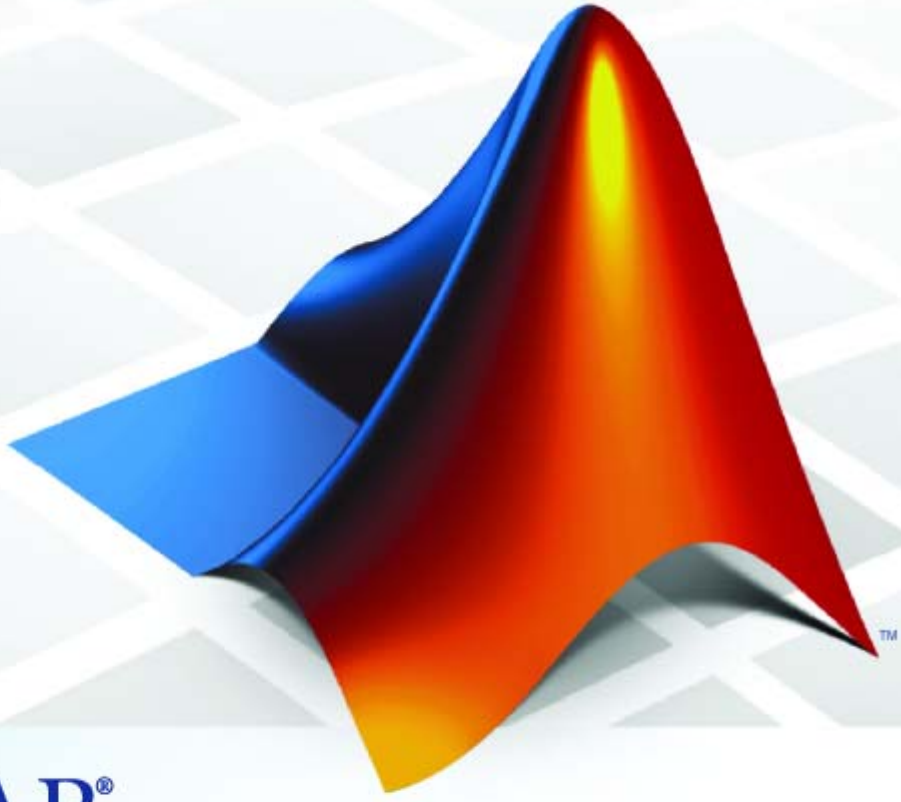
Vehicle Network Toolbox™ 1
User's Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Vehicle Network Toolbox™ User's Guide

Trademarks

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

Patents

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

Revision History

March 2009	Online only	New for Version 1.0 (Release 2009a)

Contents

1 Getting Started

2 Using a CAN Database

3 Monitoring CAN Message Traffic

4 Using the Vehicle Network Toolbox Block Library

Function Reference

5

Functions — Alphabetical List

6

Property Reference

7

1

Getting Started

Product Overview

In this section...

Getting to Know the Vehicle Network Toolbox

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

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

Main Features

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

The toolbox provides you with these main features:

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

CAN Connectivity

The Vehicle Network Toolbox provides host-side CAN connectivity using defined CAN devices. CAN is the predominant protocol in automotive electronics by which many distributed control systems in a vehicle function. For example, in a common design when you press a button to lock the doors in your car, a control unit in the door reads that input and transmits lock commands to control units in the other doors. These commands exist as data in CAN messages, which the control units in the other doors receive and act on by triggering their individual locks in response.

Vector Device and Driver Support

You can use the Vehicle Network Toolbox with devices supported by Vector. These devices and drivers provide a link to the CAN bus on which you can send and receive messages. See "Supported Hardware" on page 1-7 for more information.

Vehicle Network Toolbox Functions

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

Simulink Library Support

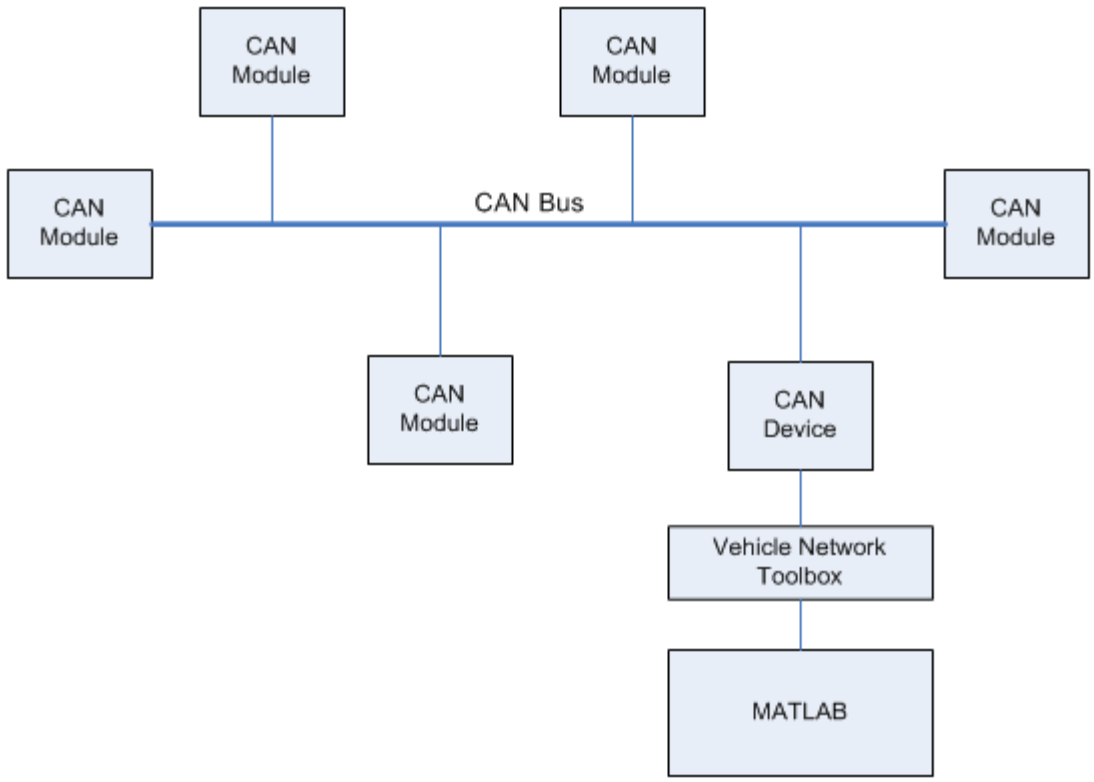
With the Vehicle Network Toolbox block library and other blocks from the Simulink library, you can create sophisticated models to connect to a live network and to simulate message traffic on a CAN bus.

CAN Tool Interface

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

Interaction Between the Toolbox and Its Components

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



In this illustration:

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

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

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

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

Expected Background

This document assumes that you are already familiar with the following products:

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

Related Products

The MathWorks™ provides several products that are relevant to the kinds of tasks you can perform with the Vehicle Network Toolbox software and that extend the capabilities of MATLAB. For information about these related products, see toolbox product page on the MathWorks Web site.

Installation Requirements

Installing Components

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

- Current MATLAB version
- Vehicle Network Toolbox software
- Vector hardware, drivers, and XL driver library

Installing Hardware Devices and Drivers

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

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

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

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

Installing the XL Driver Library

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

Installing the Toolbox

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

```
ver
```

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

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

```
http://www.mathworks.com/support
```

Supported Hardware

The Vehicle Network Toolbox supports the following Vector devices:

- CANcaseXL
- CANboardXL
- CANboardXL pxi
- CANboardXL PCIe
- CANcardXL
- CANcardX

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

CAN Communication Session

In this section...

