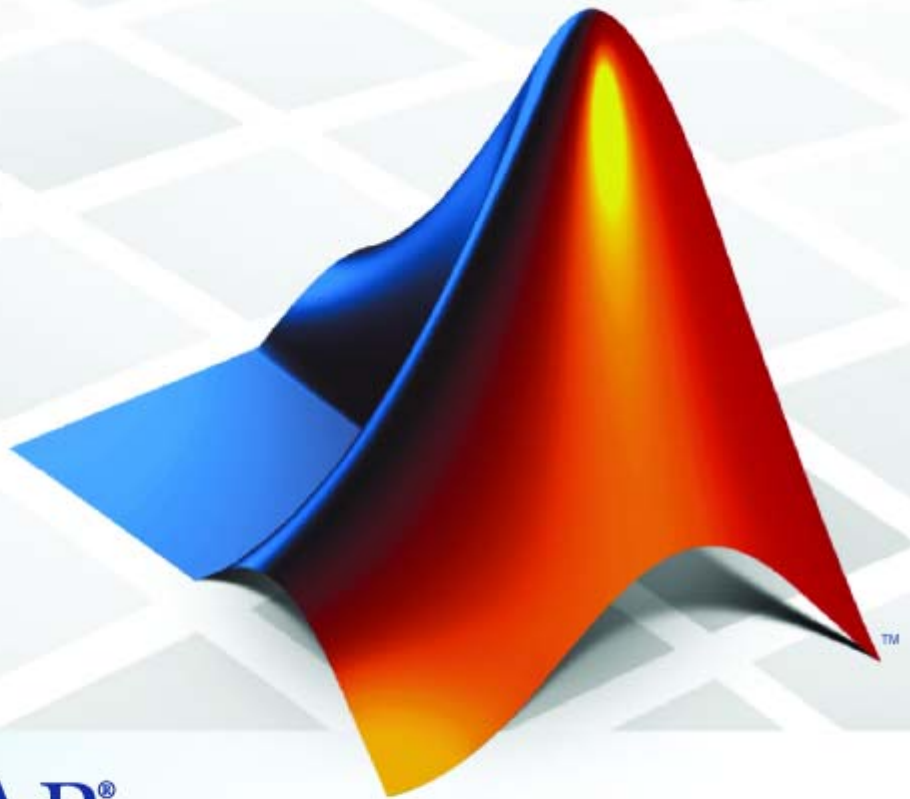


Vehicle Network Toolbox™ 1

User's Guide



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab@mathworks.com)
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



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

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Vehicle Network Toolbox™ User's Guide

© COPYRIGHT 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

March 2009	Online only	New for Version 1.0 (Release 2009a)
September 2009	Online only	Revised for Version 1.1 (Release 2009b)
March 2010	Online only	Revised for Version 1.2 (Release 2010a)

Getting Started

1

Product Overview	1-2
Getting to Know the Vehicle Network Toolbox	1-2
Main Features	1-2
Interaction Between the Toolbox and Its Components	1-4
Expected Background	1-5
Related Products	1-5
Installation Requirements	1-6
Supported Hardware	1-7
CAN Communication Session	1-9
Workflow Overview	1-9
Configuring CAN Communications	1-11
Saving and Loading a CAN Channel	1-21
Performing Advanced Configurations	1-22
Disconnecting Channels and Cleaning Up	1-27
Accessing the Toolbox	1-30
Exploring the Toolbox	1-30
Getting Help	1-30
Viewing Examples	1-30

Using a CAN Database

2

Vector CANdb Support	2-2
Loading and Creating Messages Using the .dbc File ...	2-3
Loading the CAN Database	2-3
Creating a CAN Message	2-3
Accessing Signals in the Constructed CAN Message	2-4
Adding a Database to a CAN Channel	2-4

Updating Database Information	2-5
Other Uses of the CAN Database	2-6
Viewing Message Information in the CAN Database	2-6
Viewing Signal Information in a CAN Message	2-7
Attaching a CAN Database to Existing Messages	2-7

Monitoring CAN Message Traffic

3

CAN Tool	3-2
Opening the CAN Tool	3-2
CAN Tool Fields	3-2
Using the CAN Tool	3-6
Viewing Messages on a Channel	3-6
Configuring the Channel Bus Speed	3-6
Saving the Message Log File	3-7
Viewing Unique Messages	3-7

Using the Vehicle Network Toolbox Block Library

4

Overview	4-2
Opening the Vehicle Network Toolbox Block Library	4-3
Using the MATLAB Command Window	4-3
Using the Simulink Library Browser	4-4
Building Simulink Models to Transmit and Receive Messages	4-5
Build a Message Transmit Model	4-5
Build a Message Receive Model	4-10

Save and Run the Model	4-17
------------------------------	------

Function Reference

5

CAN Channel Construction	5-2
CAN Channel Configuration	5-3
CAN Channel Execution	5-4
CAN Channel Status	5-5
CAN Database	5-6
CAN Message Handling	5-7
CAN Message Filtering	5-8
Information and Help	5-9
Graphical Tools	5-10
Vector Informatik	5-11

Functions — Alphabetical List

6

Property Reference

7

CAN Channel Base Properties	7-2
Channel Status Properties	7-2
CAN Message Properties	7-2
CAN Database Properties	7-3
Receiving Messages	7-3
Error Logging	7-4
Device-Specific Properties	7-5
Device Settings	7-5
Transceiver Settings	7-5
Bit Timing Settings	7-5

Properties — Alphabetical List

8

Block Reference

9

Index

Getting Started

- “Product Overview” on page 1-2
- “CAN Communication Session” on page 1-9
- “Accessing the Toolbox” on page 1-30

Product Overview

In this section...
“Getting to Know the Vehicle Network Toolbox” on page 1-2
“Main Features” on page 1-2
“Interaction Between the Toolbox and Its Components” on page 1-4
“Expected Background ” on page 1-5
“Related Products” on page 1-5
“Installation Requirements” on page 1-6
“Supported Hardware” on page 1-7

Getting to Know the Vehicle Network Toolbox

The Vehicle Network Toolbox™ provides the ability to communicate with in-vehicle networks using Controller Area Network (CAN) protocol. It is a comprehensive toolbox with a MATLAB® interface, Simulink® modeling support and a simple utility that allows you to monitor CAN traffic.

You can learn more about the Vehicle Network Toolbox by following a simple workflow and some easy examples. This chapter introduces the toolbox and provides some guidelines and examples to use the Vehicle Network Toolbox to interface with the CAN bus.

Main Features

The Vehicle Network Toolbox product is a collection of functions built on the MATLAB technical computing environment.

The toolbox provides you with these main features:

- “CAN Connectivity” on page 1-3
- “Vector Device and Driver Support” on page 1-3
- “Vehicle Network Toolbox Functions” on page 1-3
- “Simulink Library Support” on page 1-3

- “CAN Tool Interface” on page 1-3

CAN Connectivity

The Vehicle Network Toolbox provides host-side CAN connectivity using defined CAN devices. CAN is the predominant protocol in automotive electronics by which many distributed control systems in a vehicle function.

For example, in a common design when you press a button to lock the doors in your car, a control unit in the door reads that input and transmits lock commands to control units in the other doors. These commands exist as data in CAN messages, which the control units in the other doors receive and act on by triggering their individual locks in response.

Vector Device and Driver Support

You can use the Vehicle Network Toolbox with devices supported by Vector. These devices and drivers provide a link to the CAN bus on which you can send and receive messages.

See “Supported Hardware” on page 1-7 for more information.

Vehicle Network Toolbox Functions

Using a set of well-defined functions, you can transfer messages between the MATLAB workspace and a CAN bus using a CAN device. You can run test applications that can log and record CAN messages for you to process and analyze. You can also replay recorded sequences of messages.

Simulink Library Support

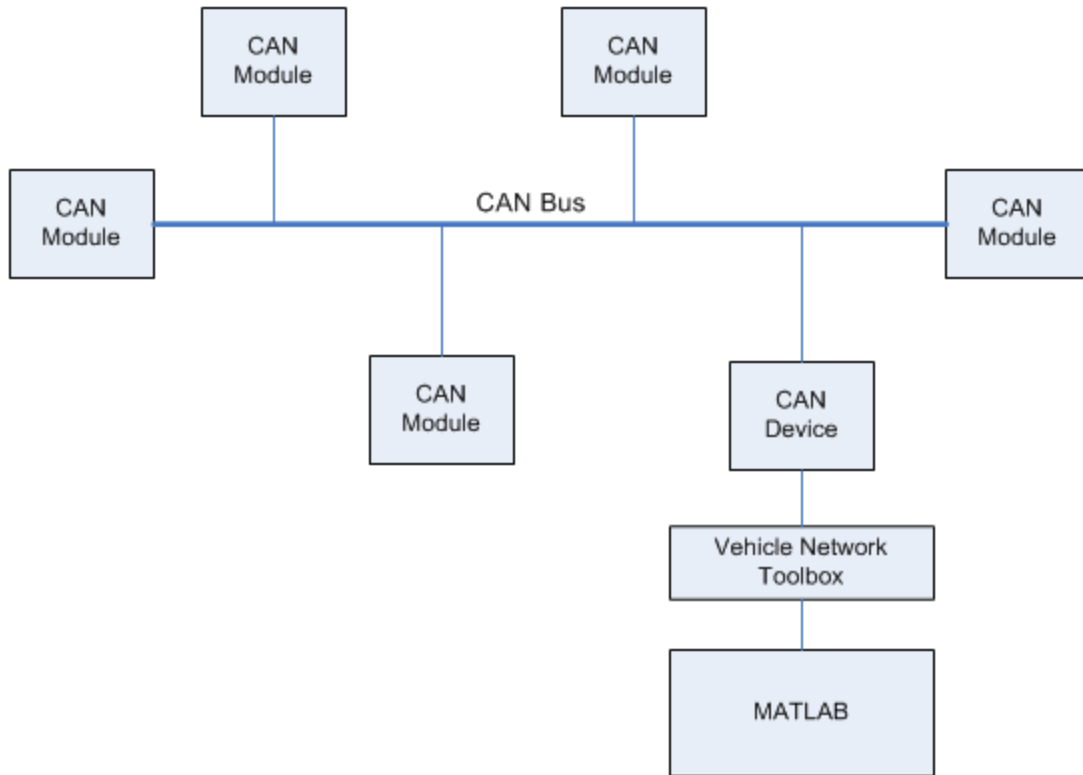
With the Vehicle Network Toolbox block library and other blocks from the Simulink library, you can create sophisticated models to connect to a live network and to simulate message traffic on a CAN bus.

CAN Tool Interface

Using this simple graphical user interface, you can monitor message traffic on a selected device and channel. You can then analyze these messages.

Interaction Between the Toolbox and Its Components

The Vehicle Network Toolbox is a conduit between MATLAB and the CAN bus.



In this illustration:

- Six CAN modules are attached to a CAN bus.
- One module, which is a CAN device, is attached to the Vehicle Network Toolbox, built on the MATLAB technical computing environment.

Using the Vehicle Network Toolbox from MATLAB, you can configure a channel on the CAN device to:

- Transmit messages to the CAN bus.
- Receive messages from the CAN bus.
- Trigger a callback function to run when the channel receives a message.
- Attach the database to the configured CAN channel to interpret received CAN messages.
- Use the CAN database to construct messages to transmit.
- Log and record messages and analyze them in MATLAB.
- Replay live recorded sequence of messages in MATLAB.
- Build Simulink models to connect to a CAN bus and to simulate message traffic.
- Monitor message traffic with the CAN Tool.

The Vehicle Network Toolbox is a comprehensive solution for CAN connectivity in MATLAB and Simulink. Refer to the function and block chapters for more information.

Expected Background

This document assumes that you are familiar with these products:

- MATLAB — To write scripts and functions, and to use functions with the command-line interface.
- Simulink — To create simple models to connect to a CAN bus or to simulate those models.
- Vector CANdb — To understand CAN databases and message and signal definitions.

Related Products

The MathWorks™ provides several products that are relevant to the kinds of tasks you can perform with the Vehicle Network Toolbox software and

that extend the capabilities of MATLAB. For information about these related products, see the toolbox product page on the MathWorks Web site.

Installation Requirements

- “Installing Components” on page 1-6
- “Installing Vector Hardware Devices and Drivers” on page 1-6
- “Installing Kvaser Hardware Devices and Drivers” on page 1-7
- “Installing the Toolbox” on page 1-7

Installing Components

To communicate on CAN networks from the MATLAB workspace, install these components:

- Current MATLAB version
- Vehicle Network Toolbox software
- Hardware, drivers, and driver libraries for your Vector or Kvaser devices

Installing Vector Hardware Devices and Drivers

You need the latest version of the XL Plug & Play drivers for your device to use with Windows® XP or Windows Vista™.

The documentation from Vector provides installation instructions for hardware devices such as CANcaseXL, CANboardXL, and CANcardXL, drivers, and support libraries.

These drivers are available for download from the Vector Web site at:

https://www.vector-worldwide.com/va_downloadcenter_us.html

Installing the XL Driver Library. Download and install the latest version of the XL Driver Library from the Vector Web site. After you install, copy the file `vxlapl.dll` from the installation folder to the `windows root\system32` directory.

Installing Kvaser Hardware Devices and Drivers

You need the latest version of the driver for your device to use with Windows XP or Windows Vista. Refer to your Kvaser device documentation for hardware installation instructions.

Drivers for your Kvaser devices are available on the Kvaser Web site at:

<http://www.kvaser.com/download/main.php>

Installing the Driver Library. Download and install the latest version of the SDK Driver Library from the Kvaser Web site.

Installing the Toolbox

Determine if Vehicle Network Toolbox software is installed on your system by typing the following in the MATLAB Command Window:

```
ver
```

The Command Window displays information about the MATLAB version you are running, including a list of installed add-on products and their version numbers. Check the list to see if the Vehicle Network Toolbox name appears.

For information about installing the toolbox, refer to the installation documentation for your platform. If you experience installation difficulties, look for the installation and license information at the MathWorks Web site:

<http://www.mathworks.com/support>

Supported Hardware

The Vehicle Network Toolbox supports Vector and Kvaser devices.

Supported Vector Devices

- CANcaseXL
- CANcaseXLe
- CANboardXL

- CANboardXL pxi
- CANboardXL PCIe
- CANcardXL
- CANcardX

You can also use the toolbox with virtual CAN channels available with Vector hardware drivers.

Supported Kvaser Devices

Support for Kvaser CAN devices, including these product families:

- WLAN
- PCMCIA
- Leaf
- Memorator
- PCI
- USB

You can also use the toolbox with virtual CAN channels available with Kvaser hardware drivers.

For a full list of devices, see the Supported Hardware page.

CAN Communication Session

In this section...
“Workflow Overview” on page 1-9
“Configuring CAN Communications” on page 1-11
“Saving and Loading a CAN Channel” on page 1-21
“Performing Advanced Configurations” on page 1-22
“Disconnecting Channels and Cleaning Up” on page 1-27

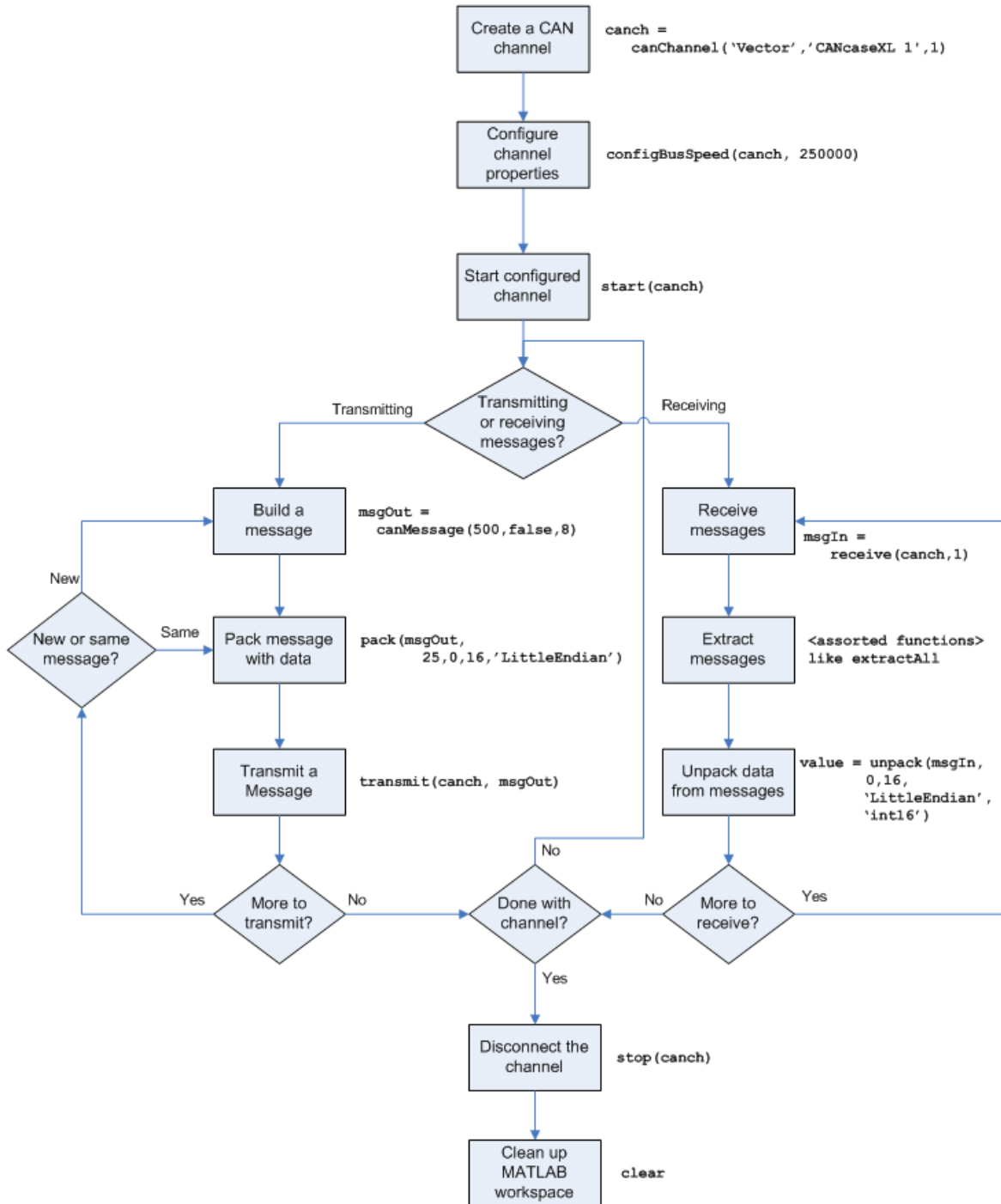
Workflow Overview

This section takes you through the workflow for connecting to a CAN device and then communicating with the CAN bus.

The subsequent sections map to the following CAN workflow chart.

Subsequent sections also provide interconnected code examples. You can use these examples and try them sequentially to understand how the communication works.

Typical CAN Workflow



Configuring CAN Communications

The following sections provide a sequential workflow for configuring CAN communications. You can use the provided examples and try them in a MATLAB Command Window to follow along.

This example creates two CAN channel objects using the `canHWInfo` function to obtain information about the devices installed on your system. You edit the properties of the first channel and create a message using the `canMessage` function. You transmit the message from the first channel using the `transmit` function, and receive it on the other using the `receive` function.

- “Prerequisites” on page 1-11
- “Checking for the Installed CAN Hardware” on page 1-12
- “Creating a CAN Channel Object” on page 1-13
- “Configuring Properties” on page 1-14
- “Starting the Configured Channel” on page 1-15
- “Creating a Message Object” on page 1-16
- “Packing a Message” on page 1-17
- “Transmitting a Message” on page 1-18
- “Receiving a Message” on page 1-19
- “Unpacking a Message” on page 1-21

Prerequisites

Before you follow this example, make sure you:

- Complete your toolbox installation before you try out the examples.
- Connect the two channels in your CAN device with a loopback connector.

The following examples use the Vector CANcaseXL hardware. You can substitute it with any other supported hardware.

Checking for the Installed CAN Hardware

- 1 Get information about the CAN hardware devices on your system:

```
info = canHWInfo
```

MATLAB displays the following information:

```
CAN Devices Detected:
```

```
Vector Devices:
```

```
CANcaseXL 1 Channel 1 (SN: 24365)
```

```
To connect, use - canChannel('Vector', 'CANcaseXL 1', 1)
```

```
CANcaseXL 1 Channel 2 (SN: 24365)
```

```
To connect, use - canChannel('Vector', 'CANcaseXL 1', 2)
```

```
Virtual 1 Channel 1
```

```
To connect, use - canChannel('Vector', 'Virtual 1', 1)
```

```
Virtual 1 Channel 2
```

```
To connect, use - canChannel('Vector', 'Virtual 1', 2)
```

```
Kvaser Devices:
```

```
Virtual 1 Channel 1
```

```
To connect, use - canChannel('Kvaser', 'Virtual 1', 1)
```

```
Virtual 1 Channel 2
```

```
To connect, use - canChannel('Kvaser', 'Virtual 1', 2)
```

```
Use GET on the output of CANHWINFO for more information.
```

- 2 Get details about all available CAN channels by typing:

```
info.VendorInfo(1).ChannelInfo(1)
```

- 3 Press **Enter**. MATLAB displays information such as:

```
can.vector.ChannelInfo handle
```

```
Package: can.vector
```

```
Properties:
```

```
    Device: 'CANcaseXL 1'  
DeviceChannelIndex: 1  
DeviceSerialNumber: 24365  
ObjectConstructor: 'canChannel('Vector', 'CANcaseXL 1', 1)'
```

Creating a CAN Channel Object

Note This example assumes that you have a loopback connection between the two channels on your CAN device.

