

# Vehicle Network Toolbox™

## User's Guide

**R2012a**

MATLAB®  
& SIMULINK®

## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*Vehicle Network Toolbox™ User's Guide*

© COPYRIGHT 2009–2012 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](http://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

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

**Revision History**

March 2009 Online only  
September 2009 Online only  
March 2010 Online only  
September 2010 Online only  
April 2011 Online only  
September 2011 Online only  
March 2012 Online only

New for Version 1.0 (Release 2009a)  
Revised for Version 1.1 (Release 2009b)  
Revised for Version 1.2 (Release 2010a)  
Revised for Version 1.3 (Release 2010b)  
Revised for Version 1.4 (Release 2011a)  
Revised for Version 1.5 (Release 2011b)  
Revised for Version 1.6 (Release 2012a)



## Getting Started

**1**

<b>Product Description</b> .....	1-2
Key Features .....	1-2
<b>Product Capabilities</b> .....	1-3
Vehicle Network Toolbox Characteristics .....	1-3
Interaction Between the Toolbox and Its Components .....	1-4
Expected Background .....	1-6
Related Products .....	1-6
<b>Install Required Components</b> .....	1-8
Required Components .....	1-8
Install Devices and Drivers .....	1-8
Install the Toolbox .....	1-10
Supported Hardware .....	1-10
<b>Vehicle Network Communication in MATLAB</b> .....	1-12
Transmit Workflow .....	1-13
Receive Workflow .....	1-14
<b>Vehicle Network Communication Examples</b> .....	1-15
Prerequisites .....	1-15
Discover Installed Hardware .....	1-16
Create a CAN Channel .....	1-17
Configure Properties .....	1-18
Start the Channel .....	1-19
Create a Message .....	1-20
Pack a Message .....	1-21
Transmit a Message .....	1-22
Receive a Message .....	1-23
Unpack a Message .....	1-25
Save and Load a CAN Channel .....	1-25
Filter Messages .....	1-26
Multiplex Signals .....	1-27
Configure Silent Mode .....	1-30

Disconnect Channels and Clean Up .....	1-31
<b>Accessing the Toolbox .....</b>	<b>1-34</b>
Explore the Toolbox .....	1-34
Get Help .....	1-34
View Examples .....	1-34

## Using a CAN Database

# 2

<b>Vector CAN Database Support .....</b>	<b>2-2</b>
<b>Load and Create Messages With a .dbc File .....</b>	<b>2-3</b>
Load the CAN Database .....	2-3
Create a CAN Message .....	2-4
Access Signals in the Constructed CAN Message .....	2-4
Add a Database to a CAN Channel .....	2-5
Update Database Information .....	2-5
Create and Process Messages Using Database Definitions .....	2-5
<b>Other Uses of the CAN Database .....</b>	<b>2-17</b>
View Message Information in a CAN Database .....	2-17
View Signal Information in a CAN Message .....	2-18
Attach a CAN Database to Existing Messages .....	2-19

## Monitoring CAN Message Traffic

# 3

<b>CAN Tool .....</b>	<b>3-2</b>
About the CAN Tool .....	3-2
Opening the CAN Tool .....	3-2
CAN Tool Fields .....	3-2
<b>Using the CAN Tool .....</b>	<b>3-9</b>

Viewing Messages on a Channel .....	3-9
Configuring the Channel Bus Speed .....	3-9
Filtering CAN Messages in the CAN Tool .....	3-10
Attaching a Database .....	3-11
Changing the Message Count .....	3-13
Changing the Number Format .....	3-13
Viewing Unique Messages .....	3-13
Saving the Message Log File .....	3-14

## Communications in Simulink

# 4

<b>Overview</b> .....	4-2
<b>CAN Communication in Simulink</b> .....	4-3
Message Transmission Workflow .....	4-4
Message Reception Workflow .....	4-5
<b>Opening the Vehicle Network Toolbox Block Library</b> .....	4-7
Using the MATLAB Command Window .....	4-7
Using the Simulink Library Browser .....	4-8
<b>Building Simulink Models to Transmit and Receive Messages</b> .....	4-10
Build a Message Transmit Model .....	4-10
Build a Message Receive Model .....	4-15
Save and Run the Model .....	4-21
<b>Create Custom Blocks</b> .....	4-25

## Function Reference

# 5

<b>CAN Channel Construction</b> .....	5-2
---------------------------------------	-----

<b>CAN Channel Configuration</b> .....	<b>5-3</b>
<b>CAN Channel Execution</b> .....	<b>5-4</b>
<b>CAN Channel Status</b> .....	<b>5-5</b>
<b>CAN Database</b> .....	<b>5-6</b>
<b>CAN Message Handling</b> .....	<b>5-7</b>
<b>CAN Message Filtering</b> .....	<b>5-8</b>
<b>Information and Help</b> .....	<b>5-9</b>
<b>Graphical Tools</b> .....	<b>5-10</b>
<b>Vector Informatik</b> .....	<b>5-11</b>

## Functions — Alphabetical List

### 6

## Property Reference

### 7

<b>CAN Channel Base Properties</b> .....	<b>7-2</b>
Channel Status Properties .....	<b>7-2</b>
CAN Message Properties .....	<b>7-2</b>
CAN Database Properties .....	<b>7-3</b>
Receiving Messages .....	<b>7-3</b>
Error Logging .....	<b>7-4</b>
<b>Device-Specific Properties</b> .....	<b>7-5</b>
Device Settings .....	<b>7-5</b>



Transceiver Settings .....	7-5
Bit Timing Settings .....	7-5

## Properties — Alphabetical List

---

**8**

## Block Reference

---

**9**

## Index

---



# Getting Started

---

- “Product Description” on page 1-2
- “Product Capabilities” on page 1-3
- “Install Required Components” on page 1-8
- “Vehicle Network Communication in MATLAB” on page 1-12
- “Vehicle Network Communication Examples” on page 1-15
- “Accessing the Toolbox” on page 1-34

## Product Description

### Communicate with in-vehicle networks using CAN protocol

Vehicle Network Toolbox™ provides connectivity to CAN devices from MATLAB® and Simulink®. The toolbox provides functions for encoding, decoding, and filtering CAN messages, enabling you to transfer messages between the MATLAB workspace and a CAN bus. You can use the toolbox to run test sequences that log and record CAN messages for later analysis. You can also use it to connect with industry-standard CAN database files and replay recorded sequences of messages. The toolbox includes Simulink blocks that enable you to simulate message traffic on a CAN bus or connect models to a live network. Vehicle Network Toolbox supports CAN interface devices from Vector, Kvaser and National Instruments®.

### Key Features

- MATLAB functions for transmitting and receiving CAN packets
- CAN communication blocks for connecting a CAN bus to a Simulink model
- Bit packing and unpacking functions and blocks for simplified encoding and decoding of CAN messages
- Message traffic GUI for visualizing live CAN network traffic
- Message filtering and logging
- Support for Vector and Kvaser interface hardware and Vector CAN database (.dbc) files

# Product Capabilities

**In this section...**

“Vehicle Network Toolbox Characteristics” on page 1-3

“Interaction Between the Toolbox and Its Components” on page 1-4

“Expected Background ” on page 1-6

“Related Products” on page 1-6

## Vehicle Network Toolbox Characteristics

The toolbox is a collection of functions built on the MATLAB technical computing environment.

You can use the toolbox to:

- “Connect to CAN Devices” on page 1-3
- “Use Supported CAN Devices and Drivers” on page 1-4
- “Communicate Between MATLAB and CAN Bus” on page 1-4
- “Simulate CAN Communication” on page 1-4
- “Visualize CAN Communication” on page 1-4

## Connect to CAN Devices

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.

## **Use Supported CAN Devices and Drivers**

You can use Vehicle Network Toolbox to communicate over the CAN bus using supported Vector, Kvaser, or National Instruments devices and drivers.

See “Supported Hardware” on page 1-10 for more information.

## **Communicate Between MATLAB and CAN Bus**

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.

## **Simulate CAN Communication**

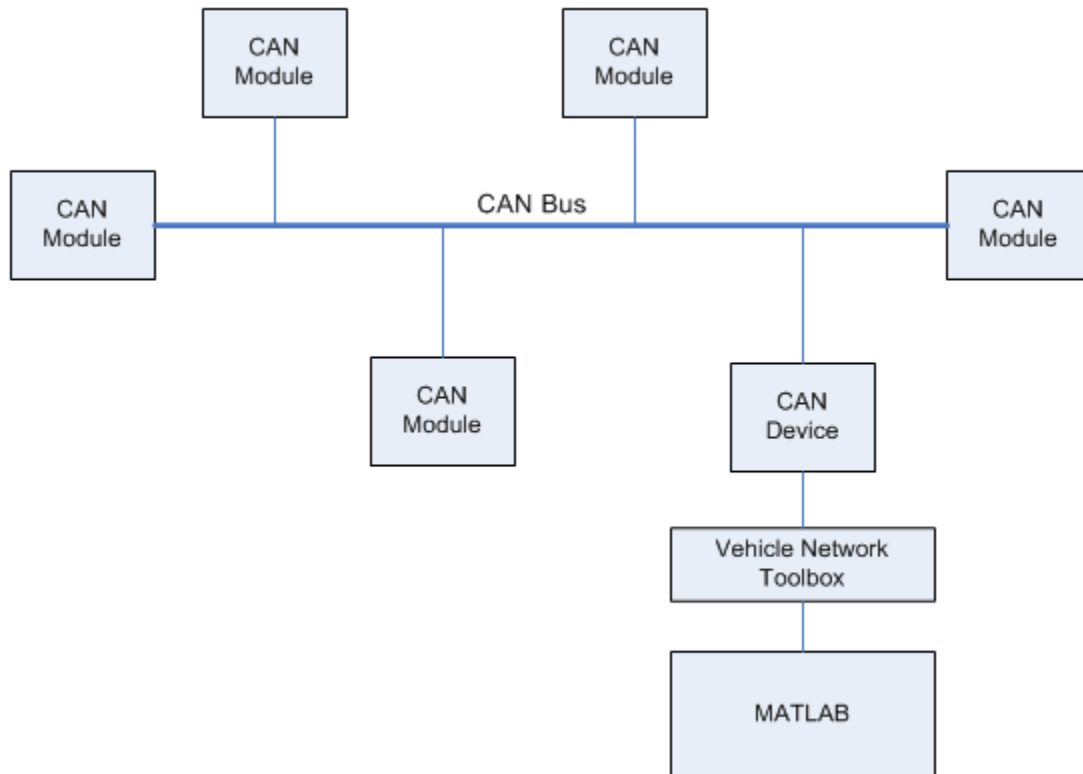
With Vehicle Network Toolbox block library and other blocks from the Simulink library, you can create sophisticated models to connect to a live network and to simulate message traffic on a CAN bus.

## **Visualize CAN Communication**

Using CAN Tool, a 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**

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 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.

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**

MathWorks provides several products that are relevant to the kinds of tasks you can perform with 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.

## Install Required Components

In this section...
“Required Components” on page 1-8
“Install Devices and Drivers” on page 1-8
“Install the Toolbox” on page 1-10
“Supported Hardware” on page 1-10

### Required Components

To communicate on CAN networks from the MATLAB workspace and using Simulink models, install these components:

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

---

**Note** To use the Vehicle Network Toolbox block library, install the current Simulink version.

---

### Install Devices and Drivers

- “Vector Hardware Devices and Drivers” on page 1-8
- “Kvaser Hardware Devices and Drivers” on page 1-9
- “National Instruments Devices and Drivers” on page 1-10

### Vector Hardware Devices and Drivers

You need the latest version of the driver for your device to use with Windows<sup>®</sup> XP, Windows Vista<sup>™</sup>, or Windows 7.

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](https://www.vector-worldwide.com/va_downloadcenter_us.html).

Drivers for 32-bit and 64-bit Windows are available in separate packages. Download and install the latest version and check the appropriate box during the installation for the hardware you want to use. Virtual channels are also installed with the hardware drivers.

**XL Driver Library.** Download and install the latest version of the XL Driver Library from the Vector Web site. A single package contains the driver library for 32-bit and 64-bit Windows. After you install the library:

- If you have a 32-bit system, copy the file `vx1api.dll` from the installation folder to the `windows root\system32` folder.
- If you have a 64-bit system, copy the file `vx1api64.dll` from the installation folder to the `windows root\system32` folder.
- If you have a 64-bit system and need to run the 32-bit version of Vehicle Network Toolbox software. Copy the `vx1api.dll` from the installation folder to the `windows root\syswow64` folder.

## **Kvaser Hardware Devices and Drivers**

You need the latest version of the driver for your device to use with Windows XP, Windows Vista, or Windows 7. 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>

Drivers for 32-bit and 64-bit Windows are available in the same package. Virtual channels are also installed with the hardware drivers.

## National Instruments Devices and Drivers

You need the latest version of the driver for your device to use with Windows XP, Windows Vista, or Windows 7. Refer to your National Instruments device documentation for hardware installation instructions.

Install the NI CAN programming library from the National Instruments Web site. This will install both the drivers and the driver libraries.

Drivers for 32-bit and 64-bit Windows are available in the same package. Virtual channels are also installed with the hardware drivers.

---

**Note** You can use National Instruments only on a 32-bit system or a 32-bit MATLAB installed on a 64-bit system.

---

## Install 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 display 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 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

Vehicle Network Toolbox supports Vector, Kvaser, and National Instruments CAN 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.

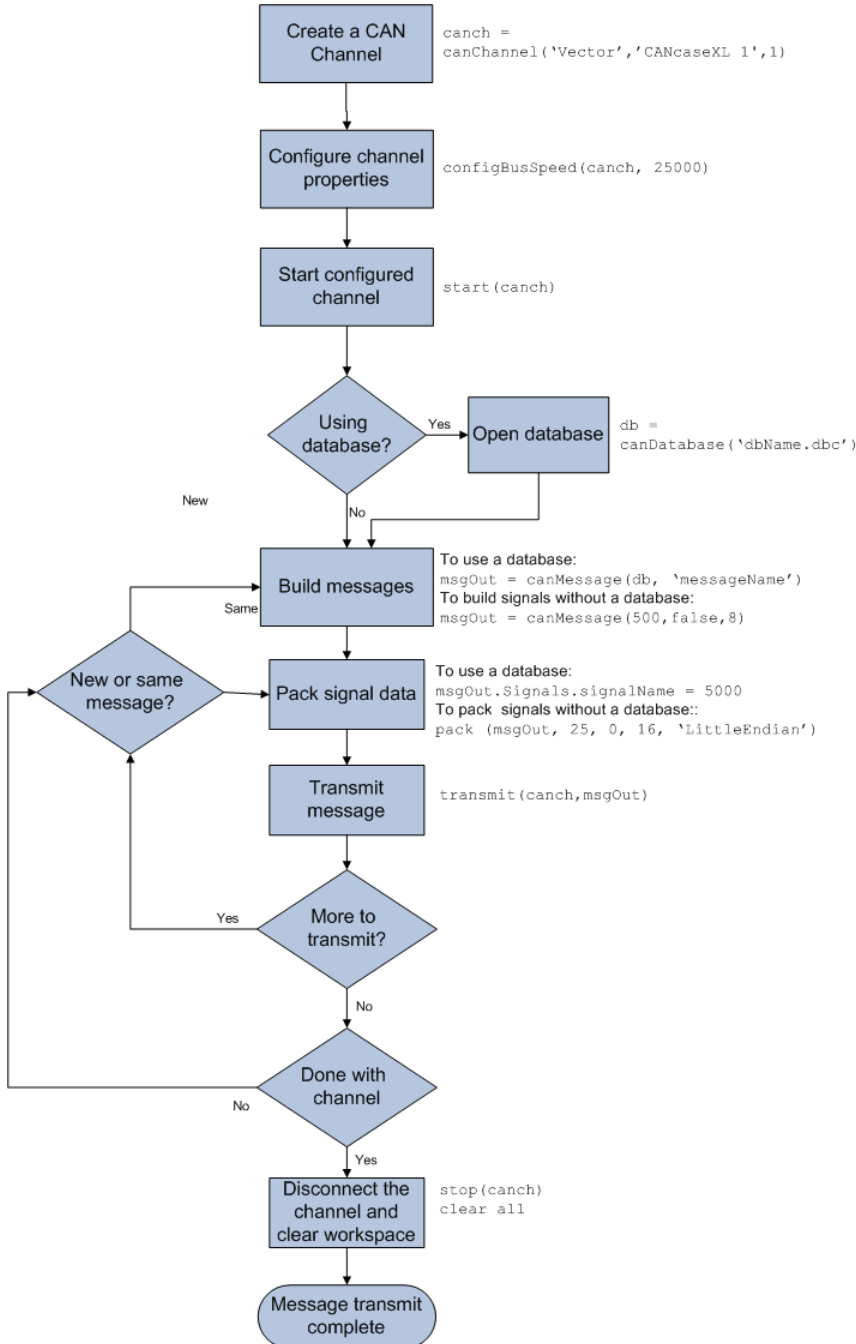
## Supported National Instruments Devices

For a complete list of supported NI CAN devices, see the Supported Hardware page

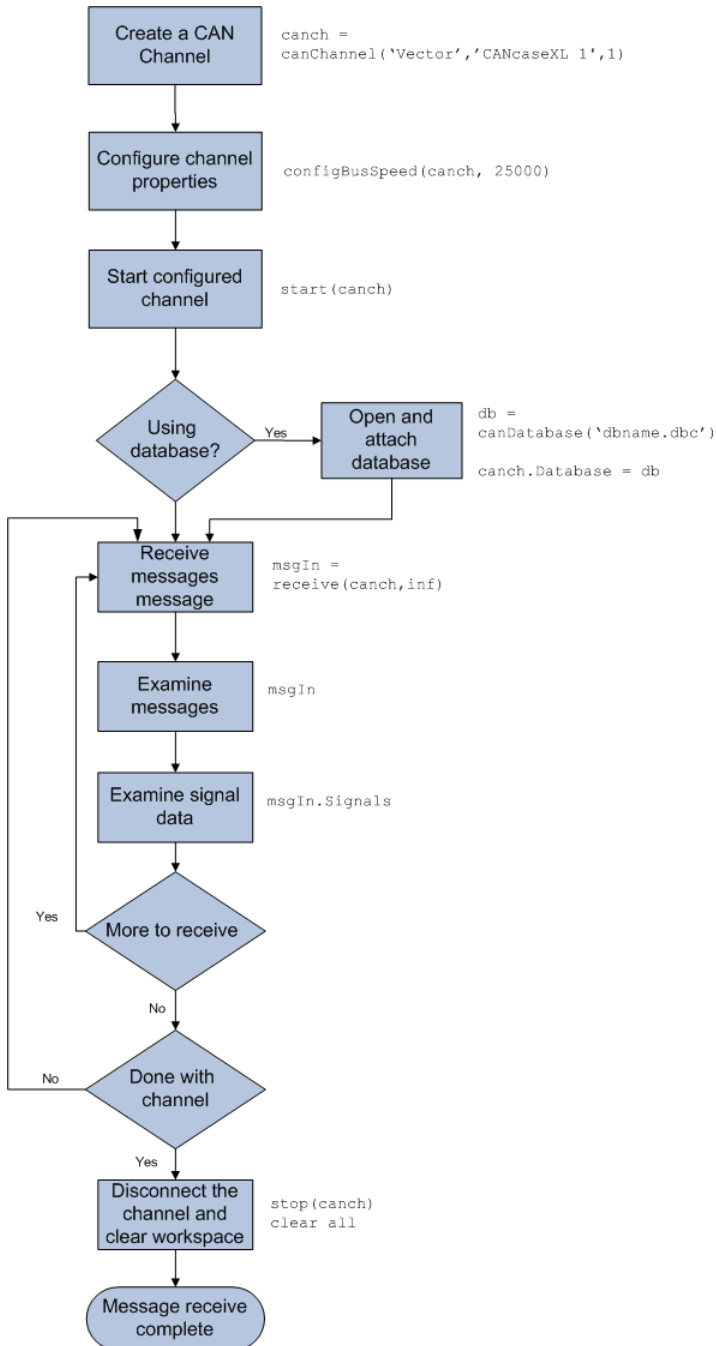
## **Vehicle Network Communication in MATLAB**

Workflows in this section are sequential and will help you understand how the communication works. You can also see code snippets and map them to the examples in the next section

## Transmit Workflow



## Receive Workflow





# Vehicle Network Communication Examples

## In this section...

“Prerequisites” on page 1-15  
“Discover Installed Hardware” on page 1-16  
“Create a CAN Channel” on page 1-17  
“Configure Properties” on page 1-18  
“Start the Channel” on page 1-19  
“Create a Message” on page 1-20  
“Pack a Message” on page 1-21  
“Transmit a Message” on page 1-22  
“Receive a Message” on page 1-23  
“Unpack a Message” on page 1-25  
“Save and Load a CAN Channel” on page 1-25  
“Filter Messages” on page 1-26  
“Multiplex Signals” on page 1-27  
“Configure Silent Mode” on page 1-30  
“Disconnect Channels and Clean Up” on page 1-31

## Prerequisites

Examples follow a sequential workflow for configuring CAN communications. Use these examples sequentially in the MATLAB Command Window.