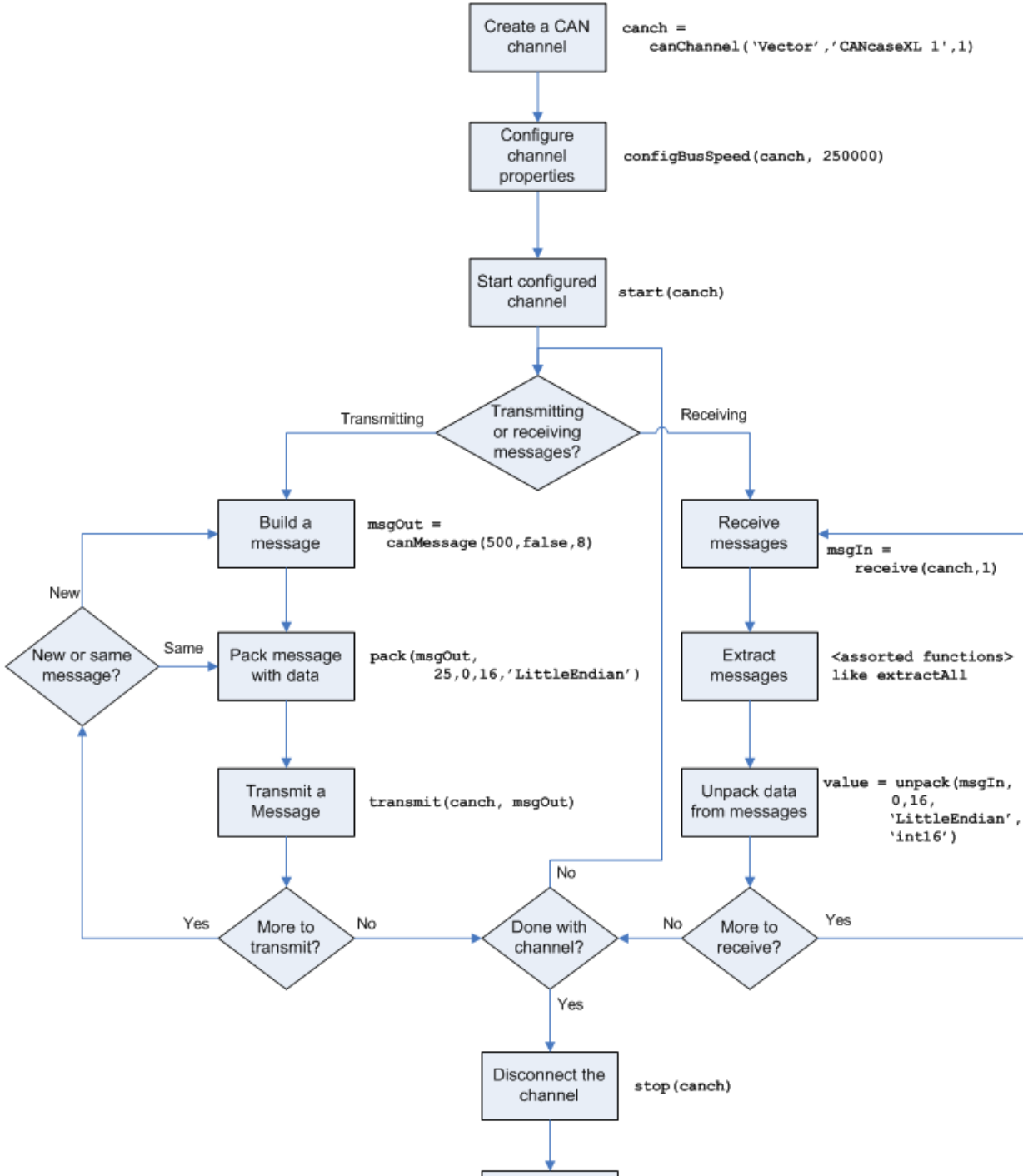
Workflow Overview

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

The subsequent sections map to the following CAN workflow chart.

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

Typical CAN Workflow



Configuring CAN Communications

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

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

Prerequisites

Before you follow this example, make sure you:

- Complete your Toolbox Installation before you try out the examples.
- Connect the two channels in your CAN device in a loopback.

Checking for the Installed CAN Hardware

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

```
info = canHWInfo
```

MATLAB displays the following information:

```
info =

CAN Devices Detected:
  Vector Devices:
    CANcaseXL 1 Channel 1
       To connect, use - canChannel('Vector', 'CANcaseXL 1', 1)
    CANcaseXL 1 Channel 2
       To connect, use - canChannel('Vector', 'CANcaseXL 1', 2)
    Virtual 1 Channel 1
       To connect, use - canChannel('Vector', 'Virtual 1', 1)
    Virtual 1 Channel 2
       To connect, use - canChannel('Vector', 'Virtual 1', 2)
```

2 You can get details about all available CAN channels by typing:

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

Press **Enter** and MATLAB displays information like:

```
          can.vector.ChannelInfo handle
 Package: can.vector

 Properties:
              Device: 'CANcaseXL 1'
   DeviceChannelIndex: 1
   DeviceSerialNumber: 24811
    ObjectConstructor: 'canChannel('Vector', 'CANcaseXL 1', 1)'
```

Creating a CAN Channel Object

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

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

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

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

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

---

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

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

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

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

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

3 Create a second CAN channel object.

```
canch1 = canChannel('Vector','CANcaseXL 1',2)
```

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

Configuring Properties

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

Tip Configure property values before you start the channel.

---

1 Display the properties on `canch`:

```
get (canch)
```

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(canch, 250000)
```

3 To see the changed property value, type:

```
get(canch)
```

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 (`canch1`) by repeating steps 2 and 3.

Starting the Configured Channel

Start your CAN channels after you configure all properties.

1 Start the first channel:

```
start(canch)
```

2 Start the second channel:

```
start(canch1)
```

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:

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

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

Creating a Message Object

After 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:

```
can.Message (Normal Frame)

          ID: 500 / 1F4 (Hex)
    Extended: 0

        Data: [   0   0   0   0   0   0   0   0 ]
              [  00  00  00  00  00  00  00  00 ] (Hex)
```

The fields in the message show:

- **can.Message (Normal Frame)** — Specifies that the message is not an error or a remote frame.
- **ID** — The ID you specified and its hexadecimal equivalent.

- **Extended** — A logical `0` (false) because you did not specify an extended ID.
- **Data** — A uint8 array of 0s specified by the data length.

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

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

Packing a Message

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

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

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

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

2. To see the packed data, type:

   ```
   message
   ```

MATLAB displays your message properties with the specified data:

```
can.Message (Normal Frame)

          ID: 500 / 1F4 (Hex)
    Extended: 0

        Data: [  25   0   0   0   0   0   0   0 ]
              [  19  00  00  00  00  00  00  00 ] (Hex)
```

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

Transmitting a Message

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

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

   ```
   transmit(canch, messageout)
   ```

2. To display the channel status, type:

   ```
   canch
   ```

   MATLAB displays the updated status of the channel:

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

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

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

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

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

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

Receiving a Message

After your channel is online, use the `receive` function to receive available messages. For this example, receive the message on the second configured channel object, `canch1`.

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

```
canch1
```

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 and store it as `messagein` on `canch1`, type:

```
messagein = receive(canch1, 1)
```

MATLAB returns the received message properties:

```
can.Message (Normal Frame)

          ID: 500 / 1F4 (Hex)
    Extended: 0
   Timestamp: 6.999441e+000

        Data: [  25   0   0   0   0   0   0   0 ]
              [  19  00  00  00  00  00  00  00 ] (Hex)
```

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

```
canch1
```

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

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

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

Unpacking a Message

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

```
value = unpack(message, 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.

Disconnecting Channels and Cleaning Up

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(canch)
```

2 Check the channel status:

```
canch
```

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 (canch1)
```

4 Check the channel status:

```
canch1
```

MATLAB displays the channel status:

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

Cleaning Up the MATLAB Workspace

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

1 Clear the first channel:

```
clear canch
```

2 Clear the second channel:

```
clear canch1
```

3 Clear the CAN messages:

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

4 Clear the unpacked value:

```
clear value
```

Performing Advanced Configurations

Configuring Message Filtering

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

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

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

```
stop (canch)
filterAcceptRange (canch, 500, 625)
start (canch)
```

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

```
canch


canch =
```

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

.
.
.
Filter History:Filters are open for Standard and Extended IDs.
              Block Range added. Starting ID:0 Ending ID:2047
              Accept Range added. Starting ID:500 Ending ID:625
```

Configuring Multiplexing

Use multiplexing to combine multiple signals into one signal and transmit it on the CAN bus. 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 if a multiplexed signal is packed. You can create only one multiplexor signal per message.

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

When you multiplex a message, you can specify both standard and multiplexed signals. While standard signals are always packed into the message, a multiplexed signal is either packed or ignored, depending on whether its multiplex value matches the value of a multiplexor signal.

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

1. `SigA`: A multiplexed signal with a multiplex value of `0`.
2. `SigB`: Another multiplexed signal with a multiplex value of `1`.
3. `MuxSig`: A multiplexor signal, whose incoming value determines which of the two multiplexed signals are active (are packed) in the message.

To try this example, create messages and signals using definitions in your own database.

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 including multiplex signals:

```
can.Message (Normal Frame)

          ID: 250 / FA (Hex)
    Extended: 0
        Name: 'Msg'

        Data: [   0   0   0   0   0   0   0   0 ]
              [  00  00  00  00  00  00  00  00 ] (Hex)

        MuxSig: 0 (Muxor)
          SigA: 0  (Active)
          SigB: N/A
```

`SigA` is active (or packed into the message) because its multiplex current value of `0` matches the value of `MuxSig` (which is `0`).

3. Change the value of the `MuxSig` to `1`:

```
m.MuxSig = 1
```

The message displays its properties with changed signal states:

```
can.Message (Normal Frame)

         ID: 250 / FA (Hex)
   Extended: 0
       Name: 'Msg'

       Data: [   1   0   0   0   0   0   0   0 ]
             [  01  00  00  00  00  00  00  00 ] (Hex)

       MuxSig: 1 (Muxor)
         SigA: N/A
         SigB: 0  (Active)
```

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

4 Change the value of `MuxSig` to `2`:

```
m.MuxSig = 2
```

the message displays its properties with changed signal states:

```
can.Message (Normal Frame)

