MATLAB® Link for Code Composer Studio™ Development Tools

Computation

Visualization

Programming

User's Guide

Version 1

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

MATLAB Link for Code Composer Studio Development Tools User's Guide

Printing History: July 2002 Online only New for Version 1.0 (Release 13)

Contents

Preface

1 Introducing Links and Embedded Objects

Preface

About MATLAB Link for Code Composer Studio Development Tools

MATLAB 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 MATLAB 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 MATLAB Link for Code Composer Studio” on page 1-3.
- 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 “About Objects for MATLAB Link Software” on page 2-1 for more information about using the embedded objects, their properties, and their methods.

Supported Hardware for Links to CCS IDE and RTDX

Using the C6000 target in Real-Time Workshop®, the MATLAB Link for Code Composer Studio supports the following boards produced by TI.

Supported Board Designation	Board Description
TMS320C6701 EVM	C6701 Evaluation Module.

Supported Board Designation	Board Description
TMS320C6711 DSK	C6711 DSP Starter Kit.
C6xxx simulators in CCS	Digital signal processor simulators in CCS. You can generate code to the simulators if you have Embedded Target for the TI TMS320C6000 DSP Platform product, and use CCS and RTDX links with them.

Links for RTDX and CCS work with any board that CCS supports. You can link to any hardware that appears in the CCS Setup Utility.

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

- Link for Code Composer Studio IDE — 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.
- Link for Real-Time Data Exchange Interface — provides a communications pathway between MATLAB and digital signal processors installed on your PC. Using objects in the MATLAB 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.
- Embedded Objects—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.

Related Products

The MathWorks provides several products that are especially relevant to the tasks you can perform with the MATLAB Link for Code Composer Studio.

For information about the products and hardware you need to run the MATLAB Link for Code Composer Studio, refer to “Requirements for MATLAB Link for Code Composer Studio” on page 1-3.

For more information about any of these products, refer to either

- The online documentation for that product, if it is installed or you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`. Navigate to the “products” area

Note The toolboxes listed below include functions that extend MATLAB capabilities. The blocksets include blocks that extend Simulink® capabilities.

Product	Description
Control System Toolbox	Design and analyze feedback control systems
Data Acquisition Toolbox	Capture and send data from plug-in data acquisition boards
DSP Blockset	Design and simulate DSP systems
Embedded Target for the TI TMS320C6000 DSP Platform	Use Simulink and the Real-Time Workshop to create models and generate target-specific code for supported TI hardware
Filter Design Toolbox	Design and analyze advanced floating-point and fixed-point filters
Image Processing Toolbox	Perform image processing, analysis, and algorithm development

Using This Guide

Expected Background

This document introduces you to using MATLAB Link for Code Composer Studio Development Tools. To get the most out of this manual, readers should be familiar with MATLAB and its associated programs, such as DSP Blockset and Simulink. We do not discuss details of digital signal processor operations and applications. For more information about digital signal processing, you may find one or more of the following books helpful:

- McClellan, J. H., R. W. Schafer, and M. A. Yoder, "*DSP First: A Multimedia Approach*," Prentice Hall, 1998.
- Lapsley, P., J. Bier, A. Sholam, and E. A. Lee, "*DSP Processor Fundamentals Architectures and Features,*" IEEE Press, 1997.
- Steiglitz, K, "*A Digital Signal Processing Primer*," Addison-Wesley Publishing Company, 1996.

For information about Code Composer Studio and Real-Time Data Exchange™ (RTDX™), refer to your Texas Instruments documentation for each product.

If You Are a New User

New users should read "Introducing Links and Embedded Objects" on page 1-1. This introduces the MATLAB Link for Code Composer Studio environment — the required software and hardware, installation requirements, and the board configuration settings that you need. To introduce the links ideas, the section includes discussions about objects and tutorials about using links and RTDX.

If You Are an Experienced User

All users should read "About Objects for MATLAB Link Software" on page 2-1 for information and examples about embedded objects, such as the properties and methods of each object, and a tutorial about working with your CCS project from MATLAB. As experienced users, you know about the link object that enables communications between MATLAB and Code Composer Studio. This section offers details about the objects for getting access to and manipulating the contents of memory, storage registers, and functions in projects in Code Composer Studio. Using the objects is the first step towards providing you with hardware-in-the-loop capability while you develop your applications.

Organization of the Document

Chapter	Description
Preface	Introduces the MATLAB Link for Code Composer Studio Development Tools environment — the software and hardware you need, and the related products that may be of interest.
Introducing Links and Embedded Objects	Provides information about using the link software to connect to Texas Instruments Code Composer Studio Integrated Development Environment and to open Real-Time Data Exchange channels to target digital signal processors.
About Objects for MATLAB Link Software	Reveals the secrets and detail about the embedded objects in the software. Here you find descriptions of the properties and methods for each object you create in MATLAB.
Link Functions Reference	Provides reference information for the functions in the MATLAB Link for Code Composer Studio. The functions listed work with the links for RTDX Interface and CCS IDE Interface.

Configuration Information

To determine whether the MATLAB 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.

```
Link for Code Composer Studio(tm)
Version 1.0   (R13)  19-Apr-2002

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 MATLAB 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 Board                              Proc Processor                Processor
 Num  Name                               Num  Name                     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 MATLAB Link for Code Composer Studio to operate with CCS, the CCS IDE must be able to run on its own.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`    5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Introducing Links and Embedded Objects

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

- Links to Code Composer Studio Integrated Development Environment (CCS IDE)
- Links to 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”
- “Properties and Property Values”
- “Setting and Retrieving Property Values”
- “Setting Property Values Directly at Construction”
- “Setting Property Values with set”
- “Retrieving Properties with get”
- “Direct Property Referencing to Set and Get Values”
- “Overloaded Functions for Links”

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 MATLAB Link for Code Composer Studio

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

MATLAB Link for Code Composer Studio runs on Microsoft Windows 98, Windows NT 4.0 Workstation and Server, and Windows 2000 platforms.

Platform Requirements—Hardware and Operating System

To run the MATLAB 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
- CD-ROM drive
- Microsoft Windows 98, Windows NT 4.0 Server or Workstation, or Windows 2000

Refer to your documentation from The MathWorks for more information on installing the software required to support MATLAB Link for Code Composer Studio, as shown in Table 1-1.

Table 1-1: Prerequisites for Using MATLAB Link for Code Composer Studio Software for Targeting

Installed Product	Additional Information
MATLAB 6.1	Core software from The MathWorks
Signal Processing Toolbox 5.0 or later	(Recommended) Software package for analyzing signals, processing signals, and developing algorithms

For information about the software required to use the MATLAB 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, MATLAB Link for Code Composer Studio requires that you install the Texas Instruments development tools and software listed in Table 2-2. Installing Code Composer Studio IDE for the C6000 series installs the software shown in the table.

Table 1-2: Required TI Software for MATLAB Link for Code Composer Studio

Installed Product	Additional Information
Assembler	Creates object code (`.obj`) for C6000 boards from assembly code.
Compiler	Compiles C code from the blocks in Simulink models into object code (`.obj`). As a by-product of the compile process, you get assembly code (`.asm`) as well.
Linker	Combines various input files, such as object files and libraries.
Code Composer Studio 2.1	Texas Instruments integrated development environment (IDE) that provides code debugging and development tools.
TI C6000 miscellaneous utilities	Various tools for developing applications for the C6000 digital signal processor family.
Code Composer Setup Utility	Program you use to configure your CCS installation by selecting your target boards or simulator.

In addition to the TI software, you need one or more of the following in any combination:

- One or more Texas Instruments TMS320C6701 Evaluation Modules
- One or more TMS320C6711 DSP Starter Kits

- One or more boards that CCS supports in the Setup utility, either C5000™ or C6000™ digital signal processor platforms
- One or more simulators from CCS

To use C5000 platforms from TI, install the Code Composer Studio IDE version that supports C5000 products.

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

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 MATLAB Link for Code Composer Studio. For a discussion of the embedded objects that are also part of this product, refer to “About Objects for MATLAB Link Software” on page 2-1. 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 MATLAB 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 2-4, helpful to explain the terms used in this User’s 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 1-9.

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 “Tables of Link Software Functions” on page 3-3, and in more detail in “Link Properties” on page 1-14. These properties are set to default values when you construct links.

Properties and Property Values

Links (or objects) in this MATLAB 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 MATLAB Link for Code Composer Studio for Texas Instruments DSP link property values either:

- Directly when you create the link
- By using the `set` function with an existing link

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 1-14, 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 specific property values
- Display a list of the link properties showing all allowed values for each property and the default setting for each property

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.

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

```
get(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');
```

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

```
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 = ccsdsp;
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 MATLAB 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 links (link objects). 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 MATLAB 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 MATLAB 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 MATLAB 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 MATLAB 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.

