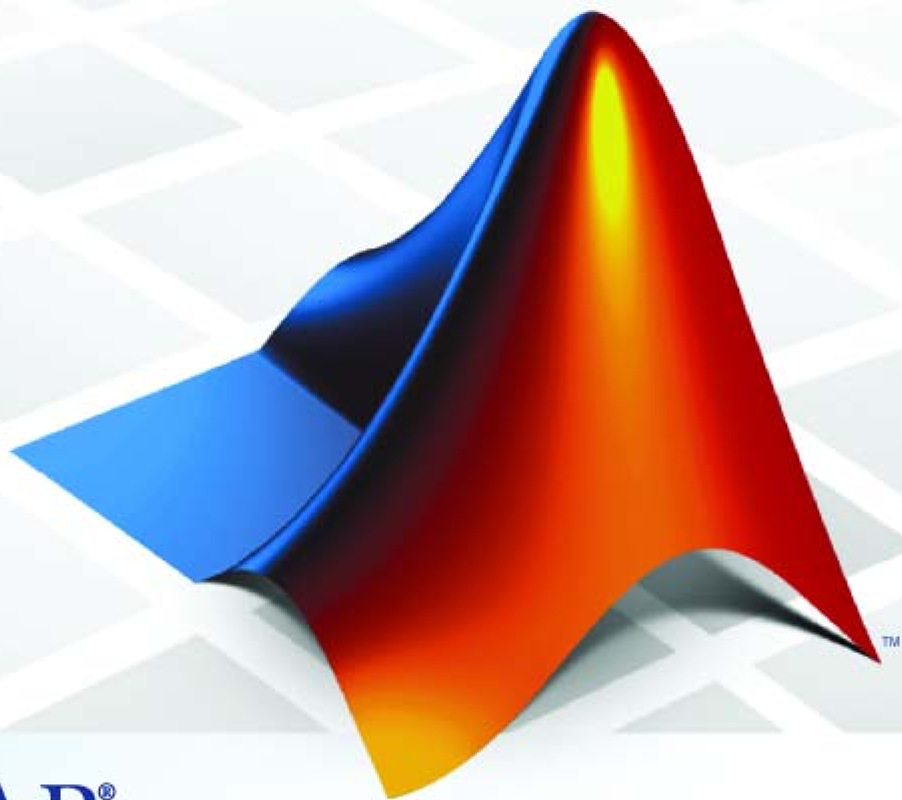
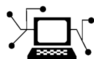
Target Support Package™ 4
User's Guide



MATLAB® & SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks



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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Target Support Package™ User's Guide

© COPYRIGHT 2009–2010 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2009	Online only	New for Version 4.0 (Release 2009b)
March 2010	Online only	Revised for Version 4.1 (Release 2010a)

Contents

1 Getting Started

Product Overview 1-2
Product Description 1-2
Key Features 1-2

Using this Guide 1-4
Expected Background 1-5

2 Preparing Models for Embedded Deployment

Setting Target Preferences 2-2
What are Target Preferences Blocks? 2-2
Locating a Target Preferences Block 2-3
Configuring a Target Preferences Block for a Supported Processor 2-3
Adding a Target Preferences Block to Your Model 2-4
Examples of Configuring Target Preferences 2-5

Setting Configuration Parameters for Embedded IDE Link 2-6
What are Configuration Parameters? 2-6
Setting Model Configuration Parameters 2-6

Working with Block Libraries 2-15

Simulink Models and Targeting 2-16
Creating Your Simulink Model for Targeting 2-16
Blocks to Avoid in Your Models 2-17

3 Supported Operating Systems

Overview **3-2**

Preparing Models to Run on Windows or Linux **3-3**

Selecting the Operating System and Scheduling Mode **3-5**

Linux-Specific Topics **3-6**
Scheduling **3-6**
Running Multirate Multitasking Executables on the Linux Development System **3-6**
Avoiding Lock-Up in Free-Running Multi-Rate Multi-Tasking Models **3-8**

Embedded Linux-Specific Topics **3-9**
Troubleshooting “sched_setaffinity: Bad address” Error . . **3-9**

Windows-Specific Topics **3-10**
Scheduling **3-10**

4 Block Reference

Host CAN Blocks (canmsglib) **4-2**

Host Communication (hostcommlib) **4-3**

5 Blocks — Alphabetical List

Index

Getting Started

- "Product Overview" on page 1-2
- "Using this Guide" on page 1-4

Product Overview

In this section...
"Product Description" on page 1-2
"Key Features" on page 1-2

Product Description

Target Support Package™ lets you deploy code generated from MathWorks products for real-time execution on embedded microprocessors, microcontrollers, and DSPs. Using Target Support Package, you can integrate peripheral devices and real-time operating systems with the algorithms created using Simulink® models, Stateflow® charts, and the Embedded MATLAB™ language subset without writing low level drivers and runtime code. The resulting executable can be deployed onto embedded hardware for on-target rapid prototyping, real-time performance analysis, and field production.

Key Features

- Provides fully integrated turnkey solution to build complete executable from generated code that includes device drivers and scheduler for real-time execution on embedded processors
- Includes target-specific code and blocks for analog I/O, digital I/O, pulse width modulation, waveform measurement, Serial communication, CAN, and more
- Provides code and blocks for multi-tasking scheduling using synchronous and asynchronous tasks, task preemption, and temporary task overruns
- Enables interactive parameter tuning and monitoring of real-time applications using Simulink external mode
- Provides specialized block libraries for Ethernet host and target communication
- Provides optimized assembly code and blocks for Texas Instruments signal processing, IQMath, and Digital Motor Control libraries

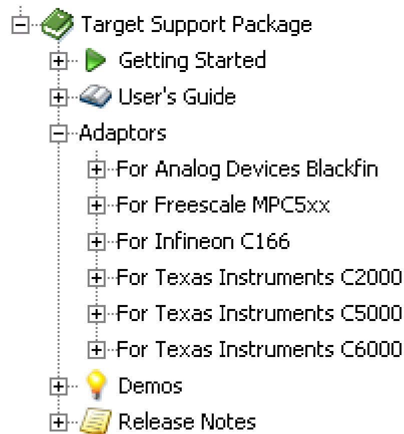
- Supports processor families including Analog Devices Blackfin®, Freescale™ MPC5xx, Infineon® C166®, Texas Instruments C2000™/C5000™/C6000™

Using this Guide

The Target Support Package™ product supports with a number of target processor families and target operating systems. The structure of this documentation provides general product information, and information that applies to specific targets.

Note It is important for you to understand the documentation structure so you can find both the general and the specific information you need.

In the online help for the Target Support Package product, the **Getting Started** and **User Guide** sections provide general information. Target-specific information is located in the **Supported Processors** sections.



Four of the six sections under **Supported Processors** contain documentation for previous stand-alone products. They are:

- **For Freescale MPC5xx** used to be the documentation for Target Support Package FM5.
- **For Infineon C166** used to be the documentation for Target Support Package IC1

- **For Texas Instruments C2000** used to be the documentation for Target Support Package TC2
- **For Texas Instruments C6000** used to be the documentation for Target Support Package TC6

We are in the process of refactoring these four sections.

Expected Background

To get the most out of this manual, you should be familiar with MATLAB® software and its associated programs, such as Signal Processing Blockset™ software and Simulink® software. We do not discuss details of digital signal processor operations and applications, except to introduce concepts related to using specific targets. 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.
- Oppenheim, A.V., R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.
- Mitra, S. K., *Digital Signal Processing — A Computer-Based Approach*, The McGraw-Hill Companies, Inc, 1998.
- Steiglitz, K, *A Digital Signal Processing Primer*, Addison-Wesley Publishing Company, 1996.

Refer to the third-party documentation for your hardware and IDEs for information about setting them up and using them.

2

Preparing Models for Embedded Deployment

- “Setting Target Preferences” on page 2-2
- “Setting Configuration Parameters for Embedded IDE Link” on page 2-6
- “Working with Block Libraries” on page 2-15
- “Simulink Models and Targeting” on page 2-16

Setting Target Preferences

In this section...

"What are Target Preferences Blocks?" on page 2-2

"Locating a Target Preferences Block" on page 2-3

"Configuring a Target Preferences Block for a Supported Processor" on page 2-3

"Adding a Target Preferences Block to Your Model" on page 2-4

"Examples of Configuring Target Preferences" on page 2-5

The following IDE's and processor families use target preferences blocks. The information in this section applies to them:

- Texas Instruments Code Composer Studio™, C2000™, C5000™, and C6000™
- Analog Devices VisualDSP++®, and Blackfin®
- Eclipse™ IDE
- Green Hills MULTI®

The following IDE's and processor families do not use target preferences blocks:

- Freescale MPC5xx
- Altium TASKING®
- Infineon C166®

The information in this section does not apply to them.

What are Target Preferences Blocks?

A target preferences block describes the environment for which you are generating code. The block includes information about the processor, hardware settings, operating system, memory mapping, and code generation

features. The Real-Time Workshop, Embedded IDE Link, and Simulink products use this information to generate code from your model.

Locating a Target Preferences Block

Target preferences blocks are located in:

- The Target Support Package block libraries for Supported Processors.
- The Embedded IDE Link™ block libraries for Supported IDEs.

To find a target preferences block:

- Use the search feature in the Simulink Library Browser.
- Browse the block libraries for your processor or IDE.

You can identify a target preference block by its board icon and label. The label includes the processor name or "Custom Board". For example:

Configuring a Target Preferences Block for a Supported Processor

Before you can generate code for a model, your model must contain a target preferences (TP) block.

If you are using a supported processor, and a preconfigured TP block is not available from Target Support Package block libraries, configure a TP block for your processor.

To configure a TP block for a supported processor:

1. Open the block library for your IDE.
2. Copy the Custom Board block to your model.

3. Open the Custom Board block.

4. Select your target processor from the **Processor** parameter, verify the default settings, and click **OK**. This action imports the appropriate default settings and applies them to the model.

