvar</code>	Variables local to the function
<code>infolist.functionname(1).uclass</code>	ccsdsp object class that matches the type of this symbol — function
<code>infolist.functionname(1).funcdecl</code>	Function declaration — where information such as the function return type is contained
<code>infolist.functionname(1).islibfunc</code>	Is this a library function?
<code>infolist.functionname(1).linepos</code>	Start and end line positions of function

infolist Structure Element	Description
<code>infolist.functionname(1).funcinfo</code>	Miscellaneous information about the function
<code>infolist.functionname(2)...</code>	...
<code>infolist.functionname(n)...</code>	...

To refer to the function structure information, `list` uses the function name as the field name.

`infolist = list(cc, 'type')` returns a structure that contains information on all defined data types in the embedded program. This method includes struct, enum and union data types and excludes typedefs. The name of a defined type is its C struct tag, enum tag or union tag. If the C tag is not defined, it is referred to by the Code Composer Studio compiler as '\$faken' where *n* is an assigned number.

`infolist = list(cc, 'type', typename)` returns a structure that contains information on the specified defined data type.

`infolist = list(cc, 'type', typenamelist)` returns a structure that contains information on the specified defined data types in the list. The returned information follows the format below when you specify option type as **type**:

infolist Structure Element	Description
<code>infolist.typename(1).type</code>	Type name
<code>infolist.typename(1).size</code>	Size of this type
<code>infolist.typename(1).uclass</code>	ccsdsp object class that matches the type of this symbol. Additional information is added depending on the type

infolist Structure Element	Description
infolist.typeName(2)....	...
infolist.typeName(n)....	...

For the field name, list uses the type name to refer to the type structure information.

Note When a variable name, type name, or function name is not a valid MATLAB structure field name, list replaces or modifies the name so it becomes valid.

Note In field names that contain the invalid dollar character \$, list replaces the \$ with DOLLAR.

Note Changing the MATLAB field name does not change the name of the embedded symbol or type.

Examples

This first example shows list used with a variable, providing information about the variable varname. Notice that the invalid field name `_with_underscore` gets changed to `Q_with_underscore`. To make the invalid name valid, list inserts the character Q before the name.

```
varname1 = '_with_underscore'; % invalid fieldname
list(cc, 'variable', varname1);
ans =

    Q_with_underscore : [varinfo]
ans. Q_with_underscore
ans=
```

list

```
        name: '_with_underscore'  
isglobal: 0  
location: [1x62 char]  
    size: 1  
    uclass: 'numeric'  
    type: 'int'  
bitsize: 16
```

To demonstrate using `list` with a defined C type, variable `typename1` includes the type argument. Since valid field names cannot contain the `$` character, `list` changes the `$` to `DOLLAR`.

```
typename1 = '$fake3'; % name of defined C type with no tag  
list(cc,'type',typename1);  
ans =
```

```
        DOLLARfake0 : [typeinfo]
```

```
ans.DOLLARfake0=
```

```
        type: 'struct $fake0'  
        size: 1  
        uclass: 'structure'  
        sizeof: 1  
        members: [1x1 struct]
```

When you request information about a project in CCS, you see a listing like the following that includes structures containing details about your project.

```
projectinfo=list(cc,'project')
```

```
projectinfo =
```

```
        name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'  
        type: 'project'  
targettype: 'TMS320C67XX'
```



```
srcfiles: [69x1 struct]  
buildcfg: [3x1 struct]
```

See Also info

load

Purpose Transfer program file (*.out, *.obj) to target processor in active project

Syntax

```
load(cc, 'filename', timeout)
load(cc, 'filename')
load(cc, 'gelfilename', timeout)
```

Description `load(cc, 'filename', timeout)` loads the file specified by `filename` into the target processor. `filename` can include a full path to a file, or just the name of a file that resides in the Code Composer Studio (CCS) working directory. Use `cd` to check or modify the working directory. Only use `load` with program files that are created by the Code Composer Studio build process.

`timeout` defines the upper limit on how long MATLAB waits for the load process to be complete. If this period is exceeded, `load` returns immediately with a timeout error.

`load(cc, 'filename')` loads the file specified by `filename` into the target processor. `filename` can include a full path to a file, or just the name of a file that resides in the CCS working directory. Use `cd` to check or modify the working directory. Only use `load` with program files that are created by the Code Composer Studio build process. `timeout` defaults to the global value you set when you created link `cc`.

Note `load` disables all open channels. Open channels revert to disabled.

`load(cc, 'gelfilename', timeout)` loads and opens the general extension language (GEL) file named `gelfilename` into CCS, in the active project. `gelfilename` needs to be the full path to the file, or just the file name if the file already shows up in your CCS workspace or project. `load` adds the GEL file to the active project only. To make a different project active so you can add your GEL file to it, use `activate`.

The `timeout` option is not required, as is true for most methods in the product. Using `load` to add a GEL file is identical to using the **File** >

Load GEL... option in Code Composer Studio IDE. Your loaded GEL file appears in the GEL files folder in CCS. To remove GEL files, use `remove`. You can load any GEL file — you must be sure the GEL file is the correct one. `load` does not attempt to verify whether the GEL file is appropriate for your hardware or project.

Examples

Taken from the CCS link tutorial, this code prepares for and loads an object file `filename.out` to a target processor.

```
projfile = ...  
fullfile(matlabroot, 'directoryname', 'directoryname', 'filename')  
projpath = fileparts(projfile)  
open(cc,projfile) % Open project file  
cd(cc,projpath) % Change Code Composer working directory
```

Now use CCS IDE to build your file. Select **Project > Build** from the menu bar in CCS IDE.

With the project build complete, load your `.out` file by entering

```
load(cc, 'filename.out')
```

See Also

`cd`, `dir`, `open`

msgcount

Purpose Number of messages in read-enabled channel queue

Syntax `msgcount(rx, 'channel')`

Description `msgcount(rx, 'channel')` returns the number of unread messages in the read-enabled queue specified by `channel` for the RTDX link `rx`. You cannot use `msgcount` on channels configured for write access.

Examples If you have created and loaded a program to the target processor, you can write data to the target, then use `msgcount` to determine the number of messages in the read queue.

1 Create and load a program to the target.

2 Write data to the target from MATLAB.

```
indata=1:100;
writemsg(cc.rtdx, 'ichannel', int32(indata));
```

3 Use `msgcount` to determine the number of messages available in the queue.

```
num_of_msgs = msgcount(cc.rtdx, 'ichannel')
```

See Also `read`, `readmat`, `readmsg`

Purpose Create and open text file, project, or build configuration in CCS IDE

Syntax
`new(cc, 'objectname', 'type')`
`new(cc, 'objectname')`

Description `new(cc, 'objectname', 'type')` creates and opens an empty object of type named `objectname` in the active project in CCS IDE. The new object can be a text file, a project, or a build configuration. String `objectname` specifies the name of the new object. When you create new text files or projects, `objectname` can include a full path description. When you save your new project or file, CCS IDE stores the file at the target of the full path.

If you do not provide a full path for your file, `new` stores the file in the CCS IDE working directory when you save it. New files open as active windows in CCS IDE; they are not placed in the active project folders based on their file extension (compare to `add`).

New build configurations always become part of the active project in CCS IDE. Since build configurations always become part of a project, you only need to enter a name to distinguish your new configuration from existing configurations in the project, such as Debug and Release.

To specify the text file or project to create, `objectname` must be the full pathname to the file, unless your file is in a directory on your MATLAB path, or the file is in your CCS working directory. Also, when you create new text files or projects, you must include the file extension in `objectname`.

`type` accepts one of the strings or entries listed in the following table.

type String	Description
<code>text</code>	Create a new text file in the active project.
<code>project</code>	Create a new project.

type String	Description
project	Create a new CCS external make project. Using this option indicates that your project uses an external makefile. Refer to your CCS documentation for more information about external projects.
projlib	Create a new library project with the .lib file extension. Refer to your CCS documentation for more information about library projects.
[]	Create a new project. The [] indicate that you are creating a .pjt file.
buildcfg	Create a new build configuration in the active project.

Use new to create the file types listed in the following table.

File Types and Extensions Supported by new and CCS IDE

File Type Created	Supported Extensions	type String Used
C/C++ source files	.c, .cpp, .cc, .ccx, .sa	'text'
Assembly source files	.a*, .s* (excluding .sa, refer to C/C++ source files)	'text'
Object and Library files	.o*, .lib	'text'
Linker command file	.cmd	'text'
Project file	.pjt	'project'
Build configuration	No extension	'buildcfg'

Caution After you create an object in CCS IDE, save the file in CCS IDE. `new` does not automatically save the file. Failing to save the file can cause you to lose your changes when you close CCS IDE.

`new(cc, 'objectname')` creates a project in CCS IDE, making it the active project. When you omit the type option, `new` assumes you are creating a new project and appends the `.pjt` extension to `objectname` to create the project `objectname.pjt`. The `.pjt` extension is the only extension `new` recognizes.

Examples

When you need a new project, create a link to CCS IDE and use the link to make a new project in CCS IDE.

```
cc=ccsdsp;  
cc.visible(1) % Make CCS IDE visible on your desktop (optional).  
new(cc, 'my_new_project.pjt', 'project');
```

New files of various types result from using `new` to create new active windows in CCS IDE. For instance, make a new C source file in CCS IDE with the following command:

```
new(cc, 'new_source.c', 'text');
```

In CCS IDE you see your new file as the active window.

See Also

`activate`, `close`, `save`

open

Purpose Open channel to target processor or load file into CCS IDE

Syntax

```
open(rx, 'channel1', 'mode1', 'channel2', 'mode2', ...)  
open(rx, channel, mode)  
open(cc, filename, filetype, timeout)  
open(cc, filename, filetype)  
open(cc, filename)
```

Description `open(rx, 'channel1', 'mode1', 'channel2', 'mode2', ...)` opens new RTDX channels associated with the link `rx`. Each new channel uses the string name `channel1`, `channel2`, and so on. For each channel, `open` configures the channel according to the associated mode string. `channel1` uses `mode1`; `channel2` uses `mode2`, and so forth. Mode strings are either:

- **r** — Configure the channel to read data from the target processor.
- **w** — Configure the channel for writing data to the target processor.

`open(rx, channel, mode)` opens a new channel to the processor associated with the link `rx`. The new channel uses the `channel` string and is configured for reading or writing according to the `mode` string.

`open(cc, filename, filetype, timeout)` loads `filename` into CCS IDE. `filename` can be the full path to the file or, if the file is in the current CCS IDE working directory, you can use a relative path, such as the name of the file. Use `cd` to determine or change the CCS IDE working directory. You use the `filetype` option to override the default file extension. Four `filetype` strings work in this function syntax.

filetype String	Extension	Description
program	.out	Executable programs for the target processor
project	.c, .a*, .s*, .o*, .lib, .cmd, .mak	CCS IDE project files

filetype String	Extension	Description
text	any	All text files
workspace	.wks	CCS IDE workspace files

To let you determine how long MATLAB waits for open to load the file into CCS IDE, `timeout` sets the upper limit, in seconds, for the period MATLAB waits for the load. If MATLAB waits more than `timeout` seconds, load returns immediately with a timeout error. Returning a timeout error does not suspend the operation; it stops MATLAB from waiting for confirmation for the operation completion.

`open(cc, filename, filetype)` loads `filename` into CCS IDE. `filename` can be the full path to the file or, if the file is in the current CCS IDE working directory, you can use a relative path, such as the name of the file. Use the `cd` function to determine or change your CCS IDE working directory. You use the `filetype` option to override the default file extension. Refer to the previous syntax for more information about `filetype`. When you omit the `timeout` option in this syntax, MATLAB uses the global timeout set in `cc`.

`open(cc, filename)` loads `filename` into CCS IDE. `filename` can be the full path to the file or, if the file is in the current CCS IDE working directory, you can use a relative path, such as the name of the file. Use the `cd` function to determine or change the CCS IDE working directory. You use the `filetype` option to override the default file extension. Refer to the previous syntax for more information about `filetype`. When you omit the `filetype` and `timeout` options in this syntax, MATLAB uses the global timeout set in `cc`, and derives the file type from the extension in `filename`. Refer to the previous syntax descriptions for more information on the input options.

Note Channels must be opened and enabled before you use them. You cannot write to or read from channels that you opened but did not enable.

Note Program files (.out extension) and project files (.mak extension) are loaded on the target processor referenced by your CCS IDE link. Workspace files are coupled to a specific target processor. As a result, open loads workspace files to the target processor that was active when you created the workspace file. This may not be the processor referred to by the CCS IDE link.

Examples

For RTDX use, open forms part of the function pair you use to open and enable a communications channel between MATLAB and your target processor.

```
cc = cc dsp;  
rx = cc.rtdx;  
open(rx, 'ichannel', 'w');  
enable(rx, 'ichannel');
```

When you are working with CCS IDE, open adopts a different operational form based on your input arguments for *filename* and the optional arguments *filetype* and *timeout*. In the CCS IDE variant, open loads the specified file into CCS IDE. For example, to load the tutorial program used in “Getting Started with Links” on page 1-11, use the following syntax

```
cc = cc dsp;  
cc.load(tutorial_6xevm.out);
```

See Also

cd, dir, load

Purpose Profiling information from executing code that includes DSP/BIOS

Syntax

```
ps=profile(cc, 'option', timeout)
ps=profile(cc, 'option')
ps=profile(cc)
```

Description `ps=profile(cc, 'option', timeout)` returns generated code profile measurements from the statistics timing objects (STS) that you defined in CCS IDE. Structure `ps` contains the information in either raw form or filtered and formatted into fields. STS objects are a service provided by the DSP/BIOS real-time kernel that can help you profile and track the way your code runs. For details about STS objects and DSP/BIOS, refer to your Texas Instruments documentation that came with CCS IDE.

To let you to define how to return the information from your STS objects, `profile` supports three formatting options for the contents of structure `ps`.

option String	Description
<code>raw</code>	Returns an unformatted list of the STS timing objects information. All time-based objects get returned and formatted.

option String	Description
report	Returns the same data as the raw option, formatted into an HTML report. Works only on projects that include DSP/BIOS. If you own Embedded Target for TI C6000 DSP, <code>profile(cc, 'report')</code> provides more information about code you generate from Simulink models, using data from the STS objects that are part of DSP/BIOS instrumentation. Refer to “Profiling Code” in your Embedded Target for TI C6000 DSP documentation for more information.
tic	Returns a formatted list of the STS timing objects information. Filters out some of the information returned with the raw option. To be returned by this option, the object must be time-based. User-defined objects are not returned. Use raw to see user-defined objects.