In the example, create two CAN channels using `canChannel`, and `canHWInfo` to obtain information about the devices installed on your system. Edit the properties of the first channel and create a message using `canMessage`. Transmit the message from the first channel using `transmit`, and receive it on the other using `receive`.

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.

## Discover Installed Hardware

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

```
info = canHWInfo
```

MATLAB displays the following information:

CAN Devices Detected

Vendor	Device	Channel	Serial Number	Constructor
Kvaser	Virtual 1	1	0	canChannel('Kvaser', 'Virtual 1', 1)
Kvaser	Virtual 1	2	0	canChannel('Kvaser', 'Virtual 1', 2)
NI	Virtual (CAN256)	1	0	canChannel('NI', 'CAN256')
NI	Virtual (CAN257)	2	0	canChannel('NI', 'CAN257')
NI	Series 847X Sync USB (CAN0)	1	14E1B6E	canChannel('NI', 'CAN0')
NI	Series 847X Sync USB (CAN1)	1	14E1B68	canChannel('NI', 'CAN1')
Vector	CANcaseXL 1	1	24365	canChannel('Vector', 'CANcaseXL 1', 1)
Vector	CANcaseXL 1	2	24365	canChannel('Vector', 'CANcaseXL 1', 2)
Vector	Virtual 1	1	0	canChannel('Vector', 'Virtual 1', 1)
Vector	Virtual 1	2	0	canChannel('Vector', '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)'
```

## Create a CAN Channel

---

**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: Standard ID Filter: Allow All | Extended ID Filter: Allow All.
```

**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.

## Configure 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.

## Start the 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: Standard ID Filter: Allow All | Extended ID Filter: Allow All.
```

## Create a Message

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.

## Pack 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.

## Transmit 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: Standard ID Filter: Allow All | Extended ID Filter: Allow All.

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.

## Receive 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.

## Unpack 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.

## Save and Load a CAN Channel

- “Saving a CAN Channel Object to a MATLAB File” on page 1-25
- “Load a Saved Channel” on page 1-25

### 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
```

### Load a Saved 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.

## Filter Messages

You can set up filters on your channel to accept messagees 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:

- `filterAllowAll`
- `filterBlockAll`
- `filterAlowOnly`

To specify message names you want to filter, create a CAN channel and attach a database to the channel:

```
canch1=(Vector', 'CANcaseXL 1',1);  
canch1.Database=('demoVNT_CANdbFiles.dbc');
```

Set a filter for the message `EngineMsg` and display the channel:

```
filterAllowOnly(canch1, 'EngineMsg');
```

```
canch1
```

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

```
Channel Parameters: Bus Speed is 500000.  
                    Bus Status is 'N/A'.  
                    Transceiver name is ''.  
                    Serial Number of this device is 0.  
                    Initialization access is allowed.  
                    'demoVNT_CANdbFiles.dbc' 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: Standard ID Filter: Allow Only | Extended ID Filter: Allow All
```

If you start the channel and receive messages, you should now only see the `EngineMsg` pass through the filter.

## Multiplex Signals

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.

## Configure 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
.
.
.
```

## Disconnect Channels and Clean Up

- “Disconnecting the Configured Channel” on page 1-31
- “Clean Up the MATLAB Workspace” on page 1-32

### 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.
```

## Clean 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...
“Explore the Toolbox” on page 1-34
“Get Help” on page 1-34
“View Examples” on page 1-34

### Explore the Toolbox

You can access 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
```

### Get 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
```

### View 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 CAN Database Support” on page 2-2
- “Load and Create Messages With a .dbc File” on page 2-3
- “Other Uses of the CAN Database” on page 2-17

### **Vector CAN Database Support**

A CAN database contains physical message and signal definitions. Using a CAN database, you can represent message and signal information in engineering units and do not need to manipulate raw data bytes. You can use a Vector CAN database with Vehicle Network Toolbox. A .dbc file contains definitions of CAN messages and signals.

Look up message and signal information and build messages using the information defined in the database file.

## Load and Create Messages With a .dbc File

### In this section...

“Load the CAN Database” on page 2-3

“Create a CAN Message” on page 2-4

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

“Add a Database to a CAN Channel” on page 2-5

“Update Database Information” on page 2-5

“Create and Process Messages Using Database Definitions” on page 2-5

### Load the CAN Database

To use a CAN database 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 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.

### Create 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')
```

### Access 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

## Add 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.

## Update 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.

## Create and Process Messages Using Database Definitions

This example shows you how to create, receive and process messages using information stored in CAN database files. This example uses the CAN database file, `|demoVNT_CANdbFiles.dbc|`.

Copyright 2008-2011, The MathWorks, Inc.

### Open the Database File

Open the database file and examine the `Messages` property to see the names of all message defined in this database.

```
db = canDatabase('demoVNT_CANdbFiles.dbc')
db.Messages
```

```
db =
```

```
can.Database handle
Package: can
```

```
Properties:
  Name: 'demoVNT_CANdbFiles'
  Path: 'C:\Work\2011.10.20_Sandbox\matlab\toolbox\vnt\vntdemos\demoV'
  Messages: {5x1 cell}
  UserData: []
```

```
ans =

    'DoorControlMsg'
    'EngineMsg'
    'SunroofControlMsg'
    'TransmissionMsg'
    'WindowControlMsg'
```

### View Message Information

Use `messageInfo` to view message information, including the identifier, data length, and a signal list.

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

```
ans =

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

You can also query for information on all messages at once.

```
messageInfo(db)
```

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

### View Signal Information

Use `signalInfo` to view signal definition information, including type, byte ordering, size, and scaling values that translate raw signals to physical values.

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

```
ans =  
  
    Name: 'EngineRPM'  
    StartBit: 0  
    SignalSize: 32  
    ByteOrder: 'LittleEndian'  
    Signed: 0  
    ValueType: 'Integer'  
    Class: 'uint32'  
    Factor: 0.1000  
    Offset: 250  
    Minimum: 250  
    Maximum: 9500  
    Units: 'rpm'  
    Comment: ''  
    Multiplexor: 0  
    Multiplexed: 0  
    MultiplexMode: 0
```

You can also query for information on all signals in the message at once.

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

```
ans =
```

```
2x1 struct array with fields:
```

```
Name  
StartBit  
SignalSize  
ByteOrder  
Signed  
ValueType  
Class  
Factor  
Offset  
Minimum  
Maximum  
Units  
Comment  
Multiplexor  
Multiplexed  
MultiplexMode
```

### Create a Message Using Database Definitions

Specify the name of the message when you create a new message to have the database definition applied. CAN signals in this messages are represented in engineering units in addition to the raw data bytes.

```
msgEngineInfo = canMessage(db, 'EngineMsg')
```

```
msgEngineInfo =
```

```
can.Message handle  
Package: can
```

```
Properties:  
    ID: 100
```

```
Extended: 0
  Name: 'EngineMsg'
Database: [1x1 can.Database]
  Error: 0
  Remote: 0
Timestamp: 0
  Data: [0 0 0 0 0 0 0 0]
  Signals: [1x1 struct]
UserData: []
```

### View Signal Information

Use the `Signals` property to see signal values for this message. You can directly write to and read from these signals to pack or unpack data from the message.

```
msgEngineInfo.Signals
```

```
ans =
```

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

### Change Signal Information

Write directly to the signal to change a value and read its current value back.

```
msgEngineInfo.Signals.EngineRPM = 5500.25
msgEngineInfo.Signals
```

```
msgEngineInfo =
```

```
can.Message handle
Package: can
```

```
Properties:
  ID: 100
  Extended: 0
  Name: 'EngineMsg'
  Database: [1x1 can.Database]
  Error: 0
  Remote: 0
  Timestamp: 0
  Data: [23 205 0 0 0 0 0 0]
  Signals: [1x1 struct]
  UserData: []
```

```
ans =
```

```
VehicleSpeed: 0
EngineRPM: 5.5003e+03
```

When you write directly to the signal, the value is translated, scaled, and packed into the message data using the database definition.

```
msgEngineInfo.Signals.VehicleSpeed = 70.81
msgEngineInfo.Signals
```

```
msgEngineInfo =
```

```
can.Message handle
Package: can
```

```
Properties:
  ID: 100
  Extended: 0
  Name: 'EngineMsg'
  Database: [1x1 can.Database]
  Error: 0
  Remote: 0
  Timestamp: 0
```

```
Data: [23 205 0 0 71 0 0 0]
Signals: [1x1 struct]
UserData: []
```

```
ans =
```

```
VehicleSpeed: 71
EngineRPM: 5.5003e+03
```

### Receive Messages with Database Information

Attach a database to a CAN channel that receives messages to apply database definitions to incoming messages automatically. The database decodes only messages that are defined. All other messages are received in their raw form.

```
rxCh = canChannel('Vector', 'Virtual 1', 2);
rxCh.Database = db
```

```
rxCh =
```

```
Summary of CAN Channel using 'Vector' 'Virtual 1' Channel 2.
```

```
Channel Parameters: Bus Speed is 500000.
                   Bus Status is 'N/A'.
                   Transceiver name is ''.
                   Serial Number of this device is 0.
                   Initialization access is allowed.
                   'demoVNT_CANdbFiles.dbc' 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: Standard ID Filter: Allow All | Extended ID Filter:
```

### Receive Messages

Start the channel, generate some message traffic and receive messages with physical message decoding.

```
start(rxCh);  
generateMsgsDb();  
rxMsg = receive(rxCh, Inf)
```

```
rxMsg =
```

```
1x413 can.Message handle  
Package: can
```

```
Properties:
```

```
  ID  
  Extended  
  Name  
  Database  
  Error  
  Remote  
  Timestamp  
  Data  
  Signals  
  UserData
```

Stop the channel and clear it from the workspace.

```
stop(rxCh);  
clear rxCh
```

### Examine a Received Message

Inspect a received message to see the applied database decoding.

```
rxMsg(10)
```



```
rxMsg(10).Signals
```

```
ans =
```

```
can.Message handle  
Package: can
```

```
Properties:
```

```
    ID: 100  
Extended: 0  
    Name: 'EngineMsg'  
Database: [1x1 can.Database]  
    Error: 0  
    Remote: 0  
Timestamp: 0.1370  
    Data: [34 131 0 0 52 0 0 0]  
    Signals: [1x1 struct]  
    UserData: []
```

```
ans =
```

```
    VehicleSpeed: 52  
    EngineRPM: 3607
```

### **Extract Most Recent Message by Name**

Use `extractRecent` and specify a message name to extract the most recent occurrence of a message.

```
msgRecentWindows = extractRecent(rxMsg, 'WindowControlMsg')
```

```
msgRecentWindows =
```

```
can.Message handle  
Package: can
```

```
Properties:
  ID: 600
  Extended: 0
  Name: 'WindowControlMsg'
  Database: [1x1 can.Database]
  Error: 0
  Remote: 0
  Timestamp: 5.5373
  Data: [62 68 0 0]
  Signals: [1x1 struct]
  UserData: []
```

### **Extract All Instances of a Specified Message by Name**

Use `extractAll` and specify a message name to extract all instances of a specified message.

```
allMsgEngine = extractAll(rxMsg, 'EngineMsg')
```

```
allMsgEngine =

1x222 can.Message handle
Package: can
```

```
Properties:
  ID
  Extended
  Name
  Database
  Error
  Remote
  Timestamp
  Data
  Signals
  UserData
```

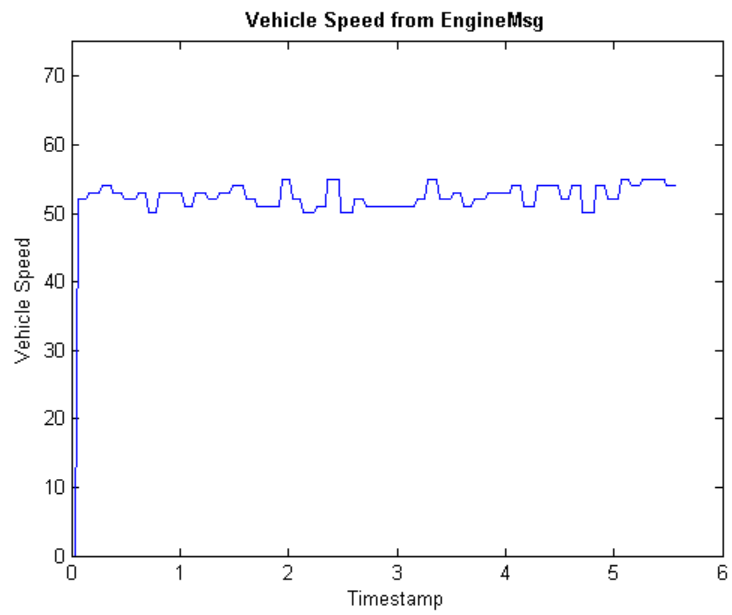
### Plot Physical Signal Values

Plot the values of database decoded signals over time. Reference the message timestamps and the signal values in variables.

```
signals = [allMsgEngine.Signals]
plot([allMsgEngine.Timestamp], [signals.VehicleSpeed])
title('Vehicle Speed from EngineMsg', 'FontWeight', 'bold')
xlabel('Timestamp')
ylabel('Vehicle Speed')
axis([0 6 0 75])
```

```
signals =
```

```
1x222 struct array with fields:
    VehicleSpeed
    EngineRPM
```