5. In your model, edit the label of the TP block with the name of your processor.

To make reusing the TP block easier:

1. In your model, select **File** > **New** > **Library**.

2. Copy your new TP block to the library.

3. Save the library in your default Current Folder in MATLAB.

When you need the block again, open the library by entering the library name on the MATLAB command line.

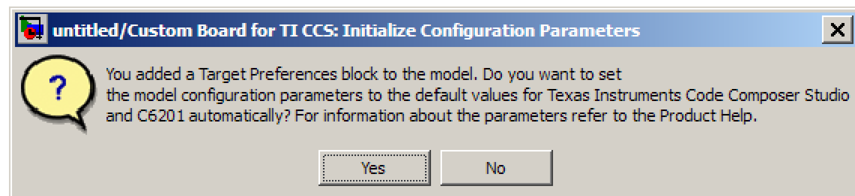
Adding a Target Preferences Block to Your Model

Before you can generate code for a model, your model must contain a target preferences (TP) block.

To add a TP block to your model:

1. Copy a TP block from a block library to your model, or create one, as described in “Configuring a Target Preferences Block for a Supported Processor” on page 2-3.

2. Click **Yes** if you get a dialog box that asks whether to “set the model configuration parameters to the default values”. For example:



This action applies the appropriate default settings for your IDE and processor to the Configuration Parameters dialog box.

Clicking **No** dismisses the dialog box and does not set the parameters. If the configuration parameters are incorrect, the software will generate error messages when you generate code . For more information, see "Setting Configuration Parameters for Embedded IDE Link" on page 2-6.

3. Open the TP block, verify the default settings, and click **OK**. This action applies the appropriate default settings to the model.

Note Your model must contain only one TP block.

Other tips for using TP blocks:

- The TP block stands alone. It does not connect to other blocks.
- To generate code for a model, place the TP block at the top level of your model.
- To generate code for a subsystem, place the TP block at the subsystem level of your model.
- For detailed information about the TP block parameters, see Target Preferences/Custom Board.

Examples of Configuring Target Preferences

There is no generic procedure for configuring a target preferences block. Setting the **Processor** parameter applies the appropriate default for a specific processor.

You typically reconfigure TP to achieve a specific purpose. For example:

- "Configuring a Target Preferences Block for a Supported Processor" on page 2-3
- "Generating Code for Any C64x+™ Processor or Board"

Setting Configuration Parameters for Embedded IDE Link

In this section...

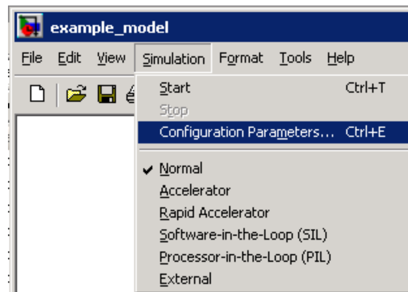
"What are Configuration Parameters?" on page 2-6

"Setting Model Configuration Parameters" on page 2-6

What are Configuration Parameters?

The **Configuration Parameters** dialog box specifies the settings for a model's active *configuration set*. These parameters determine the type of solver used, import and export settings, and other values that determine how the model runs. See Configuration Sets for more information.

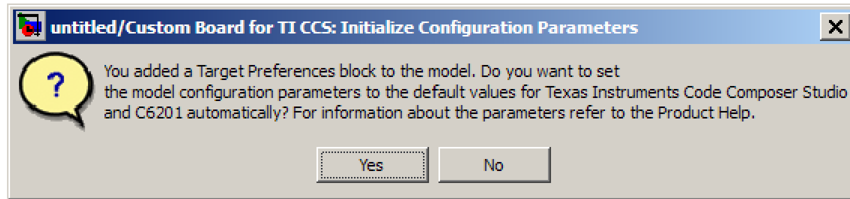
To display the dialog box, select **Simulation > Configuration Parameters** in the Model Editor, or press **Ctrl+E**. The dialog box appears.



For comprehensive information about configuration parameters in Simulink see, "Configuration Parameters Dialog Box"

Setting Model Configuration Parameters

The Embedded IDE Link software sets the appropriate default values for your processor and IDE when you drop a target preferences block in your model and click **Yes** in response to dialog box that asks whether to "set the model configuration parameters to the default values". For example:



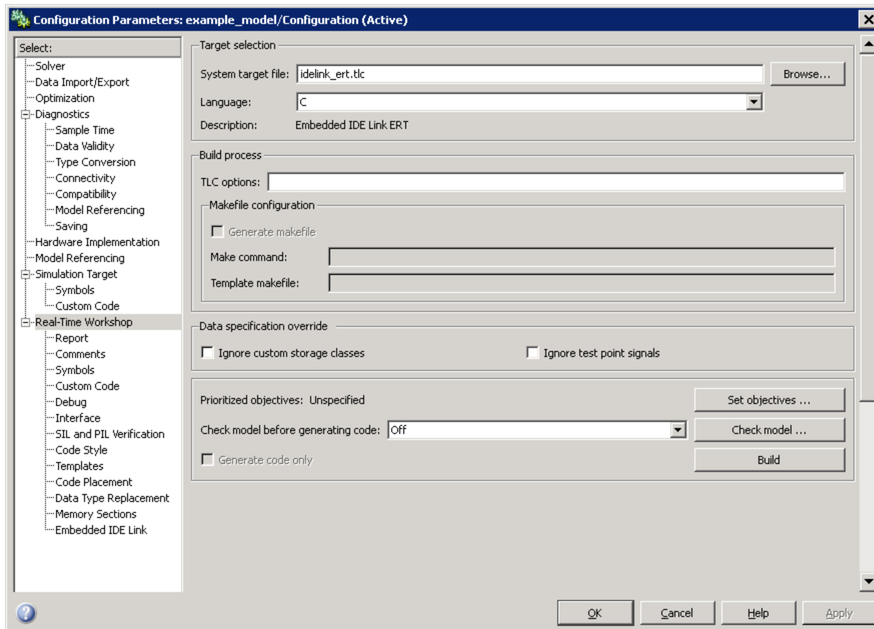
The following subsections provides a quick overview of the panes and parameters you are most with which you are most likely to interact.

efer to “About the TLC Debugger” in your Real-Time Workshop processor Language Compiler documentation.

Note The following subtopics assume you’ve added a target preferences block to your model and accepted the default values.

Real-Time Workshop Pane

The default **System target file** is `idelink_ert.tlc`. When you select `idelink_ert.tlc` or `idelink_grt.tlc`, the dialog box displays a new pane for Embedded IDE Link at the bottom of the select tree.



To use Real-Time Workshop® Embedded Coder™ software or the Processor-in-the-Loop feature, leave **System target file** set to `idelink_ert.tlc`.

Disregard the **Build process** options. Embedded IDE Link software does not use makefiles to generate code. Code generation is project based so the options in this group do not apply.

Note Embedded IDE Link software has a separate feature that automatically generates makefiles, which you can use to build applications with your software development toolchain. For more information, see Generating Makefiles.

If you generate code from a model that uses custom storage classes (CSC), leave **Ignore custom storage classes** unselected.

To use a system target file that does not support CSCs, such as `idelink_grt.tlc`, without reconfiguring your parameter and signal objects, select **Ignore custom storage classes**. When you select **Ignore custom storage classes**:

- The software treats objects with CSCs as if you set their storage class attribute to `Auto`.
- The storage class of signals that have CSCs does not appear on the signal line, even when you select `Storage class` from **Format > Port/Signals Display** in your Simulink menus.

Embedded IDE Link Pane Parameters

On the select tree, the Embedded IDE Link entry provides options in these areas:

- **Run-Time** — Set options for run-time operations, like the build action
- **Project Options** — Set build options for your project code generation
- **Code Generation** — Configure your code generation requirements
- **Link Automation** — Export an IDE handle object, such as `IDE_Obj`, to your MATLAB workspace
- **Diagnostic options** — Determine how the code generation process responds when you use source code replacement, either in the Target Preferences block **Board custom code** options, or in the Real-Time Workshop® **Custom Code** options in the configuration parameters.

For more information, see Embedded IDE Link Users Guide .

Build format. Select `Project` to build a project for your IDE, or select `Makefile` to generate a makefile for your development tool chain.

For more information, see Build Format.

Build action. Your selection for **Build action** determines what happens when you click **Build** or press **Ctrl+B**. Your selection tells Real-Time Workshop software when to stop the code generation and build process.

To run your model on the processor, select `Build_and_execute`. This selection is the default build action; Real-Time Workshop software automatically downloads and runs the model on your board.

The actions are cumulative—each listed action adds features to the previous action on the list and includes all the previous features.

If you set **Build format** to `Project`, select one of the following options:

- `Create_Project` — Directs Real-Time Workshop software to start the IDE and populate a new project with the files from the build process. This option offers a convenient way to build projects in the IDE.
- `Archive_library` — Directs Real-Time Workshop software to archive the project for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your the IDE projects for models that you use in model referencing.
- `Build` — Builds the executable COFF file, but does not download the file to the processor.
- `Build_and_execute` — Directs Real-Time Workshop software to build, download, and run your generated code as an executable on your processor.
- `Create_processor_in_the_loop_project` — Directs the Real-Time Workshop code generation process to create PIL algorithm object code as part of the project build.

If you set **Build format** to `Makefile`, select one of the following options:

- `Create_makefile` — Creates a makefile.
- `Archive_library` — Creates a makefile and an archive library.
- `Build` — Creates a makefile and an executable.
- `Build_and_execute` — Creates a makefile and an executable. Then it evaluates the execute instruction in the current configuration. For more information, see Execute.

Note When you build and execute a model on your processor, the Real-Time Workshop software build process resets the processor automatically. You do not need to reset the board before building models.