Table 1-3: Properties for the Links in MATLAB Link for Code Composer Studio

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`—The board with which the link communicates
- `procnum`—The processor on the board, if the board has multiple processors
- `timeout`—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 `appversion` 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 `appversion` 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 `ccsdsp` 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 MATLAB 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:

.rtdxchannel{i,1}	Channel name of the ith-channel, i from 1 to the number of open channels
.rtdxchannel{i,2}	Handle for the ith-channel
.rtdxchannel{i,3}	Mode of the ith-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
```

Tutorial 2-1—Using Links and Embedded Objects

The Link for Code Composer™ Studio IDE (CCS IDE), a part of the MATLAB 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 MATLAB 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.

For 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 1-31 portion of the tutorial.

Using the read and write functions gets a bit complicated. MATLAB supports only double-precision values for calculations, but 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.
  - `boardprocsel`—select the board to target. Although you can use this generally, the MATLAB 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
  - `disp`—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
- MATLAB 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 MATLAB 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

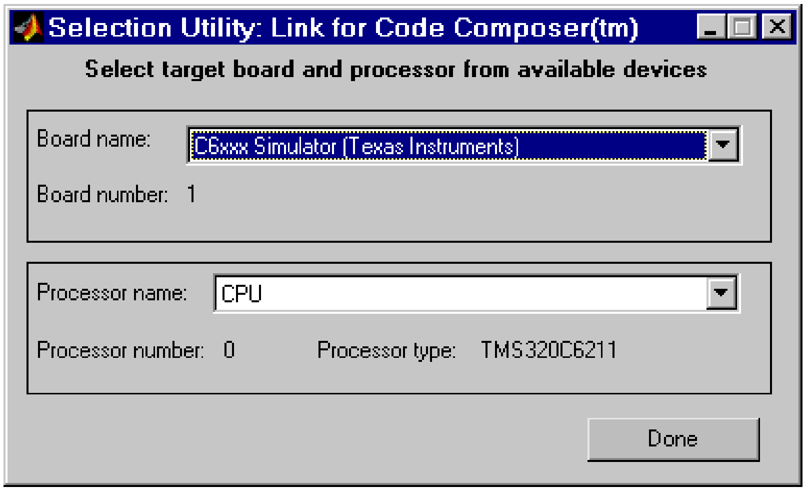
```
ccsboardinfo
```

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, `boardprocsel`, to select a target board, type

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

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



Select target board and processor from available devices

Board name: C6xxx Simulator (Texas Instruments)

Board number: 1

Processor name: CPU

Processor number: 0

Processor type: TMS320C6211

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.

`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 Type `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 `disp(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 MATLAB 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
- `disp`—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 =

       procname: 'CPU'
    isbigendian: 0
         family: 320
      subfamily: 103
      revfamily: 1
        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 MATLAB 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 which 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, do one of the following depending on the class of your target processor.

   C54xx processor family—Type the following commands to load the project file. Notice that these functions also change your CCS IDE working directory.

   ```
   projfile = fullfile(matlabroot,'toolbox','tiddk','tidemos',...
   'ccstutorial','ccstut_54xx.pjt')
   projpath = fileparts(projfile)
   open(cc,projfile) % Open project file
   cd(cc,projpath) % Change working directory of Code Composer(only)
   ```

   C6x11 processor family—Type the following commands to load the project file. Notice that these functions also change your CCS IDE working directory.

   ```
   projfile = fullfile(matlabroot,'toolbox','tiddk','tidemos',...
   'ccstutorial','ccstut_6x11.pjt')
   projpath = fileparts(projfile)
   open(cc,projfile) % Open project file
   cd(cc,projpath) % Change Code Composer working directory
   ```

   C6x0x processor family—Type the following commands to load the project file. Notice that these functions also change your CCS IDE working directory.

```
projfile = fullfile(matlabroot,'toolbox','tiddk','tidemos',...
'ccstutorial','ccstut_6x0x.pjt')
projpath = fileparts(projfile)
open(cc,projfile) % Open project file
cd(cc,projpath) % Change Code Composer working directory
```

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,'a.out')` to load the target execution file.

4 You now have a loaded program file and associated symbol table. To determine the memory address of the global symbol `ddat`, 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 MATLAB 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 MATLAB 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`. They are declared and initialized in lines 10 and 11 of the source code. You access these processor memory arrays from MATLAB using the functions `read` and `write`.

The MATLAB 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
```

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 returns

```
idatv =


   1     0   508     0
```

Because you requested 16-bit integers, whose maximum value is 512, the values 647 and 7000 come back as zeros since they cannot be represented as 16-bit integers. Using `int32` would have returned the full values for all the data in `idatv`.

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     0   508     0
```

Because you requested 16-bit integers, whose maximum value is 512, the values 647 and 7000 come back as zeros since they cannot be represented as 16-bit integers. Using `int32` would have returned the full values for all the data in `idatv`.

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 MATLAB 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 MATLAB 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.

MATLAB 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 MATLAB 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')
```

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`.

```
NUMERIC Object
  Symbol Name             : idat
  Address                 : [ 40060 0]
  Wordsize                : 16 bits
  Address Units per value : 2 AU
  Representation          : signed
  Binary point position   : 0
  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 equivalaent 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
  Symbol Name             : idat
  Address                 : [ 40060 0]
  Wordsize                : 16 bits
  Address Units per value : 2 AU
  Representation          : unsigned
  Binary point position   : 0
  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 MATLAB Link for Code Composer Studio, you can manipulate more complex data like strings, struactures, bitfields, eumerated 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 datatypes 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
  Symbol Name         : myStruct
  Address             : [ 40032 0]
  Size                : [ 1 ]
  Total Address Units : 28 AU
  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
  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
```

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 :
  Symbol Name             : myString
  Address                 : [ 40104 0]
  Total wordsize          : 8 bits
  Address Units per value : 1 AU
  Representation          : signed
  Binary point position   : 0
  Size                    : [ 29 ]
  Total address units     : 29 AU
  Array ordering          : col-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 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 MATLAB Link for Code Composer Studio have COM handles to CCS. Until you delete these handles, the CCS process (`cc_app.exe` in the 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 your create with `ccsdsp` and `createobj`. To clean up the objects created during the tutorial, the tutorial program enters

```
clear cc 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 MATLAB 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.

Tutorial 2-2—Using Links for RTDX

The MATLAB 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 MATLAB 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 CCS IDE simulator to run this tutorial. Within the tutorial we use the TMS320C6701 EVM as the target board, with the C6701 DSP on the C6701 EVM 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.

In all cases, developers create algorithms that they need to accomplish the desired result. Once they have the algorithms, developers 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. The MATLAB 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 MATLAB 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` and `disp`—return the results of functions `get` and `set`. When you omit the closing semicolon (;) on a function, `disp` 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 MATLAB 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 considered 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 MATLAB Link for Code Composer Studio to develop your processing system.

In this tutorial you use an executable program named `tutorial_6xevm.out` as your application. When you use the RTDX and CCS IDE links to develop applications, replace `tutorial_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 the dot notation (direct property referencing) to access functions and link properties. For example, we use

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

to open and configure `ichan` for write mode. You could use an equivalent syntax instead 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 MATLAB Link for Code Composer Studio and the links do not distinguish the source of the executable—whether you used the MATLAB 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 MATLAB Link for Code Composer Studio provides the links to the processor.

Note Program `tutorial_6xevm.out` targets the C6701 EVM. We compiled, built, and linked the program as an executable to run on the C6701 digital signal processor. To use the tutorial without changes, target your C6701 EVM 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 MATLAB 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 type

```
ccsboardinfo
```

at the prompt, the MATLAB Link for Code Composer Studio returns a list like the following one that identifies the boards and processors in your computer.

```
Board   Board                          Proc   Processor               Processor
Num     Name                           Num    Name                    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.

```
tgt_dir = fullfile(matlabroot,'toolbox','tiddk','tidemos','tutorial');
cd(cc,tgt_dir); % Or cc.cd(tgt_dir)
dir(cc); % Or cc.dir
```

4 You have reset the directory path to find the tutorial file. Now open the file.

```
cc.open('tutorial_6xevm.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 MATLAB 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 MATLAB 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 reverify 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 = 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,readdmsg('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 MATLAB 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('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',[4 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)      % 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 MATLAB 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 = ccsdsp('boardnum',1)` at the beginning of this tutorial, the `ccsdsp` 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 MATLAB 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
```

About Objects for MATLAB Link Software

Introduction to Objects

Within your MATLAB 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, MATLAB 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:

Object Name	Inherits From	Description
Bitfield	Memory Class	Describes and provides access to the contents of a bitfield defined in your code
Enum	Numeric Class	Describes and provides access to the contents of an enumerated datatype stored in memory defined in your code
Numeric	Memory Class	Describes and provides access to the contents of a numeric datatype stored in memory defined in your code
Pointer	Numeric Class	Describes and provides access to the contents of a pointer stored in a memory location on your target
Renum	Rnumeric Class	Describes and provides access to the contents of a pointer stored in a memory location on your target

Object Name	Inherits From	Description
Rpointer	Rnumeric Class	Describes and provides access to the contents of a pointer stored in a register on your target
Rstring	Rnumeric Class	Describes and provides access to the contents of a string stored in a register on your target
String	Numeric Class	Describes and provides access to the contents of a pointer stored in a memory location on your target
Structure	None	Describes and provides access to the contents of a structure stored in memory on your target

In the Inherits From column you see the name of another class. Objects that inherit from another class contain all the properties and methods of the inherited from class as well as their unique properties.

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 MATLAB 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 MATLAB Links for Code Composer Studio requires discussing the objects, classes, properties, and methods you use to manipulate and access data. To insure we use the same terms and understand them in the same way, this section provides definitions of some terms commonly used throughout the User's Guide.

Definitions of Useful Object-Oriented Terms

abstract class	a class without instances. Abstract classes expect that their concrete subclasses will add to their structure and behavior.
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 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. Often, type and class are used interchangeably, although they are slightly different. In this User's Guide the terms are interchangeable.
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.
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 used interchangebly in this guide.

handle	a means to access any object that MATLAB Link for Code Composer Studio creates. Used in this User's Guide to refer to the object, interchangeably with 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.
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. Or taken another way, a collection of properties and methods. Some sources call properties "variables."
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	programming method that organizes programs as cooperative collections of objects, each of which represents an instance of some type, and whose types are members of an heirarchy, united through relationships that are not inheritance relationships.
object-oriented progamming	programming implementation that organizes programs as cooperative collections of objects, each of which represents an instance of some class, and whose classes are members of a heirarchy of classes united through inheritance relationships.
property	part of an object—a variable to some. Also called attribute, it is part of the structure that defines the state of an object.
subclass	a class that inherits from one or more classes, called its superclasses.
superclass	a class that other classes inherit from. The inheriting classes are called subclasses.
state	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.
structure	the concrete representation of the state of an object.

For more information about objects and working with their properties and methods (or functions) refer to "Constructing Link Objects" on page 1-6.

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 Code Composer Studio. Only by using `read` and `write` can you acess and change information on your target or in your project in CCS.

