

# Vehicle Network Toolbox™

## User's Guide



MATLAB® & SIMULINK®

R2019a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Vehicle Network Toolbox™ User's Guide*

© COPYRIGHT 2009–2019 by The MathWorks, Inc.

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

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

## Trademarks

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

## Patents

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

## Revision History

March 2009	Online only	New for Version 1.0 (Release 2009a)
September 2009	Online only	Revised for Version 1.1 (Release 2009b)
March 2010	Online only	Revised for Version 1.2 (Release 2010a)
September 2010	Online only	Revised for Version 1.3 (Release 2010b)
April 2011	Online only	Revised for Version 1.4 (Release 2011a)
September 2011	Online only	Revised for Version 1.5 (Release 2011b)
March 2012	Online only	Revised for Version 1.6 (Release 2012a)
September 2012	Online only	Revised for Version 1.7 (Release 2012b)
March 2013	Online only	Revised for Version 2.0 (Release 2013a)
September 2013	Online only	Revised for Version 2.1 (Release 2013b)
March 2014	Online only	Revised for Version 2.2 (Release 2014a)
October 2014	Online only	Revised for Version 2.3 (Release 2014b)
March 2015	Online only	Revised for Version 2.4 (Release 2015a)
September 2015	Online only	Revised for Version 3.0 (Release 2015b)
March 2016	Online only	Revised for Version 3.1 (Release 2016a)
September 2016	Online only	Revised for Version 3.2 (Release 2016b)
March 2017	Online only	Revised for Version 3.3 (Release 2017a)
September 2017	Online only	Revised for Version 3.4 (Release 2017b)
March 2018	Online only	Revised for Version 4.0 (Release 2018a)
September 2018	Online only	Revised for Version 4.1 (Release 2018b)
March 2019	Online only	Revised for Version 4.2 (Release 2019a)



## 1

## Getting Started

<b>Vehicle Network Toolbox Product Description</b> .....	<b>1-2</b>
Key Features .....	<b>1-2</b>
<b>Toolbox Characteristics and Capabilities</b> .....	<b>1-3</b>
Vehicle Network Toolbox Characteristics .....	<b>1-3</b>
Interaction Between the Toolbox and Its Components .....	<b>1-4</b>
Prerequisite Knowledge .....	<b>1-6</b>
<b>Vehicle Network Communication in MATLAB</b> .....	<b>1-7</b>
Transmit Workflow .....	<b>1-7</b>
Receive Workflow .....	<b>1-8</b>
<b>Transmit and Receive CAN Messages</b> .....	<b>1-9</b>
Discover Installed Hardware .....	<b>1-9</b>
Create CAN Channels .....	<b>1-10</b>
Configure Channel Properties .....	<b>1-13</b>
Start the Channels .....	<b>1-14</b>
Create a Message .....	<b>1-15</b>
Pack a Message .....	<b>1-16</b>
Transmit a Message .....	<b>1-17</b>
Receive a Message .....	<b>1-18</b>
Unpack a Message .....	<b>1-21</b>
Save and Load CAN Channels .....	<b>1-21</b>
Disconnect Channels and Clean Up .....	<b>1-22</b>
<b>Filter Messages</b> .....	<b>1-24</b>
<b>Multiplex Signals</b> .....	<b>1-25</b>
<b>Configure Silent Mode</b> .....	<b>1-28</b>

## Hardware Support Package Installation

### 2

<b>Vehicle Network Toolbox Supported Hardware</b> .....	2-2
<b>Install Hardware Support Package for Device Driver</b> .....	2-3
Install Support Packages .....	2-3
Update or Uninstall Support Packages .....	2-3

## CAN Communication Workflows

### 3

<b>CAN Transmit Workflow</b> .....	3-2
<b>CAN Receive Workflow</b> .....	3-4

## Using a CAN Database

### 4

<b>Load .dbc Files and Create Messages</b> .....	4-2
Vector CAN Database Support .....	4-2
Load the CAN Database .....	4-2
Create a CAN Message .....	4-3
Access Signals in the Constructed CAN Message .....	4-3
Add a Database to a CAN Channel .....	4-4
Update Database Information .....	4-4
Create and Process Messages Using Database Definitions ...	4-4
<b>View Message Information in a CAN Database</b> .....	4-16
<b>View Signal Information in a CAN Message</b> .....	4-18
<b>Attach a CAN Database to Existing Messages</b> .....	4-19

## Monitoring Vehicle CAN Bus

### 5

<b>Vehicle CAN Bus Monitor</b> .....	5-2
About the Vehicle CAN Bus Monitor .....	5-2
Opening the Vehicle CAN Bus Monitor .....	5-2
Vehicle CAN Bus Monitor Fields .....	5-2
<b>Using the Vehicle CAN Bus Monitor</b> .....	5-9
View Messages on a Channel .....	5-9
Configure the Channel Bus Speed .....	5-9
Filter CAN Messages in Vehicle CAN Bus Monitor .....	5-10
Attach a Database .....	5-10
Change the Message Count .....	5-13
Change the Number Format .....	5-13
View Unique Messages .....	5-13
Save the Message Log File .....	5-14

## XCP Communication Workflows

### 6

<b>XCP Database and Communication Workflow</b> .....	6-2
--	-----

## A2L File

### 7

<b>Inspect the Contents of an A2L File</b> .....	7-2
Access an A2L File .....	7-2
Access Measurement Information .....	7-2
Access Event Information .....	7-4

## **Universal Measurement & Calibration Protocol (XCP)**

### **8**

<b>XCP Hardware Connection</b> .....	<b>8-2</b>
Create XCP Channel Using CAN Device .....	<b>8-4</b>
Configure the Channel to Unlock the Slave .....	<b>8-4</b>
<b>Read a Single Value</b> .....	<b>8-6</b>
<b>Write a Single Value</b> .....	<b>8-7</b>
<b>Read a Calibrated Measurement</b> .....	<b>8-8</b>
<b>Acquire Measurement Data via Dynamic DAQ Lists</b> .....	<b>8-9</b>
<b>Stimulate Measurement Data via Dynamic STIM Lists</b> .....	<b>8-10</b>

### **J1939**

### **9**

<b>J1939 Interface</b> .....	<b>9-2</b>
<b>J1939 Parameter Group Format</b> .....	<b>9-3</b>
<b>J1939 Network Management</b> .....	<b>9-5</b>
Address Claiming .....	<b>9-5</b>
<b>J1939 Transport Protocols</b> .....	<b>9-6</b>
<b>J1939 Channel Workflow</b> .....	<b>9-7</b>

## **CAN Communications in Simulink**

### **10**

<b>Vehicle Network Toolbox Simulink Blocks</b> .....	<b>10-2</b>
--	-------------



<b>CAN Communication Workflows in Simulink</b> .....	<b>10-3</b>
Message Transmission Workflow .....	<b>10-3</b>
Message Reception Workflow .....	<b>10-5</b>
<b>Open the Vehicle Network Toolbox Block Library</b> .....	<b>10-7</b>
Using the Simulink Library Browser .....	<b>10-7</b>
Using the MATLAB Command Window .....	<b>10-8</b>
<b>Build CAN Communication Simulink Models</b> .....	<b>10-9</b>
Build a Message Transmit Model .....	<b>10-9</b>
Build a Message Receive Model .....	<b>10-14</b>
Save and Run the Model .....	<b>10-23</b>
<b>Create Custom CAN Blocks</b> .....	<b>10-28</b>
Blocks Using Simulink Buses .....	<b>10-28</b>
Blocks Using CAN Message Data Types .....	<b>10-30</b>

## Hardware Limitations

# 11

<b>Vector Hardware Limitations</b> .....	<b>11-2</b>
<b>Kvaser Hardware Limitations</b> .....	<b>11-3</b>
<b>File Format Limitations</b> .....	<b>11-4</b>
CDFX-File .....	<b>11-4</b>
BLF-File .....	<b>11-4</b>

## XCP Communications in Simulink

# 12

<b>Vehicle Network Toolbox XCP Simulink Blocks</b> .....	<b>12-2</b>
<b>Open the Vehicle Network Toolbox XCP Block Library</b> .....	<b>12-3</b>
Using the MATLAB Command Window .....	<b>12-3</b>
Using the Simulink Library Browser .....	<b>12-4</b>

<b>13</b>	<b>Functions – Alphabetical List</b>
<b>14</b>	<b>Properties – Alphabetical List</b>
<b>15</b>	<b>Block Reference</b>

# Getting Started

---

- “Vehicle Network Toolbox Product Description” on page 1-2
- “Toolbox Characteristics and Capabilities” on page 1-3
- “Vehicle Network Communication in MATLAB” on page 1-7
- “Transmit and Receive CAN Messages” on page 1-9
- “Filter Messages” on page 1-24
- “Multiplex Signals” on page 1-25
- “Configure Silent Mode” on page 1-28

## Vehicle Network Toolbox Product Description

### Communicate with in-vehicle networks using CAN, J1939, and XCP protocols

Vehicle Network Toolbox provides MATLAB® functions and Simulink® blocks to send, receive, encode, and decode CAN, CAN FD, J1939, and XCP messages. The toolbox lets you identify and parse specific signals using industry-standard CAN database files and then visualize the decoded signals using the CAN Bus Monitor app. Using A2L description files, you can connect to an ECU via XCP on CAN or Ethernet. You can access messages and measurement data stored in MDF files.

The toolbox simplifies communication with in-vehicle networks and lets you monitor, filter, and analyze live CAN bus data, or log and record messages for later analysis and replay. You can simulate message traffic on a virtual CAN bus or connect to a live network or ECU. Vehicle Network Toolbox supports CAN interface devices from Vector, Kvaser, PEAK-System, and National Instruments®.

### Key Features

- MATLAB functions for transmitting and receiving CAN, CAN FD, J1939, and XCP messages
- Simulink blocks for communicating over CAN, CAN FD, J1939 or XCP protocols
- XCP support for interacting with ECUs over Ethernet or CAN
- Vector CAN database (.dbc) file, A2L description file, and MDF file support
- Vehicle CAN Bus Monitor app to configure devices and visualize live CAN network traffic
- Signal packing and unpacking for simplified encoding and decoding of CAN messages, CAN FD messages, and J1939 parameter groups
- Support for Vector, Kvaser, PEAK-System, and National Instruments CAN interface devices and for virtual channels

# Toolbox Characteristics and Capabilities

**In this section...**

“Vehicle Network Toolbox Characteristics” on page 1-3

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

“Prerequisite Knowledge” on page 1-6

## Vehicle Network Toolbox Characteristics

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

You can use the toolbox to:

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

### Connect to CAN Devices

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

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

### Use Supported CAN Devices and Drivers

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

See “Vehicle Network Toolbox Supported Hardware” on page 2-2 for more information.

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

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

## **Simulate CAN Communication**

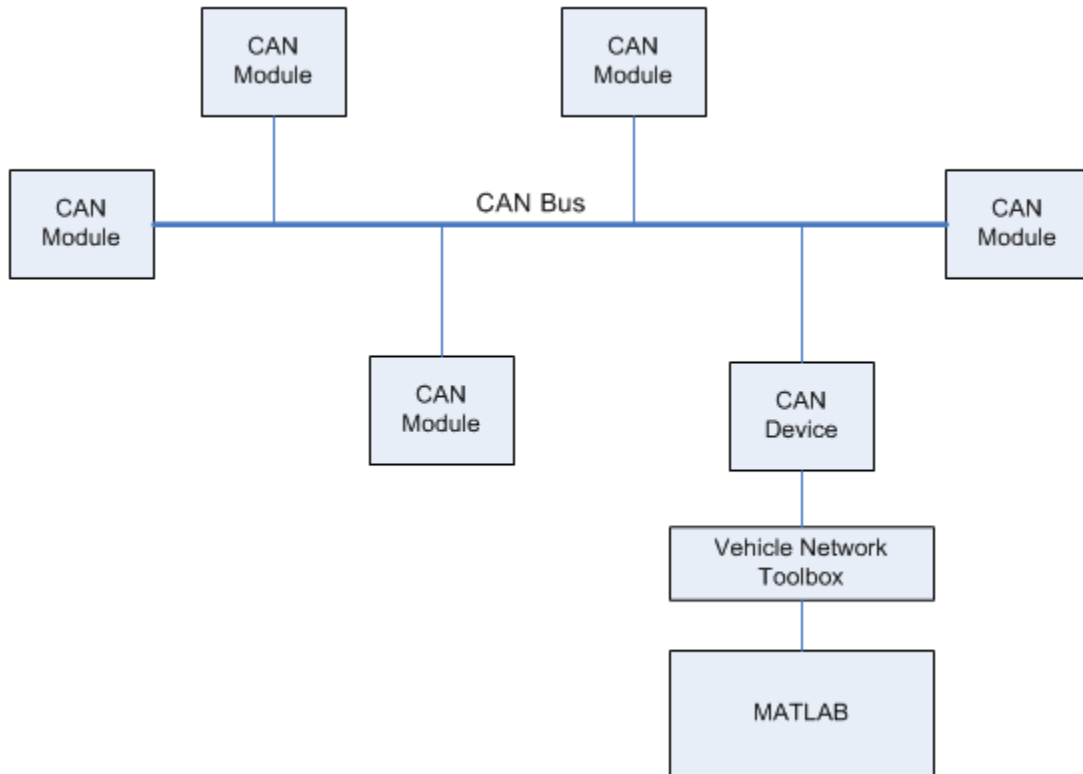
With Vehicle Network Toolbox block library and other blocks from the Simulink library, you can create sophisticated models to connect to a live network and to simulate message traffic on a CAN bus.

## **Visualize CAN Communication**

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

## **Interaction Between the Toolbox and Its Components**

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



In this illustration:

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

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

- Transmit messages to the CAN bus.
- Receive messages from the CAN bus.
- Trigger a callback function to run when the channel receives a message.

- Attach the database to the configured CAN channel to interpret received CAN messages.
- Use the CAN database to construct messages to transmit.
- Log and record messages and analyze them in MATLAB.
- Replay live recorded sequence of messages in MATLAB.
- Build Simulink models to connect to a CAN bus and to simulate message traffic.
- Monitor CAN traffic with the “Vehicle CAN Bus Monitor” on page 5-2.

Vehicle Network Toolbox is a comprehensive solution for CAN connectivity in MATLAB and Simulink. Refer to the Functions and Simulink Blocks for more information.

## Prerequisite Knowledge

The Vehicle Network Toolbox document set assumes that you are familiar with these products:

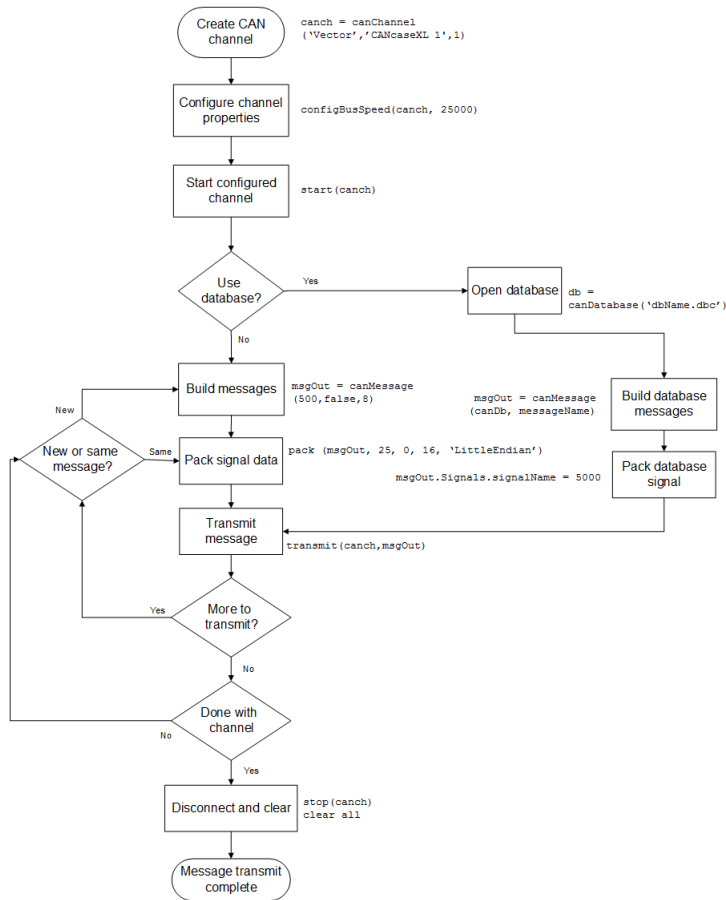
- MATLAB — To write scripts and functions, and to use functions with the command-line interface.
- Simulink — To create simple models to connect to a CAN bus or to simulate those models.
- Vector CANdb — To understand CAN databases, along with message and signal definitions.



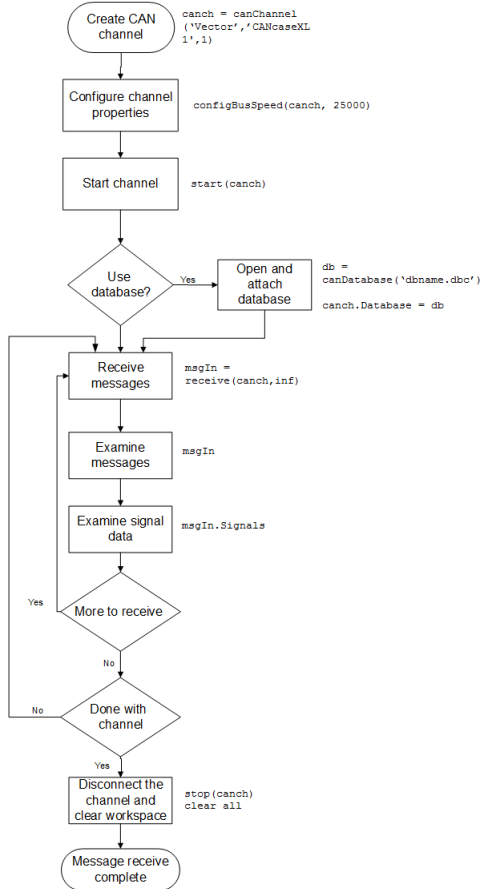
# Vehicle Network Communication in MATLAB

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

## Transmit Workflow



## Receive Workflow



## Transmit and Receive CAN Messages

### In this section...

“Discover Installed Hardware” on page 1-9  
 “Create CAN Channels” on page 1-10  
 “Configure Channel Properties” on page 1-13  
 “Start the Channels” on page 1-14  
 “Create a Message” on page 1-15  
 “Pack a Message” on page 1-16  
 “Transmit a Message” on page 1-17  
 “Receive a Message” on page 1-18  
 “Unpack a Message” on page 1-21  
 “Save and Load CAN Channels” on page 1-21  
 “Disconnect Channels and Clean Up” on page 1-22

### Discover Installed Hardware

In the example, you discover your system CAN devices with `canHWInfo`, then create two CAN channels using `canChannel`. Later, you edit the properties of the first channel and create a message using `canMessage`, then transmit the message from the first channel using `transmit`, and receive it on the other channel using `receive`.

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

```
info = canHWInfo
```

```
info =
```

```
CAN Devices Detected
```

Vendor	Device	Channel	Serial Number	Constructor
MathWorks	Virtual 1	1	0	canChannel('MathWorks','Virtual 1',1)
MathWorks	Virtual 1	2	0	canChannel('MathWorks','Virtual 1',2)
Vector	VN1610 1	1	18959	canChannel('Vector','VN1610 1',1)
Vector	VN1610 1	2	18959	canChannel('Vector','VN1610 1',2)
Vector	Virtual 1	1	0	canChannel('Vector','Virtual 1',1)
Vector	Virtual 1	2	0	canChannel('Vector','Virtual 1',2)
PEAK-System	PCAN-USB Pro	1	0	canChannel('PEAK-System','PCAN_USBBUS1')
PEAK-System	PCAN-USB Pro	2	0	canChannel('PEAK-System','PCAN_USBBUS2')
Kvaser	USBcan Professional 1	1	10680	canChannel('Kvaser','USBcan Professional 1')
Kvaser	USBcan Professional 1	2	10680	canChannel('Kvaser','USBcan Professional 1')

```
Kvaser | Virtual 1 | 1 | 0 | canChannel('Kvaser','Virtual 1',1)
Kvaser | Virtual 1 | 2 | 0 | canChannel('Kvaser','Virtual 1',2)
NI | 9862 CAN/HS (CAN1) | 1 | 17F5094 | canChannel('NI','CAN1')
NI | 9862 CAN/HS (CAN2) | 1 | 17F50B2 | canChannel('NI','CAN2')
```

- 2 Save the MathWorks virtual device information to a variable. The indexing indicates vendors; so from this `canHWInfo` output, `info.VendorInfo(1)` corresponds to MathWorks, `info.VendorInfo(2)` corresponds to Vector, etc.

```
mwvirt = info.VendorInfo(1)
```

```
mwvirt =
```

```
VendorInfo with properties:
```

```
VendorName: 'MathWorks'
VendorDriverDescription: 'MathWorks Virtual CAN Driver'
VendorDriverVersion: '1'
ChannelInfo: [1x2 can.mathworks.ChannelInfo]
```

- 3 Get details about the first available virtual CAN channel on the device.

```
mwvirt.ChannelInfo(1)
```

```
ans =
```

```
ChannelInfo with properties:
```

```
Device: 'Virtual 1'
DeviceChannelIndex: 1
DeviceSerialNumber: 0
ObjectConstructor: 'canChannel('MathWorks','Virtual 1',1)'
```

---

**Note** To modify this example for a hardware CAN device, make a loopback connection between the two channels.

---

## Create CAN Channels

Create two MathWorks virtual CAN channels.

```
canch1 = canChannel('MathWorks','Virtual 1',1)
canch2 = canChannel('MathWorks','Virtual 1',2)
```

```
canch1 =
```

```
Channel with properties:
```

```
Device Information
```

```
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'
```

```
Status Information
```

```
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

```
Channel Information
```

```
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 500000
    SJW: []
    TSEG1: []
    TSEG2: []
    NumOfSamples: []
```

```
Other Information
```

```
    Database: []
    UserData: []
```

```
canch2 =
```

```
Channel with properties:
```

```
Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 2
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'

Status Information
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'

Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 500000
    SJW: []
    TSEG1: []
    TSEG2: []
    NumOfSamples: []

Other Information
    Database: []
    UserData: []
```

For each channel, notice that its initial `Running` value is 0 (stopped), and its bus speed is 500000.

---

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

You cannot create arrays of CAN channel objects. Each object you create must be assigned to its own scalar variable.

---

## Configure Channel 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 Change the bus speed of both channels to 250000, then view the channel `BusSpeed` property to verify the setting.

```
configBusSpeed(canch1,250000)
canch1.BusSpeed
```

```
ans =
    250000
```

- 2 You can also see the updated bus speed in the channel display.

```
canch1
```

```
canch1 =
```

```
Channel with properties:
```

```
Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'

Status Information
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
```

```
FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

## Channel Information

```
    BusStatus: 'N/A'  
    SilentMode: 0  
    TransceiverName: 'N/A'  
    TransceiverState: 'N/A'  
    ReceiveErrorCount: 0  
    TransmitErrorCount: 0  
    BusSpeed: 250000  
    SJW: []  
    TSEG1: []  
    TSEG2: []  
    NumOfSamples: []
```

## Other Information

```
    Database: []  
    UserData: []
```

- 3 In a similar way, change the bus speed of the second channel.

```
configBusSpeed(canch2,250000)
```

## Start the Channels

After you configure their properties, start both channels. Then view the updated status setting of the first channel.

```
start(canch1)  
start(canch2)  
canch1
```

```
canch1 =
```

```
Channel with properties:
```

## Device Information

```
    DeviceVendor: 'MathWorks'  
    Device: 'Virtual 1'  
    DeviceChannelIndex: 1  
    DeviceSerialNumber: 0  
    ProtocolMode: 'CAN'
```



```
Status Information
    Running: 1
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: 12-Jul-2017 17:04:48
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'

Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 250000
    SJW: []
    TSEG1: []
    TSEG2: []
    NumOfSamples: []

Other Information
    Database: []
    UserData: []
```

Notice that the channel status is now indicated as **Online**.

## Create a Message

After you set all the property values as desired and your channels are online, you are ready to transmit and receive messages on the CAN bus. For this exercise, transmit a message using `canch1` and receive it using `canch2`. To transmit a message, create a message object and pack the message with the required data.

Build a CAN message with a standard type ID of 500, and a data length of 8 bytes.

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

```
messageout =
```

```
    Message with properties:
```

```
Message Identification
  ProtocolMode: 'CAN'
    ID: 500
  Extended: 0
  Name: ''

Data Details
  Timestamp: 0
  Data: [0 0 0 0 0 0 0 0]
  Signals: []
  Length: 8

Protocol Flags
  Error: 0
  Remote: 0

Other Information
  Database: []
  UserData: []
```

Some of the properties of the message indicate:

- **Error** — A logical 0 (false) because the message is not an error.
- **Remote** — A logical 0 (false) because the message is not a remote frame.
- **ID** — The ID you specified.
- **Extended** — A logical 0 (false) because you did not specify an extended ID.
- **Data** — A uint8 array of 0s, with size specified by the data length.

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

## Pack a Message

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

- 1 Use the `pack` function to pack your message with these input parameters: a Data value of 25, start bit of 0, signal size of 16, and byte order using little-endian format. View the message `Data` property to verify the settings.

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

```
ans =  
  
1x8 uint8 row vector  
  
25    0    0    0    0    0    0    0
```

The only message property that changes from packing is **Data**. Refer to the `pack` function to understand more about its input arguments.

## Transmit a Message

Now you can transmit the packed message. Use the `transmit` function, supplying the channel `canch1` and the message as input arguments.

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

```
canch1 =
```

```
Channel with properties:
```

```
Device Information
```

```
    DeviceVendor: 'MathWorks'
```

```
        Device: 'Virtual 1'
```

```
    DeviceChannelIndex: 1
```

```
    DeviceSerialNumber: 0
```

```
    ProtocolMode: 'CAN'
```

```
Status Information
```

```
    Running: 1
```

```
    MessagesAvailable: 1
```

```
    MessagesReceived: 0
```

```
    MessagesTransmitted: 1
```

```
    InitializationAccess: 1
```

```
    InitialTimestamp: 12-Jul-2017 17:04:48
```

```
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

```
Channel Information
```

```
    BusStatus: 'N/A'
```

```
    SilentMode: 0
```

```
    TransceiverName: 'N/A'
```

```
TransceiverState: 'N/A'  
ReceiveErrorCount: 0  
TransmitErrorCount: 0  
    BusSpeed: 250000  
        SJW: []  
        TSEG1: []  
        TSEG2: []  
    NumOfSamples: []
```

```
Other Information  
    Database: []  
    UserData: []
```

MATLAB displays the updated channel. In the **Status** section, the **MessagesTransmitted** value increments by 1 each time you transmit a message. The message to be received is available to all devices on the bus, so it shows up here even for the transmitting channel.

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

## Receive a Message

Use the `receive` function to receive the available message on `canch2`.

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

```
canch2
```

```
canch2 =
```

```
Channel with properties:
```

```
Device Information
```

```
    DeviceVendor: 'MathWorks'  
        Device: 'Virtual 1'  
    DeviceChannelIndex: 2  
    DeviceSerialNumber: 0  
    ProtocolMode: 'CAN'
```

```
Status Information
```

```
    Running: 1  
    MessagesAvailable: 1
```

```

    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: 12-Jul-2017 17:04:48
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'

Channel Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 250000
    SJW: []
    TSEG1: []
    TSEG2: []
    NumOfSamples: []

Other Information
    Database: []
    UserData: []

```

The channel status indicates 1 MessagesAvailable.

- 2 Receive one message on canch2 and assign it to messagein.

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

```
messagein =
```

```
Message with properties:
```

```
Message Identification
```

```

    ProtocolMode: 'CAN'
    ID: 500
    Extended: 0
    Name: ''

```

```
Data Details
```

```

    Timestamp: 0.1101
    Data: [25 0 0 0 0 0 0 0]
    Signals: []

```

```
        Length: 8

Protocol Flags
    Error: 0
    Remote: 0

Other Information
    Database: []
    UserData: []
```

Note the message `Data` property. This matches the data transmitted from `canch1`.

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

- 3 To check if the channel received the message, view the channel display.

```
canch2
```

```
canch2 =
```

```
Channel with properties:
```

```
Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 2
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN'
```

```
Status Information
```

```
    Running: 1
    MessagesAvailable: 0
    MessagesReceived: 1
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: 12-Jul-2017 17:04:48
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

```
Channel Information
```

```
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
```

```

ReceiveErrorCount: 0
TransmitErrorCount: 0
    BusSpeed: 250000
        SJW: []
        TSEG1: []
        TSEG2: []
    NumOfSamples: []

Other Information
    Database: []
    UserData: []

```

The channel display indicates 1 MessagesReceived and 0 MessagesAvailable.

## Unpack a Message

After your channel receives a message, specify how to unpack the message and interpret the data in the message. Use `unpack` to specify the parameters for unpacking a message; these should correspond to the parameters used for packing.