## Other Uses of the CAN Database

### In this section...

“View Message Information in a CAN Database” on page 2-17

“View Signal Information in a CAN Message” on page 2-18

“Attach a CAN Database to Existing Messages” on page 2-19

### View Message Information in a CAN Database

You can look up information on message definitions by a single message by name, or a single message by ID. You can also look up information on all message definitions in the database by typing:

```
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 on 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.

### View Signal Information in a CAN Message

You can get signal definition information on a specific signal or all signals in a CAN message with database definitions attached. 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.

You can also access the `Signals` property of the message to view physical signal information. When you create physical signals using database information, you can directly write to and read from these signals to pack or unpack data from the message. When you write directly to the signal name, the value is translated, scaled, and packed into the message data.

## Attach 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 3-9

# CAN Tool

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

## About the CAN Tool

Vehicle Network Toolbox provides a graphical user interface that monitors CAN message traffic on selected channels. Using the CAN Tool you can:

- view live CAN message data.
- Configure connection to the CAN bus.
- View unique messages.
- Attache a database to view signal information.
- Save the messages to a log file.

You can

## Opening the CAN Tool

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

## CAN Tool Fields

The tool has the following menus, buttons and table.

Timestamp	ID	Name	Length	Data
40.746557	1F4		4	95 50 E7 E5
40.744624	1F4		4	39 5E 2C C4
40.743608	1F4		4	4B 3D 62 43
40.742576	1F4		4	EC A2 0E FD
40.741560	1F4		4	EA 70 3A 9B
40.740569	2EE		8	C8 29 EF 11 61 B0 0...
40.739570	1F4		4	E0 27 2F ED
40.738439	1F4		4	FE 03 75 85
40.737407	1F4		4	7D CA 41 9E
40.736375	1F4		4	E9 83 44 57
40.735621	2EE		8	D9 3D 1E A5 1B 49 1...
40.734302	1F4		4	E8 89 DB 85
40.733262	1F4		4	3A 7F 24 40
40.732557	1F4		4	B8 15 ED 09
40.731558	1F4		4	2B D9 A6 84
40.730239	2EE		8	ED DE 44 82 7D 0B 6...
40.729133	1F4		4	7E 70 A2 27
40.728560	1F4		4	AA 8A 2D 65
40.727060	1F4		4	05 99 55 1D
40.725963	1F4		4	AC BD 62 A8
40.725004	2EE		8	4D 45 A7 C3 E8 7A ...
40.723898	1F4		4	1B 1C 04 6C
40.722866	1F4		4	19 45 72 92
40.721826	1F4		4	B0 84 32 D6
40.720810	1F4		4	89 03 9A 3B
40.719819	2EE		8	BA 1B FA 6A CB 6E ...
40.718705	1F4		4	82 42 93 77
40.717672	1F4		4	1A 52 BA 5D

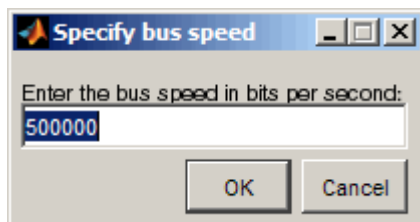
## File Menu

- **Save Messages** — Saves messages to a log file.

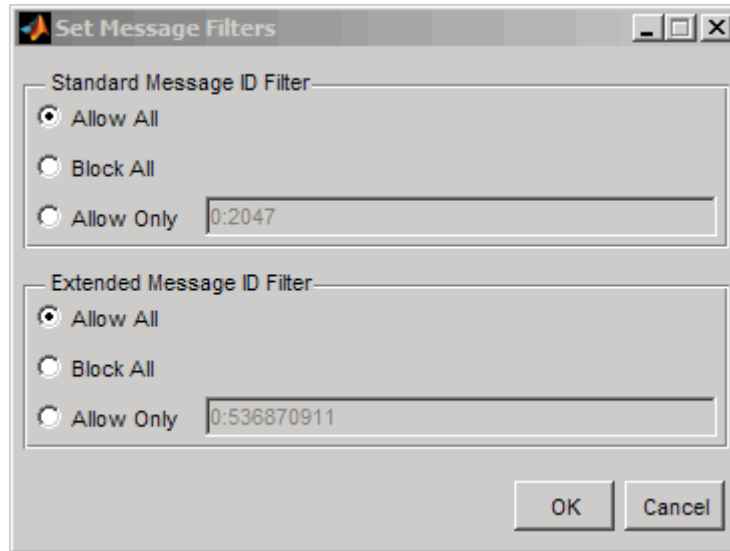
- **Clear Messages** — Clears messages in the CAN Tool window.
- **Exit** — Click to exit the CAN Tool window.

### Configure Menu

- **Channel** — Displays all available CAN devices and channels on your system. Select the CAN channel to monitor.
- **Bus Speed** — Opens the Specified bus speed dialog box. To change the bus speed of the selected channel, type the new value in bits per second in the box.



- **Message Filters** — Opens the Set Message Filters dialog box. Select an option in the dialog box to configure hardware filters to block or allow messages.



- **Standard Message ID Filter**
  - Allow All — Select to allow all standard ID messages.
  - Block All — Select to block all standard ID messages.
  - Allow only — Select to set up custom filtering of messages. Type the standard message IDs that you want to allow.
- **Extended Message ID Filter**
  - Allow All — Select to allow all extended ID messages.
  - Block All — Select to block all extended ID messages.
  - Allow only — Select to set up custom filtering of messages. Type the extended message IDs that you want to allow.
- **Database** — Selects the database to attach to the CAN messages on the selected channel.

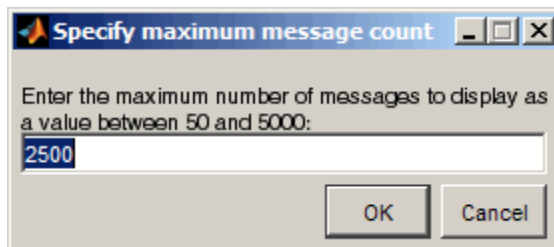
## Run Menu

- **Start** — Click to view message activity on the selected channel.

- **Pause** — Click to pause the display of message activity on the selected channel.
- **Stop** — Click to stop the display of message activity on the selected channel.

### View Menu

- **Maximum message count** — Opens the Specify maximum message count dialog box. To change the maximum number of messages displayed at a time in the CAN Tool, type the new value in the box.



- **Number Format** — Select the number format to display message identifier data. Choose **Hexadecimal** or **Decimal**.
- **Show Unique Messages** — Select this option to display the most recent instance of each message received on the selected channel. If you select this option, the tool displays a simplified version of the message traffic. In this view, messages do not 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.

### Help Menu

- **canTool Help** — Select this option to see the online help for CAN Tool.
- **About Vehicle Network Toolbox** — Select this option to view the toolbox version and release information.

## Buttons

**Start** 

Displays message activity on the selected channel.

**Pause** 

Pauses the display of message activity on the selected channel.

**Stop** 


Stops displaying messages on the selected channel.

**Save messages** 


Click this button to save the current message list on the selected channel to a file.


**Clear messages** 

Click this button to clear messages in the CAN Tool window.


**Show unique messages** 

Select this option to display the most recent instance of each message received on the selected channel. If you select this option, the tool displays a simplified version of the message traffic. In this view, messages do not 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.

**Docking** 

Click this button to dock the CAN Tool to the MATLAB desktop. To undock, click this button .

**Undocking** 

Click this button to undock the CAN Tool from the MATLAB desktop. To dock, click this button. .

### Message Table

#### Timestamp

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

#### ID

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

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

To change the format to decimal, select **View > Number Format > Decimal**.

#### Name

Displays the name of the message, if available.

#### Length

Displays the length of the message in bytes.

#### Data

Displays the data in the message in hexadecimal format.

To change the format to decimal, select **View > Number Format > Decimal**.

If you are using a database, click the + sign to see signal information. The tool displays the signal name and the physical value of the message, as defined in the attached database.



## Using the CAN Tool

### In this section...

- “Viewing Messages on a Channel” on page 3-9
- “Configuring the Channel Bus Speed” on page 3-9
- “Filtering CAN Messages in the CAN Tool” on page 3-10
- “Attaching a Database” on page 3-11
- “Changing the Message Count” on page 3-13
- “Changing the Number Format” on page 3-13
- “Viewing Unique Messages” on page 3-13
- “Saving the Message Log File” on page 3-14

## Viewing Messages on a Channel

- 1** Open the CAN Tool and select the device and channel connected to your CAN bus by selecting **Configure > Channel**.
- 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**, or select **Run > Start**.
- 4** To pause the display, click **Pause** or select **Run > Pause**.
- 5** To stop the display, click **Stop** or select **Run > Stop**.

## Configuring the Channel Bus Speed

Configure the bus speed when your network speed 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** Select **Configure > Bus Speed**.

**2** Type the desired value in the Specify bus speed dialog box.

**3** Click **OK**.

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.

### Filtering CAN Messages in the CAN Tool

Filter CAN messages to allow or block messages displayed in the CAN Tool.

To set up filters:

- 1** Select **Configure > Message Filters**.
- 2** To set filters on standard message IDs, select:
  - a** Allow All to set the hardware filter to allow all messages with standard IDs.
  - b** Block All to set the hardware filter to block all messages with standard IDs.
  - c** Allow Only to set up custom filters. Type the standard IDs of the message you want to filter. You can type a range or single IDs. The default is 0:2047.
- 3** To set filters on extended message IDs, select:
  - a** Allow All to set the hardware filter to allow all messages extended IDs.
  - b** Block All to set the hardware filter to block all messages extended IDs.
  - c** Allow Only to set up custom filters. Type the extended IDs of the message you want to filter. You can type a range or single IDs. The default is 0:536870911.

---

**Note** If you are using a custom filter, change the default range to the desired range. The default range allows all messages and you should select Allow All to allow all incoming messages with extended IDs.

---

**4** Click OK.


## **Attaching a Database**

Attach a database to the CAN Tool to see signal information of the displayed messages.

- 1** Select **Run > Stop** to stop the message display in the CAN Tool.
- 2** Select **Configure > Database**.
- 3** Select the database to attach and start the message display again.

When the tool displays the messages, it shows the message name in the Message table.

### 3 Monitoring CAN Message Traffic

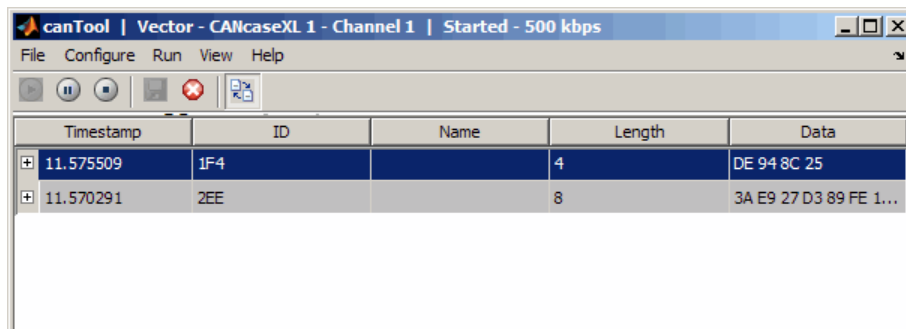


The screenshot shows the canTool interface with a list of CAN messages. The first message is selected, and its details are visible in the table below.

Timestamp	ID	Name	Length	Data
4.987068	1F4	TestMessage	4	54 6C 45 32
4.986069	1F4	TestMessage	4	AE B4 71 05
4.985070	1F4	TestMessage	4	AF 5C BC 65
4.983955	1F4	TestMessage	4	0B C1 3E 71
4.982890	1F4	TestMessage	4	59 26 95 43
4.981916	2EE		8	86 87 DC 7C 64 AB ...
4.980810	1F4	TestMessage	4	31 23 B2 18
4.979761	1F4	TestMessage	4	A0 C5 EE F8
4.978688	1F4	TestMessage	4	67 72 5D C3
4.977656	1F4	TestMessage	4	38 1B 1C 10
4.976542	2EE		8	7A A3 8B A5 8B B8 8...
4.975444	1F4	TestMessage	4	51 1E F0 A5