Determining an Object's 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 and `whos` lets you do this.

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.

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`.

About the Relationships Between Objects

MATLAB Links 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
memory	An abstract class. The numeric class 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 memory class always describe objects that reside in DSP memory on your target.
register	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 register class always describe objects that reside in DSP registers on your target.

The Sub Classes

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 `createobj`. Numeric inherits from the abstract class memory.
enum	A subclass of the numeric class. You can create an object of this class using `createobj`.
pointer	A subclass of the numeric class. You can create an object of this class using `createobj`.
string	A subclass of the numeric class. You can create an object of this class using `createobj`.

Class Name	Description
bitfield	A subclass of the memory class. You can use `createobj` to make a bitfield object.
rnumeric	A superclass from which the renum, rpointer, and rstring subclasses inherit properties and methods. You can create an object of this class using `createobj`. Rnumeric inherits from the abstract class register.
renum	A subclass of the register class. You can create an object of this class using `createobj`.
rpointer	A subclass of the register class. You can create an object of this class using `createobj`.
rstring	A subclass of the register class. You can create an object of this class using `createobj`.

Other classes

Class Name	Description
structure	A class containing information about a structure in memory on your target.

Class Diagrams for the MATLAB Link for Code Composer Studio

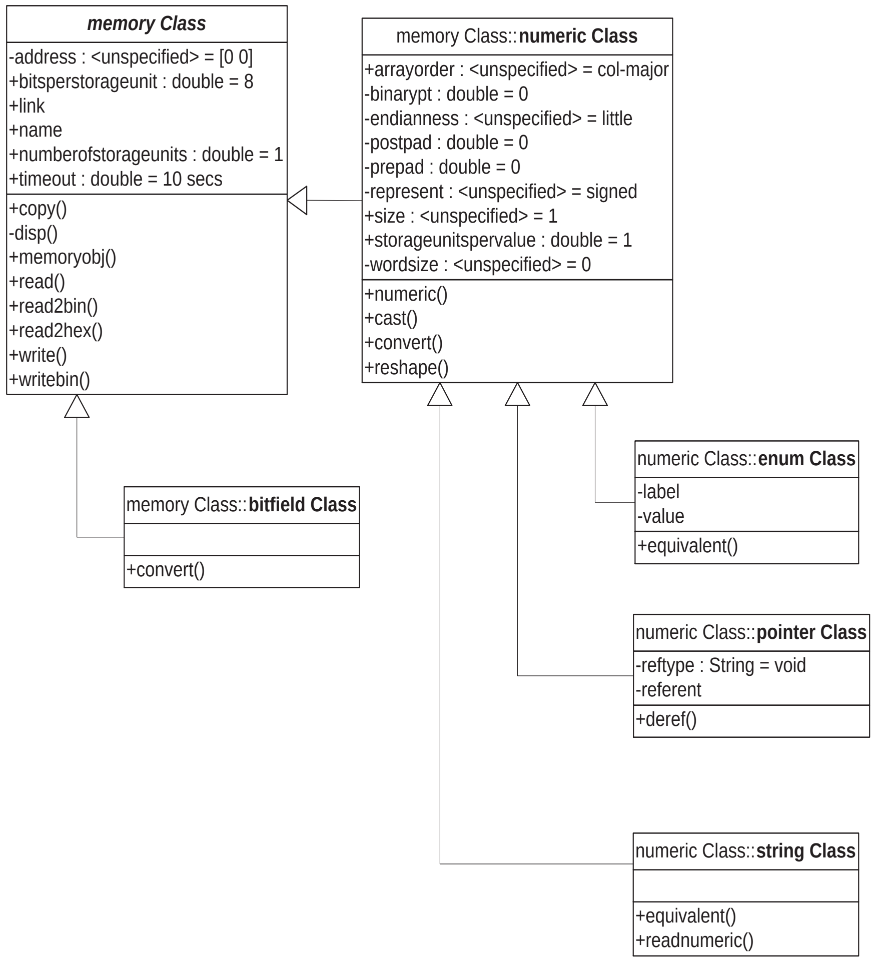
One of the most important features of object-oriented programming is the relationship between the classes that compose the system. Class relationships lend themselves to graphical layout like a tree structure, where the structure of the tree shows clearly the super- and subclasses, the base classes, 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:

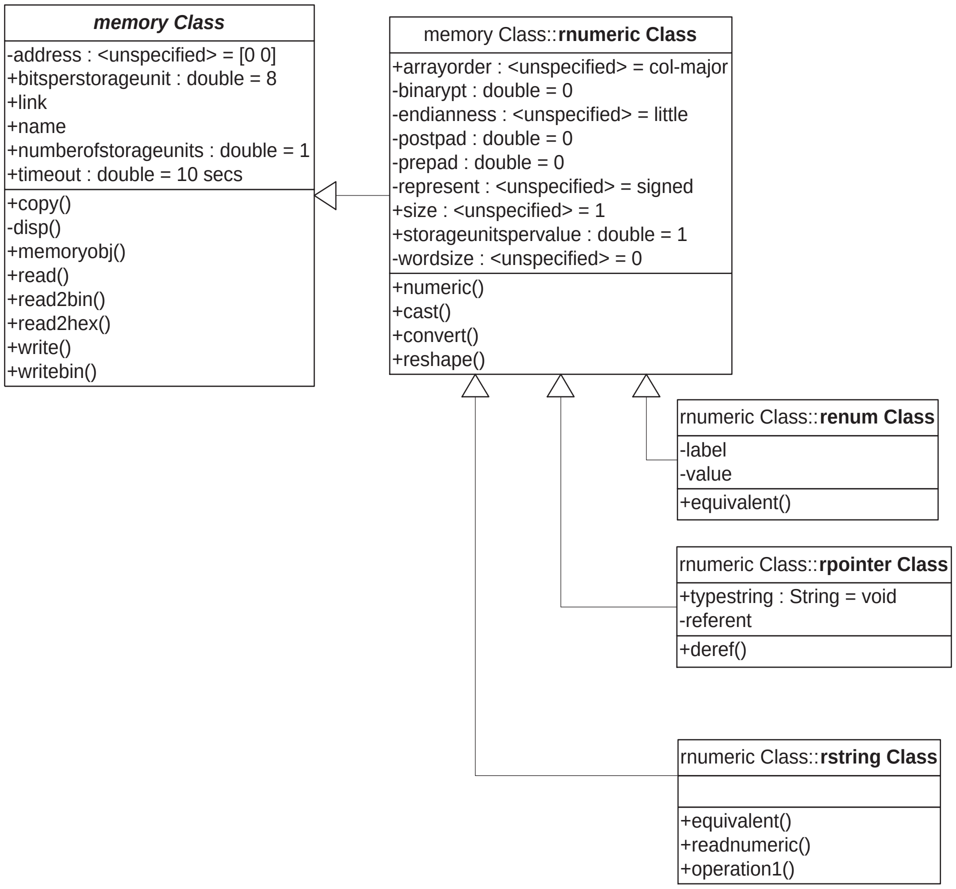
- "Numeric Objects—Their Methods and Properties" on page 2-15
- "Bitfield Objects—Their Methods and Properties" on page 2-17
- "Enum Objects—Their Methods and Properties" on page 2-19
- "Pointer Objects—Their Methods and Properties" on page 2-21
- "String Objects—Their Methods and Properties" on page 2-23
- "Rnumeric Objects—Their Methods and Properties" on page 2-25
- "Renum Objects—Their Methods and Properties" on page 2-27
- "Rpointer Objects—Their Methods and Properties" on page 2-29
- "Rstring Objects—Their Methods and Properties" on page 2-31
- "Structure Objects—Their Methods and Properties" on page 2-32

Detailed descriptions of the properties appear in the reference pages in section "Reference for the Properties of the Objects" on page 2-34.

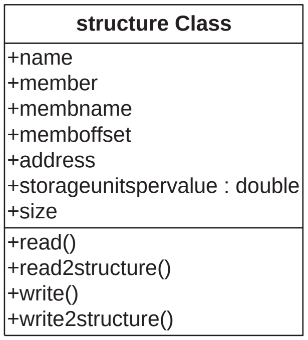
Class Diagram of the Memory Class


memory Class
-address : <unspecified> = [0 0]
+bitsperstorageunit : double = 8
+link
+name
+numberofstorageunits : double = 1
+timeout : double = 10 secs
+copy()
-disp()
+memoryobj()
+read()
+read2bin()
+read2hex()
+write()
+writebin()
memory Class::numeric Class
+arrayorder : <unspecified> = col-major
-binarypt : double = 0
-endianness : <unspecified> = little
-postpad : double = 0
-prepad : double = 0
-represent : <unspecified> = signed
+size : <unspecified> = 1
+storageunitspervalue : double = 1
-wordsize : <unspecified> = 0
+numeric()
+cast()
+convert()
+reshape()
memory Class::bitfield Class
+convert()
numeric Class::enum Class
-label
-value
+equivalent()
numeric Class::pointer Class
-reftype : String = void
-referent
+deref()
numeric Class::string Class
+equivalent()
+readnumeric()

Class Diagram of the Register Class


memory Class
-address : <unspecified> = [0 0]
+bitsperstorageunit : double = 8
+link
+name
+numberofstorageunits : double = 1
+timeout : double = 10 secs
+copy()
-disp()
+memoryobj()
+read()
+read2bin()
+read2hex()
+write()
+writebin()
memory Class::**rnumeric Class**
+arrayorder : <unspecified> = col-major
-binarypt : double = 0
-endianness : <unspecified> = little
-postpad : double = 0
-prepad : double = 0
-represent : <unspecified> = signed
+size : <unspecified> = 1
+storageunitspervalue : double = 1
-wordsize : <unspecified> = 0
+numeric()
+cast()
+convert()
+reshape()
rnumeric Class::**renum Class**
-label
-value
+equivalent()
rnumeric Class::**rpointer Class**
+typestring : String = void
-referent
+deref()
rnumeric Class::**rstring Class**
+equivalent()
+readnumeric()
+operation1()

Class Diagram of the Structure 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.