When you choose **raw**, variable `ps` contains an undocumented list of the information provided by CCS IDE. The **tic** option provides the same information in `ps`, as a collection of fields.

Fields in <code>ps</code>	Description
<code>ps.cpuload</code>	Execution time in percent of total time spent out of the idle task.
<code>ps.sts</code>	Vector of defined STS objects in the project.
<code>ps.sts(n).name</code>	User-defined name for an STS object <code>sts(n)</code> . Value for <code>n</code> ranges from 1 to the number of defined STS objects.
<code>ps.sts(n).units</code>	Either Hi Time or Low Time. Describes the timer applied by this STS object, whether high- or low- resolution time based.

Fields in ps	Description
ps, sts(n).max	Maximum measured profile period for sts(n), in seconds.
ps.sts(n).avg	Average measured profile period for sts(n), in seconds.
ps.sts(n).count	Number of STS measurements taken while executing the program.

Note For the information gathered during the reporting periods to be accurate, your CLK and STS must be configured correctly for your target. Use the DSP/BIOS configuration file to add and configure CLK and STS objects for your project.

With projects that you generate that use DSP/BIOS, the `report` option creates a report that contains all of the information provided by the other options, plus additional data that comes from DSP/BIOS instrumentation in the project. You enable the DSP/BIOS report capability with the **Profile performance at atomic subsystem boundaries** option on the TI C6000 Code Generation option on the **Real-Time Workshop** pane of the Simulink Configuration Parameters dialog box.

`ps=profile(cc, 'option')` defaults to the timeout period specified in the link `cc`.

`ps=profile(cc)` returns the profile information in `ps` as a formatted structure of fields.

Example

You use `profile` to view information about your application running on your target. This example presents both forms of the data returned in `ps`. Open and build one of the DSP/BIOS-enabled projects from the TI DSP/BIOS Tutorial Module, such as `volume.pjt` located in the folder `ti\tutorial\target\volume2`. When you specify the project to open, enter the full path name to the project file.

profile

```
cc=ccsdsp;  
open(cc, '..\tutorial\sim62xx\volume2\volume.pjt');  
build(cc, 'all')
```

In CCS IDE, open the file `volume.cdb` that contains the DSP/BIOS configuration. For details about STS and CLK objects, refer to your TI documentation.

Review the settings for the existing CLK and STS objects already in place in the project. When you use `profile`, the information returned comes from these objects. Make any changes you require and save the DSP/BIOS configuration file. Now rebuild your project, either in CCS IDE or from MATLAB, then load the file `volume.out` generated by the build process. If you get a timeout error, add the `timeout` option to the build command, specifying a long timeout period, such as 60 seconds. Often, when you receive the time out error the build has been completed successfully.

```
build(cc, 'all')  
load(cc, '..\tutorial\sim62xx\volume2\debug\volume.out')
```

With the project built and loaded, run your program.

```
run(cc) % Assumes that volume2 is the active project.
```

Running `profile` returns structure `ps` containing STS and CLK information that DSP/BIOS gathered while your program ran.

```
ps=profile(cc)  
  
ps =  
  
    cpuload: 0  
      obj: [3x1 struct]  
  
ps.obj(1)  
  
ans =
```

```
        name: 'KNL_swi'  
        units: 'Hi Time'  
          max: 1.1759e-005  
          avg: 2.7597e-006  
        count: 29  
  
    for k=1:length(ps.obj),disp(k),disp(ps.obj(k)),end;  
    1  
  
        name: 'KNL_swi'  
        units: 'Hi Time'  
          max: 1.1759e-005  
          avg: 2.7597e-006  
        count: 29  
  
    2  
  
        name: 'processing_SWI'  
        units: 'Hi Time'  
          max: 1.1489e-005  
          avg: 1.1474e-005  
        count: 2  
  
    3  
  
        name: 'TSK_idle'  
        units: 'Hi Time'  
          max: -16.1465  
          avg: 0  
        count: 0
```

Omitting the format option caused profile to return the data fully formatted and slightly filtered. Adding the raw option to profile returns the same information without filtering any of the returned data.

```
ps=profile(cc,'raw')
```

profile

```
ps =  
  
    cpload: 0  
    error: 0  
    avgperiod: 1000  
    rate: 1000  
    obj: [4x1 struct]  
  
for k=1:length(ps.obj),disp(k),disp(ps.obj(k)),end;  
1  
  
    name: 'KNL_swi'  
    units: 'Hi Time'  
    max: 1564  
    total: 10644  
    avg: 367.0345  
    pdfactor: 0.0075  
    count: 29  
  
2  
  
    name: 'processing_SWI'  
    units: 'Hi Time'  
    max: 1528  
    total: 3052  
    avg: 1526  
    pdfactor: 0.0075  
    count: 2  
  
3  
  
    name: 'TSK_idle'  
    units: 'Hi Time'  
    max: -2.1475e+009  
    total: 0  
    avg: 0
```



```
pdfactor: 0.0075
count: 0

4

    name: 'IDL_busyObj'
    units: 'User Def'
    max: -2.1475e+009
    total: 0
    avg: 0
pdfactor: 0
count: 0
```

Your results can differ from this example depending on your computer and target. In the raw data in this example, one extra timing object appears — `IDL_busyObj`. As defined in the `.cdb` file, this is not a time based object (**Units** is `'User Def'`) and is not returned by specifying `tic` as the format option in `profile`.

See Also

`ccsdsp`

read

Purpose

Data from memory on target processor or in CCS

Syntax

```
mem = read(cc,address,count,'datatype',timeout)
mem = read(cc,address,'datatype',count)
mem = read(cc,address,'datatype')
data = read(objname,structindex)
data = read(objname,structindex,member)
data = read(objname,member)
data = read(objname,structindex,member,memberindex)
data = read(objname)
data = read(objname,index)
data = read(objname,member,memberindex,structindex)
data = read(...,timeout)
```

Description

Link Object Syntaxes

`mem = read(cc,address,count,'datatype',timeout)` returns data from the processor referred to by `cc`. The `address`, `count`, and `datatype` input arguments define the memory block to be read. The data block to read begins at the memory location defined by `address`. `count` determines the number of values to read, starting at `address`. `datatype` defines the format of the raw data stored in the referenced memory block.

Note Do not attempt to read data from the target while it is running.

To check values in memory on a running target, such as values that change during processing, insert one or more breakpoints in the project code and perform the read operation while the target code is paused at one of the breakpoints. After you read the data, release the breakpoint.

`read` uses the `datatype` parameter to determine the number of bytes to read per stored value. `timeout` is an optional input argument you use to specify when to terminate long read processes and data transfers. For details about each input parameter, read the following descriptions.

`address` — `read` uses `address` to define the beginning of the memory block to read. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the target processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector `[location, page]`, a string, or a decimal value. When the processor has only one memory page, as is true for many digital signal processors, the value of the page portion of the memory address is 0. By default, `ccsdsp` sets the page to 0 at creation if you omit the page property as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Offset is 31 decimal on the page referred to by <code>cc(page)</code>
10	Decimal	Offset is 10 decimal on the page referred to by <code>cc(page)</code>
[18,1]	Vector	Offset is 18 decimal on memory page 1 (<code>cc(page) = 1</code>)

To specify the address in hexadecimal format, enter the address property value as a string. `read` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that when you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc(page)`.

`count` — a numeric scalar or vector that defines the number of datatype values to read. Entering a scalar for `count` causes `read` to return `mem` as a column vector which has `count` elements. `count` can be a vector to define multidimensional data blocks. The elements of `count` define

read

the dimensions of the data matrix returned in mem. The following table shows examples of input arguments to count and how read responds.

Input	Response
n	Read n values into a column vector. Return the vector in mem.
[m,n]	Read (m*n) values from memory into an m-by-n matrix in column major order. Return the matrix in mem.
[m,n,p,...]	Read (m*n*p*...) values from the processor memory in column major order. Return the data in an m-by-n-by-p-by... multidimensional matrix and return the matrix in mem.

`datatype` — a string that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the specified `datatype`. `datatype` determines how many bytes to check for each memory value. `read` supports the following data types:

datatype String	Description
double	Double-precision floating point values
int8	Signed 8-bit integers
int16	Signed 16-bit integers
int32	Signed 32-bit integers
single	Single-precision floating point data
uint8	Unsigned 8-bit integers
uint16	Unsigned 16-bit integers
uint32	Unsigned 32-bit integers

To limit the time that `read` spends transferring data from the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. Time out is defined as the number of

seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `read` defaults to the global timeout defined in `cc`.

Working With Negative Values

Writing a negative value causes the data written to be saturated because `char` is unsigned on the processor. Hence, a 0 (a `NULL`) is written instead. A warning results as well, as this example shows.

```
cc = ccstdsp;
ff = createobj(cc,'g_char'); % Where g_char is in the code.
write(ff,-100);
Warning: Underflow: Saturation was required to fit the data into
an addressable unit.
```

When you try to read the data you wrote, the character being read is 0 (`NULL`) — so there seems to be nothing returned by the `read` function.

You can demonstrate this by the following code, after writing a negative value as shown in the previous example.

```
readnumeric(x)
ans =
0
read(x) % Reads the NULL character
ans = % Apparently nothing is returned.

double(read(x)) % Read the numeric equivalent of NULL.
ans = % Again, appears not to return a value.
```

`mem = read(cc,address,'datatype',count)` reads data from memory on the processor referred to by `cc` and defined by the `address`, and `datatype` input arguments. The data block being read begins at the memory location defined by `address`. `count` determines the number of values to be read. When you omit the `timeout` option, `timeout` defaults to the value specified by the `timeout` property in `cc`.

`mem = read(cc, address, 'datatype')` reads the memory location defined by the address input argument from the processor memory referred to by `cc`. The data block being read begins at the memory location defined by `address`. When you omit the count option, count defaults to a value of 1. This syntax reads one memory location of `datatype`.

Note `read` does not coerce data type alignment in your processor memory. You can write and read data of any type (`datatype`) to and from any memory location (`address`). Certain combinations of address and `datatype` are difficult for some processors to use. To ensure seamless read operation, use the `address` function to extract address values that are compatible with the alignment required by your target processor.

Like the `isreadable`, `iswritable`, and `write` functions, `read` checks for valid address values. Illegal address values are any address space larger than the available space for the processor — 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When `read` identifies an illegal address, it returns an error message stating that the address values are out of range.

Reading Structures

Reading data from structures in memory represents a special subset of general `read`. In each syntax, `objname` accesses a structure in memory on the target or in CCS.

`data = read(objname, structindex)` reads the structure element referred to by `structindex`.

`data = read(objname, structindex, member)` returns the value of the specified member of the structure as identified by `structindex`.

`data = read(objname, member)` returns the value of `member` from the structure accessed by `objname`, for all indexes — the entire structure variable.

`data = read(objname,structindex,member,memberindex)` returns the index for member in the accessed structure.

Embedded Object Syntaxes

`read` works with all of the objects you create with `createobj`. To transfer data from Code Composer Studio to MATLAB, use the `read` function — `read` — depending on the data to access. Note that `read` and its variants are the only way to get data from CCS to MATLAB as objects.

`data = read(objname)` reads all the data in memory at the location accessed by object `objname`, and converts the data into a numeric representation. Properties of `objname`, such as `wordsize`, `storageunitspervalue`, `size`, `represent`, and `binarypt`, determine how `read` performs the numeric conversion. `data` is a numeric array whose dimensions are defined by the `size` property of `objname`. Object property `size` is the *dimensions* vector. Each element in the *dimensions* vector contains the size of the data array in that dimension. When `size` is a scalar, `data` is a column vector of the length specified by `size`.

For example, when `size` is `[2 3]`, `data` is a 2-by-3 array.

Properties of the Object

`objname`, the object that accesses the data, has the following properties, if the object is a numeric object. The properties differ for different types of objects, such as structure objects or register objects.

Property	Options	Description
<code>size</code>	Greater than 1	Specifies the dimensions of the output numeric array.

Property	Options	Description
arrayorder	col-major or row-major	Defines how to map sequential memory locations into arrays. col-major is the default, and the MATLAB standard. C uses 'row-major' ordering most often.
represent	float, signed, unsigned, fract	Determines the numeric representation used in the output data. <ul style="list-style-type: none">• float — IEEE floating point representation, either 32- or 64 bits• signed — two's complement signed integers• unsigned — unsigned binary integer• fract — fractional fixed-point data
wordsize	Greater than 1	(Read-only) Calculated from other object properties such as storageunitspvalue
binarypt	0 to wordsize	Determines the position of the binary point in a word to specify its interpretation

`data = read(objname,index)` reads the specified element in the memory location accessed by `objname`. `index` is a scalar or a vector that identifies the particular data element to return. When you enter `[]` for `index`, `read` returns all the data stored at the memory location. When you enter a scalar for `index`, `read` returns a column vector of length size containing the data from the memory space. When `index` is a vector, `read` returns the element in the array specified by the entries in the vector. For example, if you are reading data from a 3-by-3-by-3 array, setting `index` to be `[2 2 2]` returns the element `data(2,2,2)`. To return more than one element, use MATLAB standard range notation for the vector elements in `index`. As an example, when `index` is `[1:6]`, `read` returns the first six elements of `data`. You must remember that the number of elements in the vector in `index` must be either one (a scalar) or the same as the number of dimensions in `data` and specified by the property size. When `data` is a four dimensional array, your vector in `index` must have four elements, one for each array dimension. Otherwise, `read` cannot determine which elements to return.

`data = read(objname,member,memberindex,structindex)` reads the members of the structure that `objname` accesses. When you omit all of the input arguments except `objname`, `read` returns the entire structure. `member`, `memberindex`, and `structindex` (an optional input argument) specify which structure member to read:

- `member` — Specifies the name of the member of the structure to read.
- `memberindex` — Provides the index of the data element to read.
- `structindex` — Identifies the structure to read when `objname` accesses a structure containing structures or a vector.

Note that the class of the object data from the `read` operation depends on the class of the member being read — numeric values return numeric objects, string values return string objects, and so on.

`data = read(...,timeout)` During `read` operations, the `timeout` property of `objname` determines the time allowed to complete the read. Including a value for the `timeout` input argument in the `read` syntax lets you override the `timeout` property setting for `objname` with the

value you enter for argument `timeout`. For reading large data arrays, being able to explicitly set the `timeout` value as an input option may be necessary to let `read` run to completion. Note that using the `timeout` input option does not change the `timeout` property value for `objname`.