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

```
value =
    int16
    25
```

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

## Save and Load CAN Channels

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

To save `canch1` to the MATLAB file `mycanch.mat`, type:

```
save mycanch.mat canch1
```

If you have saved a CAN channel in a MATLAB file, you can load the channel into MATLAB using the `load` function. For example, to reload the channel from `mycanch.mat` that was created earlier, type:

```
load mycanch.mat
```

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

## Disconnect Channels and Clean Up

- “Disconnect the Configured Channels” on page 1-22
- “Clean Up the MATLAB Workspace” on page 1-23

### Disconnect the Configured Channels

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

- 1 Stop the first channel.

```
stop(canch1)
```

- 2 Check the channel status.

```
canch1
.
.
.
Status Information
      Running: 0
MessagesAvailable: 1
MessagesReceived: 0
MessagesTransmitted: 1
```

- 3 Stop the second channel.

```
stop(canch2)
```

- 4 Check the channel status.

```
canch2
```

```
.
```



```
Status Information
      Running: 0
MessagesAvailable: 0
MessagesReceived: 1
MessagesTransmitted: 0
```

## Clean Up the MATLAB Workspace

When you no longer need these objects and variables, remove them from the MATLAB workspace with the `clear` command.

- 1 Clear each channel.

```
clear canch1
clear canch2
```

- 2 Clear the CAN messages.

```
clear messageout
clear messagein
```

- 3 Clear the unpacked value.

```
clear value
```

## See Also

### Related Examples

- “Filter Messages” on page 1-24
- “Multiplex Signals” on page 1-25
- “Configure Silent Mode” on page 1-28

## Filter Messages

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:

- `filterAllowAll`
- `filterBlockAll`
- `filterAllowOnly`

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

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

Set a filter on the channel to allow only the message `EngineMsg`, and display the channel `FilterHistory` property.

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

```
Standard ID Filter: Allow Only | Extended ID Filter: Allow All
```

When you start the channel and receive messages, only those marked `EngineMsg` pass through the filter.

For more information about using a message database, see “Message Database”.

## See Also

### Related Examples

- “Transmit and Receive CAN Messages” on page 1-9

## Multiplex Signals

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

- **Standard signal** — This signal is always active. You can create one or more standard signals.
- **Multiplexor signal** — Also called the mode signal, it is always active and its value determines which multiplexed signal is currently active in the message data. You can create only one multiplexor signal per message.
- **Multiplexed signal** — This signal is active when its multiplex value matches the value of the multiplexor signal. You can create one or more multiplexed signals in a message.

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

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

For example,

- 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')
```

```
m =
```

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

```
Properties:
  ID: 250
  Extended: 0
  Name: 'Msg'
  Database: [1x1 can.Database]
  Error: 0
  Remote: 0
  Timestamp: 0
  Data: [0 0 0 0 0 0 0 0]
  Signals: [1x1 struct]
```

Methods, Events, Superclasses

- 3 To display the signals, type:

```
m.Signals
```

```
ans =
```

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

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

- 4 If you want to make **SigB** active, change the value of the **MuxSig** to 1.

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

To display the signals, type:

```
m.Signals
```

```
ans =
```

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

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

- 5 Change the value of MuxSig to 2.

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

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

```
m.Signals
```

```
ans =
```

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

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

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

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

---

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

---

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

## See Also

### Related Examples

- “Transmit and Receive CAN Messages” on page 1-9

## Configure Silent Mode

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

- 1 Change the `SilentMode` property of the first CAN channel, `canch1` to `true`.

```
canch.SilentMode = true
```

- 2 To see the changed property value, type:

```
canch1.SilentMode
```

```
ans =
```

```
    1
```

## See Also

### Related Examples

- “Transmit and Receive CAN Messages” on page 1-9

# Hardware Support Package Installation

---

- “Vehicle Network Toolbox Supported Hardware” on page 2-2
- “Install Hardware Support Package for Device Driver” on page 2-3

# Vehicle Network Toolbox Supported Hardware



As of this release, Vehicle Network Toolbox supports the following hardware.

Support Package	Vendor	Earliest Release Available	Last Release Available
Kvaser CAN Devices	Kvaser	R2014a	Current
National Instruments NI-CAN Devices	National Instruments	R2014a	R2015b
National Instruments NI-XNET CAN Devices	National Instruments	R2014a	Current
PEAK-System CAN Devices	PEAK-System	R2014a	Current
Vector CAN Devices	Vector	R2014a	Current

For a complete list of supported hardware, see Hardware Support.

For instructions on installing the drivers, see “Install Hardware Support Package for Device Driver” on page 2-3.

## See Also

### More About

- “Vendor Limitations”



## Install Hardware Support Package for Device Driver

In this section...
“Install Support Packages” on page 2-3
“Update or Uninstall Support Packages” on page 2-3

To communicate with a CAN device, you must install the required driver on your system.

The drivers are available in the support packages for the following vendors:

- National Instruments (NI-XNET CAN)
- Kvaser
- Vector
- PEAK-System

---

**Note** For deployed applications, the target machine also needs the appropriate drivers installed. If the target machine does not have MATLAB on it, you must install the vendor drivers manually.

---

### Install Support Packages

To install the support package for the required driver:

- 1 On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
- 2 In the left pane of the Add-On Explorer, scroll to **Filter by Type** and check **Hardware Support Packages**.
- 3 Under **Filter by Hardware Type** check **CAN Devices**. The Add-On Explorer displays all the support packages for the supported vendors of CAN devices. Click the support package for your device vendor.
- 4 Click **Install > Install**. Sign in to your MathWorks® account if necessary, and proceed.

### Update or Uninstall Support Packages

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

## See Also

### More About

- “Get Add-Ons” (MATLAB)
- “Vendor Limitations”

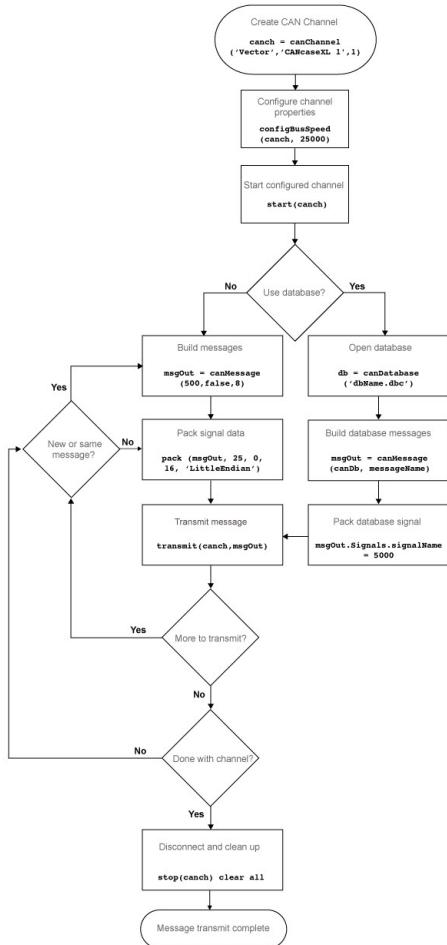
# CAN Communication Workflows

---

- “CAN Transmit Workflow” on page 3-2
- “CAN Receive Workflow” on page 3-4

## CAN Transmit Workflow

This workflow helps you create a CAN channel and transmit messages.



## See Also

### Functions

canChannel | canDatabase | canMessage | canMessageImport | configBusSpeed | pack | start | stop | transmit | transmitConfiguration | transmitEvent | transmitPeriodic

### Properties

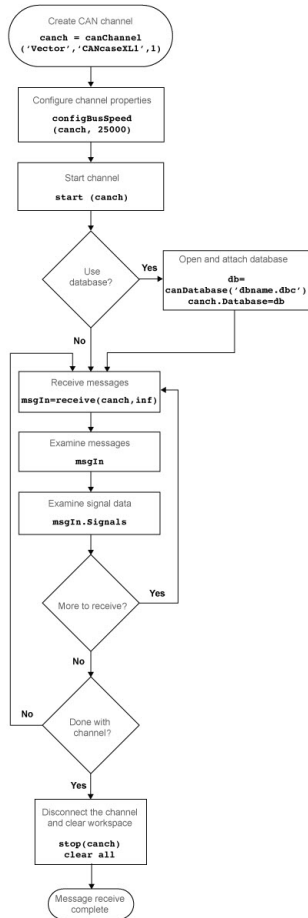
Data | Database | Error | Extended | ID | Name | Remote | Signals | Timestamp | UserData

### Blocks

CAN Pack | CAN Replay | CAN Transmit

## CAN Receive Workflow

Use this workflow to receive and unpack CAN messages.



## See Also

### Functions

attachDatabase | canDatabase | configBusspeed | extractAll | extractRecent  
| extractTime | receive | stop | unpack

### Properties

MessageReceivedFcn | MessageReceivedFcnCount | MessagesAvailable |  
MessagesReceived | MessagesTransmitted | ReceiveErrorCount | TransmitErrorCount

### Blocks

CAN Log | CAN Receive | CAN Unpack





# Using a CAN Database

---

- “Load .dbc Files and Create Messages” on page 4-2
- “View Message Information in a CAN Database” on page 4-16
- “View Signal Information in a CAN Message” on page 4-18
- “Attach a CAN Database to Existing Messages” on page 4-19

# Load .dbc Files and Create Messages

### In this section...

“Vector CAN Database Support” on page 4-2

“Load the CAN Database” on page 4-2

“Create a CAN Message” on page 4-3

“Access Signals in the Constructed CAN Message” on page 4-3

“Add a Database to a CAN Channel” on page 4-4

“Update Database Information” on page 4-4

“Create and Process Messages Using Database Definitions” on page 4-4

## Vector CAN Database Support

Vehicle Network Toolbox allows you to use a Vector CAN database. The database `.dbc` file contains definitions of CAN messages and signals. Using the information defined in the database file, you can look up message and signal information, and build messages. You can also represent message and signal information in engineering units so that you do not need to manipulate raw data bytes.

## Load the CAN Database

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

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

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

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

---

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

---

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

## Create a CAN Message

This example shows you how to create a message using a database constructed specifically for this example. You can access this database in the **Toolbox > VNT > VNTDemos** subfolder in your MATLAB installation folder. This database has a message, `EngineMsg`. To try this example, create messages and signals using definitions in your own database.

- 1 Create the CAN database object.

```
cd ([matlabroot '\examples\vnt'])
d = canDatabase('demoVNT_CANdbFiles.dbc');
```

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

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

## Access Signals in the Constructed CAN Message

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

- 1 To display signals in your message, type:

```
sig = message.Signals
sig =
    struct with fields:
        VehicleSpeed: 0
        EngineRPM: 250
```

- 2 Change the value of the `EngineRPM` signal:

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

- 3 Reassign the signals and display them again to see the change.

```
sig = message.Signals
```

```
sig =  
    struct with fields:  
        VehicleSpeed: 0  
        EngineRPM: 300
```

### Add a Database to a CAN Channel

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

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

For more information, see the Database property.

### Update Database Information

When you make changes to a database file:

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

### Create and Process Messages Using Database Definitions

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

#### Open the Database File

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

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

```
db =
```

```
Database with properties:
```

```
    Name: 'demoVNT_CANdbFiles'
```

```

        Path: 'C:\TEMP\Bdoc19a_1067994_6688\ib99EA80\9\tpc40fffbfb\ex80654288\demo
        Nodes: {}
        NodeInfo: [0x0 struct]
        Messages: {5x1 cell}
        MessageInfo: [5x1 struct]
        Attributes: {}
        AttributeInfo: [0x0 struct]
        UserData: []

```

```
ans =
```

```
5x1 cell array
```

```

{'DoorControlMsg'   }
{'EngineMsg'        }
{'SunroofControlMsg'}
{'TransmissionMsg' }
{'WindowControlMsg' }

```

### View Message Information

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

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

```
ans =
```

```
struct with fields:
```

```

        Name: 'EngineMsg'
    ProtocolMode: 'CAN'
        Comment: ''
            ID: 100
    Extended: 0
        J1939: []
        Length: 8
            DLC: 8
            BRS: 0
        Signals: {2x1 cell}
    SignalInfo: [2x1 struct]
        TxNodes: {0x1 cell}

```

```
Attributes: {}  
AttributeInfo: [0x0 struct]
```

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

```
messageInfo(db)
```

```
ans =
```

```
5x1 struct array with fields:
```

```
Name  
ProtocolMode  
Comment  
ID  
Extended  
J1939  
Length  
DLC  
BRS  
Signals  
SignalInfo  
TxNodes  
Attributes  
AttributeInfo
```

### View Signal Information

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

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

```
ans =
```

```
struct with fields:
```

```
Name: 'EngineRPM'  
Comment: ''  
StartBit: 0  
SignalSize: 32
```

```
ByteOrder: 'LittleEndian'  
Signed: 0  
ValueType: 'Integer'  
Class: 'uint32'  
Factor: 0.1000  
Offset: 250  
Minimum: 250  
Maximum: 9500  
Units: 'rpm'  
ValueTable: [0x1 struct]  
Multiplexor: 0  
Multiplexed: 0  
MultiplexMode: 0  
RxNodes: {0x1 cell}  
Attributes: {}  
AttributeInfo: [0x0 struct]
```

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

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

```
ans =
```

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

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

```
Attributes
AttributeInfo
```

### Create a Message Using Database Definitions

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

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

```
msgEngineInfo =
```

```
Message with properties:
```

```
Message Identification
```

```
ProtocolMode: 'CAN'
             ID: 100
Extended: 0
Name: 'EngineMsg'
```

```
Data Details
```

```
Timestamp: 0
Data: [0 0 0 0 0 0 0 0]
Signals: [1x1 struct]
Length: 8
```

```
Protocol Flags
```

```
Error: 0
Remote: 0
```

```
Other Information
```

```
Database: [1x1 can.Database]
UserData: []
```

### View Signal Information

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

```
msgEngineInfo.Signals
```



```
ans =  
  
struct with fields:  
  
    VehicleSpeed: 0  
    EngineRPM: 250
```

### Change Signal Information

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

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

```
msgEngineInfo =  
  
Message with properties:  
  
    Message Identification  
        ProtocolMode: 'CAN'  
        ID: 100  
        Extended: 0  
        Name: 'EngineMsg'  
  
    Data Details  
        Timestamp: 0  
        Data: [23 205 0 0 0 0 0 0]  
        Signals: [1x1 struct]  
        Length: 8  
  
    Protocol Flags  
        Error: 0  
        Remote: 0  
  
    Other Information  
        Database: [1x1 can.Database]  
        UserData: []
```

```
ans =  
  
struct with fields:
```

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

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

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

```
msgEngineInfo =
```

```
Message with properties:
```

```
Message Identification
```

```
ProtocolMode: 'CAN'
```

```
ID: 100
```

```
Extended: 0
```

```
Name: 'EngineMsg'
```

```
Data Details
```

```
Timestamp: 0
```

```
Data: [23 205 0 0 71 0 0 0]
```

```
Signals: [1x1 struct]
```

```
Length: 8
```

```
Protocol Flags
```

```
Error: 0
```

```
Remote: 0
```

```
Other Information
```

```
Database: [1x1 can.Database]
```

```
UserData: []
```

```
ans =
```

```
struct with fields:
```

```
VehicleSpeed: 71
```

```
EngineRPM: 5.5003e+03
```

## Receive Messages with Database Information

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

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

```
rxCh =
```

```
Channel with properties:
```

```
Device Information
```

```
    DeviceVendor: 'MathWorks'
        Device: 'Virtual 1'
    DeviceChannelIndex: 2
    DeviceSerialNumber: 0
        ProtocolMode: 'CAN'
```

```
Status Information
```

```
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

```
Channel Information
```

```
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    BusSpeed: 500000
        SJW: []
        TSEG1: []
        TSEG2: []
    NumOfSamples: []
```

```
Other Information
```

```
    Database: [1x1 can.Database]
```

```
UserData: []
```

### Receive Messages

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

```
start(rxCh);  
generateMsgsDb();  
rxMsg = receive(rxCh, Inf, 'OutputFormat', 'timetable');  
rxMsg(1:15, :)
```

```
ans =
```

```
15x8 timetable
```

Time	ID	Extended	Name	Data	Length
0.0054711 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.0054911 sec	200	false	'TransmissionMsg'	[1x8 uint8]	8
0.0054972 sec	400	false	'DoorControlMsg'	[1x8 uint8]	8
0.0055005 sec	600	false	'WindowControlMsg'	[1x4 uint8]	4
0.0055028 sec	800	false	'SunroofControlMsg'	[1x2 uint8]	2
0.018215 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.041219 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.041224 sec	200	false	'TransmissionMsg'	[1x8 uint8]	8
0.067195 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.092198 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.092205 sec	200	false	'TransmissionMsg'	[1x8 uint8]	8
0.11622 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.11623 sec	400	false	'DoorControlMsg'	[1x8 uint8]	8
0.14321 sec	100	false	'EngineMsg'	[1x8 uint8]	8
0.14321 sec	200	false	'TransmissionMsg'	[1x8 uint8]	8

Stop the channel and clear it from the workspace.

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

### Examine a Received Message

Inspect a received message to see the applied database decoding.

```
rxMsg(10, :)
rxMsg.Signals{10}
```

```
ans =
```

```
1x8 timetable
```

Time	ID	Extended	Name	Data	Length	Signal
0.092198 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st

```
ans =
```

```
struct with fields:
```

```
VehicleSpeed: 50
EngineRPM: 3.5696e+03
```

### Extract All Instances of a Specified Message

Use MATLAB notation to extract all instances of a specified message by name.

```
allMsgEngine = rxMsg(strcmpi('EngineMsg', rxMsg.Name), :);
allMsgEngine(1:15, :)
```

```
ans =
```

```
15x8 timetable
```

Time	ID	Extended	Name	Data	Length	Signal
0.0054711 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.018215 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.041219 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.067195 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.092198 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.11622 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.14321 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st
0.16721 sec	100	false	'EngineMsg'	[1x8 uint8]	8	[1x1 st

```

0.19222 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s
0.21819 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s
0.24121 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s
0.26719 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s
0.29321 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s
0.31821 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s
0.34224 sec    100    false    'EngineMsg'    [1x8 uint8]    8    [1x1 s

```

### Plot Physical Signal Values

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

```

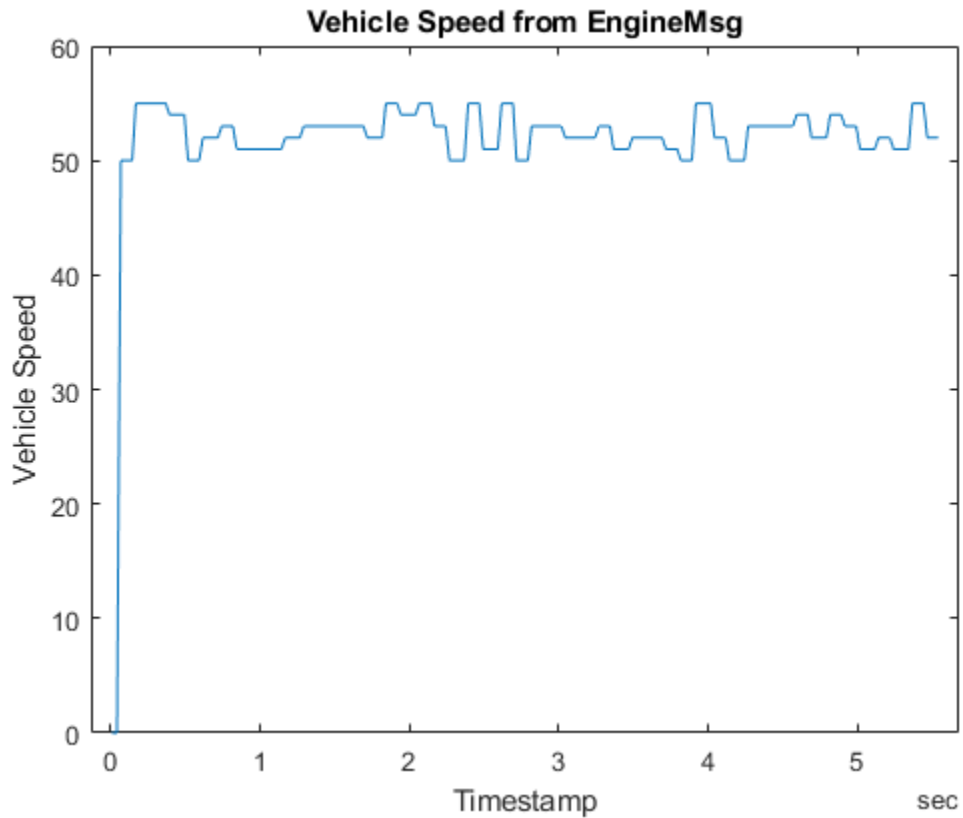
signalTimetable = canSignalTimetable(rxMsg, 'EngineMsg');
signalTimetable(1:15, :)
plot(signalTimetable.Time, signalTimetable.VehicleSpeed)
title('Vehicle Speed from EngineMsg', 'FontWeight', 'bold')
xlabel('Timestamp')
ylabel('Vehicle Speed')

```

ans =

15x2 timetable

Time	VehicleSpeed	EngineRPM
0.0054711 sec	0	250
0.018215 sec	0	250
0.041219 sec	0	250
0.067195 sec	50	3569.6
0.092198 sec	50	3569.6
0.11622 sec	50	3569.6
0.14321 sec	50	3569.6
0.16721 sec	55	3621.3
0.19222 sec	55	3621.3
0.21819 sec	55	3621.3
0.24121 sec	55	3621.3
0.26719 sec	55	3621.3
0.29321 sec	55	3663.9
0.31821 sec	55	3663.9
0.34224 sec	55	3663.9



## See Also

### More About

- “View Message Information in a CAN Database” on page 4-16
- “View Signal Information in a CAN Message” on page 4-18
- “Attach a CAN Database to Existing Messages” on page 4-19

### View Message Information in a CAN Database

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

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

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

```
msgInfo = messageInfo(db)
```

```
msgInfo =
```

```
5x1 struct array with fields:
```

```
    Name  
    Comment  
    ID  
    Extended  
    Length  
    Signals
```

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

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

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

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

```
msgInfo =
```

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

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

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



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

## See Also

### Functions

messageInfo

### More About

- “Load .dbc Files and Create Messages” on page 4-2
- “View Signal Information in a CAN Message” on page 4-18
- “Attach a CAN Database to Existing Messages” on page 4-19

# View Signal Information in a CAN Message

You can get signal definition information on a specific signal or all signals in a CAN message with database definitions attached. Provide the message name or the ID as a parameter in the command:

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

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

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

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

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

## See Also

### Functions

`signalInfo`

### More About

- “Load .dbc Files and Create Messages” on page 4-2
- “View Message Information in a CAN Database” on page 4-16
- “Attach a CAN Database to Existing Messages” on page 4-19

## Attach a CAN Database to Existing Messages

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

To attach a database to a message, type:

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

---

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

---

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

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

---

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

---

## See Also

### Functions

attachDatabase

### More About

- “Load .dbc Files and Create Messages” on page 4-2
- “View Message Information in a CAN Database” on page 4-16
- “View Signal Information in a CAN Message” on page 4-18



# Monitoring Vehicle CAN Bus

---

- “Vehicle CAN Bus Monitor” on page 5-2
- “Using the Vehicle CAN Bus Monitor” on page 5-9

## Vehicle CAN Bus Monitor

In this section...
“About the Vehicle CAN Bus Monitor” on page 5-2
“Opening the Vehicle CAN Bus Monitor” on page 5-2
“Vehicle CAN Bus Monitor Fields” on page 5-2

### About the Vehicle CAN Bus Monitor

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

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

You cannot programmatically configure the Vehicle CAN Bus Monitor. However, you can use it to independently visualize bus traffic generated on CAN channels by MATLAB or Simulink CAN blocks.

### Opening the Vehicle CAN Bus Monitor

To open the Vehicle CAN Bus Monitor, type `canTool` in the MATLAB Command Window.

### Vehicle CAN Bus Monitor Fields

The CAN bus monitor has the following menus, buttons and table.

Timestamp	ID	Name	Length	Data
49.271169	4B1		8	4C 08 4B 0E 4D 02 ...
49.266196	201		8	17 A7 00 00 00 00 8...
49.171186	4B1		8	4C 08 4B 0E 4D 02 ...
49.166148	201		8	17 A7 00 00 00 00 8...
49.071145	4B1		8	69 00 67 13 6A ED 6...
49.066181	201		8	43 7E 00 00 00 00 A...
48.971186	4B1		8	4C 08 4B 0E 4D 02 ...
48.966132	201		8	49 5B 00 00 00 00 1...
48.871334	4B1		8	45 E7 44 18 47 B6 4...
48.866132	201		8	43 9D 00 00 00 00 1...
48.771129	4B1		8	4C 08 4B 0E 4D 02 ...
48.766140	201		8	4C 24 00 00 00 00 9...
48.671171	4B1		8	4C 08 4B 0E 4D 02 ...
48.666116	201		8	55 D0 00 00 00 00 9...
48.571138	4B1		8	4C 08 4B 0E 4D 02 ...
48.566133	201		8	55 D0 00 00 00 00 5...
48.471114	4B1		8	4C 08 4B 0E 4D 02 ...
48.466117	201		8	21 F0 00 00 00 00 A...
48.371122	4B1		8	03 0B 01 75 04 A1 F...
48.366133	201		8	25 D3 00 00 00 00 6...
48.271106	4B1		8	0B 05 09 68 0C A2 ...
48.266109	201		8	59 69 00 00 00 00 3...
48.171172	4B1		8	0B 05 09 68 0C A2 ...
48.166109	201		8	59 69 00 00 00 00 3...
48.071148	4B1		8	2F 22 2D 66 30 DD ...
48.066101	201		8	30 F7 00 00 00 00 5...
47.971115	4B1		8	0B 8C 09 EF 0D 2A ...

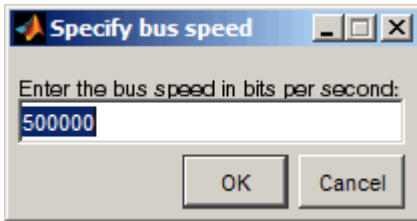
### File Menu

- **Save Messages** — Saves messages to a log file.
- **Clear Messages** — Clears messages in the Vehicle CAN Bus Monitor window.

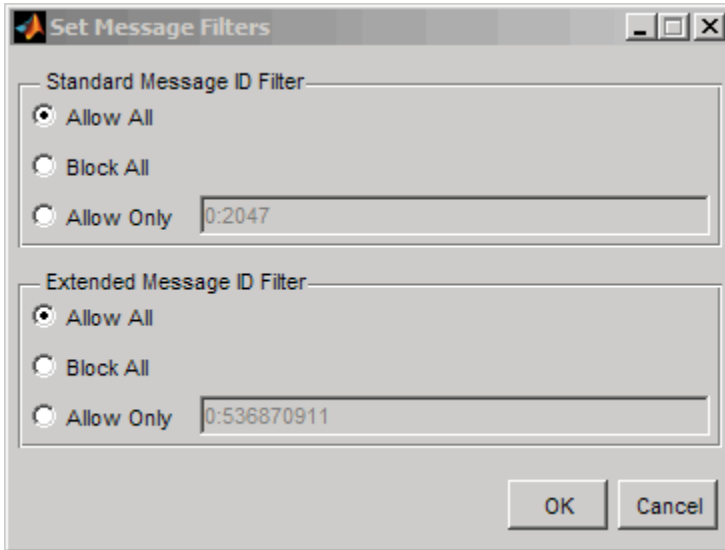
- **Exit** — Click to exit the Vehicle CAN Bus Monitor.

**Configure Menu**

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



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



- Standard Message ID Filter
  - Allow All — Select to allow all standard ID messages.



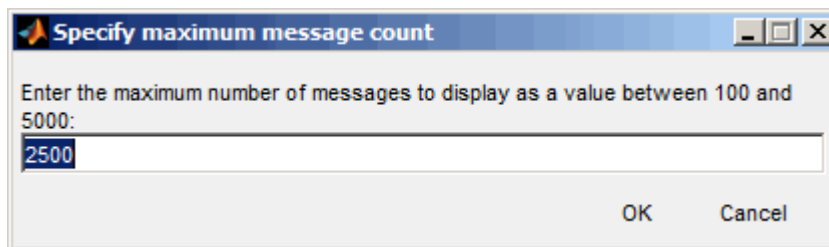
- **Block All** — Select to block all standard ID messages.
- **Allow only** — Select to set up custom filtering of messages. Type the standard message IDs that you want to allow.
- **Extended Message ID Filter**
  - **Allow All** — Select to allow all extended ID messages.
  - **Block All** — Select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

### Run Menu

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

### View Menu

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



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

option, the tool displays all instances of all messages in the order that the selected channel receives them.

### Help Menu

- **Documentation** — Select this option to see the documentation for the Vehicle CAN Bus Monitor.
- **About Vehicle Network Toolbox** — Select this option to view the toolbox version and release information.

### Buttons

**Start** 

Displays message activity on the selected channel.

**Pause** 

Pauses the display of message activity on the selected channel.

**Stop** 

Stops displaying messages on the selected channel.

**Save messages** 

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

**Clear messages** 


Click this button to clear messages in the Vehicle CAN Bus Monitor window.

**Show unique messages** 

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


**Docking** 

Click this button to dock the Vehicle CAN Bus Monitor to the MATLAB desktop. To

undock, click .

**Undocking** 

Click this button to undock the Vehicle CAN Bus Monitor from the MATLAB desktop.

To dock, click .

**Message Table****Timestamp**

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

**ID**

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

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

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

**Name**

Displays the name of the message, if available.

**Length**

Displays the length of the message in bytes.

**Data**

Displays the data in the message in hexadecimal format.