Properties of Numeric Objects

Property Name	Date Type	Default Value	Description
`arrayorder`	`{'col-major' 'row-major'}`	col-major	Describes the ordering of the data moved from linear memory storage to n-dimensional arrays
`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
`name`	`mxArray`	None	Name of the embedded symbol in the symbol table
`address`	`mxArray`	[0 0]	Memory address of the symbol, in [Offset Page] format
`bitsperstorageunit`	`double`	8	Bits per addressable unit in the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the memory object

Methods of Numeric Objects

Name	Overloaded?	Description
`cast`	Yes	Copy an object and change the data type for a value at the same time
`convert`	No	Change the data type for a value
`reshape`	Yes	Change 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 bitfield 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.

Properties of Bitfield Objects

Property Name	**Date Type**	**Default Value**	**Description**
`name`	`mxArray`	None	Name of the embedded symbol in the symbol table
`address`	`mxArray`	[0 0]	Memory address of the symbol, in [Offset Page] format
`bitsperstorageunit`	`double`	8	Bits per addressable unit in the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the memory object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Bitfield Objects

Name	Overloaded?	Description
`copy`	Yes	Copy an existing memory object by creating a new pointer to the object
`disp`	Yes	Display the properties of the memory object
`read`	Yes	Return the contents of the memory location specified by the symbol
`write`	Yes	Write 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 enumerated 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.

Properties of Enum Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the embedded symbol in the symbol table
`address`	`mxArray`	[0 0]	Memory address of the symbol, in [Offset Page] format
`bitsperstorageunit`	`double`	8	Bits per addressable unit in the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the memory object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Enum Objects

Name	Overloaded?	Description
`equivalent`	No	Return 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.

Properties of Pointer Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the embedded symbol in the symbol table
`address`	`mxArray`	[0 0]	Memory address of the symbol, in [Offset Page] format
`bitsperstorageunit`	`double`	8	Bits per addressable unit in the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the memory object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Pointer Objects

Name	Overloaded?	Description
`deref`	No	Return 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.

Properties of String Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the embedded symbol in the symbol table
`address`	`mxArray`	[0 0]	Memory address of the symbol, in [Offset Page] format
`bitsperstorageunit`	`double`	8	Bits per addressable unit in the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the memory object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of String Objects

Name	Overloaded?	Description
`equivalent`	Yes	Return the equivalent numeric value for the input string

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 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.

Properties of Rnumeric Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the register symbol in the symbol table
`regname`	`mxArray`	None	Name of the register on the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the register object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Rnumeric Objects

Name	Overloaded?	Description
`cast`	No	Change the data type of the input argument to another data type
`convert`	No	Convert the data type to the specified data type
`reshape`	No	Reshape the object in MATLAB
`read`	Yes	Return the contents of the memory location specified by the symbol
`write`	Yes	Write one or more values to the memory 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 an 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.

Properties of Renum Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the register symbol in the symbol table
`regname`	`mxArray`	None	Name of the register on the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the register object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Renum Objects

Name	Overloaded?	Description
`equivalent`	No	Return the equivalent string or numeric
`write`	Yes	Write one or more values to the memory 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.

Properties of Rpointer Objects

Property Name	**Date Type**	**Default Value**	**Description**
`name`	`mxArray`	None	Name of the register symbol in the symbol table
`regname`	`mxArray`	None	Name of the register on the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the register object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Rpointer Objects

Name	Overloaded?	Description
`deref`	No	Return the data to which the specified pointer points
`read`	Yes	Return the contents of the memory location specified by the symbol
`write`	Yes	Write one or more values to the memory 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.

Properties of Rstring Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the register symbol in the symbol table
`regname`	`mxArray`	None	Name of the register on the signal processor
`numberofstorageunits`	`double`	1	Number of register units needed to represent the register object
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Rstring Objects

Name	Overloaded?	Description
`equivalent`	Yes	Return the equivalent numeric value for the input string

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.

Properties of Structure Objects

Property Name	Date Type	Default Value	Description
`name`	`mxArray`	None	Name of the C or Assembly function
`filename`	`mxArray`	None	Name of the file that contains the function
`address`	`mxArray`	None	Address of the function
`type`	`mxArray`	None	Return type for the function
`savedregs`	`mxArray`	None	Registers to preserve
`variables`	`mxArray`	None	The input and local arguments for the function
`inputvars`	`mxArray`	None	The input arguments for the function
`outputvars`	`mxArray`	None	The returned arguments from the function
`link`	`MATLAB handle`	None	Object handle that identifies the memory object
`timeout`	`double`	10 seconds	Timeout period for link functions/methods.

Methods of Structure Objects

Name	Overloaded?	Description
`getmember`	No	Return an object that accesses one member of a structure
`read`	Yes	Read a structure from the symbol table
`write`	Yes	Write changes or values to the structure in memory

Reference for the Properties of the Objects

This section presents details of the properties that apply to the objects in MATLAB 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, most 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 may not include all the features listed; in particular some pages may not provide examples or the range of valid property values or referrals.

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 (for memory locations) or alphanumeric (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 in not necessarily 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

`arrayorder` is a string with one of two possible values—`row-major` (C style intrepretation) or `column-major` (normal MATLAB style).

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

An 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 number of bits per address location (addressable unit) on the target. Memory locations and registers may have different values on a target. And 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 `storageunitspervalue`.

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 you CCS storage location.

Examples

If your CCS installation is in a folder called Applications on your D: drive, you might see a string like

```
'D:\Applications\ti\cc\bin\'
```

for the `ccsappexe` property value.

endianness

Description

Specifies whether to interpret the bit pattern in memory as 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 from `ddat` shows you the endianness for `ddat`.

```
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
```

isrecursive

Description

Indicates that the refernced pointer points to itself. When you dereference the pointer repeatedly, you may eventually find the pointer points to void. You should only see this in structures that have pointer that refer to the structure.

Characteristics

Double data type. isrecursive default is zero.

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 that 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
    storageunitspervalue: 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 stucture 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 the access myStruct, might look like

```
get(cvar)
                    name: 'myStruct'
                  member: [1x1 ccs.containerobj]
                membname: {'iy'  'iz'}
              memboffset: [0 24]
                 address: [40032 0]
    storageunitspervalue: 28
                    size: 1
    numberofstorageunits: 28
              arrayorder: 'row-major'
```

where `member` returns as a 1-by-1 MATLAB array with a handle to the object that contains it named `ccs.containerobj`.

membname

Description

Contains the names of the fields in a structure or union accessed by a structure object.

Characteristics

`membname` is one or more strings providing the names of the structure fields, formatted as a cell array.

Range

Strings in membname 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 access the structure. Using `get` with `var`, you can review the names of the fields in the structure by looking at the membname property for var.

```
var = createobj(cc,'var')
get(var,'membname')
'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. Also any valid register address, such as A0 or PC.

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]
                membname: {'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.

Characteristics

ASCII characters that compose valid C variable names.

Range

Any valid C variable name that occurs in your project.

numberofstorageunits

Description

Reports the number of 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. Combined with property `numberofstorageunits`, you can determine the storage used by the symbol.

Range

Any number of addressable unit 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, create a link object, then open two RRTDX 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.

page

Description

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. When you get the properties of an object, the address comes back in the format [address page].

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, when you create an object to access ddat, you can get the properties for the object and see the 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 page property value is 0. Since this example targets a C6711 digital signal processor, the page property value is always zero—the C6711 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 defined in the C source code. 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 declared in your C source code.

represent

Description

Contains a string that specifies the data type for the accesses symbol. Memory locations consist of bits and bytes. Property represent reports to MATLAB how to interpret the data stored in memory.

Characteristics

A string that defines the data type for the variable—one of:

- `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 String	represent Property Value
`'double'`	`'float'`
`'single'`	`'float'`
`'int32'`	`'signed'`
`'int16'`	`'signed'`
`'int8'`	`'signed'`
`'uint32'`	`'unsigned'`
`'uint16'`	`'binary'`
`'uint8'`	`'binary'`
`'long double'`	`'float'`
`'double_c'`	`'float'`
`'float'`	`'float'`
`'long'`	`'signed'`
`'int'`	`'signed'`
`'char'`	`'signed'`
`'unsigned long'`	`'signed'`
`'unsigned int'`	`'unsigned'`
`'unsigned char'`	`'binary'`
’Q0.15’	`'signed'`
’Q0.31’	`'unsigned'`

Various TI processors restrict the sizes of the datatypes used by objects in MATLAB 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

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 `ccsdsp`, `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 one 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.

storageunitspervalue

Description

Describes how many storage units (addressable units) make up the accessed symbol.

Characteristics

Given in addressable units (AU), `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 link tutorial, 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 ccsdsp]
                 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 MATLAB 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 timeout 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 seconds is the default value.

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
```

typestring

Description

Describes the data type of the referent for the pointer 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.

Link Functions Reference

Using the Link Function Reference

These sections provide complete information on each function in the links in MATLAB Link for Code Composer Studio, in a structured format. Refer to these pages when you need details about a specific function. For help on a function, enter

`help ccshelp/functionname` or `help rtdxhelp/functionname`

or use the Help desk to access the function reference page.

Contents of Function Reference Pages

Function reference pages are listed in alphabetical order by the function name. Each entry contains the following information:

- **Purpose** — describes why you use the block or function.
- **Syntax** — lists each syntax and option that applies to the function.
- **Description** — describes what the function does, by presenting all possible syntax structures for the function. Each syntax in the Syntax section appears in a description.
- **Examples** — shows the function in use and demonstrates some of the parameters and options for the function.
- **See Also** — lists related blocks and functions. This is an optional category.