- 1 Create the first CAN channel on an installed CAN device:

```
canch1 = canChannel('Vector', 'CANcaseXL 1', 1)
```

Notes You cannot use the same variable to create multiple channels sequentially. Clear any channel in use before using the same variable to construct a new CAN Channel.

You cannot create arrays of CAN channel objects. Each object you create must exist as its own individual variable.

- 2 Press **Enter** after you create the connection. MATLAB displays a summary of the channel properties:

```
Summary of CAN Channel using 'Vector' 'CANcaseXL 1' Channel 1.
```

```
Channel Parameters: Bus Speed is 500000.  
                   Bus Status is 'N/A'.  
                   Transceiver name is 'CANpiggy 251mag (Highspeed)'.  
                   Serial Number of this device is 24811.  
                   Initialization access is allowed.  
                   No database is attached.
```

```
Status: Offline - Waiting for START.  
0 messages available to RECEIVE.  
0 messages transmitted since last start.  
0 messages received since last start.
```

```
Filter History: Filters are open for Standard and Extended IDs.
```

3 Create a second CAN channel object.

```
canch2 = canChannel('Vector', 'CANcaseXL 1', 2)
```

You used the `canChannel` function to connect to the CAN device. To identify installed devices, use the `canHWInfo` function.

Configuring Properties

You can set the behavior of your CAN channel by configuring its property values. For this exercise, change the bus speed of channel 1 to 250000 using the `configBusSpeed` function.

Tip Configure property values before you start the channel.

1 Display the properties on `canch1`:

```
get(canch1)
```

MATLAB displays all properties on the configured channel:

```
General Settings:  
BusStatus = 'N/A'  
Database = []  
InitializationAccess = 1  
MessageReceivedFcn = []  
MessageReceivedFcnCount = 1  
MessagesAvailable = 0  
MessagesReceived = 0  
MessagesTransmitted = 0  
ReceiveErrorCount = 0  
Running = 0
```

```
SilentMode = 0
TransmitErrorCount = 0
```

```
Device Settings:
Device = 'CANcaseXL 1'
DeviceChannelIndex = 1
DeviceSerialNumber = 24811
DeviceVendor = 'Vector'
```

```
Transceiver Settings:
TransceiverName = 'CANpiggy 251mag (Highspeed)'
TransceiverState = 16
```

```
Bit Timing Settings:
BusSpeed = 500000
SJW = 1
TSEG1 = 4
TSEG2 = 3
NumOfSamples = 1
```

- 2 Change the BusSpeed property of the channel to 250000:

```
configBusSpeed(canch1, 250000)
```

- 3 To see the changed property value, type:

```
get(canch1)
```

MATLAB displays all properties on the configured channel as before, with the changed BusSpeed property value:

```
.
.
.
BusSpeed = 250000
```

- 4 Change the bus speed of the second channel (canch2) by repeating steps 2 and 3.

Starting the Configured Channel

Start your CAN channels after you configure all properties.

- 1 Start the first channel:

```
start(canch1)
```

- 2 Start the second channel:

```
start(canch2)
```

- 3 To check that the channel is online, type the channel name in the Command Window. The **Status** section indicates that the channel is now online, as in this example:

```
>> canch1
.
.
.
Status: Online.
      0 messages available to RECEIVE.
      0 messages transmitted since last start.
      0 messages received since last start.

Filter History: Filters are open for Standard and Extended IDs.
```

Creating a Message Object

After you set all the property values as desired and your channels are online, you are ready to transmit and receive messages on the CAN bus. For this exercise, transmit a message using `canch` and receive it using `canch1`. To transmit a message, create a message object and pack the message with the required data.

- 1 Build a CAN message of ID 500 of standard type and a data length of 8 bytes:

```
messageout = canMessage(500, false, 8)
```

The message object is now:

```
messageout =
    can.Message handle
```

```

Package: can

Properties:
  ID: 500
  Extended: 0
  Name: ''
  Database: []
  Error: 0
  Remote: 0
  Timestamp: 0
  Data: [0 0 0 0 0 0 0 0]
  Signals: []

```

Methods, Events, Superclasses

The fields in the message show:

- **can.Message (Normal Frame)** — Specifies that the message is not an error or a remote frame.
- **ID** — The ID you specified and its hexadecimal equivalent.
- **Extended** — A logical 0 (false) because you did not specify an extended ID.
- **Data** — A uint8 array of 0s specified by the data length.

Refer to the `canMessage` function to understand more about the input arguments.

You can also use a database to create a CAN message. Refer to Using a CAN Database for more information.

Packing a Message

After you define the message, pack it with the required data.

- 1 Use the `pack` function to pack your message with these input parameters:

```
pack(messageout, 25, 0, 16, 'LittleEndian')
```

Here you are specifying the `data` value to be 25, the start bit to be 0, the signal size to be 16, and the byte order to be little-endian format.

2 To see the packed data, type:

```
messageout
```

MATLAB displays your message properties with the specified data:

```
messageout =  
  
can.Message handle  
Package: can  
  
Properties:  
    ID: 500  
Extended: 0  
    Name: ''  
Database: []  
    Error: 0  
    Remote: 0  
Timestamp: 0  
    Data: [25 0 0 0 0 0 0 0]  
    Signals: []  
  
Methods, Events, Superclasses
```

The only field that changes after you specify the data is **Data**. Refer to the `pack` function to understand more about the input arguments.

Transmitting a Message

After you define the message and pack it with the required data, you are ready to transmit the message. For this example, use `canch` to transmit the message.

1 Use the `transmit` function to transmit the message, supplying the channel and the message as input arguments:

```
transmit(canch1, messageout)
```

2 To display the channel status, type:

```
canch1
```


MATLAB displays the updated status of the channel:

```
Summary of CAN Channel using 'Vector' 'CANcaseXL 1' Channel 1.
```

```
Channel Parameters: Bus Speed is 250000.
                   Bus Status is 'ErrorPassive'.
                   Transceiver name is 'CANpiggy 251mag (Highspeed)'.
                   Serial Number of this device is 24811.
                   Initialization access is allowed.
                   No database is attached.
```

```
Status: Online.
        1 messages available to RECEIVE.
        1 messages transmitted since last start.
        0 messages received since last start.
```

```
Filter History: Filters are open for Standard and Extended IDs.
```

In the **Status** section, messages transmitted since last start count increments by 1 each time you transmit a message.

Refer to the `transmit` function to understand more about the input arguments.

Receiving a Message

After your channel is online, use the `receive` function to receive available messages. For this example, receive the message on the second configured channel object, `canch2`.

1 To see messages available to be received on this channel, type:

```
canch2
```

The channel status displays available messages:

```
.
.
.
Status: Online.
        1 messages available to RECEIVE.
```

```
0 messages transmitted since last start.  
0 messages received since last start.
```

2 To receive one message from canch1 and store it as messagein, type:

```
messagein = receive(canch2, 1)
```

MATLAB returns the received message properties:

```
messagein =  
  
can.Message handle  
Package: can  
  
Properties:  
    ID: 500  
Extended: 0  
    Name: ''  
Database: []  
    Error: 0  
Remote: 0  
Timestamp: 709.0403  
    Data: [25 0 0 0 0 0 0 0]  
    Signals: []  
  
Methods, Events, Superclasses
```

3 To check if the channel received the message, type:

```
canch2
```

MATLAB returns the channel properties, and the status indicates that the channel received one message:

```
.  
. .  
. .  
Status: Online.  
    0 messages available to RECEIVE.  
    0 messages transmitted since last start.  
    1 messages received since last start.
```

Refer to the `receive` function to understand more about its input arguments.

Unpacking a Message

After your channel receives a message, specify how to unpack the message and interpret the data in the message. Use `unpack` to specify the parameters for unpacking a message:

```
value = unpack(messagein, 0, 16, 'LittleEndian', 'int16')
```

The unpacked message returns a value based on your parameters:

```
value =  
  
25
```

Refer to the `unpack` function to understand more about its input arguments.

Saving and Loading a CAN Channel

- “Saving a CAN Channel Object to a MATLAB File” on page 1-21
- “Loading a Saved CAN Channel” on page 1-21

Saving a CAN Channel Object to a MATLAB File

You can save a CAN channel object to a file using the `save` function anytime during the CAN communication session.

For example, create a channel object `canch1`. To save it to the MATLAB file `mycanch.mat`, type:

```
save mycanch.mat canch1
```

Loading a Saved CAN Channel

If you have saved a CAN channel as a MATLAB file, you can load it into a session using the `load` function. For example, to reload `mycanch.mat` created above, type:

```
load mycanch.mat
```

The loaded CAN channel object reconnects to the specified hardware and reconfigures itself to the specifications when the channel was saved.

Performing Advanced Configurations

- “Configuring Message Filtering” on page 1-22
- “Configuring Multiplexing” on page 1-23
- “Configuring Silent Mode” on page 1-26

Configuring Message Filtering

You can set up filters on your channel to accept messages based on the filtering parameters you specify. Set up your filters before putting your channel online. For more information on message filtering, see these functions:

- `filterAcceptRange`
- `filterBlockRange`
- `filterReset`
- `filterSet`

To specify a range of message IDs that you want the channel to accept, type:

```
stop(canch1)
filterAcceptRange(canch1, 500, 625)
```

Now you can build a message, and then pack, transmit, receive, and unpack it. If you display your channel settings, you see the status of the message filters on it.

```
canch1
Summary of CAN Channel using 'Vector' 'CANcaseXL 1' Channel 1.
.
.
.
Filter History: Filters are open for Standard and Extended IDs.
Block Range added. Starting ID: 0 Ending ID: 2047
Accept Range added. Starting ID: 500 Ending ID: 625
```

Configuring Multiplexing

Use multiplexing to represent multiple signals in one signal's location in a CAN message's data. A multiplexed message can have three types of signals:

Standard signal

This signal is always active. You can create one or more standard signals.

Multiplexor signal

Also called the mode signal, it is always active and its value determines which multiplexed signal is currently active in the message data. You can create only one multiplexor signal per message.

Multiplexed signal

This signal is active when its multiplex value matches the value of the multiplexor signal. You can create one or more multiplexed signals in a message.

Multiplexing works only with a CAN database with message definitions that already contain multiplex signal information. This example shows you how to access the different multiplex signals using a database constructed specifically for this purpose. This database has one message with these signals:

- SigA: A multiplexed signal with a multiplex value of 0.
- SigB: Another multiplexed signal with a multiplex value of 1.
- MuxSig: A multiplexor signal, whose value determines which of the two multiplexed signals are active in the message.

1 Create a CAN database:

```
d = canDatabase('Mux.dbc')
```

Note This is an example database constructed for creating multiplex messages. To try this example, use your own database.

2 Create a CAN message:

```
m = canMessage(d, 'Msg')
```

The message displays all its properties:

```
m =  
  
  can.Message handle  
  Package: can  
  
  Properties:  
    ID: 250  
    Extended: 0  
    Name: 'Msg'  
    Database: [1x1 can.Database]  
    Error: 0  
    Remote: 0  
    Timestamp: 0  
    Data: [0 0 0 0 0 0 0 0]  
    Signals: [1x1 struct]  
  
  Methods, Events, Superclasses
```

3 To display the signals, type:

```
m.Signals  
ans =  
  
    SigB: 0  
    SigA: 0  
    MuxSig: 0
```

MuxSig is the multiplexor signal, whose value determines which of the two multiplexed signals are active in the message. SigA and SigB are the multiplexed signals that are active in the message if their multiplex values match MuxSig. In the example shown, SigA is active because its current multiplex value of 0 matches the value of MuxSig (which is 0).

4 If you want to make SigB active, change the value of the MuxSig to 1:

```
m.Signals.MuxSig = 1
```

To display the signals, type:

```
m.Signals
```

```
ans =

    SigB: 0
    SigA: 0
    MuxSig: 1
```

SigB is now active because its multiplex value of 1 matches the current value of MuxSig (which is 1).

5 Change the value of MuxSig to 2:

```
m.Signals.MuxSig = 2
```

Here, neither of the multiplexed signals are active because the current value of MuxSig does not match the multiplex value of either SigA or SigB.

```
m.Signals

ans =

    SigB: 0
    SigA: 0
    MuxSig: 2
```

Always check the value of the multiplexor signal before using a multiplexed signal value.

```
if (m.Signals.MuxSig == 0)
    % Feel free to use the value of SigA however is required.
end
```

This ensures that you are not using an invalid value because the toolbox does not prevent or protect reading or writing inactive multiplexed signals.

Note You can access both active and inactive multiplexed signals regardless of the value of the multiplexor signal.

Refer to the `canMessage` function to learn more about creating messages.

Configuring Silent Mode

The `SilentMode` property of a CAN channel specifies that the channel can only receive messages and not transmit them. Use this property to observe all message activity on the network and perform analysis without affecting the network state or behavior. See `SilentMode` for more information.

- 1 Create a CAN channel object `canch1` and display its properties:

```
get(canch1)
```

MATLAB displays all properties on the configured channel:

General Settings:

```
BusStatus = 'N/A'  
Database = []  
InitializationAccess = 1  
MessageReceivedFcn = []  
MessageReceivedFcnCount = 1  
MessagesAvailable = 0  
MessagesReceived = 0  
MessagesTransmitted = 0  
ReceiveErrorCount = 0  
Running = 0  
SilentMode = 0  
TransmitErrorCount = 0
```

Device Settings:

```
Device = 'CANcaseXL 1'  
DeviceChannelIndex = 1  
DeviceSerialNumber = 24811  
DeviceVendor = 'Vector'
```

Transceiver Settings:

```
TransceiverName = 'CANpiggy 251mag (Highspeed)'  
TransceiverState = 16
```

Bit Timing Settings:

```
BusSpeed = 500000  
SJW = 1  
TSEG1 = 4
```



```
TSEG2 = 3
NumOfSamples = 1
```

- 2 Change the `SilentMode` property of the channel to `true`:

```
canch1.SilentMode = true
```

- 3 To see the changed property value, type:

```
get(canch1)
```

MATLAB displays all properties on the configured channel as before, with the changed `SilentMode` property value:

```
.
.
.
SilentMode = 1
.
.
.
```

Disconnecting Channels and Cleaning Up

- “Disconnecting the Configured Channel” on page 1-27
- “Cleaning Up the MATLAB Workspace” on page 1-28

Disconnecting the Configured Channel

When you no longer need to communicate with your CAN bus, disconnect the CAN channel that you configured. Use the `stop` function to disconnect.

- 1 Stop the first channel:

```
stop(canch1)
```

- 2 Check the channel status:

```
canch1
```

MATLAB displays the channel status:

```
.  
. .  
. . .  
Status: Offline - Waiting for START.  
      1 messages available to RECEIVE.  
      1 messages transmitted since last start.  
      0 messages received since last start.
```

3 Stop the second channel:

```
stop(canch2)
```

4 Check the channel status:

```
canch2
```

MATLAB displays the channel status:

```
Status: Offline - Waiting for START.  
      0 messages available to RECEIVE.  
      0 messages transmitted since last start.  
      1 messages received since last start.
```

Cleaning Up the MATLAB Workspace

When you no longer need the objects you used, remove them from the MATLAB workspace. To remove channel objects and other variables from the MATLAB workspace, use the `clear` function.

1 Clear the first channel:

```
clear canch1
```

2 Clear the second channel:

```
clear canch2
```

3 Clear the CAN messages:

```
clear('messageout', 'messagein')
```

4 Clear the unpacked value:

clear value

Accessing the Toolbox

In this section...
“Exploring the Toolbox” on page 1-30
“Getting Help” on page 1-30
“Viewing Examples” on page 1-30

Exploring the Toolbox

You can access the Vehicle Network Toolbox from the MATLAB Command Window directly by using any Vehicle Network Toolbox function. To see a list of all the functions available, type:

```
help vnt
```

Getting Help

The toolbox functions are grouped by usage. Click a specific function for more information.

To access the online documentation for the Vehicle Network Toolbox, type:

```
doc vnt
```

To access the reference page for a specific function, type:

```
doc function_name
```

Viewing Examples

To follow examples in this guide use the Vector CANcaseXL device, with the Vector XL Driver Library version 6.4 or later. The Examples index in the Help browser lists these examples.

Using a CAN Database

- “Vector CANdb Support” on page 2-2
- “Loading and Creating Messages Using the .dbc File” on page 2-3
- “Other Uses of the CAN Database” on page 2-6

Vector CANdb Support

The Vehicle Network Toolbox supports the use of a Vector CAN database. A .dbc file contains definitions of CAN messages and signals.

Use the Vehicle Network Toolbox to look up message and signal information and build messages using the information defined in the database file.

Loading and Creating Messages Using the .dbc File

In this section...

“Loading the CAN Database” on page 2-3

“Creating a CAN Message” on page 2-3

“Accessing Signals in the Constructed CAN Message” on page 2-4

“Adding a Database to a CAN Channel” on page 2-4

“Updating Database Information” on page 2-5

Loading the CAN Database

To use a CANdb file, load the database into your MATLAB session. At the MATLAB command prompt, type:

```
db = canDatabase('filename.dbc')
```

Here *db* is a variable you chose for your database handle and *filename.dbc* is the actual file name of your CAN database. If your CAN database is not in the current working directory, type the path to the database:

```
db = canDatabase('path\filename.dbc')
```

Tip CAN database file names containing non-alphanumeric characters such as equal signs, ampersands, and so forth are incompatible with the Vehicle Network Toolbox. You can use periods in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

This command returns a database object that you can use to create and interpret CAN messages using information stored in the database. Refer to the `canDatabase` function for more information.

Creating a CAN Message

This example shows you how to create a message using a database constructed specifically for a demo. You can access this database in the **Toolbox > VNT > VNTDemos** subfolder in your MATLAB installation

folder. This database has a message, EngineMsg. To try this example, create messages and signals using definitions in your own database.

- 1 Create the CAN database object:

```
d = canDatabase('demoVNT_CANdbFiles.dbc')
```

- 2 Create a CAN message using the message name in the database:

```
message = canMessage(d, 'EngineMsg')
```

Accessing Signals in the Constructed CAN Message

You can access the two signals defined for the message you created in the example database, message. You can also change the values for some signals.

- 1 To display signals in your message, type:

```
sig = (message.Signals)
```

Signals in the message are displayed as follows:

```
sig =  
  
    VehicleSpeed: 0  
    EngineRPM: 250
```

- 2 Change the value of the EngineRPM signal:

```
message.Signals.EngineRPM = 300
```

- 3 Display signal information again to see the change:

```
sig  
  
sig =  
  
    VehicleSpeed: 0  
    EngineRPM: 300
```

Adding a Database to a CAN Channel

To add a database to the CAN channel canch, type:


```
canch.Database = canDatabase('Mux.dbc')
```

For more information, see the Database property.

Updating Database Information

When you make changes to a database file:

- 1** Reload the database file into your MATLAB session using the `canDatabase` function.
- 2** Reattach the database to messages using the `attachDatabase` function.

Other Uses of the CAN Database

In this section...
“Viewing Message Information in the CAN Database” on page 2-6
“Viewing Signal Information in a CAN Message” on page 2-7
“Attaching a CAN Database to Existing Messages” on page 2-7

Viewing Message Information in the CAN Database

You can get information about the definition of messages in the database, a single message by name, or a single message by ID. To get message information about all messages in the database, type:

```
msgInfo = messageInfo(database name)
```

This command returns the message structure of information about messages in the database. For example:

```
msgInfo =  
  
5x1 struct array with fields:  
    Name  
    Comment  
    ID  
    Extended  
    Length  
    Signals
```

To get information about a single message by message name, type:

```
msgInfo = messageInfo(database name, 'message name')
```

This command returns information about the message as defined in the database. For example:

```
msgInfo = messageInfo(db, 'EngineMsg')  
  
msgInfo =
```

```
Name: 'EngineMsg'  
Comment: ''  
ID: 100  
Extended: 0  
Length: 8  
Signals: {2x1 cell}
```

Here the function returns information about message with name `EngineMsg` in the database `db`. You can also use the message ID to get information about a message. For example, to view the example message given here by inputting the message ID, type:

```
msgInfo = messageInfo(db, 100, false)
```

This command provides the database name, the message ID, and a Boolean value for the extended value of the ID.

To learn how to use it and work with the database, see the `messageInfo` function.

Viewing Signal Information in a CAN Message

You can get information about all signals in a CAN message. Provide the message name or the ID as a parameter in the command:

```
sigInfo = signalInfo(db, 'EngineMsg')
```

You can also get information about a specific signal by providing the signal name:

```
sigInfo = signalInfo(db, 'EngineMsg', 'EngineRPM')
```

To learn how to use this property and work with the database, see the `signalInfo` function.

Attaching a CAN Database to Existing Messages

You can attach a `.dbc` file to messages and apply the message definition defined in the database. Attaching a database allows you to view the messages in their physical form and use a signal-based interaction with the message data.

To attach a database to a message, type:

```
attachDatabase(message name, database name)
```

Note If your message is an array, all messages in the array are associated with the database that you attach.