For more information, see Build action.

Overrun notification. To enable the overrun indicator, choose one of three ways for the processor to respond to an overrun condition in your model:

- `None` — Ignore overruns encountered while running the model.
- `Print_message` — When the DSP encounters an overrun condition, it prints a message to the standard output device, `stdout`.
- `Call_custom_function` — Respond to overrun conditions by calling the custom function you identify in **Function name**.

For more information, see "Overrun notification".

Function name. When you select `Call_custom_function` from the **Overrun notification** list, you enable this option. Enter the name of the function the processor should use to notify you that an overrun condition occurred. The function must exist in your code on the processor.

For more information, see "Function name".

Configuration. The **Configuration** parameter defines sets of build options that apply to all of the files generated from your model.

The `Release` and `Debug` option apply build settings that are defined by your IDE. For more information, refer to your IDE documentation.

`Custom` has the same default values as `Release`, but:

- Leaves **Compiler options string** empty and s
- Specifies a memory model that uses `Far Aggregate` for data and `Far` for functions.

For more information, see “Configuration”.

Compiler options string. To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, Embedded IDE Link does not set any optimization flags.

Click **Get From IDE** to import the compiler option setting from the current project in the IDE. To reset the compiler option to the default value, click **Reset**.

For more information, see “Compiler options string”.

Linker options string. To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, Embedded IDE Link does not set any linker options.

Click **Get From IDE** to import the linker options string from the current project in the IDE. To reset the linker options to the default value of no options, click **Reset**.

For more information, see “Linker options string”.

System stack size (MAUs). Enter the amount of memory to use for the stack. For more information, refer to **Enable local block outputs** on the **Optimization** pane of the Configuration Parameters dialog box. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory. Also refer to the online Help system for more information about Real-Time Workshop options for configuring and building models and generating code.

For more information, see “System stack size (MAUs)”.

System heap size (MAUs). Enter the amount of memory to use for the heap.

For more information, see “System heap size (MAUs)”.

Profile real-time execution. To enable the real-time execution profile capability, select **Profile real-time execution**. With this selected, the build process instruments your code to provide performance profiling at the task level or for atomic subsystems. When you run your code, the executed code reports the profiling information in an HTML report.

For more information, see “Profile real-time execution”.

Link Automation. When you use Real-Time Workshop to build a model for a processor, Embedded IDE Link makes a connection between MATLAB software and the IDE.

Constructors create objects that reference the link between the IDE and MATLAB. Link automation refers to the same object, named `IDE_Obj` in the function reference pages.

Although `IDE_Obj` is a bridge to a specific instance of the IDE, it is an object that contains information about the IDE instance it refers to, such as the board and processor it accesses. In this pane, the **Export IDE link handle to base workspace** option lets you instruct Embedded IDE Link to export the object to your MATLAB workspace, giving it the name you assign in **IDE link handle name**.

Maximum time allowed to build project (s). Specifies how long the software waits for the IDE to build the software.

For more information, see “Maximum time allowed to build project (s)”.

Maximum time allowed to complete IDE operations (s). Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages.

For more information, see “Maximum time allowed to complete IDE operations (s)”.

Export IDE link handle to base workspace. Directs the software to export the IDE_Obj object to your MATLAB workspace.

For more information, see “Export IDE link handle to base workspace”.

IDE link handle name. Specifies the name of the IDE_Obj object that the build process creates.

For more information, see "IDE link handle name".

Source file replacement. Selects the diagnostic action to take if the software detects conflicts when you replace source code with custom code. The diagnostic message responds to both source file replacement in the Embedded IDE Link parameters and in the Real-Time Workshop **Custom code** parameters in the configuration parameters for your model.

The following settings define the messages you see and how the code generation process responds:

- `none` — Does not generate warnings or errors when it finds conflicts.
- `warning` — Displays a warning. `warn` is the default value.
- `error` — Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

The build operation continues if you select `warning` and the software detects custom code replacement problems. You see warning messages as the build progresses

Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files. Use `none` when the replacement process is correct and you do not want to see multiple messages during your build.

For more information, see "Source file replacement".

Working with Block Libraries

For general information about working with block libraries in Simulink, see "Working with Block Libraries".

Simulink Models and Targeting

In this section...

"Creating Your Simulink Model for Targeting" on page 2-16

"Blocks to Avoid in Your Models" on page 2-17

Creating Your Simulink Model for Targeting

You create real-time digital signal processing models the same way you create other Simulink models—by combining standard DSP blocks and C-MEX S-functions.

You add blocks to your model in several ways:

- Use blocks from the Signal Processing Blockset software
- Use other Simulink discrete-time blocks
- Use the blocks provided for your processor family
- Use blocks that provide the functions you need from any blockset installed on your computer
- Create and use custom blocks

Once you have designed and built your model, you generate C code and build the real-time executable by clicking **Build** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. The automatic build process creates the file `modelname.out` containing a real-time model image in COFF file format that can run on your target.

The file `modelname.out` is an executable whose format is target-specific. You can load the file to your target and execute it in real time. Refer to your Real-Time Workshop documentation for more information about the build process.

Blocks to Avoid in Your Models

Many blocks in the blocksets communicate with your MATLAB workspace. All blocks generate code, but they do not work in the generated code as they do on your desktop.

You avoid using certain blocks, such as the Scope block and some source and sink blocks, in Simulink models that you use on Target Support Package targets. These blocks waste time in the generated code waiting to send or receive data from your MATLAB workspace, slowing your signal processing application without adding instrumentation value.

The following table describes blocks you should *not* use in your target models.

Block Name/Category	Library	Description
Scope	Simulink, Signal Processing Blockset software	Provides oscilloscope view of your output. Do not use the **Save data to workspace** option on the **Data history** pane in the Scope Parameters dialog box.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	Signal Processing Blockset	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	Signal Processing Blockset	Send data to your MATLAB workspace.

Block Name/Category	Library	Description
Signal To Workspace	Signal Processing Blockset	Send a signal to your MATLAB workspace.
Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
To Wave device	Signal Processing Blockset	Send data to a `.wav` device.
From Wave device	Signal Processing Blockset	Get data from a `.wav` device.

In general, using blocks to add instrumentation to your application is a valuable tool. In most cases, blocks you add to your model to display results or create plots, such as Histogram blocks, add to your generated code without affecting your running application.

3

Supported Operating Systems

- "Overview" on page 3-2
- "Preparing Models to Run on Windows or Linux" on page 3-3
- "Selecting the Operating System and Scheduling Mode" on page 3-5
- "Linux-Specific Topics" on page 3-6
- "Embedded Linux-Specific Topics" on page 3-9
- "Windows-Specific Topics" on page 3-10

Overview

You can build executables that run on Intel® x86 and Athlon/K5/K6 processors running Windows® and Linux®. For example, you can build executables and run them on your host development system or on a target operating system with the appropriate hardware and operating system.

If you are using MontaVista Linux Pro, you can also build executables that run on ARM® target processors running Embedded Linux. For example, you can build executables for the Texas Instruments™ TMS320DM355 DVEVM,

Preparing Models to Run on Windows or Linux

To build an executable that runs on Windows and Linux, perform the following steps:

1. Install and configure Eclipse IDE according to the instructions in "Getting Started" section of the *Embedded IDE Link User's Guide for Use with Eclipse IDE*.

   Note To run the executable on a remote target system, such as an ARM processor, also perform the steps in "Additional Configuration Steps to Run Your Executable on a Remote Linux Target".

   ---

2. Enter `idelinklib_eclipseide` at the MATLAB prompt. This action opens the Embedded IDE Link/Supported IDEs/Eclipse IDE library.

3. Copy the **Custom Board for Eclipse IDE** target preferences block to your model.

4. Click **Yes** in response to the **Initialize Configuration Parameters** dialog box.

5. Open the **Custom Board for Eclipse IDE** block, and set **Processor** to match the target processor. For example:

   - To run generated code on your host system, set **Processor** to `Intel x86/Pentium` or `AMD K5/K6/Athlon`.
   - To run generated code on a target system with an ARM processor, set **Processor** to one of the ARM processors.

6. Set **Operating System** to `None` or to the operating system on which you will be running the executable: `Windows`, or `Linux`. For example, if your host/development system is running Linux, set **Operating System** to `None` or `Linux`.

   - Selecting `Windows` creates a **Windows** tab, which you can use to set **Scheduling Mode**.

- Selecting `Linux` creates a **Linux** tab, which you can use to set **Scheduling Mode** and **Base Rate Priority**.

7. Set the **Scheduling Mode** to one of these options:
   - If you select `real-time`, the model uses a timer to trigger the base rate at regular periods.
   - If you select `free-running`, the model does not use a timer. It completes each process or thread before running the next one.
8. For Linux, you can set the **Base Rate Priority** relative to other processes and threads. You can enter values from (the number of rates + 1) to 99.
9. In Embedded IDE Link, configure the model to build and execute:
   a. In the model, select **Simulation** > **Configuration Parameters**.
   b. Select the **Real-Time Workshop** > **Embedded IDE Link** pane.
   c. Set **Build action** to `Build and execute`.
10. Build the model. Select **Tools** > **Real-Time Workshop** > **Build Model**.

    After the build completes, Embedded IDE Link software downloads the executable to the remote system and runs it.

Selecting the Operating System and Scheduling Mode

The following table refers to the **Operating System** and **Scheduling Mode** options in the **Custom Board for Eclipse IDE** target preferences block.

Operating System	Scheduling Mode	Behavior
Windows or Linux	free_running	The model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.
Windows or Linux	real_time	The model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a Linux real-time clock.
None	Not applicable	The model generates free-running code that runs in an infinite while loop with no timing.

For more information, see "Scheduling Considerations" in the *Real-Time Workshop User's Guide*.

Linux-Specific Topics

In this section...