Tables of Link Software Functions

For quick reference purposes, the following tables list the functions available for the links for Code Composer Studio Integrated Development Environment (CCS IDE) and Real-Time Date Exchange (RTDX). Each table entry includes the function name as a link to its reference page; whether the function is overloaded; and a brief description of the function

Table 3-1: Functions Operating on Links for CCS IDE

Function	Overloaded	Description
`activate`		Change the active file or project in CCS IDE
`add`		Add a file to the active project in CCS IDE
`animate`		Run an application on the target processor until it reaches a breakpoint
`build`		Build the active project in CCS IDE
`ccsboardinfo`		Return information about the boards and simulators recognized by CCS IDE
`ccsdsp`		Create a link to CCS IDE
`cd`		Change the working directory that CCS IDE uses
`clear`	Yes	Destroys the links to CCS IDE
`close`	Yes	Close open files in CCS IDE
`delete`		Remove debug points in files in CCS IDE
`dir`		List the files in the current CCS IDE working directory
`disp`	Yes	Display the properties of a link to CCS IDE

Table 3-1: Functions Operating on Links for CCS IDE (Continued)

Function	Overloaded	Description
`display`	Yes	Display the properties of a link to CCS IDE
`get`	Yes	Returns the property values for a link to CCS IDE.
`halt`		Terminate execution of a process running on the target processor
`info`	Yes	Return information about the target processor
`isreadable`	Yes	Determine if MATLAB can read the specified memory block
`isrtdxcapable`		Determine whether the target processor or board supports RTDX
`isrunning`		Test whether the target processor is executing a process
`isvisible`		Test whether CCS IDE is running on the PC
`iswritable`	Yes	Determine if MATLAB can write to the specified memory block
`load`		Transfer a program file (*`.out`, *`.obj`) to the target processor
`new`		Create and open a new text file, project, or build configuration in CCS IDE
`open`	Yes	Load a file into CCS IDE
`profile`		Return profile information from running a DSP/BIOS-enabled program in CCS IDE

Table 3-1: Functions Operating on Links for CCS IDE (Continued)

Function	Overloaded	Description
`read`		Retrieve data from memory on the target processor
`regread`		Return a value from a specified register on the target processor
`reload`		Resend the most recently loaded program file to the target processor
`regwrite`		Write a value to a specified register on the target processor
`remove`		Remove a file from the active CCS IDE project
`reset`		Start to reset the target processor
`restart`		Restore the program counter to the entry point for the current program on the target
`run`		Execute the program loaded on the target processor
`set`	Yes	Set the properties of links for CCS IDE.
`symbol`		Return the most recent program symbol table from CCS IDE
`visible`		Set the visibility for CCS IDE window
`write`		Write data to memory on the target processor

Table 3-2: Functions Operating on Links for RTDX

Function	Overloaded	Description
`address`		Return the address and memory page for a symbol.
`clear`	Yes	Remove existing links for RTDX and CCS IDE. Uses a destructor method to eliminate the link objects.
`close`	Yes	Close an open RTDX channel.
`configure`		Define the size and number of RTDX channel buffers.
`disable`		Disable the RTDX interface, a specified channel, or all RTDX channels.
`disp`	Yes	Display the properties of an RTDX link (default display).
`display`	Yes	Display the properties of an RTDX link.
`enable`		Enable the RTDX interface, a specified channel, or all RTDX channels.
`flush`		Flush data or messages out of one or more specified RTDX channels.
`get`	Yes	Return the property values for a link for RTDX.
`info`	Yes	Return information about specified RTDX links.
`isenabled`		Determine whether the RTDX interface or one or all RTDX channels are enabled for communications.
`isreadable`	Yes	Determine whether MATLAB can read the specified RTDX channel.

Table 3-2: Functions Operating on Links for RTDX (Continued)

Function	Overloaded	Description
`iswritable`	Yes	Determine whether MATLAB can write to the specified RTDX channel.
`msgcount`		Return the number of messages in a read-enable RTDX channel.
`open`	Yes	Open an RTDX channel to a target processor.
`readmat`		Read a matrix of data from specified RTDX channels.
`readmsg`		Read messages from the specified RTDX channel.
`set`	Yes	Set the properties of a link for RTDX.
`writemsg`		Write a message to the target processor.

Link Functions—Alphabetical List

The following reference pages list the functions included in the link software. Each function listing includes a Purpose, Syntax, Description, and Examples (when needed). Where it is appropriate, a See Also section provides references to related blocks and functions.

Functions—Alphabetical List

Purpose

Make the 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 `.pjt` extension in `objectname`.
`'text'`	Makes the specified text file in CCS IDE the active document window. Include the file extension in `objectname` 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 pathname 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')
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') % Makes myproject1 the active
project
add(cc,'c6701evm_adc.c')
activate(cc,'c6701evm_adc.c','text')
```

See Also `build`, `new`, `remove`

Purpose

Add files to the active project in Code Composer Studio

Syntax

```
add(cc,'filename')
```

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 pathname to the file, unless your file is in your CCS working directory or in a directory on your MATLAB path. The MATLAB Link for Code Composer Studio searchs 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`.

Table 3-3: 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**
Object and Library files	`.o*`, `.lib`	**Libraries**
Linker command file	`.cmd`	**Project Name**

Table 3-3: File Types and Extensions Supported by add and CCS IDE

File Type	Extensions Supported	CCS Project Folder
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.

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

  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,'c6701evm_adc.c'); % Adds the source file for the ADC block
```

In CCS IDE, `c6701evm_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')
```

See Also `activate`, `cd`, `open`, `remove`

Purpose

Append one or more registers to the list of saved registers stored in the 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` indentifies the program function to which the register applies. You can add any register to the saved registers list.

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 MATLAB Link for Code Composer Studio to operate correctly.

Default Saved Registers For C54x Processors

AR1, AR6, AR7, and SP (required by MATLAB, not the compiler)

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're needed for reading function output.

`addregister(ff,reglist)` appends the register names in `reglist` to the list of registers that get preserved when a task is finished. `ff` indentifies 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.

See Also `deleteregister`

address

Purpose

Return the address and page for an entry in the 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]`. `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
`a{1}`	String reflecting the symbol name. If `address` found a symbol that matches `symbolstring`, this is the same as `symbolstring`. Otherwise this is empty.
`a{2}(1)`	Address or value of symbol entry.
`a{2}(2)`	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.adddress(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

Purpose Run an application on the target processor until it reaches a 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 Code Composer Studio 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.

See Also `halt`, `restart`, `run`

Purpose

Assign a storage location to property `outputvar` for a structure returned by a function on C6x processors

Syntax

```
assignreturnstorage(ff,address)
assignreturnstorage(ff,handle)
```

Description

`assignreturnstorage(ff,address)` sets the `outputvar` property of the function object referred to by `ff`. `outputvar` determines where the process stores the results of executing the function. To specify the address for the output, enter `address` as a numeric value.

`assignreturnstorage(ff,handle)` assigns the return storage to the structure reference by `handle`. To use this syntax, handle must refer to a structure object.

Note `assignreturnstorage` works only with functions that return structures. For functions that return other types of data, `assignreturnstorage` does not apply. You use `assignreturnstorage` only with C6x processors.

Purpose

Build the active project in CCS IDE

Syntax

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

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 recomplied.
- 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 when the requested project build overwrites any files in the project. You must respond to the dialog before CCS IDE continues the build. The dialog 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. 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 timeout 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 timeout period passed. If you omit the timeout option in the syntax, build defaults to the global timeout defined in cc.

build(cc) is the same as build(cc,timeout) except that when you omit the timeout option, build defaults to the timeout set 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 timeout 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 timeout period passed. If you omit the timeout option in the syntax, build defaults to the global timeout 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 timeout set for cc.

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(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 the datatype of an object in MATLAB Link for Code Composer Studio

Syntax

```
objname2 = cast(objname,datatype)
objname2 = cast(objname,datatype,size)
```

Description

`objname2 = cast(objname,datatype)` returns `objname2`, a copy of `objname` whose `represent` property is changed to the data type specified by `datatype`. Input argument `datatype` can be any supported datatype. After the cast operation, `read` or `write` operations apply the appropriate data conversion to implement on the target the datatype specified by the `represent` property.

The following datatypes work as input arguments to `cast`:

Datatype String	represent Property Value
'double'	'float'
'single'	'float'
'int32'	'signed'
'int16'	'signed'
'int8'	'signed'
'uint32'	'unsigned'
'uint16'	'binary'
'uint8'	'binary'
'long double'	'float'
'double_c'	'float'
'float'	'float'
'long'	'signed'
'int'	'signed'
'char'	'signed'
'unsigned long'	'signed'

Datatype String	represent Property Value
'unsigned int'	'unsigned'
'unsigned char'	'binary'
’Q0.15’	'signed'
’Q0.31’	'unsigned'

Various TI processors restrict the sizes of the datatypes used by objects in MATLAB 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

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` property. `function`, `ccsdsp`, and `rtdx` objects do not use the `represent` property and do not support `cast`.

`objname2 = cast(objname,datatype,size)` returns `objname2`, a copy of `objname`, with the specified datatype for the `represent` property, and the `size` property value set to `size`.

See Also `convert`

Purpose

Return information about all 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.
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.

Installed Board Configuration Data	Configuration Item Name	Description
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 = ccsdsp('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 Board                            Proc Processor                        Processor
 Num  Name                              Num  Name                                 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 Board                            Proc Processor                        Processor
 Num  Name                              Num  Name                                 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.

Type

```
ccsboardinfo
```

at the MATLAB prompt. You get

```
Board Board                             Proc Processor                             Processor
 Num  Name                               Num  Name                                    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 type

```
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 defined (the `3x1` array).

using the dot notation for accessing the contents of a structure has use 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).
```

See Also

`info`, `ccsdsp`

Purpose Create a link to Code Composer Studio 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` launches 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 the MATLAB 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 object	`'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.

Object	Property Name	Property	Default	Description
	`'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 timeout	10.0s	Specifies how long MATLAB waits for a response from CCS after issuing a request.
RTDX Object	`'timeout'`	Timeout	10.0s	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.