You can also dissociate a message from a database so that you can view the message in its raw form. To clear the attached database from a message, type:

```
attachDatabase(message name, [])
```

Note The database gets attached even if the database does not find the specified message. Even though the database is still attached to the message, the message is displayed in its raw mode.

For more information, see the `attachDatabase` function.

Monitoring CAN Message Traffic

- “CAN Tool” on page 3-2
- “Using the CAN Tool” on page 3-6

CAN Tool

In this section...
“Opening the CAN Tool” on page 3-2
“CAN Tool Fields” on page 3-2

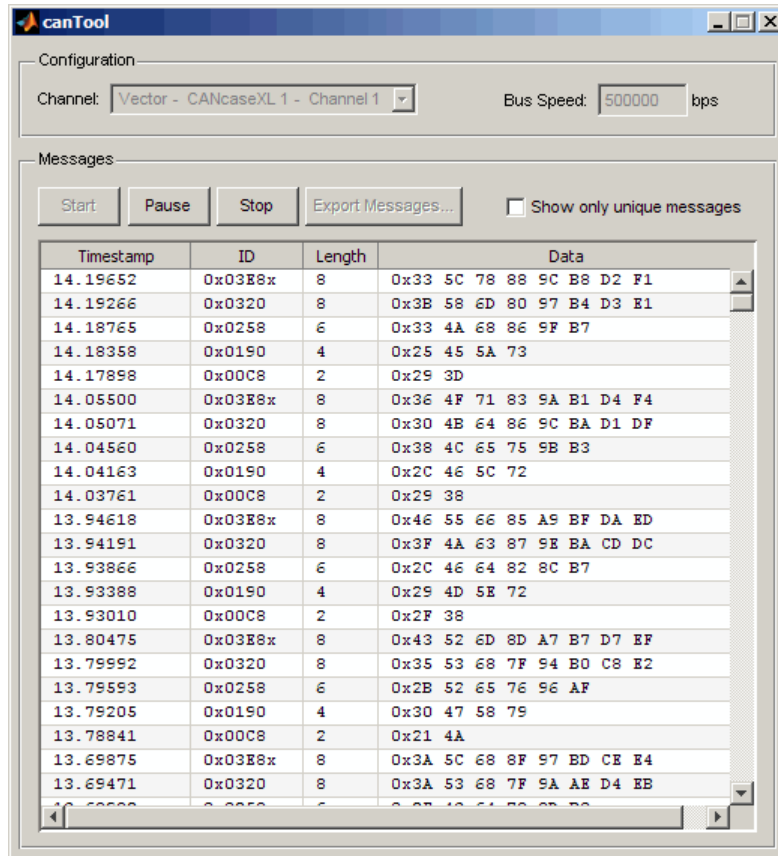
Opening the CAN Tool

The Vehicle Network Toolbox provides a graphical user interface that displays CAN message traffic on selected CAN channels.

To open the CAN Tool, type `canTool` in the MATLAB Command Window.

CAN Tool Fields

The CAN Tool is a simple interface that displays all messages received by a specific CAN channel. The tool has the following fields.



Configuration

Channel

Displays all available CAN devices and channels on your system.

Bus Speed

Displays the bus speed of the selected CAN channel. You can also change the bus speed of a channel. See Configuring the Channel Bus Speed.

Messages

Start

Click this button to view message activity on the selected channel.

Pause

Click this button to pause the display of message activity on the selected channel.

Stop

Click this button to stop displaying messages on the selected channel.

Export Messages

Click this button to export the current message list on the selected channel up to the latest message.

Show only unique messages

Select this check box to show the most recent instance of each message received on the selected channel. If you select this check box, the tool displays a simplified version of the message traffic. In this view, you will not see messages scroll up, but each message refreshes its data with each timestamp. If you do not select this option the tool displays all instances of all messages in the order that the selected channel receives them.

Messages Table

Timestamp

Displays the time, relative to the start time, that the device receives the message. The start time when you click **Start** in the tool starts at 0.

ID

Displays the message ID. This field displays a number in hexadecimal format for the ID and:

- Displays numbers only for standard IDs.
- Appends with an **x** for an extended ID.
- Displays an **r** for a remote frame.
- Displays **error** for messages with error frames.

Length

Displays the length of the message in bytes.

Data

Displays the data in the message in hexadecimal format.

Using the CAN Tool

In this section...
“Viewing Messages on a Channel” on page 3-6
“Configuring the Channel Bus Speed” on page 3-6
“Saving the Message Log File” on page 3-7
“Viewing Unique Messages” on page 3-7

Viewing Messages on a Channel

- 1 Open the CAN Tool and select the device and channel connected to your CAN bus from the **Channel** list.
- 2 The CAN Tool defaults to the bus speed set in the device driver. You can also configure a new bus speed. See *Configuring the Channel Bus Speed*
- 3 Click **Start**.
- 4 To pause the display, click **Pause**.
- 5 To stop the display, click **Stop**.

Configuring the Channel Bus Speed

Configure the bus speed when the speed of your network differs from the default value of the channel. You require initialization access for the channel to configure the bus speed.

To configure a new bus speed:

- 1 Type the desired value in the **Bus Speed** field.
- 2 Press **Enter**.

The value you set takes effect once you start the CAN channel. If an error occurs when applying the new bus speed, the value reverts to the default value specified in the hardware.

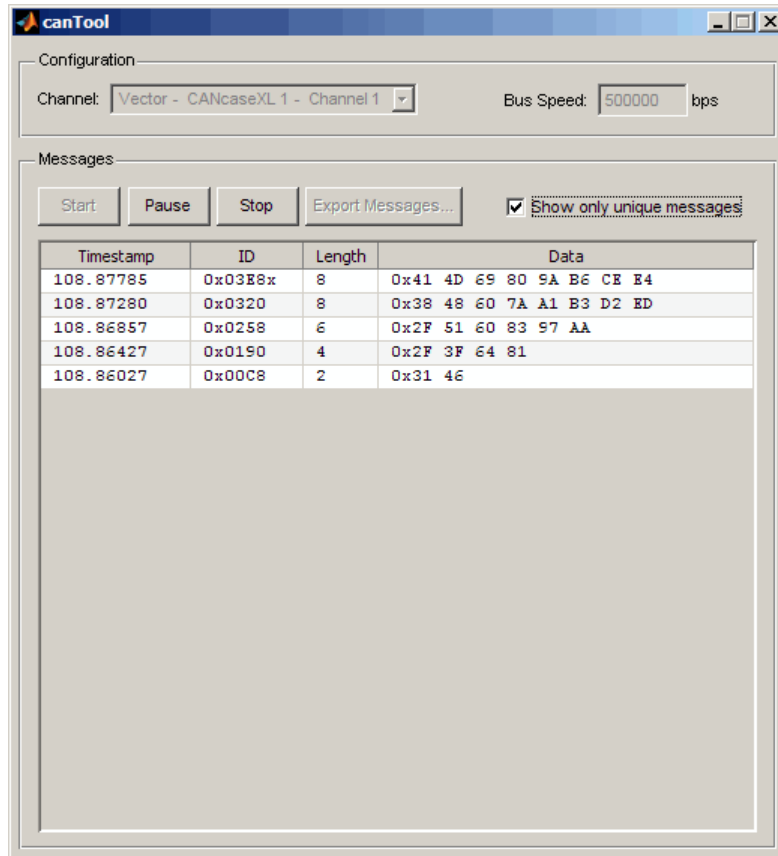
Saving the Message Log File

To save a log file of the messages currently displayed in the window, click **Export Messages**. The tool saves the messages in a MATLAB file in your current working directory.

Each time you export the messages to a file, the CAN Tool saves them as VNT CAN Log.mat with sequential numbering.

Viewing Unique Messages

To view the most recent instance of each unique message received on the channel, click **Show only unique messages**. In this view, you do not see messages scroll up, but each message refreshes its data and timestamp with each new instance.



Use this feature to get a snapshot of the IDs of messages that the selected channel receives. Use this information to analyze specific messages.

When you select the **Show only unique messages** check box, the tool continues to receive message actively. This simplified view allows you to focus on specific messages and analyze them.

To export messages when the **Show only unique messages** check box is selected, click **Pause** and then click **Export messages**. You cannot save the unique message list. This operation saves the complete message log in the window.

Using the Vehicle Network Toolbox Block Library

- “Overview” on page 4-2
- “Opening the Vehicle Network Toolbox Block Library” on page 4-3
- “Building Simulink Models to Transmit and Receive Messages” on page 4-5

Overview

This chapter describes how to use the Vehicle Network Toolbox block library. The library contains these blocks:

- **CAN Configuration** — Configure the settings of a CAN device.
- **CAN Pack** — Pack signals into a CAN message.
- **CAN Receive** — Receive CAN messages from a CAN bus.
- **CAN Transmit** — Transmit CAN messages to a CAN bus.
- **CAN Unpack** — Unpack signals from a CAN message.

The Vehicle Network Toolbox block library is a tool for simulating message traffic on a CAN network, as well for using the CAN bus to send and receive messages. You can use blocks from the block library with blocks from other Simulink libraries to create sophisticated models.

To use the Vehicle Network Toolbox block library, you require Simulink, a tool for simulating dynamic systems. Simulink is a model definition environment. Use Simulink blocks to create a block diagram that represents the computations of your system or application. Simulink is also a model simulation environment. Run the block diagram to see how your system behaves. If you are new to Simulink, read the *Simulink Getting Started Guide* in the Simulink documentation to understand its functionality better.

For more detailed information about the blocks in the Vehicle Network Toolbox block library, see Block Reference.

Opening the Vehicle Network Toolbox Block Library

In this section...

“Using the MATLAB Command Window” on page 4-3

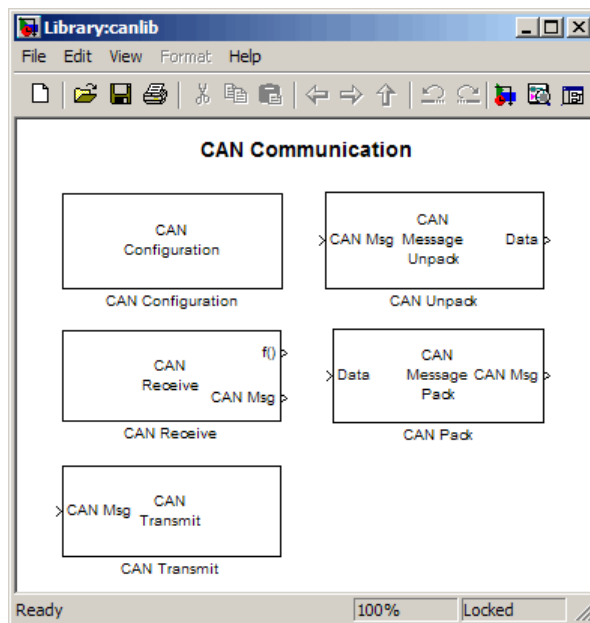
“Using the Simulink Library Browser” on page 4-4

Using the MATLAB Command Window

To open the Vehicle Network Toolbox block library, enter

```
canlib
```

in the MATLAB Command Window. MATLAB displays the contents of the library in a separate window.



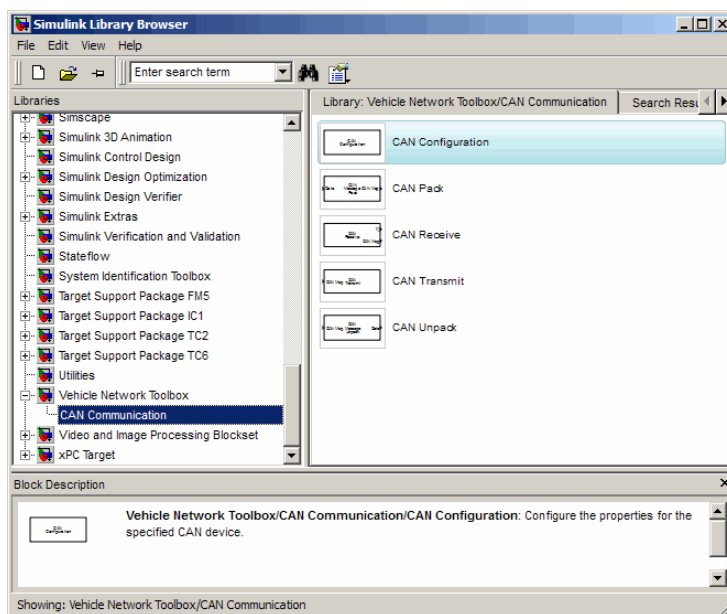
Using the Simulink Library Browser

To open the Vehicle Network Toolbox block library, start the Simulink Library Browser from MATLAB. Then select the library from the list of available block libraries displayed in the browser.

To start the Simulink Library Browser, enter

```
simulink
```

at the MATLAB Command Window. MATLAB opens the browser window. The left pane lists available block libraries, with the basic Simulink library listed first, followed by other libraries listed alphabetically under it. To open the Vehicle Network Toolbox block library, click its icon and select **CAN Communication** for the CAN blocks.



Simulink loads and displays the blocks in the library.

Building Simulink Models to Transmit and Receive Messages

In this section...

“Build a Message Transmit Model” on page 4-5

“Build a Message Receive Model” on page 4-10

“Save and Run the Model” on page 4-17

Build a Message Transmit Model

This section provides an example that builds a simple model using Vehicle Network Toolbox blocks with other blocks in the Simulink library. This example illustrates how to send data via a CAN network.

- Use virtual CAN channels to transmit messages.
- Use the CAN Configuration block to configure your CAN channels.
- Use the Constant block to send data to the CAN Pack block.
- Use the CAN Transmit block to send the data to the virtual CAN channel.

Use this section with “Build a Message Receive Model” on page 4-10 and “Save and Run the Model” on page 4-17 to build your complete model and run the simulation.

- “Step 1: Open the Block Library” on page 4-5
- “Step 2: Create a New Model” on page 4-6
- “Step 3: Drag Vehicle Network Toolbox Blocks into the Model” on page 4-7
- “Step 4: Drag Other Blocks to Complete the Model” on page 4-7
- “Step 5: Connect the Blocks” on page 4-8
- “Step 6: Specify the Block Parameter Values” on page 4-8

Step 1: Open the Block Library

To open the Vehicle Network Toolbox block library, start the Simulink Library Browser by entering:

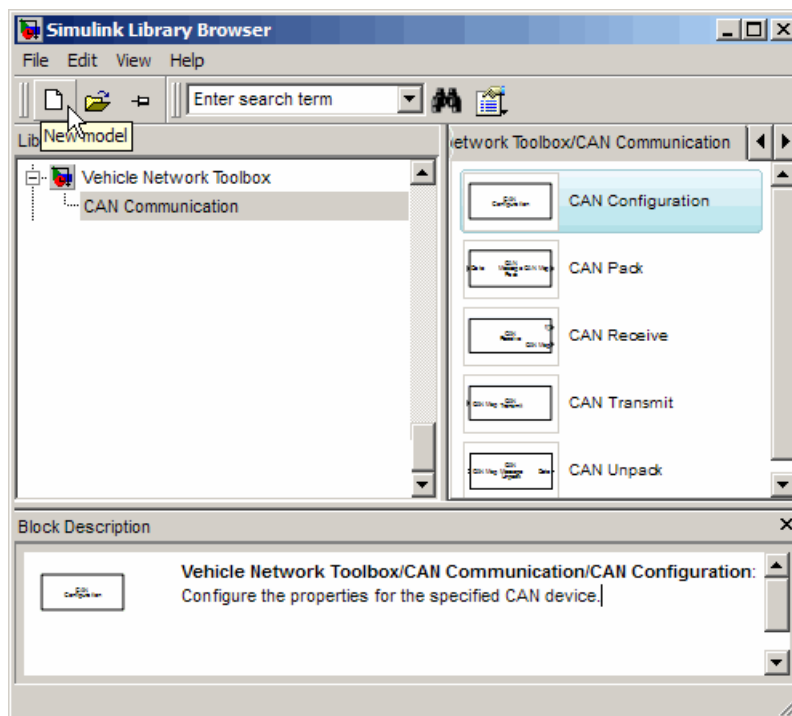
```
simulink
```

in the MATLAB Command Window. The left pane in the Simulink Library Browser lists the available block libraries. To open the Vehicle Network Toolbox block library, click its icon. Then click **CAN Communication** to open the CAN blocks. See Using the Simulink Library Browser for more information.

Step 2: Create a New Model

To use a block, add it to an existing model or create a model.

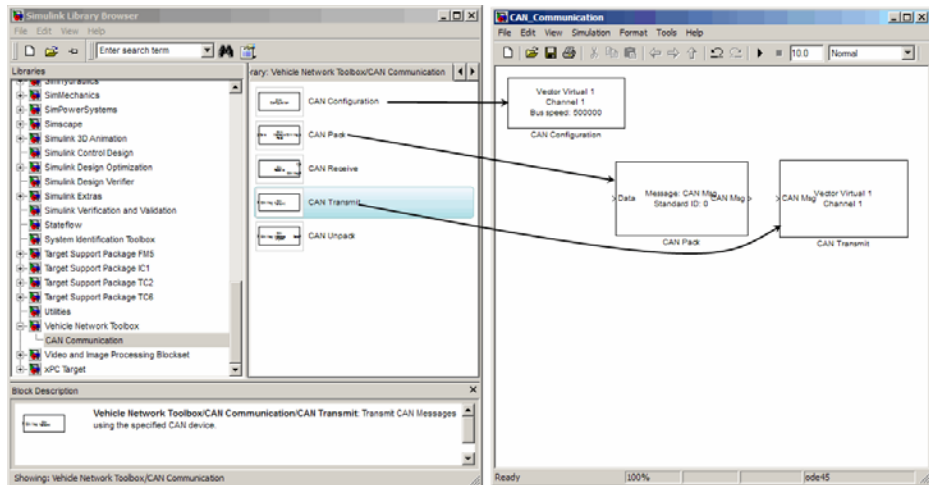
For this example, create a model by clicking the **New model** button on the toolbar.



You can also select **File > New > Model** from the Simulink Library Browser. Simulink opens an empty model window on the display. To name the new model, use the **Save** option.

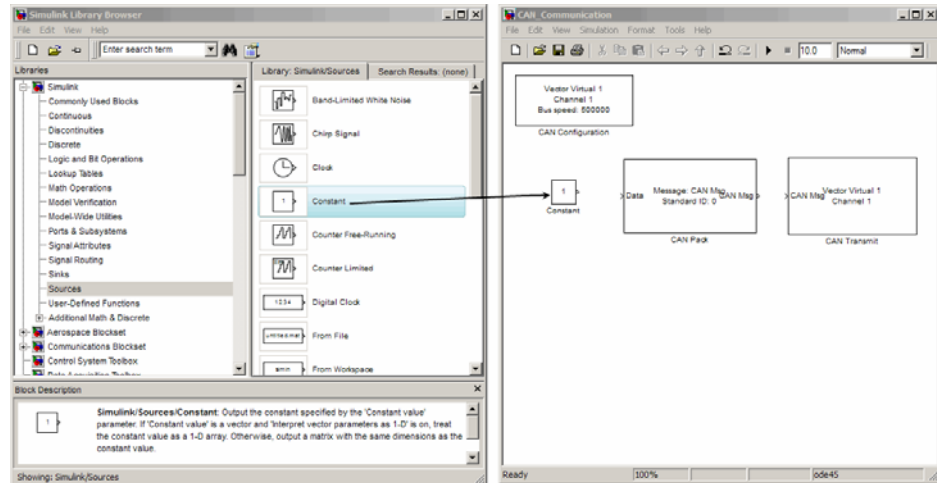
Step 3: Drag Vehicle Network Toolbox Blocks into the Model

To use the blocks in a model, click a block in the library and, holding the mouse button down, drag it into the model window. For this example, you need one instance each of the CAN Configuration, CAN Pack, and CAN Transmit blocks in your model.



Step 4: Drag Other Blocks to Complete the Model

This example requires a source block that feeds data to the CAN Pack block. Add a Constant block to your model.



Step 5: Connect the Blocks

Make a connection between the Constant block and the CAN Pack block. When you move the pointer near the output port of the Constant block, the pointer becomes a cross hair. Click the Constant block output port and, holding the mouse button, drag the pointer to the input port of the CAN Pack block. Then release the button.

In the same way, make a connection between the output port of the CAN Pack block and the input port of the CAN Transmit block.

The CAN Configuration block does not connect to any other block. This block configures the CAN channel used by the CAN Transmit block to transmit the packed message.

Step 6: Specify the Block Parameter Values

You set parameters for the blocks in your model by double-clicking the block.

Configure the CAN Configuration Block. Double-click the CAN Configuration block to open its parameters dialog box. Set the:

- **Device** to Vector Virtual 1 (Channel 1)
- **Bus speed** to 500000

- **Acknowledge Mode** to Normal

Click **OK**.

Configure the CAN Pack Block. Double-click the CAN Pack block to open its parameters dialog box. Set the:

- **Data is input as** to raw data
- **Name** to the default value CAN Msg
- **Identifier type** to the default Standard (11-bit identifier) type
- **Identifier** to 500
- **Length (bytes)** to the default length of 8

Click **OK**.