"Scheduling" on page 3-6

"Running Multirate Multitasking Executables on the Linux Development System" on page 3-6

"Avoiding Lock-Up in Free-Running Multi-Rate Multi-Tasking Models" on page 3-8

Scheduling

The base rate in the model maps to a thread and runs as fast as possible. The base rate priority selection in the OS tab allows you to set a static priority for the base rate task. By default, this rate is 40.

The process running single-tasking models has Default scheduling policy when model is single-tasking or there is a single rate in the model. Static priority of the process is 0 in this case.

Running Multirate Multitasking Executables on the Linux Development System

On Linux, multirate multitasking executables require root privileges to schedule POSIX threads with real-time priority. For Eclipse IDE to run the multirate multitasking executable with root privileges locally on your Linux development system, Eclipse IDE must have root privileges.

If all three of the following conditions are true, start Eclipse IDE with root privileges:

- Your model produces a multirate multitasking executable.
- Embedded IDE Link is using the default configuration parameters which automatically run the executable. (In Configuration Parameters, under **Real-Time Workshop** > **Embedded IDE Link**, **Build format** is `Project` and **Build action** is `Build_and_execute`.)

- The Eclipse plug-in is using the default settings, which run the executable on the local Linux development system. (During the `eclipseidesetup` process, you left **Site** set to `local`.)

If any of the following conditions are true, you do not need to start Eclipse IDE with root privileges:

- Your model produces multirate single-tasking executables or single rate executables.
- You are performing a proccesor-in-the-loop (PIL) simulation.
- You are using a Windows development platform.
- You have configured Embedded IDE Link and Eclipse to run the executable on a remote Linux target.
- You not have configured Embedded IDE Link and Eclipse to run the executable.

Starting Eclipse IDE with root privileges

1. Install and configure Eclipse IDE according to the instructions in "Getting Started" section of the *Embedded IDE Link User's Guide for Use with Eclipse IDE*.

   Note To automatically build and run the executable on a remote Linux target, such as an ARM processor, also perform the steps described in "Additional Configuration Steps to Run Your Executable on a Remote Linux Target".

   ---

2. If an Eclipse IDE handle object exists in the MATLAB workspace, delete the object. For example, delete `IDE_Obj`.
3. If Eclipse IDE is running, close it.
4. Open a command-line session and `cd` to the Eclipse installation folder. For example, if you installed Eclipse in `usr/bin/eclipse`, enter:

   ```
   cd usr/bin/eclipse
   ```

5 Start Eclipse with root privileges using `sudo ./`. For example:

```
sudo ./eclipse
```

6 When prompted, enter the root password.

7 When Eclipse starts and prompts you for the workspace, enter the same workspace you specified during the Eclipse installation and configuration process.

8 In Embedded IDE Link, configure the model to build and execute:

   a In the model, select **Simulation** > **Configuration Parameters**.

   b Select the **Real-Time Workshop** > **Embedded IDE Link** pane.

   c Set **Build action** to `Build and execute`.

9 Build the model. Select **Tools** > **Real-Time Workshop** > **Build Model**.

   When the build process finishes, the multirate multitasking executable automatically starts and runs with root privileges.

Avoiding Lock-Up in Free-Running Multi-Rate Multi-Tasking Models

Use caution if you select free-running mode for a multi-rate multi-tasking models. Because of the Rate Monotonic Scheduling requirement in Linux, the scheduler runs threads with a SCHED_FIFO scheduling policy. A process scheduled with SCHED_FIFO prevents other process from running while it is ready to run. Therefore, if there are no blocking peripherals in the model, the entire Linux system can become unresponsive while you are running the generated code. Even the shell window will be preempted from running. To avoid this kind of lock-up, apply one of the following solutions:

- Set **Scheduling Mode** to `real_time`.
- Include a blocking device driver, such as a UDP block, in your model that suspends running thread while data is not available.
- Raise the shell window priority above the base rate priority so you can kill the process running with SCHED_FIFO class.

Embedded Linux-Specific Topics

Troubleshooting "sched_setaffinity: Bad address" Error

If the libc versions used on the host to build the executable do not match those used on the target to run the executable. When you build the model, the application terminates immediately with the following error:

```
**starting the model**
Call to sched_setaffinity returned error status (-1).
sched_setaffinity: Bad addres
```

To work around this problem, you can add `-static` to the linker options. However, linking the libraries statically increases the size of the executable. To configure the linker options, complete the following steps:

1. Press **Ctrl+E** to open the model configuration parameters.
2. Select **Real-Time Workshop** > **Embedded IDE Link**.
3. Add `-static` to the **Linker options string**.

To solve this problem, update the development and target software so they match. For example, in the case of the TMS320DM355 DVEVM, see the "Installing the Software" topic in Texas Instruments *TMS320DM355 DVEVM Getting Started Guide*, literature number SPRUF73.

Windows-Specific Topics

Scheduling

The base rate in the model is mapped to a thread and runs as fast as possible. In Windows target, the timer resolution is 1 ms. The base rate priority selection in the OS tab allows you to set a static priority for the base rate task. The Windows OS does not have a selection; the default base rate priority is set to THREAD_PRIORITY_HIGHEST (10) and the process running the generated code has NORMAL_PRIORITY_CLASS.

The process running single-tasking models has Default scheduling policy when model is single-tasking or there is a single rate in the model. Static priority of the process is 0 in this case.

4

Block Reference

Host CAN Blocks (canmsglib) (p. 4-2)	Target preference blocks for target boards
Host Communication (hostcommlib) (p. 4-3)	Target preference blocks for target boards

Host CAN Blocks (canmsglib)

CAN Pack	Pack individual signals into CAN message
CAN Unpack	Unpack individual signals from CAN messages

Host Communication (hostcommlib)

Byte Pack	Convert input signals to `uint8` vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP `uint8` input vector into Simulink data type values
UDP Receive	Receive UDP packet
UDP Send	Send UDP message

Blocks — Alphabetical List

Byte Pack

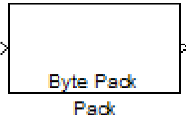
Purpose

Convert input signals to `uint8` vector

Library

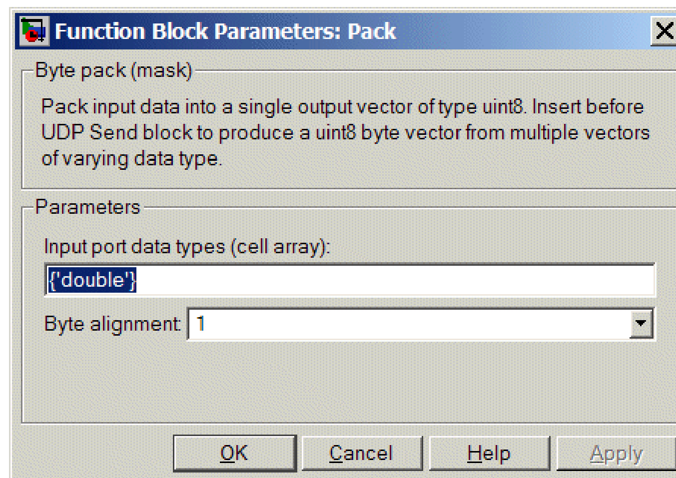
Host Communication (hostcommlib)

Description



Using the input port, the block converts data of one or more data types into a single `uint8` vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Because UDP messages are in `uint8` data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.

Dialog Box



Input port data types (cell array)

Specify the data types for the different signals as part of the parameters. The block supports all Simulink data types except characters. Enter the data types as Simulink types in the cell array, such as '`double`' or '`int32`'. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. This block determines the signal sizes automatically. The block always has at least one input port and only one output port.

Byte alignment

This option specifies how to align the data types to form the `uint8` output vector. Select one of the values in bytes from the list.

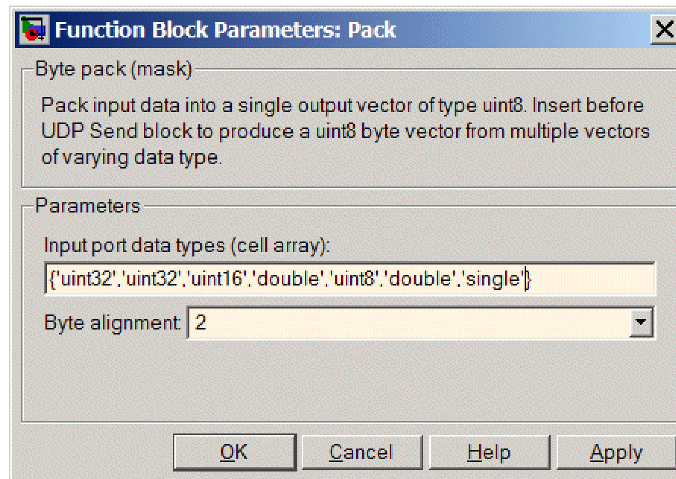
Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to `1`. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm ensures that each element in the output vector begins on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, with no holes between any data types for any combination of data types and signals.

Sometimes, you can have multiple data types of varying lengths. In such cases, specifying a 2-byte alignment can produce 1–byte gaps between uint8 or int8 values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify any of the data alignment options with any of the data types.

Example

Use a cell array to enter input data types in the **Input port data types** parameter. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data—`uint32`, `uint32`, `uint16`, `double`, `uint8`, `double`, and `single`. With this information, the block automatically provides the proper number of input ports.

Byte alignment equal to `2` specifies that each new value begins 2 bytes from the previous data boundary.