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

If you are using a database on page 5-10, click the + sign to see signal information. The tool displays the signal name and the physical value of the message, as defined in the attached database.

## **See Also**

### **More About**

- “Using the Vehicle CAN Bus Monitor” on page 5-9

## Using the Vehicle CAN Bus Monitor

This topic shows many of the tasks you can perform with the Vehicle CAN Bus Monitor.

### In this section...

“View Messages on a Channel” on page 5-9

“Configure the Channel Bus Speed” on page 5-9

“Filter CAN Messages in Vehicle CAN Bus Monitor” on page 5-10

“Attach a Database” on page 5-10

“Change the Message Count” on page 5-13

“Change the Number Format” on page 5-13

“View Unique Messages” on page 5-13

“Save the Message Log File” on page 5-14

### View Messages on a Channel

- 1 Open the Vehicle CAN Bus Monitor and select the device and channel connected to your CAN bus by selecting **Configure > Channel**.
- 2 The Vehicle CAN Bus Monitor defaults to the bus speed set in the device driver. You can also configure a new bus speed. See Configuring the Channel Bus Speed on page 5-9
- 3 Click **Start**, or select **Run > Start**.
- 4 To pause the display, click **Pause** or select **Run > Pause**.
- 5 To stop the display, click **Stop** or select **Run > Stop**.

### Configure the Channel Bus Speed

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

To configure a new bus speed:

- 1 Select **Configure > Bus Speed**.
- 2 Type the desired value in the Specify bus speed dialog box.

- 3 Click **OK**.

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

### Filter CAN Messages in Vehicle CAN Bus Monitor

Filter CAN messages to allow or block messages displayed in the Vehicle CAN Bus Monitor.

To set up filters:

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

---

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

---

- 4 Click **OK**.

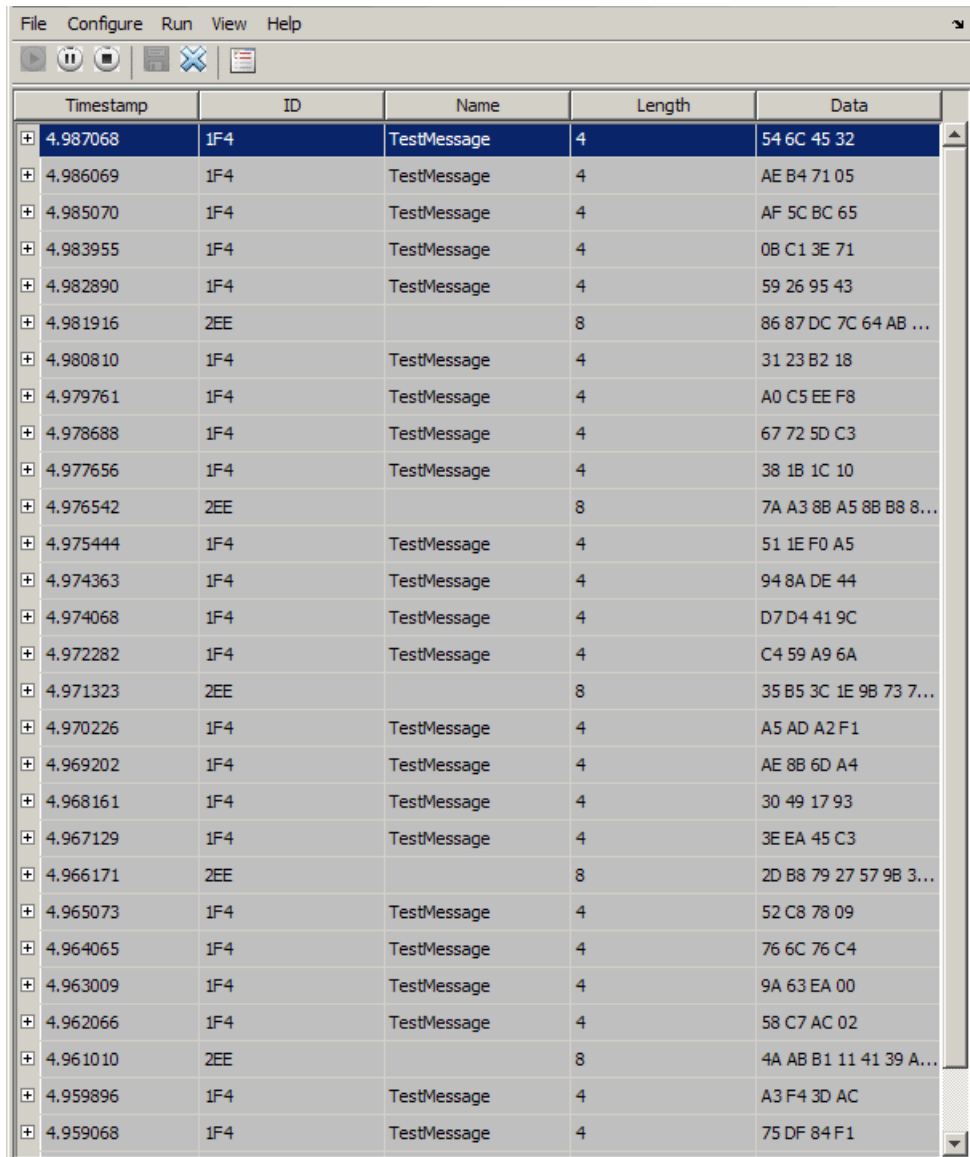
### Attach a Database

Attach a database to the Vehicle CAN Bus Monitor to see signal information of the displayed messages.

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

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

## 5 Monitoring Vehicle CAN Bus



The screenshot shows a software window with a menu bar (File, Configure, Run, View, Help) and a toolbar with icons for play, stop, refresh, and help. Below the toolbar is a table with the following columns: Timestamp, ID, Name, Length, and Data. The table contains 28 rows of data, with the first row highlighted in blue. The data in the table is as follows:

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

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



+	4.987068	1F4	TestMessage	4	54 6C 45 32
-	4.986069	1F4	TestMessage	4	AE B4 71 05
		Signal Name	Physical Value		
		Sig1	174.000000		
		Sig2	180.000000		
		Sig3	113.000000		
		Sig4	5.000000		

The tool displays the signal name as defined in the attached database and the signal's physical value.

## Change the Message Count

You can change the maximum number of messages displayed to a value from 100 through 5000.

- 1 Select **View > Maximum Message Count**.
- 2 In the Specify maximum message count dialog box, type the number of messages you want displayed at one time.
- 3 Click **OK**.

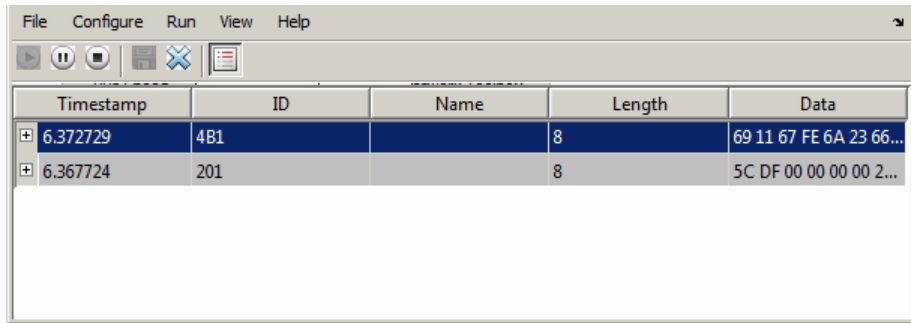
## Change the Number Format

By default the message data is displayed in hexadecimal format. To change the display to decimal format, select **View > Number Format > Decimal**.

## View Unique Messages

To view the most recent instance of each unique message received on the channel, select **View > Show Unique Messages**. In this view, you do not see messages scroll up, but each message refreshes its data and timestamp with each new instance. You can also click





The screenshot shows a software window titled 'Vehicle CAN Bus Monitor' with a menu bar (File, Configure, Run, View, Help) and a toolbar with icons for play, pause, stop, refresh, close, and print. Below the toolbar is a table with the following data:

Timestamp	ID	Name	Length	Data
6.372729	4B1		8	69 11 67 FE 6A 23 66...
6.367724	201		8	5C DF 00 00 00 00 2...


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

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

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

### Save the Message Log File

You can save a log file of the messages currently displayed. If running, you need to stop or pause the display before saving a log file.

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

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

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

## **See Also**

### **More About**

- “Vehicle CAN Bus Monitor” on page 5-2



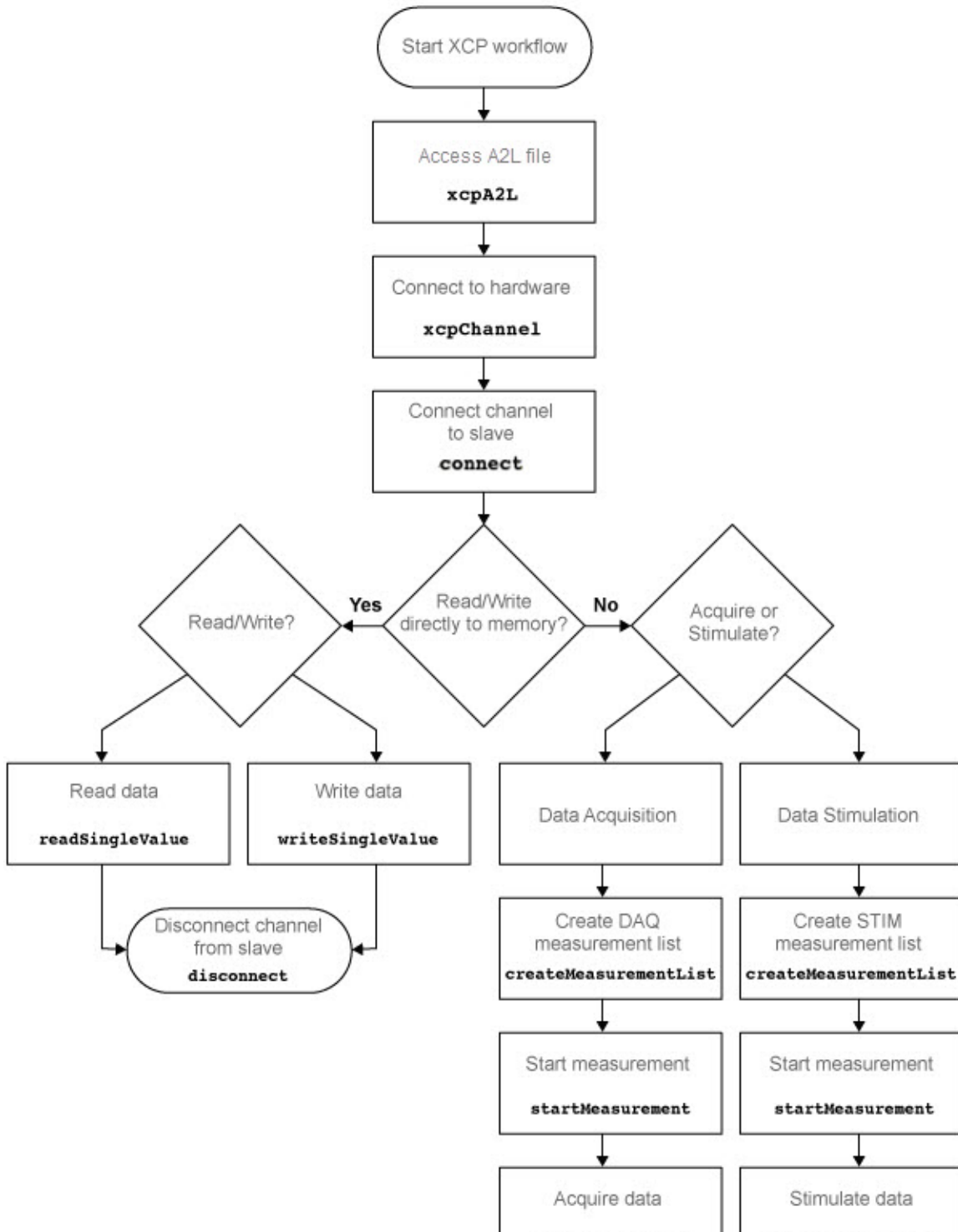
# XCP Communication Workflows

---

## **XCP Database and Communication Workflow**

This workflow helps you:

- Manage an A2L database
- Connect to an XCP device
- Create an XCP channel
- Acquire and stimulate data
- Read and write to memory



## See Also

### Functions

connect | createMeasurementList | disconnect | freeMeasurementLists |  
getEventInfo | getMeasurementInfo | isConnected | isMeasurementRunning |  
readDAQListData | readSingleValue | startMeasurement | stopMeasurement |  
viewMeasurementLists | writeSTIMListData | writeSingleValue | xcpA2L |  
xcpChannel

### Properties

A2LFileName | A2LFileName | DAQInfo | Events | FileName | FilePath | Measurements |  
ProtocolLayerInfo | SeedKeyDLL | SeedKeyDLL | SlaveName | SlaveName | SlaveName |  
TransportLayer | TransportLayer | TransportLayerCANInfo | TransportLayerDevice |  
TransportLayerDevice

### Blocks

XCP CAN Configuration | XCP CAN Data Acquisition | XCP CAN Data Stimulation | XCP  
CAN Transport Layer | XCP UDP Configuration | XCP UDP Data Acquisition | XCP UDP  
Data Stimulation



# A2L File

---

## Inspect the Contents of an A2L File

In this section...
"Access an A2L File" on page 7-2
"Access Measurement Information" on page 7-2
"Access Event Information" on page 7-4

### Access an A2L File

To use an A2L file, create a file object in your MATLAB session. At the Command Window prompt, type:

```
a2lfile = xcpA2L('filename.a2l')
```

Here `a2lfile` is a variable assigned with the A2L object, and `filename.a2l` is the name of your A2L file. If your A2L file is not in the current working directory, specify the necessary partial or full path to the file:

```
a2lfile = xcpA2L('path\filename.a2l');
```

---

**Tip** A2L file names containing non-alphanumeric characters such as equal signs or ampersands are not supported. You can use periods in your database name. Rename any A2L files with non-alphanumeric characters before you use them.

---

This command returns an A2L object that you can use for live communication with a slave module using XCP channels.

### Access Measurement Information

This example shows how to open an A2L file and access measurement information.

Open an A2L file:

```
a2lfile = xcpA2L('XCPSIM.a2l');
```

Display properties of the A2L object:

```
a2lfile
```

A2L with properties:

```

        FileName: 'XCPSIM.a2l'
        FilePath: 'H:\Documents\work\XCPSIM.a2l'
        SlaveName: 'CPP'
        ProtocolLayerInfo: [1x1 struct]
        DAQInfo: [1x1 struct]
        TransportLayerCANInfo: [1x1 struct]
        Events: {'Key T' '10 ms' '100ms' '1ms' 'FilterBypassDaq' 'FilterBypassSt'}
        Measurements: {1x38 cell}

```

View all available measurements:

### a2lfile.Measurements

ans =

Columns 1 through 8

```
'BitSlice' 'BitSlice0' 'BitSlice1' 'BitSlice2' 'Counter_B4' 'Counter_B5' 'Counter_B6' 'Counter_B7'
```

Columns 9 through 16

```
'FW1' 'KL10Output' 'PWM' 'PWMFiltered' 'PWM_Level' 'ShiftByte' 'Shifter_B0' 'Shifter_B1'
```

Columns 17 through 25

```
'Shifter_B2' 'Shifter_B3' 'TestStatus' 'Triangle' 'ampl' 'bit12Counter' 'byte1' 'byte2' 'byte3'
```

Columns 26 through 33

```
'byte4' 'byteCounter' 'bytePWMFilter' 'channel3' 'dwordCounter' 'map1InputX' 'map1InputY' 'map10'
```

Columns 34 through 38

```
'period' 'sbytePWMLevel' 'v' 'vin' 'wordCounter'
```

Get information about the BitSlice measurement:

```
getMeasurementInfo(a2lfile, 'Triangle')
```

ans =

```

        Name: 'Triangle'
        LongIdentifier: 'Triangle test signal used for PWM output PWM'
        DataType: 'SBYTE'
        Conversion: 'BitSlice.CONVERSION'
        Resolution: 0
        Accuracy: 0
        LowerLimit: -50
        UpperLimit: 50
        ECUAddress: 4951377
        ECUAddressExtension: 0
        ByteOrder: 'MSB_LAST'

```

```
SizeInBytes: 1
SizeInNibbles: 2
SizeInBits: 8
MATLABType: 'int8'
```

## Access Event Information

This example shows how to open an A2L file and access event information.

Open an A2L file:

```
a2lfile = xcpA2L('XCPSIM.a2l');
```

Display properties of the A2L object:

```
a2lfile
```

A2L with properties:

```
FileName: 'XCPSIM.a2l'
FilePath: 'H:\Documents\work\XCPSIM.a2l'
SlaveName: 'CPP'
ProtocolLayerInfo: [1x1 struct]
DAQInfo: [1x1 struct]
TransportLayerCANInfo: [1x1 struct]
Events: {'Key T' '10 ms' '100ms' '1ms' 'FilterBypassDaq' 'FilterBypassSt'}
Measurements: {1x38 cell}
```

View all available events:

```
a2lfile.Events
```

```
ans =
    'Key T'    '10 ms'    '100ms'    '1ms'    'FilterBypassDaq'    'FilterBypassSt'
```

Get information for the 10 ms event:

```
getEventInfo(a2lfile, '10 ms')
```

```
ans =
    Name: '10 ms'
    Direction: 'DAQ_STIM'
    MaxDAQList: 255
    ChannelNumber: 1
    ChannelTimeCycle: 10
    ChannelTimeUnit: 6
```

```
ChannelPriority: 0  
ChannelTimeCycleInSeconds: 0.0100
```

## See Also

### Functions

[getEventInfo](#) | [getMeasurementInfo](#) | [xcpA2L](#)



# Universal Measurement & Calibration Protocol (XCP)

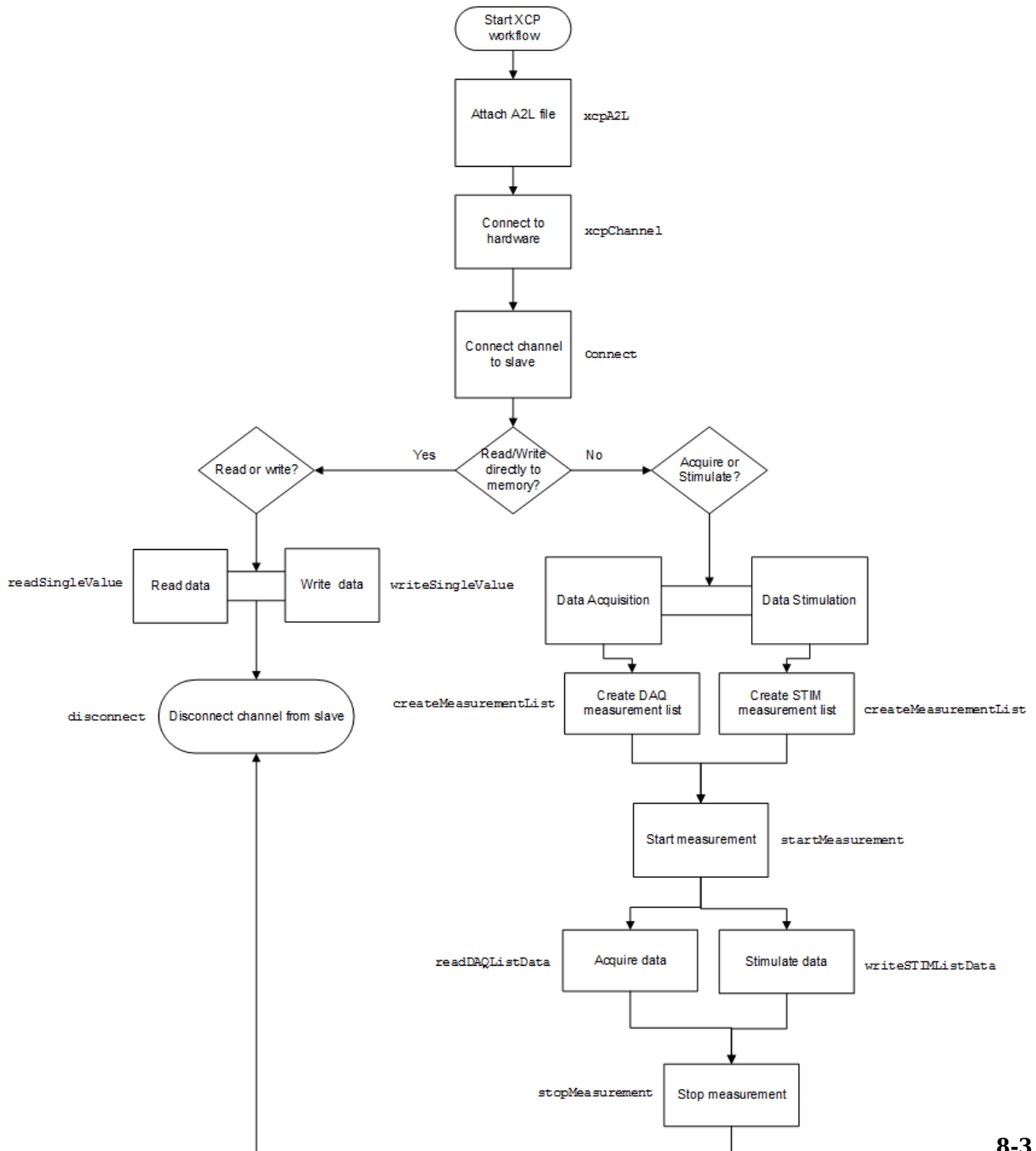
---

- “XCP Hardware Connection” on page 8-2
- “Read a Single Value” on page 8-6
- “Write a Single Value” on page 8-7
- “Read a Calibrated Measurement” on page 8-8
- “Acquire Measurement Data via Dynamic DAQ Lists” on page 8-9
- “Stimulate Measurement Data via Dynamic STIM Lists” on page 8-10

## **XCP Hardware Connection**

You can connect your XCP master to a slave module using the CAN protocol. This allows you to use events and access measurements on the slave module.





## Create XCP Channel Using CAN Device

This example shows how to create an XCP CAN channel connection and access channel properties. The example also shows how to unlock the slave using seed key security.

Access an A2L file that describes the slave module.

```
a2lfile = xcpA2L('C:\work\XCPSIM.a2l')
```

```
a2lfile =
```

```
    A2L with properties:
```

```
        FileName: 'XCPSIM.a2l'
        FilePath: 'C:\work\XCPSIM.a2l'
        SlaveName: 'CPP'
    ProtocolLayerInfo: [1x1 struct]
        DAQInfo: [1x1 struct]
    TransportLayerCANInfo: [1x1 struct]
        Events: {1x6 cell}
        Measurements: {1x38 cell}
```

Create an XCP channel using Vector virtual CAN channel 1.

```
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1)
```

```
xcpch =
```

```
    Channel with properties:
```

```
        SlaveName: 'CPP'
        A2LFileName: 'XCPSIM.a2l'
        TransportLayer: 'CAN'
    TransportLayerDevice: [1x1 struct]
        SeedKeyDLL: []
```

## Configure the Channel to Unlock the Slave

This example shows how to configure the channel to unlock the slave using a dll that contains a seed and key security algorithm when your module is locked for Stimulation operations.

Create your XCP channel and set the channel SeedKeyDLL property.

```
xcpch.SeedKeyDLL = ('C:\Work\SeedNKeyXcp.dll')
```

```
xcpch =
```

```
Channel with properties:
```

```
    SlaveName: 'CPP'  
    A2LFileName: 'XCPSIM.a2l'  
    TransportLayer: 'CAN'  
    TransportLayerDevice: [1x1 struct]  
    SeedKeyDLL: 'C:\Work\SeedNKeyXcp.dll'
```

## Read a Single Value

This example shows how to access a single value by name. The value is read directly from memory.

Create an XCP channel with access to an A2L file.

```
a2lfile = xcpA2L('C:\work\XCPSIM.a2l');  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the slave.

```
connect(xcpch)
```

Read a single value of the Triangle measurement directly from memory.

```
readSingleValue(xcpch, 'Triangle')
```

```
ans =
```

```
50
```

## Write a Single Value

This example shows how to write a single value by name. The value is written directly to memory.

Create an XCP channel linked to an A2L file.

```
a2lfile = xcpA2L('C:\work\XCPSIM.a2l');  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the slave.

```
connect(xcpch)
```

Write a single value.

```
writeSingleValue(xcpch, 'Triangle', 50)
```

## Read a Calibrated Measurement

This example shows a typical workflow for reading a calibration file and using a translation table to calibrate a measurement reading.

Read the engine management ECU calibration file.

```
a2lobj = xcpA2L('ems.a2l');
```

Connect to the ECU.

```
ch = xcpChannel(a2lobj, 'UDP', '192.168.1.55', 5555);
```

Set the table that translates a pedal position to a torque demand.

```
writeCharacteristic(ch, 'tq_accel_request', ...  
[0 2 4 9 14 24 48 72 96 144 192 204 216 228 240]);
```

Set the pedal position to 50%.

```
writeMeasurement(ch, 'pedal_position', 50);
```

Read the demand.

```
value = readMeasurement(ch, 'tq_demand')
```

```
value =  
    96
```

## See Also

### Functions

readAxis | readCharacteristic | readMeasurement | writeAxis |  
writeCharacteristic | writeMeasurement

## Acquire Measurement Data via Dynamic DAQ Lists

This example shows how to create a dynamic data acquisition list and assign measurements to the list. You can acquire data for measurements in this list from the slave.

Create an XCP channel linked to an A2L file and connect it to the slave.

```
a2lfile = xcpA2L('C:\work\XCPSIM.a2l');  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);  
connect(xcpch)
```

Create a DAQ list for the '10 ms' event with 'PWMFiltered' and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', {'PWMFiltered', 'Triangle'});
```

Start measurement activity.

```
startMeasurement(xcpch)
```

Read 10 samples of data from the configured measurement list for the 'Triangle' measurement.

```
readDAQListData(xcpch, 'Triangle', 10)
```

```
18  18  18  18  18  18  18  18  18  18
```

## Stimulate Measurement Data via Dynamic STIM Lists

This example shows how to create a dynamic data stimulation list and assign measurements to the list. You can stimulate data for specific measurements in this list.

Create an XCP channel linked to an A2L file and connect it.

```
a2lfile = xcpA2L('C:\work\XCPSIM.a2l');  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1',1);  
connect(xcpch)
```

---

**Note** If your module is locked for STIM operations, configure the channel to unlock the slave.

---

Create a STIM list for the '100ms' event with 'PWMFiltered' and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'STIM', '100ms', {'PWMFiltered', 'Triangle'});
```

Start the measurement.

```
startMeasurement(xcpch)
```

Write 10 to the configured measurement list for the 'Triangle' measurement.

```
writeSTIMListData(xcpch, 'Triangle', 10);
```



# J1939

---

- “J1939 Interface” on page 9-2
- “J1939 Parameter Group Format” on page 9-3
- “J1939 Network Management” on page 9-5
- “J1939 Transport Protocols” on page 9-6
- “J1939 Channel Workflow” on page 9-7

## **J1939 Interface**

J1939 is a high-level protocol built on the CAN bus that provides serial data communication between electronic control units (ECUs) in heavy-duty vehicles. Applications of J1939 include:

- Diesel power-train applications
- In-vehicle networks for buses and trucks
- Agriculture and forestry machinery
- Truck-trailer connections
- Military vehicles
- Fleet management systems
- Recreational vehicles
- Marine navigation systems

The J1939 protocol uses CAN as the physical layer, which defines the communication between ECUs in the vehicle network. The protocol has a second data-link layer that defines rules of communication and error detection. A third application layer defines the data transferred over the network.

## **See Also**

### **More About**

- “J1939 Parameter Group Format” on page 9-3
- “J1939 Network Management” on page 9-5
- “J1939 Transport Protocols” on page 9-6
- “J1939 Channel Workflow” on page 9-7

## J1939 Parameter Group Format

The application layer deals with parameter groups (PGs) sent and received over the network. J1939 protocol uses broadcast messages, or messages sent over the CAN bus without a defined destination. Devices on the same network can access these messages without permission or special requests. If a device requires a specific message, include the device destination address in the message identifier.

The message contains a group of parameters that define related messages. For example, a message sent to the engine controller can contain both engine speed and RPM. These parameters are represented in the CAN identifier by a parameter group number (PGN). Parameter groups use 29-bit identifiers with this message structure:

<b>Parameter</b>	<b>Priority</b>	<b>Reserved</b>	<b>Data Page</b>	<b>PDU Format</b>	<b>PDU Specific</b>	<b>Source Address</b>
Size	3 bits	1 bit	1 bit	8 bits	8 bits	8 bits

- First three bits represent the priority of the message on the network. Zero is the highest priority.
- The next bit is reserved for future use. For transmit messages, set this to zero.
- The next bit is the data page, which extends the maximum number of possible PGs in the identifier.
- The next 8 bits are the protocol data unit (PDU) format, which specifies whether the message is targeted for a single device or is broadcast. If the PDU is less than 240, then the message is sent to a specific device and if it over 240, it is sent to the entire network.
- The next 8 bits are the PDU specific, which contains the address of the device when the PDU format is less than 240. If PDU format is greater than 240, PDU specific contains group extension, or the number of extended broadcast messages in this parameter group.
- The last 8 bits contain the source address, which is the address of the device sending the parameter groups.