When you need to read one member of a structure or perform individual read operations, consider using `getmember`.

Examples

`read` reads data that you wrote to the target processor.

```
cc = ccscdsp;  
indata = 1:25;  
write(cc,0,indata,30);  
outdata=read(cc,0,25,'double',10)
```

```
outdata =
```

```
Columns 1 through 13
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13
```

```
Columns 14 through 25
```

```
14 15 16 17 18 19 20 21 22 23 24 25
```

`outdata` now contains the values in `indata`, returned from the target processor.

As a further demonstration of `read`, try the following functions after you create a link `cc` and load an appropriate program to your target. To perform the first example, `var` must exist in the symbol table loaded in CCS.

- Read one 16-bit integer at the location of target symbol `var`.

```
mlvar = read(cc,address(cc,'var'),'int16')
```

- Read 100 32-bit integers from address f000 (hexadecimal) and plot the data.

```
mlplt = read(cc, 'f000', 'int32', 100)
plot(double(mlplt))
```

- Increment the integer value stored at address 10 (decimal) of the target processor.

```
cc = ccstdsp;
ainc = 10
mlinc = read(cc, ainc, 'int32')
mlinc = int32(double(mlinc)+1)
cc.write(ainc, mlinc)
```

Reading String Variables

Using `read` to return a string creates a string object. Within the string object, the property `charconversion` controls the read operation. When you set `charconversion` to `ASCII`, `read` recognizes only the ASCII characters from 0 to 127. `ASCII` is the only accepted type for the `charconversion` property value.

While reading strings from memory, `read` continues until it encounters a null character, then it stops.

For example, if memory contains the string “Hello World” in the following format in memory (each block represents one memory location)

H	e	l	l	o		W	o	r	l	d	\0	M
---	---	---	---	---	--	---	---	---	---	---	----	---

`read` does not return the `M` because it stops at the null character `\0`.

To return a string from memory as a numeric object in MATLAB, use `readnumeric`.

Reading Enumerated Variables

If you read an enumerated data type from memory, the returned entry is a string object.

Reading Structures

The following examples show various structure read syntaxes at work. Start with a structure definition.

```
struct tag {  
    float re;  
    float im;  
} st[2] = {1,2,3,4};
```

Use read to return the information stored in the structure st.

```
st = createobj(cc, 'st')  
  
answer = read(st)  
ans =  
    [1x2 struct]  
  
answer{1}  
  
    re: 1  
    im: 2  
answer{2}  
  
    re: 3  
    im: 4  
answer = read(st,2)  
  
answer =  
    re: 3  
    im: 4  
answer = read(st,2,'re')  
  
answer =  
    3  
answer = read(st,'re')  
answer =
```

1 3

See Also getmember, isreadable, symbol, write

readbin

Purpose Data from DSP memory as binary strings

Syntax

```
data = readbin(nn)
data = readbin(nn,[],timeout)
data = readbin(nn,index)
data = readbin(nn,...)
```

Description `data = readbin(nn)` returns, as binary strings, the data accessed by numeric object `nn`. When `nn` refers to an array, the returned values stored in `data` compose a cell array of binary strings. If the `size` property of `nn` is 1, indicating that `nn` accesses a scalar, the output in `data` is an array of binary characters.

`data = readbin(nn,[],timeout)` returns all the values accessed by `nn`, as indicated by the empty square brackets, `[]`. During read operations, the `timeout` property of `nn` determines the time allowed to complete the read. Including a value for the `timeout` input argument in the `readbin` syntax lets you override the `timeout` property setting for `nn` with the value you enter for argument `timeout`. For reading large data arrays, being able to explicitly set the `timeout` value as an input option may be necessary to let `readbin` run to completion. Note that using the `timeout` input option does not change the `timeout` property value for `nn`.

`data = readbin(nn,index)` returns the data element at the index specified by `index` from the data block accessed by `nn`. Enter `index` as a scalar to return one data value, or a vector to return two or more values from the data array.

`data = readbin(nn,...)` returns binary strings as specified by your combination of all the optional input arguments.

See Also `cast`, `read`, `write`, `writebin`

Purpose	Data from DSP memory as hexadecimal strings
Syntax	<pre>data = readhex(nn) data = readhex(nn,[],timeout) data = readhex(nn,index) data = readhex(nn,...)</pre>
Description	<p><code>data = readhex(nn)</code> returns, as hexadecimal strings, the data accessed by numeric object <code>nn</code>. When <code>nn</code> refers to an array, the returned values stored in <code>data</code> compose a cell array of hexadecimal strings. If the <code>size</code> property of <code>nn</code> is 1, indicating that <code>nn</code> accesses a scalar, the output in <code>data</code> is an array of hexadecimal characters.</p> <p><code>data = readhex(nn,[],timeout)</code> returns all the values accessed by <code>nn</code>, as indicated by the empty square brackets, <code>[]</code>. During read operations, the <code>timeout</code> property of <code>nn</code> determines the time allowed to complete the read. Including a value for the <code>timeout</code> input argument lets you override the <code>timeout</code> property setting for <code>nn</code> with the value you enter for argument <code>timeout</code>. For reading large data arrays, setting the <code>timeout</code> value as an input option may be necessary to let <code>readhex</code> run to completion. Note that using the <code>timeout</code> input option does not change the <code>timeout</code> property value for <code>nn</code>.</p> <p><code>data = readhex(nn,index)</code> returns the data element at the index specified by <code>index</code> from the data block accessed by <code>nn</code>. Enter <code>index</code> as a scalar to return one data value, or a vector to return two or more values from the data array.</p> <p><code>data = readhex(nn,...)</code> returns hexadecimal strings as specified by your combination of all the optional input arguments.</p>
See Also	<code>cast</code> , <code>readbin</code> , <code>write</code>

readmat

Purpose Matrix of data from RTDX channel

Syntax

```
data = readmat(rx,channelname,'datatype',siz,timeout)
data = readmat(rx,channelname,'datatype',siz)
```

Description `data = readmat(rx,channelname,'datatype',siz,timeout)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Before you read from a channel, open and enable the channel for read access.

Replace `channelname` with the string you specified when you opened the desired channel. `channelname` must identify a channel that you defined in the program loaded on the target processor.

You cannot read data from a channel you have not opened and configured for read access. If necessary, use the RTDX tools provided in CCS IDE to determine which channels exist for the loaded program.

`data` contains a matrix whose dimensions are given by the input argument vector `siz`, where `siz` can be a vector of two or more elements. To operate properly, the number of elements in the output matrix `data` must be an integral number of channel messages.

When you omit the `timeout` input argument, `readmat` reads messages from the specified channel until the output matrix is full or the global timeout period specified in `rx` elapses.

Caution If the timeout period expires before the output data matrix is fully populated, you lose all the messages read from the channel to that point.

MATLAB supports reading five data types with `readmat`:

datatype String	Data Format
double	Double-precision floating point values. 64 bits.
int16	16-bit signed integers
int32	32-bit signed integers
single	Single-precision floating point values. 32 bits.
uint8	Unsigned 8-bit integers

`data = readmat(rx,channelname,'datatype',siz)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Before you read from a channel, open and enable the channel for read access. Replace `channelname` with the string you specified to open and enable the desired channel.

You cannot read data from a channel you have not opened and configured for read access.

`data` contains a matrix whose dimensions are given by the input argument vector `siz`, where `siz` can be a vector of two or more elements. To operate properly, the number of elements in the output matrix `data` must be an integral number of channel messages.

When you include the `timeout` input argument, `readmat` reads messages from the specified channel until the output matrix is full or the `timeout` period elapses.

Caution If the timeout period expires before the output data matrix is fully populated, you lose all the messages read from the channel to that point.

MATLAB supports reading five data types with `readmat`:

datatype String	Data Format
double	Double-precision floating point values, 64 bits.
int16	16-bit signed integers.
int32	32-bit signed integers.
single	Single-precision floating point values. 32 bits.
uint8	Unsigned 8-bit integers.

Examples

In this data read and write example, you write data to the target through the CCS IDE. You can then read the data back in two ways — either through `read` or through `readmsg`.

To duplicate this example you need to have a program loaded on the target. The channels listed in this example, `ichannel` and `ochannel`, must be defined in the loaded program. If the current program on the target defines different channels, replace the listed channels with your current ones.

```
cc = ccstdsp;
rx = cc.rtdx;
open(rx, 'ichannel', 'w');
enable(rx, 'ichannel');
open(rx, 'ochannel', 'r');
enable(rx, 'ochannel');
indata = 1:25; % Set up some data.
write(cc,0,indata,30);
outdata=read(cc,0,25, 'double', 10)
```

```
outdata =
```

```
Columns 1 through 13
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13
```

```
Columns 14 through 25
```

14 15 16 17 18 19 20 21 22 23 24 25

Now use RTDX to read the data into a 5-by-5 array called `out_array`.

```
out_array = readmat('ochannel', 'double', [5 5])
```

See Also

`readmsg`, `writemsg`

readmsg

Purpose Read messages from specified RTDX channel

Syntax

```
data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)
data = readmsg(rx,channelname,'datatype',siz,nummsgs)
data = readmsg(rx,channelname,datatype,siz)
data = readmsg(rx,channelname,datatype,nummsgs)
data = readmsg(rx,channelname,datatype)
```

Description

`data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read all the messages present in the queue when you call the `readmsg` function. Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. For example, when `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `m`-by-`n` matrices in `data`. Each output matrix in `data` must have the same number of elements (`m x n`) as the number of elements in each message. You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports strings that define the type of data you are expecting, as shown in the following table.

datatype String	Specified Data Type
<code>double</code>	Floating point data, 64-bits (double-precision).
<code>int16</code>	Signed 16-bit integer data.
<code>int32</code>	Signed 32-bit integers.
<code>single</code>	Floating-point data, 32-bits (single-precision).
<code>uint8</code>	Unsigned 8-bit integers.

When you include the `timeout` input argument in the function, `readmsg` reads messages from the specified queue until it receives `nummsgs`, or until the period defined by `timeout` expires while `readmsg` waits for more messages to be available. When the desired number of messages is not available in the queue, `readmsg` enters a wait loop and stays there until more messages become available or `timeout` seconds elapse. The `timeout` argument overrides the global `timeout` specified when you create `rx`.

`data = readmsg(rx,channelname,'datatype',siz,nummsgs)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read all the messages present in the queue when you call the `readmsg` function. Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. When `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `n`-by-`m` matrices in `data`. Each output matrix in `data` must have the same number of elements (`m x n`) as the number of elements in each message. You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports six strings that define the type of data you are expecting.

`data = readmsg(rx,channelname,datatype,siz)` reads one data message because `nummsgs` defaults to one when you omit the input argument. `readmsg` returns the message as a row vector in `data`.

`data = readmsg(rx,channelname,datatype,nummsgs)` reads the number of messages defined by `nummsgs`. `data` becomes a cell array of row matrices, `data = {msg1,msg2,...,msg(nummsgs)}`, because `siz` defaults to `[1,nummsgs]`; each returned message becomes one row matrix in the cell array. Each row matrix contains one element for each data value in the current message `msg# = [element(1),element(2),...,element(1)]` where `1` is the number of data elements in message. In this syntax, the read messages can have different lengths, unlike the previous syntax options.

readmsg

`data = readmsg(rx,channelname,datatype)` reads one data message, returning a row vector in `data`. All of the optional input arguments — `nummsgs`, `siz`, and `timeout` — use their default values.

In all calling syntaxes for `readmsg`, you can set `siz` and `nummsgs` to empty matrices, causing them to use their default values — `nummsgs = 1` and `siz = [1,l]`, where `l` is the number of data elements in the read message.

Caution If the timeout period expires before the output data matrix is fully populated, you lose all the messages read from the channel to that point.

Examples

```
cc = ccsdsp;
rx = cc.rtdx;
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(cc,0,indata,30);
outdata=read(cc,0,25,'double',10)
```

```
outdata =
```

```
Columns 1 through 13
```

```
1    2    3    4    5    6    7    8    9   10   11   12   13
```

```
Columns 14 through 25
```

```
14   15   16   17   18   19   20   21   22   23   24   25
```

Now use `RTDX` to read the messages into a 4-by-5 array called `out_array`.

```
number_msgs = msgcount(rx,'ochannel') % Check number of msgs
                                         % in read queue.
out_array = cc.rtdx.readmsg('ochannel','double',[4 5])
```

See Also

read, readmat, writemsg

readnumeric

Purpose Read object in memory and convert to numeric equivalent in MATLAB

Syntax

```
data = readnumeric(objname)
data = readnumeric(objname,index)
data = readnumeric(...,timeout)
```

Description `data = readnumeric(objname)` returns all data from the memory area specified by `objname` and converts it into a numeric representation. The properties of `objname` control the numeric conversion process. Output `data` is a numeric array that has dimensions defined by `objname.size`, which is the dimensions array. Each element in the dimensions array specifies the size of the `objname` array in that dimension. When `size` is a scalar, `data` is a column vector of the specified length.

`data = readnumeric(objname,index)` returns a subset of the numeric values from the numeric array specified by `objname`. Each row of `index` is applied as a subscript into the full `objname` array. Output `data` composes a column vector with one value per entry in the `data`. Array indices start at one and range up to the maximum value defined by the value of the property `size` for `objname`.

When `index` is a vector, each row is a single index that defines one entry from the defined numeric array. `data` is a column vector of values corresponding to the specified indices. You can pass a new `timeout` value to modify temporarily the default `timeout` property of `objname`.

`data = readnumeric(...,timeout)` adds the optional input argument `timeout` that lets you specify how long MATLAB waits for the `readnumeric` operation to return a completion message. When MATLAB does not receive notification that the operation finished within the allotted time, you get a time-out error. You may find that the operation did complete successfully in spite of the error message.

objname Array Properties

Property Name	Description
<code>objname.size</code>	Dimensions of output numeric array. This defines the size of the output.
<code>objname.arrayorder</code>	Defines how sequential memory locations are mapped into matrices in MATLAB. The default is column major ordering, which is the default arrangement in MATLAB. Alternatively, you can use row major ordering, which is the memory organization used in C numeric representations.
<code>objname.represent</code>	Defines the numeric representation in <code>objname</code> . Valid data types for <code>represent</code> are: <ul style="list-style-type: none"> • <code>float</code> — IEEE floating point representation (32 or 64 bits) • <code>signed</code> — Two's complement signed integers • <code>unsigned</code> — Unsigned binary integers • <code>fract</code> — Fractional fixed-point representation. For more information, refer to <code>objname.p</code>
<code>objname.wordsize</code>	Number of valid bits in the numeric representation. <code>wordsize</code> is computed from other properties such as <code>storageunitspvalue</code> and therefore this property is read-only.
<code>objname.binarypt</code>	Other properties of <code>objname</code> control the placement and arrangement of the numeric values in memory.

readnumeric

Changes to the numeric representation are possible by modifying the class properties. However, the `convert` method enables you to adjust the property values to implement some common data types.