         ID: 250 / FA (Hex)
   Extended: 0
       Name: 'Msg'

       Data: [   2   0   0   0   0   0   0   0 ]
             [  02  00  00  00  00  00  00  00 ] (Hex)

       MuxSig: 2 (Muxor)
         SigA: N/A
         SigB: N/A
```

Neither of the signals are active because the current value of `MuxSig` does not match the multiplex value of either `SigA` or `SigB`.

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

Configuring Silent Mode

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

1. Create a CAN channel object `canch` and display its properties:

   ```
   get(canch)
   ```

   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`:

```
canch.SilentMode = true
```

3 To see the changed property value, type:

```
get(canch)
```

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

```
SilentMode = 1
```

Accessing the Toolbox

In this section...

Exploring the Toolbox

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

```
help vnt
```

Getting Help

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

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

```
doc vnt
```

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

```
doc function_name
```

Viewing Examples

Examples in this guide use the Vector CANCaseXL device, with the XL Hardware Driver Version 6.3. The Examples index in the Help browser lists these examples.

2

Using a CAN Database

Vector CANdb Support

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

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

Loading and Creating Messages Using the .dbc File

In this section...

Loading the CAN Database

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

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

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

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

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

Creating a CAN Message

This example shows you how to create a message using a database constructed specifically for this purpose. This database has one message, `Msg`. To try this example, create messages and signals using definitions in your own database.

1. Create the CAN database object:

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

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

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

3. Create a CAN message using the message ID in the database:

```
message1 = canMessage(d, 250, false)
```

Adding a Database to a CAN Channel

To add a database to a CAN channel, type:

```
canch.Database = 'Mux.dbc'
```

For more information, see the `Database` property.

Other Uses of the CAN Database

In this section...

Viewing Messages Information in the CAN Database

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

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

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

```
msgInfo =

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

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

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

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

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

msgInfo =
```

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

Here the function returns information about message with name `EngineMsg` in the database `db`. You can also use the message ID to get information 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 `messageInfo` function.

Viewing Signal Information in a CAN Message

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

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

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

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

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

Attaching a CAN Database to Existing Messages

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

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

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

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

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

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

For more information, see the `attachDatabase` function.

3

Monitoring CAN Message Traffic

The CAN Tool

In this section...

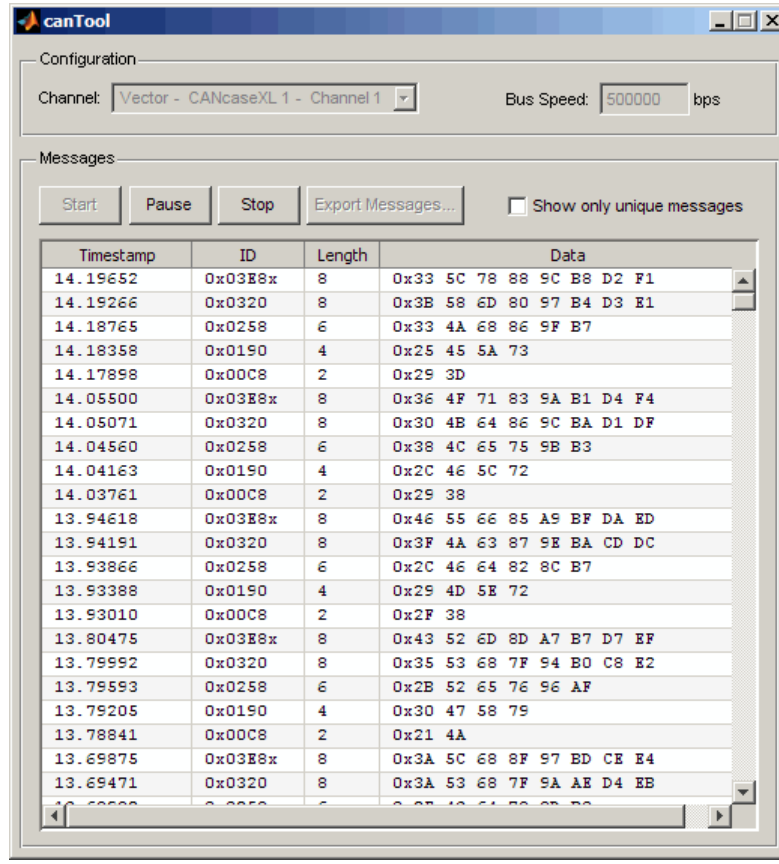
Opening the CAN Tool

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

To open the CAN Tool type `canTool` at the MATLAB command line.

Parts of the CAN Tool

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



Configuration

Channel

Displays all available CAN devices and channels on your system.

Bus Speed

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

Messages

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

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

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

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

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

Messages Table

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

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

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

Length
Displays the length of the message in bytes.

Data
Displays the data in the message in hexadecimal format.

Using the CAN Tool

In this section...

Viewing Messages on a Channel

To view messages on a channel:

1. Open the CAN Tool and select the device and channel connected to your CAN bus from the **Channel** list.
2. The CAN Tool defaults to the bus speed set in the device driver. You can also configure a new bus speed. See Configuring the Channel Bus Speed
3. Click **Start**.

Click **Pause** to pause the display.

Click **Stop** to stop the display.

Configuring the Channel Bus Speed

Configure the bus speed when the speed of your network differs from the default value of the channel. You require initialization is access for the channel to configure the bus speed, otherwise the option is disabled. If you enter an invalid value, it will return to the last valid value.

To configure a new bus speed:

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

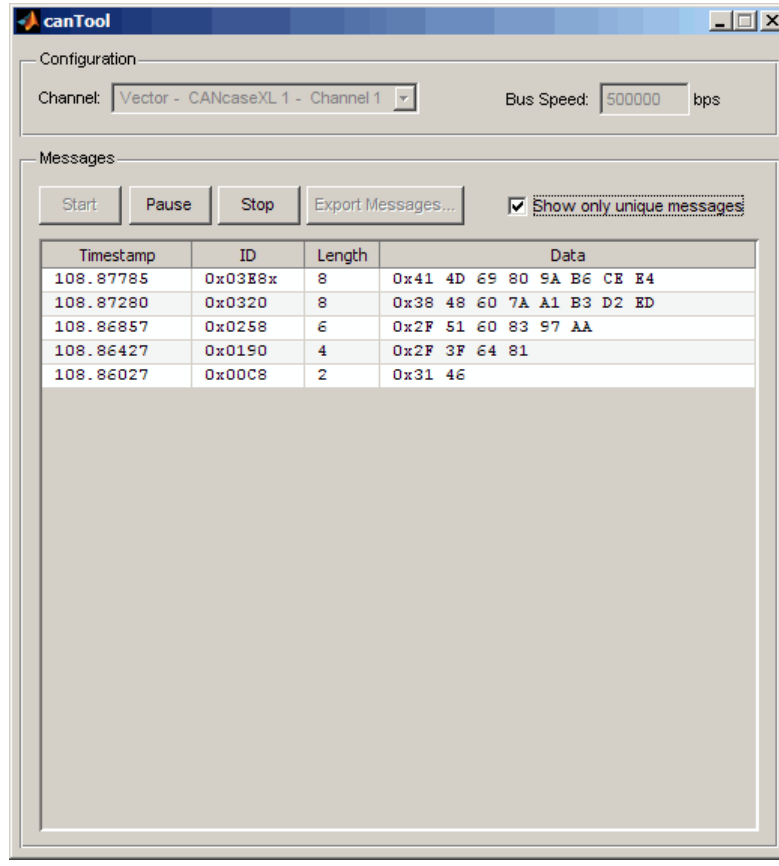
Saving the Message Log File

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

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

Viewing Unique Messages

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



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

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

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

4

Using the Vehicle Network Toolbox Block Library

Introducing the Vehicle Network Toolbox Block Library

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

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

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

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

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

Opening the Vehicle Network Toolbox Block Library

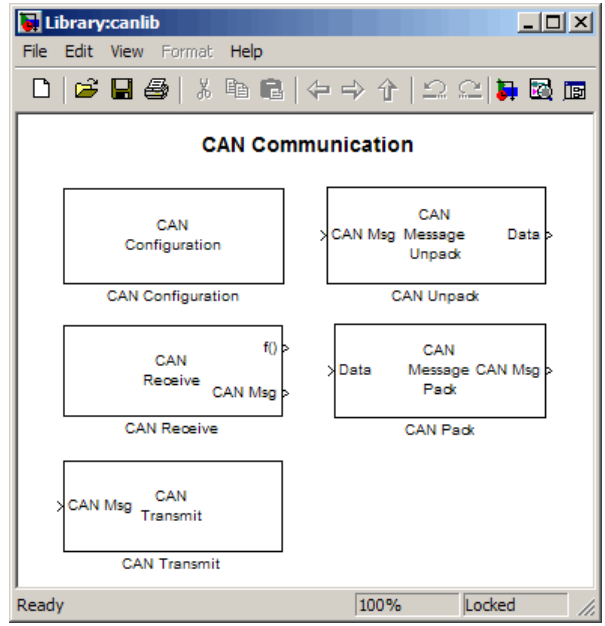
In this section...

Using the canlib Command from the MATLAB Command Window

To open the Vehicle Network block library, enter