The protocol application layer transmits the PG on the CAN network. PG length can be up to 1785 bytes and is not limited by the length of a CAN message. However, PGs larger than 8 bytes must be transmitted using a transport protocol.

## **See Also**

### **More About**

- “J1939 Interface” on page 9-2
- “J1939 Network Management” on page 9-5
- “J1939 Transport Protocols” on page 9-6
- “J1939 Channel Workflow” on page 9-7

# J1939 Network Management

Each device on a J1939 network has a unique address. The PDU Specific uses device addresses to send parameter groups (PG) to a specific device. Static addresses between zero and 253 are assigned for every device on the network. You can also assign 254, which is a null and 255, which is a global address.

## Address Claiming

The application sending a PG must claim an ECU address. The application sends an address claiming PG first, and resumes sending other PGs if there is not address conflict. If the source application encounters an address conflict, it can send a PG to the global (255) address to request all devices to declare their addresses. It can then claim one of the unused addresses.

## See Also

### More About

- “J1939 Interface” on page 9-2
- “J1939 Parameter Group Format” on page 9-3
- “J1939 Transport Protocols” on page 9-6
- “J1939 Channel Workflow” on page 9-7

## **J1939 Transport Protocols**

J1939 transport protocol breaks up PGs larger than 8 data bytes and up to 1785 bytes, into multiple packets. The transport protocol defines the rules for packaging, transmitting, and reassembling the data.

- Messages that have multiple packets are transmitted with a dedicated PGN, and have the same message ID and similar functionality.
- The length of each message in the packet must be 8 bytes or fewer.
- The first byte in the data field of a message specifies the sequence of the message (one to 255) and the next seven bytes contain the original data.
- All unused bytes in the data field are set to zero.
- A different PGN controls the message flow.

The data package is passed to the application layer after it is reassembled in the order specified by the first data-field byte.

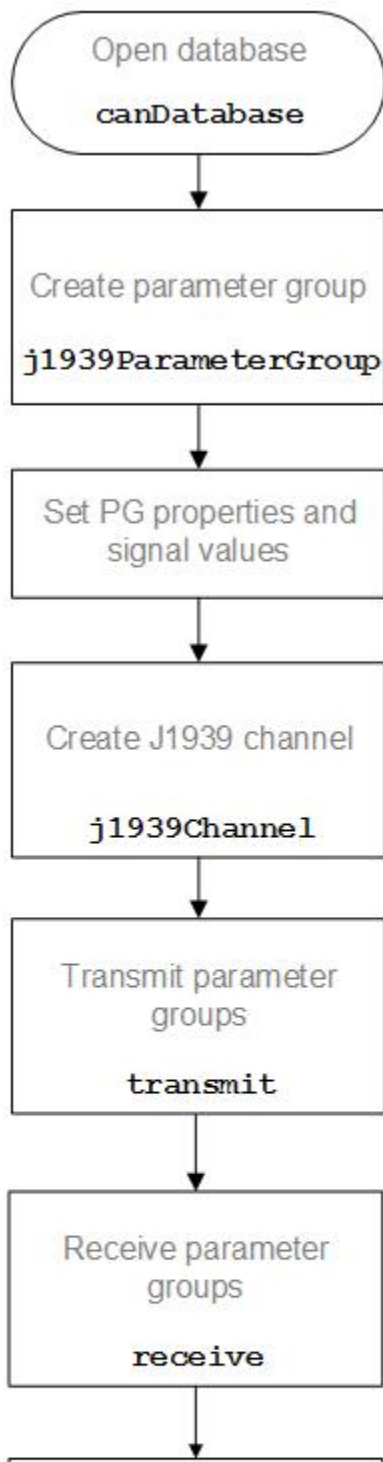
## **See Also**

### **More About**

- “J1939 Interface” on page 9-2
- “J1939 Parameter Group Format” on page 9-3
- “J1939 Network Management” on page 9-5
- “J1939 Transport Protocols” on page 9-6
- “J1939 Channel Workflow” on page 9-7

## **J1939 Channel Workflow**

Transmit and receive parameter groups (PGs) using j1939Channel via a CAN network.





## See Also

### More About

- “J1939 Interface” on page 9-2
- “J1939 Parameter Group Format” on page 9-3
- “J1939 Network Management” on page 9-5
- “J1939 Transport Protocols” on page 9-6



# CAN Communications in Simulink

---

- “Vehicle Network Toolbox Simulink Blocks” on page 10-2
- “CAN Communication Workflows in Simulink” on page 10-3
- “Open the Vehicle Network Toolbox Block Library” on page 10-7
- “Build CAN Communication Simulink Models” on page 10-9
- “Create Custom CAN Blocks” on page 10-28

## Vehicle Network Toolbox Simulink Blocks

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

- **CAN Configuration** — Configure the settings of a CAN device.
- **CAN Log** — Logs messages to file.
- **CAN Pack** — Pack signals into a CAN message.
- **CAN Receive** — Receive CAN messages from a CAN bus.
- **CAN Replay**— Replays logged messages to CAN bus or output port.
- **CAN Transmit** — Transmit CAN messages to a CAN bus.
- **CAN Unpack** — Unpack signals from a CAN message.

The CAN FD Communication block library contains similar blocks for the CAN FD protocol.

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, see “Getting Started with Simulink” (Simulink) to understand its functionality better.

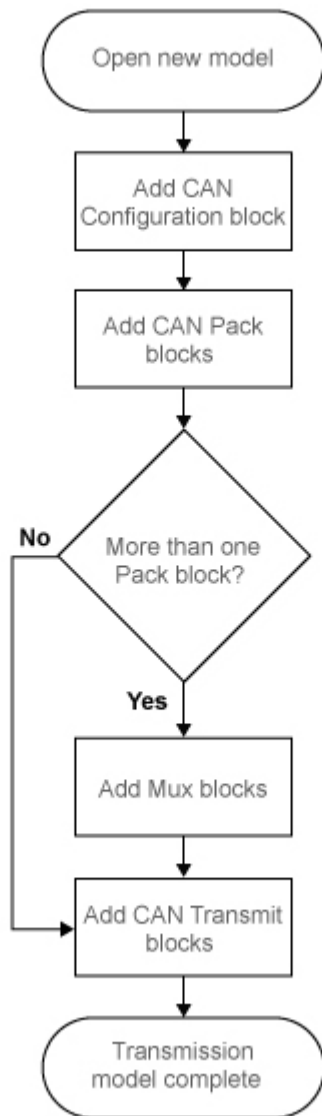
For more detailed information about the blocks in the Vehicle Network Toolbox block library see “CAN Communication in Simulink”.

## CAN Communication Workflows in Simulink

In this section...
"Message Transmission Workflow" on page 10-3
"Message Reception Workflow" on page 10-5

### Message Transmission Workflow

This workflow represents the most common CAN Transmit model. Adjust your model as needed. For more workflow examples, see "Build CAN Communication Simulink Models" on page 10-9 and the "Simulink Tutorials" in the Vehicle Network Toolbox examples.

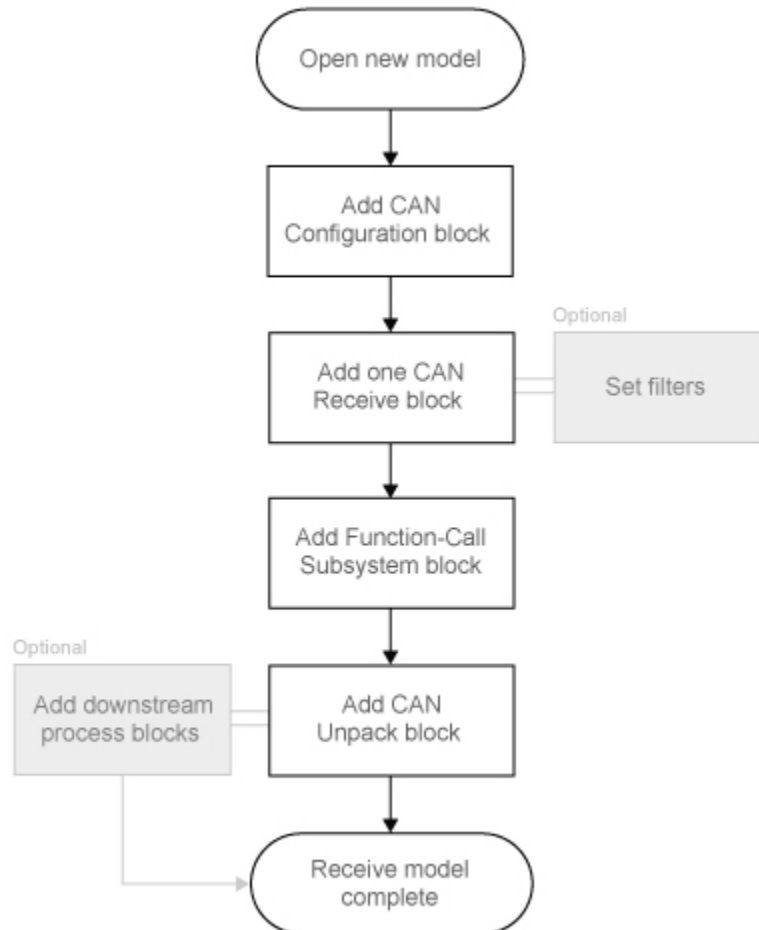


### Using Mux Blocks

- Use a Mux block to combine every message from the source if they are transmitted at the same rate.

- Use one CAN Transmit block for each configured Mux block.

## Message Reception Workflow



## Message Filtering

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

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

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

Set up your CAN Unpack block:

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

### **Downstream Processing**

For any downstream processing using received messages, include blocks:

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



## Open the Vehicle Network Toolbox Block Library

### In this section...

“Using the Simulink Library Browser” on page 10-7

“Using the MATLAB Command Window” on page 10-8

### Using the Simulink Library Browser

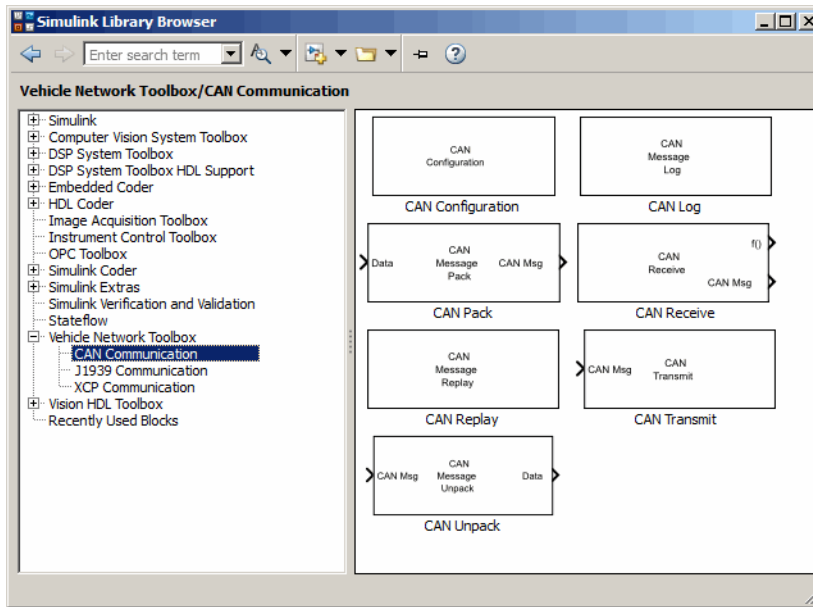
To open the Vehicle Network Toolbox block library, start Simulink by entering the following at the MATLAB command prompt:

```
simulink
```

In the Simulink start page dialog, click **Blank Model**, and then **Create Model**. An empty, Editor window opens.

In the model Editor window, click **View > Library Browser**.

The Simulink Library Browser opens. Its left pane contains a tree of available block libraries in alphabetical order. Expand the **Vehicle Network Toolbox** node and click **CAN Communication**.



## Using the MATLAB Command Window

To open the Vehicle Network Toolbox CAN Communications block library, enter `canlib` in the MATLAB Command window.

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

## Build CAN Communication Simulink Models

This topic comprises the following sections to form a complete CAN communication simulation.

### In this section...

“Build a Message Transmit Model” on page 10-9

“Build a Message Receive Model” on page 10-14

“Save and Run the Model” on page 10-23

### Build a Message Transmit Model

This section illustrates how to send data via a CAN network. The example builds a simple model using Vehicle Network Toolbox blocks with other blocks in the Simulink library, using the following steps:

- “Step 1: Create a New Model” on page 10-9
- “Step 2: Open the Block Library” on page 10-10
- “Step 3: Drag Vehicle Network Toolbox Blocks into the Model” on page 10-10
- “Step 4: Drag Other Blocks to Complete the Model” on page 10-11
- “Step 5: Connect the Blocks” on page 10-12
- “Step 6: Specify the Block Parameter Values” on page 10-12

For this portion of the example

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

Use this section with “Build a Message Receive Model” on page 10-14 and “Save and Run the Model” on page 10-23 to build your complete model and run the simulation.

#### Step 1: Create a New Model

- 1 To start Simulink and create a new model, enter the following at the MATLAB command prompt:

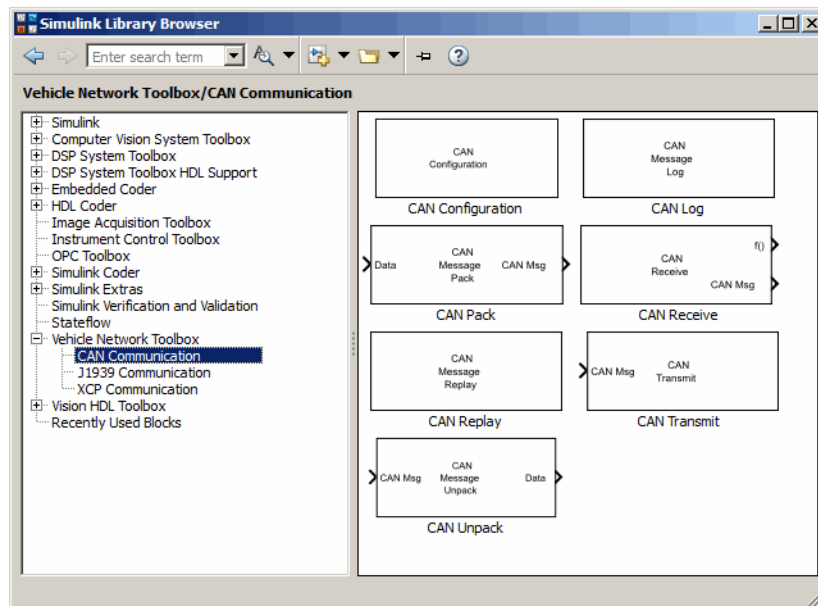
simulink

In the Simulink start page dialog, click **Blank Model**, and then **Create Model**. An empty Editor window opens.

- 2 In the Editor, click **File > Save As** to assign a name to your new model.

## Step 2: Open the Block Library

- 1 In the model Editor window, click **View > Library Browser**.
- 2 The Simulink Library Browser opens. Its left pane contains a tree of available block libraries in alphabetical order. Expand the **Vehicle Network Toolbox** node and click **CAN Communication**.

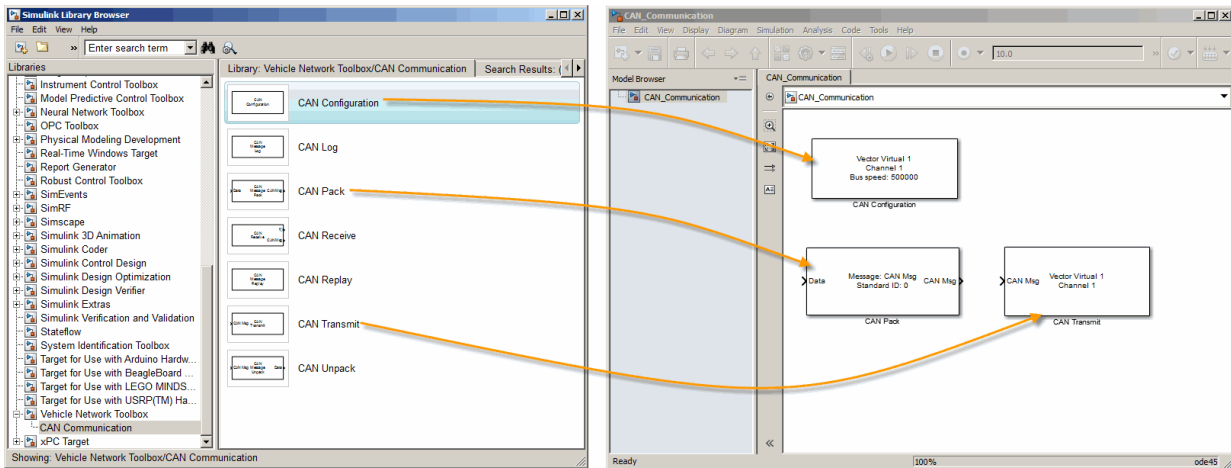


## Step 3: Drag Vehicle Network Toolbox Blocks into the Model

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

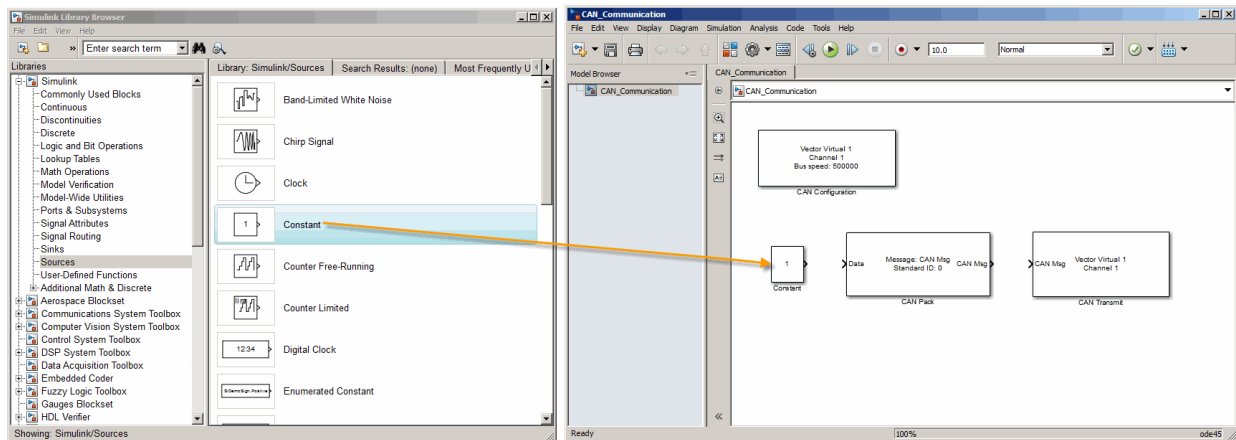
**Note** The default configuration of each block in your model is based on the first appropriate device that it finds on your system. The block settings in the images you see in the following steps might differ from those on your system until all the required settings are applied.

**Note** Block names are not shown by default in the model. To display the hidden block names while working in the model, select **Display** and clear the **Hide Automatic Names** check box.



#### Step 4: Drag Other Blocks to Complete the Model

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



### Step 5: Connect the Blocks

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

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

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

### Step 6: Specify the Block Parameter Values

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

#### Configure the CAN Configuration Block

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

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

Click **OK**.

**Configure the CAN Pack Block**

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

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

Click **OK**.

**Configure the CAN Transmit Block**

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

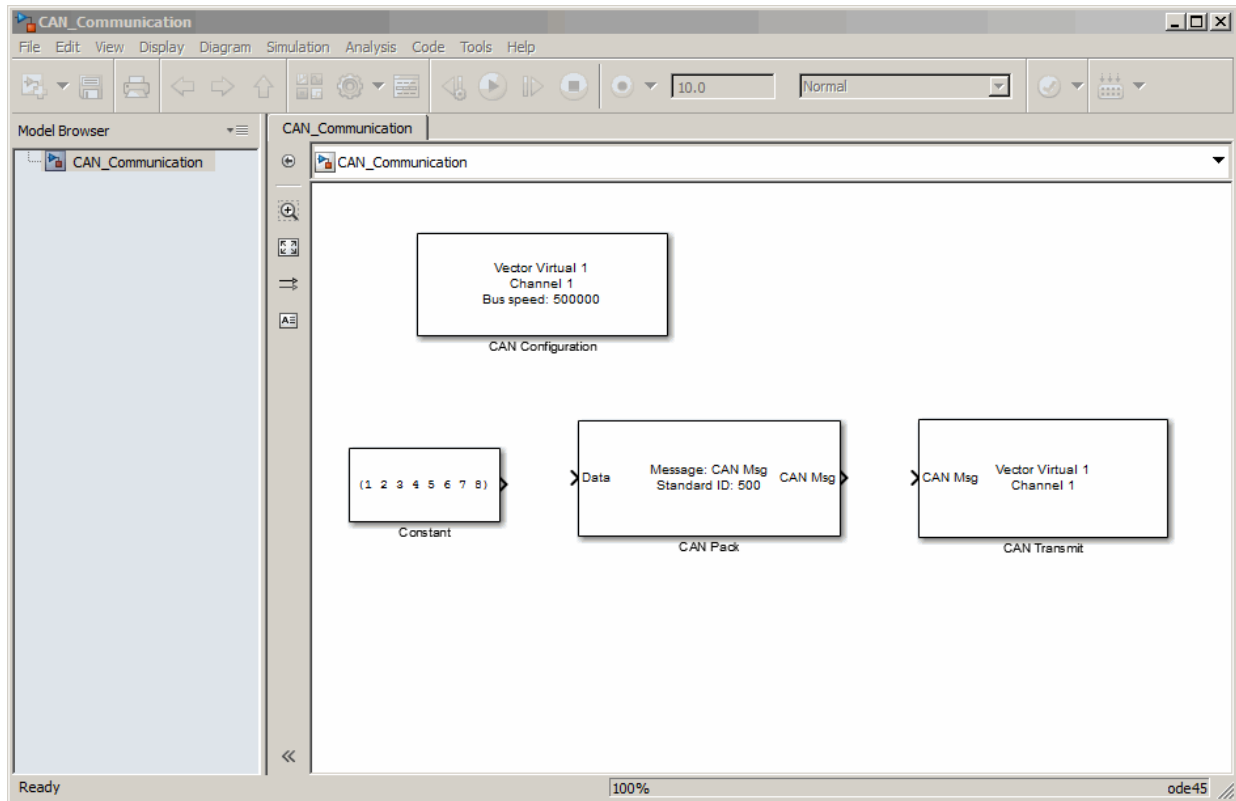
**Configure the Constant Block**

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

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

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

Your model looks like this figure.



## Build a Message Receive Model

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

- “Step 7: Drag Vehicle Network Toolbox Blocks into the Model” on page 10-15
- “Step 8: Drag Other Blocks to Complete the Model” on page 10-16
- “Step 9: Connect the Blocks” on page 10-20
- “Step 10: Specify the Block Parameter Values” on page 10-21

For this portion of the example



- Use a virtual CAN channel to receive messages.
- Use the CAN Configuration block to configure your virtual CAN channels.
- Use the CAN Receive block to receive the message sent by the blocks built in “Build a Message Transmit Model” on page 10-9.
- Use a Function-Call Subsystem block that contains the CAN Unpack block. This function takes the data from the CAN Receive block and uses the parameters of the CAN Unpack block to unpack your message data.
- Use a Scope block to show the transfer of data visually.

Use this section with “Build a Message Transmit Model” on page 10-9 and “Save and Run the Model” on page 10-23 to build your complete model and run the simulation.

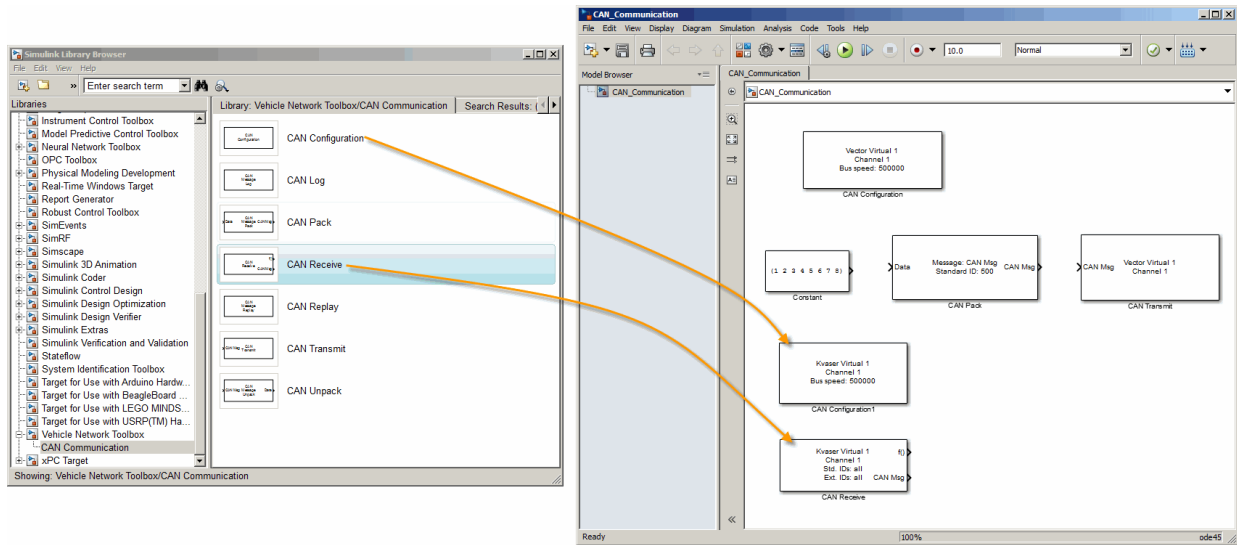
### **Step 7: Drag Vehicle Network Toolbox Blocks into the Model**

For this example, you need one instance each of the CAN Configuration, CAN Receive, and CAN Unpack blocks in your model. However, you add only the CAN Configuration and the CAN Receive blocks here. Add the CAN Unpack block into the Function-Call Subsystem described in “Step 8: Drag Other Blocks to Complete the Model” on page 10-16.