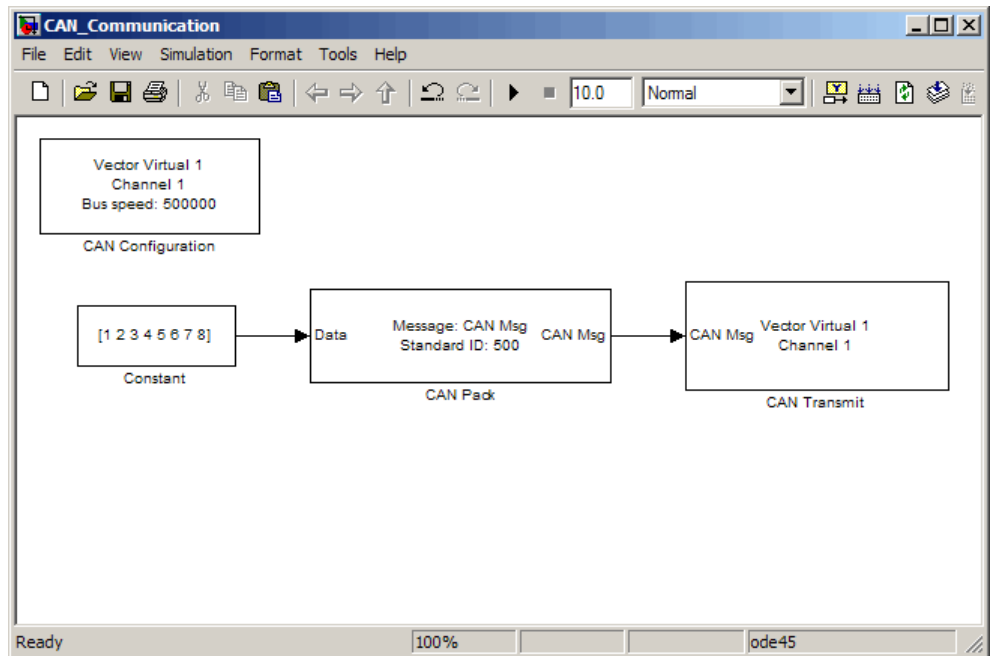
Configure the CAN Transmit Block. Double-click the CAN Transmit block to open its parameters dialog box. Set **Device** to Vector Virtual 1 (Channel 1). Click **Apply**, then **OK**.

Configure the Constant Block. Double-click the Constant block to open its parameters dialog box. On the **Main** tab, set the:

- **Constant value** to [1 2 3 4 5 6 7 8]
- **Sample time** to 0.01 seconds

On the **Signal Attributes** tab, set the **Output data type** to uint8. Click **OK**.

Your model looks like this figure.



Build a Message Receive Model

This section provides an example that builds a simple model using the Vehicle Network Toolbox blocks with other blocks in the Simulink library. This example illustrates how to receive data via a CAN network.

- Use a virtual CAN channel to receive messages.
- Use the CAN Configuration block to configure your virtual CAN channels.
- Use the CAN Receive block to receive the message sent by the blocks built in “Build a Message Transmit Model” on page 4-5.
- Use a Function–Call Subsystem block that contains the CAN Unpack block. This function takes in the data from the CAN Receive block and uses the parameters of the CAN Unpack block to unpack your message data.
- Use a Scope block to show the transfer of data visually.

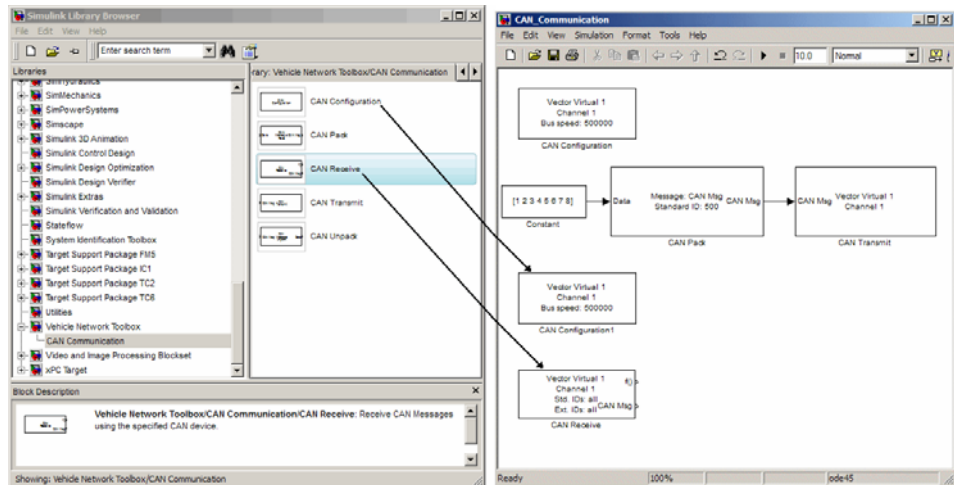
Use this section with “Build a Message Transmit Model” on page 4-5 and “Save and Run the Model” on page 4-17 to build your complete model and run the simulation.

- “Step 7: Drag Vehicle Network Toolbox Blocks into the Model” on page 4-11
- “Step 8: Drag Other Blocks to Complete the Model” on page 4-12
- “Step 9: Connect the Blocks” on page 4-14
- “Step 10: Specify the Block Parameter Values” on page 4-16

Step 7: Drag Vehicle Network Toolbox Blocks into the Model

For this example, you need one instance each of the CAN Configuration, CAN Receive, and CAN Unpack blocks in your model. However, you add only the CAN Configuration and the CAN Receive blocks here. Add the CAN Unpack block into the Function–Call Subsystem described in “Step 8: Drag Other Blocks to Complete the Model” on page 4-12.

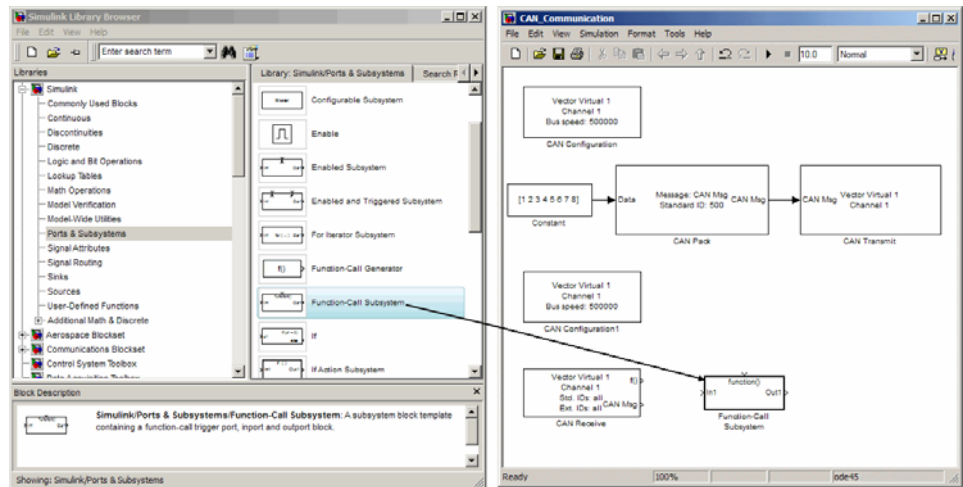
Tip Configure a separate CAN channel for the CAN Receive and CAN Transmit blocks.



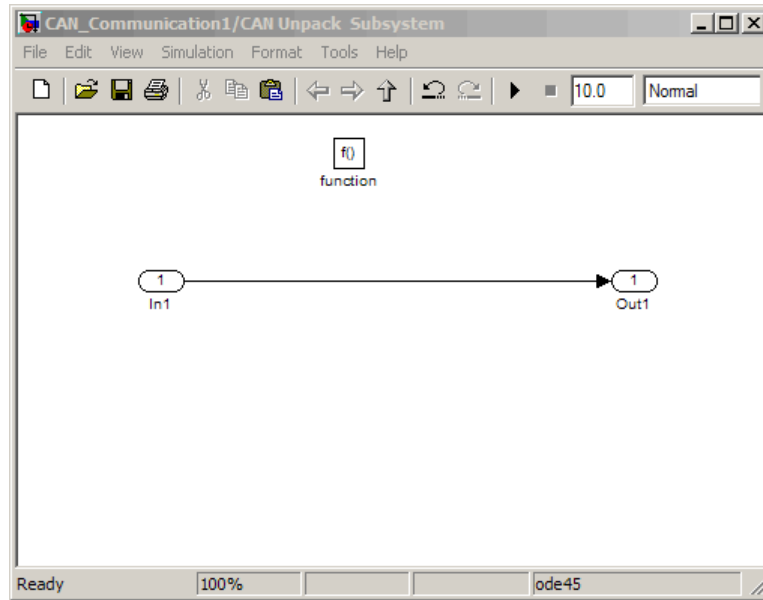
Step 8: Drag Other Blocks to Complete the Model

Use the Function–Call Subsystem block from the Simulink Ports & Subsystems block library to build your CAN Message pack subsystem.

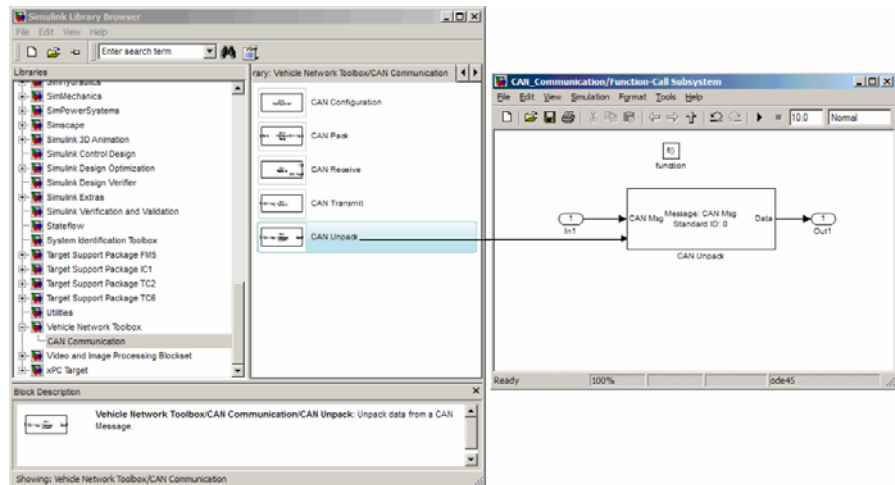
- 1 Drag the Function–Call Subsystem block into the model.



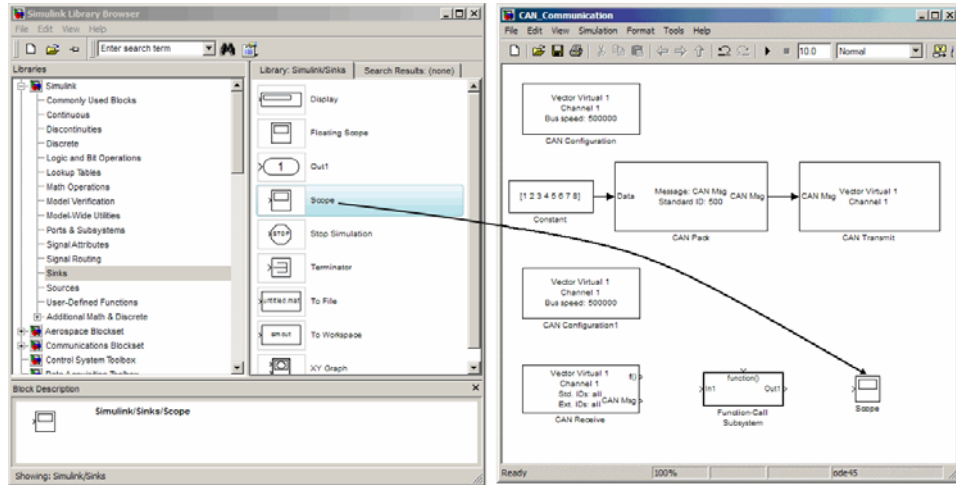
- 2 Double-click the Function–Call Subsystem block to open the subsystem model.



- Drop the CAN Unpack block from the Vehicle Network Toolbox block library in this subsystem.

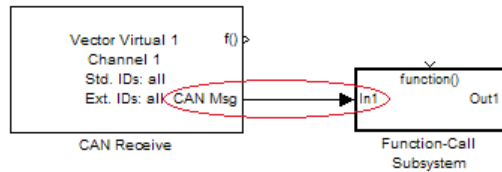


To see the results of the simulation visually, drag the Scope block from the Simulink block library into your model.

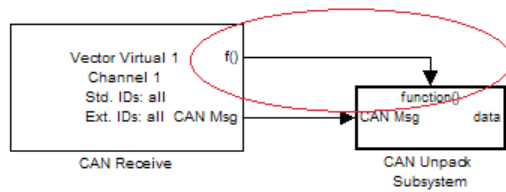


Step 9: Connect the Blocks

- 1 Connect the **CAN Msg** output port on the CAN Receive block to the **In1** input port on the Function–Call Subsystem block.

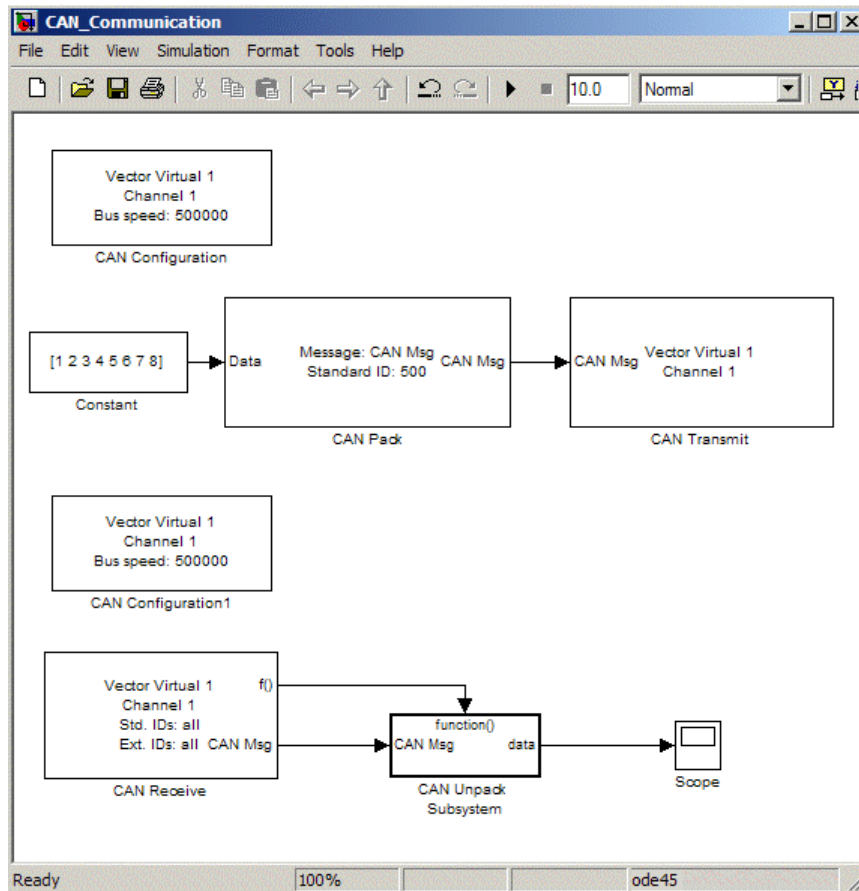


- 2 Open the Function–Call Subsystem block and:
 - Double-click **In1** to rename it to CAN Msg.
 - Double-click **Out1** to rename it to data.
- 3 Rename the Function–Call Subsystem block to CAN Unpack Subsystem.
- 4 Connect the **f()** output port on the CAN Receive block to the **function()** input port on the Function–Call Subsystem block.



5 Connect the CAN Unpack Subsystem output port to the input port on the Scope block.

Your model looks like this figure.



The CAN Configuration block does not connect to any other block. This block configures the CAN channel used by the CAN Receive block to receive the CAN message.

Step 10: Specify the Block Parameter Values

Set parameters for the blocks in your model by double-clicking the block.

Configure the CAN Configuration1 Block. Double-click the CAN Configuration block to open its parameters dialog box. Set the:

- **Device to Vector Virtual 1 (Channel 2)**
- **Bus speed** to 500000
- **Acknowledge Mode** to Normal

Click **OK**.

Configure the CAN Receive Block. Double-click the CAN Receive block to open its Parameters dialog box. Set the:

- **Device to Vector Virtual 1 (Channel 2)**
- **Sample time** to 0.01
- **Number of messages received at each timestep** to All

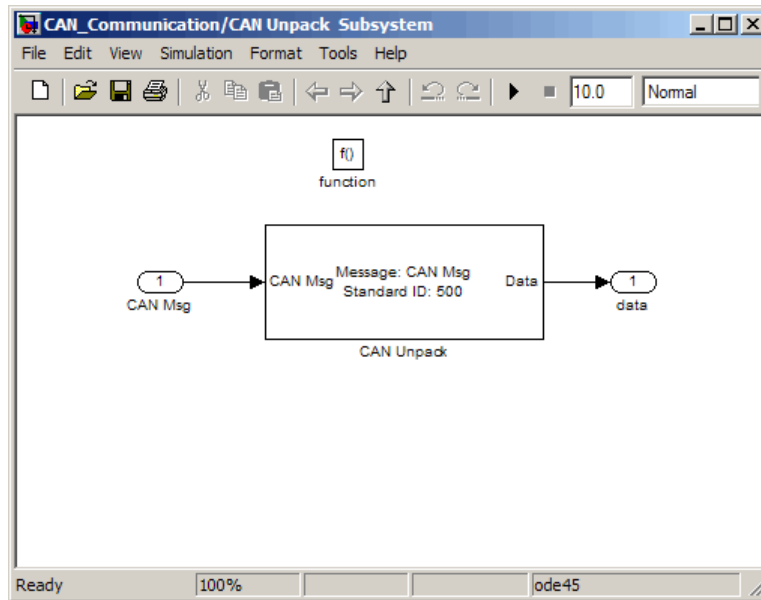
Click **OK**.

Configure the CAN Unpack Subsystem. Double-click the CAN Unpack subsystem to open the Function–Call Subsystem model. In the model, double-click the CAN Unpack block to open its parameters dialog box. Set the:

- **Data to be output as** to raw data
- **Name** to the default value CAN Msg
- **Identifier type** to the default Standard (11-bit identifier) type
- **Identifier** to 500
- **Length (bytes)** to the default length of 8

Click **OK**.

Your subsystem looks like this figure.



Save and Run the Model

This section shows you how to save the models you built, “Build a Message Transmit Model” on page 4-5 and “Build a Message Receive Model” on page 4-10.

- “Step 11: Save the Model” on page 4-17
- “Step 12: Change Configuration Parameters” on page 4-18
- “Step 13: Run the Simulation” on page 4-18
- “Step 14: View the Results” on page 4-19

Step 11: Save the Model

Before you run the simulation, save your model by clicking the **Save** icon or selecting **File > Save** from the menu bar.

Step 12: Change Configuration Parameters

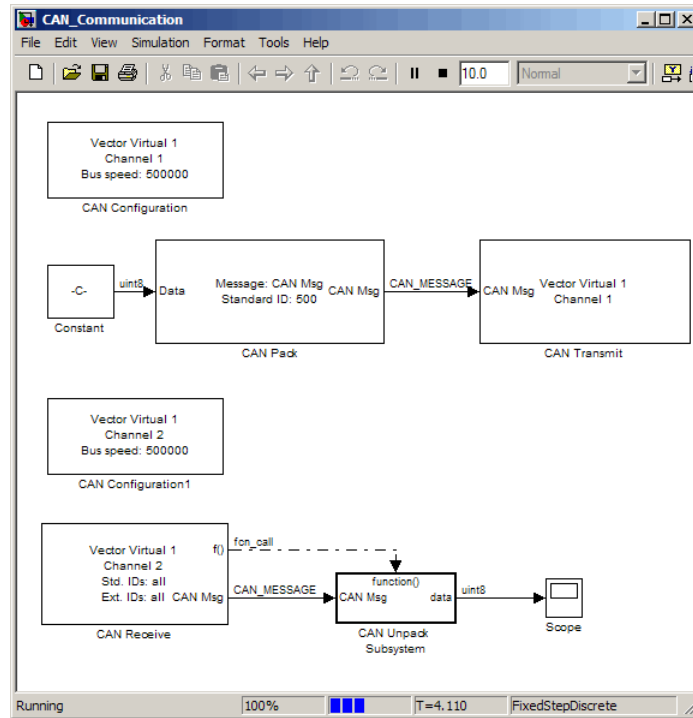
- 1 In your model window, select **Simulation > Configuration Parameters**. The Configuration Parameters dialog box opens.
- 2 In the Solver Options section, select:
 - **Fixed-step** from the **Type** list.
 - **Discrete (no continuous states)** from the **Solver** list.

Step 13: Run the Simulation

To run the simulation, click the **Start** button on the model window toolbar. Alternatively, you can use the **Simulation** menu in the model window and choose the **Start** option.

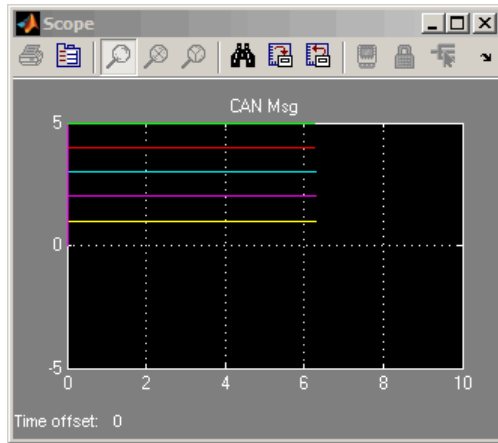
When you run the simulation, the CAN Transmit block gets the message from the CAN Pack block. It then transmits it via Virtual Channel 1. The CAN Receive block on Virtual Channel 2 receives this message and hands it to the CAN Unpack block to unpack the message.