`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.

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 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.

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.

See Also

`get`, `ccsboardinfo`, `set`

Purpose

Change the CCS IDE working directory

Syntax

```
cd(cc,'directory')
wd = cd(cc,'directory')
cd(cc,pwd)
```

Description

`cd(cc,'directory')` changes the CCS IDE working directory to the directory identified by the string `dir`. `dir` must refer to an existing directory for the change to take affect. 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(cc)` 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, 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 GEL (General Extension Language) expressions on the 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 refered 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. 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 (the General Extension Language) , 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 timeout error means the confirmation was delayed but the 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 timeout in `cc`. Use `get(cc)` to determine the global timeout 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 documention.

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
`result = cexpr(cc,'x.a')`	Returns the value of field `a` in structure `x` stored on your target. For this example, `expression` is `x.a` and `result` contains the value stored in `x.a` on the target.
`result = cexpr(cc,'StartUp()')`	Executes the GEL function `StartUp` on the target processor. `expression` is ’`StartUp`’, a function in the GEL file that loads each time you start CCS. Note that GEL function names are case sensitive — StartUp is not the same as startup. In this example, `result` is NULL or empty because GEL functions do not generate return values. Do not use an output argument with GEL expressions as input arguments.
`result = cexpr(cc,'x.b = 10')`	Sets and returns the value of the field `b` in structure `x`. Here the assignemt statement in single quotation marks replaces `expression`. `x.b` must be a structure in memory on your target and in the current program scope. After execution, `result` contains the value `10` returned from the target.
`result = cexpr(cc,['x.c[2] =' int2str(z)])`	Sets the value of `x.c[2]` to the string represented by integer `z`. In MATLAB, `result` contains the value stored in `x.c[2]` as returned from the target. Notice that the C expression is in single quotation marks, and the MATLAB `int2str` is not. Using single quotation marks directs MATLAB to ignore the C string that applies to the target processor and to evaluate `int2str`.

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 Remove links to CCS IDE and RTDX interface

Syntax

```
clear(cc)
clear('all')
```

Description `clear(cc)` clears the link associated with `cc`. The last step in any development that uses links. Clear links 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. The last step in any development that uses links. Clear links 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.

See Also `ccsdsp`, `close`, `disable`

Purpose Close files in CCS IDE or an open 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
`'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 close `.cfg` 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(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
`close(cc,'all','project')`	Close all open projects in CCS IDE.
`close(cc,'my.pjt','project')`	Close the project `my.pjt`.
`close(cc,[],project')`	Close the active project.
`close(cc,'all','text')`	Close all open text files. This includes source file, libraries, command files, and others.
`close(cc,'my_source.cpp','text')`	Close the text file `my_source.cpp`.
`close(cc,[],'text')`	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 = ccsdsp('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`

Purpose

Define the 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`. `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 that 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 Code Composer Studio IDE to verify the buffers.

See Also `readmat`, `readmsg`, `write`, `writemsg`

Purpose

Change the `represent` property for an object from one datatype 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 datatype. After you change the datatype specified in `represent`, `read` or `write` operations apply the appropriate data conversion to implement on the target the datatype specified by the `represent` property.

The following datatypes work as input arguments to `convert`:

Datatype String	represent Property Value
`'double'`	`'float'`
`'single'`	`'float'`
`'int32'`	`'signed'`
`'int16'`	`'signed'`
`'int8'`	`'signed'`
`'uint32'`	`'unsigned'`
`'uint16'`	`'binary'`
`'uint8'`	`'binary'`
`'long double'`	`'float'`
`'double_c'`	`'float'`
`'float'`	`'float'`
`'long'`	`'signed'`
`'int'`	`'signed'`
`'char'`	`'signed'`
`'unsigned long'`	`'signed'`

Datatype String	represent Property Value
'unsigned int'	'unsigned'
'unsigned char'	'binary'
’Q0.15’	'signed'
’Q0.31’	'unsigned'

Various TI processors restrict the sizes of the datatypes used by objects in MATLAB 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

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.

`convert` applies to any object that has the `represent` property. `function`, `ccsdsp`, and `rtdx` objects do not use the `represent` property and do not support `convert`.

`convert(objname,datatype,size)` returns `objname` with the specified datatype for the `represent` property, and the `size` property value set to `size`.

See Also `cast`

Purpose Make a copy of an object

Syntax

```
objname2 = copy(objname)
```

Description `objname2 = copy(objname)` returns `objname2` that is a copy of the input object specified by `objname`. All objects in the MATLAB 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 that represent embedded data or functions in a program on your target

Syntax

```
objname = createobj(cc,'symbolname');
objname = createobj(cc,'symbolname','option');
```

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 work, you must have loaded a `.out` file to your target in CCS, and the symbol must be in the current symbol table in CCS.

To increase the accuracy of the information about global symbols in your project, use `goto`, as shown here, to position the program counter to the start of main in your application in CCS.

```
goto(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 whether `symbolname` represents a static or global variable. Use either of the following strings to declare the type for symbolname in option:

- `static`—declares that `symbolname` refers to a static variable in your code
- `global`—declares that `symbolname` refers to a global variable in your code

See Also

`copy`, `ccsdsp`

Purpose Remove debug points in addresses or source files in Code Composer Studio

Syntax

```
delete(cc,addr,'type')
delete(cc,addr)
delete(cc,filename,line,'type')
delete(cc,filename,line)
```

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. 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.

Unlike 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, the MATLAB Link for Code Composer Studio returns an error reporting that it could not find the specified debugging point.

`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. *`type`* accepts one of two strings — `break` or `probe` — as defined previously. When '*`type`*' does not match the debug point type at the specified location, or no debug point exists, the MATLAB Link for Code Composer Studio returns an error that it could not find the debug point.

`delete(cc,filename,line)` defaults to '`break`' to remove a breakpoint.

See Also `address`, `insert`, `run`

Purpose

Remove one or more registers from the list of saved registers stored in the property `savedregs` of function objects

Syntax

```
deleteregister(ff,'regname')
deleteregister(ff,'reglist'
```

Description

`addregister(ff,regname)` removes register `regname` from the list of registers that get preserved or reverted when a function is finished running. `ff` indentifies 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.

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 MATLAB Link for Code Composer Studio to operate correctly.

Default Saved Registers For C54x Processors

AR1, AR6, AR7, and SP (required by MATLAB, not the compiler)

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're needed for reading function output

`deleteregister(ff,reglist)` deletes the register names in `reglist` from the list of registers that get preserved when a task is finished. `ff` indentifies 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 `addregister`

Purpose

Return an object that accesses the object a pointer object points to

Syntax

```
objname2 = deref(objname)
```

Description

`objname2 = deref(objname)` creates `objname2`, an object representing 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`.

After being returned by `deref`, `objname2` is an object which represents the target of `objname`. When you write objname2 to CCS,

See Also

`createobj`, `read`, `write`

dir

Purpose List the files in the 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`

Purpose

Disable the RTDX interface, a 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`. `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 the channel properties of an RTDX link

Syntax

```
disp(rx)
```

Description `disp(rx)` provides a formatted list of the channel property names and property values for the specified RTDX link to your target processor. When you create a new link in MATLAB and omit the closing semicolon on the `ccsdsp` function, MATLAB uses `disp` to display the configuration for the new link.

Examples The following example illustrates the display for the channel properties. Notice that `disp` does not return the name of the link (`cc`) in the display.

```
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

disp(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

rx = cc.rtdx
disp(rx)
```

```
        RTDX channels    : 0
```

See Also `display`, `get`, `set`

display

Purpose

Display the properties of a link to CCS IDE or an RTDX link

Syntax

```
display(cc)
display(rx)
```

Description

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 a link to CCS IDE. To return the configuration data, `display` calls the function `disp`.

```
display(cc)
```

```
display(rx)
```

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
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0
```

Examples

Try this example to see the display for an RTDX link to a target processor.

```
cc = ccsdsp;
rx=(cc.rtdx)    % Assign the RTDX portion of cc to rx.

RTDX channels    : 0


display(rx)
```

```
     RTDX channels    : 0
```

See Also `disp`, `get`, `set`

enable

Purpose

Enable the RTDX interface, a specified channel, or all RTDX channels

Syntax

```
enable(rx,'channel')
enable(rx,'all')
enable(rx)
```

Description

`enable(rx,'channel')` enables the open channel specified by the string `channel`, for RTDX link `rx`. `rx` represents the RTDX portion of the associated link to CCS IDE.

`enable(rx,'**all**')` enables all the open channels associated with `rx`.

`enable(rx)` enables the RTDX interface for `rx`.

Important requirements for using enable

On the target side, `enable` depends on RTDX to enable channels. Therefore the you must meet the following requirements to use `enable`.

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 `enable` to work.
5. Using code in the program running on the target to enable channels overrides the default disabled state of the channels.

Examples

To use channels to RTDX, you must both open and enable the channels.

```
cc = ccsdsp; % 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.
enable(cc.rtdx,'inputchannel') % Enable the channel so you can use
                               % it.
```

See Also

`disable`, `open`

Purpose

Return the equivalent string or numeric value for an 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. 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 a function on a target through Code Composer Studio

Syntax

```
execute(ff)
execute(ff,input1,value1,...,inputn,valuen)
```

Description `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 in the assigned location in property `outputvar` of `ff`. From MATLAB, use `read` to check the result stored in `outputvar`.

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.

You must set the property `outputvar` for `ff` before you run your function. `execute` fails if you have not set `outputvar` prior to executing `ff`.

`execute(ff,input1, value1,...,inputn,valuen)` runs the function identified by `ff`, first writing the input values assigned by the `input1`, `value1`, `input2`, `value2`, and so on pairs to `inputvars`. `input1`, `input2`,...,`inputn` must be strings

See Also `goto`, `run`, `write`

Purpose Flush data or messages out of one or more specified RTDX channels

Syntax

```
flush(rx,'channel',num,timeout)
flush(rx,'channel',num)
flush(rx,'channel',[],timeout)
flush(rx,'channel')
```

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 timeout period stored in `rx`. `flush` applies the timeout 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 timeout period stored in rx to determine how long to wait for the process to complete. Compare this to the previous syntax that specifies the timeout 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 timeout period stored in `rx`. `flush` applies the timeout 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.