---

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

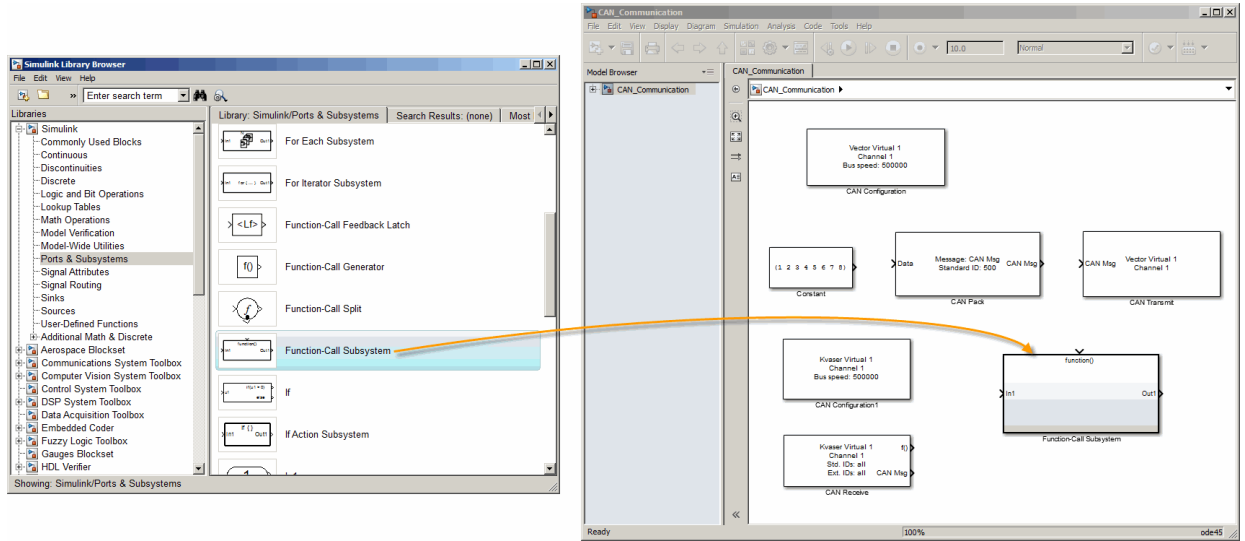
---



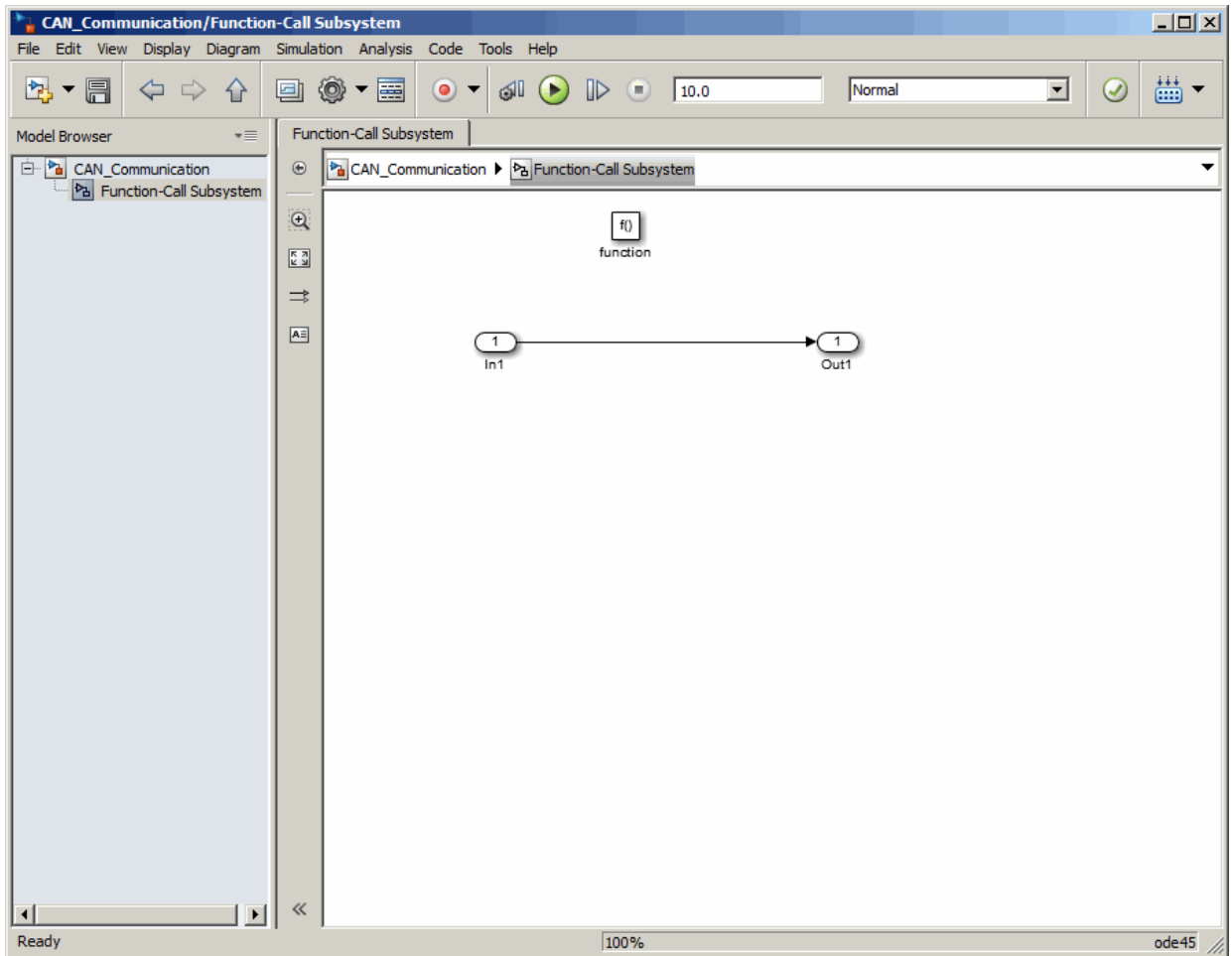
## Step 8: Drag Other Blocks to Complete the Model

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

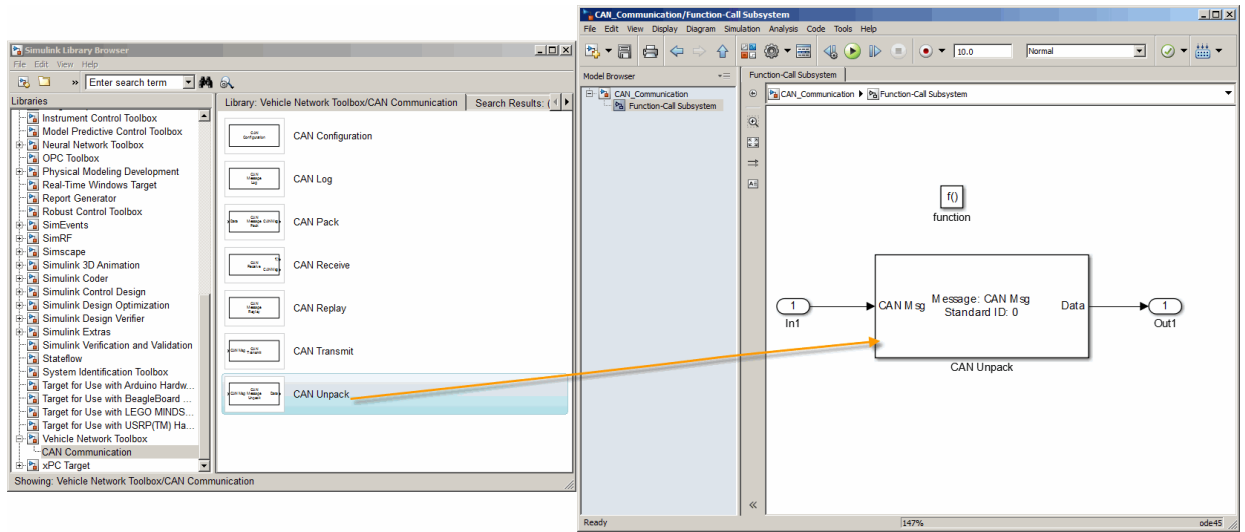
- 1 Drag the Function-Call Subsystem block into the model.



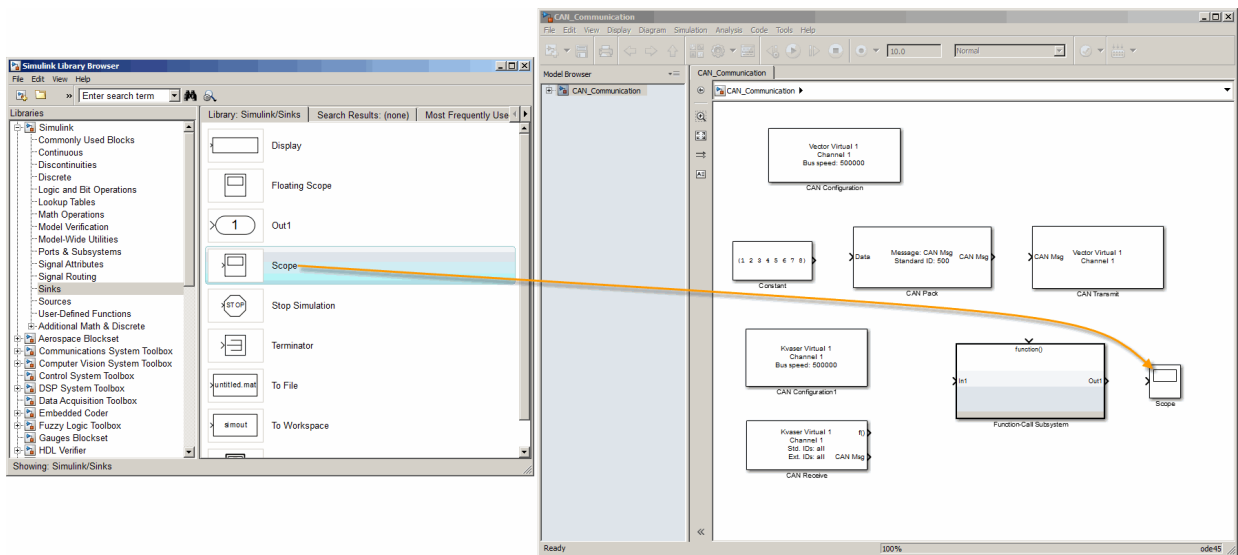
2 Double-click the Function-Call Subsystem block to open the subsystem editor.



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

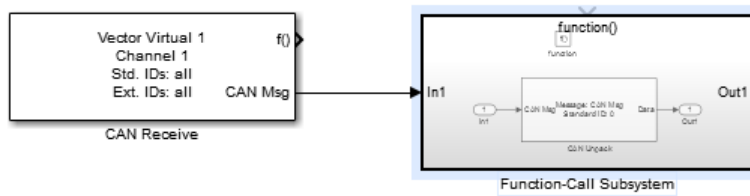


To see the results of the simulation visually, drag the Scope block from the Simulink block library into your model.

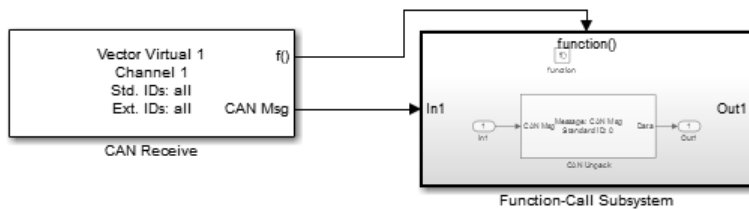


### Step 9: Connect the Blocks

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

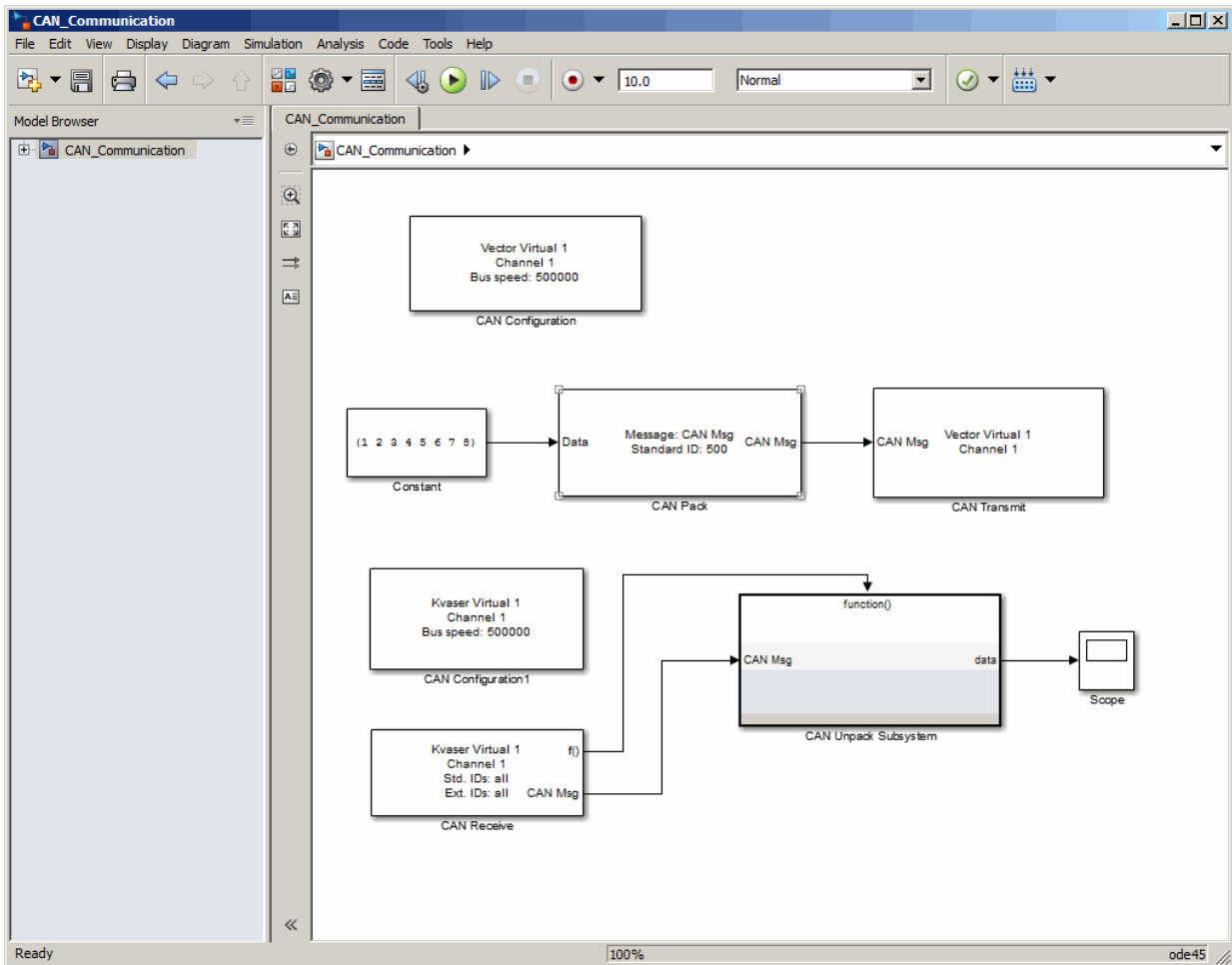


- 2 Open the Function-Call Subsystem block and:
  - Double-click **In1** to rename it to CAN Msg.
  - Double-click **Out1** to rename it to data.
- 3 Rename the Function-Call Subsystem block to CAN Unpack Subsystem.
- 4 Connect the **f()** output port on the CAN Receive block to the **function()** input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



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

Your model looks like this figure.



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

### Step 10: Specify the Block Parameter Values

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

#### Configure the CAN Configuration1 Block

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

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

Click **OK**.

### **Configure the CAN Receive Block**

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

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

Click **OK**.

### **Configure the CAN Unpack Subsystem**

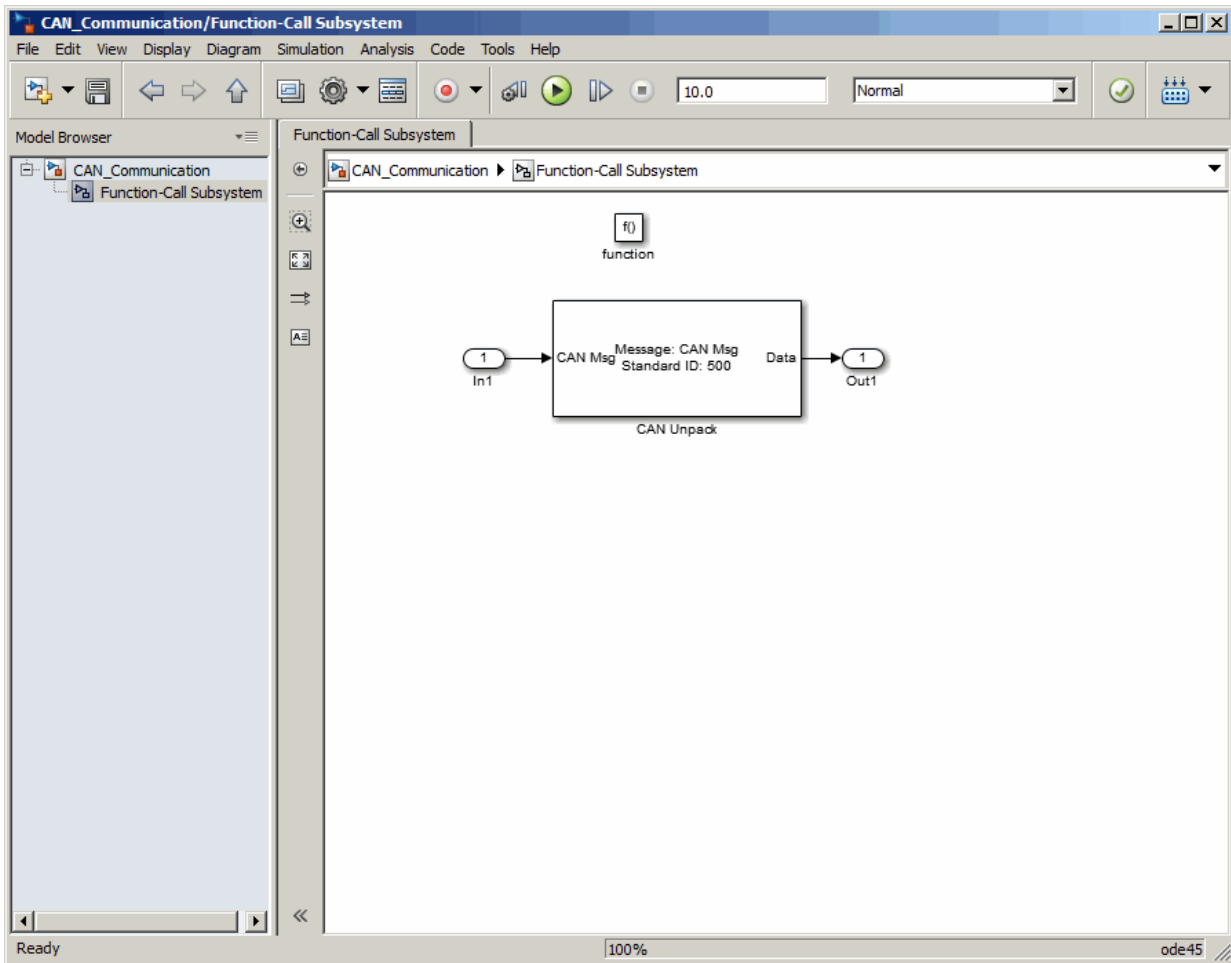
Double-click the CAN Unpack subsystem to open the Function-Call Subsystem editor. In the model, double-click the CAN Unpack block to open its parameters dialog box. Set the:

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

Click **OK**.

Your subsystem looks like this figure.





## Save and Run the Model

This section shows you how to save the models you built, “Build a Message Transmit Model” on page 10-9 and “Build a Message Receive Model” on page 10-14.

- “Step 11: Save the Model” on page 10-24
- “Step 12: Change Configuration Parameters” on page 10-24

- “Step 13: Run the Simulation” on page 10-24
- “Step 14: View the Results” on page 10-25

### **Step 11: Save the Model**

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

### **Step 12: Change Configuration Parameters**

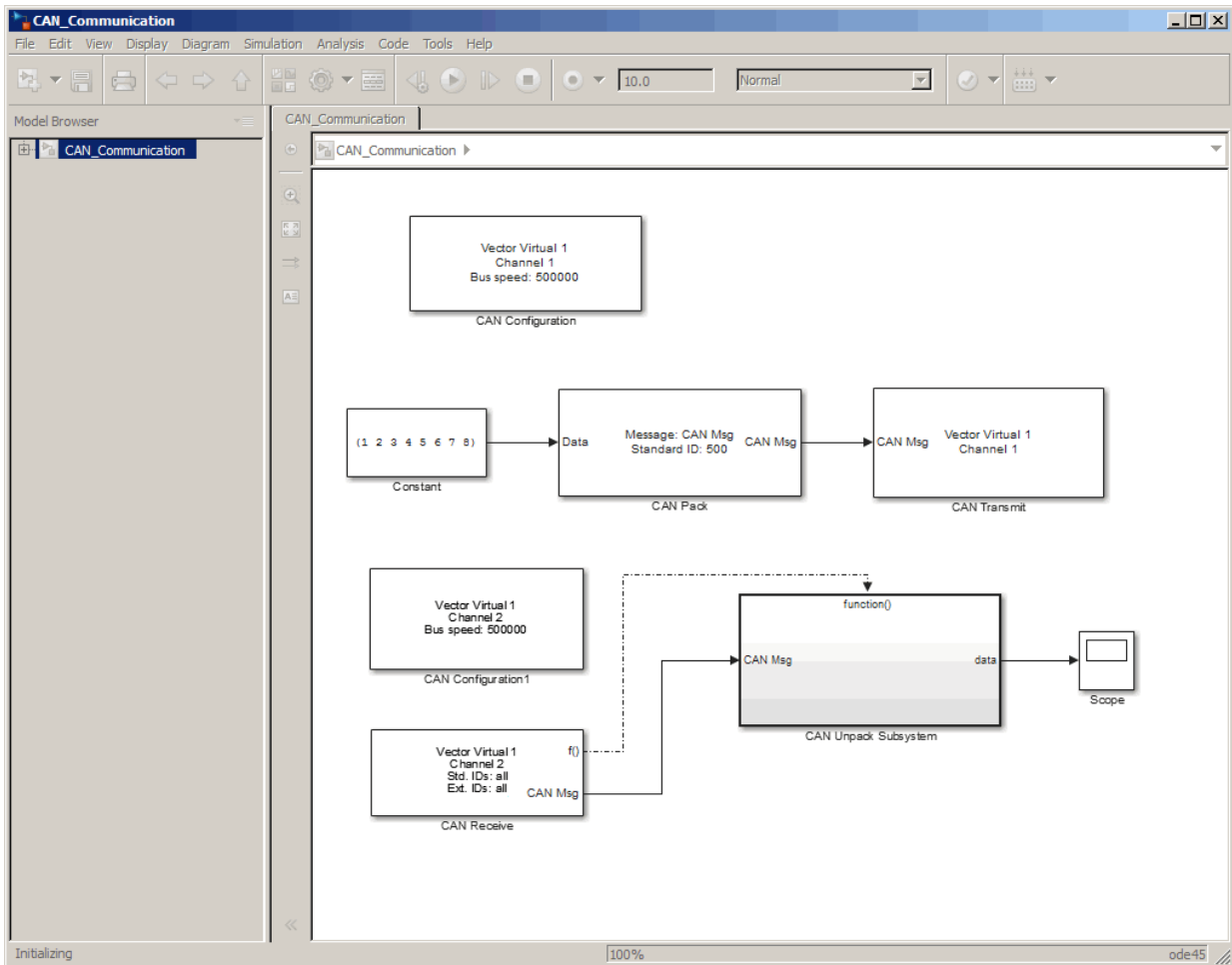
- 1 In your model window, select **Simulation > Model Configuration Parameters**. The Configuration Parameters dialog box opens.
- 2 In the Solver Options section, select:
  - **Fixed-step** from the **Type** list.
  - **Discrete (no continuous states)** from the **Solver** list.

### **Step 13: Run the Simulation**

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

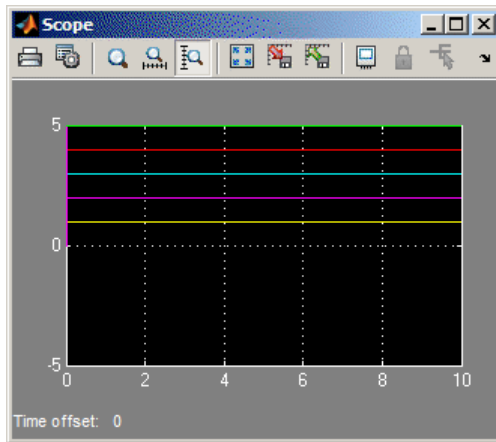
When you run the simulation, the CAN Transmit block gets the message from the CAN Pack block. It then transmits it via Virtual Channel 1. The CAN Receive block on Virtual Channel 2 receives this message and hands it to the CAN Unpack block to unpack the message.

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

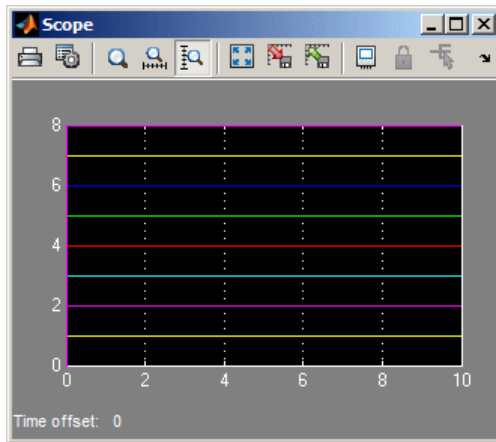


### Step 14: View the Results

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



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



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

## **See Also**

### **More About**

- [“Build and Edit a Model in the Simulink Editor” \(Simulink\)](#)

## Create Custom CAN Blocks

In this section...
“Blocks Using Simulink Buses” on page 10-28
“Blocks Using CAN Message Data Types” on page 10-30

You can create custom `Receive` and `Transmit` blocks to use with hardware currently not supported by Vehicle Network Toolbox. Choose one of the following work flows.

- “Blocks Using Simulink Buses” on page 10-28 (recommended) — Use Simulink bus signals to connect blocks. Create functions and blocks with S-Function Builder and S-Function blocks.
- “Blocks Using CAN Message Data Types” on page 10-30 — Use CAN message data types to share information. Write and compile your own C++ code to define functions, and MATLAB code to create blocks.

### Blocks Using Simulink Buses

To create custom blocks for Vehicle Network Toolbox that use Simulink CAN buses, you can use the S-function builder. For full instructions on building S-functions and blocks this way, see “Build S-Functions Automatically” (Simulink). The following example uses the steps outlined in that topic.

This example shows you how to build two custom blocks for transmitting and receiving CAN messages. These blocks use a Simulink message bus to interact with CAN Pack and CAN Unpack blocks.

- 1 Create a Simulink message bus in the MATLAB workspace for CAN or CAN FD.

```
canMessageBusType
```

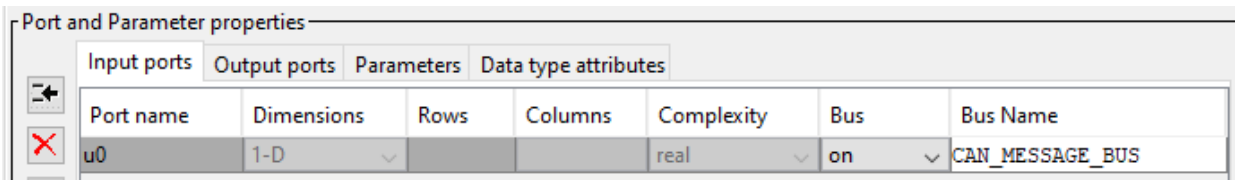
or

```
canFDMessageBusType
```

Each of these functions creates a variable in the workspace named `CAN_MESSAGE_BUS` or `CAN_FD_MESSAGE_BUS`, respectively. You use this variable later for building your S-functions.

- 2 Open a new blank model in Simulink, and add to your model an S-Function Builder block from the block library.

- 3 Double-click the S-Function Builder block to open its dialog box. The first function you build is for transmitting.
- 4 Among the settings in the dialog box, define a function name and specify usage of a Simulink bus.
  - S-function name: CustomCANTransmit
  - Data Properties: Input Ports: Bus: On, Bus Name: CAN\_MESSAGE\_BUS, as shown in the following figure.



For CAN FD, set the bus name to CAN\_FD\_MESSAGE\_BUS.

In your function and block building, use the other tabs in the dialog box to define the code for interaction with your device driver, and remove unnecessary ports.

- 5 Click **Build**. The code files are placed in the current working folder of MATLAB.
- 6 Place a new S-Function Builder block in your model, and repeat the steps to build an S-function named CustomCANReceive. Use the same settings, except for input and output ports. The receive block output port uses the same bus name as the transmit function input.
- 7 Build the receive function, and remove both S-Function Builder blocks from your model. At this point, you can use the files generated by the S-Function Builder as a set of templates, which you can further edit and compile with your own tools. Alternatively, you can use S-Function blocks to run your functions.
- 8 Add two S-Function blocks to your model. Open each block, and set its Model Parameters S-function name field, so you have one each of CustomCANTransmit and CustomCANReceive.

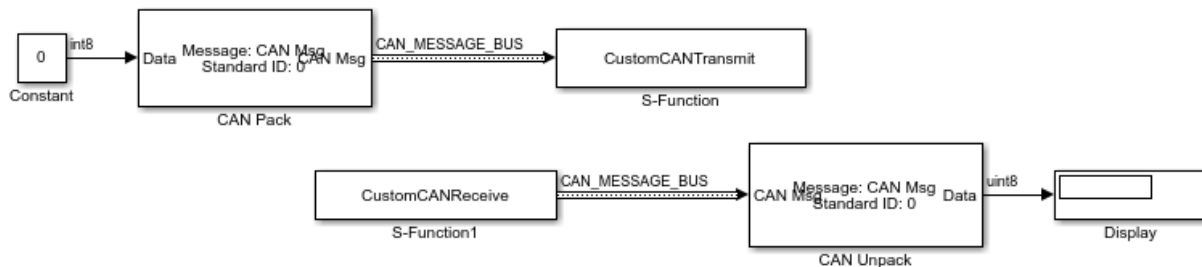
At this point you could create a mask for each block to allow access to parameters for your hardware. This example does not need masks for these blocks.

- 9 Add other necessary blocks to your model, including:
  - CAN Pack or CAN FD Pack

- CAN Unpack or CAN FD Unpack

10 Set the block parameters and connections.

A typical model might look like this. Here a Constant block and a Display block allow verification of connections and model behavior.



## Blocks Using CAN Message Data Types

---

**Note** For ease of design and to take advantage of more Simulink features, it is recommended that you use Simulink buses instead of CAN message data types when possible. See “Blocks Using Simulink Buses” on page 10-28.

---

To create your own blocks for use with other Vehicle Network Toolbox blocks, you can use a custom CAN data type. Register this custom CAN data type in a C++ S-function.

---

**Note** You must use a C++ file type S-function (.cpp) to create custom blocks that use CAN message data types. Using a C-file type S-function (.c) might cause linker errors.

---

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

- 1 Define the IMPORT\_SCANUTIL identifier that imports the required symbols when you compile the S-function:

```
#define IMPORT_SCANUTIL
```

- 2 Include the can\_datatype.h header located in `matlabroot\toolbox\vnt\vntblks\include\candatatype` at the top of the S-function:



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

---

**Note** The header `can_message.h` included by `can_datatype.h` is located in `matlabroot\toolbox\shared\can\src\scanutil\`. See the `can_message.h` file for information on the `CAN_MESSAGE` and `CAN_DATATYPE` structures.

---

- 3 Link your S-function during build to the `scanutil.lib` located in the `matlabroot\toolbox\vnt\vntblks\lib\ARCH` folder. The shared library `scanutil.dll`, is located in the `matlabroot\bin\ARCH`
- 4 Call this function in `mdlInitializeSizes` to initialize the custom CAN data type:

```
mdlInitialize_CAN_datatype(S);
```

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

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

- 6 Do one of the following:

- To create a receive block, set output port data type to `CAN_MESSAGE`:

```
ssSetOutputPortDataType(S,portID,dataTypeID);
```

- To create a transmit block, set the input port to `CAN_MESSAGE`:

```
ssSetInputPortDataType(S,portID,dataTypeID);
```

## See Also

### Functions

`canFDMessageBusType` | `canMessageBusType`

### More About

- “C/C++ S-Function Basics” (Simulink)
- “Build S-Functions Automatically” (Simulink)



# Hardware Limitations

---

This topic describes limitations of using hardware in the Vehicle Network Toolbox based on limitations placed by the hardware vendor:

- “Vector Hardware Limitations” on page 11-2
- “Kvaser Hardware Limitations” on page 11-3
- “File Format Limitations” on page 11-4

## Vector Hardware Limitations

You cannot have more than 64 physical or 32 virtual simultaneous connections using a Vector CAN device.