```
canlib
```

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



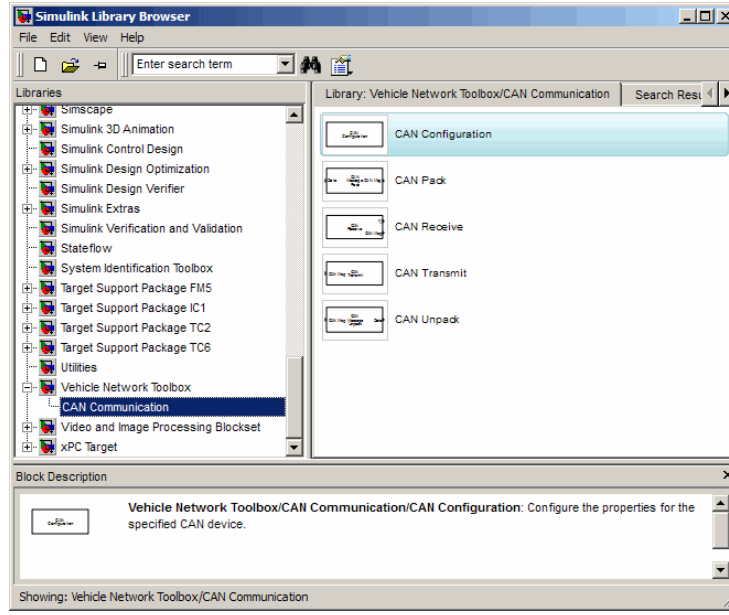
Using the Simulink Library Browser

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

To start the Simulink Library Browser, enter