While the simulation is running, the status bar at the bottom of the model window updates the progress of the simulation.

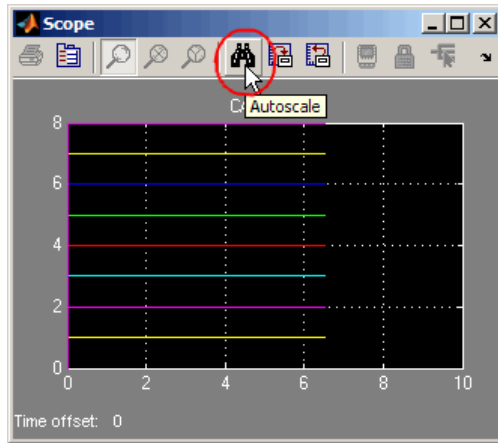


Step 14: View the Results

Double-click the Scope block to view the message transfer on a graph.



If you cannot see all the data on the graph, click the **Autoscale** toolbar button, which automatically scales both axes to display all stored simulation data.



In the graph, the horizontal axis represents the simulation time in seconds and the vertical axis represents the received data value. In the Message Transmit model, you configured blocks to pack and transmit an array of constant values, [1 2 3 4 5 6 7 8], every 0.01 second of simulation time. In the Message Receive model, these values are received and unpacked. The output in the Scope window represents the received data values.

Function Reference

CAN Channel Construction (p. 5-2)	Functions related to constructing CAN channel
CAN Channel Configuration (p. 5-3)	Functions related to configuring CAN channel
CAN Channel Execution (p. 5-4)	Functions related to executing configured CAN channel.
CAN Channel Status (p. 5-5)	Functions related to checking and modifying CAN channel status
CAN Database (p. 5-6)	Functions related to CAN database
CAN Message Handling (p. 5-7)	Functions related to working with CAN messages
CAN Message Filtering (p. 5-8)	Functions related to setting up CAN message filters
Information and Help (p. 5-9)	Functions related to displaying help information
Graphical Tools (p. 5-10)	Functions related to CAN Tools
Vector Informatik (p. 5-11)	Functions specifically related to Vector hardware functionality

CAN Channel Construction

canChannel

Construct CAN channel connected
to selected device

CAN Channel Configuration

get

Return property values

set

Configure property values

CAN Channel Execution

<code>receive</code>	Receive messages from CAN bus
<code>receiveRaw</code>	Receive raw messages from CAN bus
<code>replay</code>	Retransmit messages from CAN bus
<code>start</code>	Set CAN channel online
<code>stop</code>	Set CAN channel offline
<code>transmit</code>	Send CAN messages to CAN bus

CAN Channel Status

`configBusSpeed`

Set bit timing rate of CAN channel

CAN Database

canDatabase

Create handle to CAN database file

messageInfo

Information about CAN messages

signalInfo

Information about signals in CAN message

CAN Message Handling

<code>attachDatabase</code>	Attach CAN database to messages and remove CAN database from messages
<code>canMessage</code>	Build CAN message based on user-specified structure
<code>canMessageCompatibilityMode</code>	Enable dynamic signal property support
<code>extractAll</code>	Select all instances of message from array of messages
<code>extractRecent</code>	Select most recent message from array of messages
<code>extractTime</code>	Select messages occurring within specified time range from array of messages
<code>pack</code>	Pack signal data into CAN message
<code>unpack</code>	Unpack signal data from message

CAN Message Filtering

`filterReset`

Open CAN message acceptance filters

`filterSet`

Set specific CAN message acceptance filter configuration

Information and Help

canHWInfo

Information on available CAN devices

canSupport

Generate technical support log

Graphical Tools

canTool

Open CAN Tool

Vector Informatik

`filterAcceptRange`

Set range of CAN identifiers to pass acceptance filter

`filterBlockRange`

Set range of CAN identifiers to block via acceptance filter

Functions — Alphabetical List

attachDatabase

Purpose Attach CAN database to messages and remove CAN database from messages

Syntax `attachDatabase (message, database)`
`attachDatabase (message, [])`

Arguments

<code>message</code>	The name of the CAN message that you want to attach the database to or remove the database from.
<code>database</code>	The name of the database (.dbc file) that you want to attach to the message or remove from the message.

Description `attachDatabase (message, database)` attaches the specified database to the specified message. You can then use signal-based interaction with the message data, interpreting the message in its physical form.

`attachDatabase (message, [])` removes any attached database from the specified message. You can then interpret messages in their raw form.

Remarks If the specified message is an array, then the database attaches itself to each entry in the array. The database attaches itself to the message even if the message you specified does not exist in the database. The message then appears and operates like a raw message. To attach the database to the CAN channel directly, edit the Database property of the channel object.

Examples

```
canDb = canDatabase('C:\Database.dbc')
message = receive(canch, Inf)
attachDatabase(message, canDb)
```

See Also `canDatabase`, `receive`

Purpose Construct CAN channel connected to selected device

Syntax `canch = canChannel('vendor', 'device', devicechannelindex)`

Arguments	<code>vendor</code>	The name of the CAN device vendor. Specify the vendor name as a string.
	<code>device</code>	The CAN interface that you want to connect to.
	<code>devicechannelindex</code>	A numeric channel on the specified device.
	<code>canch</code>	The CAN channel object the you create.

Description `canch = canChannel('vendor', 'device', devicechannelindex)` returns a CAN channel connected to a device from a specified vendor. For Vector products, `device` is a combination of the device type and a device index, such as 'CANCseXL 1'. For example, if there are two CANcardXL devices, `device` can be 'CANcardXL 1' or 'CANcardXL 2'. Use `canHWInfo` to obtain a list of available devices.

Remarks The Vehicle Network Toolbox currently supports Vector and Kvaser devices. Refer to the Supported Hardware topic.

canChannel

Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
canch = canChannel('Vector','Virtual 1',2)
```

Notes You cannot use the same variable to create multiple channels sequentially. Clear any channel in use before using the same variable to construct a new CAN channel.

You cannot create arrays of CAN channel objects. Each object you create must exist as its own individual variable.

See Also

canHWInfo

Purpose Create handle to CAN database file

Syntax `candb = canDatabase('dbfile.dbc')`

Description `candb = canDatabase('dbfile.dbc')` creates a handle to the specified database file *dbfile.dbc*. You can specify just a file name, a full path, or a relative path. MATLAB looks for *dbfile.dbc* on the MATLAB path. Vehicle Network Toolbox supports the Vector CAN database (.dbc) files.

Examples `candb = canDatabase('C:\Database.dbc')`

See Also `canMessage`

canHWInfo

Purpose Information on available CAN devices

Syntax `out = canHWInfo()`

Description `out = canHWInfo()` returns information about CAN devices and displays the information on a per vendor and channel basis. Use `get` on the output of `canHWInfo` to obtain more detailed results.

Examples

```
info = canHWInfo()
get(info)
    ToolboxName: 'Vehicle Network Toolbox'
    ToolboxVersion: '1.0 (R2009a)'
    MATLABVersion: '7.8 (R2009a)'
    VendorInfo: [1x1 can.vector.VendorInfo]
```

See Also `canChannel`

Purpose Build CAN message based on user-specified structure

Syntax
`message = canMessage(id, extended, datalength)`
`message = canMessage(database, messagename)`

Arguments

<code>id</code>	The ID of the message that you specify.
<code>extended</code>	Indicates whether the message ID is of standard or extended type. The Boolean value is <code>true</code> if extended or <code>false</code> if standard.
<code>datalength</code>	The length of the data of the message, in bytes. Specify from 0 through 8.
<code>database</code>	Handle to the CAN database containing the message definition.
<code>messagename</code>	The name of the message definition in the database.
<code>message</code>	The message object returned from the function.

Description

`message = canMessage(id, extended, datalength)` creates and returns a CAN message object, from the raw message information.

`message = canMessage(database, messagename)` constructs a message using the message definition of the specified message, in the specified database.

Examples To construct a CAN message, type:

```
message = canMessage(2500, true, 4)
```

To construct a message using CAN database message definitions, create a database object using the `canDatabase` function, and then construct your message:

```
candb = ('c:\database.dbc')  
message = canMessage (candb, 'messagename')
```

canMessage

See Also

attachDatabase, canDatabase, extractAll, extractRecent,
extractTime, pack, unpack

canMessageCompatibilityMode

Purpose Enable dynamic signal property support

Note `canMessageCompatibilityMode` will be removed in a future version.

Syntax `canMessageCompatibilityMode(mode)`
`mode = canMessageCompatibilityMode()`

Arguments

<code>mode</code>	Indicates if the compatibility mode is on (allows access to dynamic signals) or off (access signals via the <code>Signals</code> property). The Boolean value for <code>mode</code> is <code>true</code> if compatibility mode is on and <code>false</code> if it is off. By default the mode is <code>false</code> .
-------------------	---

Description `canMessageCompatibilityMode(mode)` restores or removes access to dynamic signal properties for CAN messages based on your input. If you cannot update your code to use the new `Signals` property at this time, enter `true` as your `mode` value. It is recommended that you update your code and use the new `Signals` property, which provides a significant performance increase over the use of dynamic properties.

`mode = canMessageCompatibilityMode()` queries the current state of the CAN message compatibility mode.

The compatibility mode is `false` by default. Change the mode each time you start a new MATLAB session. Set the mode value to `true` to turn compatibility mode on and `false` to turn it off.

Note If `mode` is 0 (`false`), then you cannot access dynamic signals and must use the `Signals` property.

canMessageCompatibilityMode

Examples

```
% To query current state of CAN message compatibility mode:  
mode = canMessageCompatibilityMode()
```

```
% To enable dynamic signal property support:  
canMessageCompatibilityMode(true)
```

See Also

Signals

Purpose Generate technical support log

Syntax `canSupport()`

Description `canSupport()` returns diagnostic information for all installed CAN devices and saves output to the text file `canSupport.txt` in the current working directory.

For online support of Vehicle Network Toolbox software, visit the toolbox page on the MathWorks Web site.

canTool

Purpose Open CAN Tool

Syntax `canTool`

Description `canTool` starts the CAN Tool, which displays live CAN message traffic. Use the CAN Tool to view message traffic using a selected CAN device and channel. You can also export messages to a log file via this tool.

For more information about this tool, refer to Chapter 3, “Monitoring CAN Message Traffic”.

Purpose

Set bit timing rate of CAN channel

Syntax

```
configBusSpeed(canch, busspeed)
configBusSpeed(canch, busspeed, sjw, tseg1, tseg2,
    numberofsamples)
```

Arguments

<code>canch</code>	The CAN channel object that you want to set the bit timing rate for.
<code>busspeed</code>	The user-specified bit timing rate for the specified object.
<code>sjw</code>	The synchronization jump width. This value is the maximum value of time bit adjustments.
<code>tseg1</code>	The length of time at the start of the sample point within a bit time.
<code>tseg2</code>	The length of time at the end of the sample point within a bit time.
<code>numberofsamples</code>	The specified count of bit samples used.

Description

`configBusSpeed(canch, busspeed)` sets the speed of the CAN channel in a direct form that uses baseline bit timing calculation factors.

`configBusSpeed(canch, busspeed, sjw, tseg1, tseg2, numberofsamples)` sets the speed of the CAN channel `canch` to `busspeed` using the specified bit timing calculation factors to control the timing in an advanced form.

Remarks

Unless you have specific timing requirements for your CAN connection, use the direct form of `configBusSpeed`. Also note that you can set the bus speed only when the CAN channel is offline. The channel must also have initialization access to the CAN device.

Synchronize all nodes on the network for CAN to work successfully. However, over time, clocks on different nodes will get out of sync, and must resynchronize. SJW specifies the maximum width (in time) that

configBusSpeed

you can add to `tseg1` (in a slower transmitter), or subtract from `tseg2` (in a faster transmitter) to regain synchronization during the receipt of a CAN message.

Examples

To configure the bus speed using baseline bit timing calculation, type:

```
canch = canChannel('Vector','CANCaseXL 1',1)
configBusSpeed(canch,250000)
```

To specify the bit timing calculations, type:

```
canch = canChannel('Kvaser','USBcan Professional 1', 1)
configBusSpeed(channel, 500000, 1, 4, 3, 1)
```

See Also

`canChannel`

Purpose

Select all instances of message from array of messages

Syntax

```
[extracted, remainder] = extractAll(message, messagename)  
[extracted, remainder] = extractAll(message, id, extended)
```

Arguments

message	An array of CAN message objects that you specify to parse and find the specified messages by name or ID.
messagename	The name of the message that you specify to extract.
id	The ID of the message that you specify to extract.
extended	Indicates whether the message ID is a standard or extended type. The Boolean value is <code>true</code> if extended and <code>false</code> if standard.
extracted	An array of CAN message objects returned with all instances of <code>id</code> found in the message.
remainder	A CAN message object containing all messages in the original input message with all instances of <code>id</code> removed.

Description

`[extracted, remainder] = extractAll(message, messagename)` parses the given array `message`, and returns all instances of messages matching the specified message name.

`[extracted, remainder] = extractAll(message, id, extended)` parses the given array `message`, and returns all instances of messages matching the specified ID with the specified standard or extended type.

Remarks

You can specify `id` as a cell array of message names or a vector of identifiers. For example, if you pass `id` in as `[250 5000]`, `[false true]`, `extractAll` returns every instance of both CAN message 250 and message 5000 that it finds in the message array. If any `id` in the vector

extractAll

is an extended type, set `extended` to `true` and as a vector of the same length as `id`.

Examples

```
[msgOut, remainder]=extractAll(message, 'msg1')  
[msgOut, remainder]=extractAll(message,{'msg1' 'msg2' 'msg3'})  
[msgOut, remainder]=extractAll(message, 3000, true)  
[msgOut, remainder]=extractAll(message,[200 5000],[false true])
```

See Also

`extractRecent`, `extractTime`

Purpose Select most recent message from array of messages

Syntax

```
extracted = extractRecent(message)
extracted = extractRecent(message, messagename)
extracted = extractRecent(message, id, extended)
```

Arguments

message	An array of CAN message objects that you specify to parse and find the specified messages by name or ID.
messagename	The name of the message that you specify to extract.
id	The id of the message that you specify to extract.
extended	Indicates whether the message ID is a standard or extended type. The Boolean value is true if extended and false if standard.
extracted	An array of CAN message objects returned with the most recent instance of id found in the message.

Description

`extracted = extractRecent(message)` parses the given array message and returns the most recent instance of each unique CAN message found in the array.

`extracted = extractRecent(message, messagename)` parses the specified array of messages and returns the most recent instance matching the specified message name.

`extracted = extractRecent(message, id, extended)` parses the given array message and returns the most recent instance of the message matching the specified ID with the specified standard or extended type.

Remarks

You can specify id as a vector of identifiers. For example, if you pass id in as [250 500], extractRecent returns the latest instance of both CAN message 250 and message 500 if it finds them in the message array. By default, all identifiers in the vector are standard CAN message identifiers unless extended is true. If any id in the vector is

extractRecent

an extended type, then `extended` is `true` and is a vector of the same length as `id`.

Examples

```
msgOut = extractRecent(message)
msgOut = extractRecent(message, 'msg1')
msgOut = extractRecent(message, {'msg1' 'msg2' 'msg3'})
msgOut = extractRecent(message, 3000, true)
msgOut = extractRecent(message, [400, 5000], [false true])
```

See Also

`extractAll`, `extractTime`

Purpose Select messages occurring within specified time range from array of messages

Syntax `extracted = extractTime(message, starttime, endtime, msgRange)`

Arguments

<code>message</code>	An array of CAN message objects.
<code>starttime</code>	The beginning of the time range in seconds that you specify. Returns messages with a timestamp greater than or equal to the specified start time.
<code>endtime</code>	The end of the time range in seconds that you specify. Parses messages with a timestamp up to the specified end time, including the specified end time.
<code>extracted</code>	An array of CAN message objects returned with all messages that occur within and including <code>starttime</code> and <code>endtime</code> .

Description `extracted = extractTime(message, starttime, endtime, msgRange)` parses the array `message` and returns all messages with a timestamp within the specified `starttime` and `endtime`, including the `starttime` and `endtime`.

Remarks Specify the time range in increasing order from `starttime` to `endtime`. If you must specify the largest available time, `endtime` also accepts `Inf` as a valid value. The earliest acceptable time you can specify for `starttime` is 0.

Examples

```
msgRange = extractTime(message, 5, 10.5)
msgRange = extractTime(message, 0, 60)
msgRange = extractTime(message, 150, Inf)
```

See Also `extractAll`, `extractRecent`

filterAcceptRange

Purpose Set range of CAN identifiers to pass acceptance filter

Syntax `filterAcceptRange(canch, rangestart, rangeend)`

Arguments

<code>canch</code>	The CAN channel that you want to set the filter for.
<code>rangestart</code>	The first identifier of the range of message IDs that the filter accepts.
<code>rangeend</code>	The last identifier of the range of message IDs that the filter accepts.

Description `filterAcceptRange(canch, rangestart, rangeend)` sets the acceptance filter for standard identifier CAN messages. It allows messages within the given range on the CAN channel `canch` to pass. `rangestart` and `rangeend` establish the beginning and end of the acceptable range. You can use this function with Vector devices only.

Notes

- You can configure message filtering only when the CAN channel is offline.
 - CAN message filters initialize to fully open.
 - `filterReset` makes the acceptance filters fully open.
 - `filterAcceptRange` supports only standard (11-bit) CAN identifiers.
 - You must set the values from `rangestart` through `rangeend` in increasing order.
 - `filterAcceptRange` and `filterBlockRange` work together by allowing and blocking ranges of CAN messages within a single filter. You can perform both operations multiple times in sequence to custom configure the filter as desired.
-

Remarks

When you call `filterAcceptRange` on an open or reset filter, it automatically blocks the entire standard CAN identifier range, allowing only the desired range to pass. Subsequent calls to `filterAcceptRange` open additional ranges on the filter without blocking the ranges previously allowed.

Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
filterAcceptRange(canch,600,625)
filterAcceptRange(canch,705,710)
```

See Also

`filterBlockRange`, `filterReset`, `filterSet`

filterBlockRange

Purpose Set range of CAN identifiers to block via acceptance filter

Syntax `filterBlockRange(canch, rangestart, rangeend)`

Arguments

<code>canch</code>	The CAN channel that you want to set the filter for.
<code>rangestart</code>	The first identifier of the range of message IDs that the filter starts blocking at.
<code>rangeend</code>	The last identifier of the range of message IDs that the filter stops blocking at.

Description `filterBlockRange(canch, rangestart, rangeend)` blocks messages within a given range by setting an acceptance filter. You can use this function with Vector devices only.

Notes

- You can configure message filtering only when the CAN channel is offline.
 - CAN message filters initialize to fully open.
 - `filterReset` makes the acceptance filters fully open.
 - `filterBlockRange` supports only standard (11-bit) CAN identifiers.
 - You must set the values from `rangestart` through `rangeend` in increasing order.
 - `filterBlockRange` and `filterAcceptRange` work together by blocking and allowing ranges of CAN messages within a single filter. You can perform both operations multiple times in sequence to custom configure the filter as desired.
-

Examples

You can set the filter to block or accept messages within a specific range.

```
canch = canChannel('Vector','CANCaseXL 1',1)
filterBlockRange(canch, 500, 750)
filterAcceptRange(canch,600,625)
filterAcceptRange(canch,705,710)
filterBlockRange(canch,1075,1080)
```

See Also

[filterAcceptRange](#), [filterReset](#), [filterSet](#)

filterReset

Purpose Open CAN message acceptance filters

Syntax `filterReset(canch)`

Description `filterReset(canch)` resets the CAN message filters on the CAN channel `canch` for both standard and extended CAN identifier types. Then all messages of all identifier types can pass.

This function does not work if the channel is online. Make sure that the channel is offline before calling `filterReset`.

Examples Reset the message filters as shown:

```
canch = canChannel('Vector','CANCaseXL 1',1)
filterBlockRange(canch, 500, 750)
filterAcceptRange(canch,600,625)
filterAcceptRange(canch,705,710)
filterBlockRange(canch,1075,1080)
filterSet(canch, 500, 750, 'Standard')
filterReset(canch)
```

See Also `filterAcceptRange`, `filterBlockRange`, `filterSet`

Purpose Set specific CAN message acceptance filter configuration

Syntax `filterSet(canch, code, mask, idtype)`
`filterSet(canch, id, idtype)`

Arguments

<code>canch</code>	The CAN channel that you want to set the filter for.
<code>code</code>	The value required for each bit position of the identifier.
<code>mask</code>	The bits in the identifier that are relevant to the filter.
<code>id</code>	Set a filter on the CAN message with the <code>id</code> , range of <code>ids</code> , multiple ranges of <code>ids</code> , or a combination of <code>ids</code> .
<code>idtype</code>	A string specifying either a standard or an extended CAN message <code>id</code> type.

Description `filterSet(canch, code, mask, idtype)` sets the CAN message acceptance filter to the specified `code` and `mask`. You also must specify the CAN identifier type `idtype` on the CAN channel `canch`.

`filterSet(canch, id, idtype)` sets the CAN message acceptance filter by determining the best possible `code` and `mask` based on the `ID` and identifier type specified in the input argument.