If you use more than the number of connections Vector allows, you might get an error:

- In MATLAB R2013a and later:  
Unable to query hardware information for the selected CAN channel object.
- In MATLAB R2012b:  
boost thread resource allocation error.
- In MATLAB R2012a and earlier:  
An unhandled error occurred with CAN device.

To work around this issue in Simulink:

- Use only a single Receive block for message reception in Simulink and connect all downstream Unpack blocks to it.
- Use a Mux block to combine CAN messages from Unpack blocks transmitting at the same rate into a single Transmit block.

To work around this issue in MATLAB:

- Try reusing channels you have already created for your application in MATLAB.

## Kvaser Hardware Limitations

You must connect your Kvaser device before starting MATLAB.

The normal workflow with a Kvaser device is to connect the device before starting MATLAB. If you connect a Kvaser device while MATLAB is already running, you might see the following message.

```
Vehicle Network Toolbox has detected a supported Kvaser device.
```

To enable the device, shut down MATLAB. Then with the device connected, restart MATLAB.

## File Format Limitations

### CDFX-File

When using CDFX-files, the following limitations apply:

- SW-AXIS-CONT elements with the category COM\_AXIS, CURVE\_AXIS, or RES\_AXIS must use the SW-INSTANCE-REF element, and the axis must be defined in a separate instance.
- Instances with the category VAL\_BLK, MAP, CUBOID, CUBE\_4, or CUBE\_5 that represent multidimensional arrays must use the VG element to group the physical values.
- The file header must be of the form:

```
<?xml version="1.0" encoding="utf-8"?>  
<MSRSW xmlns="http://www.asam.net/schema/CDF/r2.1"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://www.asam.net/schema/CDF/r2.1 cdf_v2.1.0.sl.xsd">
```

DTD-based headers are not supported.

### BLF-File

Although Vector BLF-files support many networks, Vehicle Network Toolbox support of BLF-files is limited to only CAN and CAN FD.

## See Also

### Functions

blfread | blfwrite | cdfx

# XCP Communications in Simulink

---

- “Vehicle Network Toolbox XCP Simulink Blocks” on page 12-2
- “Open the Vehicle Network Toolbox XCP Block Library” on page 12-3

## Vehicle Network Toolbox XCP Simulink Blocks

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

- **XCP CAN Transport Layer**— Transmit and Receive XCP messages over CAN bus.
- **XCP CAN Configuration** — Configure XCP settings for CAN.
- **XCP CAN Data Acquisition** — Acquire XCP data over CAN.
- **XCP CAN Data Stimulation** — Stimulate XCP data over CAN.
  
- **XCP UDP Configuration** — Configure XCP settings for UDP.
- **XCP UDP Data Acquisition** — Acquire XCP data over UDP.
- **XCP UDP Data Stimulation** — Stimulate XCP data over UDP.

The Vehicle Network Toolbox XCP block library is a tool for handling XCP message traffic on a CAN network or by UDP. You can use blocks from the block library with blocks from other Simulink libraries to create sophisticated models.

To use the Vehicle Network Toolbox XCP 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 “Getting Started with Simulink” (Simulink) to understand its functionality better.



## Open the Vehicle Network Toolbox XCP Block Library

### In this section...

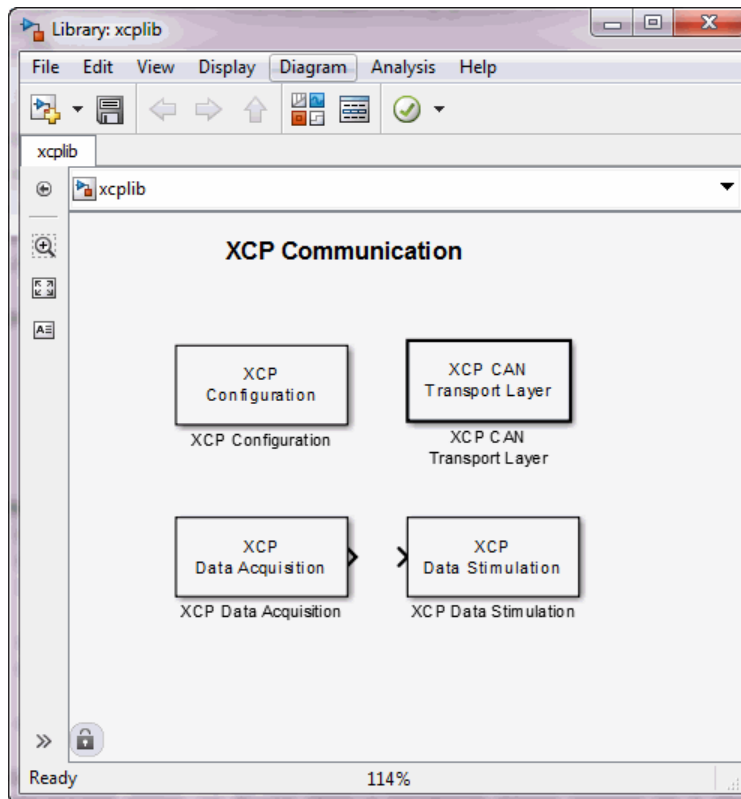
“Using the MATLAB Command Window” on page 12-3

“Using the Simulink Library Browser” on page 12-4

### Using the MATLAB Command Window

To open the Vehicle Network Toolbox block library, enter `xcpLib` in the MATLAB Command window.

The contents of the library open in a separate window.

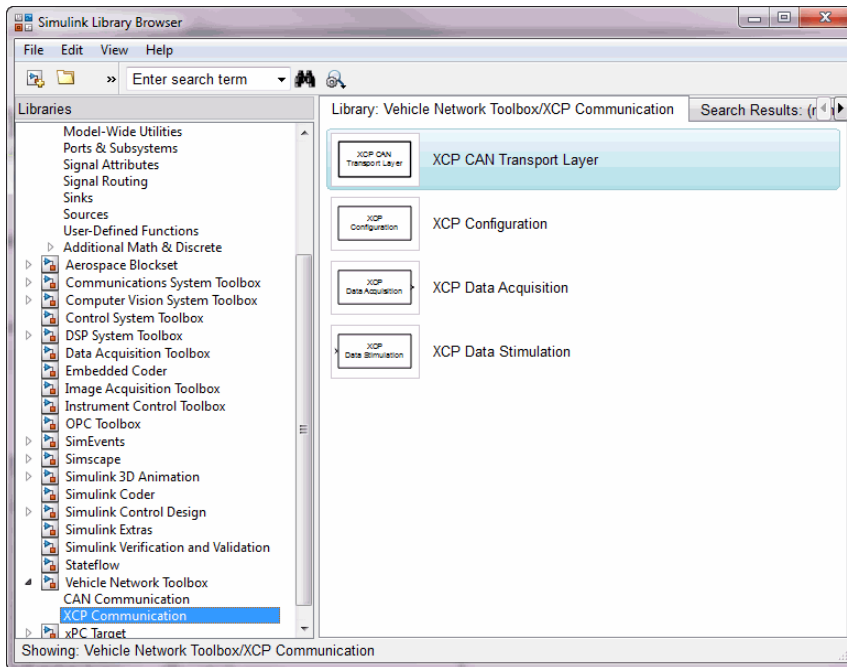


## Using the Simulink Library Browser

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

To start the Simulink Library Browser, enter `simulink` in the MATLAB Command Window.

The **Libraries** pane lists all available block libraries, with the basic Simulink library listed first, followed by other libraries listed alphabetically under it. To open the Vehicle Network Toolbox block library, click its icon and select **CAN Communication** for the CAN blocks.



Simulink loads and displays the blocks in the library.

# Functions — Alphabetical List

---

## attachDatabase

Attach CAN database to messages and remove CAN database from messages

### Syntax

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

### Description

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

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

### Examples

#### Attach CAN Database to Message

Attach Database.dbc to a received CAN message.

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

### Input Arguments

**message** — CAN message for attaching or removing database  
CAN message object

The name of the CAN message that you want to attach the database to or remove the database from, specified as a CAN message object.

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

### **database — Handle of database to attach or remove**

`canDatabase` handle

Handle of database (.dbc file) that you want to attach to the message or remove from the message, specified as a `canDatabase` handle.

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

## **Tips**

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

## **See Also**

### **Functions**

`canDatabase` | `receive`

**Introduced in R2009a**

## attributeInfo

Information about CAN database attributes

### Syntax

```
info = attributeInfo(db, 'Database', AttrName)
info = attributeInfo(db, 'Node', AttrName, NodeName)
info = attributeInfo(db, 'Message', AttrName, MsgName)
info = attributeInfo(db, 'Signal', AttrName, MsgName, SignalName)
```

### Description

`info = attributeInfo(db, 'Database', AttrName)` returns a structure containing information for the specified database attribute.

If no matches are found in the database, `attributeInfo` returns an empty attribute information structure.

`info = attributeInfo(db, 'Node', AttrName, NodeName)` returns a structure containing information for the specified node attribute.

`info = attributeInfo(db, 'Message', AttrName, MsgName)` returns a structure containing information for the specified message attribute.

`info = attributeInfo(db, 'Signal', AttrName, MsgName, SignalName)` returns a structure containing information for the specified signal attribute.

## Examples

### View Database Attribute Information

Create a CAN database object, and view information about its bus type and database version.

```
db = canDatabase('J1939DB.dbc');
db.Attributes

    'BusType'
    'DatabaseVersion'
    'ProtocolType'

info = attributeInfo(db, 'Database', 'BusType')

    Name: 'BusType'
    ObjectType: 'Database'
    DataType: 'Double'
    DefaultValue: 'CAN-test'
    Value: 'CAN'

info = attributeInfo(db, 'Database', 'DatabaseVersion')

    Name: 'DatabaseVersion'
    ObjectType: 'Database'
    DataType: 'Double'
    DefaultValue: '1.0'
    Value: '8.1'
```

## View Node Attribute Information

View node attribute information from CAN database.

```
db = canDatabase('J1939DB.dbc');
db.Nodes

    'AerodynamicControl'
    'Aftertreatment_1_GasIntake'
    'Aftertreatment_1_GasOutlet'

db.NodeInfo(1).Attributes

    'ECU'
    'NmJ1939AAC'
    'NmJ1939Function'

info = attributeInfo(db, 'Node', 'ECU', 'AerodynamicControl')

    Name: 'ECU'
    ObjectType: 'Network node'
```

```
        DataType: 'Double'  
    DefaultValue: 'ECU-1'  
        Value: 'ECU-10'
```

### **View Message Attribute Information**

View message attribute information from CAN database.

```
db = canDatabase('J1939DB.dbc');  
db.Messages  
  
    'A1'  
    'A1DEFI'  
    'A1DEFISI'  
  
db.MessageInfo(1).Attributes  
  
a = db.MessageInfo(1).Attributes  
a =  
    'GenMsgCycleTime'  
    'GenMsgCycleTimeFast'  
    'GenMsgDelayTime'  
    'VFrameFormat'  
  
info = attributeInfo(db, 'Message', 'GenMsgCycleTime', 'A1')  
  
        Name: 'GenMsgCycleTime'  
    ObjectType: 'Message'  
        DataType: 'Undefined'  
    DefaultValue: 0  
        Value: 500
```

### **View Signal Attribute Information from Message**

View message signal attribute information from CAN database.

```
db = canDatabase('J1939DB.dbc');  
s = signalInfo(db, 'A1')  
  
s =  
2x1 struct array with fields:  
    Name
```



```
Comment
StartBit
SignalSize
ByteOrder
Signed
ValueType
Class
Factor
Offset
Minimum
Maximum
Units
ValueTable
Multiplexor
Multiplexed
MultiplexMode
RxNodes
Attributes
AttributeInfo
```

```
s(1).Name
```

```
EngBlowerBypassValvePos
```

```
s(1).Attributes
```

```
    'GenSigEVName'
    'GenSigILSupport'
    'GenSigInactiveValue'
```

```
info = attributeInfo(db, 'Signal', 'GenSigInactiveValue', 'A1', 'EngBlowerBypassValvePos')
```

```
    Name: 'GenSigInactiveValue'
    ObjectType: 'Signal'
    DataType: 'Undefined'
    DefaultValue: 0
    Value: 0
```

## Input Arguments

### db — CAN database

CAN database object

CAN database, specified as a CAN database object.

Example: `db = canDatabase(_____)`

**AttrName — Attribute name**

char vector | string

Attribute name, specified as a character vector or string.

Example: `'BusType'`

Data Types: char | string

**NodeName — Node name**

char vector | string

Node name, specified as a character vector or string.

Example: `'AerodynamicControl'`

Data Types: char | string

**MsgName — Message name**

char vector | string

Message name, specified as a character vector or string.

Example: `'A1'`

Data Types: char | string

**SignalName — Signal name**

char vector | string

Signal name, specified as a character vector or string.

Example: `'EngBlowerBypassValvePos'`

Data Types: char | string

## Output Arguments

**info — Attribute information**

structure

Attribute information, returned as a structure with these fields:

<b>Field</b>	<b>Description</b>
Name	Attribute name
ObjectType	Type of attribute
DataType	Data class of attribute value
DefaultValue	Default value assigned to attribute
Value	Current value of attribute

## See Also

### Functions

[canDatabase](#) | [messageInfo](#) | [nodeInfo](#) | [signalInfo](#) | [valueTableText](#)

### Properties

[AttributeInfo](#) | [Attributes](#)

### Introduced in R2015b

## blfinfo

Get information about Vector BLF file

## Syntax

```
binf = blfinfo(blfFile)
```

## Description

`binf = blfinfo(blfFile)` parses general information about the format and contents of a Vector Binary Logging Format BLF-file and returns the information in the structure `binf`.

## Examples

### View Information about BLF-File

Retrieve and view information about a BLF-file.

```
binf = blfinfo("c:\DataFiles\MultiChannelFile.blf")
```

```
binf =
```

```
    struct with fields:
```

```
        Name: "MultiChannelFile.blf"
        Path: "c:\DataFiles\MultiChannelFile.blf"
    Application: "CANalyzer"
    ApplicationVersion: "10.0.114"
        Objects: 35
        StartTime: 18-Jul-2018 16:47:11.490
        EndTime: 18-Jul-2018 16:47:18.490
        ChannelList: [2x3 table]
```

```
binf.ChannelList
```

```
ans =
```

```
2×3 table
```

ChannelID	Protocol	Objects
1	"CAN FD"	4
2	"CAN"	4

## Input Arguments

### **blfFile** — Path to BLF-file

string | char

Path to BLF-file, specified as a string or character vector. The value can specify a file in the current folder, or a relative or full path name.

Example: "MultipleChannelFile.blf"

Data Types: char | string

## Output Arguments

### **binf** — Information from BLF-file

struct

Information from BLF-file, returned as a structure with the following fields.

```
Name
Path
Application
ApplicationVersion
Objects
StartTime
EndTime
ChannelList
```

## See Also

### Functions

`blfread` | `blfwrite`

**Introduced in R2019a**

# blfread

Read data from Vector BLF-file

## Syntax

```
mdata = blfread(blfFile)
bdata = blfread(blfFile,chanID)
bdata = blfread( ____,Name,Value)
```

## Description

`mdata = blfread(blfFile)` reads all the data from the specified BLF-file and returns a cell array of timetables to the variable `bdata`. The index of each element in the cell array corresponds to the channel number of the data in the file.

`bdata = blfread(blfFile,chanID)` reads message data for the specified channel from the BLF-file and returns a timetable.

`bdata = blfread( ____,Name,Value)` reads message data filtered by parameter options for CAN database and message IDs.

## Examples

### Read Data from BLF-File

Read message data from a BLF-file, applying optional filters.

```
data = blfread("myfile.blf",2)
candb = canDatabase("testdb.dbc");

data = blfread("myfile.blf", "Database", candb)
data = blfread("myfile.blf", "Database", candb, "CANStandardFilter", 1:10)
data = blfread("myfile.blf", "Database", candb, "CANExtendedFilter", 3:7)
data = blfread("myfile.blf", "Database", candb, "CANStandardFilter", 1:10, ...
```

```
data = blfread("myfile.blf", "CANStandardFilter", 1:10, "CANExtendedFilter", 3:7)
```

## Input Arguments

### **blfFile** — Path to BLF-file

string | char

Path to BLF-file, specified as a string or character vector. The value can specify a file in the current folder, or a relative or full path name.

Example: "MultipleChannelFile.blf"

Data Types: string | char

### **chanID** — Channel ID

numeric

Channel ID, specified as a numeric scalar value, for which to read data from the BLF-file. If not specified, all channels are read.

Example: 2

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: "CANStandardFilter", 1:8

### **Database** — CAN database

can.Database

CAN database to use for message decoding, specified as a `can.Database` object.

Example: `candb()`

### **CANStandardFilter** — Message standard IDs

numeric array



Message standard IDs, specified as an array of numeric values identifying which messages to import. Message IDs are general, and apply to both CAN and CAN FD bus types. The value can specify a scalar or an array of either a range or noncontiguous IDs. By default, all standard ID messages are imported.

Example: [1:10 45 100:123]

Data Types: string | char

### **CANExtendedFilter — Message extended IDs**

numeric array

Message extended IDs, specified as an array of numeric values identifying which messages to import. Message IDs are general, and apply to both CAN and CAN FD bus types. The value can specify a scalar or an array of either a range or noncontiguous IDs. By default, all extended ID messages are imported.

Example: [1 8:10 1001:1080]

Data Types: string | char

## **Output Arguments**

### **mdata — Message data from BLF-file**

cell array of timetables | timetable

Message data from BLF-file, returned as a cell array of timetables. If you specify a single channel to read, this returns a timetable.

## **See Also**

### **Functions**

blfinfo | blfwrite | canDatabase

**Introduced in R2019a**

## blfwrite

Write data to Vector BLF-file

### Syntax

```
blfwrite(blfFile,data,chanID,prot)
```

### Description

`blfwrite(blfFile,data,chanID,prot)` writes the specified timetables in data to the specified BLF-file. The function allows writing only to new files, so you cannot overwrite existing files or data.

### Examples

#### Write Data to a BLF-File

Write timetables of data to specified channels.

Write one data set to a single channel.

```
blfwrite("newfile.blf",data,1,"CAN")
```

Write two data sets to the same channel.

```
blfwrite("newfile.blf",{data1,data2},[1,1],["CAN FD","CAN FD"])
```

Write two data sets to separate channels with different protocols.

```
blfwrite("newfile.blf",{data1,data2},[1,2],["CAN","CAN FD"])
```

## Input Arguments

### **blfFile** — Path to BLF-file

string | char

Path to BLF-file to write, specified as a string or character vector. The value can specify a file in the current folder, or a relative or full path name.

Example: "MultipleChannelFile.blf"

Data Types: string | char

### **data** — Data to write to BLF-file

timetable

Data to write to BLF-file, specified as a timetable or cell array of timetables. You can write multiple tables for the same channel if the protocol is the same.

Data Types: timetable

### **chanID** — Channel IDs

numeric

Channel IDs, specified as a numeric scalar or array value, identifying the channels on which the data is written.

Example: [1,2,4]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **prot** — Message protocol

"CAN" "CAN FD"

Message protocol, specified as "CAN", "CAN FD". When writing multiple sets of data, specify protocol as an array of strings corresponding to the data sets being written.

Example: ["CAN", "CAN FD", "CAN"]

Data Types: char | string

## See Also

### Functions

`blfinfo` | `blfread`

**Introduced in R2019a**

# canChannel

Construct CAN channel connected to specified device

## Syntax

```
canch = canChannel(vendor,device,devicechannelindex)
canch = canChannel(vendor,device)
canch = canChannel( ____, 'ProtocolMode', 'CAN FD')
```

## Description

`canch = canChannel(vendor,device,devicechannelindex)` returns a CAN channel connected to a device from a specified vendor.

For Vector products, `device` is a character vector that combines 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 `canch = canChannel(vendor,device)` for National Instruments and PEAK-System devices.

For National Instruments, `vendor` is the character vector 'NI', and the `devicenumber` is interface number defined in the NI Measurement & Automation Explorer.

For PEAK-System devices `vendor` is the character vector 'PEAK-System', and the `devicenumber` is device number defined for the channel.

Check the CAN Device Constructor in the `canHWInfo` display for channel construction.

`canch = canChannel( ____, 'ProtocolMode', 'CAN FD')` returns a channel connected to a device supporting CAN FD. The default `ProtocolMode` setting is 'CAN', indicating standard CAN support. A channel configured for 'CAN' cannot transmit or receive CAN FD messages.

## Examples

### Create CAN Channels for Various Vendors

Create CAN channels for each of several vendors.

```
canch1 = canChannel('Vector', 'CANCaseXL 1',1);
canch2 = canChannel('Vector', 'Virtual 1',2);
canch3 = canChannel('NI', 'CAN1');
canch4 = canChannel('PEAK-System', 'PCAN_USBBUS1');
canch5 = canChannel('MathWorks', 'Virtual 1',2)
```

```
canch5 =
```

```
Channel with properties:
```

```
Device Information
```

```
DeviceVendor: 'MathWorks'
Device: 'Virtual 1'
DeviceChannelIndex: 2
DeviceSerialNumber: 0
ProtocolMode: 'CAN'
```

```
Status Information
```

```
Running: 0
MessagesAvailable: 0
MessagesReceived: 0
MessagesTransmitted: 0
InitializationAccess: 1
InitialTimestamp: [0x0 datetime]
FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

```
Channel Information
```

```
BusStatus: 'N/A'
SilentMode: 0
TransceiverName: 'N/A'
TransceiverState: 'N/A'
ReceiveErrorCount: 0
TransmitErrorCount: 0
BusSpeed: 500000
SJW: []
TSEG1: []
TSEG2: []
NumOfSamples: []
```

```
Other Information
```

```
Database: []
UserData: []
```

## Create CAN FD Channel

Create a CAN FD channel on a MathWorks virtual device.

```
canch6 = canChannel('MathWorks', 'Virtual 1', 2, 'ProtocolMode', 'CAN FD')
```

```
canch6 =
```

```
Channel with properties:
  Device Information
    DeviceVendor: 'MathWorks'
    Device: 'Virtual 1'
    DeviceChannelIndex: 2
    DeviceSerialNumber: 0
    ProtocolMode: 'CAN FD'
  Status Information
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
  Bit Timing Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    ArbitrationBusSpeed: []
    DataBusSpeed: []
  Other Information
    Database: []
    UserData: []
```

## Input Arguments

### vendor — CAN device vendor

```
'MathWorks' | 'Kvaser' | 'NI' | 'PEAK-System' | 'Vector'
```

CAN device vendor, specified as 'MathWorks', 'Kvaser', 'NI', 'PEAK-System', or 'Vector'.

Example: 'MathWorks'

Data Types: char | string

### **device — CAN to connect channel to**

character vector | string

CAN device to connect channel to, specified as a character vector or string. Valid values depend on the specified vendor.

Example: 'Virtual 1'

Data Types: char | string

### **devicechannelindex — CAN device channel port or index**

numeric value

CAN device channel port or index, specified as a numeric value.

Example: 1

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## **Output Arguments**

### **canch — CAN device channel**

CAN channel object

CAN device channel returned as a CAN channel object, with the following properties.

CAN Channel Properties:



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

#### CAN Device Properties:

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

#### Bit Timing Properties:

BusSpeed	Bit rate of bus
NumOfSamples	Display number of samples available to channel
SJW	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

## Tips

- Use `canChannelList` to obtain a list of available devices.
- You cannot have more than one `canChannel` configured on the same NI-XNET or PEAK-System device channel.

- 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

### Functions

`canChannelList`

**Introduced in R2009a**

# CAN.ChannellInfo class

**Package:** CAN

Display device channel information

---

**Note** `can.ChannelInfo` will be removed in a future release. Use `canChannelList` instead.

---

## Description

`vendor.ChannelInfo(index)` displays channel information for the device vendor with the specified index. Obtain the vendor information using `CAN.VendorInfo`.

## Input Arguments

**index — Device channel index**  
numeric value

Device channel index specified as a numeric value.

## Properties

### Device

Name of the device.

### DeviceChannelIndex

Index number of the specified device channel.

### DeviceSerialNumber

Serial number of the specified device.

## ObjectConstructor

Information on how to construct a CAN channel using this device.

## Examples

### Examine Kvaser Device Channel Information

Get information on installed CAN devices.

```
info = canHWInfo
```

```
info =
```

```
CAN Devices Detected
```

Vendor	Device	Channel	Serial Number	Constructor
Kvaser	Virtual 1	1	0	canChannel('Kvaser', 'Virtual 1', 1)
Kvaser	Virtual 1	2	0	canChannel('Kvaser', 'Virtual 1', 2)
Vector	Virtual 1	1	0	canChannel('Vector', 'Virtual 1', 1)
Vector	Virtual 1	2	0	canChannel('Vector', 'Virtual 1', 2)

Save the Kvaser device information in an object.

```
vendor = info.VendorInfo(1);
```

Get information on the first channel of the specified device.

```
vendor.ChannelInfo(1)
```

```
ans =
```

```
ChannelInfo with properties:
```

```
Device: 'Virtual 1'  
DeviceChannelIndex: 1
```

```
DeviceSerialNumber: 0  
ObjectConstructor: 'canChannel('Kvaser', 'Virtual 1', 1)'
```

## See Also

### Functions

`can.VendorInfo` | `canHWInfo`

## canChannelList

Information on available CAN devices

### Syntax

```
chans = canChannelList
```

### Description

`chans = canChannelList` returns a table of information about available CAN devices.

### Examples

#### View Available CAN Devices

View available CAN devices and programmatically read a device's supported protocol modes.

```
chans = canChannelList
```

```
chans =
```

```
4×6 table
```

Vendor	Device	Channel	DeviceModel	ProtocolMode	SerialNumber
"MathWorks"	"Virtual 1"	1	"Virtual"	"CAN, CAN FD"	"0"
"MathWorks"	"Virtual 1"	2	"Virtual"	"CAN, CAN FD"	"0"
"Vector"	"Virtual 1"	1	"Virtual"	"CAN"	"0"
"Vector"	"Virtual 1"	2	"Virtual"	"CAN"	"0"

```
pm = chans{3,5}
```

```
pm =
```

```
"CAN"
```

```
pm = chans{3, 'ProtocolMode'}
```

pm =

"CAN"

## Output Arguments

**chans** — Information on available CAN devices

table

Information on available CAN devices, returned as a table. To access specific elements, you can index into the table.

## See Also

**Functions**

canChannel

**Introduced in R2017b**

## canDatabase

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 a file name, a full path, or a relative path. MATLAB looks for `dbfile.dbc` on the MATLAB path. Vehicle Network Toolbox supports Vector CAN database (`.dbc`) files.

### Examples

#### Create CAN Database Object

Create objects for example database files.

```
candb = canDatabase([(matlabroot) '\examples\vnt\demoVNT_CANdbFiles.dbc'])
```

```
candb =
```

Database with properties:

```
    Name: 'demoVNT_CANdbFiles'  
    Path: 'F:\matlab\examples\vnt\demoVNT_CANdbFiles.dbc'  
    Nodes: {}  
    NodeInfo: [0×0 struct]  
    Messages: {5×1 cell}  
    MessageInfo: [5×1 struct]  
    Attributes: {}  
    AttributeInfo: [0×0 struct]  
    UserData: []
```



```
candb = canDatabase([(matlabroot) '\examples\vnt\J1939.dbc'])
candb =
    Database with properties:
        Name: 'J1939'
        Path: 'F:\matlab\examples\vnt\J1939.dbc'
        Nodes: {2×1 cell}
        NodeInfo: [2×1 struct]
        Messages: {2×1 cell}
        MessageInfo: [2×1 struct]
        Attributes: {3×1 cell}
        AttributeInfo: [3×1 struct]
        UserData: []
```

## Input Arguments

### **dbfile.dbc** — Database file name

char vector | string

Database file name, specified as a character vector or string.. You can specify just the name or the full path of the database file.

Example: 'J1939.dbc'

Data Types: char | string

## Output Arguments

### **candb** — CAN database

database object

CAN database, returned as a database object with the following properties:

AttributeInfo	Information on CAN database attributes
Attributes	Attribute names from CAN database
MessageInfo	Information on CAN database messages
Messages	Message names from CAN database
Name (Database)	CAN database name
NodeInfo	Information on CAN database nodes
Nodes	Node names from CAN database
Path	CAN database folder path
SignalInfo	Information on CAN database message signals
UserData	Enter custom data

## See Also

### Functions

canMessage

**Introduced in R2009a**

# canFDChannel

Construct CAN FD channel connected to specified device

## Syntax

```
canch = canFDChannel(vendor,device,devicechannelindex)
canch = canFDChannel(vendor,device)
```

## Description

`canch = canFDChannel(vendor,device,devicechannelindex)` returns a CAN FD channel connected to a device from a specified vendor.

For Vector and Kvaser products, `device` combines the device type and a device index, such as 'CANCaseXL 1'. For example, if there are two Vector devices, `device` can be 'VN1610 1' or 'VN1610 2'.

`canch = canFDChannel(vendor,device)` returns a CAN FD channel connected to a National Instruments or PEAK-System device.

For National Instruments, `vendor` is the character vector 'NI', and the `devicenum` is the interface number defined in the NI Measurement & Automation Explorer.

For PEAK-System devices `vendor` is the character vector 'PEAK-System', and `devicenum` is the device number defined for the channel.

## Examples

### Create CAN FD Channels for Various Vendors

Create CAN FD channels for each of several vendors.