```
simulink
```

at the MATLAB Command Window. MATLAB opens the browser window. The left pane lists available block libraries, with the basic Simulink library listed first, followed by other libraries listed in alphabetical order 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

This section provides an example that builds a simple model using the Vehicle Network Toolbox blocks with other blocks in the Simulink library. The 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 a CAN Transmit block to send the data to the virtual CAN channel.

Use this section in combination with the "Build a Message Receive Model" on page 4-11, and the "Save and Run The Model" on page 4-19 to build your complete model and run the simulation.

Step 1: Open the Block Library

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

To start the Simulink Library Browser, enter

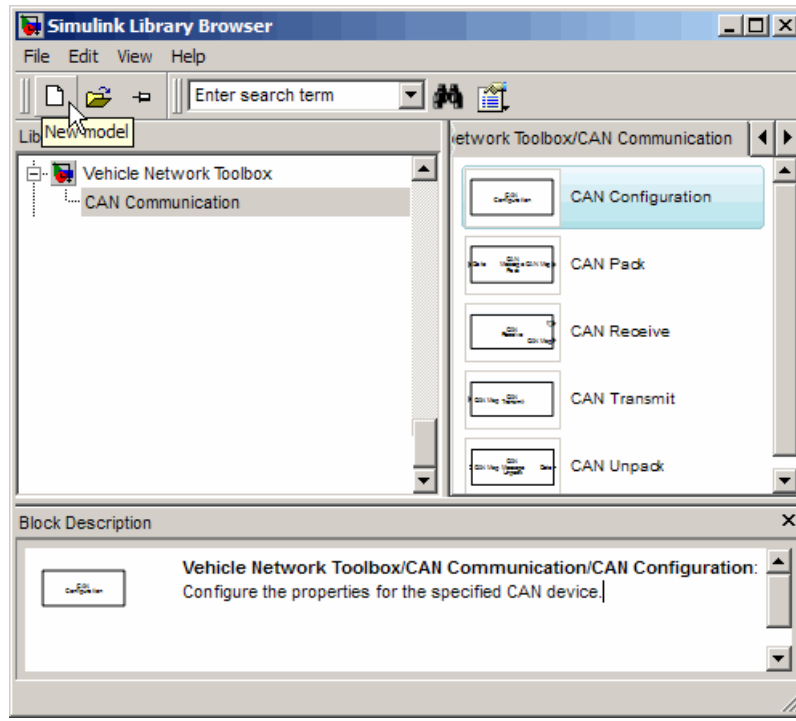
```
simulink
```

at 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 entry 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.

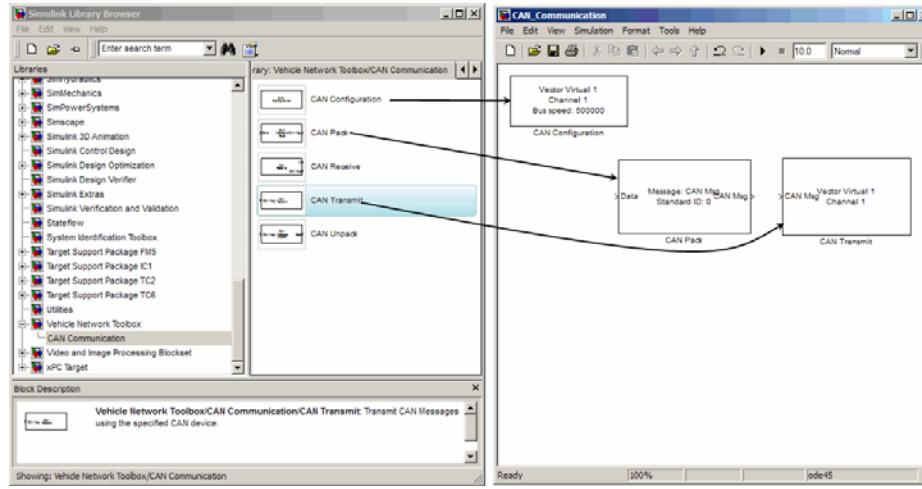
For this example, create a model by clicking the **New model** button on the toolbar.



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

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

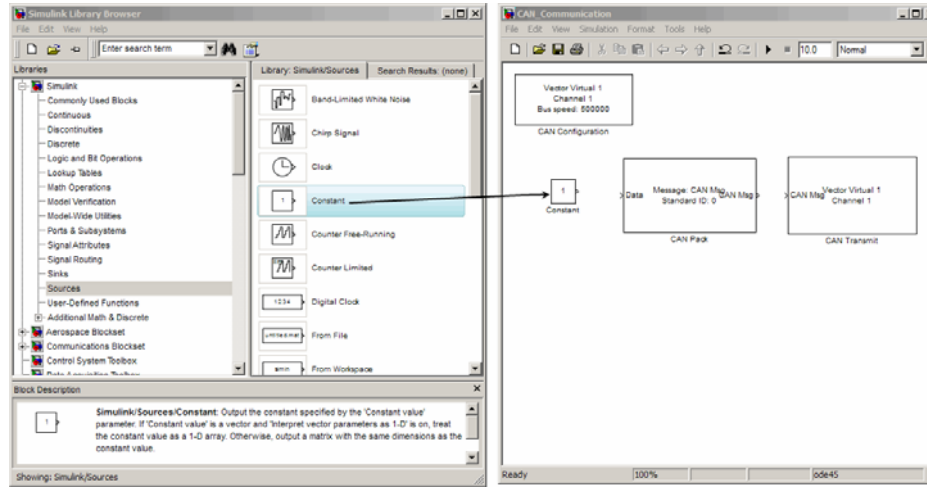
To use the blocks in a model, click a block in the library and, holding the mouse button down, drag it into the model window. For this example, you need one instance each of the CAN Configuration, CAN Pack, and the CAN Transmit block in your model.



Drag Vehicle Network Toolbox™ Blocks into Model Window

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 into your model.



Drag Constant Block to the Model Window

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 on 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 **Apply**, then **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 **Apply**, then **OK**.

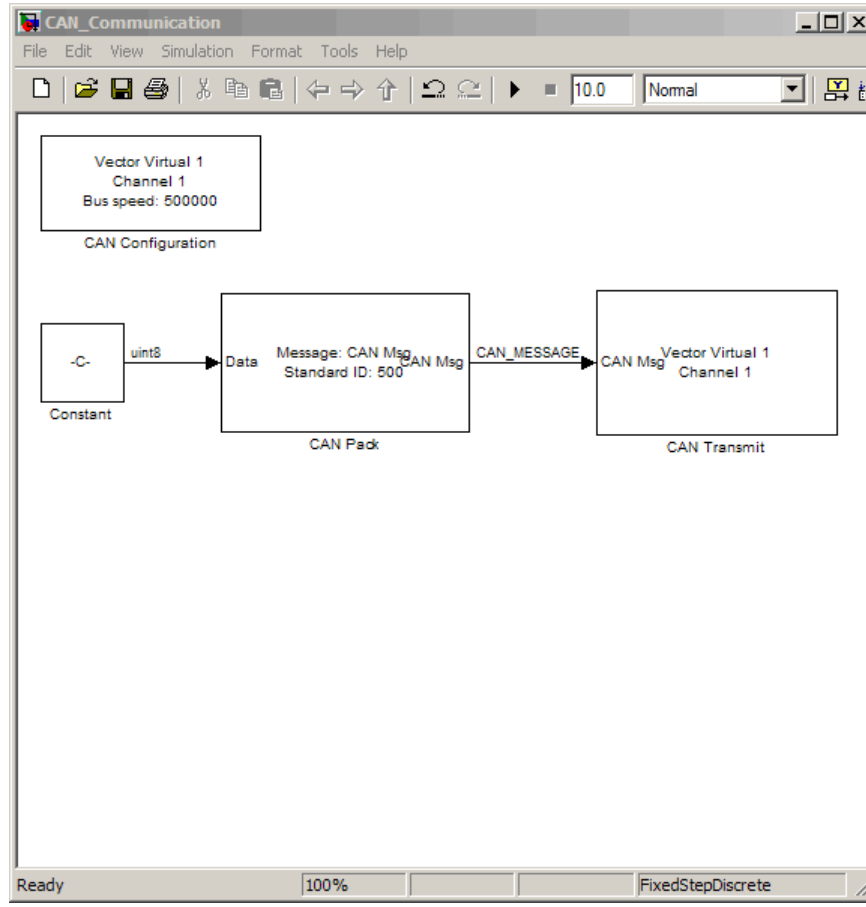
Configure the CAN Transmit Block. Double-click the CAN Transmit to open its parameters dialog box. Set **Device** to `Vector Virtual 1 (Channel 1)`. Click **Apply**, then **OK**.

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`.

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. The example illustrates how to receive data via a CAN network.

- Use a virtual CAN channel to receive messages.
- You use the CAN Configuration block to configure your virtual CAN channels.

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

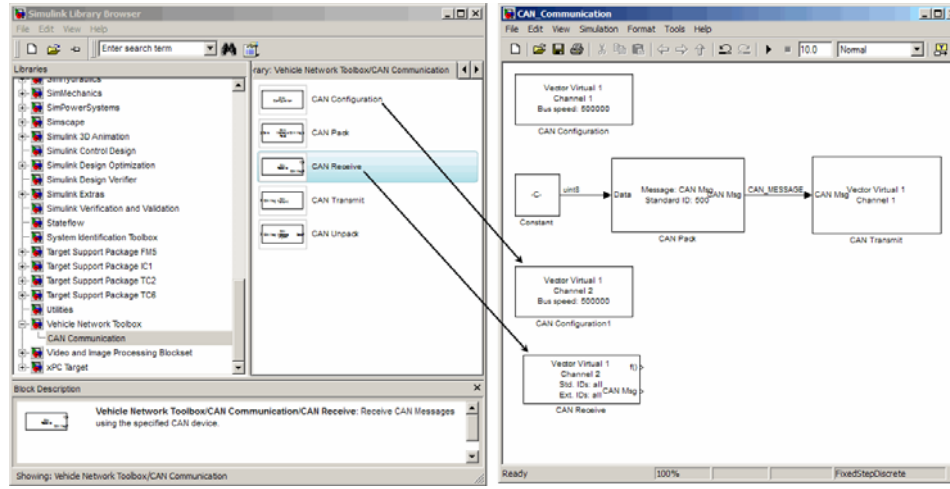
Use this section in combination with the "Build a Message Transmit Model" on page 4-5, and the "Save and Run The Model" on page 4-19 to build your complete model and run the simulation.

- "Step 7: Drag the Vehicle Network Toolbox Blocks into the Model" on page 4-12
- "Step 8: Drag Other Blocks to Complete the Model" on page 4-13
- "Step 9: Connect the Blocks" on page 4-16
- "Step 10: Specify the Block Parameter Values" on page 4-17

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

For this example, you need one instance each of the CAN Configuration, the CAN Receive, and the CAN Unpack block 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 4-13.

Note Configure a separate CAN channel for the CAN Receive and CAN Unpack blocks.

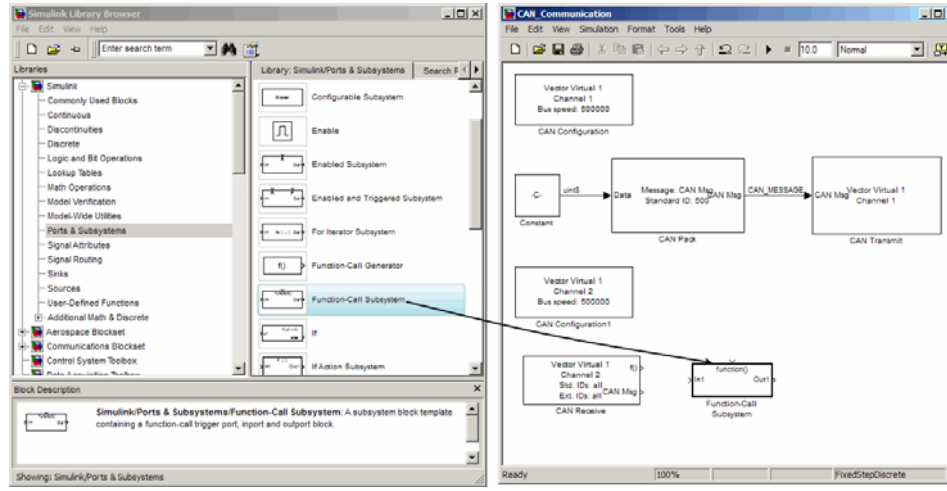


Drag Vehicle Network Toolbox™ Blocks into Model Window

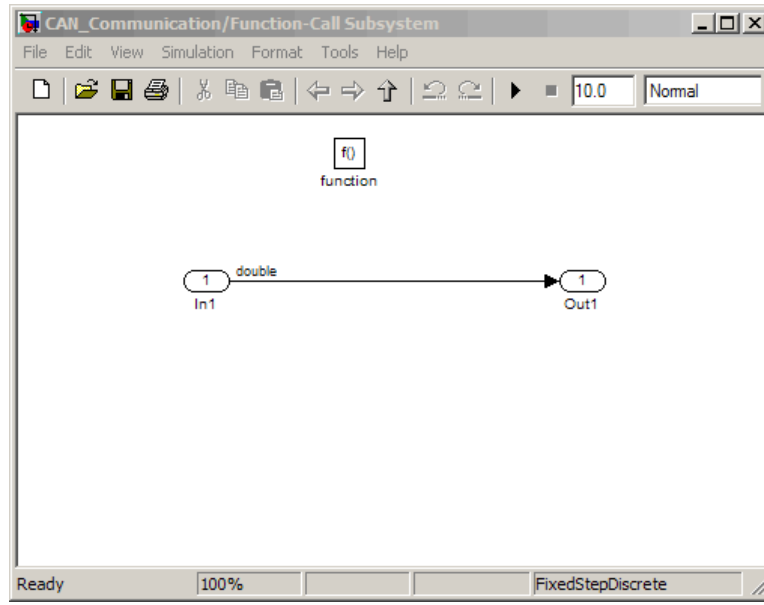
Step 8: Drag Other Blocks to Complete the Model

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

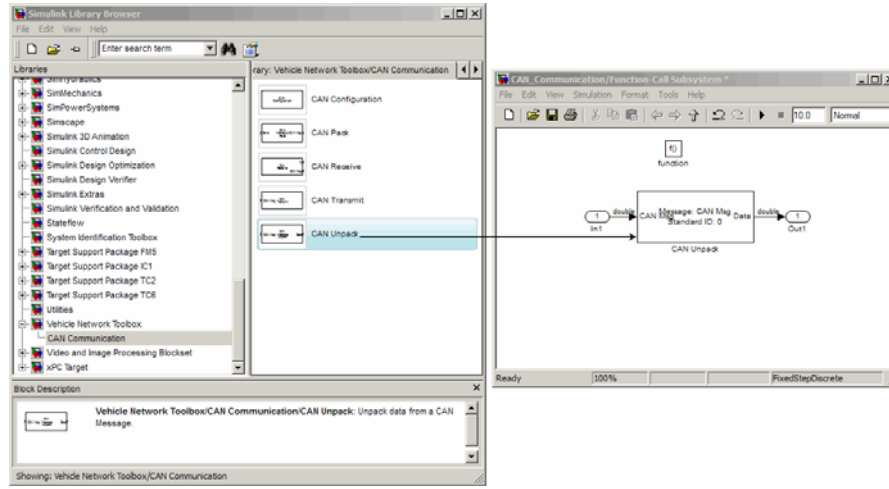
1 Drag the Function–Call Subsystem block into the model.



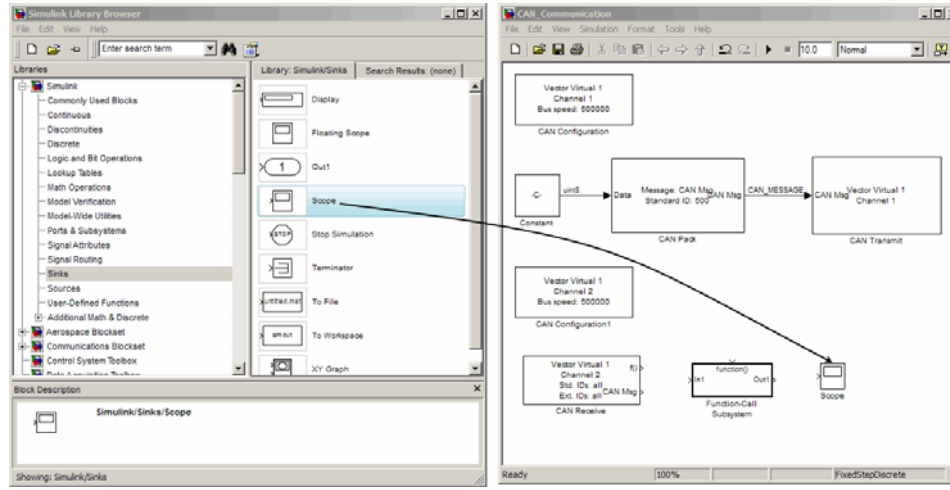
2 Double-click the Function–Call Subsystem block to open the subsystem model.



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



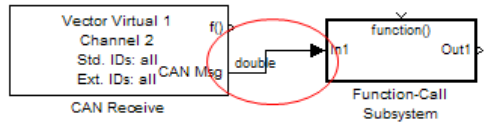
To see the results of the simulation visually, drag the Scope block from the Simulink block library into your model.