If you use `flush` with a write-configured RTDX channel, DKTI 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 = ccsdsp;
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

Return the properties of an object

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(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

Return an object that accesses one member of a structure

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
  Array ordering          : row-major
  Members                 : 'x_int', 'x_str', 'z_str'

x_str=getmember(structtest,'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 indeed 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_int is now your handle to write to or read from member s_double.

```
read(s_int)

ans =

 -1.4938e+059

write(s_int,2)
read(s_int)

ans =
2
```

See Also `createobj`, `read`, `write`

Purpose

Position the program counter to the specified location in the project code

Syntax

```
goto(cc)
goto(cc,'functionname')
goto(ff)
goto(objname,'input1',value1,...,'inputn',valuen)
```

Description

`goto(cc)` places a breakpoint at the entry point to the main function, then restarts the target and waits for the running application to stop at the breakpoint. When successful, `goto` positions the target program counter (PC) to the beginning of main. With the PC positioned to main, the target can initialize the static variables.

Note `goto` always halts at the first breakpoint it encounters, and returns that location.

`goto(cc,'functionname')` positions a breakpoint at the entry point for functionname, then runs the program on the target until it reaches the breakpoint, in the project specified by `cc`. Because CCS automatically adds a breakpoint at main, position the PC at main before you use this syntax. If you have other breakpoints between the start of main and the starting point for `functionname`, `goto` may return with the location of a breakpoint that is not the beginning of functionname. To avoid this, disable all breakpoints before you run `goto`. Specify `functionname` as a string. Note that goto returns when it encounters

`goto(ff)` postions the PC 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 must pass the necessary values for the function input arguments into the appropriate registers. You must do this whether the function has input parameters or not.

In the following list, you see the registers and memory locations that are affected by preparing to run the function. Since you can only use up to 10 input arguments for a function, only 10 arguments appear in the list.

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
`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
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	
Frame Pointer (FP)	A15	N/A	
Stack Pointer (SP)	B15	N/A	

`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 command. 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.

See Also `delete`, `execute`, `insert`, `run`

Purpose

Terminate execution of a process running on the target

Syntax

```
halt(cc,timeout)
```

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.

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

Purpose

Return information about the 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
`info.procname`	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with `cc`.
`info.isbigendian`	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`.
`info.family`	Integer	Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors.
`info.subfamily`	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 `dec2hex` to convert the value in `info.subfamily` to standard notation. For example `dec2hex(info.subfamily)` produces '67' when the processor is a member of the 67xx processor family.
`info.timeout`	Integer	Default timeout value MATLAB uses when transferring data to and from CCS. All functions that use a timeout value have an optional `timeout` input argument. When you omit the optional argument, MATLAB uses this default value—10s.

`info = info(rx)` returns info as a cell arraying containing the names of your open RTDX channels.

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
```

In this example, we are simulating 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 a debug point to a source file or address in Code Composer Studio

```
insert(cc,addr,'type')
insert(cc,addr)
insert(cc,filename,line,'type')
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. 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.

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)` 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. `type` accepts one of three strings — `break`, `probe`, or `profile` — as defined previously. When the line or file you specified does not exist, the MATLAB Link for Code Composer Studio returns an error explaining that it could not insert the debug point.

`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 MATLAB 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 cirles 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.

See Also

`address`, `delete`, `run`

isenabled

Purpose Determine whether an 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 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`

isreadable

Purpose

Determine if MATLAB can read the 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. 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`. `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. In cases where 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 to set the page parameter.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using just the memory location without the page value.

Table 3-4: 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 specified `datatype`. `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

datatype String	Number of Bytes/Value	Description
'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.

Like the `iswritable`, `read`, and `write` 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.

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 on 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');s
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 the 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`.

Examples Create a link to your C6701EVM. Test to see if the processor on the board supports RTDX. It should.

```
cc=ccsdsp; %Assumes you have one board and it is the C6701 EVM
b=isrtdxcapable(cc)
b =

   1
```

Purpose Test whether the target processor is executing a 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`.

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)

ans =

     1

halt(cc)
isrunning(cc)

ans =

     0
```

See Also `halt`, `restart`, `isrunning`

Purpose Test whether CCS IDE is running on the PC

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.

```
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. The only ways to close it are either

- Launch 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`

iswritable

Purpose Determine if MATLAB can write to the specified memory block

Syntax

```
iswritable(cc,address,'datatype',count)
iswritable(cc,address,'datatype')
```

Description

`iswritable(cc,address,'datatype',count)` returns 1 if MATLAB can write to the memory block defined by the `address`, `count`, and `datatype` input arguments on the processor referred to by `cc`. When the processor cannot write to 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 being tested begins at the memory location defined by `address`. `count` determines the number of values to write. `datatype` defines the format of data stored in the memory block. `iswritable` uses the `datatype` parameter to determine the number of bytes to write per stored value. For details about each input parameter, read the following descriptions.

`address` — `iswritable` 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. In cases where 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 to set the page parameter.

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.

Table 3-5: Examples of Address Property Values

Property Value	Address Type	Interpretation
'1F'	String	Location is 31 decimal on the page referred to by `cc(page)`

Table 3-5: Examples of Address Property Values (Continued)

Property Value	Address Type	Interpretation
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 2-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
`'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

`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.

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 ’`ichannel`’, it proceeds to write messages as required.

See Also

`hex2dec`, `iswritable`, `read`

Purpose Return various information listings from Code Composer Studio

Syntax

```
list(ff,varname)
infolist = list(cc,type,option)
infolist = list(cc,type,option)
```

Description `list(ff,varname)` lists the local variables associated with the function accessed by function object `ff`.

`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 variable
- **`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 datatype typedef is excluded from the list of datatypes.

Note, the `list` function returns dynamic Code Composer information that can be altered by the user. Returned information listings represent snapshots of the current Code Composer studio configuration only. Be aware that earlier copies of `infolist` might contain stale information.

`infolist = list(cc,'`**`project`**`')` returns a vector of structures containing project information.

infolist Structure Element	Description
`infolist(1).name`	Project file name (with path).
`infolist(1).type`	Project type—`'project'`,`'projlib'`, or `'projext'`, see `new`
`infolist(1).targettype`	String Description of Target CPU
`infolist(1).srcfiles`	Vector of structures that describes project source files. Each structure contains the name and path for each source file—`infolist(1).srcfiles.name`
`infolist(1).buildcfg`	Vector of structures that describe build configurations, each with the following entries: • `infolist(1).buildcfg.name`—the build configuration name • `infolist(1).buildcfg.outpath`—the default directory for storing the build output.
`infolist(2).…`	…
`infolist(n).…`	…

`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 returned information in each structure follows the format:

infolist Structure Element	Description
infolist.varname(1).name	Symbol name
infolist.varname(1).isglobal	Indicates whether symbol is global or local
infolist.varname(1).location	Information about the location of the symbol
infolist.varname(1).size	Size per dimension
infolist.varname(1).uclass	ccsdsp object class that matches the type of this symbol
infolist.varname(1).bitsize	Size in bits. More information is added to the structure depending on the symbol type.
infolist.(varname1).type	Datatype of symbol
infolist.varname(2).…	...
infolist.varname(n).…	...

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:

infolist Structure Element	Description
`infolist.functionname(1).name`	Function name
`infolist.functionname(1).filename`	Name of file where function is defined
`infolist.functionname(1).address`	Relevant address information such as start address and end address
`infolist.functionname(1).funcvar`	Variables local to the function
`infolist.functionname(1).uclass`	ccsdsp object class that matches the type of this symbol - 'function'
`infolist.functionname(1).funcdecl`	Function declaration; where information such the function return type is contained
`infolist.functionname(1).islibfunc`	Is this a library function?
`infolist.functionname(1).linepos`	Start and end line positions of function
`infolist.functionname(1).funcinfo`	Miscellaneous information about the function
`infolist.functionname(2)....`	...
`infolist.functionname(n)....`	...

To refer to the function structure information, `list` uses the function name as the fieldname.

`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' datatypes 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 (tm) compiler as '$faken' where n is an assigned number.

`infolist = list(cc,'`**`type`**`',typename)` returns a structure that contains information on the specified defined datatype.

`infolist = list(cc,'`**`type`**`',typenamelist)` returns a structure that contains information on the specified defined datatypes in the list. The returned information follows the format:

infolist Structure Element	Description
`infolist.typename(1).type`	Type name
`infolist.typename(1).size`	Size of this type
`infolist.typename(1).uclass`	ccsdsp object class that matches the type of this symbol. Additional information is added depending on the type
`infolist.typename(2).…`	...
`infolist.typename(n).…`	...

For the fieldname, `list` uses the type name to refer to the type structure information.

Important—when a variable name, type name, or function name is not a valid MATLAB structure fieldname, `list` replaces or modifies the name so it becomes valid.

Note In fieldnames that contain the invalid dollar character '`$`', `list` replaces the '`$`' with '`DOLLAR`.'