Notes

- You can configure message filtering only when the CAN channel is offline.
 - CAN message filters initialize to fully open.
 - Use `filterReset` to make the acceptance filters fully open.
 - `filterSet` supports either standard or extended CAN identifiers.
-

filterSet

Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
filterSet(canch,500,750, 'Standard')
filterSet(canch,2500,3000, 'Extended')
```

To let the Vehicle Network Toolbox determine the best possible code and mask option:

```
canch = canChannel('Kvaser', 'USBcan Professional 1', 1)
filterSet(canch, [500:502 1000], 'Standard')
filterSet(canch, [7500:8000 12000], 'Extended')
```

See Also

`filterAcceptRange`, `filterBlockRange`, `filterReset`

Purpose

Return property values

Syntax

```
out = get (obj)
```

Description

`out = get (obj)` returns the structure `out`, where each field name is the name of a property of the specified object and each field contains the value of that property.

Examples

Configure a CAN channel:

```
canch = canChannel('Vector','CANCaseXL 1',1)
```

Call `get` on the CAN channel object to obtain the properties of the configured CAN channel:

```
get (canch)
```

Configure a CAN message:

```
message = canMessage(250, true, 8)
```

Call `get` on the message object to obtain the properties of the configured message:

```
get (message)
```

Configure a CAN database:

```
candb = canDatabase('C:\Database.dbc')
```

call `get` on the database to obtain the properties of the configured database:

```
get (candb)
```

messageInfo

Purpose Information about CAN messages

Syntax

```
msgInfo = messageInfo(candb)
msgInfo = messageInfo(candb, 'msgName')
msgInfo = messageInfo(candb, id, extended)
```

Arguments

<code>candb</code>	The database containing the CAN messages that you want information about.
<code>msgName</code>	The name of the message you want information about.
<code>id</code>	The numeric identifier of the specified message.
<code>extended</code>	Indicates whether the message ID is in standard or extended type. The Boolean value is true if extended and false if standard.

Description

`msgInfo = messageInfo(candb)` returns information about CAN messages in the specified database `candb`.

`msgInfo = messageInfo(candb, 'msgName')` returns information about the specified message 'msgName' in the specified database `candb`.

`msgInfo = messageInfo(candb, id, extended)` returns information about the message with the specified standard or extended ID in the specified database `candb`.

Examples

```
candb = canDatabase('c:\Database.dbc')
msgInfo = messageInfo(candb)
msgInfo = messageInfo(candb, 'msgName')
msgInfo = messageInfo(candb, 500, false)
```

See Also `canDatabase`, `canMessage`, `signalInfo`

Purpose

Pack signal data into CAN message

Syntax

```
pack(message, value, startbit, signalsize, byteorder)
```

Arguments

message	The CAN message structure that you specify for the signal to be packed in.
value	The value of the signal you specify to be packed in the message.
startbit	The signal's starting bit in the data. This is the least significant bit position in the signal data. Accepted values for startbit are from 0 through 63.
signalsize	The length of the signal in bits. Accepted values for signalsize are from 1 through 64.
byteorder	The signal byte order format. Accepted values are 'LittleEndian' and 'BigEndian'.

Description

`pack(message, value, startbit, signalsize, byteorder)` takes specified input parameters and packs them into the message.

Examples

```
pack(message, 25, 0, 16, 'LittleEndian')
```

See Also

`canMessage`, `extractAll`, `extractRecent`, `extractTime`, `unpack`

receive

Purpose Receive messages from CAN bus

Syntax `message = receive(canch, messagesrequested)`

Arguments

<code>canch</code>	The CAN channel from which to receive the message.
<code>messagesrequested</code>	The maximum count of messages to receive. The specified value must be a nonzero and positive, or <code>Inf</code> .
<code>message</code>	An array of CAN message objects received from the channel.

Description `message = receive(canch, messagesrequested)` returns an array of CAN message objects received on the CAN channel `canch`. The number of messages returned is less than or equal to `messagesrequested`. If fewer messages are available than `messagesrequested` specifies, the function returns the currently available messages. If no messages are available, the function returns an empty array. If `messagesrequested` is infinite, the function returns all available messages.

To understand the elements of a message, refer to `canMessage`.

Examples

```
canch = canChannel('Vector', 'CANCaseXL 1', 1)
start(canch)
message = receive(canch, 5)
```

To receive all messages, type:

```
message = receive(canch, Inf)
```

See Also `canChannel`, `canMessage`, `transmit`

Purpose Receive raw messages from CAN bus

Note `receiveRaw` will be removed in a future version. Use `receive` instead.

Syntax `msgStructs = receiveRaw(canch, messagesrequested)`

Arguments

<code>canch</code>	The CAN channel from which to receive the message.
<code>messagesrequested</code>	The maximum count of messages to receive. The specified value must be nonzero and positive, or <code>Inf</code> .
<code>msgStructs</code>	An array of message structures received from the CAN channel.

Description `msgStructs = receiveRaw(canch, messagesrequested)` returns an array of CAN message structures received on the CAN channel `canch`. The number of messages returned is less than or equal to `messagesrequested`. If fewer messages are available than `messagesrequested` specifies, the function returns the currently available messages. If no messages are available, the function returns an empty array. If `messagesrequested` is infinite, the function returns all available messages.

To understand the elements of a message, refer to `canMessage`.

Examples Assuming that you have messages on a channel and an attached database, you can receive a raw message, convert it to an object and apply database definitions by typing:

```
canch = canChannel('Vector', 'CANcaseXL 1', 1)
start(canch)
msgStructs = receiveRaw(canch, 5)
```

receiveRaw

```
message = canMessage(msgStructs)
attachDatabase(message, canDatabase('Database.dbc'))
```

Note This example is not an exact workflow.

To receive all messages in the raw structure, type:

```
message = receiveRaw(canch, Inf)
```

Note Receive raw messages when you are concerned about performance issues.

See Also

`canChannel`, `canMessage`, `receive`, `transmit`

Purpose Retransmit messages from CAN bus

Syntax `replay(canch, message)`

Arguments

<code>canch</code>	The CAN channel that you specify to transmit the messages.
<code>message</code>	An array of message objects to replay.

Description `replay(canch, message)` retransmits the message or messages `message` on the channel `canch`, based on the relative differences of their timestamps. The replay function also replays messages from MATLAB to Simulink

To understand the elements of a message, refer to `canMessage`.

Examples This example uses a loopback connection between two channels where:

- The first channel transmits messages 2 seconds apart.
- The second channel receives them.
- The replay function retransmits the messages with the original delay.

```
ch1 = canChannel('Vector', 'CANcaseXL 1', 1)
ch2 = canChannel('Vector', 'CANcaseXL 1', 2)
start(ch1)
start(ch2)
msgTx1 = canMessage(500, false, 8)
msgTx2 = canMessage(750, false, 8)
%The first channel transmits messages 2 seconds apart
transmit(ch1, msgTx1)
pause(2)
transmit(ch1, msgTx2)
%The second channel receives them
msgRx1 = receive(ch2, Inf)
```

replay

```
%The replay function retransmits the messages with the original delay.  
replay(ch2, msgRx1)  
pause(2)  
msgRx2 = receive(ch1, Inf)
```

The timestamp differentials between messages in the two receive arrays, `msgRx1` and `msgRx2`, are equal.

See Also

`canChannel`, `canMessage`, `receive`, `transmit`

Purpose	Configure property values
Syntax	<code>set (obj, propertyname, propertyvalue)</code>
Description	<code>set (obj, propertyname, propertyvalue)</code> configures the specified property, <code>propertyname</code> , on the object <code>obj</code> , to the value specified in <code>propertyvalue</code> .
Examples	<p>To set a CAN channel property:</p> <pre>canch = canChannel('Vector', 'CANcaseXL 1', 1) set (canch, 'SilentMode', true)</pre> <p>To set a CAN message property:</p> <pre>message = canMessage(250, 8, true) set (message, 'Remote', true)</pre> <p>To set a CAN message signal property:</p> <pre>candb = canDatabase('C:\Database.dbc') message = canMessage(candb, 'Battery_Voltage') set (message, 'BatVlt', 9.3)</pre>

signalInfo

Purpose Information about signals in CAN message

Syntax

```
SigInfo = signalInfo(candb, 'msgName')  
SigInfo = signalInfo(candb, id, extended)  
SigInfo = signalInfo(candb, id, extended, 'signalName')
```

Arguments

<code>candb</code>	The database containing the signals that you want information about.
<code>msgName</code>	The name of the message that contains the signals that you want information about.
<code>id</code>	The numeric identifier of the specified message that contains the signals you want information about.
<code>extended</code>	Indicates whether the message ID is in standard or extended type. The Boolean value is <code>true</code> if extended and <code>false</code> if standard.
<code>signalName</code>	The name of the specific signal that you want information about.
<code>SigInfo</code>	The signal information object returned from the function.

Description `SigInfo = signalInfo(candb, 'msgName')` returns information about the signals in the specified CAN message `msgName`, in the specified database `candb`.

`SigInfo = signalInfo(candb, id, extended)` returns information about the signals in the message with the specified standard or extended ID `id`, in the specified database `candb`.

`SigInfo = signalInfo(candb, id, extended, 'signalName')` returns information about the specified signal `'signalName'` in the message with the specified standard or extended ID `id`, in the specified database `candb`.

Examples

```
SigInfo=signalInfo(candb, 'Battery_Voltage')  
SigInfo=signalInfo(candb, 'Battery_Voltage', 196608, true)
```



```
SigInfo=signalInfo  
(candb, 'Battery_Voltage', 196608, true, 'BatVlt')
```

See Also

canDatabase, canMessage, messageInfo

start

Purpose Set CAN channel online

Syntax `start(canch)`

Description `start(canch)` starts the CAN channel `canch` on the CAN bus to send and receive messages. The CAN channel remains online unless:

- You call `stop` on this channel.
- The channel clears from the workspace.

Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
start(canch)
```

See Also `stop`

Purpose Set CAN channel offline

Syntax `stop(canch)`

Description `stop(canch)` stops the CAN channel `canch` on the CAN bus. The CAN channel also stops running when you clear `canch` from the workspace.

Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
start(canch)
stop(canch)
```

See Also `start`

transmit

Purpose Send CAN messages to CAN bus

Syntax `transmit(canch, message)`

Arguments

<code>canch</code>	The CAN channel that you specify to transmit the message.
<code>message</code>	The message or an array of messages that you specify to transmit via a CAN channel.

Description `transmit(canch, message)` sends the array of messages onto the bus via the CAN channel.

To understand the elements of a message, refer to `canMessage`.

Remarks The Transmit function ignores the `Timestamp` property and the `Error` property.

Examples

```
message = canMessage (250, false, 8)
message.Data = ([45 213 53 1 3 213 123 43])
canch = canChannel('Vector', 'CANCaseXL 1', 1)
start(canch)
transmit(canch, message)
```

To transmit an array, construct `message1` and `message2` as in this example, and type:

```
transmit(canch, [message, message1 message2])
```

To transmit messages on a remote frame, type:

```
message = canMessage (250, false 8, true)
message.Data = ([45 213 53 1 3 213 123 43])
message.Remote = true
canch = canChannel('Vector', 'CANCaseXL 1', 1)
start(canch)
```

`transmit(canch, message)`

See Also

`canChannel`, `canMessage`, `receive`

unpack

Purpose Unpack signal data from message

Syntax `value = unpack(message, startbit, signalsize, byteorder, datatype)`

Arguments

<code>message</code>	The CAN message structure that you specify for the signal to be unpacked from.
<code>startbit</code>	The signal's starting bit in the data. This is the least significant bit position in the signal data. Accepted values for <code>starbit</code> are from 0 through 63.
<code>signalsize</code>	The length of the signal in bits. Accepted values for <code>signalsize</code> are from 1 through 64.
<code>byteorder</code>	The signal binary or binblock format. Accepted values are <code>LittleEndian</code> and <code>BigEndian</code> .
<code>datatype</code>	The data type that you want to get the unpacked value in.
<code>value</code>	The value of the message that you specify to be unpacked.

Description `value = unpack(message, startbit, signalsize, byteorder, datatype)` takes a set of input parameters to unpack the signal value from the message and returns the value as output.

Examples `value = unpack(message, 0, 16, 'LittleEndian', 'int16')`

See Also `canMessage`, `extractAll`, `extractRecent`, `extractTime`, `pack`

Property Reference

CAN Channel Base Properties
(p. 7-2)

Apply to CAN channels on all devices

Device-Specific Properties (p. 7-5)

Apply to CAN channels on specific
devices

CAN Channel Base Properties

Channel Status Properties (p. 7-2)	Setting properties that specify different status of the CAN channel
CAN Message Properties (p. 7-2)	
CAN Database Properties (p. 7-3)	
Receiving Messages (p. 7-3)	Defining actions based on available messages on a CAN Channel
Error Logging (p. 7-4)	Properties for receiving and transmitting error messages

Channel Status Properties

BusLoad	Display load on CAN bus
BusStatus	Determine status of CAN bus
Database	Store CAN database information
InitializationAccess	Determine control of device channel
Running	Determine status of CAN channel
SilentMode	Specify if channel is active or silent

CAN Message Properties

Data	Set CAN message data
Database	Store CAN database information
Error	CAN message error frame
Extended	Identifier type for CAN message
ID	Identifier for CAN message
Name (Message)	CAN message name
Remote	Specify CAN message remote frame

Signals	Display physical signals defined in CAN message
Timestamp	Display message received timestamp

CAN Database Properties

Messages	Store message names from CAN database
Name (Database)	CAN database name
Path	Display CAN database directory path
Signals	Display physical signals defined in CAN message

Receiving Messages

MessageReceivedFcn	Specify function to run
MessageReceivedFcnCount	Specify number of messages available before function is triggered
MessagesAvailable	Display number of messages available to be received by CAN channel
MessagesReceived	Display number of messages received by CAN channel
MessagesTransmitted	Display number of messages transmitted by CAN channel

Error Logging

ReceiveErrorCount

Display number of received errors detected by channel

TransmitErrorCount

Display number of transmitted errors by channel

Device-Specific Properties

Device Settings (p. 7-5)	Properties displaying vendor-specific device information
Transceiver Settings (p. 7-5)	Properties displaying the CAN channel transceiver information
Bit Timing Settings (p. 7-5)	Properties defining the bit timing and segmentation

Device Settings

Device	Display CAN channel device type
DeviceChannelIndex	Display CAN device channel index
DeviceSerialNumber	Display CAN device serial number
DeviceVendor	Display device vendor name

Transceiver Settings

TransceiverName	Display name of CAN transceiver
TransceiverState	Display state or mode of CAN transceiver

Bit Timing Settings

BusSpeed	Display speed of CAN bus
NumOfSamples	Display number of samples available to channel
SJW	Display synchronization jump width (SJW) of bit time segment

TSEG1	Display amount that channel can lengthen sample time
TSEG2	Display amount that channel can shorten sample time

Properties — Alphabetical List

BusLoad

Purpose Display load on CAN bus

Description The BusLoad property displays information about the load on the CAN network for message traffic on Kvaser devices.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Float

Values The current message traffic on a CAN network is represented as a percentage ranging from 0.00% to 100.00%.

See Also **Functions**
canChannel

Purpose Display speed of CAN bus

Description The BusSpeed property determines the bit rate at which messages are transmitted. You can set BusSpeed to an acceptable bit rate using the configBusSpeed function.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Numerical

Values The default value is assigned by the vendor driver. To change the bus speed of your channel, use the configBusSpeed function and pass the channel name and the value as input parameters.

Examples To change the current BusSpeed of the CAN channel object canch to 250000, type:

```
configBusSpeed(canch, 250000)
```

See Also

Functions
canChannel, configBusSpeed

Properties
NumOfSamples, SJW, TSEG1, TSEG2

BusStatus

Purpose Determine status of CAN bus

Description The BusStatus property displays information about the state of the CAN bus.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	String

Values

- N/A
- BusOff
- ErrorOff
- ErrorActive

See Also **Functions**
canChannel

Purpose Set CAN message data

Description Use the Data property to define your message data in a CAN message.

Characteristics

Usage	CAN message
Read only	Never
Data type	Numeric

Values The data value is a uint8 array, based on the data length you specify in the message.

Examples To load data into a message, type:

```
message.Data = [23 43 23 43 54 34 123 1]
```

If you are using a CAN database for your message definitions, change values of the specific signals in the message directly.

You can also use the pack function to load data into your message.

See Also **Functions**

canMessage, pack

Database

Purpose Store CAN database information

Description The Database property stores information about an attached CAN database.

Characteristics

Usage	CAN channel, CAN message
Read only	For a CAN message property
Data type	Database handle

Values This property displays the database information that your CAN channel or CAN message is attached to. This property displays an empty structure, [], if your channel message is not attached to a database. You can edit the CAN channel property, Database, but cannot edit the CAN message property.

Examples To see information about the database attached to your CAN message, type:

```
message.Database
```

To set the database information on your CAN channel to C:\Database.dbc, type:

```
channel.Database = canDatabase('C:\Database.dbc')
```

Tip CAN database file names containing non-alphanumeric characters such as equal signs, ampersands, and so forth are incompatible with the Vehicle Network Toolbox. You can use a period sign in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

See Also

Functions

`attachDatabase`, `canChannel`, `canDatabase`, `canMessage`

Device

Purpose Display CAN channel device type

Description The Device property displays information about the device type to which the CAN channel is connected.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	String

Values Values are automatically defined when you configure the channel with the `canChannel` function.

See Also

Functions

`canChannel`, `canHWInfo`

Properties

`DeviceChannelIndex`, `DeviceVendor`

Purpose Display CAN device channel index

Description The DeviceChannelIndex property displays the channel index on which the selected CAN channel is configured.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Numeric

Values Values are automatically defined when you configure the channel with the canChannel function.

See Also

Functions
canChannel, canHWInfo

Properties
Device, DeviceVendor

DeviceSerialNumber

Purpose Display CAN device serial number

Description The DeviceSerialNumber property displays the serial number of the CAN device.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Numeric

Values Values are automatically defined when you configure the channel with the canChannel function.

See Also **Functions**
canChannel, canHWInfo

Properties
Device, DeviceVendor

Purpose Display device vendor name

Description The DeviceVendor property displays the name of the device vendor.

Characteristics

Usage	CAN channel
Read only	Always
Data type	String

Values Values are automatically defined when you configure the channel with the `canChannel` function.

See Also **Functions**
`canChannel`, `canHWInfo`

Properties
`Device`, `DeviceChannelIndex`, `DeviceSerialNumber`

Error

Purpose CAN message error frame

Description The Error property is a read-only value that identifies the specified CAN message as an error frame. The channel sets this property to `true` when it receives a CAN message as an error frame.

Characteristics	Usage	CAN message
	Read only	Always
	Data type	Boolean

Values

- `false` — The message is not an error frame.
- `true` — The message is an error frame.

The Error property displays `false`, unless the message is an error frame.

See Also **Functions**
`canMessage`

Purpose Identifier type for CAN message

Description The Extended property is the identifier type for a CAN message. It can either be a standard identifier or an extended identifier.

Characteristics	Usage	CAN message
	Read only	Always
	Data type	Boolean

Values

- `false` — The identifier type is standard (11 bits).
- `true` — The identifier type is extended (29 bits).

Examples To set the message identifier type to extended with the ID set to 2350 and the data length to 8 bytes, type:

```
message = canMessage(2350, true, 8)
```

You cannot edit this property after the initial configuration.

See Also

Functions

`canMessage`

Properties

ID

ID

Purpose Identifier for CAN message

Description The ID property represents a numeric identifier for a CAN message.

Characteristics

Usage	CAN message
Read only	Always
Data type	Numeric

Values The ID value must be a positive integer from:

- 0 through 2047 for a standard identifier
- 0 through 536,870,911 for an extended identifier

You can also specify a hexadecimal value using the `hex2dec` function.

Examples To configure a message ID to a standard identifier of value 300 and a data length of 8 bytes, type:

```
message = canMessage(300, false, 8)
```

See Also **Functions**

`canMessage`

Properties

Extended

Purpose Determine control of device channel

Description The `InitializationAccess` property determines if the configured CAN channel object has full control of the device channel. You can change some property values of the hardware channel only if the object has full control over the hardware channel.

Note Only the first channel created on a device is granted initialization access.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Boolean

Values

- Yes — Has full control of the hardware channel and can change the property values.
- No — Does not have full control and cannot change property values.

See Also

Functions

`canChannel`

MessageReceivedFcn

Purpose Specify function to run

Description Configure `MessageReceivedFcn` as a callback function to run a string expression, a function handle, or a cell array when a specified number of messages are available.

The `MessageReceivedFcnCount` property defines the number of messages available before the configured `MessageReceivedFcn` runs.

Characteristics

Usage	CAN channel
Read only	Never
Data type	Callback function

Values The default value is an empty string. You can specify the name of a callback function that you want to run when the specified number of messages are available.

Examples `canch.MessageReceivedFcn = @Myfunction`

You can also use the `set` function to set the values of this property.

See Also **Functions**
`canChannel`, `set`

Properties
`MessageReceivedFcnCount`, `MessagesAvailable`

Purpose Specify number of messages available before function is triggered

Description Configure MessageReceivedFcnCount to the number of messages that must be available before a MessageReceivedFcn is triggered.

Characteristics

Usage	CAN channel
Read only	While channel is online
Data type	Double

Values The default value is 1. You can specify a positive integer for your MessageReceivedFcnCount.

Examples `canch.MessageReceivedFcnCount = 55`

You can also use the `set` function to set the values of this property.

See Also **Functions**

`canChannel`, `set`

Properties

`MessageReceivedFcn`, `MessagesAvailable`

Messages

Purpose Store message names from CAN database

Description The Messages property stores the names of all messages defined in the selected CAN database.

Characteristics

Usage	CAN database
Read only	Always
Data type	String

Values The Messages property displays a cell array of strings. You cannot edit this property.

See Also `canDatabase`, `messageInfo`

Purpose Display number of messages available to be received by CAN channel

Description The MessagesAvailable property displays the total number of messages available to be received by a CAN channel.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values The value is 0 when no messages are available.

See Also

Functions

canChannel

Properties

MessagesReceived, MessagesTransmitted

MessagesReceived

Purpose Display number of messages received by CAN channel

Description The MessagesReceived property displays the total number of messages received since the channel was last started.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values The value is 0 when no messages have been received. This number increments based on the number of messages the channel receives.

See Also

Functions

canChannel, canHWInfo

Properties

MessagesAvailable, MessagesTransmitted

Purpose Display number of messages transmitted by CAN channel

Description The MessagesTransmitted property displays the total number of messages transmitted since the channel was last started.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Double

Values The default is 0 when no messages have been sent. This number increments based on the number of messages the channel transmits.

See Also

Functions

canChannel

Properties

MessagesAvailable, MessagesReceived

Name (Database)

Purpose CAN database name

Description The Name (Database) property displays the name of the database.

Characteristics	Usage	CAN database
	Read only	Always
	Data type	String

Values Name is a string value. This value is acquired from the name of the database file. You cannot edit this property.

See Also	Functions
	canDatabase
	Properties
	Extended, ID

Purpose CAN message name

Description The Name (Message) property displays the name of the message.

Characteristics	Usage	CAN message
	Read only	Always
	Data type	String

Values Name is a string value. This value is acquired from the name of the message you defined in the database. You cannot edit this property if you are defining raw messages.

See Also **Functions**

canMessage

Properties

Extended, ID

NumOfSamples

Purpose Display number of samples available to channel

Description The NumOfSamples property displays the total number of samples available to this channel. If you do not specify a value, the BusSpeed property determines the default value.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Double

Values The value is a positive integer based on the driver settings for the channel.

See Also

Functions

canChannel, configBusSpeed

Properties

BusSpeed, SJW, TSEG1, TSEG2

Purpose Display CAN database directory path

Description The Path property displays the path to the CAN database.

Characteristics	Usage	CAN database
	Read only	Always
	Data type	String

Values The path name is a string value, pointing to the CAN database in your directory structure.

See Also **Functions**
canDatabase

ReceiveErrorCount

Purpose Display number of received errors detected by channel

Description The ReceiveErrorCount property displays the total number of errors detected by this channel during receive operations.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values The value is 0 when no error messages have been received.

See Also

Functions

canChannel, receive

Properties

TransmitErrorCount

Purpose Specify CAN message remote frame

Description Use the `Remote` property to specify the CAN message as a remote frame.

Characteristics

Usage	CAN message
Read only	Never
Data type	Boolean

Values

- `{false}` — The message is not a remote frame.
- `true` — The message is a remote frame.

Examples To change the default value of `Remote` and make the message a remote frame, type:

```
message.Remote = true
```

See Also

Functions

`canMessage`

Running

Purpose Determine status of CAN channel

Description The Running property displays information about the state of the CAN channel.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Boolean

Values

- `{false}` — The channel is offline.
- `true` — The channel is online.

Use the `start` function to set your channel online.

See Also **Functions**
`canChannel`, `start`

Purpose Specify if channel is active or silent

Description Specify whether the channel operates silently. By default `SilentMode` is false. In this mode, the channel both transmits and receives messages normally and performs other tasks on the network such as acknowledging messages and creating error frames.

To observe all message activity on the network and perform analysis without affecting the network state or behavior, change `SilentMode` to true. In this mode, you can only receive messages and not transmit any.

Characteristics	Usage	CAN channel
	Read only	Never
	Data type	Boolean

Values

- `{false}` — The channel is in normal or active mode.
- `true` — The channel is in silent mode.

Examples To configure the channel to silent mode, type:

```
canch.SilentMode = true
```

To configure the channel to normal mode, type:

```
canch.SilentMode = false
```

You can also use the `set` function to set the values of this property.

See Also **Functions**
`canChannel`, `set`

Signals

Purpose Display physical signals defined in CAN message

Description The `Signals` property allows you to view and edit signal values defined for a CAN message. This property displays an empty structure if the message has no defined signals or a CAN database is not attached to the message. The input values for this property depends on the signal type.

Characteristics

Usage	CAN message
Read only	Sometimes
Data type	Structure

Examples Display signals defined in the CAN message, `message`:

```
message.Signals
```

```
ans =
```

```
    VehicleSpeed: 0  
    EngineRPM: 250
```

Change the value of a signal:

```
message.Signals.EngineRPM = 300
```

See Also **Functions**

`canMessage`, `canDatabase`

Purpose Display synchronization jump width (SJW) of bit time segment

Description In order to adjust the on-chip bus clock, the CAN controller may shorten or prolong the length of a bit by an integral number of time segments. The maximum value of these bit time adjustments are termed the synchronization jump width or SJW.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Numeric

Values The value of the SJW is determined by the specified bus speed.

See Also **Functions**
canChannel, configBusSpeed

Properties
BusSpeed, NumOfSamples, TSEG1, TSEG2

Timestamp

Purpose Display message received timestamp

Description The Timestamp property displays the time at which the message was received on a CAN channel. This time is based on the receiving channel's start time.

Characteristics	Usage	CAN message
	Read only	Never
	Data type	Double

Values Timestamp displays a numeric value indicating the time the message was received, based on the start time of the CAN channel

Examples To set the time stamp of a message to 12, type:

```
message.Timestamp = 12
```

See Also **Functions**
canChannel, canMessage, receive, replay

Purpose Display name of CAN transceiver

Description The CAN transceiver translates the digital bit stream going to and coming from the CAN bus into the real electrical signals present on the bus.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	String

Values Values are automatically defined when you configure the channel with the `canChannel` function.

See Also

Functions

`canChannel`

Properties

`TransceiverState`

TransceiverState

Purpose Display state or mode of CAN transceiver

Description If your CAN transceiver allows you to control its mode, you can use the TransceiverState property to set the mode.

Characteristics	Usage	CAN channel
	Read only	Never
	Data type	Numeric

Values The values are defined by the transceiver manufacturer. Refer to your CAN transceiver documentation for the appropriate transceiver modes. Possible modes representing the numeric value specified are:

- high speed
- high voltage
- sleep
- wake up

See Also **Functions**

canChannel

Properties

TransceiverName

Purpose Display number of transmitted errors by channel

Description The TransmitErrorCount property displays the total number of errors detected by this channel during transmit operations.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values The value is 0 when no error messages have been transmitted.

See Also

Functions

canChannel, transmit

Properties

ReceiveErrorCount

TSEG1

Purpose Display amount that channel can lengthen sample time

Description The TSEG1 property displays the amount in bit time segments that the channel can lengthen the sample time to compensate for delay times in the network.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Double

Values The value is inherited when you configure the bus speed of your CAN channel.

See Also

Functions

canChannel, configBusSpeed

Properties

BusSpeed, NumOfSamples, SJW, TSEG2

Purpose Display amount that channel can shorten sample time

Description The TSEG2 property displays the amount of bit time segments the channel can shorten the sample to resynchronize.

Characteristics	Usage	CAN channel
	Read only	Always
	Data type	Double

Values The value is inherited when you configure the bus speed of your CAN channel.

See Also **Functions**
canChannel, configBusSpeed

Properties
BusSpeed, NumOfSamples, SJW, TSEG1

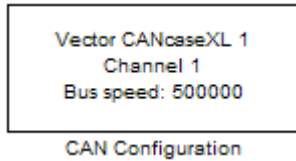
Block Reference

CAN Configuration

Purpose Configure parameters for specified CAN device

Library Vehicle Network Toolbox: CAN Communication

Description



The CAN Configuration block configures parameters for a CAN device that you can use to transmit and receive messages.

Specify the configuration of your CAN device before you configure other CAN blocks.

Use one CAN Configuration block to configure each device that sends and receives messages in your model. If you use a CAN Receive or a CAN Transmit block to receive and send messages on a device, your model checks to see if there is a corresponding CAN Configuration block for the specified device. If the device is not configured, you will see a prompt advising you to use a CAN Configuration block to configure the specified device.

Note You need a license for both the Vehicle Network Toolbox and Simulink software to use this block.

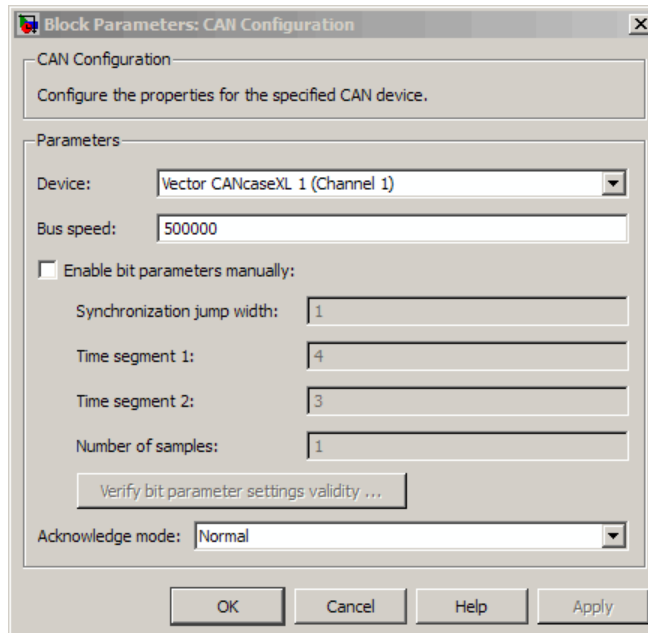
Other Supported Features

The CAN Configuration block supports the use of Simulink® Accelerator™ mode. Using this feature, you can speed up the execution of Simulink models.

For more information on this feature, see the Simulink documentation.

Dialog Box

Use the Block Parameters dialog box to select your CAN device configuration.



Device

Select the CAN device and a channel on the device that you want to use from the list. Use this device to transmit and/or receive messages. The device driver determines the default bus speed.

Bus speed

Set the bus speed property for the selected device. The default bus speed is the default assigned by the selected device.

Enable bit parameters manually

Select this check box to specify bit parameter settings manually. The bit parameter settings include **Synchronization jump width**, **Time segment 1**, **Time segment 2**, and **Number of**

CAN Configuration

samples. If you do not select this option, the device automatically assigns the bit parameters depending on the bus speed setting.

Tip Use the default bit parameter settings unless you have specific timing requirements for your CAN connection.

Synchronization jump width

Specify the maximum value of the bit time adjustments. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determine the default value. To change this value, select the **Enable bit parameters manually** check box first. Refer to the SJW property for more information.

Time segment 1

Specify the amount of bit time segments that the channel can lengthen the sample time. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determines the default value. To change this value, select the **Enable bit parameters manually** check box first. Refer to the TSEG1 property for more information.

Time segment 2

Specify the amount of bit time segments that the channel can shorten the sample time to resynchronize. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determines the default value. To change this value, select the **Enable bit parameters manually** check box first. Refer to the TSEG2 property for more information.

Number of samples

Specify the total number of samples available to this channel. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determines the default value. To change this value, select the **Enable bit parameters**

manually check box first. Refer to the NumOfSamples property for more information.

Verify bit parameter settings validity

If you have set the bit parameter settings manually, click this button to see if your settings are valid. The block then runs a check to see if the combination of your bus speed setting and the bit parameter value forms a valid value for the CAN device. If the new bit parameter values do not form a valid combination, the verification fails and displays an error message.

Acknowledge mode

Specify whether the channel is in Normal or Silent mode. By default **Acknowledge mode** is Normal. In this mode, the channel both receives and transmits messages normally and performs other tasks on the network such as acknowledging messages and creating error frames. To observe all message activity on the network and perform analysis, without affecting the network state or behavior, select **Silent**. In Silent mode, you can only receive messages and not transmit.

Note Use Silent mode only if you want to observe and analyze your network activity.

See Also

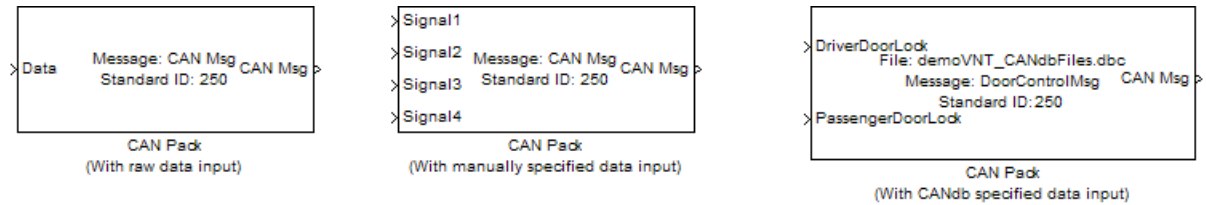
CAN Receive, CAN Transmit

CAN Pack

Purpose Pack individual signals into CAN message

Library CAN Communication

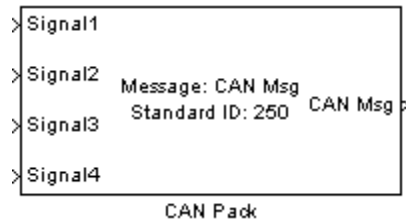
Description



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.



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:

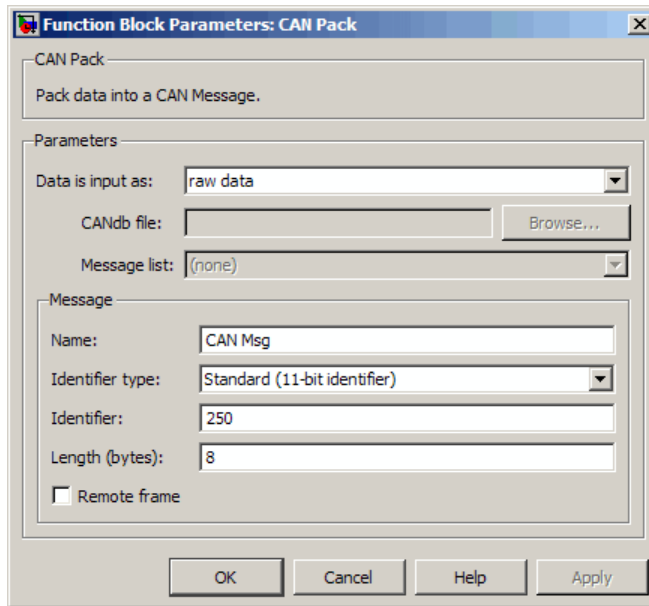
- The use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Real-Time Workshop® to deploy models to targets.

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.

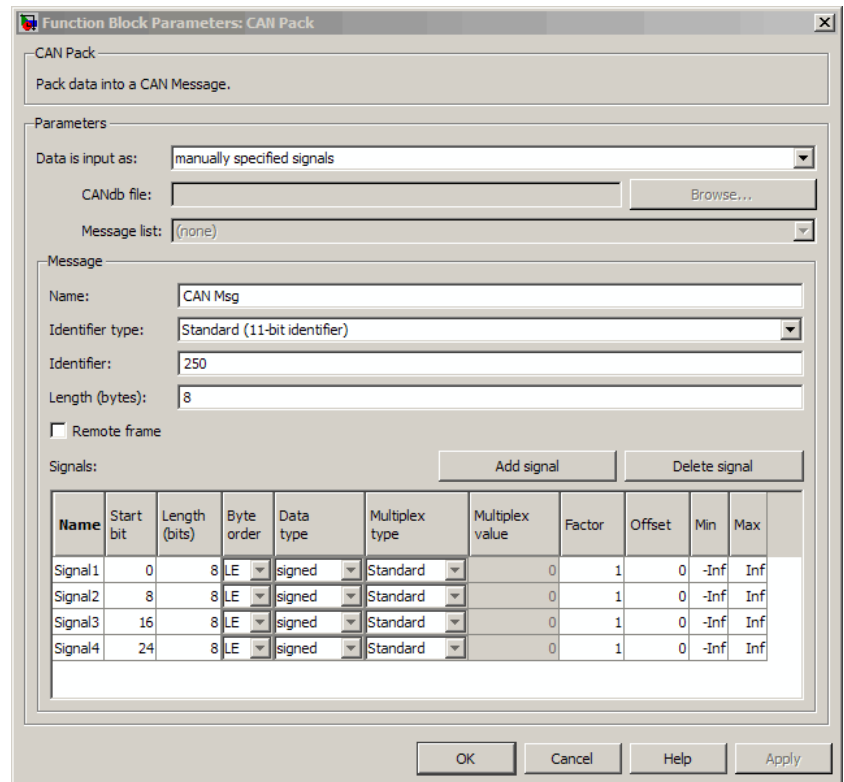


Parameters

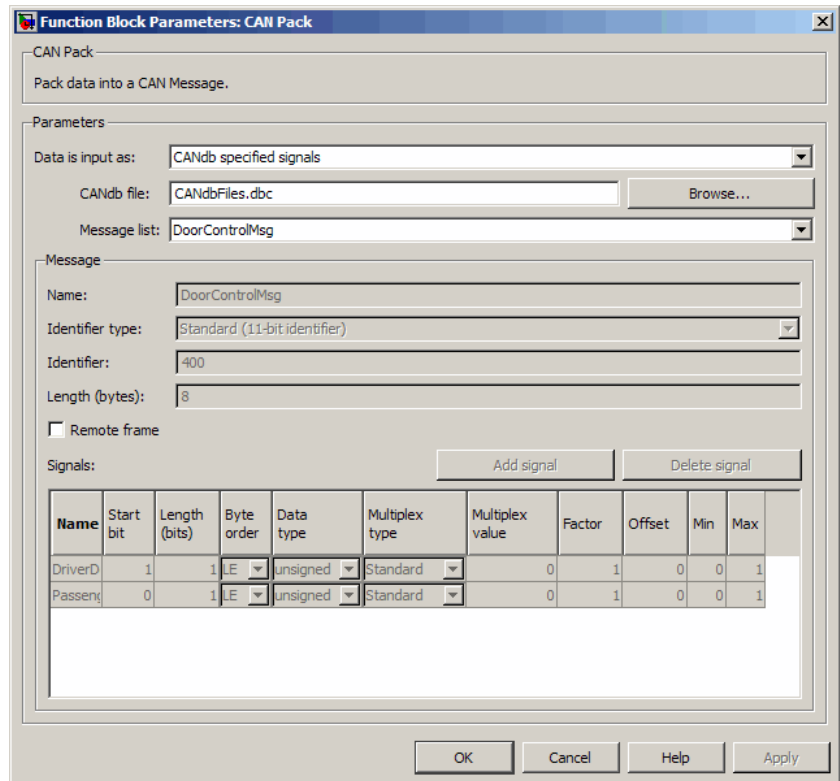
Data is input as

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 is 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:

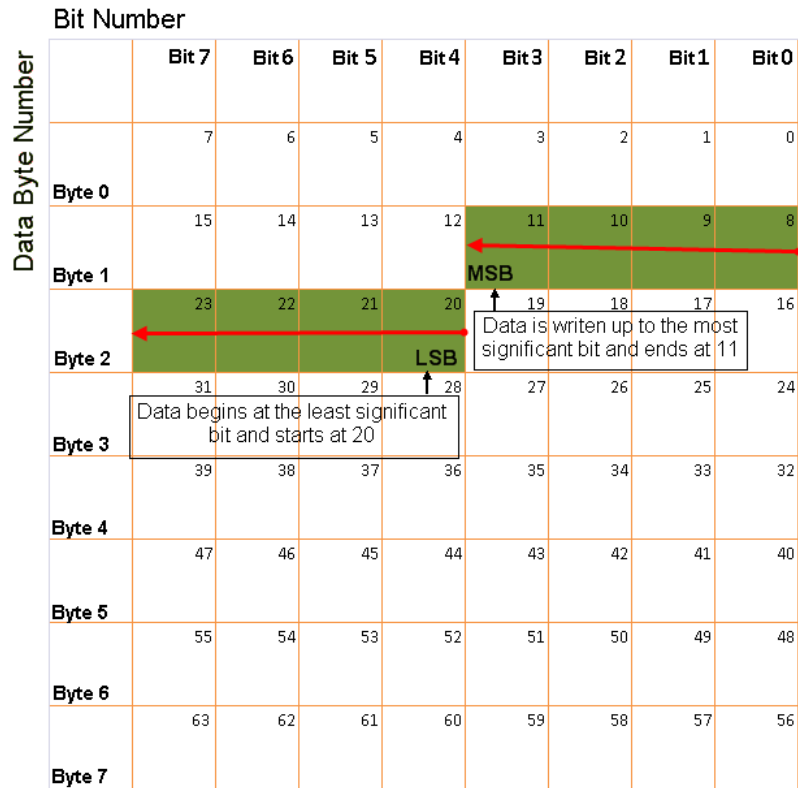
- LE: Where the byte order is in little-endian format (Intel®). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20,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

Bit Number		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	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	19	18	17	16
	Byte 3	31	30	29	28	27	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

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.

CAN Pack



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 9-16 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 9-16 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 9-16 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 9-16 to understand how physical values are converted to raw values packed into a message.

Conversion Formula

The conversion formula is

$$\text{raw_value} = (\text{physical_value} - \text{Offset}) / \text{Factor}$$

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.

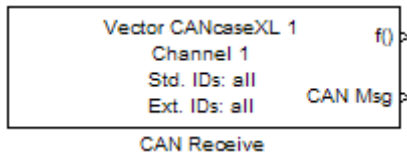
See Also CAN Unpack

CAN Receive

Purpose Receive CAN messages from specified CAN device

Library Vehicle Network Toolbox: CAN Communication

Description



The CAN Receive block receives messages from the CAN network and delivers them to the Simulink model. It outputs one message or all messages at each timestep, depending on the block parameters.

Note You need a license for both the Vehicle Network Toolbox and Simulink software to use this block.

The CAN Receive block has two output ports:

- The `f()` output port is a trigger to a Function-Call subsystem. If the block receives a new message, it triggers a Function-Call from this port. You can then connect to a Function-Call Subsystem to unpack and process a message.
- The `CAN Msg` output port contains a CAN message received at that particular timestep.

The CAN Receive block stores CAN messages in a first-in, first-out (FIFO) buffer. The FIFO buffer delivers the messages to your model in the queued order at every timestep.

Other Supported Feature

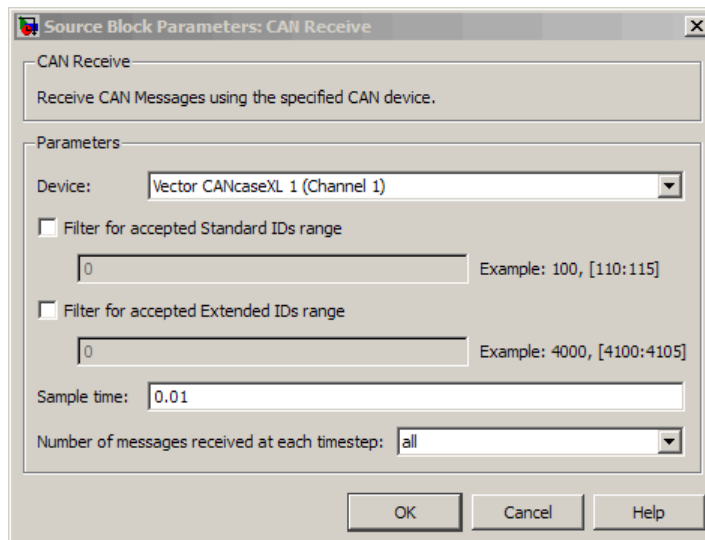
The CAN Receive block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

For more information on this feature, see the Simulink documentation.

Dialog Box

Use the Source Block Parameters dialog box to select your CAN Receive block parameters.

Tip Configure your CAN Configuration block before you configure the CAN Receive block parameters.



Device

Select the CAN device and a channel on the device you want to receive CAN messages from. This field lists all the devices installed on the system. It displays the vendor name, the device name, and the channel ID. The default is the first available device on your system.

Filter for accepted Standard IDs range

Select this check box to accept a specific range of standard IDs. By default, the CAN Receive block accepts all standard IDs.

Enter the ID ranges in the text field. You can specify a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to accept IDs from 400 through 500 and 600 through 650, enter [400:500], [600:650]. Standard IDs must be a positive integer from 0 through 2047. You can also specify a hexadecimal value using the `hex2dec` function.

Filter for accepted Extended IDs range

Select this check box to accept a specific range of extended IDs.

By default the CAN Receive block accepts all extended IDs.

Enter the ID ranges in the text field. You can specify an array of IDs separated by commas. For example, to accept IDs from 3000 through 3500 and 3600 through 3620, enter [3000:3500], [3600:3620]. Extended IDs must be a positive integer from 0 through 536870911. You can also specify a hexadecimal value using the `hex2dec` function.

Sample time

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation.

This value defines the frequency at which the CAN Receive block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify `-1` as your sample time. The default value is 0.01 (in seconds).

Number of messages received at each timestep

Select how many messages the block receives at each specified timestep. The choices are `1` and `all`. By default, the block receives one message at each timestep. Then, the FIFO buffer delivers one new message to the Simulink model. If the block does not receive any message before the next timestep it outputs the last received message.

If you select `all`, the CAN Receive block delivers all available messages in the FIFO buffer to the model during a specific timestep. The block generates one function call for every message delivered to the model for that particular timestep. The output port always contains one CAN message at a time.

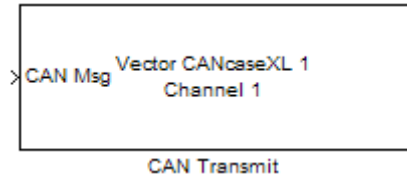
See Also CAN Configuration, CAN Unpack

CAN Transmit

Purpose Transmit CAN message to selected CAN device

Library Vehicle Network Toolbox: CAN Communication

Description



The CAN Transmit block transmits messages to the CAN network using the specified CAN device. The CAN Transmit block can transmit a single message or an array of messages during a given timestep. To transmit an array of messages, use a mux (multiplex) block from the Simulink block library.

Note You need a license for both the Vehicle Network Toolbox and Simulink software to use this block.

The CAN Transmit block has one input port. This port accepts a CAN message packed using the CAN Pack block. It has no output ports.

Other Supported Feature

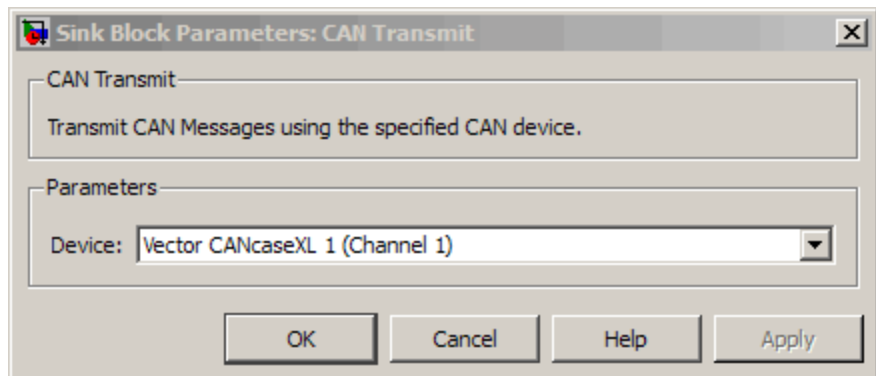
The CAN Transmit block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

For more information on this feature, see the Simulink documentation.

Dialog Box

Use the Sink Block Parameters dialog box to select your CAN Transmit block parameters.

Tip Configure your CAN Configuration block before you configure the CAN Transmit block parameters.



Device

Select the CAN device and a channel on the device to use to transmit CAN messages to the network. This list shows all the devices installed on the system. It displays the vendor name, the device name, and the channel ID. The default is the first available device on your system.

See Also

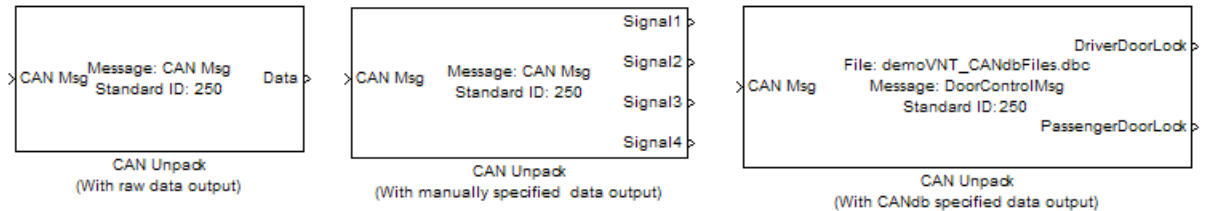
CAN Configuration, CAN Pack

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.

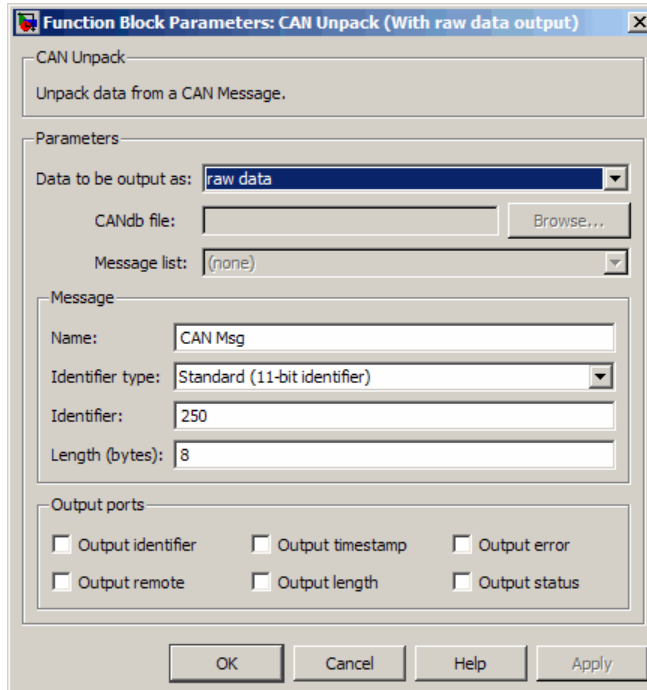
Note Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32-bits long.

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

CAN Unpack

Dialog Box

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

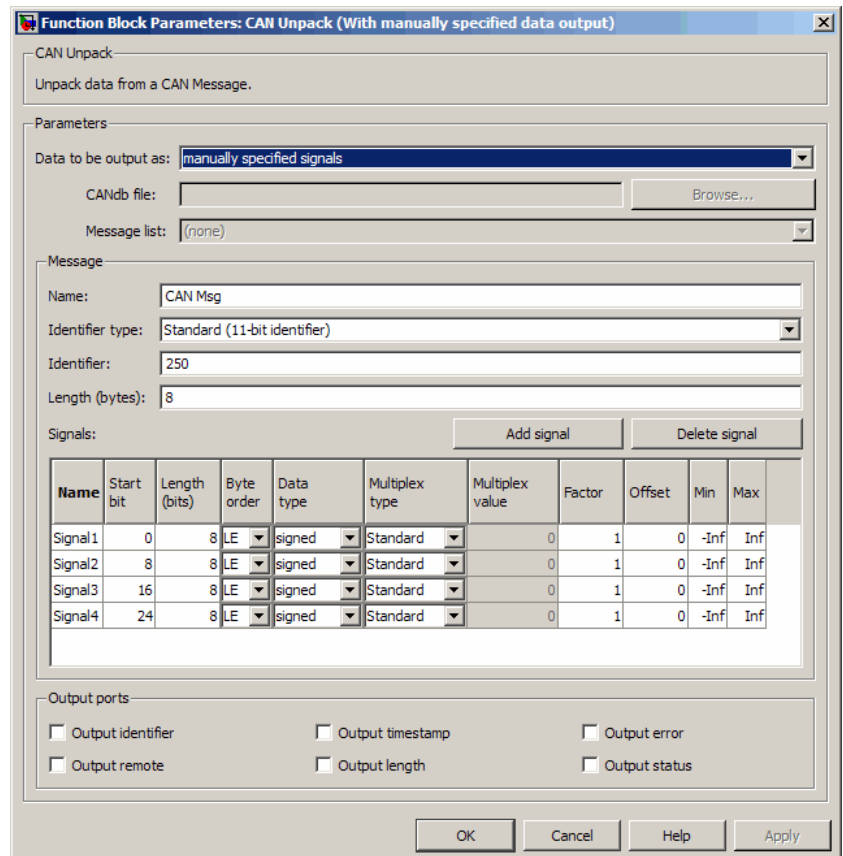


Parameters

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.

CAN Unpack

Function Block Parameters: CAN Unpack (With CANdb specified data output)

CAN Unpack
Unpack data from a CAN Message.

Parameters

Data to be output as: CANdb specified signals

CANdb file: CANdbFiles.dbc

Message list: DoorControlMsg

Message

Name: DoorControlMsg

Identifier type: Standard (11-bit identifier)

Identifier: 400

Length (bytes): 8

Signals:

Name	Start bit	Length (bits)	Byte order	Data type	Multiplex type	Multiplex value	Factor	Offset	Min	Max
DriverD	1	1	LE	unsigned	Standard	0	1	0	0	1
Passent	0	1	LE	unsigned	Standard	0	1	0	0	1

Output ports

Output identifier Output timestamp Output error

Output remote Output length Output status

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:

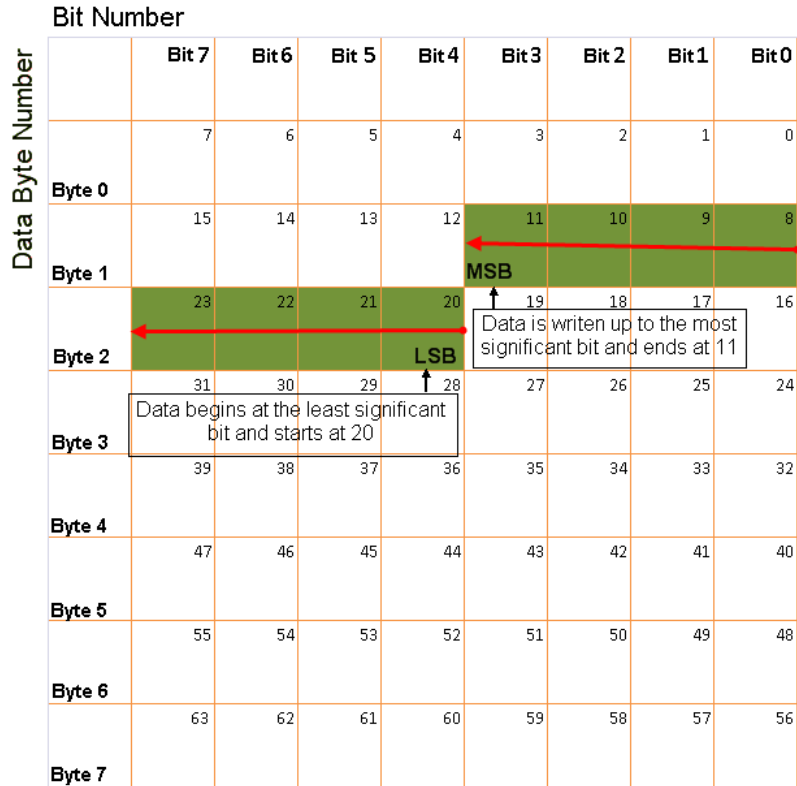
- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the 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
			7	6	5	4	3	2	1
Byte 0									
		15	14	13	12	11	10	9	8
Byte 1									
		23	22	21	20	19	18	17	16
Byte 2									
		31	30	29	28	27	26	25	24
Byte 3									
		39	38	37	36	35	34	33	32
Byte 4									
		47	46	45	44	43	42	41	40
Byte 5									
		55	54	53	52	51	50	49	48
Byte 6									
		63	62	61	60	59	58	57	56
Byte 7									

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.

CAN Unpack



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 9-35 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 9-35 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 9-35 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 9-35 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

$$\text{physical_value} = \text{raw_value} * \text{Factor} + \text{Offset}$$

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

CAN Unpack

A

attachDatabase function 6-2

B

base properties

list for can channel 7-2

bit timing settings

device-specific properties 7-5

Block Library 4-3

blocks

CAN Configuration 9-2

CAN Pack 9-6

CAN Receive 9-18

CAN Transmit 9-22

CAN Unpack 9-24

using the Vehicle Network Toolbox block
library 4-1

building

CAN messages 1-16

BusLoad property 8-2

BusSpeed property 8-3

BusStatus property 8-4

C

CAN

transmit message 1-18

workflow 1-9

can channel

base properties 7-2

CAN Channel

interface-specific properties 7-5

CAN channels

configuring properties 1-14

disconnecting 1-27

SilentMode 1-26

starting 1-15

CAN communication

session 1-9

CAN communications

configuring 1-11

CAN Configuration block 9-2

CAN devices

connecting 1-13

CAN messages

building 1-16

filtering 1-22

packing 1-17

receiving 1-19

unpacking 1-21

CAN Pack block 9-6

CAN Receive block 9-18

CAN Transmit block 9-22

CAN Unpack block 9-24

can.vector.channel, configBusSpeed
function 6-13

can.vector.channel, fileterBlockRange
function 6-22

can.vector.channel, filterAcceptRange
function 6-20

can.vector.channel, filterReset
function 6-24

can.vector.channel, filterSet function 6-25

canChannel function 6-3

canChannel, get function 6-27

canChannel, receive function 6-30

canChannel, receive raw function 6-31

canChannel, replay function 6-33

canChannel, set function 6-35

canChannel, start function 6-38

canChannel, stop function 6-39

canChannel, transmit function 6-40

canDatabase function 6-5

canHWInfo function 6-6

canMessage function 6-7

canMessageCompatibilityMode function 6-9

canSupport function 6-11

canTool function 6-12

cleaning

- MATLAB workspace 1-28
- configuring
 - CAN channel properties 1-14 1-26
 - CAN communications 1-11
 - message filtering 1-22
- connecting
 - CAN devices 1-13

D

- Data property 8-5
- Database property 8-6
- Device property 8-8
- device-specific properties
 - list by object type 7-5
- DeviceChannelIndex property 8-9
- DeviceSerialNumber property 8-10
- DeviceVendor property 8-11
- disconnecting
 - CAN channels 1-27

E

- Error property 8-12
- Extended property 8-13
- extractAll function 6-15
- extractRecent function 6-17
- extractTime function 6-19

F

- filtering
 - CAN messages 1-22
- functions
 - attachDatabase 6-2
 - canChannel 6-3
 - canChannel, transmit 6-40
 - canChannelset 6-35
 - canChannelstart 6-38
 - canDatabase 6-5
 - canHWInfo 6-6

- canMessage 6-7
- canMessageCompatibilityMode 6-9
- canSupport 6-11
- canTool 6-12
- configBusSpeed, can.vector.channel 6-13
- extractAll 6-15
- extractRecent 6-17
- extractTime 6-19
- filterAcceptRange,
 - can.vector.channel 6-20
- filterBlockRange,
 - can.vector.channel 6-22
- filterReset, can.vector.channel 6-24
- filterSet, can.vector.channel 6-25
- get, canChannel 6-27
- messageInfo, canChannel 6-28
- pack 6-29
- receive raw, canChannel 6-31
- receive, canChannel 6-30
- replay, canChannel 6-33
- signalInfo, canDatabase 6-36
- stop, canChannel 6-39
- unpack 6-42

I

- ID property 8-14
- InitializationAccess property 8-15

M

- MATLAB workspace
 - cleaning 1-28
- message
 - transmit 1-18
- message filtering
 - configuring 1-22
- messageInfo function 6-28
- MessageReceivedFcn property 8-16
- MessageReceivedFcnCount property 8-17

- messages
 - packing 1-17
 - receiving 1-19
 - unpacking 1-21
- Messages property 8-18
- MessagesAvailable property 8-19
- MessagesReceived property 8-20
- MessagesTransmitted property 8-21

N

- Name (Database) property 8-22
- Name (Message) property 8-23
- NumOfSamples property 8-24

P

- pack function 6-29
- packing
 - CAN messages 1-17
- properties
 - BusLoad 8-2
 - BusSpeed 8-3
 - BusStatus 8-4
 - Data 8-5
 - Database 8-6
 - Device 8-8
 - DeviceChannelIndex 8-9
 - DeviceSerialNumber 8-10
 - DeviceVendor 8-11
 - Error 8-12
 - Extended 8-13
 - ID 8-14
 - InitializationAccess 8-15
 - MessageReceivedFcn 8-16
 - MessageReceivedFcnCount 8-17
 - Messages 8-18
 - MessagesAvailable 8-19
 - MessagesReceived 8-20
 - MessagesTransmitted 8-21

- Name (Database) 8-22
- Name (Message) 8-23
- NumOfSamples 8-24
- ReceiveErrorCount 8-25 to 8-26
- Remote 8-27
- Running 8-28
- Signals 8-30
- SilentMode 8-29
- SJW 8-31
- synchronization jump width 8-31
- Timestamp 8-32
- TransceiverName 8-33
- TransceiverState 8-34
- TransmitErrorCount 8-35
- TSEG1 8-36
- TSEG2 8-37
- property values
 - base
 - for can channel 7-2
 - device-specific 7-5

R

- ReceiveErrorCount property 8-25 to 8-26
- receiving
 - CAN messages 1-19
- Remote property 8-27
- Running property 8-28

S

- signalInfo, signalInfo function 6-36
- Signals property 8-30
- SilentMode property 8-29
- Simulink Library Browser 4-4
- SJW property 8-31
- starting
 - CAN channels 1-15
- synchronization jump width
 - properties 8-31

T

- Timestamp
 - properties 8-32
- transceiver settings
 - device-specific properties 7-5
- TransceiverName
 - properties 8-33
- TransceiverState
 - properties 8-34
- transmit
 - CAN message 1-18
- TransmitErrorCount
 - properties 8-35
- TSEG1
 - properties 8-36
- TSEG2

- properties 8-37

U

- unpack function 6-42
- unpacking
 - CAN messages 1-21

V

- Vector CAN device
 - device-specific properties 7-5
- Vehicle Network Toolbox block library
 - using 4-1
- Vehicle Network Toolbox Block Library
 - opening 4-3