Drag The Scope Block into Model Window

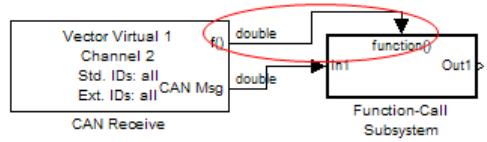
Step 9: Connect the Blocks

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



2 Rename **In1** to `CAN Msg`

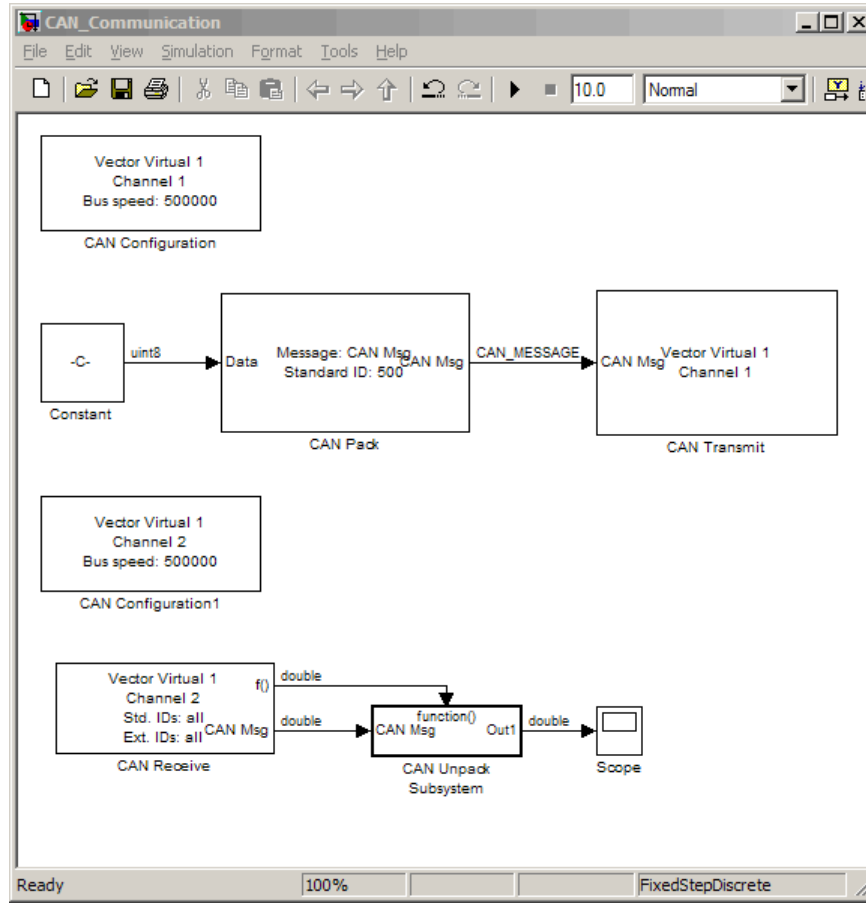
3 Connect the **f()** output port on the CAN Receive block to the **function()** input port on the Function–Call Subsystem block.



4 Rename the Function–Call Subsystem to `CAN Unpack Subsystem`.

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

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

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

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

Click **Apply**, then **OK**.

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

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

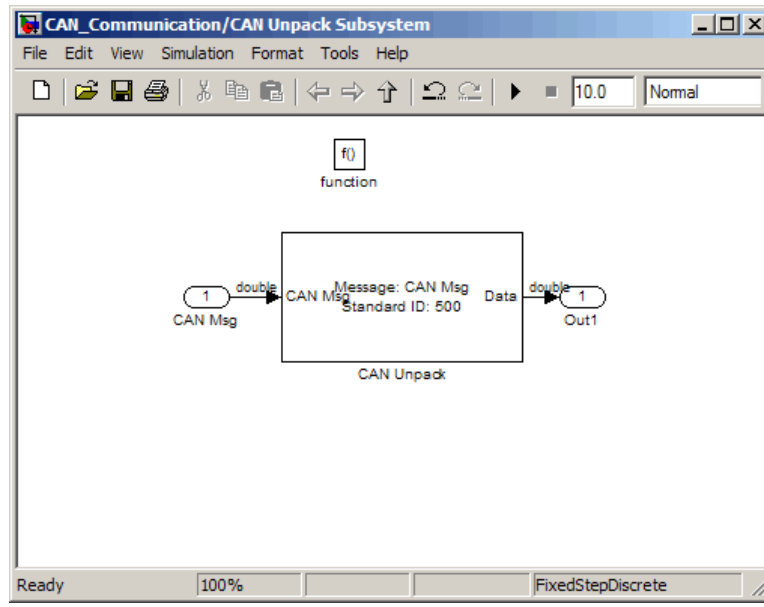
Click **Apply**, then **OK**.

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:

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

Click **Apply**, then **OK**.

Your subsystem looks like this figure:



Save and Run The Model

This section shows you how to save the models you have built in the previous two sections, "Build a Message Transmit Model" on page 4-5 and "Build a Message Receive Model" on page 4-11.

Step 11: Save the Model

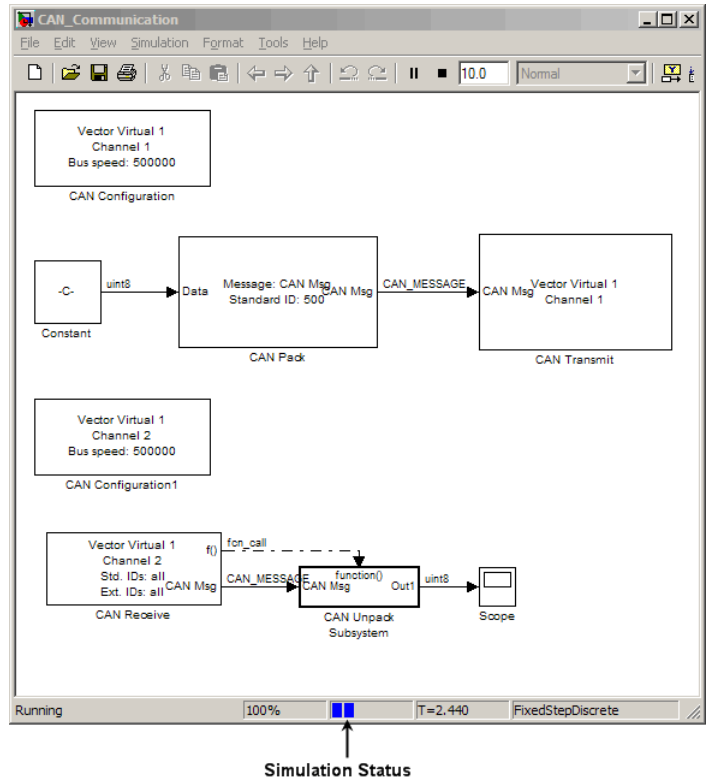
Before you run the simulation, save your model by clicking the **Save** icon or selecting **File > Save** from the menu.

Step 12: 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.

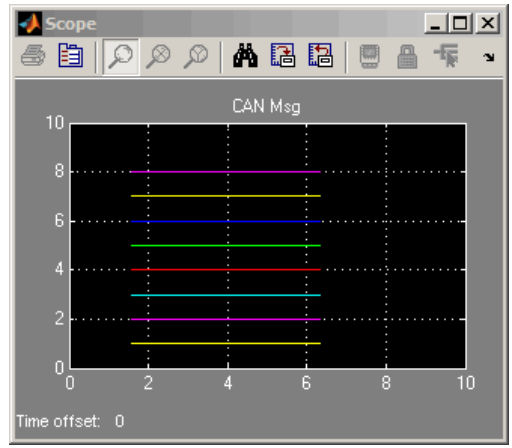
When you run the simulation, the CAN Transmit block gets the message from the CAN Pack block. It then transmits it via Virtual channel 1. The CAN Receive block on Virtual Channel 2 receives this message and hands it to the CAN Unpack block to unpack the message.

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



Step 13: View the Results

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



5

Function Reference

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

`receive`	Receive messages from CAN bus
`receiveRaw`	Receive raw messages from CAN bus
`replay`	Retransmit messages from CAN bus
`start`	Set CAN channel online
`stop`	Set CAN channel offline
`transmit`	Send CAN messages to CAN bus

CAN Channel Status

CAN Database

`canDatabase`	Create handle to CAN database file
`messageInfo`	Information about CAN messages
`signalInfo`	Information about signals in CAN message

CAN Message Handling

`attachDatabase`	Attach CAN database to messages and remove CAN database from messages
`canMessage`	Build CAN message based on user-specified structure
`extractAll`	Select all instances of message from array of messages
`extractRecent`	Select most recent message from array of messages
`extractTime`	Select messages occurring within specified time range from array of messages
`pack`	Pack signal data into CAN message
`unpack`	Unpacks signal data from message

Information and Help

`canHWInfo`	Information on available CAN devices
`canSupport`	Generate technical support log

Graphical Tools

`canTool`	Open CAN Tool

Vector Informatik

These functions are specific to the Vector Informatik `CAN` device.

`configBusSpeed`	Set bit timing rate of CAN channel
`filterAcceptRange`	Set range of CAN identifiers to pass acceptance filter
`filterBlockRange`	Set range of CAN identifiers to block via acceptance filter
`filterReset`	Open CAN message acceptance filters
`filterSet`	Set specific CAN message acceptance filter configuration

6

Functions — Alphabetical List

attachDatabase

Purpose

Attach CAN database to messages and remove CAN database from messages

Syntax

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

Arguments

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

Description

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

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

Remarks

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

Examples