See Also

`convert`, `getmember`, `read`, `write`

Purpose Value from target processor register

Syntax

```
reg = regread(cc,'regname','represent',timeout)
reg = regread(cc,'regname','represent')
reg = regread(cc,'regname')
```

Description `reg = regread(cc,'regname','represent',timeout)` reads the data value in the `regname` register of the target processor and returns the value in `reg` as a double-precision value. For convenience, `regread` converts each return value to the MATLAB double datatype independent of the datatype defined by `represent`. Making this conversion lets you manipulate the data in MATLAB. String `regname` specifies the name of the source register on the target. `ccsdsp` object `cc` defines the target to read from. Valid entries for `regname` depend on your target processor. Register names are not case-sensitive — `a0` is the same as `A0`.

For example, the TMS320C6xxx processor family provides the following register names that are valid entries for `regname`:

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTEP, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

regread

Note Use `read` (called a direct memory read) to read memory-mapped registers. For the TMS320C5xxx processor family, register PC is memory mapped and thus available using `read`, not `regread`. Use `regread` to read from all other registers.

The `represent` input argument defines the format of the data stored in `regname`. Input argument `represent` takes one of three input strings:

represent string	Description
2scomp	Source register contains a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
binary	Source register contains an unsigned binary integer.
ieee	Source register contains a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are reading from 32 and 64 bit registers on the target.

To limit the time that `regread` spends transferring data from the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `regread` defaults to the global `timeout` defined in `cc`.

`reg = regread(cc, 'regname', 'represent')` reads the data value in the `regname` register of the target processor and returns the value in `reg` as a double-precision value. String `regname` specifies the name of the source register on the target. Link `cc` defines the target to read from. For convenience, `regread` converts each return value to the MATLAB `double` datatype independent of the datatype defined by `represent`. Making this conversion lets you manipulate the data in

MATLAB. The represent input argument defines the format of the data stored in regname.

`reg = regread(cc, 'regname')` reads the data value in the regname register of the target processor and returns the value in reg. String regname specifies the name of the source register on the target. Link cc defines the target to read from. For convenience, regread converts each return value to the MATLAB double datatype independent of the datatype of the source. Making this conversion lets you manipulate the data in MATLAB.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations at any time during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables during intermediate times in program operation may not get reflected in the register.

This is true for local variables as well.

One way to see this is to write a line of code that uses the variable and see if the result is consistent.

```
register int a = 100;
int b;
...
b = a + 2;
```

Reading the register assigned to a may return an incorrect value for a but if b returns the expected 102 result, nothing is wrong with the code or the Link for CCS.

regread

Examples

For the C5xxx processor family, most registers are memory-mapped and consequently are available using `read` and `write`. However, the PC register is not memory-mapped. The following command demonstrates how to read the PC register. To identify the target, `cc` is a link for CCS IDE.

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

To tell MATLAB what datatype you are reading, the string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

In response, MATLAB displays

```
ans =  
  
33824
```

For processors in the C6xxx family, `regread` lets you access processor registers directly. To read the value in general purpose register A0, type the following function.

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

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

Now read the value stored in register B2 as an unsigned binary integer, by typing

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

See Also

`read`, `regwrite`, `write`

Purpose Write data values to registers on target processor

Syntax

```
regwrite(cc, 'regname', value, 'represent', timeout)
regwrite(cc, 'regname', value, 'represent')
regwrite(cc, 'regname', value,)
```

Description `regwrite(cc, 'regname', value, 'represent', timeout)` writes the data in `value` to the `regname` register of the target processor. `regwrite` converts `value` from its representation in the MATLAB workspace to the representation specified by `represent`. The `represent` input argument defines the format of the data when it is stored in `regname`. Input argument `represent` takes one of three input strings:

represent string	Description
2scomp	Write value to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
binary	Write value to the destination register as an unsigned binary integer.
ieee	Write value to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the target.

String `regname` specifies the name of the destination register on the target. Link `cc` defines the target to write value to. Valid entries for `regname` depend on your target processor. Register names are not case-sensitive — `a0` is the same as `A0`. For example, the TMS320C6xxx processor family provides the following register names that are valid entries for `regname`:

regwrite

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTR, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

Note Use `write` (called a direct memory write) to write memory-mapped registers. For the TMS320C5xxx processor family, register PC is memory mapped and thus available using `write`, not `regwrite`. Use `regwrite` to write to all other registers.

To limit the time that `regwrite` spends transferring data to the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You might find this useful for limiting prolonged data transfer operations.

If you omit the `timeout` input argument in the syntax, `regwrite` defaults to the global timeout defined in `cc`. If the write operation exceeds the time specified, `regwrite` returns with a timeout error. Generally, timeout errors do not stop the register write process. The write process stops while waiting for CCS IDE to respond that the write operation is complete.

`regwrite(cc, 'regname', value, 'represent')` writes the data in `value` to register `regname` of the target processor. `regwrite` converts `value` from its representation in the MATLAB workspace to the representation specified by `represent`. The `represent` input argument

defines the data format when it is stored in regname. Input argument represent takes one of the following input strings:

represent string	Description
2scomp	Write value to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the represent argument.
binary	Write value to the destination register as an unsigned binary integer.
ieee	Write value to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the target.

String regname specifies the name of the destination register on the target. Link cc defines the target to write value to. Valid entries for regname depend on your target processor. Register names are not case-sensitive — a0 is the same as A0. For example, the TMS320C6xxx processor family provides the following register names that are valid entries for regname:

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTP, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

regwrite

Note Use `write` (called a direct memory write) to write memory-mapped registers. For the TMS320C5xxx processor family, register PC is memory mapped and thus available using `write`, not `regwrite`. Use `regwrite` to write to all other registers.

`regwrite(cc, 'regname', value,)` writes the data in `value` to the `regname` register of the target processor. `regwrite` converts `value` from its representation in the MATLAB workspace to the representation specified by `represent`. The `represent` input argument defines the format of the data when it is stored in `regname`. Input argument `represent` takes one of the following input strings:

represent string	Description
2scomp	Write <code>value</code> to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
binary	Write <code>value</code> to the destination register as an unsigned binary integer.
ieee	Write <code>value</code> to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the target.

String `regname` specifies the name of the destination register on the target. Link `cc` defines the target to write `value` to. Valid entries for `regname` depend on your target processor. Register names are not case-sensitive — `a0` is the same as `A0`. For example, the TMS320C6xxx processor family provides the following register names that are valid entries for `regname`:

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTR, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

When you omit the `represent` argument, `regwrite` takes value from the function and writes it to the designated register as a two's complement value signed integer.

Note Use `write` (called a direct memory write) to write to memory-mapped registers. For the TMS320C5xxx processor family, register PC is memory mapped and thus available using `write`, not `regwrite`. Use `regwrite` to write to all other registers.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations at any time during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables during intermediate times in program operation may not get reflected in the register.

regwrite

This is true for any local variables as well.

One way to see this is to write a line of code that uses the variable and see if result is consistent.

```
register int a = 100;
int b;
...
b = a + 2;
```

Reading the register assigned to a may return an incorrect value for a but if b returns the expected 102 result, nothing is wrong with the code or the Link for CCS.

Examples

To write a new value to the PC register on a C5xxx family processor, type

```
regwrite(cc, 'pc', hex2dec('100'), 'binary')
```

specifying that you are writing the value 256 (the decimal value of 0x100) to register pc as binary data.

To write a 64-bit value to a register pair, such as B1:B0, the following syntax specifies the value as a string, representation, and target registers.

```
regwrite(cc, 'b1:b0', hex2dec('1010'), 'ieee')
```

Registers B1:B0 now contain the value 4112 in double-precision format.

See Also

read, regread, write

Purpose

Reload most recent program file to target signal processor

Syntax

```
s = reload(cc,timeout)
s = reload(cc)
```

Description

`s = reload(cc,timeout)` resends the most recently loaded program file to the target processor. If you have not loaded a program file in the current session (so there is no previously loaded file), `reload` returns the null entry `[]` in `s` indicating that it could not load a file to the target. Otherwise, `s` contains the full path name to the program file. After you reset your target processor or after any event produces changes in your target processor memory, use `reload` to restore the program file to the target for execution.

To limit the time CCS IDE spends trying to reload the program file to the target, `timeout` specifies how long the load process can take. If the load process exceeds the timeout limit, CCS IDE stops trying to load the program file and returns an error stating that the time period expired. Exceeding the allotted time for the reload operation usually indicates that the reload was successful but CCS IDE did not receive confirmation before the timeout period passed.

`s = reload(cc)` reloads the most recent program file, using the `timeout` value set when you created link `cc`, the global timeout setting.

Using reload with multiprocessor boards

When your target board contains more than one processor, `reload` calls the reloading function for each processor represented by `cc`, reloading the most recently loaded program on each processor.

This is the same as calling `reload` for each processor individually through `ccsdsp` objects for each one.

Examples

After you create a link, use the link to reload your most recently loaded project. If you have not loaded a project in this session, `reload` returns an error and an empty value for `s`. Loading a project eliminates the error.

reload

```
cc=ccsdsp;
s=reload(cc,23)
Warning: No action taken - First load a valid Program file before
you reload
> In E:\nightly\toolbox\tiddk\tiddk\@ccs\@ccsdsp\reload.m at line
23

s =

    ''

open(cc,'D:\ti\tutorial\sim62xx\gelsolid\hellodsp.pjt',...
'project')

build(cc)

load(cc,'hellodsp.pjt')
halt(cc)
s=reload(cc,23)

s =

D:\ti\tutorial\sim62xx\gelsolid\Debug\hellodsp.out
```

See Also

cd, load, open

Purpose Remove file from active CCS IDE project

Syntax
`remove(cc, 'filename')`
`remove(cc, 'gelfilename')`

Description `remove(cc, 'filename')` deletes the file specified by `filename` from the active project in CCS IDE. You can remove files that exist in the active project only. `filename` must match the name of an existing file exactly to remove the file.

`remove(cc, 'gelfilename')` deletes the file specified by `gelfilename` from the active project in CCS IDE. You can remove files that exist in the active project only. `gelfilename` must match the name of an existing file exactly to remove the file.

Examples After you have a project in CCS IDE, you can delete files from it using `remove` from the MATLAB command line. For example, build a project and load the resulting `.out` file. With the project build complete, load your `.out` file by typing

```
load(cc, 'filename.out')
```

Now remove one file from your project, such as the GEL file.

```
remove(cc, 'gelfilename')
```

You see in CCS IDE that the GEL file no longer appears in the GEL files folder in CCS.

See Also `activate`, `add`, `cd`, `open`

reset

Purpose Reset target processor

Syntax `reset(cc, timeout)`
`reset(cc)`

Description `reset(cc, timeout)` stops program execution on the target processor and asynchronously performs a processor reset, returning all processor register contents to their power up settings. The reset function returns after the processor halts.

To allow you to determine how long `reset` waits for the processor to halt, input option `timeout` lets you set the waiting period in seconds. After you use `reset`, the routine returns after the processor halts or after `timeout` seconds elapses, whichever comes first.

`reset(cc)` stops program execution on the target processor and asynchronously performs a processor reset, returning all processor register contents to their power up settings. The reset function returns after the processor halts. `reset` uses the global timeout value defined in `cc` to determine how long to wait for the processor to halt before returning. Use `get` to examine the global timeout value for the link.

Use `run` to restart the program loaded on the target.

Compare to `halt` which does not reset the processor after the program stops running.

Using reset with multiprocessor boards

When your target board contains more than one processor, `reset` calls the processor resetting function for each processor represented by `cc`, resetting each processor.

This is the same as calling `reset` for each processor individually through `ccsdsp` objects for each one.

Note that the `run` and `halt` methods still apply as mentioned earlier in this section.

See Also `halt`, `restart`, `run`

Purpose

Reshape an array maintaining same number of elements

Syntax

```
reshape(x, [m, n])
reshape(x, [m, n, p...])
reshape(x, [m n p ...])
reshape(x, [..., [], ...])
```

Description

`reshape(x, [m, n])` returns the m-by-n array whose elements are taken by column from `x`. If `x` does not have $m*n$ elements, `reshape` returns an error from the operation.

Generally, `reshape(x, siz)` returns an n-dimensional array with the same elements as `x`, but reshaped to `size(siz)`. Note that `prod(siz)` must be the same as `prod(size(x))`.

`reshape(x, [m, n, p...])` returns an n-dimensional array with the same number of elements as `x`, but reshaped to have size m-by-n-by-p-by-.... For the reshape operation to work, $m*n*p*...$ must equal `prod(size(x))`.

`reshape(x, [m n p ...])` is the same as the preceding syntax.

`reshape(x, [..., [], ...])` calculates the length of the dimension replaced by `[]` in the command, so that the product of the dimensions equals `prod(size(x))`. For the length calculation to succeed, `prod(size(x))` must be evenly divisible by the product of the known dimensions (all the dimensions exclusive of the unknown dimension). Within the call, you are allowed to use one set of square brackets for one unknown dimension.

Examples

See Also

restart

Purpose Restore program counter to entry point for current program

Syntax `restart(cc,timeout)`
`restart(cc)`

Description `restart(cc,timeout)` halts the processor immediately and resets the program counter (PC) to the program entry point for the loaded program. Use `run` to execute the program after you use `restart`. `restart` does not execute the program after resetting the PC. `timeout` allows you to specify how long `restart` waits for the processor to stop and return the PC to the program entry point. Specify the value for `timeout` in seconds. After you use `restart`, the `restart` routine returns after resetting the PC or after `timeout` seconds elapse, whichever comes first. If the timeout period expires, `restart` returns a timeout error.