```
ch1 = canFDChannel('Vector','VN1610 1',1);
ch2 = canFDChannel('Kvaser','USBcan Pro 1',1);
```

```
ch3 = canFDChannel('NI', 'CAN0');
ch4 = canFDChannel('PEAK-System', 'PCAN_USBBUS1');
ch5 = canFDChannel('MathWorks', 'Virtual 1', 1)

ch5 =
Channel with properties:

Device Information
    DeviceVendor: 'MathWorks'
        Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0
        ProtocolMode: 'CAN FD'

Status Information
    Running: 0
    MessagesAvailable: 0
    MessagesReceived: 0
    MessagesTransmitted: 0
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
    FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'

Bit Timing Information
    BusStatus: 'N/A'
    SilentMode: 0
    TransceiverName: 'N/A'
    TransceiverState: 'N/A'
    ReceiveErrorCount: 0
    TransmitErrorCount: 0
    ArbitrationBusSpeed: []
    DataBusSpeed: []

Other Information
    Database: []
    UserData: []
```

## Input Arguments

### vendor — CAN device vendor

'MathWorks' | 'Kvaser' | 'NI' | 'PEAK-System' | 'Vector'

CAN device vendor, specified as 'MathWorks', 'Kvaser', 'NI', 'PEAK-System', or 'Vector'.

Example: 'MathWorks'

Data Types: char | string

### device — CAN FD device to connect channel to

character vector | string

CAN FD device to connect channel to, specified as a character vector or string. Valid values depend on the specified vendor.

Example: 'Virtual 1'

Data Types: char | string

**devicechannelindex** — CAN FD device channel port or index  
numeric value

CAN FD device channel port or index, specified as a numeric value.

Example: 1

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

**canch** — CAN FD device channel  
CAN FD channel object

CAN FD device channel returned as a CAN channel object, with the following properties.

CAN Channel Properties:

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

CAN Device Properties:

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

**Bit Timing Properties:**

BusSpeed	Bit rate of bus
NumOfSamples	Display number of samples available to channel
SJW	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

## Tips

- Use `canFDChannelList` to obtain a list of available device channels.
- You cannot have more than one CAN FD channel configured on the same NI-XNET or PEAK-System device channel.
- 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 channel object.
- You cannot create arrays of channel objects. Each object you create must exist as its own individual variable.

## See Also

**Functions**`canFDChannelList`**Introduced in R2018b**

# canFDChannelList

Information on available CAN FD device channels

## Syntax

```
chans = canFDChannelList
```

## Description

`chans = canFDChannelList` returns a table of information about available CAN FD devices.

## Examples

### View Available CAN FD Device Channels

View available CAN FD device channels and programmatically read supported protocol modes.

```
chans = canFDChannelList
```

```
chans =
```

```
2x6 table
```

Vendor	Device	Channel	DeviceModel	ProtocolMode	SerialNumber
"MathWorks"	"Virtual 1"	1	"Virtual"	"CAN, CAN FD"	"0"
"MathWorks"	"Virtual 1"	2	"Virtual"	"CAN, CAN FD"	"0"

```
pm = chans{2,5}
```

```
pm =
```

```
"CAN, CAN FD"
```

```
pm = chans{2, 'ProtocolMode'}
```

```
pm =  
    "CAN, CAN FD"
```

## Output Arguments

### **chans — Information on available CAN FD devices**

table

Information on available CAN FD device channels, returned as a table. To access specific elements, you can index into the table.

## See Also

### **Functions**

canFDChannel

**Introduced in R2018b**



# canFDMessage

Build CAN FD message based on user-specified structure

## Syntax

```
message = canFDMessage(id,extended,datalength)
message = canFDMessage(candb,messageName)
```

## Description

`message = canFDMessage(id,extended,datalength)` creates a CAN FD message object from the raw message information.

`message = canFDMessage(candb,messageName)` creates a message using the message definition in the specified database. Because `ProtocolMode` is defined in the message database, you cannot specify it as an argument to `canFDMessage` when using a database.

## Examples

### Create a CAN FD Message with Database Definitions

Create a CAN FD message using the definitions of a CAN database.

```
candb = canDatabase(string([(matlabroot) '\examples\vnt\CANFDExample.dbc']));
message3 = canFDMessage(candb,'CANFDMessage')
```

```
message3 =
```

```
Message with properties:
```

```
Message Identification
  ProtocolMode: 'CAN FD'
           ID: 1
           Extended: 0
```

```
        Name: 'CANFDMessage'

Data Details
  Timestamp: 0
    Data: [1x48 uint8]
  Signals: []
  Length: 48
  DLC: 14

Protocol Flags
  BRS: 1
  ESI: 0
  Error: 0

Other Information
  Database: [1x1 can.Database]
  UserData: []
```

### Create a CAN FD Message

Create a CAN FD message with a standard ID format.

```
message2 = canFDMessage(1000, false, 64)
```

```
message2 =
```

```
Message with properties:
```

```
Message Identification
  ProtocolMode: 'CAN FD'
    ID: 1000
  Extended: 0
  Name: ''

Data Details
  Timestamp: 0
    Data: [1x64 uint8]
  Signals: []
  Length: 64
  DLC: 15

Protocol Flags
```

```
BRS: 0  
ESI: 0  
Error: 0
```

```
Other Information  
Database: []  
UserData: []
```

## Input Arguments

### **id** — ID of message

numeric value

ID of the message, specified as a numeric value. If this ID used an extended format, set the `extended` argument `true`.

Example: 2500

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **extended** — Specify if message ID is extended

`true` | `false`

Specifies whether the message ID is of standard or extended type, specified as `true` or `false`. The logical value `true` indicates that the ID is of extended type (29 bits), `false` indicates standard type (11 bits).

Example: `true`

Data Types: `logical`

### **dataLength** — Length of message data

integer value 0 to 64

The length of the message data, specified as an integer value of 0 through 64, inclusive.

Example: 64

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **canDb** — CAN database

CAN database object

CAN database, specified as a database object. The database contains the message definition.

Example: `candb = canDatabase('CANDatabase.dbc')`

**messagename — Name of message**

char vector | string

The name of the message definition in the database, specified as a character vector or string.

Example: `'VehicleDataMulti'`

Data Types: `char` | `string`

## Output Arguments

**message — CAN FD message**

CAN message object

CAN FD message, returned as a CAN message object, with the following properties:

Property	Purpose
BRS	CAN FD bit rate switch, as true or false
Data	Data of CAN message or J1939 parameter group
Database	CAN database information
DLC	Data length code value
Error	CAN message error frame, as true or false
ESI	CAN FD error state indicator, as true or false
Extended	True or false indication of extended CAN Identifier type
ID	Identifier for CAN message
Length	Message length in bytes
Name	CAN message name

<b>Property</b>	<b>Purpose</b>
ProtocolMode	Protocol mode defined as CAN or CAN FD
Remote	Specify if CAN message is remote frame
Signals	Physical signals defined in CAN message or J1939 parameter group
Timestamp	Message received timestamp
UserData	Custom data

## See Also

### Functions

attachDatabase | canDatabase | extractAll | extractRecent | extractTime | pack | unpack

### Introduced in R2018b

## canFDMessageBusType

Create Simulink CAN FD message bus

### Syntax

```
canFDMessageBusType  
canFDMessageBusType(modelName)
```

### Description

`canFDMessageBusType` creates a Simulink CAN FD message bus object named `CAN_FD_MESSAGE_BUS` in the base workspace. The values of the object properties are read-only, but useful for showing the structure of its data.

`canFDMessageBusType(modelName)` creates a Simulink CAN FD message bus object named `CAN_FD_MESSAGE_BUS` in the data dictionary associated with the specified model, `modelName`.

### Examples

#### Create CAN FD Message Bus Object

Create and view the properties of a Simulink CAN FD message bus object.

```
canFDMessageBusType  
CAN_FD_MESSAGE_BUS  
  
CAN_FD_MESSAGE_BUS =  
  
    Bus with properties:  
  
    Description: ''  
    DataScope: 'Auto'  
    HeaderFile: ''
```

```

Alignment: -1
Elements: [12x1 Simulink.BusElement]

```

View the Elements properties of the bus.

`CAN_FD_MESSAGE_BUS.Elements`

ans =

12x1 BusElement array with properties:

```

Min
Max
DimensionsMode
SampleTime
Description
Unit
Name
DataType
Complexity
Dimensions

```

## Input Arguments

**modelName** — Name of model

char vector | string

Name of model, specified as a character vector or string, whose data dictionary is updated with the bus object.

Example: 'CANFDModel'

Data Types: char | string

## See Also

### Blocks

CAN FD Pack | CAN FD Receive | CAN FD Replay

### Topics

“Create Custom CAN Blocks” on page 10-28

“Composite Signals” (Simulink)

**Introduced in R2018a**



# canFDMessageReplayBlockStruct

Convert CAN FD messages for use as CAN Replay block output

## Syntax

```
msgstructofarrays = canFDMessageReplayBlockStruct(msgs)
```

## Description

`msgstructofarrays = canFDMessageReplayBlockStruct(msgs)` formats the specified CAN FD messages for use with the CAN FD Replay block. The CAN FD Replay block requires a specific format for CAN FD messages, defined by a structure of arrays containing the ID, Extended, Data, and other message elements.

Use this function to assign the formatted message structure to a variable. Then save that variable to a MAT-file. The CAN FD Replay block mask allows selection of this MAT file and the variable within it, to replay the messages in a Simulink model.

## Examples

### Create Message Structure for CAN FD Replay Block

Create a message structure for the CAN FD Replay block, and save it to a MAT-file.

```
canMsgs = canFDMessageReplayBlockStruct(messages);  
save('ReplayBlockMessages.mat', 'canMsgs');
```

## Input Arguments

### **msgs** — Original CAN FD messages

CAN message objects | CAN FD message timetable

Original CAN FD messages, specified as a CAN FD message timetable or an array of CAN message objects.

## Output Arguments

### **msgstructofarrays — Formatted CAN FD messages**

struct

Formatted CAN FD messages, returned as structure of arrays containing the ID, Extended, Data, and other elements of the messages.

## See Also

### **Functions**

canFDMessageTimetable | save

### **Blocks**

CAN Replay

**Introduced in R2018b**

# canFDMessageTimetable

Convert CAN or CAN FD messages into timetable

## Syntax

```
msgtimetable = canFDMessageTimetable(msg)
msgtimetable = canFDMessageTimetable(msg,database)
```

## Description

`msgtimetable = canFDMessageTimetable(msg)` creates a CAN FD message timetable from an existing CAN FD message timetable, an array of CAN message objects, or a CAN FD message structure from the CAN FD Log block. The output message timetable contains the raw message information (ID, Extended, Data, etc.) from the messages. If CAN message objects are input which contain decoded information, that decoding is retained in the CAN FD message timetable.

`msgtimetable = canFDMessageTimetable(msg,database)` uses the database to decode the message names and signals for the timetable along with the raw message information. Specify multiple databases in an array to decode message names and signals in the timetable within a single call.

The input `msg` can also be a timetable of data created by using `read` on an `mdfDatastore` object. In this case, the function converts the timetable of ASAM standard logging format data to a Vehicle Network Toolbox CAN FD message timetable.

## Examples

### Convert Log Block Output to Timetable

Convert log block output to a CAN FD message timetable.

```
load LogBlockOutput.mat;
db = canDatabase('myDatabase.dbc');
msgTimetable = canFDMessageTimetable(canMsgs,db);
```

### Convert Message Objects to CAN FD Message Timetable

Convert an array of CAN message objects to a CAN FD message timetable.

```
msgTimetable = canFDMessageTimetable(canMsgs);
```

### Decode Message Timetable with Database

Decode an existing CAN FD message timetable with a database.

```
db = canDatabase('myDatabase.dbc')
msgTimetable = canFDMessageTimetable(msgTimetable,db)
```

The result is returned to the original timetable variable.

### Convert an ASAM MDF Message Timetable

Convert an existing ASAM format message timetable, and decode using a database.

```
m = mdf('CANandCANFD.MF4');
db = canDatabase('CustomerDatabase.dbc');
mdfData = read(m);
msgTimetable = canFDMessageTimetable(mdfData{2},db);
```

Compare the two timetables.

```
mdfData{2}(1:4,1:6)
```

```
ans =
```

```
4×6 timetable
```

Time	CAN_DataFrame_BusChannel	CAN_DataFrame_FlagsEx	CAN_DataFrame_Dir	CAN_DataFrame_SingleWire
0.30022 sec	1	2.1095e+06	1	0
0.45025 sec	1	2.0972e+06	1	0
0.60022 sec	1	2.1095e+06	1	0
0.75013 sec	1	2.1095e+06	1	0

```
msgTimetable(1:4,1:8)
```

```
ans =
```

```
4x8 timetable
```

Time	ID	Extended	Name	ProtocolMode	Data	Length	DLC	Signals
0.30022 sec	768	false	''	'CAN FD'	[1x64 uint8]	64	15	[0x0 struct]
0.45025 sec	1104	false	''	'CAN'	[1x8 uint8]	8	8	[0x0 struct]
0.60022 sec	768	false	''	'CAN FD'	[1x64 uint8]	64	15	[0x0 struct]
0.75013 sec	1872	false	''	'CAN FD'	[1x24 uint8]	24	12	[0x0 struct]

## Input Arguments

### msg — Raw CAN messages

CAN FD message timetable, array, or structure

Raw CAN messages, specified as a CAN FD message timetable, an array of CAN message objects, a CAN message structure from the CAN log block, or an `asam.MDF` object..

Example: `canFDMessage()`

### database — CAN database

database object

CAN database, specified as a database object.

Example: `database = canDatabase('CANDatabase.dbc')`

## Output Arguments

### msgtimetable — CAN FD message timetable

timetable

CAN FD messages returned as a timetable.

## See Also

### Functions

`canDatabase` | `canSignalTimetable` | `mdfDatastore` | `read` (`MDFDatastore`)

**Introduced in R2018b**

# canHWInfo

(To be removed) Information on available CAN devices

---

**Note** canHWInfo will be removed in a future release. Use `canChannelList` instead.

---

## Syntax

```
hw = canHWInfo
```

## Description

`hw = canHWInfo` returns information about CAN devices, and displays the information organized by vendors and channels.

## Examples

### Detect CAN Devices

Detect the available CAN devices and investigate a device channel.

```
hw = canHWInfo
```

```
hw =
```

```
CAN Devices Detected
```

Vendor	Device	Channel	Serial Number	Constructor...
MathWorks	Virtual 1	1	0	canChannel(...
MathWorks	Virtual 1	2	0	canChannel(...
Kvaser	Virtual 1	1	0	canChannel(...
Kvaser	Virtual 1	2	0	canChannel(...
NI	Virtual (CAN256)	1	0	canChannel(...
NI	Virtual (CAN257)	2	0	canChannel(...
NI	Series 847X Sync USB (CAN0)	1	12345C	canChannel(...
NI	9862 CAN/HS (CAN1)	1	12345A	canChannel(...
Vector	Virtual 1	1	0	canChannel(...
Vector	Virtual 1	2	0	canChannel(...
PEAK-System	PCAN-USB Pro (PCAN_USBBUS1)	1	0	canChannel(...

```
PEAK-System | PCAN-USB Pro (PCAN_USBBUS2) | 2 | 0 | canChannel(...
```

View the Vector properties to see its `VendorDriverVersion`.

```
v = hw.VendorInfo(4)
```

```
v =
```

```
VendorInfo with properties:
```

```
VendorName: 'Vector'
VendorDriverDescription: 'XL Driver Library'
VendorDriverVersion: '9000022'
ChannelInfo: [1x2 can.vector.ChannelInfo]
```

View the first Vector channel information.

```
c1 = hw.VendorInfo(4).ChannelInfo(1)
```

```
c1 =
```

```
ChannelInfo with properties:
```

```
Device: 'Virtual 1'
DeviceChannelIndex: 1
DeviceSerialNumber: 0
ObjectConstructor: 'canChannel('Vector','Virtual 1',1)'
```

## Output Arguments

### **hw** — CAN devices detected

`can.HardwareInfo` object

CAN devices detected, returned as a `can.HardwareInfo` object. You can programmatically access vendor and channel information by indexing into the output object `VendorInfo` property.



## See Also

### Functions

canChannel | canChannelList

**Introduced in R2009a**

## canMessage

Build CAN message based on user-specified structure

### Syntax

```
message = canMessage(id,extended,dataLength)
message = canMessage(id,extended,dataLength,'ProtocolMode','CAN FD')
message = canMessage(candb,messageName)
```

### Description

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

`message = canMessage(id,extended,dataLength,'ProtocolMode','CAN FD')` creates a CAN FD message. The default `ProtocolMode` is standard `'CAN'`.

`message = canMessage(candb,messageName)` creates a message using the message definition in the specified database. Because `ProtocolMode` is defined in the message database, you cannot specify it as an argument to `canMessage` when using a database.

### Examples

#### Create a CAN Message

Create a CAN message with an extended ID format.

```
message1 = canMessage(2500,true,4)
```

```
message1 =
```

```
    Message with properties:
```

```
    Message Identification
```

```
ProtocolMode: 'CAN'
  ID: 5000
  Extended: 1
  Name: ''

Data Details
  Timestamp: 0
  Data: [0 0 0 0]
  Signals: []
  Length: 4

Protocol Flags
  Error: 0
  Remote: 0

Other Information
  Database: []
  UserData: []
```

### Create a CAN FD Message

Create a CAN FD message with a standard ID format.

```
message2 = canMessage(1000, false, 64, 'ProtocolMode', 'CAN FD')
```

```
message2 =
```

```
Message with properties:
```

```
Message Identification
  ProtocolMode: 'CAN FD'
  ID: 1000
  Extended: 0
  Name: ''

Data Details
  Timestamp: 0
  Data: [1×64 uint8]
  Signals: []
  Length: 64
  DLC: 15

Protocol Flags
```

```
        BRS: 0
        ESI: 0
        Error: 0

Other Information
  Database: []
  UserData: []
```

### Create a Message with Database Definitions

Create a message using the definitions of a CAN database.

```
candb = canDatabase(string([(matlabroot) '\examples\vnt\VehicleInfo.dbc']))
message3 = canMessage(candb, 'WheelSpeeds')
```

```
message3 =
```

```
Message with properties:
```

```
Message Identification
```

```
  ProtocolMode: 'CAN'
             ID: 1200
  Extended: 0
  Name: 'WheelSpeeds'
```

```
Data Details
```

```
  Timestamp: 0
  Data: [0 0 0 0 0 0 0 0]
  Signals: [1x1 struct]
  Length: 8
```

```
Protocol Flags
```

```
  Error: 0
  Remote: 0
```

```
Other Information
```

```
Database: [1x1 can.Database]  
UserData: []
```

## Input Arguments

### **id** — ID of message

numeric value

ID of the message, specified as a numeric value. If this ID used an extended format, set the `extended` argument `true`.

Example: 2500

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **extended** — Indicate if message ID is extended

`true` | `false`

Indicates whether the message ID is of standard or extended type, specified as `true` or `false`. The logical value `true` indicates that the ID is of extended type, `false` indicates standard type.

Example: `true`

Data Types: `logical`

### **dataLength** — Length of message data

integer value 0-8

The length of the message data, specified as an integer value of 0 through 8, inclusive.

Example: 8

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **canDb** — CAN database

CAN database object

CAN database, specified as a database object. The database contains the message definition.

Example: `canDb = canDatabase('J1939.dbc')`

**messagename — Name of message**

char vector | string

The name of the message definition in the database, specified as a character vector or string.

Example: 'VehicleDataMulti'

Data Types: char | string

## Output Arguments

**message — CAN message**

CAN message object

CAN message, returned as a CAN message object, with the following properties:

Property	Purpose
BRS	CAN FD bit rate switch, as true or false
Data	Data of CAN message or J1939 parameter group
Database	CAN database information
DLC	Data length code value
Error	CAN message error frame, as true or false
ESI	CAN FD error state indicator, as true or false
Extended	True or false indication of extended CAN Identifier type
ID	Identifier for CAN message
Length	Message length in bytes
Name	CAN message name
ProtocolMode	Protocol mode defined as CAN or CAN FD
Remote	Specify if CAN message is remote frame
Signals	Physical signals defined in CAN message or J1939 parameter group

Property	Purpose
Timestamp	Message received timestamp
UserData	Custom data

## See Also

### Functions

attachDatabase | canDatabase | extractAll | extractRecent | extractTime | pack | unpack

**Introduced in R2009a**

## canMessageBusType

Create Simulink CAN message bus

### Syntax

```
canMessageBusType  
canMessageBusType(modelName)
```

### Description

`canMessageBusType` creates a Simulink CAN message bus object named `CAN_MESSAGE_BUS` in the base workspace. The values of the object properties are read-only, but useful for showing the structure of its data.

`canMessageBusType(modelName)` creates a Simulink CAN message bus object of type `CAN_MESSAGE_BUS` in the data dictionary associated with the specified model, `modelName`.

### Examples

#### Create CAN Message Bus Object

Create and view the properties of a Simulink CAN message bus object.

```
canMessageBusType  
CAN_MESSAGE_BUS  
  
CAN_MESSAGE_BUS =  
  
    Bus with properties:  
  
    Description: ''  
    DataScope: 'Auto'  
    HeaderFile: ''
```



```

Alignment: -1
Elements: [7x1 Simulink.BusElement]

```

View the Elements properties.

```
CAN_MESSAGE_BUS.Elements
```

```
ans =
```

```
7x1 BusElement array with properties:
```

```

Min
Max
DimensionsMode
SampleTime
Description
Unit
Name
DataType
Complexity
Dimensions

```

## Input Arguments

**modelName** — Name of model

char vector | string

Name of model, specified as a character vector or string, whose data dictionary is updated with the bus object.

Example: 'CANModel'

Data Types: char | string

## See Also

### Blocks

CAN Pack | CAN Receive | CAN Replay

### Topics

“Create Custom CAN Blocks” on page 10-28

“Composite Signals” (Simulink)

**Introduced in R2017b**

# canMessageImport

Import CAN messages from third-party log file

## Syntax

```
message = canMessageImport(file,vendor)
message = canMessageImport(file,vendor,candb)
message = canMessageImport( ____, 'OutputFormat', 'timetable')
```

## Description

`message = canMessageImport(file,vendor)` imports CAN messages from the log file, `file`, from a third-party vendor, `vendor`. All the messages in the log file are imported as an array of CAN message objects.

After importing, you can analyze, transmit, or replay these messages.

`canMessageImport` assumes that the information in the imported log file is in a hexadecimal format, and that the timestamps in the imported log file are absolute values.

`message = canMessageImport(file,vendor,candb)` applies the information in the specified database to the imported CAN log messages.

To import Vector log files with symbolic message names, specify an appropriate database file.

`message = canMessageImport( ____, 'OutputFormat', 'timetable')` returns a timetable of messages. This is the recommended output format for optimal performance and representation of CAN messages within MATLAB.

## Examples

### Import Raw Messages

Import raw messages from a log file.

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

### Import Messages with Database

Import messages from a log file, using database information for physical messages.

```
candb = canDatabase('myDatabase.dbc');  
message = canMessageImport('MsgLog.txt', 'Kvaser', candb, 'OutputFormat', 'timetable');
```

## Input Arguments

### **file** — Name of CAN message log file

char vector | string

Name of CAN message log file, specified as a character vector or string.

Example: 'MsgLog.asc'

Data Types: char | string

### **vendor** — Name of vendor

char vector | string

Name of vendor, specified as a character vector or string, whose CAN message log file you are importing from.

You can import message logs only in certain file formats: ASCII files from Vector, and text files from Kvaser.

Example: 'Vector'

Data Types: char | string

### **candb** — CAN database

database object

CAN database, specified as a database object. This is the database whose information is applied to the imported log file messages.

Example: `candb = canDatabase('CANdb.dbc')`

## Output Arguments

### **message — Imported messages**

array of CAN message objects | timetable

Imported messages, returned as an array of CAN message objects or as a timetable of messages.

## See Also

### **Functions**

`canDatabase` | `receive` | `transmit`

**Introduced in R2010b**

## canMessageReplayBlockStruct

Convert CAN messages for use as CAN Replay block output

### Syntax

```
msgstructofarrays = canMessageReplayBlockStruct(msgs)
```

### Description

`msgstructofarrays = canMessageReplayBlockStruct(msgs)` formats specified CAN messages for use with the CAN Replay block. The CAN Replay block requires a specific format for CAN messages, defined by a structure of arrays containing the ID, Extended, Data, and other message elements.

Use this function to assign the formatted message structure to a variable. Then save this variable to a MAT-file. The CAN Replay block mask allows selection of this MAT file and the variable within it, to define the messages to replay in a Simulink model.

### Examples

#### Create CAN Replay Block Message Structure

Create a message structure for the CAN Replay block, and save it to a MAT-file.

```
canMsgs = canMessageReplayBlockStruct(messages);  
save('ReplayBlockMessages.mat', 'canMsgs');
```

### Input Arguments

#### **msgs** — Original CAN messages

CAN message objects | CAN message timetable

Original CAN messages, specified as a CAN message timetable or an array of CAN message objects.

## Output Arguments

### **msgstructofarrays — Formatted CAN messages**

struct

Formatted CAN messages, returned as structure of arrays containing the ID, Extended, Data, and other elements of the messages.

## See Also

### **Functions**

canMessageTimetable | save

### **Blocks**

CAN Replay

**Introduced in R2017a**

## canMessageTimetable

Convert CAN messages into timetable

### Syntax

```
msgtimetable = canMessageTimetable(msg)
msgtimetable = canMessageTimetable(msg,database)
```

### Description

`msgtimetable = canMessageTimetable(msg)` creates a CAN message timetable from existing raw messages. The output message timetable contains the raw message information (ID, Extended, Data, etc.) from the messages. If CAN message objects are input which contain decoded information, that decoding is retained in the CAN message timetable. A timetable of CAN message data can often provide better performance than using CAN message objects.

`msgtimetable = canMessageTimetable(msg,database)` uses the database to decode the message names and signals for the timetable along with the raw message information. Specify multiple databases in an array to decode message names and signals in the timetable within a single call.

The input `msg` can also be a timetable of data created by using `read` on an `mdf` object. In this case, the function converts the timetable of ASAM standard logging format data to a Vehicle Network Toolbox CAN message timetable.

### Examples

#### Convert Log Block Output to Timetable

Convert log block output to a CAN message timetable.



```
load LogBlockOutput.mat
db = canDatabase('myDatabase.dbc')
msgTimetable = canMessageTimetable(canMsgs,db)
```

## Convert CAN Message Objects to Timetable

Convert legacy CAN message objects to a CAN message timetable.

```
msgTimetable = canMessageTimetable(canMsgs);
```

## Decode Message Timetable with Database

Decode an existing CAN message timetable with a database.

```
db = canDatabase('myDatabase.dbc')
msgTimetable = canMessageTimetable(msgTimetable,db)
```

## Convert an ASAM MDF Message Timetable

Convert an existing ASAM format message timetable, and decode using a database.

```
m = mdf('mdfFiles\CANonly.MF4');
db = canDatabase('dbFiles\dGenericVehicle.dbc');
mdfData = read(m);
msgTimetable = canMessageTimetable(mdfData{1},db);
```

Compare the two timetables.

```
mdfData{1}(1:4,1:6)
```

```
ans =
```

```
4x6 timetable
```

Time	CAN_DataFrame_DataLength	CAN_DataFrame_WakeUp	CAN_DataFrame_SingleWire	CAN_DataFrame_IDE
0.019968 sec	4	0	0	0
0.029964 sec	4	0	0	0
0.039943 sec	4	0	0	0
0.049949 sec	4	0	0	0

```
msgTimetable(1:4,1:6)
```

```
ans =
```

```
4x6 timetable
```

Time	ID	Extended	Name	Data	Length	Signals
0.019968 sec	100	false	''	[1x4 uint8]	4	[0x0 struct]
0.029964 sec	100	false	''	[1x4 uint8]	4	[0x0 struct]
0.039943 sec	100	false	''	[1x4 uint8]	4	[0x0 struct]
0.049949 sec	100	false	''	[1x4 uint8]	4	[0x0 struct]

## Input Arguments

### **msg** — CAN message data

CAN message timetable, array, or structure

CAN message data, specified as a CAN message timetable, an array of CAN message objects, or a CAN message structure from the CAN log block.

### **database** — CAN database

database handle

CAN database, specified as a database handle.

## Output Arguments

### **msgtimetable** — CAN message timetable

timetable

CAN messages returned as a timetable.

## See Also

### Functions

canDatabase | canSignalTimetable | mdf

**Introduced in R2017a**

# canSignalImport

Import CAN log file into decoded signal timetables

## Syntax

```
sigtimetable = canSignalImport(file,vendor,database)
sigtimetable = canSignalImport(file,vendor,database,msgnames)
```

## Description

`sigtimetable = canSignalImport(file,vendor,database)` imports a CAN message log file from the specified vendor directly into decoded signal value timetables using the provided database. The function returns a structure with a field for each unique message in the timetable. Each field value is a timetable of all the signals in all instances of that message. Use this form of syntax to convert an entire set of messages in a single function call.

`sigtimetable = canSignalImport(file,vendor,database,msgnames)` returns signal timetables for only the messages specified by `msgnames`, which can specify one or more message names. Use this syntax form to import signals from only a subset of messages.

## Examples

### Import Signals from Log for All Messages

Create signal timetables from all messages in a log file.