```
candb = canDatabase('C:\Database.dbc')
msg = receive(canch, Inf)
attachDatabase(messsage, candb)
```

See Also

`canDatabase`, `receive`

Purpose

Construct CAN channel connected to selected device

Syntax

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

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.

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'`.

Use `canHWInfo` to obtain a list of available devices.

Remarks

The Vehicle Network Toolbox currently supports Vector devices.

- `CANboardXL_PCIe`
- `CANboardXL_PXI`
- `CANcardX`
- `CANcardXL`
- `CANcaseXL`
- `Virtual`

Examples

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

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

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

See Also

`canHWInfo`

Purpose

Create handle to CAN database file

Syntax

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

Description

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

Examples

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

See Also

`canMessage`

canHWInfo

Purpose

Information on available CAN devices

Syntax

```
out = canHWInfo()
```

Description

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

Examples

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

See Also

`canChannel`

Purpose

Build CAN message based on user-specified structure

Syntax

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

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.
`message`	The message object returned from the function.

Description

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

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

`message = canMessage(database, id, extended)` constructs a message using the message definition of the specified ID and type, in the specified database.

Examples

```
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')
message = canMessage (candb, 800, false)
```

See Also

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

Purpose

Generate technical support log

Syntax

```
canSupport()
```

Description

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

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

canTool

Purpose

Open CAN Tool

Syntax

```
canTool
```

Description

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

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

Purpose

Set bit timing rate of CAN channel

Syntax

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

Arguments

`canch`	The CAN channel object that you want to set the bit timing rate for.
`busspeed`	The user-specified bit timing rate for the specified object.
`sjw`	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.

Description

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

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

Remarks

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

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

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

Examples

```
canch = canChannel('Vector','CANCaseXL 1',1)
configBusSpeed(canch,250000)
canch = canChannel('Vector','CANCaseXL 1',1)
configBusSpeed(canch,500000,1,4,3,1)
```

See Also

`canChannel`

Purpose

Select all instances of message from array of messages

Syntax

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

Arguments

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

Description

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

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

Remarks

You can specify `id` as a cell array of message names or a vector of identifiers. For example, if you pass `id` in as [250 5000], [false true], `extractAll` returns every instance of both CAN message `250` and message `500` that it finds in the `message` array. If any `id` in the vector

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

Examples

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

See Also

`extractRecent`, `extractTime`

Purpose

Select most recent message from array of messages

Syntax

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

Arguments

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

Description

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

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

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

Remarks

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

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

Examples

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

See Also

`extractAll`, `extractTime`

Purpose

Select messages occurring within specified time range from array of messages

Syntax

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

Arguments

`message`	An array of CAN message objects.
`starttime`	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.
`endtime`	The end of the time range in seconds that you specify. Parses messages with timestamp up to the specified end time, including the specified end time.
`extracted`	An array of CAN message objects returned with all messages that occur within and including `starttime` and `endtime`.

Description

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

Remarks

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

Examples

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

See Also

`extractAll`, `extractRecent`

filterAcceptRange

Purpose

Set range of CAN identifiers to pass acceptance filter

Syntax

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

Arguments

`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.

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.

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.
- Set the values from `rangestart` through `rangeend` in increasing order.
- `filterAcceptRange` and `filterBlockRange` work together by allowing and blocking ranges of CAN messages within a single filter. You can perform both operations multiple times in sequence to custom configure the filter as desired.

Remarks

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

Examples

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

See Also

`filterBlockRange`, `filterReset`, `filterSet`

filterBlockRange

Purpose

Set range of CAN identifiers to block via acceptance filter

Syntax

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

Arguments

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

Description

`filterBlockRange(canch, rangestart, rangeend)` allows you to block messages within a given range by setting an acceptance filter.

Notes

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

Examples

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

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

See Also

`filterAcceptRange`, `filterReset`, `filterSet`

filterReset

Purpose

Open CAN message acceptance filters

Syntax

```
filterReset(canch)
```

Description

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

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

Examples

Reset the message filters as shown:

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

See Also

`filterAcceptRange`, `filterBlockRange`, `filterSet`

Purpose

Set specific CAN message acceptance filter configuration

Syntax

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

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

Description

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

Notes

- You can configure message filtering only when the CAN channel is offline.
- CAN message filters initialize to fully open.
- Use `filterReset` to make the acceptance filters fully open.
- `filterSet` supports either standard or extended CAN identifiers.
- To configure filtering for standard CAN identifiers, use either `filterSet` or `filterAcceptRange`/`filterBlockRange` as both choices operate on a single filter.
- To configure filtering for extended CAN identifiers, use only `filterSet`.

Examples

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

See Also

`filterAcceptRange`, `filterBlockRange`, `filterReset`

Purpose

Return property values

Syntax

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

Description

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

Examples

Configure a CAN channel:

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

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

```
get (canch)
```

Configure a CAN message:

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

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

```
get (message)
```

Configure a CAN database:

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

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

```
get (candb)
```

messageInfo

Purpose

Information about CAN messages

Syntax

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

Arguments

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

Description

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

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

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

Examples

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

See Also

`canDatabase`, `canMessage`, `signalInfo`

Purpose

Pack signal data into CAN message

Syntax

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

Arguments

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

Description

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

Examples

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

See Also

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

receive

Purpose

Receive messages from CAN bus

Syntax

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

Arguments

`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`.
`message`	An array of CAN message objects received from the channel.

Description

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

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

Examples

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

To receive all messages, type:

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

See Also

`canChannel`, `canMessage`, `transmit`

Purpose

Receive raw messages from CAN bus

Syntax

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

Arguments

`canch`	The CAN channel from which to receive the message.
`messagerequested`	The maximum count of messages to receive. The specified value must be nonzero and positive, or `Inf`.
`message`	An array of message structures received from the CAN channel.

Description

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

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

Examples

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

```
canch = canChannel('Vector','CANCaseXL 1',1)
start(canch)
message = receiveRaw(canch,5)
message = canMessage(msgStructs)
attachDatabase(message, canDatabase('Database.dbc'))
```

Note This example is not an exact workflow.

To receive all messages in the raw structure, type:

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

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

See Also

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

Purpose

Retransmit messages from CAN bus

Syntax

```
replay(canch, message)
```

Arguments

`canch`	The CAN channel that you specify to transmit the messages.
`message`	An array of message objects to replay.

Description

`replay(canch, message)` retransmits the message or messages `message` on the channel `canch`, based on the relative differences of their timestamps.

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

Remarks

If you have a loopback connection between two channels, you can:

- Transmit messages 2 seconds apart from one channel.
- Receive them on the other channel.
- Use `replay` to retransmit the messages with the original delay.

Examples

The timestamp differentials between messages in the two receive arrays are equal.