`restart(cc)` halts the processor immediately and resets the PC to the program entry point for the loaded program. Use `run` to execute the program after you use `restart`. `restart` does not execute the program after resetting the PC. When you omit the `timeout` argument, `restart` uses the global default timeout period defined in `cc` to determine how long to wait for the processor to stop and the PC to be reset to the program entry point.

Using restart with multiprocessor boards

When your target board contains more than one processor, `restart` calls the processor restarting function for each processor represented by `cc`, restarting the program loaded on each processor.

This is the same as calling `restart` for each processor individually through `ccdsp` objects for each one.

Examples When you are developing algorithms for your target processor, `restart` becomes a particularly useful function. Rather than resetting the target after each algorithm test, use the `restart` function to return the program counter to the program entry point. Since `restart` restores your local variables to their initial settings, but does not reset the processor, you are ready to rerun your algorithm with new values.

When your process gets lost or halts, restart is a quick way to restore your program.

See Also

halt, isrunning, run

resume

Purpose Resume execution of stopped or paused function

Syntax `resume(ff)`

Description `resume(ff)` restarts the function `ff` from where you stopped it or paused it. The function runs until completion or until it encounters a breakpoint.

See Also `restart`, `run`

Purpose Execute program loaded on target processor

Syntax

```
run(cc, 'state', timeout)
run(cc, 'main')
run(cc, 'tofunc', 'functionname')
run(ff)
run(ff, input1, value1, input2, value2, ..., inputn, valuen)
output = run(ff)
```

Description `run(cc, 'state', timeout)` starts to execute the program loaded on the target processor referred to by `cc`. Program execution starts from the location of the program counter. After starting program execution, the input argument `state` determines when you regain program control.

To define the action of `run`, `state` accepts strings that set the state of the processor:

state String	Run Action
main	Reset the program counter then run the program until the PC reaches main. Stop at main.
run	Start to execute the program. Wait until the program is running, then return. The program continues to run. If you omit the option argument, <code>run</code> defaults to this setting. Sets the processor to the running state and returns. This is useful when you want to continue to work in MATLAB while the processor executes a program.
runtohalt	Start to execute the program. Wait to return until the program encounters a breakpoint or the program execution terminates. Sets the processor to the running state and returns when the processor halts.

state String	Run Action
tofunc	Run the program from the current position of the program counter to the start of a specified function functionname.
tohalt	Changes the state of a running process to runtohalt, and waits for the processor to halt before returning. Use this when you want to stop a running process cleanly. If the processor is already stopped when you use this state setting, run returns immediately.

To allow you to specify how long run waits for the processor to start executing the loaded program before returning, the input argument `timeout` lets you set the waiting period in seconds.

After you use `run`, the routine returns after confirming that the program started to execute, or after `timeout` seconds elapses, whichever comes first. If the timeout period expires, `run` returns a time-out error.

`run(cc, 'main')` resets the program counter in your project then runs the program linked to `cc` until the counter reaches the start of `main`.

`run(cc, 'tofunc', 'functionname')` runs the program from the current position of the program counter until the counter reaches the function `functionname`. Compare this to `run(cc, 'main')` which resets the program counter before executing the program. Using the `tofunc` option does not reset the program counter.

`run(ff)` runs the function `ff` and puts the return value in the appropriate location. `run` performs a `goto` followed by `execute to run ff`.

`run(ff, input1, value1, input2, value2, ..., inputn, valuen)` writes the input values for `ff` before running the function, where `valuen` is the value for the input argument `inputn`. You can pass up to 10 input arguments and their values when you call `run`.

`output = run(ff)` puts the return value from running `ff` in `output`.

Using run with multiprocessor boards

When your target board contains more than one processor, run calls the program running function for each processor represented by cc, running the program loaded on each processor.

This is the same as calling run for each processor individually through ccdsp objects for each one. The other information about run on a single processor applies to each processor in the multiple processor target cases.

Examples

After you build and load a program to your target, use run to start execution.

```
cc = ccdsp('boardnum',0,'procnum',0); % Create a link to CCS
                                     % IDE.
cc.load('tutorial_6xevm.out'); % Load an executable file to the
                               % target.
cc.rtdx.configure(1024,4); % Configure four buffers for data
                           % transfer needs.

cc.rtdx.open('ichan','w'); % Open RTDX channels for read and
                           % write.

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

cc.restart; % Return the PC to the beginning of the current
            % program.

cc.run('run'); % Run the program to completion.
```

This example uses a tutorial program included with Link for Code Composer Studio. Set your CCS IDE working directory to be the one that holds your project files. The load function uses the current working directory unless you provide a full path name in the input arguments.

Rather than using the dot notation to access the RTDX functions, you can create an alias to the `cc` link and use the alias in later commands. Thus, if you add the line

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

to the program, you can replace

```
cc.rtdx.configure(1024,4);
```

with

```
configure(rx,1024,4);
```

See Also

halt, isrunning, restart

Purpose Save files and projects in CCS IDE

Syntax `save(cc, 'filename', 'type')`

Description `save(cc, 'filename', 'type')` save the file in CCS IDE identified by filename of type 'type'. type identifies the type of file to save, either project files when you use 'project' for type, or text files when you use 'text' for the type option. To save a specific file in CCS IDE, filename must match the name of the file to save exactly. If you replace filename with 'all', save writes every open file whose type matches the type option. File types recognized by save include these extensions.

type String	Affected files
project	Project files with the .pjt extension.
text	All files with these extensions — a*, .c, .cc, .ccx, .cdb, .cmd, .cpp, .lib, .o*, .rcp, and .s*. Note that 'text' does not save .cfg files.

When you replace filename with the null entry [], save writes to storage the current active file window in CCS IDE, or the active project when you specify project for the type option.

Examples To clarify the different save options, here are commands that save open files or projects in CCS IDE.

Command	Result
<code>save(cc, 'all', 'project')</code>	Save all open projects in CCS IDE.
<code>save(cc, 'my.pjt', 'project')</code>	Save the project my.pjt.
<code>save(cc, [], 'project')</code>	Save the active project.

save

Command	Result
<code>save(cc, 'all', 'text')</code>	Save all open text files. This includes source file, libraries, command files, and others.
<code>save(cc, 'my_source.cpp', 'text')</code>	Save the text file <code>my_source.cpp</code> .
<code>save(cc, [], 'text')</code>	Save the active file window.

See Also

`add`, `cd`, `close`, `open`

Purpose

Set properties of links to CCS IDE and RTDX interface

Syntax

```
set(cc, 'propertyname', 'propertyvalue')
set(cc, 'proprname1', 'propvalue1', 'proprname2', 'propvalue2')
v = set(cc)
cc.propertyname = propertyvalue
set(rx, 'propertyname', 'propertyvalue')
set(rx, 'proprname1', 'propvalue1', 'proprname2', 'propvalue2')
v = set(rx)
rx.propertyname = propertyvalue
```

Description

`set(cc, 'propertyname', 'propertyvalue')` sets the specified property of `cc` to the specified value.

`set(cc, 'proprname1', 'propvalue1', 'proprname2', 'propvalue2')` sets multiple properties (`proprname1`, `proprname2`) of `cc` to corresponding property values (`propvalue1`, `propvalue2`) with a single statement. `cc` must be a link.

`v = set(cc)` returns the properties and range of acceptable values of link `cc`. When the range of values for a property is not finite, `set` returns `{}` for the property value. When you omit the output argument, MATLAB displays the results on the screen.

`cc.propertyname = propertyvalue` uses the dot notation to set `propertyname` to `propertyvalue`.

`set(rx, 'propertyname', 'propertyvalue')` sets the specified property of `rx` to the specified value.

`set(rx, 'proprname1', 'propvalue1', 'proprname2', 'propvalue2')` sets multiple properties (`proprname1`, `proprname2`) of `rx` to corresponding property values (`propvalue1`, `propvalue2`) with a single statement.

`v = set(rx)` returns the properties and range of values of link `rx`. `rx` is the RTDX portion of a link for CCS IDE. When the range of values for a property is not finite, `set` returns `{}` for the property value. When you omit the output argument, MATLAB displays the results on the screen.

`rx.propertyname = propertyvalue` uses the dot notation to set `propertyname` to `propertyvalue` for link `rx`.

Examples

Create a link to CCS IDE

```
cc = ccsdsp;
```

Now review the properties of `cc` to see the acceptable values for each property.

```
v=set(cc)
```

```
v =
```

```
    timeout: {}  
    page: {}  
eventwaitms: {}
```

The properties accept any input value, as shown by the `{}` entries returned.

Set `timeout` to 10 s and `page` to 2. Property `eventwaitms` cannot be set. It is read-only.

```
set(cc,'timeout',10,'page',2)  
get(cc)
```

```
ans =
```

```
    app: [1x1 activex]  
dspboards: [1x1 activex]  
dspboard: [1x1 activex]  
dsptasks: [1x1 activex]  
dsptask: [1x1 activex]  
dspuser: [1x1 activex]  
    rtdx: [1x1 rtdx]  
apiversion: [1 0]  
ccsappexe: 'D:\ticcs\cc\bin\cc_app.exe'
```

```
boardnum: 0
procnum: 0
timeout: 10
page: 2
```

Reset page to 0 since this is a C6xxx processor.

```
cc.page = 0
get(cc)
```

```
ans =
```

```
    app: [1x1 activex]
 dspboards: [1x1 activex]
 dspboard: [1x1 activex]
 dsptasks: [1x1 activex]
 dsptask: [1x1 activex]
 dspuser: [1x1 activex]
    rtdx: [1x1 rtdx]
apiversion: [1 0]
ccsappexe: 'D:\ticcs\cc\bin\cc_app.exe'
boardnum: 0
procnum: 0
timeout: 10
page: 0
```

See Also

get

symbol

Purpose Program symbol table from CCS IDE

Syntax `s = symbol(cc)`

Description `s = symbol(cc)` returns the symbol table for the program loaded in CCS IDE. `symbol` only applies after you load a target program file. `s` is an array of structures where each row in `s` presents the symbol name and address in the table. Therefore, `s` has two columns; one is the symbol name, and the other is the symbol address and symbol page. For example, this table shows a few possible elements of `s`, and their interpretation.

s Structure Field	Contents of the Specified Field
<code>s(1).name</code>	String reflecting the symbol entry name.
<code>s(1).address(1)</code>	Address or value of symbol entry.
<code>s(1).address(2)</code>	Memory page for the symbol entry. For TI C6xxx processors, the page is 0.

You can use field `address` in `s` as the address input argument to `read` and `write`.

If you use `symbol` and the symbol table does not exist, `s` returns empty and you get a warning message.

Symbol tables are a portion of a COFF object file that contains information about the symbols that are defined and used by the file. When you load a program to the target, the symbol table resides in CCS IDE. While CCS IDE may contain more than one symbol table at a time, `symbol` accesses the symbol table belonging to the program you last loaded on the target.

Examples Demonstrating this function requires that you load a program file to your target. In this example, build and load the Link for Code Composer Studio demo program `c6711dskafxr`. Start by entering `c6711dskafxr` at the MATLAB prompt.

```
c6711dskafxr;
```

Now set the simulation parameters for the model and build the model to your target. With the model loaded on your target, use `symbol` to return the entries stored in the symbol table in CCS IDE.

```
cc = ccdsp;  
s = symbol(cc);
```

`s` contains all the symbols and their addresses, in a structure you can display with the following code:

```
for k=1:length(s),disp(k),disp(s(k)),end;
```

MATLAB lists the symbols from the symbol table in a column.

See Also

`load`, `run`

visible

Purpose Set whether CCS IDE window is visible while CCS runs

Syntax `visible(cc,state)`

Description `visible(cc,state)` sets CCS IDE to be visible or not visible on the desktop. Input argument `state` accepts either 0 or 1 to set the visibility. Setting `state` equal to 0 makes CCS IDE not visible on the desktop. However, the CCS IDE process runs in the background while the window is not visible.

Running CCS IDE without making it visible lets you use the CCS IDE functions from MATLAB, without interacting with CCS IDE. If you need to interact with CCS IDE, set `state` equal to 1. This makes CCS IDE visible and you can use the features of the user window.

An important feature of `visible` is that it creates a new link to CCS IDE when you change the IDE visibility. As a result, after you use

```
visible(cc,state)
```

to make CCS IDE show on your desktop, the MATLAB `clear all` function does not remove the visibility handle. You must remove the handle explicitly before you use `clear`.

To see the visibility difference, open Code Composer Studio and use Windows Task Manager to look at the applications and processes running on your computer. When CCS IDE is visible (the normal startup and operating mode for the IDE), CCS IDE appears listed on the **Applications** page of Task Manager. And the process `cc_app.exe` shows up on the **Processes** page as a running process. When you set CCS IDE to not visible (`state` equal 0), CCS IDE disappears from the **Applications** page, but remains on the **Processes** page, with a process ID (PID), using CPU and memory resources.

Note When you close MATLAB while CCS IDE is not visible, MATLAB closes CCS if it launched the IDE. This happens because the operating system treats CCS as a subprocess in MATLAB when CCS is not visible. By having MATLAB close the invisible IDE when you close MATLAB, you do not need to worry about CCS remaining open with no way to close it except by using Windows Task Manager.

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

Examples

Test to see whether CCS IDE is running. Then change the visibility and check again. Start by launching CCS IDE. Then open MATLAB and at the prompt, enter

```
cc=ccsdsp;
```

MATLAB creates a link to CCS IDE and leaves CCS IDE visible on your desktop.

```
invisible(cc)
```

```
ans =  
    1
```

Now, change the visibility state to 0, or invisible, and check the state.

```
visible(cc,0)  
invisible(cc)
```

```
ans =  
    0
```

Notice that CCS IDE is not visible on your desktop. Recall that MATLAB did not open CCS IDE. When you close MATLAB with

visible

CCS IDE in this invisible state, CCS IDE remains running in the background. To close it, do one of the following operations.

- Start MATLAB. Create a new link to CCS IDE. Use the new link to make CCS IDE visible. Close CCS IDE.
- Open Windows Task Manager. Click **Processes**. Find and highlight `cc_app.exe`. Click **End Task**.

See Also

`isvisible`, `load`

Purpose

Write data to memory on target processor

Syntax

```
write(cc,address,data,timeout)
write(cc,address,data)
write(objname)
write(objname,index)
write(objname,structindex,mem1,value1,...,memn,valuen,
      memindex)
write(...,timeout)
```

Description**Link Object Syntaxes**

`write(cc,address,data,timeout)` sends a block of data to memory on the processor referred to by `cc`. The address and data input arguments define the memory block to write — where the memory starts and what data is being written. The memory block to write begins at the memory location defined by `address`. `data` is the data to write, and can be a scalar, a vector, a matrix, or a multidimensional array.