4.974363	1F4	TestMessage	4	94 8A DE 44
4.974068	1F4	TestMessage	4	D7 D4 41 9C
4.972282	1F4	TestMessage	4	C4 59 A9 6A
4.971323	2EE		8	35 B5 3C 1E 9B 73 7...
4.970226	1F4	TestMessage	4	A5 AD A2 F1
4.969202	1F4	TestMessage	4	AE 8B 6D A4
4.968161	1F4	TestMessage	4	30 49 17 93
4.967129	1F4	TestMessage	4	3E EA 45 C3
4.966171	2EE		8	2D B8 79 27 57 9B 3...
4.965073	1F4	TestMessage	4	52 C8 78 09
4.964065	1F4	TestMessage	4	76 6C 76 C4
4.963009	1F4	TestMessage	4	9A 63 EA 00
4.962066	1F4	TestMessage	4	58 C7 AC 02
4.961010	2EE		8	4A AB B1 11 41 39 A...
4.959896	1F4	TestMessage	4	A3 F4 3D AC
4.959068	1F4	TestMessage	4	75 DF 84 F1

4 Click the plus (+) sign to see the details of the message.






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

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

To save messages when **Show Unique Messages** is selected, click **Pause** and then click **Save**. You cannot save just the unique message list. This operation saves the complete message log in the window.

## Saving the Message Log File

You can save a log file of the messages currently displayed while the tool is running. You need not stop or pause the display to save a log file.

To save a log file of the messages currently displayed in the window, select **File > Save Messages** or click .

The tool saves the messages in a MATLAB file in your current working folder by default. You can change the location by browsing to a different folder in the Save dialog box.

Each time you save the message log to a file, the CAN Tool saves them as VNT CAN Log.mat with sequential numbering by default. You can change the name by typing a new name in Save dialog box.

# Communications in Simulink

---

- “Overview” on page 4-2
- “CAN Communication in Simulink” on page 4-3
- “Opening the Vehicle Network Toolbox Block Library” on page 4-7
- “Building Simulink Models to Transmit and Receive Messages” on page 4-10
- “Create Custom Blocks” on page 4-25

## 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 Log**— Logs messages to file.
- **CAN Pack** — Pack signals into a CAN message.
- **CAN Receive** — Receive CAN messages from a CAN bus.
- **CAN Replay**— Replays logged messages to CAN bus or output port.
- **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.

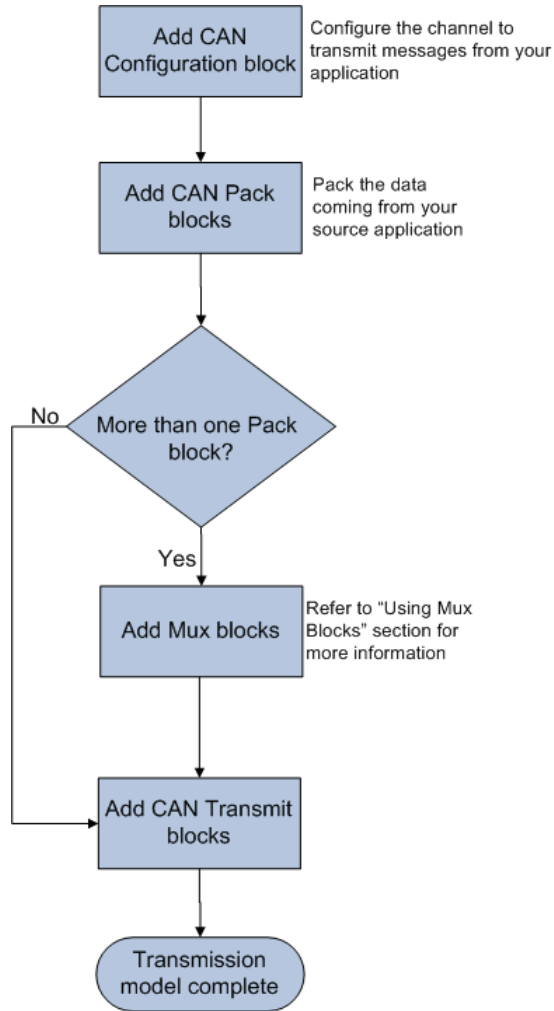
For more detailed information about the blocks in the Vehicle Network Toolbox block library, see Block Reference.



## CAN Communication in Simulink

In this section...
“Message Transmission Workflow” on page 4-4
“Message Reception Workflow” on page 4-5

## Message Transmission Workflow

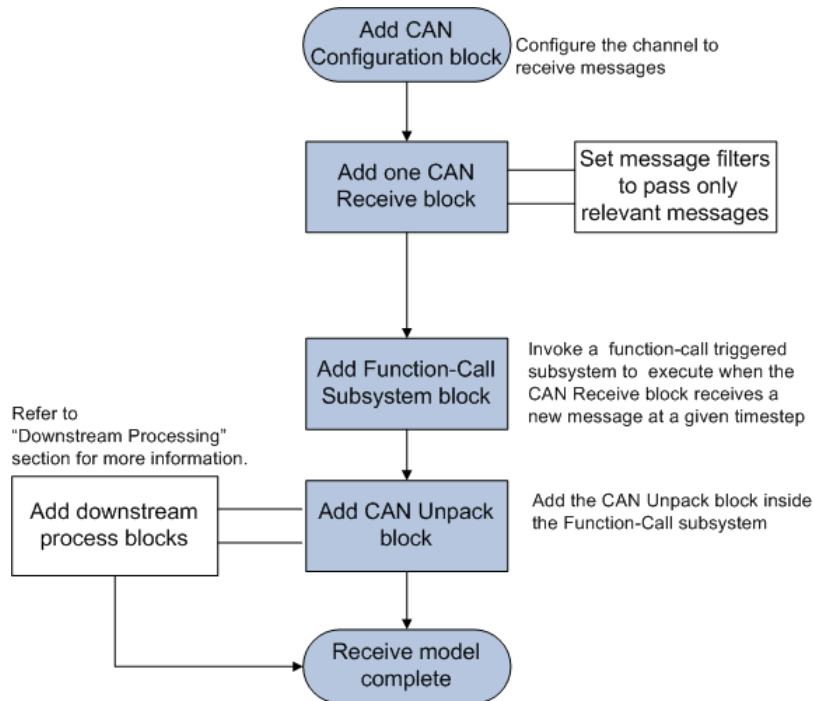


This workflow represents the most common CAN Transmit model. Adjust your model as needed. For more workflow examples, see “Building Simulink Models to Transmit and Receive Messages” on page 4-10 and the Vehicle Network Toolbox demos.

## Using Mux Blocks

- Use a Mux block to combine every message from the source if they are transmitted at the same rate.
- Use one CAN Transmit block for each configured Mux block.

## Message Reception Workflow



For workflow examples, see “Building Simulink Models to Transmit and Receive Messages” on page 4-10 and the Vehicle Network Toolbox demos.

## Message Filtering

Set up filters to process only relevant messages. This ensures optimal simulation performance.

Do not set up filters if you need to parse all bus communications.

### **Function-Call Triggered Message Processing**

Set up your CAN Unpack block:

- In a function-call triggered subsystem if you want to unpack every message received by your CAN Receive block.
- Without a function-call triggered subsystem if you want to unpack only the most recent message received by your CAN Receive block.  
Set up this system if your receive block is filtering for a single message.

### **Downstream Processing**

For any downstream processing using received messages, include blocks:

- Within the function-call subsystem if your downstream process must respond to all messages received in a single timestep in this model.
- Outside the function-call subsystem if your downstream process responds only to the most recent message received in a given timestep in this model. In this case, the CAN Unpack block will not respond to any other messages received, irrespective of the messages ID.

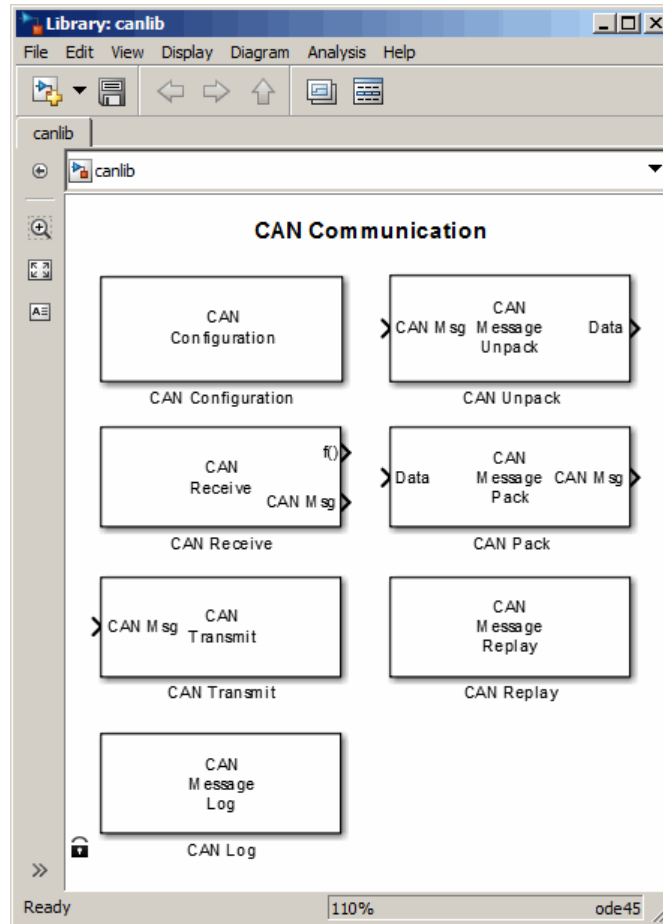
## Opening the Vehicle Network Toolbox Block Library

In this section...
“Using the MATLAB Command Window” on page 4-7
“Using the Simulink Library Browser” on page 4-8

### 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.

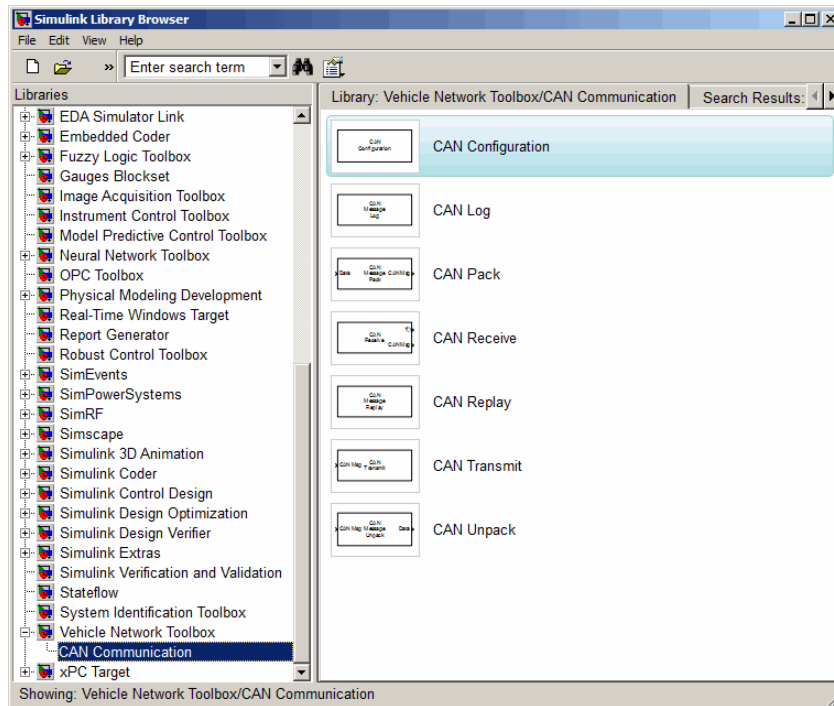


### Using the Simulink Library Browser

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` in the MATLAB Command 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 4-10
“Build a Message Receive Model” on page 4-15
“Save and Run the Model” on page 4-21

### 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 4-15 and “Save and Run the Model” on page 4-21 to build your complete model and run the simulation.

- “Step 1: Open the Block Library” on page 4-10
- “Step 2: Create a New Model” on page 4-11
- “Step 3: Drag Vehicle Network Toolbox Blocks into the Model” on page 4-12
- “Step 4: Drag Other Blocks to Complete the Model” on page 4-12
- “Step 5: Connect the Blocks” on page 4-13
- “Step 6: Specify the Block Parameter Values” on page 4-13

#### Step 1: Open the Block Library

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



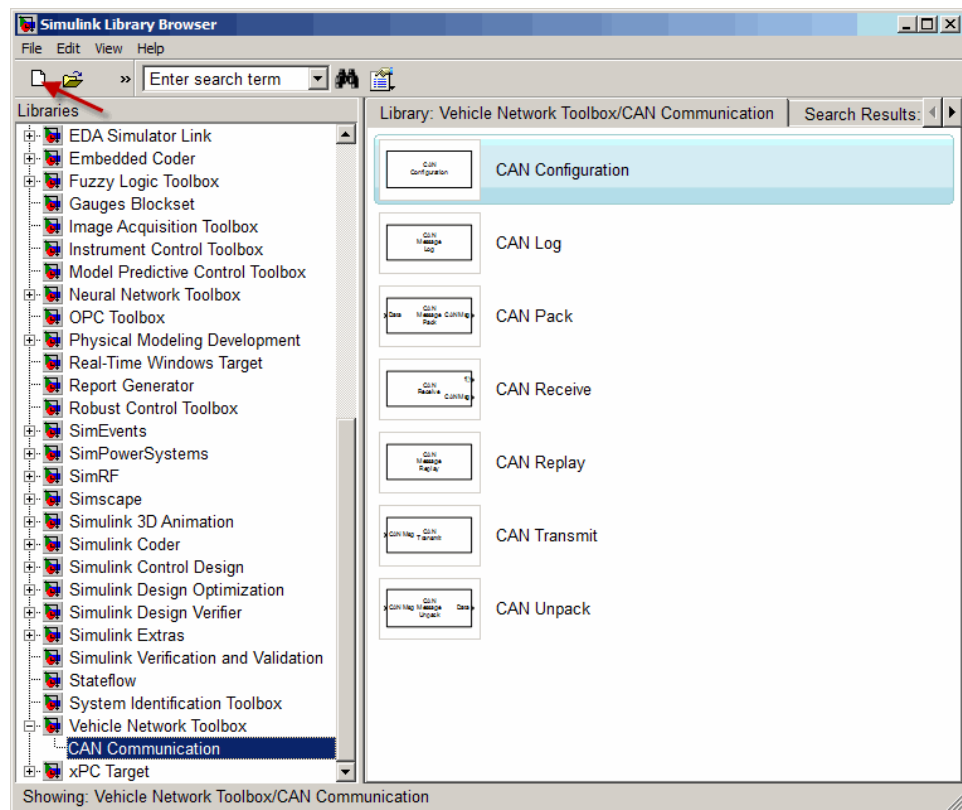
simulink

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.

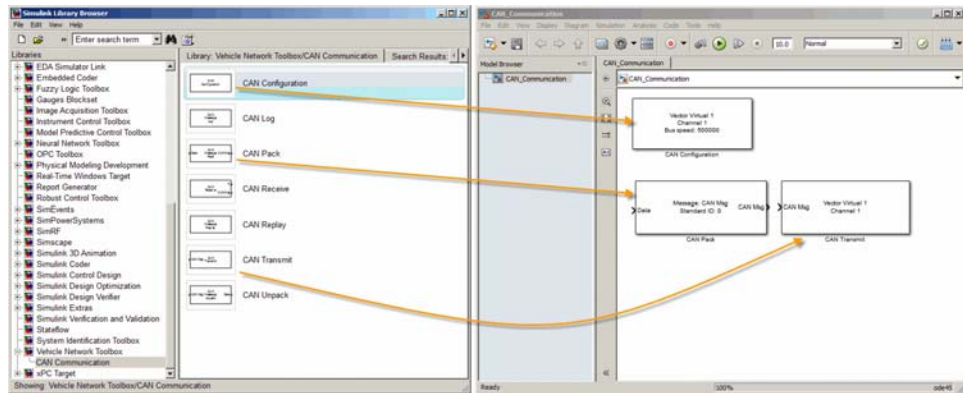
For this example, create a model by clicking the **New model** button on the toolbar.



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.

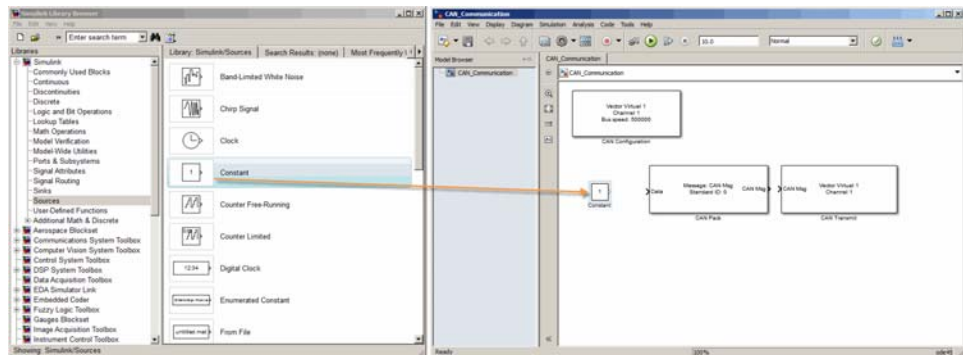
### Step 3: Drag Vehicle Network Toolbox Blocks into the Model

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**.

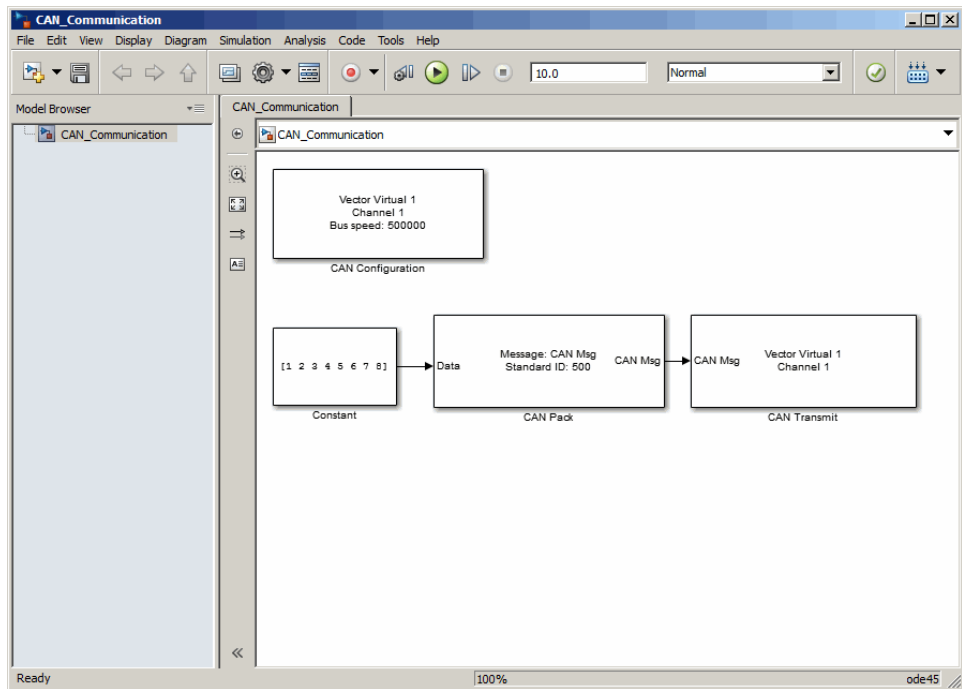
**Configure the CAN Transmit Block.** Double-click the CAN Transmit block to open its parameters dialog box. Set **Device** to Vector Virtual 1 (Channel 1). Click **Apply**, then **OK**.

**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 4-10.
- Use a Function–Call Subsystem block that contains the CAN Unpack block. This function takes 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 4-10 and “Save and Run the Model” on page 4-21 to build your complete model and run the simulation.

- “Step 7: Drag Vehicle Network Toolbox Blocks into the Model” on page 4-15
- “Step 8: Drag Other Blocks to Complete the Model” on page 4-16
- “Step 9: Connect the Blocks” on page 4-18
- “Step 10: Specify the Block Parameter Values” on page 4-20

### Step 7: Drag Vehicle Network Toolbox Blocks into the Model

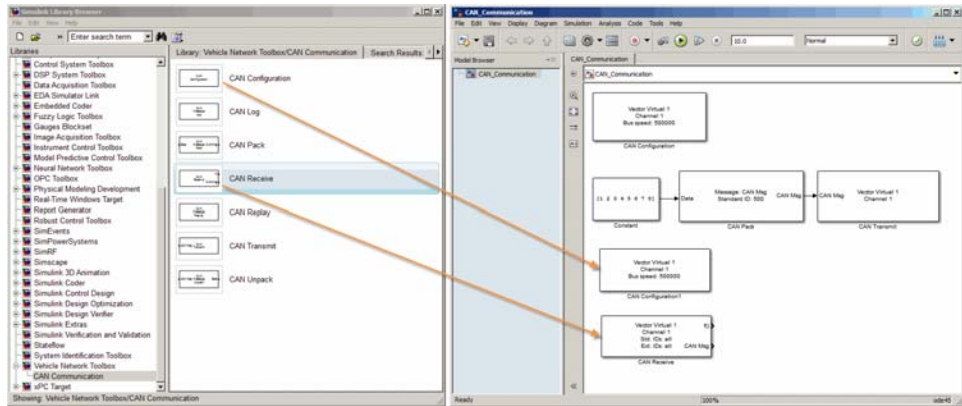
For this example, you need one instance each of the CAN Configuration, CAN Receive, and CAN Unpack blocks in your model. However, you add only the CAN Configuration and the CAN Receive blocks here. Add the CAN Unpack block into the Function–Call Subsystem described in “Step 8: Drag Other Blocks to Complete the Model” on page 4-16.

---

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

---

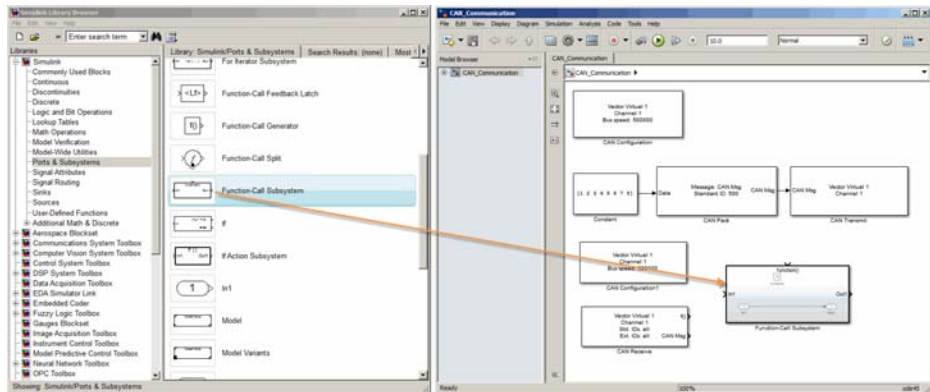
## 4 Communications in Simulink



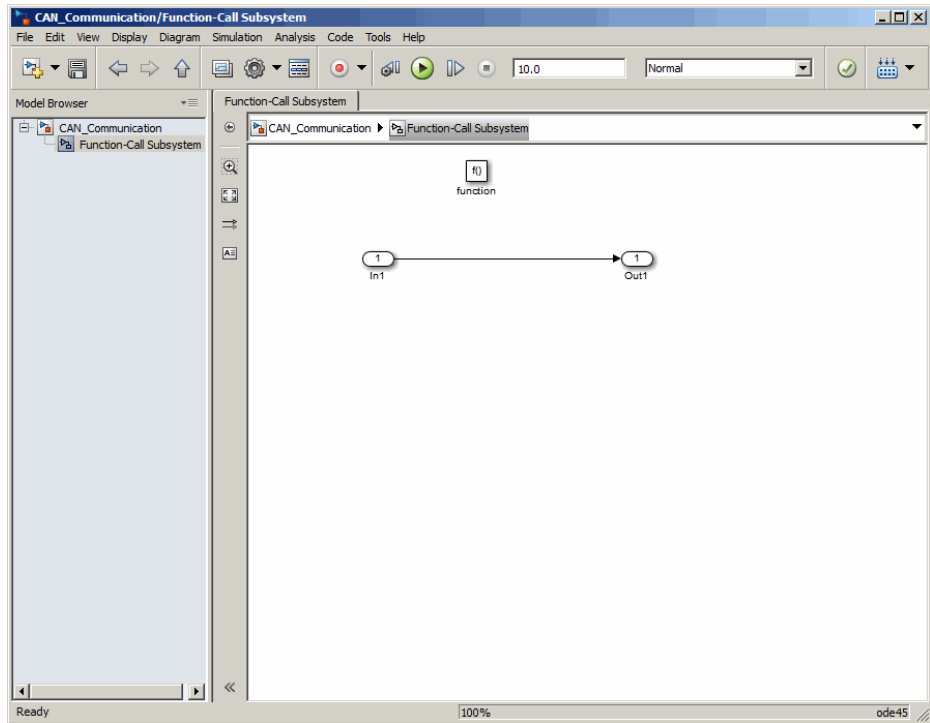
### Step 8: Drag Other Blocks to Complete the Model

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

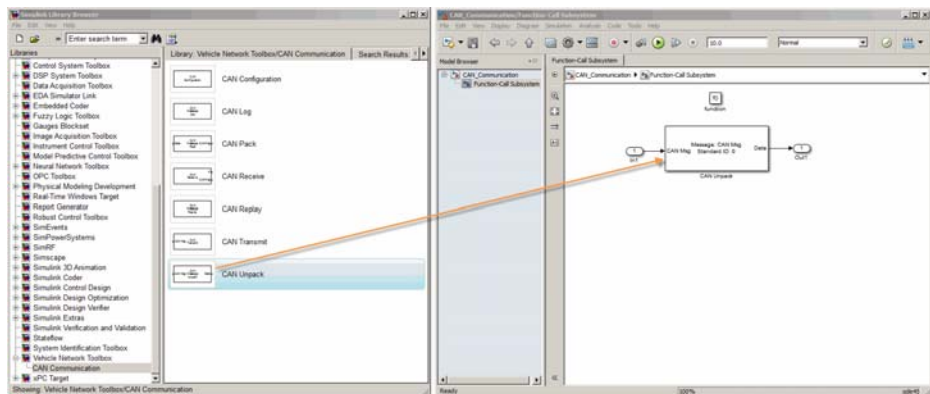
1 Drag the Function–Call Subsystem block into the model.



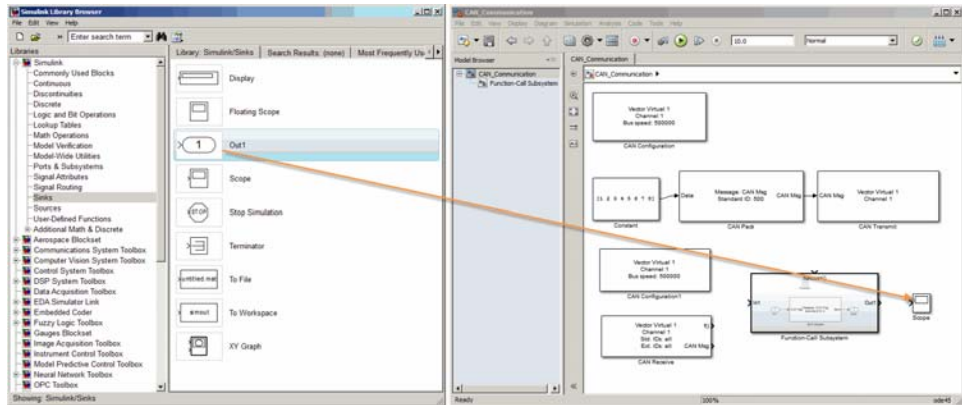
2 Double-click the Function–Call Subsystem block to open the subsystem model.



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

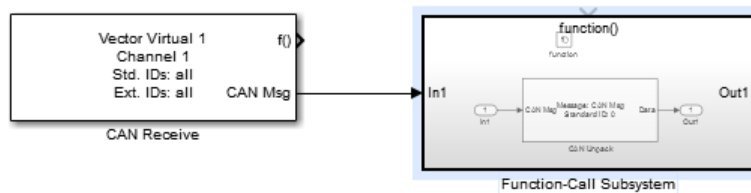


To see the results of the simulation visually, drag the Scope block from the Simulink block library into your model.



### 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.



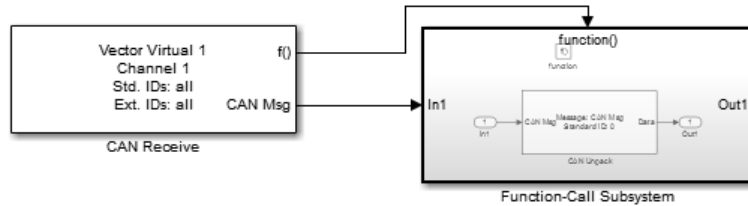
2 Open the Function–Call Subsystem block and:

- 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**.

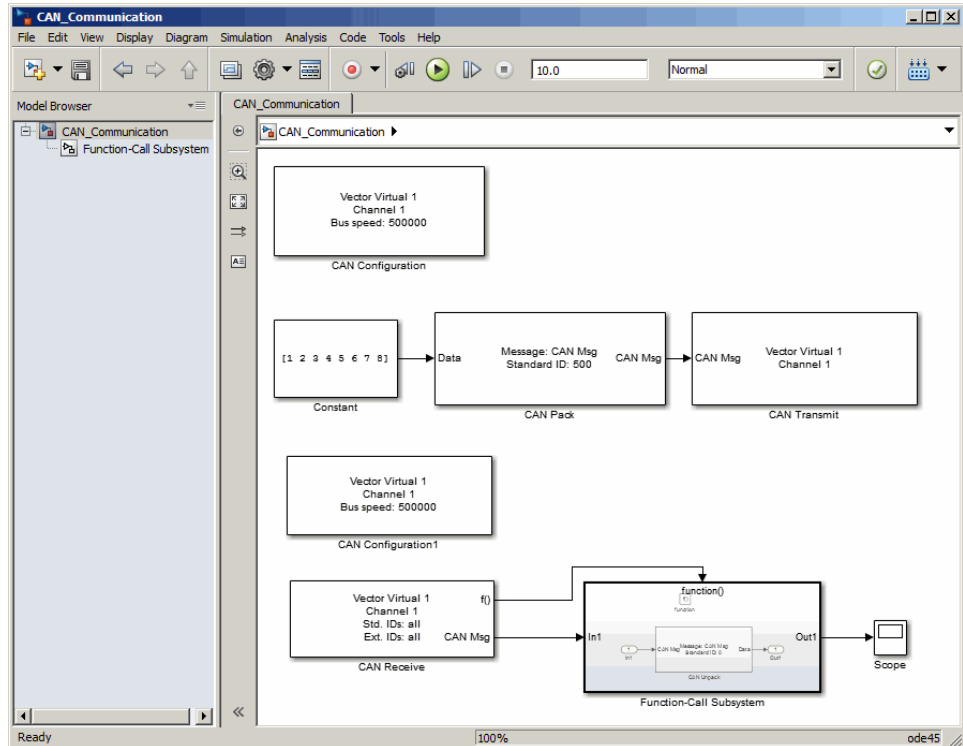
4 Connect the **f()** output port on the CAN Receive block to the **function()** input port on the Function–Call Subsystem block.





**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

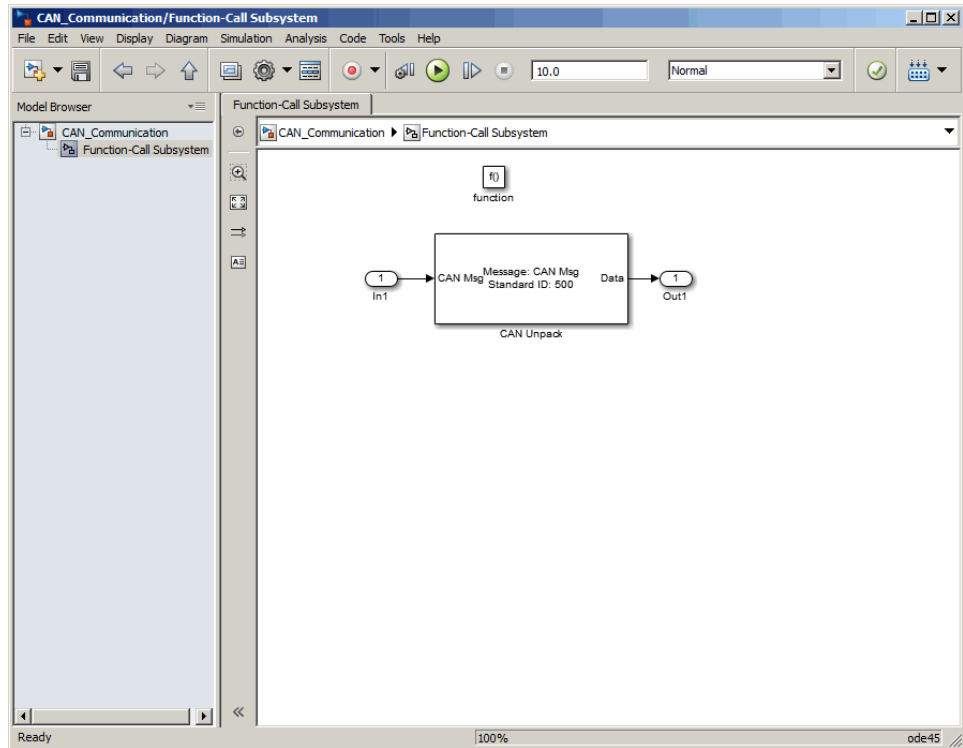
Click **OK**.

**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 4-10 and “Build a Message Receive Model” on page 4-15.

- “Step 11: Save the Model” on page 4-22
- “Step 12: Change Configuration Parameters” on page 4-22
- “Step 13: Run the Simulation” on page 4-22
- “Step 14: View the Results” on page 4-23

### 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

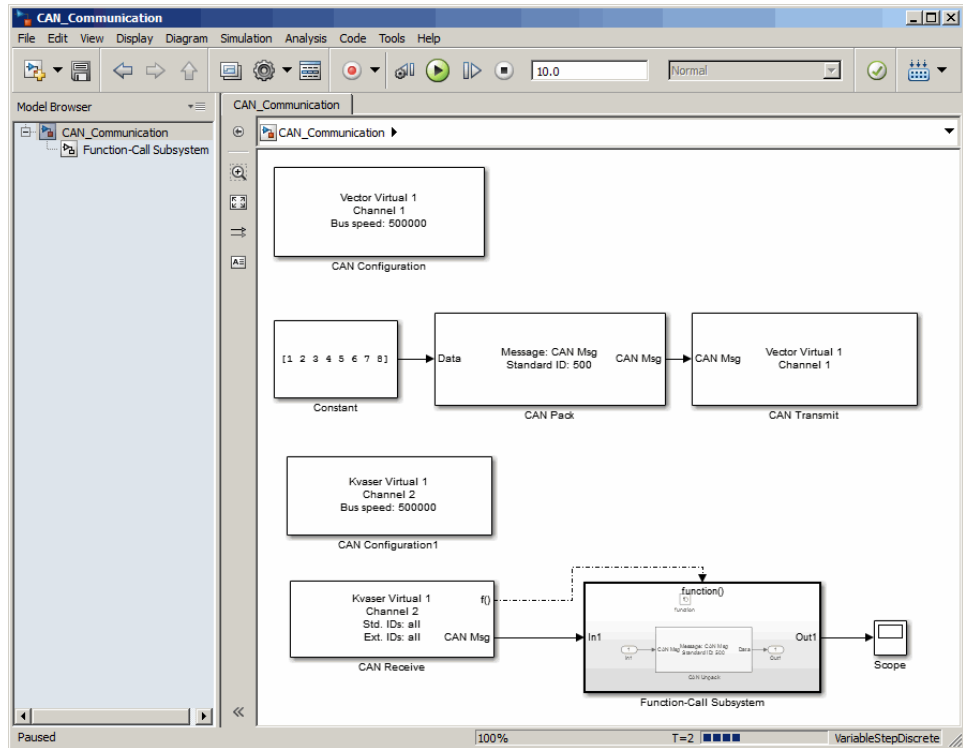
- 1 In your model window, select **Simulation > Model Configuration Parameters**. The Configuration Parameters dialog box opens.
- 2 In the Solver Options section, select:
  - **Fixed-step** from the **Type** list.
  - **Discrete (no continuous states)** from the **Solver** list.

### 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.

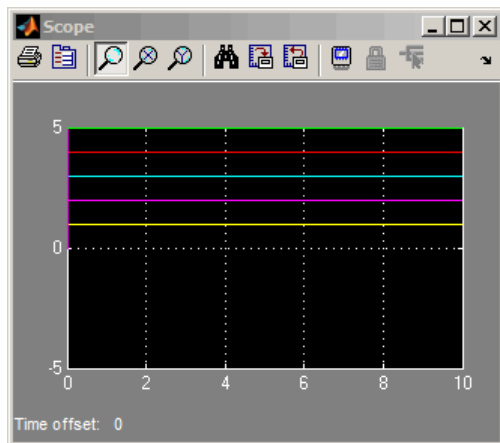
When you run the simulation, the CAN Transmit block gets the message from the CAN Pack block. It then transmits it via Virtual Channel 1. The CAN Receive block on Virtual Channel 2 receives this message and hands it to the CAN Unpack block to unpack the message.

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

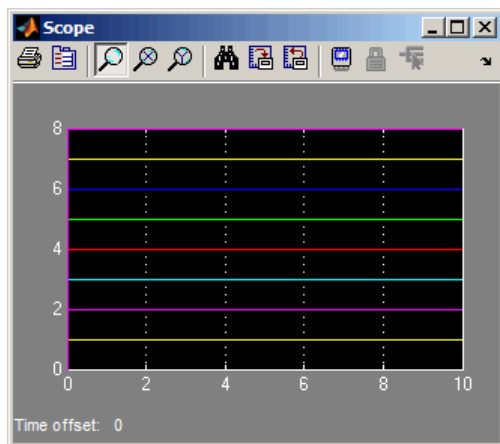


### Step 14: View the Results

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.

## Create Custom Blocks

You can create custom **Receive** and **Transmit** blocks to use with hardware currently not supported by the Vehicle Network Toolbox.

Vehicle Network Toolbox blocks use a custom CAN data type. To use the blocks you create with other Vehicle Network Toolbox blocks, register this custom CAN data type.

To use the custom data type defined in the Vehicle Network Toolbox, write a C/C++ S-function.

To register and use the custom CAN data type, in your S-function:

- 1 Include the `can_datatype.h` header located in `[MATLABROOT]\toolbox\vnt\vntblks\include\candatatype` at the top of the S-function:

```
#include "can_datatype.h"
```

---

**Note** The header `can_message.h` included by `can_datatype.h` is located in `[MATLABROOT]\toolbox\shared\can\src\scanutil\`.

---

- 2 Link your S-function during build to the `scanutil.lib` located in the `[MATLABROOT]\toolbox\vnt\vntblks\lib\<ARCH>` directory. The shared library `scanutil.dll`, is located in the `[MATLABROOT]\bin\<ARCH>`
- 3 Call this function in `mdlInitializeSizes` to initialize the custom CAN data type:

```
mdlInitialize_CAN_datatype(S);
```

- 4 Get custom data type ID using `ssGetDataTypeId`:

```
dataTypeID = ssGetDataTypeId(S,SL_CAN_MESSAGE_DTYPE_NAME);
```

- 5 Do one of the following:

- To create a receive block, set output port data type to `CAN_MESSAGE`:  
`ssSetOutputPortDataType(S, portID, dataTypeID );`
- To create a transmit block, set the input port to `CAN_MESSAGE`:  
`ssSetInputPortDataType(S, portID, dataTypeID );`

For more information on S-functions, see “Overview of S-Functions” in the Simulink User’s Guide.



# 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>transmitPeriodic</code>	Configure a message for periodic transmit
<code>receive</code>	Receive 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
<code>transmitConfiguration</code>	Display messages configured for automatic transmit
<code>transmitEvent</code>	Configure a message for event-based transmit

## **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>canMessageImport</code>	Import CAN message log file from a third-party vendor tool
<code>discard</code>	Discard all messages from CAN channel
<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**

<code>filterAllowAll</code>	Allow all messages of specified identifier type
<code>filterAllowOnly</code>	Configure message filter to allow only specified messages
<code>filterBlockAll</code>	Configure filter to block messages with specified identifier type
<code>filterReset</code>	Open CAN message acceptance filters
<code>filterSet</code>	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, [])`

**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.

**Input Arguments** `message`

The name of the CAN message that you want to attach the database to or remove the database from.

`database`

Handle containing the database (.dbc file) that you want to attach to the message or remove from the message.

**Tips** 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)
canch = canChannel('vendor', 'devicenumber')
```

## 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 'CANCaseXL 1'. For example, if there are two CANcardXL devices, `device` can be 'CANcardXL 1' or 'CANcardXL 2'.

For National Instruments CAN devices, use `canch = canChannel('vendor', 'devicenumber')`, where the `vendor` is the literal string 'NI', and the `devicenumber` is the interface number defined for the channel in NI Measurement & Automation Explorer.

## Tips

- You cannot have more than one active connection to the same NI device channel.
- Use `canHWInfo` to obtain a list of available devices.

## Input Arguments

`vendor`

The name of the CAN device vendor. Specify the vendor name as a string.

`device`

The CAN interface that you want to connect to.

`devicechannelindex`

A numeric channel on the specified device.

`canch`

The CAN channel object the you create.

## Properties

### CAN Channel Properties

BusLoad	Display load on CAN bus
Database	Store CAN database information
MessageReceivedFcn	Specify function to run
MessageReceivedFcnCount	Specify number of messages available before function is triggered
Running	Determine status of CAN channel
SilentMode	Specify if channel is active or silent
TransceiverName	Display name of CAN transceiver
TransceiverState	Display state or mode of CAN transceiver
UserData	Enter custom data

### CAN Device Properties

Device	Display CAN channel device type
Device(NI)	Display NI CAN channel device type
DeviceChannelIndex	Display CAN device channel index
DeviceSerialNumber	Display CAN device serial number
DeviceVendor	Display device vendor name
InitializationAccess	Determine control of device channel



## Bit Timing Properties

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

## Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
canch = canChannel('Vector','Virtual 1',2)
canch = canChannel('NI','CAN1')
```

---

**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

# canDatabase

---

**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.

**Input Arguments** `dbfile.dbc`  
Database file name. You can specify just the name or the full path of the database file.

<b>Properties</b>	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
	UserData	Enter custom data

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

**See Also** `canMessage`

<b>Purpose</b>	Information on available CAN devices
<b>Syntax</b>	<code>out = canHWInfo()</code>
<b>Description</b>	<code>out = canHWInfo()</code> returns information about CAN devices and displays the information on a per vendor and channel basis. Use <code>get</code> on the output of <code>canHWInfo</code> to obtain more detailed results.
<b>Output Arguments</b>	<code>out</code> Handle that will hold the results of <code>canHWInfo</code> .
<b>Examples</b>	<pre>info = canHWInfo; get(info)     ToolboxName: 'Vehicle Network Toolbox'     ToolboxVersion: '1.3 (R2010b)'     MATLABVersion: '7.11 (R2010b)'     VendorInfo: [1x2 can.VendorInfo]</pre>
<b>See Also</b>	<code>canChannel</code>

# canMessage

---

## Purpose

Build CAN message based on user-specified structure

## Syntax

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

## 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.

## Input Arguments

`id`

The ID of the message that you specify.

`extended`

Indicates whether the message ID is of standard or extended type. The Boolean value is `true` if extended or `false` if standard.

`datalength`

The length of the data of the message, in bytes. Specify from 0 through 8.

`database`

Handle to the CAN database containing the message definition.

`messagename`

The name of the message definition in the database.

## Output Arguments

`message`

The message object returned from the function.

## 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
UserData	Enter custom data

## 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')
```

## See Also

[attachDatabase](#) | [canDatabase](#) | [extractAll](#) | [extractRecent](#) | [extractTime](#) | [pack](#) | [unpack](#)

# canMessageImport

---

**Purpose** Import CAN message log file from a third-party vendor tool

**Syntax**  
message = canMessageImport('file', 'vendor')  
message = canMessageImport('file', 'vendor', database)

**Description** message = canMessageImport('file', 'vendor') allows you to import a CAN message log file, file, from a third-party vendor, vendor, into Vehicle Network Toolbox. All the messages in the log file are imported as CAN messages, compatible with MATLAB and displayed as an array.

message = canMessageImport('file', 'vendor', database) allows you to apply a specify a database handle, database. The information in the specified database is applied to the imported CAN log messages.

Once imported, you can analyze, transmit, or replay these messages.

## Tips

- You can import message logs only in certain file formats. You can import only ASCII files from Vector and text files from Kvaser.
- canMessageImport assumes that the information in the imported log file is in a hexadecimal format.
- canMessageImport assumes that the timestamps in the imported log file are absolute values.
- To import Vector log files with symbolic message names, specify an appropriate database file in the input arguments.

## Input Arguments

file

Name of the CAN message log file to import.

vendor

Name of the vendor whose CAN message log file to import.

database

Name of the database whose information to attach to the imported log file.

## Output Arguments

message

The message object returned by the function.

## Examples

To import a log file, type:

```
message = canMessageImport('MsgLog.asc', 'Vector')
```

To specify a database name, type:

```
database = canDatabase('myDatabase.dbc')  
message = canMessageImport('MsgLog.txt', 'Kvaser', database)
```

## See Also

[canDatabase](#) | [receive](#) | [transmit](#)

# canSupport

---

**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.



**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 save messages to a log file via this tool.

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

# configBusSpeed

---

<b>Purpose</b>	Set bit timing rate of CAN channel
<b>Syntax</b>	<pre>configBusSpeed(canch, busspeed) configBusSpeed(canch, busspeed, sjw, tseg1, tseg2,     numberofsamples)</pre>
<b>Description</b>	<p><code>configBusSpeed(canch, busspeed)</code> sets the speed of the CAN channel in a direct form that uses baseline bit timing calculation factors.</p> <p><code>configBusSpeed(canch, busspeed, sjw, tseg1, tseg2, numberofsamples)</code> sets the speed of the CAN channel <code>canch</code> to <code>busspeed</code> using the specified bit timing calculation factors to control the timing in an advanced form.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• Unless you have specific timing requirements for your CAN connection, use the direct form of <code>configBusSpeed</code>. 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.</li><li>• 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 you can add to <code>tseg1</code> (in a slower transmitter), or subtract from <code>tseg2</code> (in a faster transmitter) to regain synchronization during the receipt of a CAN message.</li></ul>
<b>Input Arguments</b>	<p><code>canch</code> The CAN channel object that you want to set the bit timing rate for.</p> <p><code>busspeed</code> The user-specified bit timing rate for the specified object.</p> <p><code>sjw</code></p>

The synchronization jump width. This value is the maximum value of time bit adjustments.

tseg1

The length of time at the start of the sample point within a bit time.

tseg2

The length of time at the end of the sample point within a bit time.

numberofsamples

The specified count of bit samples used.

## 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

# discard

---

<b>Purpose</b>	Discard all messages from CAN channel
<b>Syntax</b>	<code>discard(canch)</code>
<b>Description</b>	<code>discard(canch)</code> discards messages that are available to receive on the channel, <code>canch</code> .
<b>Input Arguments</b>	<code>canch</code> CAN channel that you want to discard the messages from.
<b>Examples</b>	<p>Set up a CAN channel to receive messages and discard the messages received by the channel.</p> <p>Create a CAN channel to receive messages and start the channel:</p> <pre>rxCh = canChannel('Vector','CANcaseXL 1',1); start (rxCh)</pre> <p>Discard all messages in this channel:</p> <pre>discard(rxCh);</pre>
<b>See Also</b>	<code>canChannel</code>

<b>Purpose</b>	Select all instances of message from array of messages
<b>Syntax</b>	<pre>[extracted, remainder] = extractAll(message, messagename) [extracted, remainder] = extractAll(message, id, extended)</pre>
<b>Description</b>	<pre>[extracted, remainder] = extractAll(message, messagename)</pre> parses the given array <code>message</code> , and returns all instances of messages matching the specified message name.  <pre>[extracted, remainder] = extractAll(message, id, extended)</pre> parses the given array <code>message</code> , and returns all instances of messages matching the specified ID with the specified standard or extended type.
<b>Tips</b>	You can specify <code>id</code> as a cell array of message names or a vector of identifiers. For example, if you pass <code>id</code> in as <code>[250 5000]</code> , <code>[false true]</code> , <code>extractAll</code> returns every instance of both CAN message 250 and message 5000 that it finds in the <code>message</code> array. If any <code>id</code> in the vector is an extended type, set <code>extended</code> to <code>true</code> and as a vector of the same length as <code>id</code> .
<b>Input Arguments</b>	<p><code>message</code></p> <p>An array of CAN message objects that you specify to parse and find the specified messages by name or ID.</p> <p><code>messagename</code></p> <p>The name of the message that you specify to extract.</p> <p><code>id</code></p> <p>The ID of the message that you specify to extract.</p> <p><code>extended</code></p> <p>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.</p>

# extractAll

---

## Output Arguments

`extracted`

An array of CAN message objects returned with all instances of `id` found in the `message`.

`remainder`

A CAN message object containing all messages in the original input `message` with all instances of `id` removed.