```
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)
transmit(ch1, msgTx1)
pause(2)
transmit(ch1, msgTx2)
msgRx1 = receive(ch2, Inf)
replay(canch2, msgRx1)
```

```
pause(2)
msgRx2 = receive(ch1, Inf)
```

See Also

canChannel, canMessage, receive, transmit

Purpose

Configure property values

Syntax

```
set (obj, propertyname, propertyvalue)
```

Description

`set (obj, propertyname, propertyvalue)` configures the specified property, `propertyname`, on the object `obj`, to the value specified in `propertyvalue`.

Examples

To set a CAN channel property:

```
canch = canChannel('Vector', 'CANcaseXL 1', 1)
set (canch, 'SilentMode', true)
```

To set a CAN message property:

```
message = canMessage(250, 8, true)
set (message, 'Remote', true)
```

To set a CAN message signal property:

```
candb = canDatabase('C:\Database.dbc')
message = canMessage(candb, 'Battery_Voltage')
set (message, 'BatVlt', 9.3)
```

signalInfo

Purpose

Information about signals in CAN message

Syntax

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

Arguments

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

Description

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

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

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

Examples

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

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

See Also

canDatabase, canMessage, messageInfo

start

Purpose

Set CAN channel online

Syntax

```
start(canch)
```

Description

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

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

Examples

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

See Also

`stop`

Purpose

Set CAN channel offline

Syntax

```
stop(canch)
```

Description

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

Examples

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

See Also

`start`

transmit

Purpose

Send CAN messages to CAN bus

Syntax

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

Arguments

`canch`	The CAN channel that you specify to transmit the message.
`message`	The message or an array of messages that you specify to transmit via a CAN channel.

Description

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

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

Remarks

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

Examples

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

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

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

To transmit messages on a remote frame, type:

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

See Also

`canChannel`, `canMessage`, `receive`

unpack

Purpose

Unpacks signal data from message

Syntax

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

Arguments

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

Description

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

Examples

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

See Also

`canMessage`, `extractAll`, `extractRecent`, `extractTime`, `pack`

7

Property Reference

CAN Channel Base Properties (p. 7-2)	Apply to CAN channels on all devices
Device-Specific Properties (p. 7-4)	Apply to CAN channels on specific devices

CAN Channel Base Properties

Channel Status Properties

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

CAN Message Properties

`Data`	Set CAN message data
`Database`	Store CAN database information
`Error`	CAN message error frame
`Extended`	Identifier type for CAN message
`ID`	Identifier for CAN message
`Remote`	Specify CAN message remote frame
`Timestamp`	Display message received timestamp

CAN Database Properties

`Messages`	Stores message names from CAN database
`Name (Database)`	CAN database name
`Path`	Display CAN database directory path

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

Vector Device Settings

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

Transceiver Settings

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

Bit Timing Settings

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

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

8

Properties — Alphabetical List

BusSpeed

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`

Purpose

Determine status of CAN bus

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	String

Values

- N/A
- `BusOff`
- `ErrorOff`
- `ErrorActive`

See Also

Functions

`canChannel`

Data

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`

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	String

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 = 'C:\Database.dbc'
```

See Also

Functions

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

Device

Purpose

Display CAN channel device type

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	String

Values

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

See Also

Functions

`canChannel`, `canHWInfo`

Properties

`DeviceChannelIndex`, `DeviceVendor`

Purpose

Display CAN device channel index

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Numeric

Values

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

See Also

Functions

`canChannel`, `canHWInfo`

Properties

`Device`, `DeviceVendor`

DeviceSerialNumber

Purpose

Display CAN device serial number

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Numeric

Values

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

See Also

Functions

`canChannel`, `canHWInfo`

Properties

`Device`, `DeviceVendor`

Purpose

Display device vendor name

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	String

Values

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

See Also

Functions

`canChannel`, `canHWInfo`

Properties

`Device`, `DeviceChannelIndex`, `DeviceSerialNumber`

Error

Purpose

CAN message error frame

Description

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

Characteristics

Usage	CAN message
Read only	Always
Data type	Boolean

Values

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

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

See Also

Functions

`canMessage`

Purpose

Identifier type for CAN message

Description

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

Characteristics

Usage	CAN message
Read only	Always
Data type	Boolean

Values

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

Examples

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

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

You cannot edit this property after the initial configuration.

See Also

Functions

`canMessage`

Properties

`ID`

ID

Purpose

Identifier for CAN message

Description

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

Characteristics

Usage	CAN message
Read only	Always
Data type	Numeric

Values

The ID value must be a positive integer from:

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

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

Examples

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

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

See Also

Functions

`canMessage`

Properties

`Extended`

Purpose

Determine control of device channel

Description

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

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Boolean

Values

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

See Also

Functions

`canChannel`

MessageReceivedFcn

Purpose

Specify function to run

Description

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

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

Characteristics

Usage	CAN channel
Read only	Never
Data type	Callback function

Values

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

Examples

```
canch.MessageReceivedFcn = @Myfunction
```

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

See Also

Functions

`canChannel`, `set`

Properties

`MessageReceivedFcnCount`, `MessagesAvailable`

Purpose

Specify number of messages available before function is triggered

Description

You configure `MessageReceivedFcnCount` to the number of messages that must be available before a `MessageReceivedFcn` is triggered.

Characteristics

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

Values

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

Examples

```
canch.MessageReceivedFcnCount = 55
```

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

See Also

Functions

`canChannel`, `set`

Properties

`MessageReceivedFcn`, `MessagesAvailable`

Messages

Purpose

Stores message names from CAN database

Description

This property stores the names of all the messages defined in the selected CAN database.

Characteristics

Usage	CAN database
Read only	Always
Data type	String

Values

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

See Also

`canDatabase`, `messageInfo`

Purpose

Display number of messages available to be received by CAN channel

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

The value is `0` when no messages are available.

See Also

Functions

`canChannel`

Properties

`MessagesReceived`, `MessagesTransmitted`

MessagesReceived

Purpose

Display number of messages received by CAN channel

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

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

See Also

Functions

`canChannel`, `canHWInfo`

Properties

`MessagesAvailable`, `MessagesTransmitted`

Purpose

Display number of messages transmitted by CAN channel

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

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

See Also

Functions

`canChannel`

Properties

`MessagesAvailable`, `MessagesReceived`

Name (Database)

Purpose

CAN database name

Description

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

Characteristics

Usage	CAN database
Read only	Always
Data type	String

Values

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

See Also

Functions

`canDatabase`

Properties

`Extended`, `ID`

Purpose

CAN message name

Description

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

Characteristics

Usage	CAN message
Read only	Always
Data type	String

Values

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

See Also

Functions

`canMessage`

Properties

`Extended`, `ID`

NumOfSamples

Purpose

Display number of samples available to channel

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

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

See Also

Functions

`canChannel`, `configBusSpeed`

Properties

`BusSpeed`, `SJW`, `TSEG1`, `TSEG2`

Purpose

Display CAN database directory path

Description

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

Characteristics

Usage	CAN database
Read only	Always
Data type	String

Values

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

See Also

Functions

`canDatabase`

ReceiveErrorCount

Purpose

Display number of received errors detected by channel

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

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

See Also

Functions

`canChannel`, `receive`

Properties

`TransmitErrorCount`

Purpose

Specify CAN message remote frame

Description

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

Characteristics

Usage	CAN message
Read only	Never
Data type	Boolean

Values

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

Examples

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

```
message.Remote = true
```

See Also

Functions

`canMessage`

Running

Purpose

Determine status of CAN channel

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Boolean

Values

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

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

See Also

Functions

`canChannel`, `start`

Purpose

Specify if channel is active or silent

Description

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

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

Characteristics

Usage	CAN channel
Read only	Never
Data type	Boolean

Values

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

Examples

To configure the channel to silent mode, type:

```
canch.SilentMode = true
```

To configure the channel to normal mode, type:

```
canch.SilentMode = false
```

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

See Also

Functions

`canChannel`, `set`

SJW

Purpose

Display synchronization jump width (SJW) of bit time segment

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Numeric

Values

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

See Also

Functions

`canChannel`, `configBusSpeed`

Properties

`BusSpeed`, `NumOfSamples`, `TSEG1`, `TSEG2`

Purpose

Display message received timestamp

Description

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

Characteristics

Usage	CAN message
Read only	Never
Data type	Double

Values

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

Examples

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

```
message.Timestamp = 12
```

See Also

Functions

`canChannel`, `canMessage`, `receive`, `replay`

TransceiverName

Purpose

Display name of CAN transceiver

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	String

Values

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

See Also

Functions

`canChannel`

Properties

`TransceiverState`

Purpose

Display state or mode of CAN transceiver

Description

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

Characteristics

Usage	CAN channel
Read only	Never
Data type	Numeric

Values

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

- `high speed`
- `high voltage`
- `sleep`
- `wake up`

See Also

Functions

`canChannel`

Properties

`TransceiverName`

TransmitErrorCount

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.

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

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

See Also

Functions

`canChannel`, `configBusSpeed`

Properties

`BusSpeed`, `NumberOfSamples`, `SJW`, `TSEG2`

TSEG2

Purpose

Display amount that channel can shorten sample time

Description

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

Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

Values

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

See Also

Functions

`canChannel`, `configBusSpeed`

Properties

`BusSpeed`, `NumberOfSamples`, `SJW`, `TSEG1`

Index

E

F

I

M

N

P

R

S

T

U

V