Data get written to memory in column-major order. `timeout` is an optional input argument you use to terminate long write processes and data transfers. For details about each input parameter, read the following descriptions.

Note Do not attempt to write data to the target while it is running.

To update values in memory on a running target, such as values to change during processing, insert one or more breakpoints in the project code and perform the write operation while the target code is paused at one of the breakpoints. After you read the data, release the breakpoint.

`address` — `write` uses `address` to define the beginning of the memory block to write to. You provide values for `address` as either decimal or hexadecimal representations of a memory location in the target processor. The full address at a memory location consists of two parts:

the offset and the memory page, entered as a vector [*location*, *page*], a string, or a decimal value.

When the processor has only one memory page, as is true for many digital signal processors, the value of the page portion of the memory address is 0. By default, `ccsdsp` sets the page value to 0 at creation if you omit page as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Offset is 31 decimal on the page referred to by <code>cc(page)</code>
10	Decimal	Offset is 10 decimal on the page referred to by <code>cc(page)</code>
[18,1]	Vector	Offset is 18 decimal on memory page 1 (<code>cc(page) = 1</code>)

To specify the address in hexadecimal format, enter the address property value as a string. `write` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the `write` uses `hex2dec`. When you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `cc(page)`.

`data` — the scalar, vector, or array of values that are written to memory on the processor. `write` supports the following data types:

Datatypes	Description
double	Double-precision floating point values
int8	Signed 8-bit integers
int16	Signed 16-bit integers
int32	Signed 32-bit integers
single	Single-precision floating point data
uint8	Unsigned 8-bit integers
uint16	Unsigned 16-bit integers
uint32	Unsigned 32-bit integers

To limit the time that `write` spends transferring data from the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You may find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `write` defaults to the global timeout defined in `cc`.

`write(cc, address, data)` ends a block of data to memory on the processor referred to by `cc`. The `address` and `data` input arguments define the memory block to be written — where the memory starts and what data is being written. The memory block to be written to begins at the memory location defined by `address`. `data` is the data to be written, and can be a scalar, a vector, a matrix, or a multidimensional array. Data get written to memory in column-major order. Refer to the preceding syntax for details about the input arguments. In this syntax, `timeout` defaults to the global timeout period defined in `cc.timeout`. Use `get` to determine the default timeout value.

Like the `isreadable`, `iswritable`, and `read` functions, `write` checks for valid address values. Illegal address values would be any address space larger than the available space for the processor — 2^{32} for the C6xxx processor family and 2^{16} for the C5xxx series. When the function

identifies an illegal address, it returns an error message stating that the address values are out of range.

Working with Negative Values

Writing a negative value causes the data written to be saturated because char is unsigned on the processor. Hence, a 0 (a NULL) is written instead. A warning results as well, as this example shows.

```
cc = ccsdsp;
ff = createobj(cc, 'g_char'); % Where g_char is in the code.
write(ff, -100);
Warning: Underflow: Saturation was required to fit the data into
an addressable unit.
```

When you try to read the data you wrote, the character being read is 0 (NULL) — so there seems to be nothing returned by the read function.

You can demonstrate this by the following code, after writing a negative value as shown in the previous example.

```
readnumeric(x)
ans =
0
read(x) % Reads the NULL character
ans = % Apparently nothing is returned.

double(read(x)) % Read the numeric equivalent of NULL.
ans = % Again, appears not to return a value.
```

Embedded Object Syntaxes

`write` works with all of the objects you create with `createobj`. To transfer data from MATLAB to Code Composer Studio, use one of the `write` functions — `write` — depending on the data to write. Note that `write` and its variants are the only way to get data from MATLAB to CCS from objects.

`write(objname)` writes all the data in `objname` to the location accessed by object `objname`. Properties of `objname`, such as `wordsize`,

`storageunitspervalue`, `size`, `represent`, and `binarypt` — determine how `write` performs the numeric conversion.

`data` is a numeric array whose dimensions are defined by the `size` property of `objname`. Object property `size` is the *dimensions* vector. Each element in the dimensions vector contains the size of the data array in that dimension. When `size` is a scalar, `data` is a column vector of the length specified by `size`.

For example, when `size` is `[2 3]`, `data` is a 2-by-3 array.

Properties of the Object

`objname`, the object that accesses the data, has the following properties, if the object is a numeric object. The properties are different for different types of objects, such as structure objects or register objects.

Property	Options	Description
<code>size</code>	Greater than 1	Specifies the dimensions of the output numeric array.
<code>arrayorder</code>	<code>col-major</code> or <code>row-major</code>	Defines how to map sequential memory locations into arrays. <code>col-major</code> is the default, and the MATLAB standard. C uses <code>row-major</code> ordering most often.

write

Property	Options	Description
represent	float, signed, unsigned, fract	Determines the numeric representation used in the output data. <ul style="list-style-type: none">• float — IEEE floating point representation, either 32- or 64 bits• signed — two's complement signed integers• unsigned — unsigned binary integer• fract — fractional fixed-point data
wordsize	Greater than 1	(Read-only) Calculated from other object properties such as storageunitspvalue
binarypt	0 to wordsize	Determines the position of the binary point in a word to specify its interpretation

`write(objname, index)` reads the specified element in the memory location accessed by `objname`. Input argument `index` is a scalar or a vector that identifies the particular data element to return. When you enter `[]` for `index`, `write` returns all the data stored at the memory location. When you enter a scalar for `index`, `write` returns a column vector of length `size` containing the data from the memory space. When `index` is a vector, `write` returns the element in the array specified by the entries in the vector.

For example, if you are reading data from a 3-by-3-by-3 array, setting `index` to be `[2 2 2]` returns the element data(2,2,2). To return more than one element, use MATLAB standard range notation for the vector elements in `index`. As an example, when `index` is `[1:6]`, `write` returns the first six elements of data. Remember that the number of elements in the vector in `index` must be either one (a scalar) or the same as the number of dimensions in `data` and specified by the property size.

When `data` is a four dimensional array, your vector in `index` must have four elements, one for each array dimension. Otherwise, `write` cannot determine which elements to return.

`write(objname,structindex,mem1,value1,...,memn,valuen,memindex)` reads the members of the structure that `objname` accesses. When you omit all of the input arguments except `objname`, `write` returns the entire structure. `memn,valuen,memindex`, and `stindex` (an optional input argument) specify which structure member to read:

- `memn` — Specifies the name of the member of the structure to write
- `valuen` — Specifies the value to write to `memn`
- `memindex` — Provides the index of the data element to write
- `structindex` — Identifies the structure to write when `objname` accesses a structure containing structures or a vector

Note that the class of the object data from the `write` operation depends on the class of the member being read — numeric values return numeric objects, string values return string objects, and so on.

`write(...,timeout)` During `write` operations, the `timeout` property of `objname` determines the time allowed to complete the write. Including a value for the `timeout` input argument in the `write` syntax lets you override the `timeout` property setting for `objname` with the value you enter for argument `timeout`.

For reading large data arrays, setting the `timeout` value as an input option may be necessary to let `write` run to completion. Note that using

the `timeout` input argument does not change the `timeout` property value for `objname`.

When you need to write one member of a structure or to do individual write operations, consider using `getmember`.

Notes About Using `write` With Embedded Objects

When you are writing data into memory on your target, consider a number of points that affect how `write` performs the write operation.

- The data you write to the target can be either numeric or hexadecimal format.
- When the data you are writing contains values that exceed the representable range for the variable data type and word size, the value written saturates to the maximum or minimum representable value for the variable representation. For example, if you try to write the value 70000 into an unsigned, 16-bit variable, the write operation stores 65535 into memory. 65535 is the maximum representable value for unsigned, 16-bit integers. Similarly, if you try to write -3 to the same variable, the stored value will be 0. You cannot represent negative numbers in the unsigned format.
- When you write more data elements to memory than fit in the specified size of the memory block, only the number of elements that fit in the memory block get written to the target. Excess elements do not get stored and are lost.
- When you write fewer data elements to memory than fit in the specified size of the memory block, all the elements get written to the memory block on the target. Memory space in the block which does not receive new elements is not affected by the write operation and remains unchanged.
- Use separate `write` operations to write multiple data elements to different locations within the memory block accessed by an object. For example, to write to the fifth and eighth elements of a 1-by-10 array in memory accessed by an object, use `write` twice — once to write to the fifth memory location and the again to write to the

eight location. You cannot combine the write operations in a single command unless the memory locations are contiguous. Refer to the next item in this list for information about writing to contiguous memory locations within a memory block.

- To write a block of data into contiguous locations in the memory block accessed by the object, supply just the starting index for the locations in the memory block.

Notes About Writing Strings to Memory

Writing strings to memory has some idiosyncrasies. Recall the following points when you use `write` with string data.

- Data that you write to memory can be numeric or string data.
- When your data is strings or characters, the write operation is controlled by the `charconversion` property value for the object. `write` accepts and writes only characters with ASCII values from 0 to 127 when the `charconversion` property value is ASCII.
- Numeric data is not restricted in any way when you use `write`.
- `write` appends a null character as the last element written to memory, except when
 - You write numeric data.
 - The object points to a single C character (size equals 1).
 - The amount of data you are writing exceeds the allocated space.
- When the string data you write does not fill the allotted space in memory, `write` does not fill the extra space with zeros — no zero padding.

Notes About Writing to Structures

When you are writing data to a particular index within the structure, consider using `getmember` to create an object that accesses the desired member. Then use your new object as `objname` in the `write` function call.

Refer to the section “Embedded Object Examples” on page 5-242 for examples of write in use with structures.

Examples

Link Object Examples

Create a link to a target processor and write data to the target. In this example, CCS IDE recognizes one board having one processor.

```
cc = ccdsp;  
cc.visible(1);  
write(cc, '50', 1:250);  
mem = read(cc, 0, 50, 'double') % Returns 50 values as a column  
                                % vector in mem.
```

It may be more convenient to return the data in an array. If you enter a vector for count, mem contains a matrix of dimensions the same as vector count.

```
write(cc, 10, 1:100);  
mem=read(cc, 10, [10 10], 'double')
```

mem =

1	11	21	31	41	51	61	71	81	91
2	12	22	32	42	52	62	72	82	92
3	13	23	33	43	53	63	73	83	93
4	14	24	34	44	54	64	74	84	94
5	15	25	35	45	55	65	75	85	95
6	16	26	36	46	56	66	76	86	96
7	17	27	37	47	57	67	77	87	97
8	18	28	38	48	58	68	78	88	98
9	19	29	39	49	59	69	79	89	99
10	20	30	40	50	60	70	80	90	100

Embedded Object Examples

The following examples show you some of the details about using write with embedded objects. Also, you can find an example or two for each

of the items in the list from the section “Notes About Using write With Embedded Objects” on page 5-240.

When you try to write more elements to the memory space than the space can hold, write ignores the extra elements, storing only the ones that fit. In this example, mm is an object that access a 1-by-10 variable in memory.

- Writing 15 elements to the 1-by-10 array

```
write(mm,[1:15])
```

results in elements 1 through 10 (or [1:10]) being written to memory. Elements 11 through 15 are ignored.

- Writing 5 element to the 1-by-10 array

```
write(mm,[1:5])
```

results in elements [1:5] being written to memory without changing the values in memory for element [6:10].

To write multiple element to different indices in the 1-by-10 array, use multiple write calls.

```
write(mm,5,6)
```

writes value 6 to the fifth index in the array. Now to write another value to a different index, use

```
write(mm,7,9)
```

which writes value 9 to the seventh element of the array. Trying to use one call like

```
write(mm,[5 7],[6 9])
```

to write 6 into index 5 and 9 into index 7 does not work.

write

Examples That Write Strings

Embedded object `mm` accesses a 1-by-12 array in memory on the target.

To write a string to target memory, use

```
write(mm, 'Hello World')
```

which writes 11 characters to memory plus the additional null character at the end of the string.

H	e	l	l	o		W	o	r	l	d	\0	M
---	---	---	---	---	--	---	---	---	---	---	----	---

Notice that the `M` at the end of the memory space is not affected by the write operation. Now write a new string to memory, such as “Ciao.”

```
write(mm, 'Ciao')
```

After writing to memory, the stored string looks like:

C	i	a	o	\0		W	o	r	l	d	\0	M
---	---	---	---	----	--	---	---	---	---	---	----	---

where the fifth element now holds the null character that resulted from writing `Ciao` to indices 1 through 4, plus the null character in index 5. All the characters after index 5 remain the same. Recall that if you now read the memory, the read operation stops at the first null character and does not return “World” or “M.”

See Also

`read`, `symbol`

Purpose	Write binary data to DSP memory
Syntax	<pre>writebin(mm,data) writebin(mm,data,[]) writebin(mm,data,index) writebin(...,timeout)</pre>
Description	<p><code>writebin(mm,data)</code> writes a block of binary strings data into the memory block described by <code>mm</code>. <code>data</code> is string containing 0 or 1 or a cell array of binary strings of 0s and 1s. Writing to the target fails when <code>data</code> has more entries than the memory range covers as specified by <code>mm</code>. Conversely, when <code>data</code> has fewer elements than the memory range allows, <code>writebin</code> starts writing data at the first address in the memory location.</p> <p><code>writebin(mm,data,[])</code> same as <code>writebin(mm,data)</code>.</p> <p><code>writebin(mm,data,index)</code> Writes a single binary string data to the specified index (the address offset).</p> <p><code>writebin(...,timeout)</code> adds the optional <code>timeout</code> input argument to specify the time allowed for the write operation to finish. Changing the default time out value may be necessary when you write large arrays to memory. Note that when MATLAB returns an error that the time-out period expired, it does not necessarily mean the write failed. Only that MATLAB did not receive notification about the write operation finishing before the allotted time passed. The write operation usually works correctly in spite of the message.</p>
See Also	<code>read</code> , <code>write</code>

writemsg

Purpose Write messages to specified RTDX channel

Syntax `data = writemsg(rx,channelname,data)`
`data = writemsg(rx,channelname,data)`

Description `data = writemsg(rx,channelname,data)` writes data to a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for write access. All messages must be the same type for a single write operation. `writemsg` takes the elements of matrix data in column-major order.

To limit the time that `writemsg` spends transferring messages from the target processor, the optional argument `timeout` tells the message transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You may find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, write defaults to the global timeout defined in `cc`.