## 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)
```

## 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.

## Tips

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

## Input 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.

# extractRecent

---

extended

Indicates whether the message ID is a standard or extended type. The Boolean value is true if extended and false if standard.

## Output Arguments

extracted

An array of CAN message objects returned with the most recent instance of id found in the message.

## 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](#)



<b>Purpose</b>	Select messages occurring within specified time range from array of messages
<b>Syntax</b>	<pre>extracted = extractTime(message, starttime, endtime, msgRange)</pre>
<b>Description</b>	<pre>extracted = extractTime(message, starttime, endtime, msgRange)</pre> parses the array <code>message</code> and returns all messages with a timestamp within the specified <code>starttime</code> and <code>endtime</code> , including the <code>starttime</code> and <code>endtime</code> .
<b>Tips</b>	Specify the time range in increasing order from <code>starttime</code> to <code>endtime</code> . If you must specify the largest available time, <code>endtime</code> also accepts <code>Inf</code> as a valid value. The earliest acceptable time you can specify for <code>starttime</code> is 0.
<b>Input Arguments</b>	<p><code>message</code> An array of CAN message objects.</p> <p><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.</p> <p><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.</p>
<b>Output Arguments</b>	<p><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>.</p>

# extractTime

---

## Examples

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

## See Also

[extractAll](#) | [extractRecent](#)

<b>Purpose</b>	Allow all messages of specified identifier type
<b>Syntax</b>	<code>filterAllowAll(canch, 'type')</code>
<b>Description</b>	<code>filterAllowAll(canch, 'type')</code> opens the filter on the specified CAN channel to allow all messages matching the specified identifier type to pass the acceptance filter.
<b>Input Arguments</b>	<p><code>canch</code> The CAN channel on which you want to filter messages.</p> <p><code>type</code> The identifier type by which to filter. CAN messages are 'Standard' and 'Extended'.</p>
<b>Examples</b>	<p>To allow all standard and extended message typed to pass the filter, type:</p> <pre>canch = canChannel('Vector', 'CANCaseXL 1', 1) filterAllowAll(canch, 'Standard') filterAllowAll(canch, 'Extended')</pre>
<b>See Also</b>	<code>filterAllowOnly</code>   <code>filterBlockAll</code>

# filterAllowOnly

---

<b>Purpose</b>	Configure message filter to allow only specified messages
<b>Syntax</b>	<code>filterAllowOnly(canch, 'name')</code> <code>filterAllowOnly(canch, ids, 'type')</code>
<b>Description</b>	<p><code>filterAllowOnly(canch, 'name')</code> configures the filter on the channel <code>canch</code>, to pass only messages with the specified name.</p> <p><code>filterAllowOnly(canch, ids, 'type')</code> configures the filter on the channel <code>canch</code>, to pass only messages of the specified type with the specified identifier.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use <code>Database</code> to attach a database to your CAN channel and filter messages using message names.</li><li>• The <code>id</code> value is stored as a decimal value. To convert a hexadecimal to a decimal value, use the <code>hex2dec</code> function.</li></ul>
<b>Input Arguments</b>	<p><code>canch</code> The CAN channel on which you want to filter messages.</p> <p><code>name</code> the name of the CAN message that you want to allow. You can specify a single name as a string or a cell array of message names.</p> <p><code>ids</code> The CAN message ID or IDs that you want to allow. You can specify:</p> <ul style="list-style-type: none"><li>• Single value, such as <code>600</code></li><li>• Multiple values, such as <code>[600,610]</code></li><li>• Range of values, such as <code>[600:800]</code></li><li>• Multiple ranges, such as <code>[200:400, 600:800]</code></li></ul> <p><code>type</code></p>

The identifier type by which to filter messages. CAN messages are 'Standard' and 'Extended'.

## Examples

To filter a database defined message with name 'EngineMsg', type:

```
canch = canChannel('Vector','CANCaseXL 1',1);  
canch.Database = canDatabase('candatabase.dbc');  
filterAllowOnly(canch,'EngineMsg')
```

---

To filter messages by identifier, type:

```
canch = canChannel('Vector','CANCaseXL 1',1);  
filterAllowOnly(canch,[602 612],'Standard');
```

## See Also

[filterAllowAll](#) | [filterBlockAll](#) | [hex2dec](#)

# filterBlockAll

---

<b>Purpose</b>	Configure filter to block messages with specified identifier type
<b>Syntax</b>	<code>filterBlockAll(canch, 'type')</code>
<b>Description</b>	<code>filterBlockAll(canch, 'type')</code> configures the message filter to block all messages matching the specified identifier type.
<b>Input Arguments</b>	<code>canch</code> The CAN channel on which you want to filter messages.  <code>type</code> The identifier type by which to filter messages. CAN messages are 'Standard' and 'Extended'.
<b>Examples</b>	To block all standard message types, type:  <pre>canch = canChannel('Vector', 'CANCASEXL 1', 1) filterBlockAll(canch, 'Standard')</pre>
<b>See Also</b>	<code>filterAllowAll</code>   <code>filterAllowOnly</code>

**Purpose** Set range of CAN identifiers to pass acceptance filter

---

**Note** `filterAcceptRange` will be removed in a future version. Use `filterAllowAll`, `filterAllowOnly`, or `filterBlockAll` instead.

---

**Syntax** `filterAcceptRange(canch, rangestart, rangeend)`

---

**Note** You cannot set filters on an NI device channel.

---

**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.
-

# filterAcceptRange

---

## Tips

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.

## Input Arguments

`canch`

The CAN channel that you want to set the filter for.

`rangestart`

The first identifier of the range of message IDs that the filter accepts.

`rangeend`

The last identifier of the range of message IDs that the filter accepts.

## Examples

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

## See Also

`filterBlockRange` | `filterReset` | `filterSet`



## Purpose

Set range of CAN identifiers to block via acceptance filter

---

**Note** `filterBlockRange` will be removed in a future version. Use `filterAllowAll`, `filterAllowOnly`, or `filterBlockAll` instead.

---

## Syntax

```
filterBlockRange(canch, rangestart, rangeend)
```

---

**Note** You cannot set filters on an NI device channel.

---

## 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.

## Tips

- 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.

## Input Arguments

`canch`

The CAN channel that you want to set the filter for.

`rangestart`

# filterBlockRange

---

The first identifier of the range of message IDs that the filter starts blocking at.

rangeend

The last identifier of the range of message IDs that the filter stops blocking at.

## 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](#)

**Purpose** Open CAN message acceptance filters

---

**Note** `filterReset` will be removed in a future version. Use `filterAllowAll`, `filterAllowOnly`, or `filterBlockAll` instead.

---

**Syntax** `filterReset(canch)`

---

**Note** You cannot set filters on an NI device channel.

---

**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`.

**Input Arguments** `canch`  
The CAN channel that you want to reset the filter for.

**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`

# filterSet

---

**Purpose** Set specific CAN message acceptance filter configuration

---

**Note** `filterSet` will be removed in a future version. Use `filterAllowAll`, `filterAllowOnly`, or `filterBlockAll` instead.

---

**Syntax**

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

---

**Note** You cannot set filters on an NI device channel.

---

**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.

- Tips**
- 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.

**Input Arguments**

`canch`  
The CAN channel that you want to set the filter for.

`code`  
The value required for each bit position of the identifier.

**mask**

The bits in the identifier that are relevant to the filter.

**id**

Set a filter on the CAN message with the `id`, range of `ids`, multiple ranges of `ids`, or a combination of `ids`.

**idtype**

A string specifying either a standard or an extended CAN message `id` type.

## Examples

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

To let 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`

# get

---

**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)
```

<b>Purpose</b>	Information about CAN messages
<b>Syntax</b>	<pre>msgInfo = messageInfo(candb) msgInfo = messageInfo(candb, 'msgName') msgInfo = messageInfo(candb, id, extended)</pre>
<b>Description</b>	<p><code>msgInfo = messageInfo(candb)</code> returns information about CAN messages in the specified database <code>candb</code>.</p> <p><code>msgInfo = messageInfo(candb, 'msgName')</code> returns information about the specified message <code>'msgName'</code> in the specified database <code>candb</code>.</p> <p><code>msgInfo = messageInfo(candb, id, extended)</code> returns information about the message with the specified standard or extended ID in the specified database <code>candb</code>.</p>
<b>Input Arguments</b>	<p><code>candb</code> The database containing the CAN messages that you want information about.</p> <p><code>msgName</code> The name of the message you want information about.</p> <p><code>id</code> The numeric identifier of the specified message.</p> <p><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.</p>
<b>Output Arguments</b>	<p><code>msgInfo</code> Handle for the returned CAN messages in the specified database.</p>
<b>Examples</b>	<pre>candb = canDatabase('c:\Database.dbc') msgInfo = messageInfo(candb)</pre>

# messageInfo

---

```
msgInfo = messageInfo(candb, 'msgName')  
msgInfo = messageInfo(candb, 500, false)
```

## See Also

[canDatabase](#) | [canMessage](#) | [signalInfo](#)



---

<b>Purpose</b>	Pack signal data into CAN message
<b>Syntax</b>	<code>pack(message, value, startbit, signalsize, byteorder)</code>
<b>Description</b>	<code>pack(message, value, startbit, signalsize, byteorder)</code> takes specified input parameters and packs them into the message.
<b>Input Arguments</b>	<p><code>message</code> The CAN message structure that you specify for the signal to be packed in.</p> <p><code>value</code> The value of the signal you specify to be packed in the message.</p> <p><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>startbit</code> are from 0 through 63.</p> <p><code>signalsize</code> The length of the signal in bits. Accepted values for <code>signalsize</code> are from 1 through 64.</p> <p><code>byteorder</code> The signal byte order format. Accepted values are 'LittleEndian' and 'BigEndian'.</p>
<b>Examples</b>	<code>pack(message, 25, 0, 16, 'LittleEndian')</code>
<b>See Also</b>	<code>canMessage</code>   <code>extractAll</code>   <code>extractRecent</code>   <code>extractTime</code>   <code>unpack</code>

# receive

---

**Purpose** Receive messages from CAN bus

**Syntax** `message = receive(canch, messagesrequested)`

**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`.

## Input Arguments

`canch`

The CAN channel from which to receive the message.

`messagesrequested`

The maximum count of messages to receive. The specified value must be a nonzero and positive, or `Inf`.

## Output Arguments

`message`

An array of CAN message objects received from the channel.

## Properties

### Receive Message Properties

<code>MessageReceivedFcn</code>	Specify function to run
<code>MessageReceivedFcnCount</code>	Specify number of messages available before function is triggered
<code>MessagesAvailable</code>	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 Log Properties

ReceiveErrorCount	Display number of received errors detected by channel
TransmitErrorCount	Display number of transmitted errors by channel

### 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](#)

# replay

---

**Purpose** Retransmit messages from CAN bus

**Syntax** `replay(canch, message)`

## Arguments

**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`.

## Input Arguments

`canch`

The CAN channel that you specify to transmit the messages.

`message`

An array of message objects to replay.

## 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)
%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`

# set

---

<b>Purpose</b>	Configure property values
<b>Syntax</b>	<code>set (obj, propertyname, propertyvalue)</code>
<b>Description</b>	<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> .
<b>Examples</b>	<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>

<b>Purpose</b>	Information about signals in CAN message
<b>Syntax</b>	<pre>SigInfo = signalInfo(candb, 'msgName') SigInfo = signalInfo(candb, id, extended) SigInfo = signalInfo(candb, id, extended, 'signalName')</pre>
<b>Description</b>	<p><code>SigInfo = signalInfo(candb, 'msgName')</code> returns information about the signals in the specified CAN message <code>msgName</code>, in the specified database <code>candb</code>.</p> <p><code>SigInfo = signalInfo(candb, id, extended)</code> returns information about the signals in the message with the specified standard or extended ID <code>id</code>, in the specified database <code>candb</code>.</p> <p><code>SigInfo = signalInfo(candb, id, extended, 'signalName')</code> returns information about the specified signal <code>'signalName'</code> in the message with the specified standard or extended ID <code>id</code>, in the specified database <code>candb</code>.</p>
<b>Input Arguments</b>	<p><code>candb</code> The database containing the signals that you want information about.</p> <p><code>msgName</code> The name of the message that contains the signals that you want information about.</p> <p><code>id</code> The numeric identifier of the specified message that contains the signals you want information about.</p> <p><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.</p> <p><code>signalName</code></p>

# signalInfo

---

The name of the specific signal that you want information about.

## Output Arguments

SigInfo

The signal information object returned from the function.

## 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



**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`

# 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`

---

<b>Purpose</b>	Send CAN messages to CAN bus
<b>Syntax</b>	<code>transmit(canch, message)</code>
<b>Description</b>	<code>transmit(canch, message)</code> sends the array of messages onto the bus via the CAN channel. To understand the elements of a message, refer to <code>canMessage</code> .
<b>Tips</b>	The Transmit function ignores the <code>Timestamp</code> property and the <code>Error</code> property.
<b>Input Arguments</b>	<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.
<b>Examples</b>	<pre>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)</pre> <p>To transmit an array, construct <code>message1</code> and <code>message2</code> as in this example, and type:</p> <pre>transmit(canch, [message, message1 message2])</pre> <p>To transmit messages on a remote frame, type:</p> <pre>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)</pre>

# transmit

---

```
start(canch)
transmit(canch, message)
```

## **See Also**

`canChannel` | `canMessage` | `receive`

<b>Purpose</b>	Display messages configured for automatic transmit
<b>Syntax</b>	<code>transmitConfiguration(<i>canch</i>)</code>
<b>Description</b>	<p><code>transmitConfiguration(<i>canch</i>)</code> displays information about all messages in the CAN Channel, <i>canch</i>, configured for periodic transmit or event-based transmit.</p> <p>For more information on periodic transmit of messages, refer to <code>transmitPeriodic</code>.</p> <p>For more information on event-based transmit of messages, refer to <code>transmitEvent</code>.</p>
<b>Input Arguments</b>	<p><i>canch</i></p> <p>Name of the CAN channel configured for periodic transmit or event-based transmit.</p>
<b>Examples</b>	<p>Create a CAN channel and configure two messages:</p> <pre>canch = canChannel('Vector', 'Virtual 1', 1); msg1 = canMessage(500, false, 8); msg2 = canMessage(750, false, 8);</pre> <p>Transmit <i>msg1</i> and <i>msg2</i> using <code>transmitEvent</code> and <code>transmitPeriodic</code> respectively:</p> <pre>transmitEvent(canch, msg1, 'On'); transmitPeriodic(canch, msg2, 'On', 1);</pre> <p>Display messages on <i>canch</i> configured for periodic or event-based transmit:</p> <pre>transmitConfiguration(canch)</pre> <p>The function returns information about periodic configuration:</p> <p>Periodic Messages</p>

# transmitConfiguration

---

ID	Extended Name	Data	Rate (seconds)
750	false	0 0 0 0 0 0 0 0	1.000000

## Event Messages

ID	Extended Name	Data
500	false	0 0 0 0 0 0 0 0

## See Also

[canChannel](#) | [canMessage](#) | [transmitEvent](#) | [transmitPeriodic](#)

<b>Purpose</b>	Configure a message for event-based transmit
<b>Syntax</b>	<code>transmitEvent(canch, msg, 'state')</code>
<b>Description</b>	<code>transmitEvent(canch, msg, 'state')</code> enables an event-based transmit of the CAN message, <code>msg</code> , on the channel, <code>canch</code> when <code>'state'</code> is <code>On</code> and disables it when <code>'state'</code> is <code>Off</code> .
<b>Input Arguments</b>	<p><code>canch</code></p> <p>The name of the CAN channel on which the specified message is enabled for event-based transmit.</p> <p><code>msg</code></p> <p>The message enabled for event-based transmission on the specified CAN channel.</p> <p><code>'state'</code></p> <p>Specify whether event-based transmission is enabled on the specified message. Input <code>Off</code> to enable event-based transmission on the specified message, and <code>Off</code> for disabling it.</p>
<b>Examples</b>	<p>Construct a CAN channel and configure a message on the channel:</p> <pre>canch = canChannel('Vector', 'CANCASEXL 1', 1); msg = canMessage(500, false, 4);</pre> <p>Enable the message for event-based transmit, start the channel, and change some data to trigger the event-based transmit:</p> <pre>transmitEvent(ch, msg, 'On'); start(canch); msg.Data = [1 2 3 4];</pre>
<b>See Also</b>	<code>canChannel</code>   <code>canMessage</code>   <code>transmitConfiguration</code>   <code>transmitPeriodic</code>

# transmitPeriodic

---

<b>Purpose</b>	Configure a message for periodic transmit
<b>Syntax</b>	<pre>transmitPeriodic(canch, msg, 'On', period) transmitPeriodic(canch, msg, 'Off')</pre>
<b>Description</b>	<p><code>transmitPeriodic(canch, msg, 'On', period)</code> enables periodic transmit of the message, <code>msg</code>, on the channel, <code>canch</code>, to transmit at the specified period, <code>period</code>.</p> <p><code>transmitPeriodic(canch, msg, 'Off')</code> disables periodic transmit of the message, <code>msg</code>, on the channel, <code>canch</code>.</p>
<b>Tips</b>	You can enable and disable periodic transmit even when the channel is running. This functionality allows you to make changes to the state of the channel without stopping the channel.
<b>Input Arguments</b>	<p><code>canch</code> The name of the CAN channel on which the specified message is enabled for periodic transmit.</p> <p><code>msg</code> The message enabled for periodic transmission on the specified CAN channel.</p> <p><code>'state'</code> Specify whether periodic transmission is enabled on the specified message. Input <code>On</code> for enabling periodic transmission on the specified message and <code>Off</code> to disable it. If you enable periodic transmission, specify a period value.</p> <p><code>period</code> Specify a period in seconds. This value is used to transmit the message in the specified period. By default this value is 0.500 seconds.</p>



## Examples

Construct a CAN channel and configure a message on the channel:

```
canch = canChannel('Vector', 'Virtual 1', 1);  
msg = canMessage(500, false, 4);
```

Enable the message for periodic transmit. Set the period value to 1 sec, start the channel, and update the data in the message you want to send periodically:

```
transmitPeriodic(canch, msg, 'On', 1);  
start(canch);  
msg.Data = [1 2 3 4];
```

## See Also

[canChannel](#) | [canMessage](#) | [transmitConfiguration](#) | [transmitEvent](#)

# unpack

---

<b>Purpose</b>	Unpack signal data from message
<b>Syntax</b>	<code>value = unpack(message, startbit, signalsize, byteorder, datatype)</code>
<b>Description</b>	<code>value = unpack(message, startbit, signalsize, byteorder, datatype)</code> takes a set of input parameters to unpack the signal value from the message and returns the value as output.
<b>Input Arguments</b>	<p><code>message</code> The CAN message structure that you specify for the signal to be unpacked from.</p> <p><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.</p> <p><code>signalsize</code> The length of the signal in bits. Accepted values for <code>signalsize</code> are from 1 through 64.</p> <p><code>byteorder</code> The signal binary or binblock format. Accepted values are <code>LittleEndian</code> and <code>BigEndian</code>.</p> <p><code>datatype</code> The data type that you want to get the unpacked value in.</p>
<b>Output Arguments</b>	<p><code>value</code> The value of the message that you specify to be unpacked.</p>
<b>Examples</b>	<code>value = unpack(message, 0, 16, 'LittleEndian', 'int16')</code>

## See Also

[canMessage](#) | [extractAll](#) | [extractRecent](#) | [extractTime](#) | [pack](#)

# unpack

---

# 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 CAN channel
CAN Message Properties (p. 7-2)	Setting properties that specify CAN message properties
CAN Database Properties (p. 7-3)	Setting properties of CAN channels attached to CAN database
Receiving Messages (p. 7-3)	Defining actions based on available messages on 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
UserData	Enter custom data

## **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
UserData	Enter custom data

## 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
UserData	Enter custom data

## 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 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
Device(NI)	Display NI 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 property

---

**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 property

---

**Purpose** Determine status of CAN bus

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

<b>Characteristics</b>	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 property

---

**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 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 property

---

**Purpose** Display CAN channel device type

**Description** For National Instruments devices, the `Device` property displays the device number on the hardware.  
For all other devices, the `Device` property displays information about the device type to which the CAN channel is connected.

<b>Characteristics</b>	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`, `DeviceType`, `DeviceVendor`

**Purpose** Display NI CAN channel device type

**Description** For National Instruments devices, the `DeviceType` property displays information about the device type to which the CAN channel is connected.

<b>Characteristics</b>	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`

# DeviceChannelIndex property

---

**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 property

---

**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	<ul style="list-style-type: none"><li>• Numeric</li><li>• Hexadecimal String (NI CAN devices only)</li></ul>

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

**See Also**

**Functions**  
canChannel, canHWInfo

**Properties**  
Device, DeviceVendor

# DeviceVendor property

---

**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`

**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`

# Extended property

---

**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.

<b>Characteristics</b>	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



**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

# InitializationAccess property

---

**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.

---

<b>Characteristics</b>	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 property

---

**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`

# MessageReceivedFcnCount property

---

**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`

**Purpose** Store message names from CAN database

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

<b>Characteristics</b>	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`

# MessagesAvailable property

---

**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 property

---

**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

# MessagesTransmitted property

---

**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) property

---

**Purpose** CAN database name

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

<b>Characteristics</b>	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

# Name (Message) property

---

**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 property

---

**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.

---

**Note** This property is not available for National Instruments CAN devices. The channel displays NaN for the value.

---

<b>Characteristics</b>	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

# Path property

---

**Purpose** Display CAN database directory path

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

<b>Characteristics</b>	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.

<b>See Also</b>	<b>Functions</b>
	<code>canDatabase</code>

# ReceiveErrorCount property

---

**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

# Remote property

---

**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`

**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`

# SilentMode property

---

**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.

<b>Characteristics</b>	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`



**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`

# SJW property

---

**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.

---

**Note** This property is not available for National Instruments CAN devices. The channel displays NaN for the value.

---

<b>Characteristics</b>	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

**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.

<b>Characteristics</b>	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

# TransceiverName property

---

**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.

<b>Characteristics</b>	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`

**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.

<b>Characteristics</b>	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

# TransmitErrorCount property

---

**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

**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.

---

**Note** This property is not available for National Instruments CAN devices. The channel displays NaN for the value.

---

<b>Characteristics</b>	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

# TSEG2 property

---

**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.

---

**Note** This property is not available for National Instruments CAN devices. The channel displays NaN for the value.

---

<b>Characteristics</b>	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



**Purpose** Enter custom data

**Description** Use the UserData property to enter custom data to be stored in your CAN channel, message, or database object. When you save an object with UserData specified, you automatically save the custom data. When you load an object with UserData specified, you automatically load the custom data.