The example shows the following data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

When the signals are scalar values (no matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

See Also

Byte Reversal, Byte Unpack

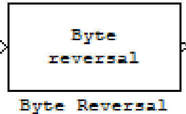
Purpose

Reverse order of bytes in input word

Library

Host Communication (hostcommlib)

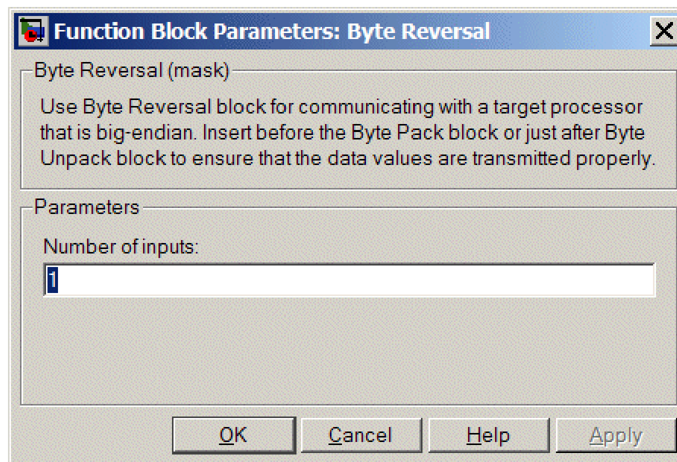
Description



Byte reversal changes the order of the bytes in data you input to the block. Use this block when your process communicates between targets that use different endianness, such as between Intel processors that are little endian and others that are big endian. Texas Instruments processors are little-endian by default.

To exchange data with a processor that has different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

Dialog Box



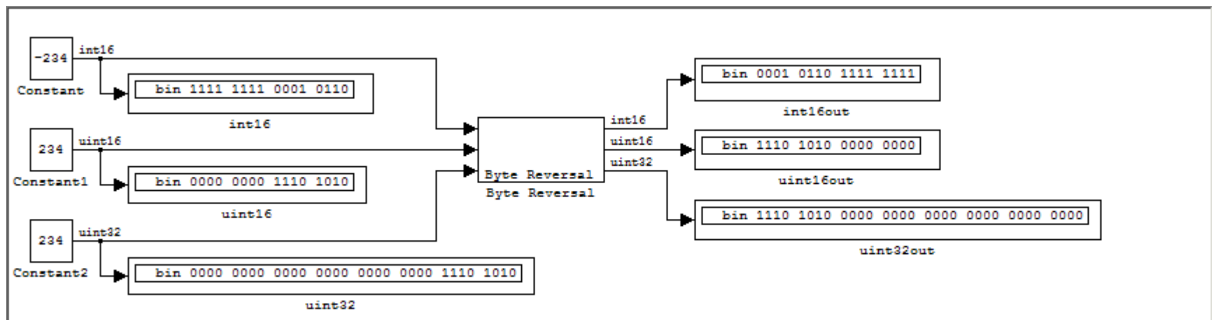
Number of inputs

Specify the number of input ports for the block. The number of input ports adjusts automatically to match value so the number of outputs equals the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1, and so on.

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. In this figure, the input and output ports match for each path.



See Also

Byte Pack, Byte Unpack

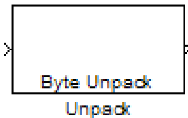
Purpose

Unpack UDP `uint8` input vector into Simulink data type values

Library

Host Communication (hostcommlib)

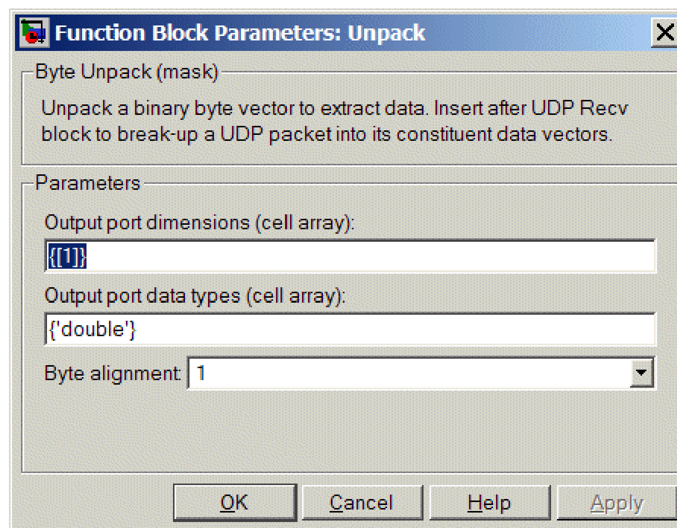
Description



Byte Unpack is the inverse of the Byte Pack block. It takes a UDP message from a UDP receive block as a `uint8` vector, and outputs Simulink data types in various sizes depending on the input vector.

The block supports all Simulink data types.

Dialog Box



Output port dimensions (cell array)
Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model. Entering one value means that the block applies that dimension to all data types.

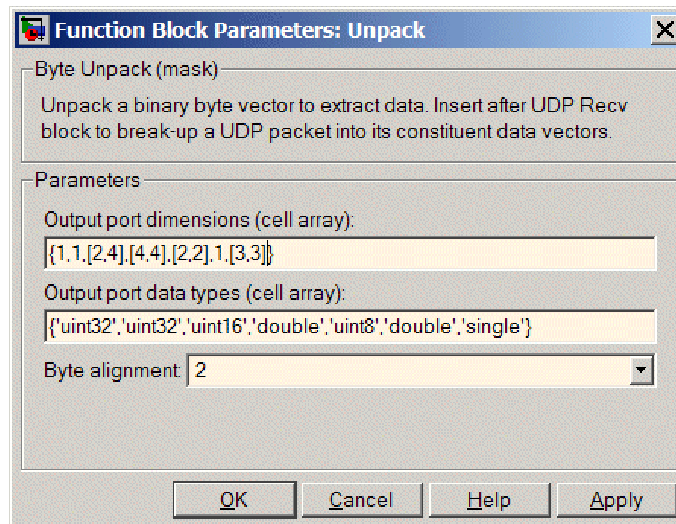
Output port data types (cell array)
Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types—`single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`, and `Boolean`. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

Byte Alignment
This option specifies how to align the data types to form the input `uint8` vector. Match this setting with the corresponding Byte Pack block alignment value of `1`, `2`, `4`, or `8` bytes.

Example

This figure shows the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry shown is the same as the **Input port data types (cell array)** entry in the Byte Pack block

```
{'uint32','uint32','uint16','double','uint8','double','single'}.
```



In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalar values and matrices to demonstrate entering nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

See Also

Byte Pack, Byte Reversal

CAN Pack

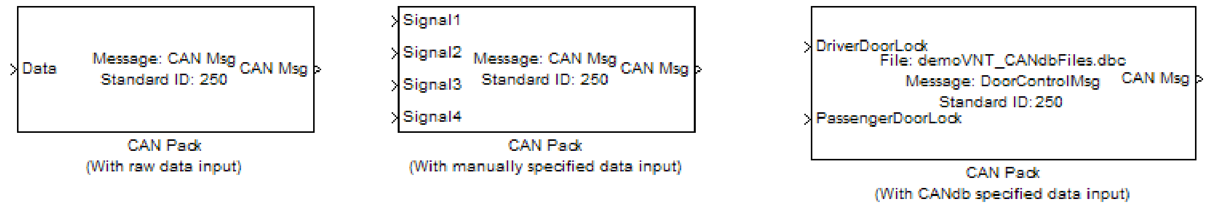
Purpose

Pack individual signals into CAN message

Library

CAN Communication

Description



CAN Pack
(With raw data input)

CAN Pack
(With manually specified data input)

CAN Pack
(With CANdb specified data input)

The CAN Pack block loads signal data into a message at specified intervals during the simulation.

Note To use this block, you also need a license for Simulink software.

CAN Pack block has one input port by default. The number of input ports is dynamic and depends on the number of signals you specify for the block. For example, if your block has four signals, it has four input ports.

CAN Pack

This block has one output port, CAN Msg. The CAN Pack block takes the specified input parameters and packs the signals into a message.

Other Supported Features

The CAN Pack block supports:

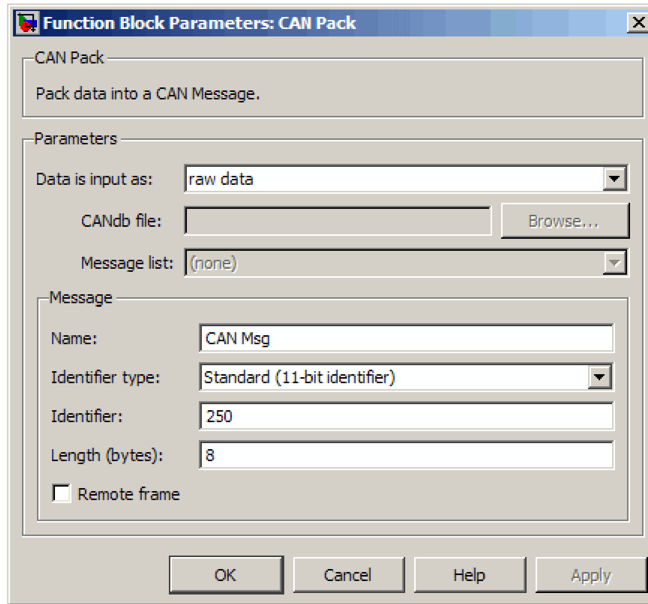
- The use of Simulink® Accelerator™ mode. Using this feature, you can speed up the execution of Simulink models.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Real-Time Workshop to deploy models to targets.

Note Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32-bits long.

For more information on these features, see the Simulink documentation.

Dialog Box

Use the Function Block Parameters dialog box to select your CAN Pack block parameters.