`writemsg` supports the following data types: `uint8`, `int16`, `int32`, `single`, and `double`.

`data = writemsg(rx,channelname,data)` uses the global timeout setting assigned to `cc` when you create the link.

Examples After you load a program to your target, configure a link in RTDX for write access and use `writemsg` to write data to the target. Recall that the program loaded on the target must define `ichannel` and the channel must be configured for write access.

```
cc=ccsdsp;
rx = cc.rtdx;
open(rx,'ichannel','w'); % Could use rx.open('ichannel','w')
enable(rx,'ichannel');
inputdata(1:25);
writemsg(rx,'ichannel',int16(inputdata));
```


As a further illustration, the following code snippet writes the messages in matrix `indata` to the write-enabled channel specified by `ichan`. Note again that this example works only when `ichan` is defined by the program on the target and enabled for write access.

```
indata = [1 4 7; 2 5 8; 3 6 9];
writemsg(cc.rtdx, 'ichan', indata);
```

The matrix `indata` is written by column to `ichan`. The preceding function syntax is equivalent to

```
writemsg(cc.rtdx, 'ichan', [1:9]);
```

See Also

`readmat`, `readmsg`, `write`

Supported Hardware

This appendix provides the details about the hardware and simulators that work with the Link for Code Composer Studio.

Introduction to Supported Platforms (p. A-2)	Describes the hardware that works with the links, embedded objects, and RTDX
Supported Versions of Code Composer Studio (p. A-7)	Lists versions for Link of CCS and the CCS versions they support
Continuing Issues in Link for Code Composer Studio (p. A-9)	Reviews some considerations and information about working with Link for Code Composer Studio and various targets.

Introduction to Supported Platforms

This appendix lists the hardware and simulators that work with the latest released version of Link for Code Composer Studio Development Tools. Generally the product supports boards and simulators from a given family. In some cases, only the simulators work, as noted in the tables in the next sections.

Supported Hardware and Simulators

The Link for Code Composer Studio supports the following processors and boards produced by TI and others.

Supported Hardware And Simulators	Description
C2000	
Simulators (C24x, C27x, C28x)	Simulators for the C2000 DSP family
C2401 eZdsp	Starter kit for the C2401 processor
C2407 eZdsp	Starter kit for the C2407 processor
C2812 eZdsp	Starter kit for the C2812 processor
C2808 eZdsp	Starter kit for the C2808 processor
C5000	
Simulators (C54x, C5x)	Simulators for the C5000 DSP family
C5402 DSK	DSP starter kit for the C5402 processor
C5416 DSK	DSP starter kit for the C5416 processor
C5510 DSK	DSP starter kit for the C5510 processor
C6000	
Simulators (C62x, C64x, C67x)	Simulators for the C6000 DSP family
C6211 DSK	DSP starter kit for the C6211 processor

Supported Hardware And Simulators	Description
C6416 DSK	DSP starter kit for the C6416 processor
DM64x	C6400 processor-based video card
C6701 EVM	Evaluation module for the C6701 processor
C6711 DSK	DSP starter kit for the C6711 processor
C6713 DSK	DSP starter kit for the C6713 processor
OMAP	
OMAP1510	Boards and simulators based on the OMAP1510.
OMAP5910	Boards and simulators based on the OMAP5910.
TMS470	
TMS470R1x	Boards and simulators based on the TMS470R1x processor.
TMS470R2x	Boards and simulators based on the TMS470R2x processor.

Link Features Supported by Each Processor or Family

Specific Link Features Supported for Each Processor Family

Within the collection of hardware that Link for Code Composer Studio supports, some features of the link do not apply.

Debug mode includes operations that CCS handles and that Link for Code Composer Studio enables you to use from MATLAB — a “Yes” tells you that the listed hardware supports MATLAB interaction with CCS. Embedded Objects support indicates that the board family supports using objects in MATLAB to work with symbol table entries in CCS. A “Yes” in

the Hardware-in-the-Loop column means the board family supports using function objects to run functions on your target from MATLAB.

Link for Code Composer Studio provides components that work with and use CCS IDE and TI Real-Time Data Exchange (RTDX):

- *Debug Component* — lets you use objects to create links between CCS IDE and MATLAB. From the command window, you can run applications in CCS IDE, send to and receive data from target memory, and check the processor status, as well as other functions such as starting and stopping applications running on your digital signal processors.
- *Data Manipulation Component* — provides object methods and properties that let you access and manipulate information stored in memory and registers on digital signal processors, or in your Code Composer Studio project. From MATLAB you gather information from you project, work with the information in MATLAB, doing things like converting data types, creating function declarations, or changing values, and return the information to your project — all from the MATLAB command line.
- *Function Call Component* — enables you to write scripts in MATLAB that exercise functions from your project on your target processor. From MATLAB, you can generate data, send the data to your target and use a C function in your program to manipulate the data on your hardware or simulator. Afterwards, you return the output to MATLAB so you can analyze the results.
- *Real-Time Data Exchange (RTDX) Component* — provides a communications pathway between MATLAB and digital signal processors installed on your PC. Using objects in the Link for Code Composer Studio, you open channels to processors on boards in your computer and send and retrieve data about the processors and executing applications, as well as send data to the processes for use and get data from the applications.

In the next table, each processor family appears with headings that specify the support provided.

Processor Family Support for Link for Code Composer Studio Functions

Processor Family	Debug Mode	Data Manipulation	Function Call Support	RTDX
C24xx	Yes	No	No	No
C27xx	Yes	No	No	No
C28xx	Yes	Yes	Yes	Yes
C54xx	Yes	Yes	Yes	Yes
C55xx	Yes	Yes	No	Yes
C62xx	Yes	Yes	Yes	Yes
C64x and C64x+	Yes	Yes	Yes	Yes
C67x and C67x+	Yes	Yes	Yes	Yes
OMAP1510				
• C55x DSP	Yes	Yes	No	Yes
• TMS470R2x	Yes	Yes	No	No
TMS470R1x	Yes	Yes	No	No
TMS470R2x	Yes	Yes	No	No

OMAP Coemulation Support

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

Custom Hardware Support

Link for Code Composer Studio supports processors as shown in the previous tables. If your custom hardware

- Uses one or more of the processors shown in the preceding tables

- You are able to use Code Composer Studio IDE to interact with your board/processor combination

Link for Code Composer Studio should work with your hardware.

Supported Versions of Code Composer Studio

The following table lists versions of Link for Code Composer Studio Development Tools and the versions of Code Composer Studio they support.

Link for CCS Development Tools Version	MATLAB Release	Supported Code Composer Studio Version(s)
2.1	R2006b	<ul style="list-style-type: none"> • CCS 3.2 for C64x+ processors • CCS 3.1for C2000, C5000, C6000, and OMAP processors
2.0	R2006a+	CCS 3.1for C2000, C5000, C6000, and OMAP processors
1.5	R2006a	CCS 3.1for 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
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

Link for CCS Development Tools Version	MATLAB Release	Supported Code Composer Studio Version(s)
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 OMAPprocessors
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 OMAPprocessors
1.3	R13SP1+	CCS 2.12 for C2000, C5000, C6000, and OMAP processors

Continuing Issues in Link for Code Composer Studio

Some long-standing issues affect the Link for Code Composer Studio product. When you are using links to work with Code Composer Studio, recall the information provided in this section. The latest issues in the list appear at the bottom. Note that HIL refers to “hardware in the loop,” sometimes called processor in the loop (PIL) in other applications, and often called function calls here.

- “Function Call Support for Different Compiler Options” on page A-10
- “Function Calls on Functions That Use Global Variables” on page A-10
- “Demonstration Programs Do Not Run Properly Without Correct GEL Files” on page A-11
- “Issues Using USB-Based RTDX Emulators and the C6416 DSK and C6713 DSK” on page A-12
- “Error When Accessing type Property of ccstdsp Object Having Size>1” on page A-13
- “Changing the represent Property of an Object” on page A-14
- “Changing Values of Local Variables Does Not Take Effect” on page A-15
- “Code Composer Studio Cannot Find a File After You Halt a Program” on page A-15
- “C54x XPC Register Can Be Modified Only Through the PC Register” on page A-17
- “Working with Multiple Installed Versions of Code Composer Studio” on page A-17
- “Changing CCS Versions During a MATLAB Session” on page A-18
- “createobj and address Return Inconsistent Page Information on C5xxx Targets” on page A-18
- “MATLAB Hangs When Code Composer Studio Cannot Find a Target” on page A-20
- “Different Read Techniques Appear to Return Different Values” on page A-22

Function Call Support for Different Compiler Options

The CCS project compiler settings that usually return the best results during function call operation appear in the following table:

Compiler Option	Preferred Setting
Debug info	Full Symbolic Debug (-g)
Optimization level	none
Optimization Speed vs. Size	none
Product level optimization	none

Issues When You Use Other Compiler Settings

If you make the following selections for the compiler settings to use, consider these comments.

- Setting the **Debug Info** option (applies to CCS 2.21 and earlier versions only)
 - Selecting Dwarf Debug (-gw) — in some cases, you need to supply the function declaration manually when you select Dwarf Debug.
 - Selecting No Debug — no debug information is made available. In all cases, you are required to supply the function declaration manually using declare.
- Setting **Optimization Level** to File (-o3)

Input variables are not listed as locally declared variables of the function. As a result, Link for Code Composer Studio may generate warnings while constructing function object.

Function Calls on Functions That Use Global Variables

For functions which use global variables, the global variables must be initialized before you attempt to perform function call processing. Without initialization, the function call process returns incorrect results. The global variables are automatically initialized when you configure the CCS project as follows:

- 1 Your project has function main defined.

- 2 Your project links to an appropriate run-time support library, such as `rts6400.lib`.
- 3 Your project has the load-time or run-time autoinitialization (`-c` or `-cr` option) set.

Using other configurations for your CCS project bypasses the proper initialization processes. Refer to your TI documentation on run-time initialization for more information.

Demonstration Programs Do Not Run Properly Without Correct GEL Files

To run the Link for Code Composer Studio 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`, `rtdxlmsdemo`
Emulator: XDS-510
GEL file to load: `c5416_dsk.gel`
- Board: C6416 DSK
Demos: `rtdxtutorial`, `rtdxlmsdemo`
Emulator: XDS-510
GEL file to load: `DSK6416.gel`
- Board: C6713 DSK
Demos: `rtdxtutorial`, `rtdxlmsdemo`
Emulator: XDS-510
GEL file to load: `DSK6713.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.

Issues Using USB-Based RTDX Emulators and the C6416 DSK and C6713 DSK

You may encounter a few problems when you try to use the USB-based RTDX emulators with the C6713 and C6416 DSP Starter Kits. The problems relate to setting up RTDX and opening/closing RTDX channels.

1 Setting up and cleaning up RTDX.

If you do not set up RTDX correctly, your hardware might end up in a bad state and RTDX data transfers may not work correctly. Rerunning the application without setting up RTDX properly yields the same result. To bring the hardware back to a working state, you have to recycle power to your board. Likewise, if RTDX is not cleaned up correctly after running an application, your hardware can go into a bad state.

2 When you close and reopen CCS for *DSP Starter Kit for TMS320C6416* or *DSP Starter Kit for TMS320C6713*, you have to adhere to the two second close and reopen requirement as noted in TI documentation.

In the *Quick Start Installation Guide*, under “Debug Hints and Trouble Shooting,” item 6 states

“The LED above the USB connector illuminates when the DSK is powered on. Do not launch Code Composer Studio until the LED turns off.”

When your CCS application terminates, the USB bus is nonenumerated. It takes a few seconds (roughly two seconds in Windows 2000 or Windows XP) to enumerate the USB bus again.

Consequently, although the CCS application may appear to have gone away from the desktop, there can still be some processes running. We recommend that you follow the above guidelines when communicating with a C6416 DSK, C6713 DSK, or XDS510USB on a close and reopen sequence.

References

The information in this discussion comes from the following TI publications — `dsk6416_releasenotes.htm` and `dsk6713_releasenotes.htm`

- Section 3.0 Installation
- Section 5.0 Some Basics on How it Works
- *Quick Start Installation Guide*, “Debug Hints and Trouble Shooting”

To avoid having problems in MATLAB when you work with links, note these recommended tasks (in order) for creating handles to CCS from MATLAB.

- 1 Assuming CCS IDE is not open (`cc_app.exe` is not in the Windows Task Manager), create a handle to CCS (`cc_app.exe` appears in the Task Manager).

```
cc = ccsdsp
```

- 2 Clear the handle to CCS (`cc_app.exe` disappears from the Task Manager).

```
clear cc
```

- 3 Wait about two seconds before creating a new handle to CCS.

```
pause(2);
cc = ccsdsp
```

Error When Accessing type Property of ccsdsp Object Having Size> 1

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

```
cc
Array of CCSDSP 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 the represent Property of an Object

An object's `represent` property is writable. You can change it to modify the access format. For example, an object with `represent` set to `float` can be changed to `represent` set to `signed`. After the change, the data is read as a signed integer. Likewise, the data is written as a signed integer.

Here's one example of changing the property value for `represent`. Create a `ccsdsp` object to start.

```
x = createobj(cc, 'g_double')
NUMERIC Object stored in memory:
Symbol name      : g_double
Address         : [ 14648 0]
Data type       : double
Word size      : 64 bits
Address units per value : 8 au
Representation  : float
Size           : [ 1 ]
Total address units : 8 au
Array ordering  : row-major
Endianness     : little

read(x)

ans =
    17.0010
set(x, 'represent', 'signed')
```



```
read(x)

ans =
  4.6255e+018
```

Take care when you change the value of the `represent` property to `float`. Only change this property when the word referenced by the object is at least 32 bits.

As one example, if an object is a 16-bit integer where `represent=signed`, you cannot change the value for `represent` to `float` because to access floating point data, the data must be at least 32 bits long.

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 target 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 target, 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 Multiple Installed Versions 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 Link for Code Composer Studio attaches to when you create a `ccsdsp` 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 = ccstdsp
```

MATLAB Link for Code Composer Studio starts the CCS version you last used. If you last used your C5000 version, the `cc` object access the C5000 version.

Workaround

To make your `ccstdsp` object access the correct target:

- 1** Start and close the appropriate CCS version before you create the `ccstdsp` object in MATLAB.
- 2** Create the `ccstdsp` object using the `boardnum` and `procnum` properties to select your target, if needed.