---

**Note** To avoid unexpected results when you save and load an object with UserData, specify your custom data in simple data types and constructs.

---

**Characteristics**

Usage	CAN channel, CAN Message, CAN Database
Read only	Never
Data type	User defined

**See Also** **Functions**  
canChannel, canMessage, canDatabase

# **UserData**

---

# Block Reference

---

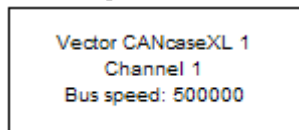
# CAN Configuration

---

**Purpose** Configure parameters for specified CAN device

**Library** Vehicle Network Toolbox: CAN Communication

## Description



CAN Configuration

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 Vehicle Network Toolbox and Simulink software to use this block.

---

## Other Supported Features

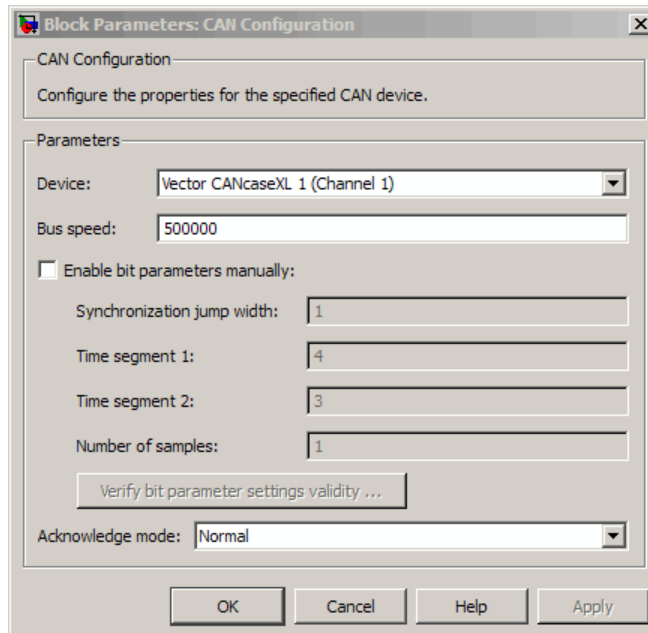
The CAN Configuration block supports the use of Simulink Accelerator™ mode. Using this feature, you can speed up the execution of Simulink models.

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

The CAN Configuration block supports the use of code generation when you use it with the CAN Receive and CAN Transmit blocks.

## 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

---

**Note** This option is disabled if you are using an NI CAN channel.

---

# CAN Configuration

---

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 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.

---

## Notes

- You cannot specify the mode if you are using NI virtual channels.
  - Use Silent mode only if you want to observe and analyze your network activity.
- 

## See Also

CAN Receive, CAN Transmit

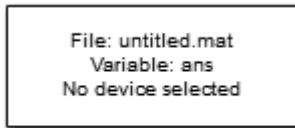
# CAN Log

---

**Purpose** Log received CAN messages

**Library** Vehicle Network Toolbox: CAN Communication

## Description



CAN Log

The CAN Log block logs CAN messages from the CAN network or messages sent to the blocks input port to a .mat file. You can load the saved messages into MATLAB for further analysis or into another Simulink model.

---

**Note** If your model uses a National Instruments device, you cannot connect CAN Receive block and CAN Log to the same channel on the device.

---

You cannot connect a channel on a National Instruments device to more than one block. Configure your CAN Log block to log from the Simulink input port. Refer to the **Basic CAN Message Replay and Logging** demo in the Vehicle Network Toolbox Simulink Demos for more information.

The Log block appends the specified filename with the current date and time, creating unique log files for repeated logging.

If you want to use messages logged using Simulink blocks in the MATLAB Command window, use `canMessage` to convert messages to the correct format. Refer to the **Basic CAN Message Replay and Logging** demo in the Vehicle Network Toolbox Simulink Demos for information.



---

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

---

### Other Supported Features

The CAN Log 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.

The CAN Log block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see Code Generation.

## Code Generation

Vehicle Network Toolbox Simulink blocks allow you to generate code, enabling models containing these blocks to run successfully in Accelerator, Rapid Accelerator, External, and Deployed modes.

### Code Generation with the Simulink Coder™

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder™ products together to generate code (on the host end) that you can use to implement your model for a practical application. For more information on code generation, see Building Executables.

### Shared Library Dependencies

The block generates code with limited portability. The block uses precompiled shared libraries, such as DLLs, to support I/O for specific types of devices. With this block, you can use the packNGo function supported by the Simulink Coder, to set up and manage the build information for your models. The packNGo function allows you to package model code and dependent shared libraries into a zip file for deployment. You do not need MATLAB installed on the target system, but the target system needs to be supported by MATLAB.

To set up packNGo:

```
set_param(gcs, 'PostCodeGenCommand', 'packNGo(buildInfo)');
```

# CAN Log

---

In this example, `gcs` is the current model that you wish to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and the source code in the zip file can be built to create an executable which can be run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information on packNGo, see packNGo.

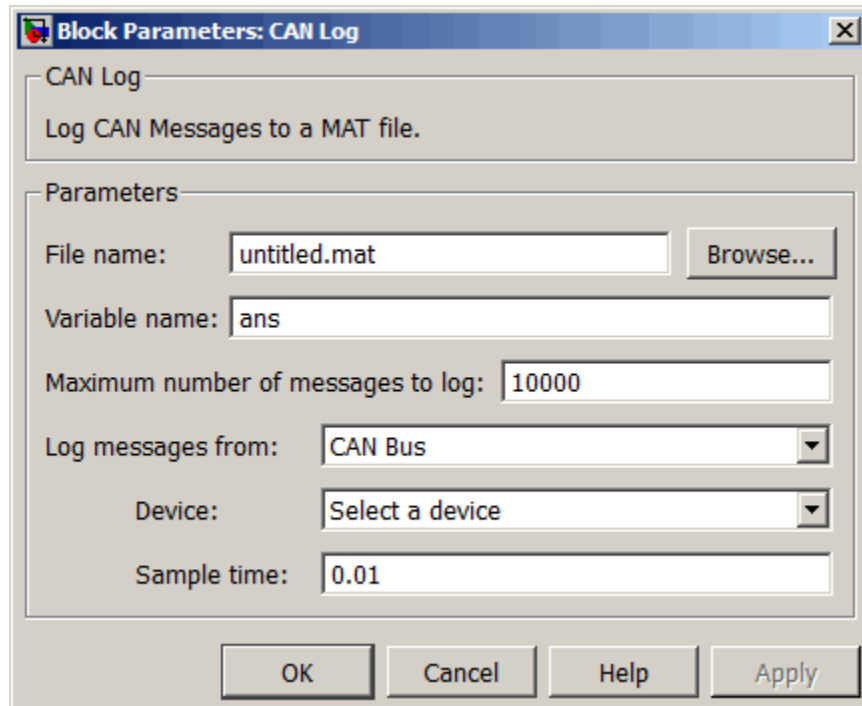
## Dialog Box

Use the Block Parameters dialog box to configure your message logging.

---

**Tip** If you are logging from the network, you need to configure your CAN channel with a CAN Configuration block.

---



**File name**

Type the name and path of the file to log CAN messages to, or click **Browse** to browse to a file location.

The model appends the log file name with the current date and time in the YYYY-*MMM*-DD\_hhmmss format. You can also open the block mask and specify a unique name to differentiate between your files for repeated logging.

**Variable Name**

Type the variable saved in the MAT-file that holds the CAN message information.

**Maximum number of messages to log**

Specify the maximum number of messages this block can log from the selected device or port. The specified value must be a positive integer. If you do not specify a value the block uses the default value of 10,000 messages. The log file saves the most recent messages up to the specified maximum number.

**Log messages from**

Select the source of the messages logged by the block. Possible values are CAN Bus or Input port. To log messages from the network, you must specify a device.

**Device**

Select the device on the CAN network that you want to log messages from. This field is unavailable if you select Input port for **Log messages from** option.

**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 Log 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. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

# CAN Log

---

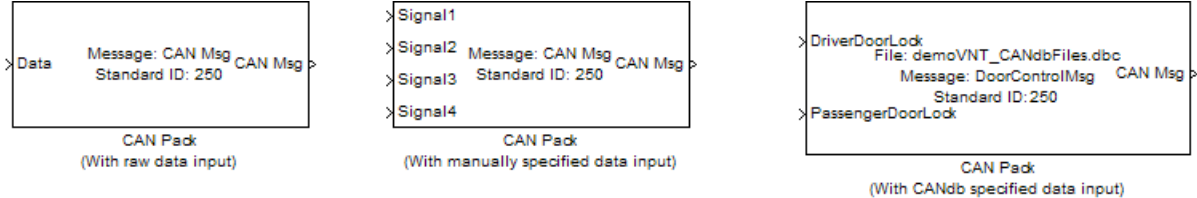
## **See Also**

CAN Replay

**Purpose** Pack individual signals into CAN message

**Library** CAN Communication  
Embedded Coder/ Embedded Targets/ Host 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 Simulink Coder to deploy models to targets.

---

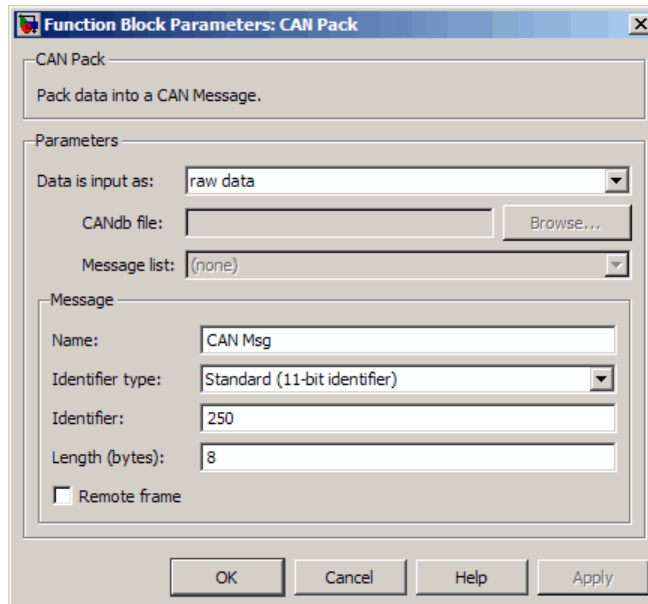
**Note** Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32-bits long.

---

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.



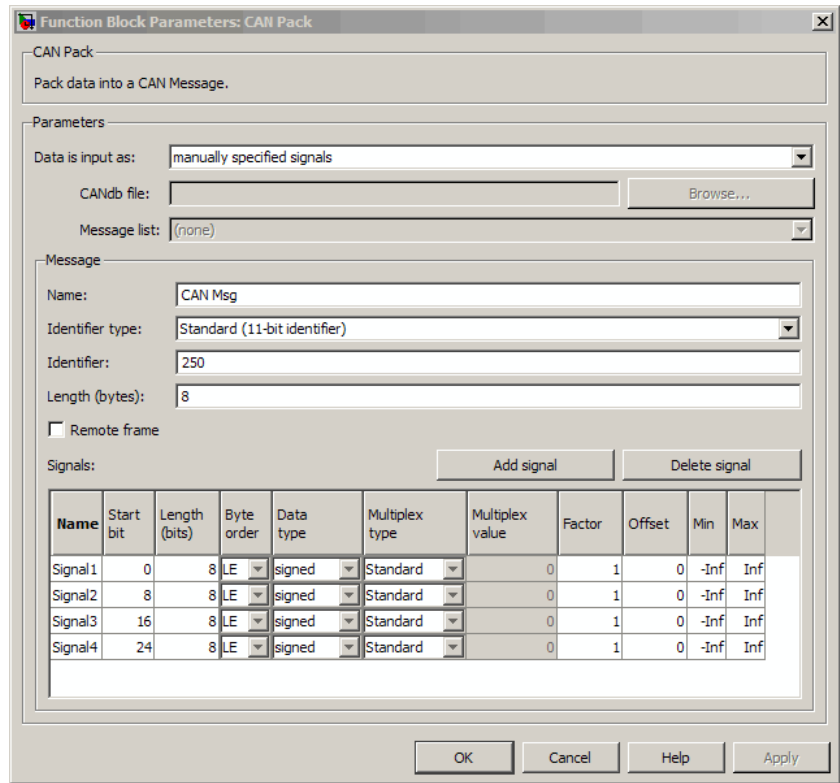
### Parameters

#### 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.

# CAN Pack



The dialog box is titled "Function Block Parameters: CAN Pack". It contains the following sections:

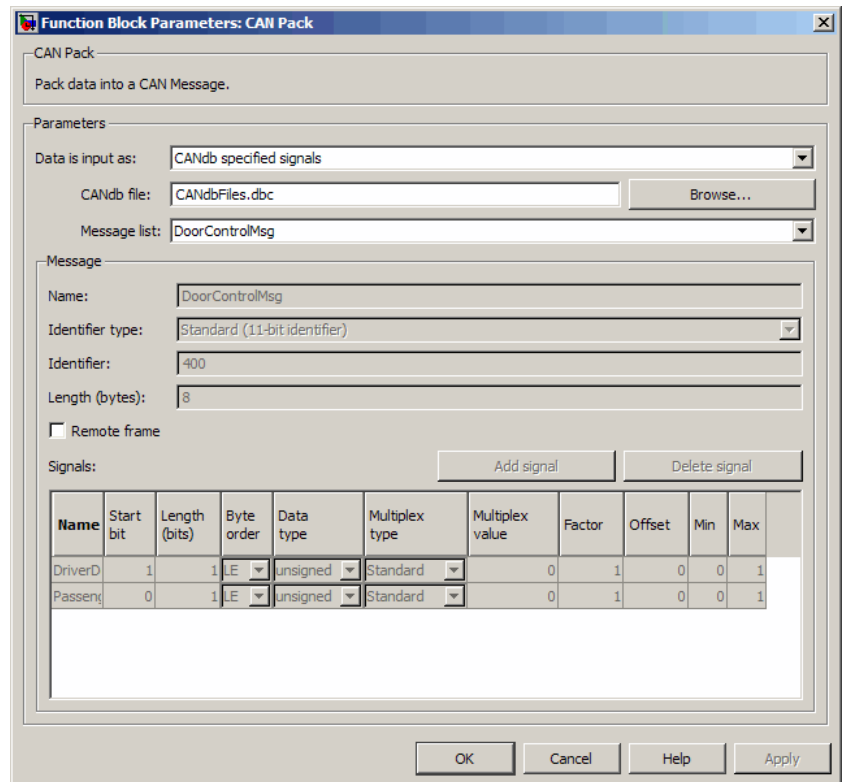
- CAN Pack:** Pack data into a CAN Message.
- Parameters:**
  - Data is input as: manually specified signals
  - CANdb file: [empty] Browse...
  - Message list: (none)
- Message:**
  - Name: CAN Msg
  - Identifier type: Standard (11-bit identifier)
  - Identifier: 250
  - Length (bytes): 8
  - Remote frame
- Signals:** Add signal Delete signal
- Table:**

Name	Start bit	Length (bits)	Byte order	Data type	Multiplex type	Multiplex value	Factor	Offset	Min	Max
Signal1	0	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal2	8	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal3	16	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal4	24	8	LE	signed	Standard	0	1	0	-Inf	Inf

Buttons: OK Cancel Help Apply

- **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.

---

**Note** File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

---

## 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

Data begins at the least significant bit and starts at 20

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

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

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you

pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.

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

### 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 9-21 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 9-21 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 9-21 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 9-21 to understand how physical values are converted to raw values packed into a message.

**Conversion Formula**

The conversion formula is

$$\text{raw\_value} = (\text{physical\_value} - \text{Offset}) / \text{Factor}$$

# CAN Pack

---

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

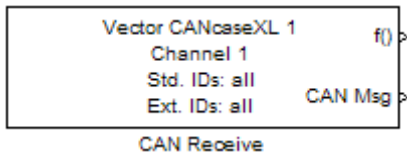
CAN Unpack



**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 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.

---

**Note** You cannot have more than one Receive block connected to the same NI device channel.

---

## 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.

The CAN Receive block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see Code Generation.

## Code Generation

Vehicle Network Toolbox Simulink blocks allow you to generate code, enabling models containing these blocks to run successfully in Accelerator, Rapid Accelerator, External, and Deployed modes.

### Code Generation with the Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder products together to generate code (on the host end) that you can use to implement your model for a practical application. For more information on code generation, see Building Executables.

### Shared Library Dependencies

The block generates code with limited portability. The block uses precompiled shared libraries, such as DLLs, to support I/O for specific types of devices. With this block, you can use the packNGo function supported by the Simulink Coder, to set up and manage the build information for your models. The packNGo function allows you to package model code and dependent shared libraries into a zip file for deployment. You do not need MATLAB installed on the target system, but the target system needs to be supported by MATLAB.

To set up packNGo:

```
set_param(gcs, 'PostCodeGenCommand', 'packNGo(buildInfo)');
```

In this example, `gcs` is the current model that you wish to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and the source code in the zip file can be built to create an executable which can be run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information on `packNGo`, see `packNGo`.

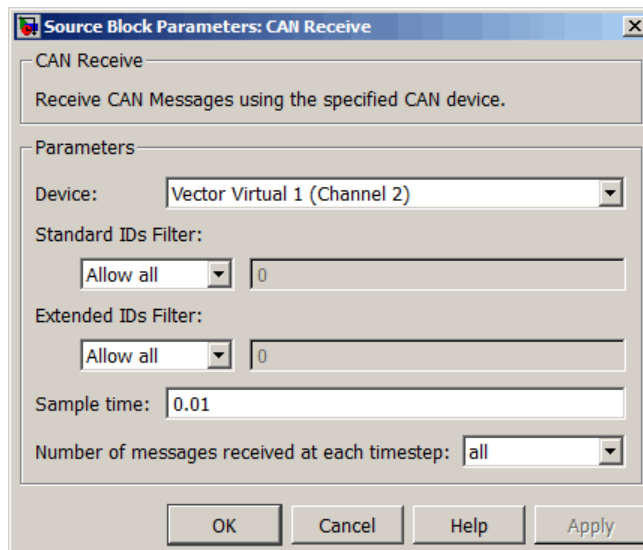
## Dialog Box

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.

## Standard IDs Filter

Select the filter on this block for standard IDs. Valid choices are:

- **Allow all:** allows all standard IDs to pass the filter. This is the default filter state of the CAN Receive block
- **Allow only:** Allows only ID or range of IDs specified 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.
- **Block all:** Blocks all standard IDs from passing the filter.

## Extended IDs Filter

Select the filter on this block for extended IDs. Valid choices are:

- **Allow all:** allows all extended IDs to pass the filter. This is the default filter state of the CAN Receive block
- **Allow only:** Allows only ID or range of IDs specified 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 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.
- **Block all:** Blocks all extended IDs from passing the filter.

## 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. You can also specify a MATLAB variable for 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

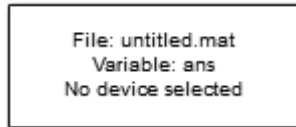
# CAN Replay

---

**Purpose**                      Replay logged CAN messages

**Library**                     Vehicle Network Toolbox: CAN Communication

## Description



CAN Replay

The CAN Replay block replays logged messages from a `.mat` file to a CAN network or to Simulink. You need a CAN Configuration block to replay to the network.

To replay messages logged in the MATLAB Command window in your Simulink model, convert them into a compatible format using `vntslgate` and save it to a separate file. Refer to the **Basic CAN Message Replay and Logging** demo in the Vehicle Network Toolbox Simulink Demos for information.

---

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

---

## Replay Timing

When you replay logged messages, Simulink uses the original timestamps on the messages. When you replay to a network, the timestamps correlate to real time, and when you replay to the Simulink input port it correlates to simulation time. If the timestamps in the messages are all 0, all messages are replayed as soon as the simulation starts, because simulation time and real time will be ahead of the timestamps in the replayed messages.