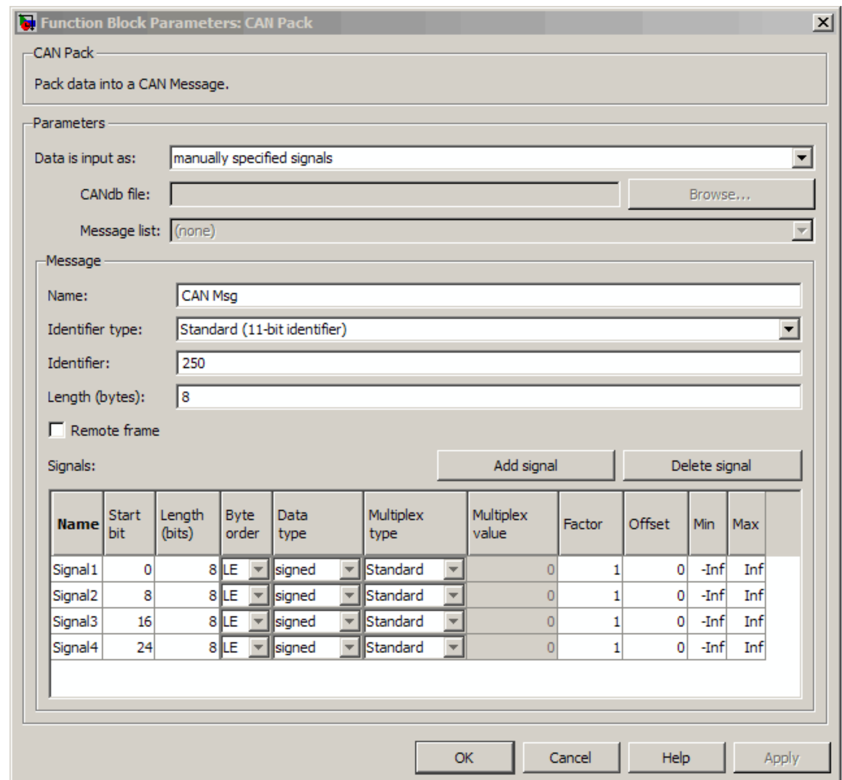


Parameters

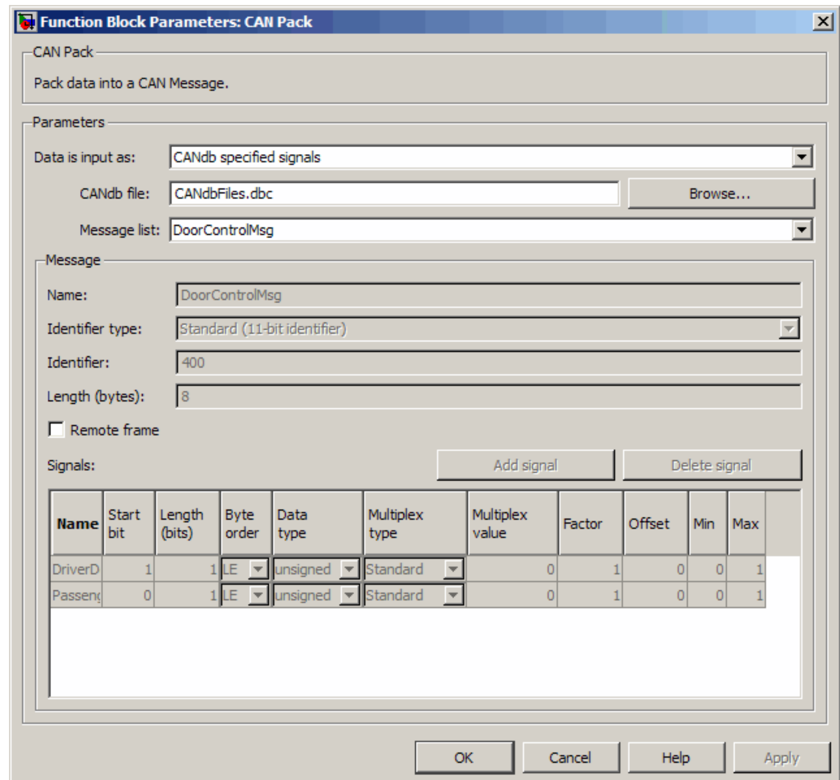
Data is input as

Select your data signal:

- **raw data**: Input data as a uint8 vector array. If you select this option, you only specify the message fields. All other signal parameter fields are unavailable. This option opens only one input port on your block.
- **manually specified signals**: Allows you to specify data signal definitions. If you select this option, use the **Signals** table to create your signals. The number of input ports on your block depends on the number of signals you specify.



- **CANdb specified signals**: Allows you to specify a CAN database file that contains message and signal definitions. If you select this option, select a CANdb file. The number of input ports on your block depends on the number of signals specified in the CANdb file for the selected message.



CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** list. Click **Browse** to find the appropriate CANdb file on your system. The message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

Message list

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** field and you select a CANdb

file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

Message

Name
Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to input raw data or manually specify signals. This option in unavailable if you choose to use signals from a CANdb file.

Identifier type
Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to input raw data or manually specify signals. For `CANdb specified signals`, the **Identifier type** inherits the type from the database.

Identifier
Specify your CAN message ID. This number must be a positive integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

Length (bytes)
Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your data input, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to input raw data or manually specify signals.

Remote frame
Specify the CAN message as a remote frame.

Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit any fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

Name
: Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit
: Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. The start bit must be an integer from 0 through 63.

Length (bits)
: Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order
: Select either of the following options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.

Bit Number

Data Byte Number

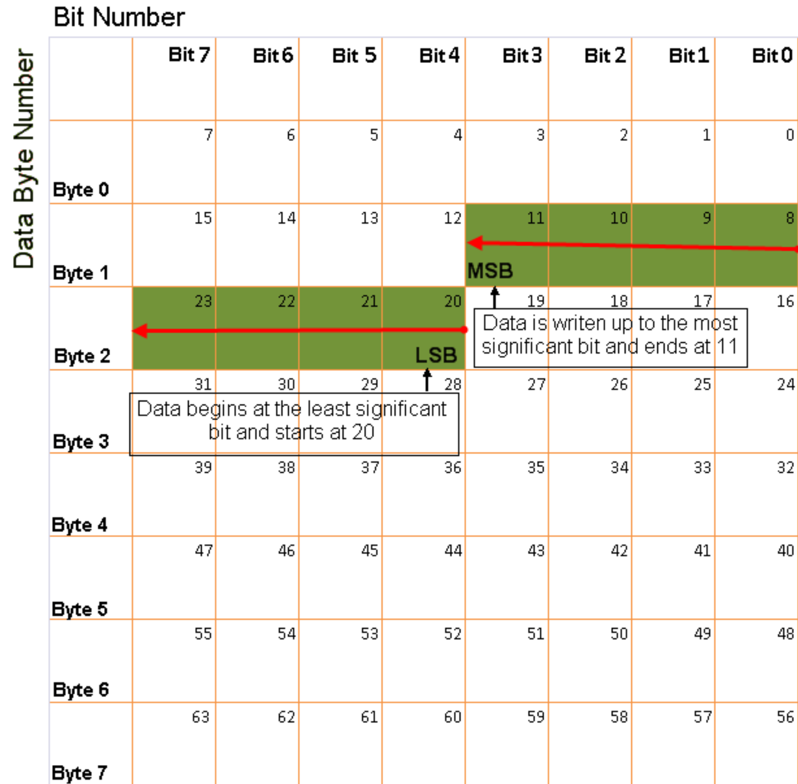
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15	14	13	12	11	10	9	8
Byte 2	23	22	21	20 LSB	19	18	17	16
Byte 3	31	30	29	28	27 MSB	26	25	24
Byte 4	39	38	37	36	35	34	33	32
Byte 5	47	46	45	44	43	42	41	40
Byte 6	55	54	53	52	51	50	49	48
Byte 7	63	62	61	60	59	58	57	56

Data begins at the least significant bit and starts at 20

Data is writen up to the most significant bit and ends at 27

Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Multiplex type

Specify how the block packs the signals into the CAN message at each timestep:

- `Standard`: The signal is always packed at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is always packed. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is packed if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	N/A
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	N/A

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block packs Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block packs Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that timestep.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to pack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See "Conversion Formula" on page 5-20 to understand how physical values are converted to raw values packed into a message.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See "Conversion Formula" on page 5-20 to understand how physical values are converted to raw values packed into a message.

Min

Specify the minimum physical value of the signal. The default value is `-inf` (negative infinity). You can specify any number for the minimum value. See "Conversion Formula" on page 5-20 to understand how physical values are converted to raw values packed into a message.

Max

Specify the maximum physical value of the signal. The default value is `inf`. You can specify any number for the maximum value. See "Conversion Formula" on page 5-20 to understand how physical values are converted to raw values packed into a message.

Conversion Formula

The conversion formula is

```
raw_value = (physical_value - Offset) / Factor
```

where `physical_value` is the value of the signal after it is saturated using the specified **Min** and **Max** values. `raw_value` is the packed signal value.

See Also

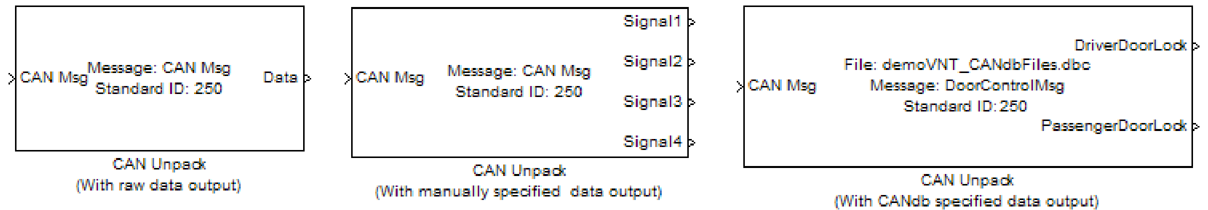
CAN Unpack

CAN Unpack

Purpose Unpack individual signals from CAN messages

Library CAN Communication

Description



The CAN Unpack block unpacks a CAN message into signal data using the specified output parameters at every timestep. Data is output as individual signals.

Note To use this block, you also need a license for Simulink software.