Recall that `ccsboardinfo` returns the `boardnum` and `procnum` values for the targets that CCS recognizes.

Changing CCS Versions During a MATLAB Session

You can use only one version of CCS in a single MATLAB session. Link for Code Composer Studio does not support using multiple versions of CCS in a MATLAB session. To use another CCS version, exit MATLAB and restart it. Then create your links to the new version of CCS.

createobj and address Return Inconsistent Page Information on C5xxx Targets

The address page of a C5xxx variable given by the `createobj` and `address` methods are sometimes inconsistent.

Though the pages are not the same, they are pointing to the same location in memory. When you write data to one location, such as `page=0`, and then you read from the other location, such as `page=1`, you return the same value. The following example should help clarify the situation.

Getting information about an object in memory through `createobj`:

```
ibufobj = createobj(cc, 'ibuf')
```

```

NUMERIC Object stored in memory:
Symbol name : ibuf
Address : [ 8913 1] <----- Notice the Page = 1 indication.
Data type : int
Word size : 16 bits
Address units per value : 1 au
Representation : signed
Size : [ 10 ]
Total address units : 10 au
Array ordering : row-major
Endianness : big

```

Now use address to get the same information about ibuf.

```

address(cc,'ibuf')
ans =
8913 0 <----- Notice the Page = 0 indication.

```

Though the pages appear to be different, reading either of the two yields the same result.

```

read(ibufobj)
ans =
Columns 1 through 10
1 0 0 0 0 0 0 0 0 0

read(cc,address(cc,'ibuf'),'int16',10)
ans =
Columns 1 through 10
1 0 0 0 0 0 0 0 0 0

```

Modify the second element of ibuf to 2 in CCS and then read the value from MATLAB through both ibuf and ibufobj.

```

read(cc,address(cc,'ibuf'),'int16',10)
ans =
Columns 1 through 10
1 2 0 0 0 0 0 0 0 0

```

```
read(ibufobj)
ans =
Columns 1 through 10
1 2 0 0 0 0 0 0 0 0
```

For the final check, modify `ibuf` from MATLAB using `ibufobj` and then read from MATLAB. The results are the same.

```
write(ibufobj,1:10)
>> read(ibufobj)
ans =
Columns 1 through 10
1 2 3 4 5 6 7 8 9 10
>> read(cc,address(cc,'ibuf'),'int16',10)
ans =
Columns 1 through 10
1 2 3 4 5 6 7 8 9 10
```

Modify `ibuf` from MATLAB using `address` and then read `ibuf` from MATLAB. Again the results are the same.

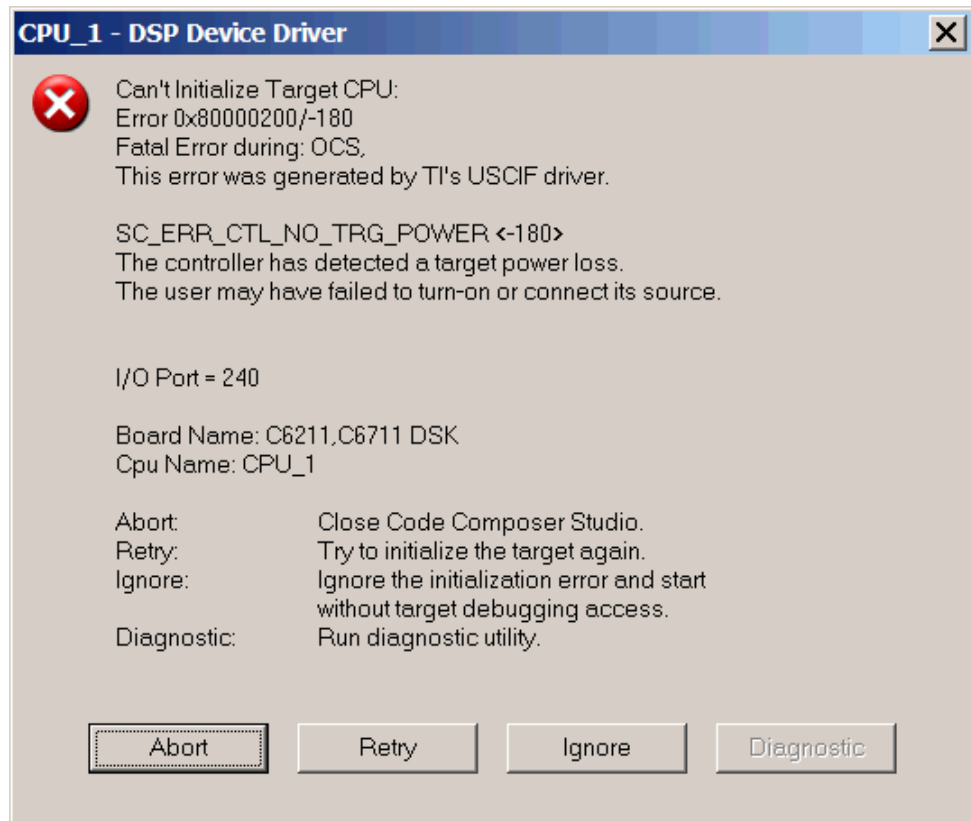
```
write(cc,address(cc,'ibuf'),int16(10:-1:1))
read(cc,address(cc,'ibuf'),'int16',10)
ans =

Columns 1 through 10
10 9 8 7 6 5 4 3 2 1

read(ibufobj)
ans =
Columns 1 through 10
10 9 8 7 6 5 4 3 2 1
```

MATLAB Hangs When Code Composer Studio Cannot Find a Target

In MATLAB, when you create a `ccsdsp` object, the construction process for the object automatically starts CCS. If CCS cannot find a target 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 target.



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 to open CCS, such as when you create a `ccsdsp` object, the system returns control to MATLAB after a considerable delay, and issues this warning:

```
??? Unable to establish connection with Code Composer Studio.
```

- **Ignore** — Launches CCS without connecting to any target. 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 launch CCS, you get control immediately and Link for Code Composer Studio creates the `ccsdsp` object.

Since CCS is not connected to a target, you cannot use the object to perform target operations from MATLAB, such as loading or running programs.

- **Retry** — CCS tries again to initialize the target. If CCS continues not to find your hardware target, the same DSP Device Driver dialog box reappears. This process continues until either CCS finds the target 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.

Different Read Techniques Appear to Return Different Values

When you read the value of a pointer on your C54x target, the result can seem to depend on how you read the value. If you check the value in the MATLAB workspace browser, you see that read returns the same values in both cases.

The following example shows this happening with the variable `g_vptr`.

In source code you have the following prototype.

```
double mydouble;  
void *g_vptr = &mydouble;
```

In MATLAB, perform these operations to set a value to read.

```
ptr = createobj(cc, 'g_vptr');  
convert(ptr, 'Double *'); % Use double to represent ptr.  
ptr1 = deref(ptr1);  
write(ptr1, 10^20);
```

With the variables defined as shown, reading the data returns different results depending on which read syntax you use to read the data.

```
result1 = read(cc, ptr1.address, 'single') % Return the value...  
                                             % in single format.
```

returns

```
1.0000000e+020
```


and

```
result2 = read(ptr1)
```

returns

```
1.000000020040877e+020
```

The results appear to differ after the seventh decimal place. If you go to the MATLAB workspace browser to look at the values, you see that `result1` and `result2` are the same. The apparent difference occurs because the syntax

```
result1 = read(cc, ptr1.address, 'single')
```

explicitly states that `result1` is returned in `single` format, as controlled by the `single` input argument. On the other hand,

```
result2 = read(ptr1)
```

converts the data from `ptr1` on the target to double-precision format. That is,

```
result2 = double(result1)
```

In general, use `read()` when you want to access the data on the target. Use the `read(object, ...)` syntax when you are manipulating data on the target.

A

- abbreviate property names 1-64
- abstract class 2-5
- access properties 1-63
- address property 2-120
- apiversion 1-71
- apiversion property 2-121
- arrayorder property 2-122

B

- base class 2-5
- behavior 2-5
- binarypt property 2-123
- bitfield object 2-18
- bitsperstorageunit property 2-123
- boardnum 1-72
- boardnum property 2-123

C

- C and library functions compared 2-58
- CCS IDE links
 - tutorial about using 1-11
- ccsappexe 1-72
- ccsappexe property 2-124
- ccsdsp 1-61
- charconversion property 2-124
- class 2-6
- class diagram 2-6
- class, abstract 2-5
- class, base 2-5
- class, container 2-6
- constructor 2-6
- container class 2-6
- createobj function 1-13
- custom data types 2-109
- custom type definitions 2-109

D

- Data Type Manager 2-109
- diagram
 - object 2-7
- diagram, class 2-6

E

- embedded object properties
 - address 2-120
 - apiversion 2-121
 - arrayorder 2-122
 - binarypt 2-123
 - bitsperstorageunit 2-123
 - boardnum 2-123
 - ccsappexe 2-124
 - endianness 2-125
 - label 2-127
 - link 2-128
 - member 2-129
 - memname 2-130
 - memoffset 2-131
 - name 2-132
 - numberofstorageunits 2-132
 - numChannels 2-133
 - page 2-134
 - postpad 2-135
 - prepad 2-135
 - procnum 2-136
 - represent 2-136
 - rtdx 2-139
 - rtdxChannel 2-140
 - storageunitspervalue 2-142
 - timeout 2-143
 - typestring 2-145
 - value 2-145
 - wordsize 2-146
- embedded objects
 - bitfield 2-18
 - enum 2-21

- function 2-42
- numeric 2-15
- pointer 2-24
- renum 2-33
- rnumeric 2-30
- rpointer 2-36
- rstring 2-39
- string 2-27
- structure 2-46
- type 2-52

endianness property 2-125

enum object 2-21

export filters to CCS IDE from FDATool 3-1

- select the export data type 3-8
- set the Export mode option 3-4
- set the Target selection options 3-14
- set Variable names in C header file 3-6
- set Variable names in target symbol table 3-6

exporting filters to CCS IDE from FDATool tutorial 3-10

F

FDATool 3-1

- See also* export filters to CCS IDE from FDATool

filename property 2-126

function 2-6

- createobj 1-13

function object 2-42

- using with declare 2-57

functions

- library 2-59
- library and C 2-58
- overloading 1-68

G

getting properties 1-65

H

hardware requirements for MATLAB Link for Code Composer Studio 1-8

I

inheritance 2-7

inputnames property 2-126

inputvars property 2-127

instance 2-7

instantiation 2-7

L

label property 2-127

library and C functions compared 2-58

library functions 2-59

link filters properties

- getting 1-67

link properties

- about 1-69
- apiversion 1-71
- boardnum 1-72
- ccsappexe 1-72
- numchannels 1-72
- page 1-73
- procnum 1-73
- quick reference table 1-69
- rtdx 1-73
- rtdxchannel 1-74
- setting 1-67
- timeout 1-75
- version 1-75

link properties, details about 1-71

link property 2-128

links

- closing CCS IDE 1-36
- closing RTDX 1-55
- communications for RTDX 1-46
- creating links for CCS IDE 1-15

- creating links for RTDX 1-43
- details 1-71
- introducing the function object
 - tutorial 2-78
- introducing the links for CCS IDE
 - tutorial 1-11
- introducing the tutorial for using links for RTDX 1-38
- loading files into CCS IDE 1-18
- quick reference 1-69
- running applications using RTDX 1-48
- selecting targets for CCS IDE 1-14
- tutorial about using links for CCS IDE 1-11
- tutorial about using links for RTDX 1-38
- working with your target 1-21

M

- MATLAB Link for Code Composer Studio
 - hardware and OS requirements 1-8
 - listing link functions 1-59
 - requirements for TI software 1-9
- member property 2-129
- memname property 2-130
- memoffset property 2-131
- method 2-7
 - function 2-7

N

- name property 2-132
- numberofstorageunits property 2-132
- numchannels 1-72
- numChannels property 2-133
- numeric object 2-15

O

- object 2-7
 - aggregation 2-5

- behavior 2-5
- ccsdsp 1-61
- class 2-6
- composition 2-6
- constructor 2-6
- function 2-6
- handle 2-6
- inheritance 2-7
- instance 2-7
- method 2-7
- property 2-8
- state 2-8
- structure 2-8
- object diagram 2-7
 - See also* class diagram
- object, instantiation 2-7
- object-based programming 2-8
- object-oriented programming 2-8
- offset property 2-134
- OS requirements for MATLAB Link for Code Composer Studio 1-8
- outputvar property 2-134
- overloading 1-68

P

- page 1-73
- page property 2-134
- pointer object 2-24
- postpad property 2-135
- prepad property 2-135
- processor registers, saved 5-7 5-92
- procnum 1-73
- procnum property 2-136
- programming
 - object-based 2-8
 - object-oriented 2-8
- properties
 - abbreviating names 1-64
 - link properties 1-69

- referencing directly 1-67
- retrieving 1-63
 - function for 1-65
- retrieving by direct property
 - referencing 1-67
- setting 1-63
- property 2-8
 - charconversion 2-124
 - filename 2-126
 - inputnames 2-126
 - inputvars 2-127
 - offset 2-134
 - outputvar 2-134
 - savedregisters 2-141
 - size 2-140
 - type 2-144
 - typelist 2-144
 - typename 2-144
 - wordsize 2-146
- property values
 - abbreviating 1-66

R

- registers, saved 5-7 5-92
- renum object 2-33
- represent property 2-136
- rnumeric object 2-30
- rpointer object 2-36
- rstring object 2-39
- rt dx 1-73
- RTDX links
 - tutorial about using 1-38
- rt dx property 2-139 to 2-140
- rt dxchannel 1-74

S

- saved processor registers 5-7 5-92
- savedregisters property 2-141

- set properties 1-63
- size property 2-140
- state 2-8
- storageunitspvalue property 2-142
- string object 2-27
- structure 2-8
- structure object 2-46
- structure-like referencing 1-67
- subclass 2-8
 - superclass 2-8
- superclass 2-8
 - subclass 2-8

T

- timeout 1-75
- tutorials
 - embedded objects and HIL 2-77
 - function calls 2-77
 - links for CCS 1-11
 - links for RTDX 1-38
- type object 2-52
- type property 2-144
- typedefs 2-111
 - about 2-109
 - adding 2-111
 - managing 2-111
 - removing 2-111
- typelist property 2-144
- typename property 2-144
- typestring property 2-145

U

- use declare with function objects 2-57

V

- value property 2-145
- version 1-75

W

wordsize property 2-146