Note Changing the MATLAB fieldname 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 fieldname '`_with_underscore`' gets changed to '`Q_with_underscore`.' To make the invalid name valid, the character '`Q`' is inserted before the name.

```
varname1 = '_with_underscore'; % invalid fieldname
list(cc,'variable',varname1);
ans =

    Q_with_underscore : [varinfo]
ans. Q_with_underscore
ans=

       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 fieldnames 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

Purpose

Transfer a program file (*.out, *.obj) to the target processor

Syntax

```
load(cc,'filename',timeout)
load(cc,'filename')
```

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 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` defaults to the global value you set when you created link `cc`.

Note `load` disables all open channels. Open channels revert to `disabled`.

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 typing

```
load(cc,'filename.out')
```

See Also `cd`, `dir`, `open`

Purpose

Return the number of messages in a 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 a new 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 four strings or entries listed in the following table.

type string	Description
`’text’`	Create a new text file in the active project.
`’project’`	Create a new project.
`’projext’`	Create a new CCS external make project. Using this option idicates that your project uses and external makefile. Refer to your CCS documentation for more information about external projects.

type string	Description
’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 following 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`

Purpose

Open a channel to a target processor or load a 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
'**`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.

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) get 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 = ccsdsp;
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 “Tutorial 2-1—Using Links and Embedded Objects”, use the following syntax

```
cc = ccsdsp;
cc.load(tutorial_6xevm.out);
```

See Also

`cd`, `dir`, `load`

Purpose

Return code profiling information from executing code with or without 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**
'raw'	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, `profile(cc,'report')` 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 ps	Description
`ps.cpuload`	Execution time in percent of total time spent out of the idle task.
`ps.sts`	Vector of defined STS objects in the project.
`ps.sts(n).name`	User-defined name for an STS object `sts(n)`. Value for `n` ranges from 1 to the number of defined STS objects.
`ps.sts(n).units`	Either '`Hi Time`' or '`Low Time`.' Describes the timer applied by this STS object, whether high- or low-resolution time based.
`ps,sts(n).max`	Maximum measured profile period for `sts(n)`, in seconds.

Fields in ps	Description
`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 Paramters** dialog.

`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

Since 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 pathname to the project file.

```
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 timeout 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 out any of the returned data.

```
ps=profile(cc,'raw')

ps =

      cpuload: 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

Purpose

Retrieve data from memory on the target processor or in CCS

Syntax

```
mem = read(cc,address,'datatype',count,timeout)
mem = read(cc,address,'datatype',count)
mem = read(cc,address,'datatype')
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 be read begins at the memory location defined by `address`. `count` determines the number of values to be read, starting at `address`. `datatype` defines the format of the raw data stored in the referenced memory block.

`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. In cases where 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 to set the `page` parameter.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using just the memory location without the page value.

Table 3-6: Examples of Address Property Values

Property Value	Address Type	Interpretation
'1F'	String	Offset is 31 decimal on the page referred to by `cc(page)`
10	Decimal	Offset is 10 decimal on the page referred to by `cc(page)`
[18,1]	Vector	Offset is 18 decimal on memory page 1 (`cc(page) = 1`)

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 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`.

Input	**Response**
`[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. timeout 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`.

`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 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.

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
`size`	Greater than 1	Specifies the dimensions of the output numeric array.
`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. • `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 `storageunitspervalue`
`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 to do individual read operations, consider using `getmember`.

Examples

In its most straightforward form, `read` reads data that you wrote to the target processor.

```
cc = ccsdsp;
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 = ccsdsp;
  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 date type from memory, the returned entry is a string object.

Reading Structures

See Also

`getmember`, `isreadable`, `symbol`, `write`

Purpose

Read a matrix of values from an 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 try to 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

datatype String	Data Format
`'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 try to 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.

datatype String	Data Format
'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, you can replace the listed channels with your current ones.

```
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 data into a 5-by-5 array called `out_array`.

```
out_array = readmat('ochannel','double',[5 5])
```

See Also

`readmsg`, `writemsg`

Purpose

Read messages from the 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`. Thus, 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 six strings that define the type of data you are expecting.

datatype String	Specified Data Type
`'double'`	Floating point data, 64-bits (double-precision).
`'int16'`	Signed 16-bit integer data.
`'int32'`	Signed 32-bit integers.
`'single'`	Floating point data, 32-bits (single- precision).
`'uint8'`	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`. Thus, 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. `readmsgs` 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(l)] where l is the number of data elements in message. In this syntax, the read messages can have different lengths, unlike the previous syntax options.

`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 matrixes, causing them to use their default settings — `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 a string object and convert to the numeric equivalent in MATLAB

Syntax

```
data = readnumeric(objname)
data = readnumeric(objname,index)
data = readnumeric(…,timeout)
```

Description

```
data = readnumeric(objname)
```

```
data = readnumeric(objname,index)
```

```
data = readnumeric(…,timeout)
```

See Also `getmember`, `read`, `write`

Purpose

Return a value from a specified 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. Link `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`, `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.

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 `represent` 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.

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`

regwrite

Purpose Write data values to specified registers on a 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 `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.

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` option 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. They stop 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 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 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.

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 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 `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.

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.

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`

reload

Purpose Reload to the target signal processor the most recently loaded program file

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 pathname 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.

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.

```
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 a file from the active CCS IDE project

Syntax `remove(cc,'filename')`

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.

Examples After you have a project in CCS IDE, you can delete files from it using `remove` from the MATLAB command line.

See Also `activate`, `add`, `cd`, `open`

Purpose

Initiate a reset of the 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 setting 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.

See Also

`halt`, `restart`, `run`

reshape

Purpose Change the shape of an array maintaining the 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 columnwise 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))` bust be evenly divisible by the product of the known dimensions (all the dimensions exclusive of the unknown dimension). Within the function call, you are allowed to use only one set of square brackets, `[]`, for one unknown dimension.

Purpose

Restore the program counter to the entry point for the 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 `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 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. 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.

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. Also, in the case where your process gets lost or halts, restart is a quick way to restore your program.

See Also

`halt`, `isrunning`, `run`

Purpose

Execute the program loaded on the target processor

Syntax

```
run(cc,state,timeout)
```

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 three strings that set the state of the processor:

State String	Run Action
`'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, `run` 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.
`'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 timeout error.

Examples

After you build and load a program to your target, use run to start execution.

```
cc = ccsdsp('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 DKTI. 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 pathname 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')
save(cc,'filename')
```

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 six commands that save open files or projects in CCS IDE.

Command	Result
`save(cc,'all','project')`	Save all open projects in CCS IDE.
`save(cc,'my.pjt','project')`	Save the project `my.pjt`.
`save(cc,[],project')`	Save the active project.
`save(cc,'all','text')`	Save all open text files. This includes source file, libraries, command files, and others.

Command	Result
`save(cc,'my_source.cpp','text')`	Save the text file `my_source.cpp`.
`save(cc,[],'text')`	Save the active file window.

See Also `add`, `cd`, `close`, `open`

Purpose

Set the properties of links for CCS IDE and RTDX interface

Syntax

```
set(cc,'propertyname','propertyvalue')
set(cc,'propname1','propvalue1','propname2','propvalue2')
v = set(cc)
cc.propertyname = 'propertyvalue'
set(rx,'propertyname','propertyvalue')
set(rx,'propname1','propvalue1','propname2','propvalue2')
v = set(rx)
rx.propertyname = 'propertyvalue'
```

Description

`set(cc,'propertyname','propertyvalue')` sets the specified property of `cc` to the specified value.

`set(cc,'propname1','propvalue1','propname2','propvalue2')` sets multiple properties (`propname1`, `propname2`) 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,'propname1','propvalue1','propname2','propvalue2')` sets multiple properties (`propname1`, `propname2`) 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

Purpose

Return the most recent 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, 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
`s(1).name`	String reflecting the symbol entry name.
`s(1).address(1)`	Address or value of symbol entry.
`s(1).address(2)`	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`.

It 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 theMATLAB Link for Code Composer Studio demo program c6701evmafxr. Start by entering `c6701evmafxr` at the MATLAB prompt.

```
c6701evmafxr;
```

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 = ccsdsp;
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`

Purpose Set whether CCS IDE window is visible while CCS is running

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 being left open with no way to close it without using Windows Task Manager. If CCS IDE is not visible when you open MATLAB, closing MATLAB leaves CCS IDE running in as invisible state. More directly, 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, type

```
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.

```
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. The only ways to close it are either

- Launch 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 the target processor

Syntax

```
write(cc,address,data,timeout)
write(cc,address,data)
write(objname)
write(objname,index)
write(objname,stindex,member1,value1,…,membern,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 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. `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.

`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. In cases where 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 to set the page parameter.

For processors that have one memory page, setting the page value to 0 lets you specify all memory locations in the processor using just the memory location without the page value.

Table 3-7: Examples of Address Property Values

Property Value	Address Type	Interpretation
`'1F'`	String	Offset is 31 decimal on the page referred to by `cc(page)`
`10`	Decimal	Offset is 10 decimal on the page referred to by `cc(page)`
`[18,1]`	Vector	Offset is 18 decimal on memory page 1 (`cc(page) = 1`)

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 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)`.

`data` — the scalar, vector, or array of values that are written to memory on the processor. `write` 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

`datatype` String (Continued)	Description
'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 out 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.

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 differ for different types of objects, such as structure objects or register objects.

Property	Options	Description
`size`	Greater than 1	Specifies the dimensions of the output numeric array.
`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. • `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 `storageunitspervalue`
`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`. `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`. 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, `write` cannot determine which elements to return.

`write(objname,stindex,member1,value1,…,membern,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. `membern`, `valuen`, `memindex`, and `stindex` (an optional input argument) specify which structure member to read:

- `membern`—specifies the name of the member of the structure to write
- `valuen`—specifies the value to write to `membern`
- `memberindex`—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, being able to explicitly set the `timeout` value as an input option may be necessary to let `write` run to completion. Note that using the `timeout` input option 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 date 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 eighth 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 idiosyncracies. 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” for samples of `write` in use.

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 = ccsdsp;
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".

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.

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. Alll 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`

writemsg

Purpose

Write messages to the specified RTDX channel

Syntax

```
data = writemsg(rx,channelname,data,timeout)
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 datatypes: uint8, int16, int32, single, and double.

`data = writemsg(rx,channelname,data)` ises 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 column-wise to `ichan`. The preceding function syntax is equivalent to:

```
writemsg(cc.rtdx, 'ichan', [1:9]);
```

See Also

`readmat`, `readmsg`, `write`

Index