The CAN Unpack block has one output port by default. The number of output ports is dynamic and depends on the number of signals you specify for the block to output. For example, if your block has four signals, it has four output ports.

Other Supported Features

The CAN Unpack block supports:

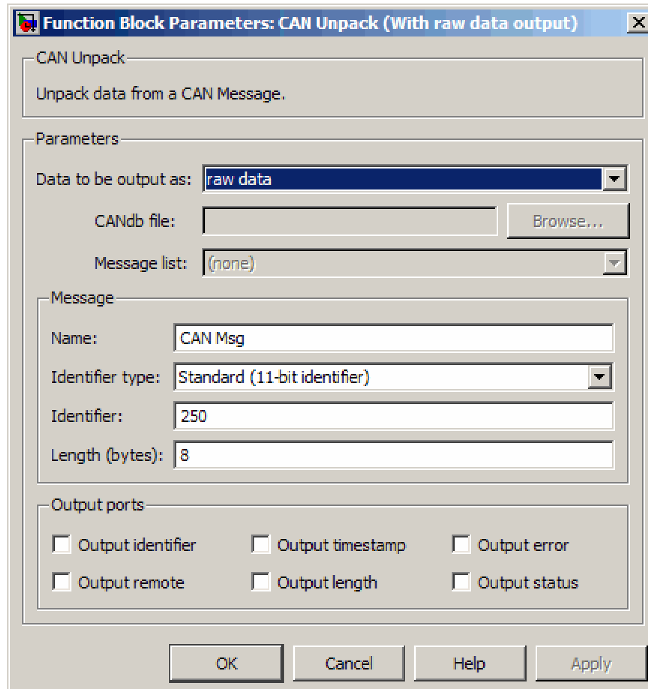
- The use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Real-Time Workshop to deploy models to targets.

Note Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32-bits long.

For more information on these features, see the Simulink documentation.

Dialog Box

Use the Function Block Parameters dialog box to select your CAN message unpacking parameters.

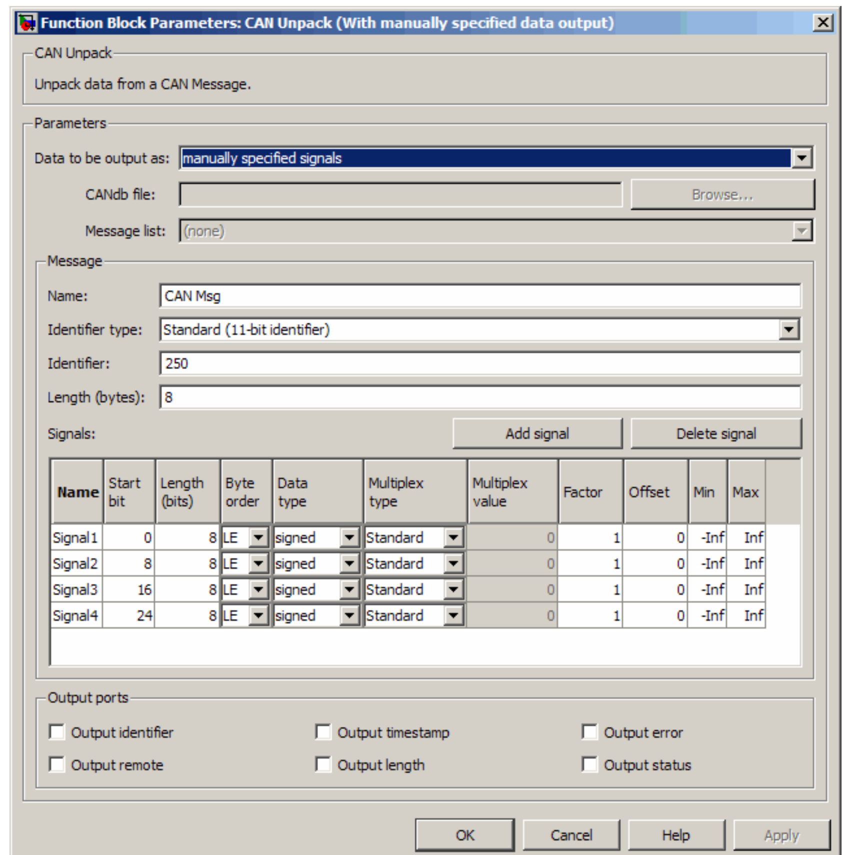


Parameters

Data to be output as

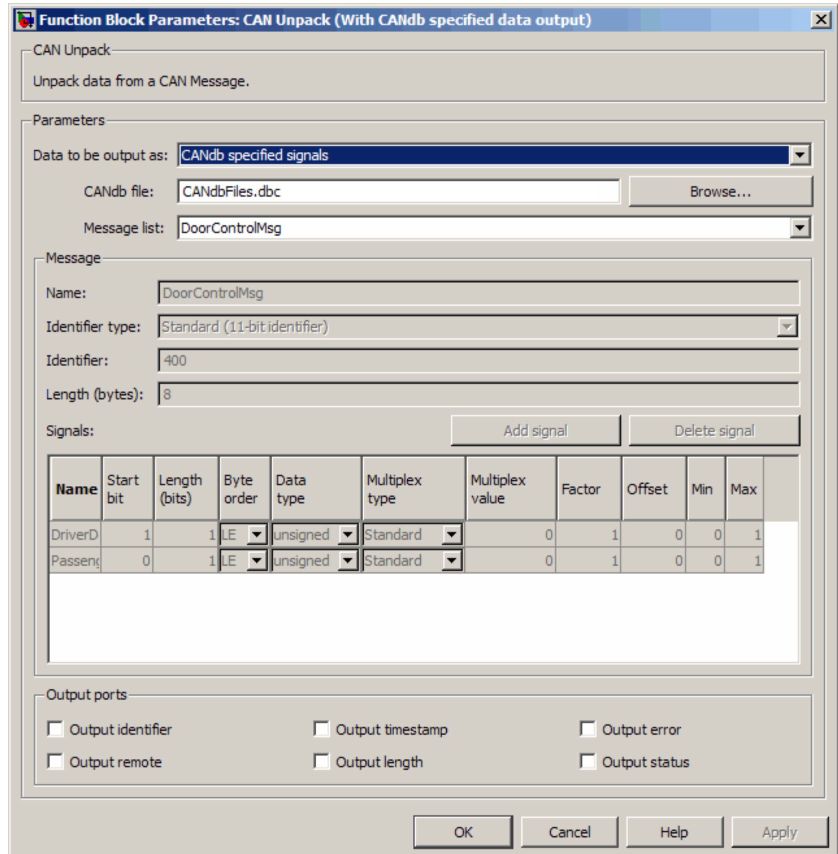
Select your data signal:

- **raw data**: Output data as a uint8 vector array. If you select this option, you only specify the message fields. All other signal parameter fields are unavailable. This option opens only one output port on your block.
- **manually specified signals**: Allows you to specify data signals. If you select this option, use the `Signals` table to create your signals message manually.



The number of output ports on your block depends on the number of signals you specify. For example, if you specify four signals, your block has four output ports.

- **CANdb specified signals**: Allows you to specify a CAN database file that contains data signals. If you select this option, select a CANdb file.



The number of output ports on your block depends on the number of signals specified in the CANdb file. For example, if the selected message in the CANdb file has four signals, your block has four output ports.

CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data to be output as** list. Click **Browse** to find the appropriate CANdb file on your system. The messages and signal definitions specified in the CANdb file populate the

Message section of the dialog box. The signals specified in the CANdb file populate **Signals** table.

Message list
This option is available if you specify that your data is to be output as a CANdb file in the **Data to be output as** list and you select a CANdb file in the **CANdb file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Message

Name
Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to output raw data or manually specify signals.

Identifier type
Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to output raw data or manually specify signals. For CANdb-specified signals, the **Identifier type** inherits the type from the database.

Identifier
Specify your CAN message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If you specify `1`, the block unpacks all messages that match the length specified for the message. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

Length (bytes)
Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your output data, the CANdb file defines the length of your message. If not, this field

defaults to `8`. This option is available if you choose to output raw data or manually specify signals.

Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit any fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

Name
Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit
Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message. The start bit must be an integer from 0 through 63.

Length (bits)
Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order
Select either of the following options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.

Bit Number

Data Byte Number

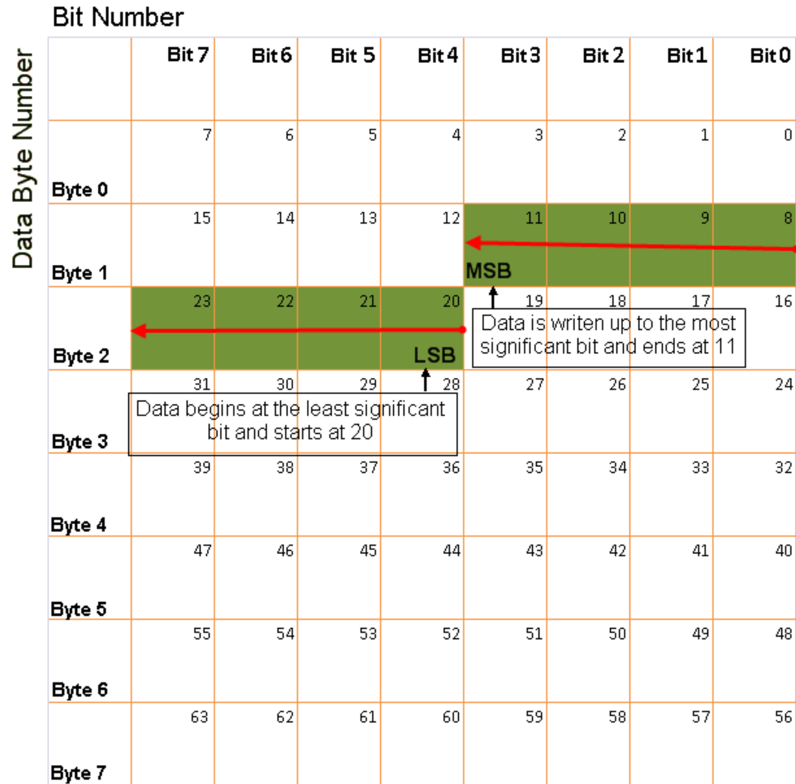
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15	14	13	12	11	10	9	8
Byte 2	23	22	21	20 LSB	19	18	17	16
Byte 3	31	30	29	28	27 MSB	26	25	24
Byte 4	39	38	37	36	35	34	33	32
Byte 5	47	46	45	44	43	42	41	40
Byte 6	55	54	53	52	51	50	49	48
Byte 7	63	62	61	60	59	58	57	56

Data begins at the least significant bit and starts at 20

Data is writen up to the most significant bit and ends at 27

Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Multiplex type

Specify how the block unpacks the signals from the CAN message at each timestep:

- `Standard`: The signal is always unpacked at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is always unpacked. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is unpacked if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	N/A
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	N/A

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block unpacks Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block unpacks Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that timestep.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to unpack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). See "Conversion Formula" on page 5-33 to understand how unpacked raw values are converted to physical values.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. See "Conversion Formula" on page 5-33 to understand how unpacked raw values are converted to physical values.