## Other Supported Features

The CAN Replay 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.

The CAN Log block supports the use of code generation along with the `packNGo` function to group required source code and dependent shared libraries. For more information, see Code Generation.

## Code Generation

Vehicle Network Toolbox Simulink blocks allow you to generate code, enabling models containing these blocks to run successfully in Accelerator, Rapid Accelerator, External, and Deployed modes.

### Code Generation with the Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder products together to generate code (on the host end) that you can use to implement your model for a practical application. For more information on code generation, see Building Executables.

### Shared Library Dependencies

The block generates code with limited portability. The block uses precompiled shared libraries, such as DLLs, to support I/O for specific types of devices. With this block, you can use the `packNGo` function supported by the Simulink Coder, to set up and manage the build information for your models. The `packNGo` function allows you to package model code and dependent shared libraries into a zip file for deployment. You do not need MATLAB installed on the target system, but the target system needs to be supported by MATLAB.

To set up `packNGo`:

```
set_param(gcs, 'PostCodeGenCommand', 'packNGo(buildInfo)');
```

In this example, `gcs` is the current model that you wish to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and the source code in the zip file can be built to create an executable which can be run

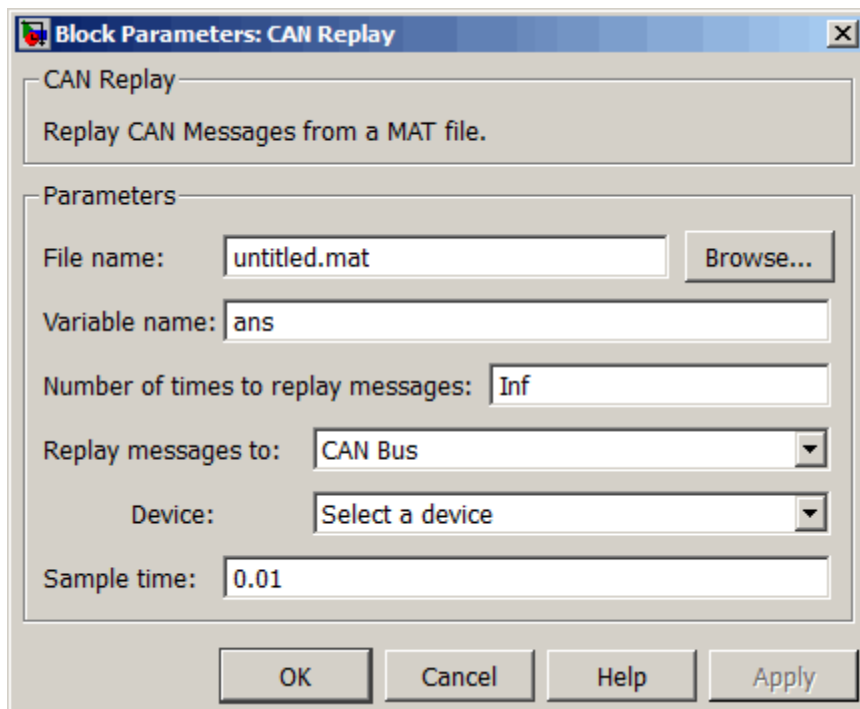
# CAN Replay

independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information on packNGo, see packNGo.

## Dialog Box

Use the Source Block Parameters dialog box to select your message replay options.

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





**File name**

Specify the name and path of the file that contains logged CAN messages that you can replay. You can click **Browse** to browse to a file location and select the file.

**Variable name**

Specify the variable saved in the MAT-file that holds the CAN message information.

**Number of times to replay messages**

Specify the number of times you want the message replayed in your model. You can specify any positive integer, including **Inf**. Specifying **Inf** continuously replays messages until simulation stops.

**Replay messages to**

Specify if the model is replaying messages to the CAN network or an output port. Select a device to replay to the CAN network.

**Device**

Select the device on the CAN network to replay messages to. This field is unavailable if you select **Input port** for **Replay message to** option.

**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 Replay 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. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

**See Also**

CAN Log

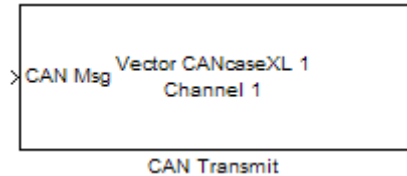
# CAN Transmit

---

**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 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.

The CAN Transmit block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see Code Generation.

## Code Generation

Vehicle Network Toolbox Simulink blocks allow you to generate code, enabling models containing these blocks to run successfully in Accelerator, Rapid Accelerator, External, and Deployed modes.

### Code Generation with the Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder products together to generate code (on the host end) that you can use to implement your model for a practical application. For more information on code generation, see [Building Executables](#).

### Shared Library Dependencies

The block generates code with limited portability. The block uses precompiled shared libraries, such as DLLs, to support I/O for specific types of devices. With this block, you can use the `packNGo` function supported by the Simulink Coder, to set up and manage the build information for your models. The `packNGo` function allows you to package model code and dependent shared libraries into a zip file for deployment. You do not need MATLAB installed on the target system, but the target system needs to be supported by MATLAB.

To set up `packNGo`:

```
set_param(gcs, 'PostCodeGenCommand', 'packNGo(buildInfo)');
```

In this example, `gcs` is the current model that you wish to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and the source code in the zip file can be built to create an executable which can be run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information on `packNGo`, see [packNGo](#).

# CAN Transmit

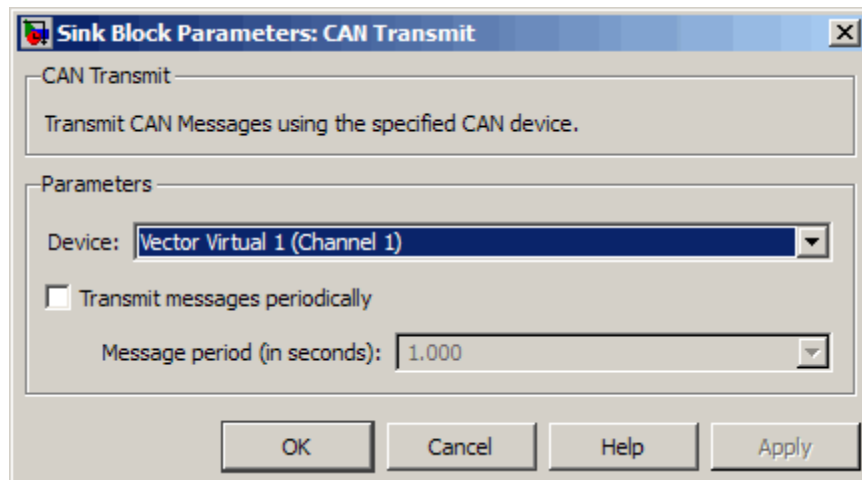
## Dialog Box

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.

### Transmit messages periodically

Select this option to enable periodic transmit of the message on the configured channel to transmit at the specified period.

**Message period (in seconds)**

Specify a period in seconds. This value is used to transmit the message in the specified period. By default this value is 1.000 seconds.

**See Also**

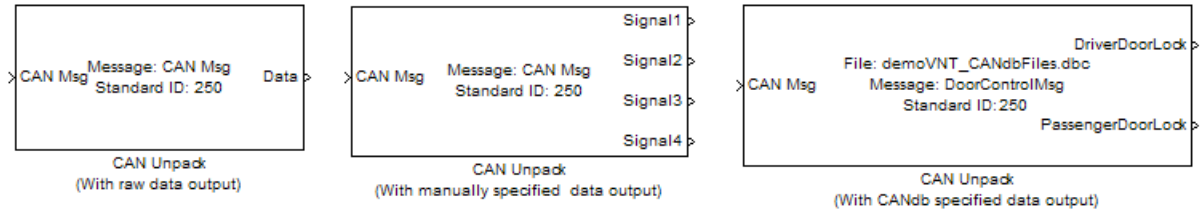
CAN Configuration, CAN Pack

# CAN Unpack

**Purpose** Unpack individual signals from CAN messages

**Library** CAN Communication  
Embedded Coder/ Embedded Targets/ Host 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 Simulink Coder 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.

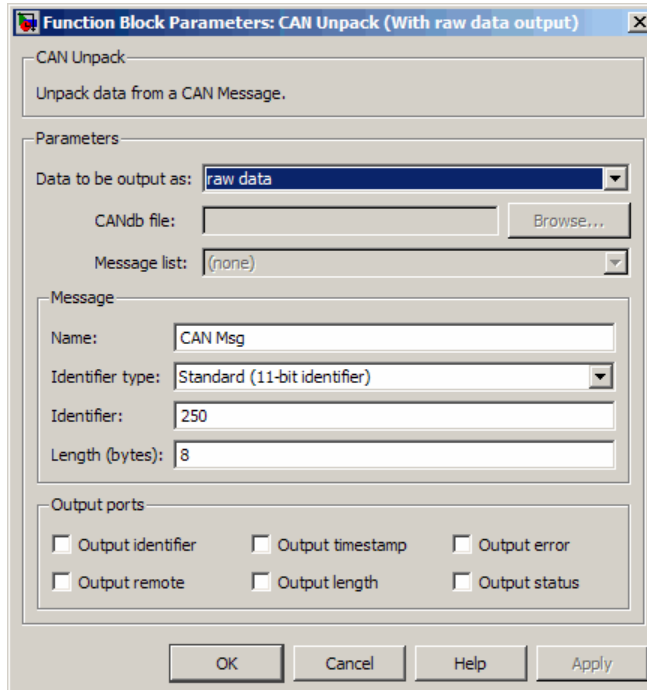
---

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.



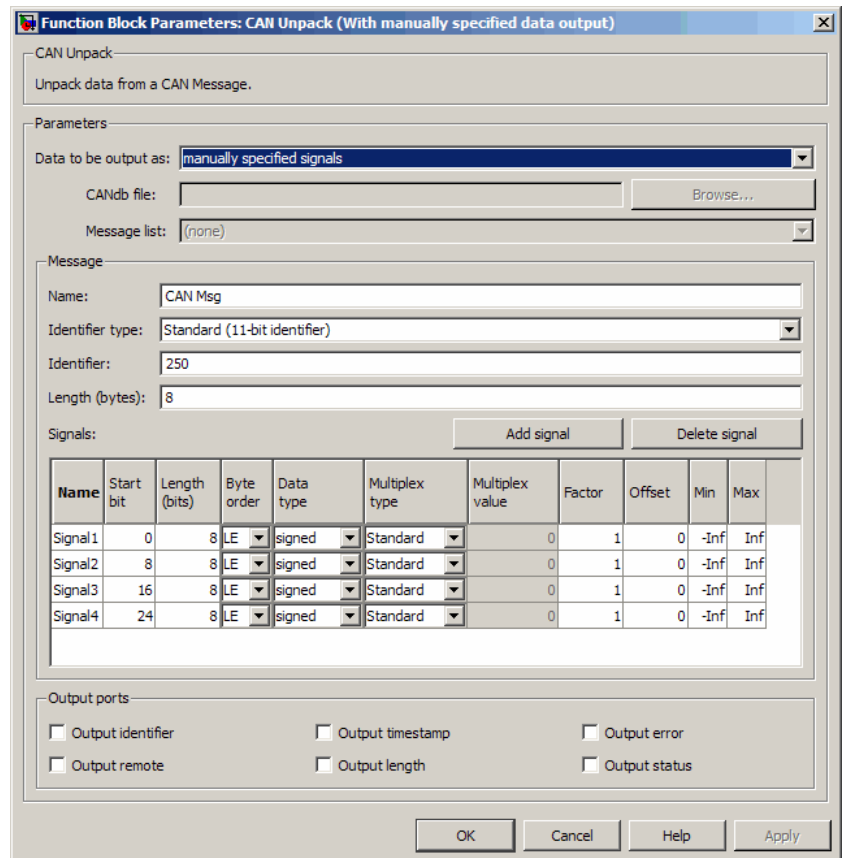
## Parameters

### 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 Browse...

Message list: DoorControlMsg

Message

Name: DoorControlMsg

Identifier type: Standard (11-bit identifier)

Identifier: 400

Length (bytes): 8

Signals: Add signal Delete signal

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

OK Cancel Help Apply

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.

---

**Note** File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

---

### 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 an integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If you specify **1**, the block unpacks all messages that match the length specified for the

message. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

## **Length (bytes)**

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

## **Signals Table**

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

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

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

### **Name**

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

### **Start bit**

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

### **Length (bits)**

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

### **Byte order**

Select either of the following options:

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

Bit Number

Data Byte Number	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	7	6	5	4	3	2	1	0
Byte 1	15	14	13	12	11	10	9	8
Byte 2	23	22	21	20	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

Data begins at the least significant bit and starts at 20

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

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

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the start, which is the least

# CAN Unpack

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.

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

## 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 9-47 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 9-47 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 9-47 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 9-47 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

# CAN Unpack

---

$\text{physical\_value} = \text{raw\_value} * \text{Factor} + \text{Offset}$

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

## A

attachDatabase function 6-2

## B

base properties

list for can channel 7-2

bit timing settings

device-specific properties 7-5

Block Library 4-7

blocks

CAN Configuration 9-2

CAN Log 9-6

CAN Pack 9-11

CAN Receive 9-23

CAN Replay 9-28

CAN Transmit 9-32

CAN Unpack 9-36

using the Vehicle Network Toolbox block

library 4-1

building

CAN messages 1-20

BusLoad property 8-2

BusSpeed property 8-3

BusStatus property 8-4

## C

CAN

transmit message 1-22

workflow 1-12

can channel

base properties 7-2

CAN Channel

interface-specific properties 7-5

CAN channels

configuring properties 1-18

disconnecting 1-31

SilentMode 1-30

starting 1-19

CAN communication

session 1-12

CAN communications

configuring 1-13

CAN Configuration block 9-2

CAN devices

connecting 1-17

CAN Log block 9-6

CAN messages

building 1-20

filtering 1-26

packing 1-21

receiving 1-23

unpacking 1-25

CAN Pack block 9-11

CAN Receive block 9-23

CAN Replay block 9-28

CAN Transmit block 9-32

CAN Unpack block 9-36

can.vector.channel, configBusSpeed

function 6-14

can.vector.channel, fileterBlockRange

function 6-29

can.vector.channel, filterAcceptRange

function 6-27

can.vector.channel, filterAllowAll

function 6-23

can.vector.channel, filterAllowOnly

function 6-24

can.vector.channel, filterBlockAll

function 6-26

can.vector.channel, filterReset

function 6-31

can.vector.channel, filterSet function 6-32

canChannel function 6-3

canChannel, get function 6-34

canChannel, receive function 6-38

canChannel, replay function 6-40

canChannel, set function 6-42

canChannel, start function 6-45

- canChannel, stop function 6-46
- canChannel, transmit function 6-47
- canDatabase function 6-6
- canHWInfo function 6-7
- canMessage function 6-8
- canMessageImport function 6-10
- canSupport function 6-12
- canTool function 6-13
- cleaning
  - MATLAB workspace 1-32
- configuring
  - CAN channel properties 1-18 1-30
  - CAN communications 1-13
  - message filtering 1-26
- connecting
  - CAN devices 1-17

## D

- Data property 8-5
- Database property 8-6
- Device property 8-8 to 8-9
- device-specific properties
  - list by object type 7-5
- DeviceChannelIndex property 8-10
- DeviceSerialNumber property 8-11
- DeviceVendor property 8-12
- discard function 6-16
- disconnecting
  - CAN channels 1-31

## E

- Error property 8-13
- Extended property 8-14
- extractAll function 6-17
- extractRecent function 6-19
- extractTime function 6-21

## F

- filtering
  - CAN messages 1-26
- functions
  - attachDatabase 6-2
  - canChannel 6-3
  - canChannel, transmit 6-47
  - canChannelset 6-42
  - canChannelstart 6-45
  - canDatabase 6-6
  - canHWInfo 6-7
  - canMessage 6-8
  - canMessageImport 6-10
  - canSupport 6-12
  - canTool 6-13
  - configBusSpeed, can.vector.channel 6-14
  - discard 6-16
  - extractAll 6-17
  - extractRecent 6-19
  - extractTime 6-21
  - filterAcceptRange,
    - can.vector.channel 6-27
  - filterAllowAll 6-23 to 6-24
  - filterBlockAll 6-26
  - filterBlockRange,
    - can.vector.channel 6-29
  - filterReset, can.vector.channel 6-31
  - filterSet, can.vector.channel 6-32
  - get, canChannel 6-34
  - messageInfo, canChannel 6-35
  - pack 6-37
  - receive, canChannel 6-38
  - replay, canChannel 6-40
  - signalInfo, canDatabase 6-43
  - stop, canChannel 6-46
  - transmitConfiguration 6-49
  - transmitEvent 6-51
  - transmitPeriodic 6-52
  - unpack 6-54

**I**

ID property 8-15  
 InitializationAccess property 8-16

**M**

MATLAB workspace  
   cleaning 1-32  
 message  
   transmit 1-22  
 message filtering  
   configuring 1-26  
 messageInfo function 6-35  
 MessageReceivedFcn property 8-17  
 MessageReceivedFcnCount property 8-18  
 messages  
   packing 1-21  
   receiving 1-23  
   unpacking 1-25  
 Messages property 8-19  
 MessagesAvailable property 8-20  
 MessagesReceived property 8-21  
 MessagesTransmitted property 8-22

**N**

Name (Database) property 8-23  
 Name (Message) property 8-24  
 NumOfSamples property 8-25

**P**

pack function 6-37  
 packing  
   CAN messages 1-21  
 properties  
   BusLoad 8-2  
   BusSpeed 8-3  
   BusStatus 8-4  
   Data 8-5

Database 8-6  
 Device 8-8 to 8-9  
 DeviceChannelIndex 8-10  
 DeviceSerialNumber 8-11  
 DeviceVendor 8-12  
 Error 8-13  
 Extended 8-14  
 ID 8-15  
 InitializationAccess 8-16  
 MessageReceivedFcn 8-17  
 MessageReceivedFcnCount 8-18  
 Messages 8-19  
 MessagesAvailable 8-20  
 MessagesReceived 8-21  
 MessagesTransmitted 8-22  
 Name (Database) 8-23  
 Name (Message) 8-24  
 NumOfSamples 8-25  
 ReceiveErrorCount 8-26 to 8-27  
 Remote 8-28  
 Running 8-29  
 Signals 8-31  
 SilentMode 8-30  
 SJW 8-32  
 synchronization jump width 8-32  
 Timestamp 8-33  
 TransceiverName 8-34  
 TransceiverState 8-35  
 TransmitErrorCount 8-36  
 TSEG1 8-37  
 TSEG2 8-38  
 UserData 8-39  
 property values  
   base  
     for can channel 7-2  
   device-specific 7-5

**R**

ReceiveErrorCount property 8-26 to 8-27

receiving

    CAN messages 1-23

Remote property 8-28

Running property 8-29

## **S**

signalInfo, signalInfo function 6-43

Signals property 8-31

SilentMode property 8-30

Simulink Library Browser 4-8

SJW property 8-32

starting

    CAN channels 1-19

synchronization jump width

    properties 8-32

## **T**

Timestamp

    properties 8-33

transceiver settings

    device-specific properties 7-5

TransceiverName

    properties 8-34

TransceiverState

    properties 8-35

transmit

    CAN message 1-22

    transmitConfiguration function 6-49

TransmitErrorCount

    properties 8-36

transmitEvent

    functions 6-51

transmitPeriodic function 6-52

TSEG1

    properties 8-37

TSEG2

    properties 8-38

## **U**

unpack function 6-54

unpacking

    CAN messages 1-25

UserData

    properties 8-39

## **V**

Vector CAN device

    device-specific properties 7-5

Vehicle Network Toolbox block library

    using 4-1

Vehicle Network Toolbox Block Library

    opening 4-7