Min

Specify the minimum raw value of the signal. The default value is `-inf` (negative infinity). You can specify any number for the minimum value. See "Conversion Formula" on page 5-33 to understand how unpacked raw values are converted to physical values.

Max

Specify the maximum raw value of the signal. The default value is `inf`. You can specify any number for the maximum value. See "Conversion Formula" on page 5-33 to understand how unpacked raw values are converted to physical values.

Output Ports

Selecting an **Output ports** option adds an output port to your block.

Output identifier

Select this option to output a CAN message identifier. The data type of this port is **uint32**.

Output remote
Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

Output timestamp
Select this option to output the message time stamp. This option adds a new output port to the block. The data type of this port is **double**.

Output length
Select this option to output the length of the message in bytes. This option adds a new output port to the block. The data type of this port is **uint8**.

Output error
Select this option to output the message error status. This option adds a new output port to the block. The data type of this port is **uint8**.

Output status
Select this option to output the message received status. The status is `1` if the block receives new message and `0` if it does not. This option adds a new output port to the block. The data type of this port is **uint8**.

If you do not select any **Output ports** option, the number of output ports on your block depends on the number of signals you specify.

Conversion Formula

The conversion formula is

```
physical_value = raw_value *   Factor + Offset
```

where `raw_value` is the unpacked signal value. `physical_value` is the scaled signal value which is saturated using the specified **Min** and **Max** values.

See Also

CAN Pack

Purpose Receive UDP packet

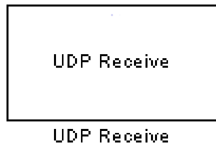
Library Host Communication (hostcommlib)

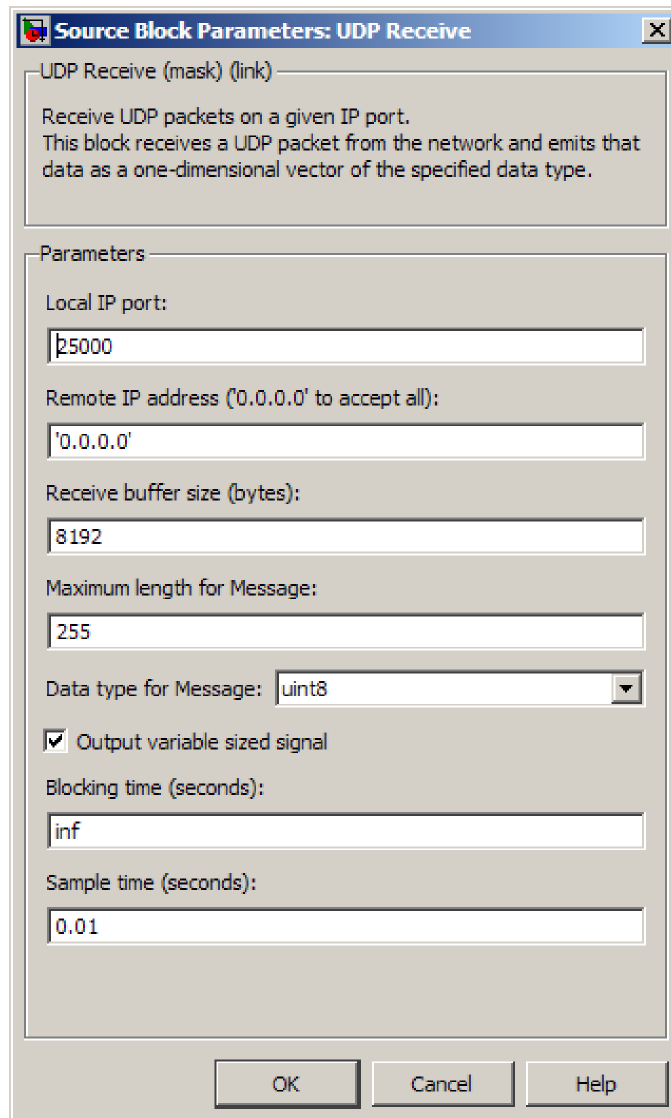
Linux (linuxlib)

Windows (windowslib)

Note If your target system uses Linux or Windows, get the UDP block from the appropriate library, `linuxlib` or `windowslib`.

Description The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block output, emits the contents of a single UDP packet as a data vector.





Dialog

Local IP port

Specify the IP port number upon to receive UDP packets. This value defaults to 25000. The value can range 1–65535.

Remote IP address (0.0.0.0 to accept all)
Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from any other address. To accept packets from any IP address, enter `'0.0.0.0'`. This value defaults to `'0.0.0.0'`.

Receive buffer size (bytes)
Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to `8192`.

Maximum length for Message
Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of any UDP packet. The system truncates data that exceeds this length. This value defaults to `255`.

If you disable **Output variable sized signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message
Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Output variable sized signal
If your model supports signals of varying length, enable the **Output variable sized signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable sized signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.

- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length
Set the data type of the Length output. This option defaults to `double`.

Blocking time (seconds)
For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note This parameter appears only in the UDP Receive block from the Target Support Package product.

Sample time (seconds)
Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Output port width
Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than any packet you expect to receive.

> **Note** This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block. For more information, see "New UDP Send and UDP Receive Blocks".

UDP receive buffer size (bytes)

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

> **Note** This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block. For more information, see "New UDP Send and UDP Receive Blocks".

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Send

Purpose

Send UDP message

Library

Host Communication (hostcommlib)

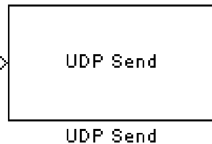
Linux (linuxlib)

Windows (windowslib)

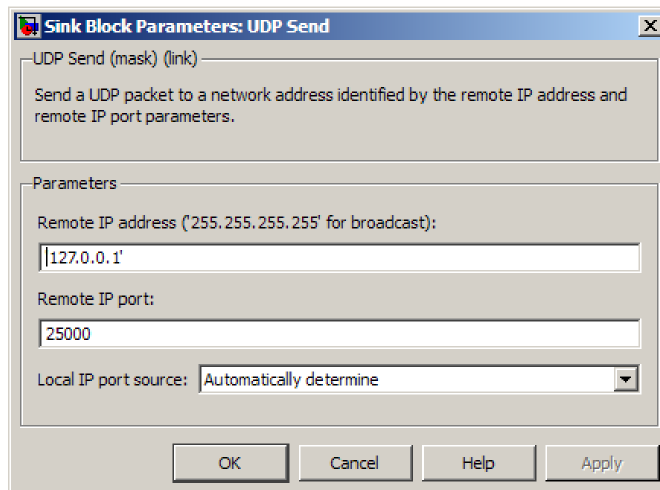
Note If your target system uses Linux or Windows, get the UDP block from the appropriate library, `linuxlib` or `windowslib`.

Description

The UDP Send block transmits an input vector as a UDP message over an IP network port.



Dialog Box



IP address (255.255.255.255 for broadcast)
Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, `'255.255.255.255'`.

Remote IP port
Specify the port to which the block sends the message. The value defaults to 25000, but the values range from 1–65535.

Local IP port source
To let the system automatically assign the port number, select `Assign automatically`. To specify the IP port number using the **Local IP port** parameter, select `Specify`.

Local IP port
Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Sample time
Sample time tells the block how long to wait before polling for new messages.

Note This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block. For more information, see "New UDP Send and UDP Receive Blocks".

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

Purpose

Custom or demo block

Description

This help topic serves as a landing page if you click the help button for a custom or demo block. These blocks are typically undocumented because they are not part of the standard block libraries.

To provide online help for custom blocks you create, see “Providing Your Own Help and Demos”.

Index

B

block recommendations 2-17
blocks
  CAN Pack 5-10
  CAN Unpack 5-22
  use in target models 2-17
blocks to avoid in models 2-17
Byte Pack block 5-2
Byte Reversal block 5-5
Byte Unpack block 5-7

C

CAN Pack block 5-10
CAN Unpack block 5-22
Custom Demo block 5-41

E

Embedded IDE Link™
  build format 2-9
  code generation options 2-9

G

generate optimized code 2-9

H

heap size, set heap size 2-12

O

optimization, processor specific 2-9

P

processor configuration options
  build action 2-9
  overrun action 2-11
processor specific optimization 2-9

S

select blocks for models 2-17
set heap size 2-12
set stack size 2-12
stack size, set stack size 2-12

T

table of blocks to avoid in models 2-17
Target Support Package™
  create Simulink® model for targeting 2-16
  expected background for use 1-5

U

UDP Receive block 5-34
UDP Send block 5-39