```
db = canDatabase('MyDatabase.dbc');
sigtimetable = canSignalImport('MsgLog.asc', 'Vector', db);
```

## Import Signals from Log for Specified Messages

Create signal timetables from specified messages in a log file.

```
db = canDatabase('MyDatabase.dbc');  
sigtimetable1 = canSignalImport('MsgLog.asc', 'Vector', db, 'Message1');  
sigtimetable2 = canSignalImport('MsgLog.asc', 'Vector', db, {'Message1', 'Message2'});
```

## Input Arguments

### **file** — CAN message log file

character vector | string

CAN message log file, specified as a character vector or string.

Example: 'MyDatabase.dbc'

Data Types: char | string

### **vendor** — Vendor file format

'Kvaser' | 'Vector'

Vendor file format, specified as a character vector or string. The supported file formats are those defined by Vector and Kvaser.

Example: 'Vector'

Data Types: char | string

### **database** — CAN database

database handle

CAN database, specified as a database handle.

### **msgnames** — Message names

char | string | cell

Message names, specified as a character vector, string, or array.

Example: 'message1'

Data Types: char | string | cell

## Output Arguments

### **sigtimetable — CAN signals**

structure

CAN signals, returned as a structure. The structure field names correspond to the messages of the input, and each field value is a timetable of CAN signals.

Data Types: `struct`

## See Also

### **Functions**

`canDatabase` | `canMessageImport` | `canSignalTimetable`

**Introduced in R2017a**

## canSignalTimetable

Create CAN signal timetable from CAN message timetable

### Syntax

```
sigtimetable = canSignalTimetable(msgtimetable)
sigtimetable = canSignalTimetable(msgtimetable,msgnames)
```

### Description

`sigtimetable = canSignalTimetable(msgtimetable)` converts a timetable of CAN message information into individual timetables of signal values. The function returns a structure with a field for each unique message in the timetable. Each field value is a timetable of all the signals in that message. Use this syntax form to convert an entire set of messages in a single function call.

`sigtimetable = canSignalTimetable(msgtimetable,msgnames)` returns signal timetables for only the messages specified by `msgnames`, which can specify one or more message names. Use this syntax form to quickly convert only a subset of messages into signal timetables.

### Examples

#### Create CAN Signal Timetables from All Messages

Create CAN signal timetables from all messages in a CAN message timetable.

```
sigTable = canSignalTimetable(msgTimetable);
```

#### Create CAN Signal Timetable from Specified Messages

Create CAN signal timetables from only specified messages in a CAN message timetable.

```
sigTable1 = canSignalTimetable(msgTimetable, 'Message1');  
sigTable2 = canSignalTimetable(msgTimetable, {'Message1', 'Message2'});
```

## Input Arguments

### **msgtimetable** — CAN message timetable

timetable

CAN messages, specified as a timetable.

### **msgnames** — Message names

char | string | cell

Message names, specified as a character vector, string, or array.

Data Types: char | string | cell

## Output Arguments

### **sigtimetable** — CAN signals

structure

CAN signals, returned as a structure. The structure field names correspond to the messages of the input, and each field value is a timetable of CAN signals.

Data Types: struct

## See Also

### **Functions**

canMessageTimetable | canSignalImport

**Introduced in R2017a**

## canSupport

Generate technical support log

### Syntax

```
canSupport
```

### Description

canSupport generates diagnostic information for all installed CAN devices and saves the results to the text file `cansupport.txt` in the current working folder. The MATLAB Editor opens the file for you to view.

For online support, see the **Product Resources** section of the Vehicle Network Toolbox web page.

### Examples

#### Generate Support Log

Generate a technical support log file and view it in the MATLAB editor.

```
canSupport
```

### See Also

#### Functions

```
canChannelList
```

#### External Websites

Vehicle Network Toolbox



**Introduced in R2009a**

## **canTool**

Open Vehicle CAN Bus Monitor

### **Syntax**

```
canTool
```

### **Description**

canTool starts the Vehicle CAN Bus Monitor, which displays live CAN message traffic. This app allows you to view message traffic using a selected CAN device and channel, and to save messages to a log file.

### **Examples**

#### **Open Vehicle CAN Bus Monitor**

Open the Vehicle CAN Bus Monitor app.

```
canTool
```

### **See Also**

#### **Apps**

**Vehicle CAN Bus Monitor**

#### **Topics**

“Using the Vehicle CAN Bus Monitor” on page 5-9

“Vehicle CAN Bus Monitor” on page 5-2

**Introduced in R2009a**

## CAN.VendorInfo class

**Package:** CAN

Display available device vendor information

---

**Note** `can.VendorInfo` will be removed in a future release. Use `canChannelList` instead.

---

### Syntax

```
info = canHWInfo  
info.VendorInfo(index)
```

### Description

`info = canHWInfo` creates an object with information of all available CAN hardware devices.

`info.VendorInfo(index)` displays available vendor information obtained from `canHWInfo` for the device with the specified `index`.

### Input Arguments

**index** — Device channel index

numeric value

Device channel index specified as a numeric value.

### Properties

**VendorName**

Name of the device vendor.

## VendorDriverDescription

Description of the device driver installed for this vendor.

## VendorDriverVersion

Version of the device driver installed for this vendor.

## ChannelInfo

Information on the device channels available for this vendor.

# Examples

## Examine Kvaser Vendor Information

Get information on installed CAN devices.

```
info = canHWInfo
```

```
info =
```

```
CAN Devices Detected
```

Vendor	Device	Channel	Serial Number	Constructor
Kvaser	Virtual 1	1	0	canChannel('Kvaser', 'Virtual 1', 1)
Kvaser	Virtual 1	2	0	canChannel('Kvaser', 'Virtual 1', 2)
Vector	Virtual 1	1	0	canChannel('Vector', 'Virtual 1', 1)
Vector	Virtual 1	2	0	canChannel('Vector', 'Virtual 1', 2)

Use GET on the output of canHWInfo for more information.

Parse the objects VendorInfo class.

```
info.VendorInfo
```

```
ans =
```

```
1x2 heterogeneous VendorInfo (VendorInfo, VendorInfo) array with properties:
```

```
VendorName
VendorDriverDescription
```

VendorDriverVersion  
ChannelInfo

## See Also

### Functions

CAN.ChannelInfo | canHWInfo

# **cdfx**

Access information contained in CDFX-file

## **Syntax**

```
cdfxObj = cdfx(CDFXfile)
```

## **Description**

`cdfxObj = cdfx(CDFXfile)` creates an `asam.cdfx` object and imports the calibration data from the specified CDFX-file.

## **Examples**

### **Access CDFX-File**

Create an `asam.cdfx` object containing the calibration data from a CDFX-file.

```
cdfxObj = cdfx('c:\DataFiles\AllCategories_VCD.cdfx')
```

```
cdfxObj =
```

```
    CDFX with properties:
```

```
        Name: "AllCategories_VCD.cdfx"
```

```
        Path: "c:\DataFiles\AllCategories_VCD.cdfx"
```

```
        Version: "CDF20"
```

## **Input Arguments**

**CDFXfile** — Calibration data format CDFX-file

char | string

Calibration data format CDFX-file, specified as a character vector or string. `CDFXFile` can specify the file name in the current folder, or the full or relative path to the CDFX-file. For restrictions on the file content, see “File Format Limitations” on page 11-4.

Example: `'ASAMCDFExample.cdfx'`

Data Types: `char` | `string`

## Output Arguments

### **`cdfxObj` — CDFX-file object**

`asam.cdfx` object

CDFX-file object, returned as an `asam.cdfx` object. Use the object to access the calibration data.

## See Also

### **Functions**

`getValue` | `instanceList` | `setValue` | `systemList` | `write`

**Introduced in R2019a**



# channelList

Information on available MDF groups and channels

## Syntax

```
chans = channelList(mdfobj)
channelList(mdfObj,chanName)
channelList(mdfObj,chanName,'ExactMatch',true)
```

## Description

`chans = channelList(mdfobj)` returns a table of information about channels and groups in the specified MDF file.

`channelList(mdfObj,chanName)` searches the MDF file to generate a list of channels matching the specified channel name. The search by default is case-insensitive and identifies partial matches. A table is returned containing information about the matched channels and the containing channel groups. If no matches are found, an empty table is returned.

`channelList(mdfObj,chanName,'ExactMatch',true)` searches the channels for an exact match, including case sensitivity. This is useful if a channel name is a substring of other channel names.

## Examples

### View Available MDF Channels

View all available MDF channels.

```
mdfObj = mdf('File01.mf4');
chans = channelList(mdfObj)
```

```
chans =
```

4×9 table

ChannelName	ChannelGroupNumber	ChannelGroupNumSamples
"Float_32_LE_Offset_64"	2	10000
"Float_64_LE_Master_Offset_0"	2	10000
"Sigend_Int16_LE_Offset_32"	1	10000
"Unsigend_UInt32_LE_Master_Offset_0"	1	10000

### View Specific MDF Channels

Filter on channel names.

```
chans = channelList(mdf0bj, 'Float')
```

chans =

2×9 table

ChannelName	ChannelGroupNumber	ChannelGroupNumSamples
"Float_32_LE_Offset_64"	2	10000
"Float_64_LE_Master_Offset_0"	2	10000

```
chans = channelList(mdf0bj, 'Float', 'ExactMatch', true)
```

chans =

0×9 empty table

## Input Arguments

### mdf0bj — MDF file

MDF file object

MDF file, specified as an MDF file object.

Example: `mdf('File01.mf4')`

### chanName — Name of channel

char vector | string

Name of channel, specified as a character vector or string. By default, case-insensitive and partial matches are returned.

Example: 'Channel1'

Data Types: char | string

## Output Arguments

### **chans — Information on available MDF channels**

table

Information on available MDF channels, returned as a table. To access specific elements, you can index into the table.

## See Also

### **Functions**

mdf

**Introduced in R2018b**

## configBusSpeed

Set bit timing rate of CAN channel

### Syntax

```
configBusSpeed(canch, busspeed)
```

```
configBusSpeed(canch, busspeed, SJW, TSeg1, TSeg2, numsamples)
```

```
configBusSpeed(canch, arbbusspeed, databusspeed)
```

```
configBusSpeed(canch, arbbusspeed, arbSJW, arbTSeg1, arbTSeg2,  
databusspeed, dataSJW, dataTSeg1, dataTSeg2)
```

```
configBusSpeed(canch, clockfreq, arbBRP, arbSJW, arbTSeg1, arbTSeg2,  
dataBRP, dataSJW, dataTSeg1, dataTSeg2)
```

### Description

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

- 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.

`configBusSpeed(canch, busspeed, SJW, TSeg1, TSeg2, numsamples)` 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.

---

**Note** Before you can start a channel to transmit or receive CAN FD messages, you must configure its bus speed.

---

`configBusSpeed(canch, arbuspeed, databuspeed)` sets the arbitration and data bus speeds of `canch` using default bit timing calculation factors for CAN FD. This syntax supports NI and MathWorks virtual devices.

`configBusSpeed(canch, arbuspeed, arbSJW, arbTSeg1, arbTSeg2, databuspeed, dataSJW, dataTSeg1, dataTSeg2)` sets the data and arbitration bus speeds of `canch` using the specified bit timing calculation factors in an advanced form for CAN FD. This syntax supports Kvaser and Vector devices.

`configBusSpeed(canch, clockfreq, arbBRP, arbSJW, arbTSeg1, arbTSeg2, dataBRP, dataSJW, dataTSeg1, dataTSeg2)` sets the data and arbitration bus speeds of `canch` using the specified bit timing calculation factors in an advanced form for CAN FD. This syntax supports PEAK-System devices.

## Examples

### Configure Bus Speed

Configure the bus speed using baseline bit timing calculation.

Configure for CAN.

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

Configure CAN FD on MathWorks virtual channel.

```
canch = canChannel('MathWorks', 'Virtual 1', 1, 'ProtocolMode', 'CAN FD');  
configBusSpeed(canch, 1000000, 2000000)
```

Configure CAN FD on National Instruments device.

```
canch = canChannel('NI', 'CAN1', 'ProtocolMode', 'CAN FD');  
configBusSpeed(canch, 1000000, 2000000)
```

### Specify Bit Timing Parameters

Configure the bus speed, specifying the bit timing parameters.

Configure CAN on Kvaser device.

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

Configure CAN FD on Kvaser device.

```
canch = canChannel('Kvaser', 'USBcan Pro 1', 1, 'ProtocolMode', 'CAN FD');  
configBusSpeed(canch, 1e6, 2, 6, 3, 2e6, 2, 6, 3)
```

Configure CAN FD on Vector device.

```
canch = canChannel('Vector', 'VN1610 1', 1, 'ProtocolMode', 'CAN FD');  
configBusSpeed(canch, 1e6, 2, 6, 3, 2e6, 2, 6, 3)
```

Configure CAN FD on PEAK-System device.

```
canch = canChannel('PEAK-System', 'PCAN_USBBUS1', 'ProtocolMode', 'CAN FD');  
configBusSpeed(canch, 20, 5, 1, 2, 1, 2, 1, 3, 1)
```

## Input Arguments

### **canch — CAN channel**

CAN channel object

CAN channel, specified as a CAN channel object.

### **busspeed — Bit rate for channel**

double

Bit rate for channel, specified as a double. Provide the speed of the network in bits per second.

Example: 250000

Data Types: double

### **SJW — Synchronization jump width**

double

Synchronization jump width, specified as a double. Define the length of a bit on the network.

Data Types: double

### **TSeg1 — Time segment 1**

double

Time segment 1, specified as a double, which defines the section before a bit is sampled on the network.

Data Types: double

### **TSeg2 — Time segment 2**

double

Time segment 2, specified as a double, which defines the section after a bit is sampled on a network.

Data Types: double

### **numSamples — Number of samples for bit state**

double

Number of samples for bit state, specified as a double. Specify the number of samples used for determining the bit state of a network.

Data Types: double

### **arbusspeed — Arbitration bit rate for channel**

double

Arbitration bit rate for channel, specified as a double. Provide the speed of the network in bits per second.

Example: 250000

Data Types: double

### **arbSJW — Arbitration synchronization jump width**

double

Arbitration synchronization jump width, specified as a double. Define the length of a bit on the network.

Data Types: double

### **arbTSeg1 — Arbitration time segment 1**

double

Arbitration time segment 1, specified as a double, which defines the section before a bit is sampled on the network.

Data Types: double

**arbTSeg2 — Arbitration time segment 2**

double

Arbitration time segment 2, specified as a double, which defines the section after a bit is sampled on a network.

Data Types: double

**databusspeed — Data bit rate for channel**

double

Data bit rate for channel, specified as a double. Provide the speed of the network in bits per second.

Example: 250000

Data Types: double

**dataSJW — Data synchronization jump width**

double

Data synchronization jump width, specified as a double. Define the length of a bit on the network.

Data Types: double

**dataTSeg1 — Data time segment 1**

double

Data time segment 1, specified as a double, which defines the section before a bit is sampled on the network.

Data Types: double

**dataTSeg2 — Data time segment 2**

double

Data time segment 2, specified as a double, which defines the section after a bit is sampled on a network.

Data Types: double

**clockfreq — Clock frequency**

double



Clock frequency for channel in MHz, specified as a double.

Example: 250000

Data Types: double

**arbBRP — Arbitration clock prescalar for time quantum**

double

Arbitration clock prescalar for time quantum, specified as a double.

Example: 250000

Data Types: double

**dataBRP — Data clock prescalar for time quantum**

double

Data clock prescalar for time quantum, specified as a double.

Example: 250000

Data Types: double

## See Also

### Functions

canChannel | start

**Introduced in R2009a**

## configBusSpeed (J1939)

Configure bit timing of J1939 channel

### Syntax

```
configBusSpeed(chan, busspeed)  
configBusSpeed(chan, busspeed, SJW, TSeg1, TSeg2, numsamples)
```

### Description

`configBusSpeed(chan, busspeed)` sets the speed of the J1939 channel `chan` to `busspeed` in a direct form that uses default bit timing calculation factors.

---

**Note** You can set bit timing only when the channel is offline and has initialization access to the device.

---

`configBusSpeed(chan, busspeed, SJW, TSeg1, TSeg2, numsamples)` sets the speed of the channel using specified bit timing calculation factors.

---

**Note** Unless you have specific timing requirements provided for your network, you should use the direct form of the function.

---

### Examples

#### Set Bus Speed for Channel Directly

Use the direct form of syntax to configure a J1939 channel bus speed.

```
db = canDatabase('MyDatabase.dbc');
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);
configBusSpeed(chan, 250000)
```

### Set Bus Speed for Channel with Calculation Factors

Use the advanced form of syntax to configure a J1939 channel bus speed with specific calculation factors.

```
db = canDatabase('MyDatabase.dbc');
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);
configBusSpeed(chan, 500000, 1, 4, 3, 1)
```

## Input Arguments

### chan — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

### busspeed — Bit rate for channel

double

Bit rate for channel, specified as a double. Provide the speed of the network in bits per second.

Example: 250000

Data Types: double

### SJW — Synchronization jump width

double

Synchronization Jump Width, specified as a double. Define the length of a bit on a network.

Data Types: double

### TSeg1 — Time segment 1

double

Time segment 1, specified as a double, which defines the section before a bit is sampled on a network.

Data Types: double

### **TSeg2 — Time segment 2**

double

Time segment 2, specified as a double, which defines the section after a bit is sampled on a network.

Data Types: double

### **numSamples — Number of samples for bit state**

double

Number of samples for bit state, specified as a double. Specify the number of samples used for determining the bit state of a network.

Data Types: double

## **See Also**

### **Functions**

`j1939Channel | start | stop | transmit`

### **Introduced in R2015b**

# connect

Connect XCP channel to slave module

## Syntax

```
connect(xcpch)
```

## Description

`connect(xcpch)` creates an active connection between the XCP channel and the slave module, enabling active messaging between the channel and the slave.

## Examples

### Connect to a Slave Module

Create an XCP channel connected to a Vector CAN device on a virtual channel and connect it.

Link an A2L file to and create an XCP channel with it.

```
a2lfile = xcpA2L('XCPSIM.a2l')  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the channel and verify that it is connected.

```
connect(xcpch)  
isConnected(xcpch)
```

ans =

1

## Input Arguments

### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

### **Functions**

`readSingleValue` | `writeSingleValue` | `xcpA2L` | `xcpChannel`

**Introduced in R2013a**

# createMeasurementList

Create measurement list for XCP channel

## Syntax

```
createMeasurementList(xcpch, resource, eventName, measurementName)  
createMeasurementList(xcpch, resource, eventName, {measurementName,  
measurementName, measurementName})
```

## Description

`createMeasurementList(xcpch, resource, eventName, measurementName)` creates a data stimulation list for the XCP channel with the specified event and measurement.

`createMeasurementList(xcpch, resource, eventName, {measurementName, measurementName, measurementName})` creates a data stimulation list for the XCP channel with the specified event and list of measurements.

## Examples

### Create a DAQ Measurement List

Create an XCP channel connected to a Vector CAN device on a virtual channel and set up a DAQ measurement list.

```
a2lfile = xcp.A2L('XCPSIM.a2l')  
xcpch = xcp.Channel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1)  
xcpch =
```

Channel with properties:

```
SlaveName: 'CPP'  
A2LFileName: 'XCPSIM.a2l'
```

```
        TransportLayer: 'CAN'  
TransportLayerDevice: [1x1 struct]  
        SeedKeyDLL: []
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data acquisition measurement list with the '10 ms' event and 'Triangle' measurement.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', 'Triangle');
```

### Create a Data Stimulation List

Create an XCP channel connected to a Vector CAN device on a virtual channel and set up a STIM measurement list.

```
a2l = xcp.A2L('XCPSIM.a2l')  
xcpch = xcp.Channel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1)  
xcpch =  
    Channel with properties:  
        SlaveName: 'CPP'  
        A2LFileName: 'XCPSIM.a2l'  
        TransportLayer: 'CAN'  
TransportLayerDevice: [1x1 struct]  
        SeedKeyDLL: []
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data stimulation measurement list with the '100ms' event and 'PWM' and 'ShiftByte' measurements.

```
createMeasurementList(xcpch, 'STIM', '100ms', {'PWM', 'ShiftByte'});
```

## Input Arguments

### **xcpch** — XCP channel

XCP channel object



XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

**resource — Measurements list type**

'DAQ' | 'STIM'

Measurement list type, specified as 'DAQ' or 'STIM'.

Example: 'DAQ'

Data Types: `char` | `string`

**eventName — Name of event**

character vector | string

Name of event, specified as a character vector or string. The event is used to trigger the specified measurement list. The list of available events depends on your A2L file.

Data Types: `char` | `string`

**measurementName — Name of single XCP measurement**

character vector | string | array

Name of a single XCP measurement, specified as a character vector or string; or a set of measurements, specified as a cell array of character vectors or array of strings. Make sure `measurementName` matches the corresponding measurement names defined in your A2L file.

## See Also

`freeMeasurementLists` | `startMeasurement` | `viewMeasurementLists`

**Introduced in R2013a**

## discard

Discard all messages from CAN channel

### Syntax

```
discard(canch)
```

### Description

`discard(canch)` discards messages that are available to receive on the channel `canch`.

### Examples

#### Discard Messages Received by a CAN Channel

Set up a CAN channel to receive messages, then discard the messages.

Create a CAN channel to receive messages and start the channel.

```
rxCh = canChannel('Vector', 'CANcaseXL 1', 1);  
start (rxCh)
```

Discard all messages in this channel.

```
discard(rxCh);
```

### Input Arguments

#### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, that you want to discard the messages from.

Example: `canChannel('NI', 'CAN1')`

## See Also

### Functions

`canChannel`

**Introduced in R2012a**

## discard (J1939)

Discard available parameter groups on J1939 channel

### Syntax

```
discard(chan)
```

### Description

`discard(chan)` deletes all parameter groups available on the J1939 channel `chan`. The channel also deactivates when it is cleared from memory.

### Examples

#### Discard Parameter Groups on Channel

Delete all the parameter groups on a J1939 channel.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);  
start(chan)
```

```
discard(chan)
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

## See Also

### Functions

`j1939Channel | start`

**Introduced in R2015b**

## disconnect

Disconnect from slave module

### Syntax

```
disconnect(xcpch)
```

### Description

`disconnect(xcpch)` disconnects the specified XCP channel from the slave module. Disconnecting the channel stops active messaging between the channel and the slave module.

### Examples

#### Disconnect an Active XCP Connection

Create an XCP channel using a CAN module, connect the channel and disconnect it from the specified slave module.

Link an A2L file

```
a2l = xcpA2L('XCPSIM.a2l')
```

Create an XCP channel using a Vector CAN modules's virtual channel. Check to see if channel is connected.

```
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the channel and check to see if channel is connected.

```
connect(xcpch)  
isConnected(xcpch)
```

```
ans =
```

```
    1
```

Disconnect the channel and check if connection is active.

```
disconnect(xcpch)  
isConnected(xcpch)
```

```
ans =
```

```
    0
```

## Input Arguments

### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

`connect` | `isConnected` | `xcpA2L` | `xcpChannel`

**Introduced in R2013a**

## extractAll

Select all instances of CAN message from message array

### Syntax

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

### Description

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

`extracted = extractAll(message, id, extended)` parses the given array `message`, and returns all instances of messages matching the specified ID value and type.

`[extracted, remainder] = extractAll( ___ )` assigns to `extracted` those messages that match the search, and returns to `remainder` those that do not match.

## Examples

### Extract Messages by Name and ID

Extract messages by matching name and IDs.

Extract messages by name.

```
msgOut = extractAll(msgs, 'DoorControlMsg');
```

Extract all messages with IDs 200 and 5000. Note that 5000 requires an extended style ID.

```
msgOut = extractAll(msgs, [200 5000], [false true]);
```

Extract messages and also return the remainder.



```
[msgOut,remainder] = extractAll(msgs,{'DoorControlMsg','WindowControlMsg'});
```

## Input Arguments

### **message** — CAN messages to parse

array of CAN message objects

CAN messages to parse, specified as an array of CAN message objects. This is the collection from which you extract messages by specific names or IDs.

### **messagename** — Name of message to extract

char vector | string | cell

Name of message to extract, specified as a character vector, string, or array that supports these types.

Example: 'DoorControlMsg'

Data Types: char | string | cell

### **id** — ID of message to extract

numeric value or vector

ID of message to extract, specified as a numeric value or vector. Using this argument also requires that you specify an extended argument.

Example: [200 400]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **extended** — Indication of extended ID type

true | false

Indication of extended ID type, specified as a logical true or false. Use a value true if the ID type is extended, or false if standard. This argument is required if you specify a message ID.

If the message ID is a numeric vector, use a logical vector of the same length for extended. For example, if you specify id and extended as [250 5000], [false true], then extractAll returns all instances of CAN messages 250 and 5000 found within in the message array.

Example: `true`

Data Types: `logical`

## Output Arguments

### **extracted** — Extracted CAN messages

array of CAN messages

Extracted CAN messages, returned as an array of CAN message objects. These are the messages whose name or ID matches the specified value.

### **remainder** — Unmatched CAN messages

array of CAN messages

Unmatched CAN messages, returned as an array of CAN message objects. These are the messages in the original set whose name or ID does not match the specified value.

## See Also

### **Functions**

`extractRecent` | `extractTime`

**Introduced in R2009a**

## extractAll (J1939)

Occurrences of specified J1939 parameter groups

### Syntax

```
extractedPGs = extractAll(pgrp, pname)  
[extractedPGs, remainderPGs] = extractAll(pgrp, pname)
```

### Description

`extractedPGs = extractAll(pgrp, pname)` returns all parameter groups whose name occurs in `pname`.

`[extractedPGs, remainderPGs] = extractAll(pgrp, pname)` also returns a parameter group array, `remainder`, containing all groups from the original array not matching the specified names in `pname`.

### Examples

#### Extract Parameter Groups

Extracts all the parameter groups with a name of 'PG1' or 'PG2'.

```
extractedPGs = extractAll(pgrp, {'PG1' 'PG2'})
```

#### Extract Parameter Groups and Remainder

Extract all parameter groups with a name of 'PG1' or 'PG2', and also return unmatched parameter groups to a different array.

```
[extractedPGs,remainderPGs] = extractAll(parameterGroups, {'PG1' 'PG2'})
```

## Input Arguments

### **pgrp — J1939 parameter group**

array of ParameterGroup objects

J1939 parameter groups, specified as an array of ParameterGroup objects. Use the `j1939ParameterGroup` or `receive` function to create ParameterGroup objects.

### **pgname — Names of J1939 parameter groups to extract**

char vector | string | cell array of char vectors

Names of J1939 parameter groups to extract, specified as a character vector, string, or array of these.

Example: 'PG1'

Data Types: char | string | cell

## Output Arguments

### **extractedPGs — Extracted parameter groups**

array of ParameterGroup objects

Extracted parameter groups, returned as an array of ParameterGroup objects. These parameter groups have names matching any of those specified in the `pgname` argument.

### **remainderPGs — Remainder of parameter groups**

array of ParameterGroup objects

Remainder of parameter groups, returned as an array of ParameterGroup objects. These are all the parameter groups with names not matching any of those specified in the `pgname` argument.

## See Also

### **Functions**

`extractRecent` | `extractTime` | `j1939ParameterGroup`

**Introduced in R2015b**

## extractRecent

Select most recent CAN message from array of messages

### Syntax

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

### Description

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

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

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

### Examples

#### Extract Recent Messages

Extract most recent message for each name.

```
msgOut = extractRecent(msgs);
```

Extract recent messages for specific names.

```
msgOut1 = extractRecent(msgs, 'DoorControlMsg');
msgOut2 = extractRecent(msgs, {'DoorControlMsg' 'WindowControlMsg'});
```

Extract recent messages with IDs 200 and 5000. Note that 5000 requires an extended style ID.

```
msgOut = extractRecent(msgs,[200 5000],[false true]);
```

## Input Arguments

### **message** — CAN messages to parse

array of CAN message objects

CAN messages to parse, specified as an array of CAN message objects. This is the collection from which you extract recent messages.

### **messagename** — Name of message to extract

char vector | string | cell

Name of message to extract, specified as a character vector, string, or array that supports these types.

Example: 'DoorControlMsg'

Data Types: char | string | cell

### **id** — ID of message to extract

numeric value or vector

ID of message to extract, specified as a numeric value or vector. Using this argument also requires that you specify an extended argument.

Example: [200 400]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **extended** — Indication of extended ID type

true | false

Indication of extended ID type, specified as a logical true or false. Use a value true if the ID type is extended, or false if standard. This argument is required if you specify a message ID.

If the message ID is a numeric vector, use a logical vector of the same length for extended. For example, if you specify id and extended as [250 5000], [false true], then extractAll returns all instances of CAN messages 250 and 5000 found within in the message array.

Example: `true`

Data Types: `logical`

## Output Arguments

### **extracted** — Extracted CAN messages

array of CAN messages

Extracted CAN messages, returned as an array of CAN message objects. These are the most recent messages matching the search criteria.

## See Also

### **Functions**

`extractAll` | `extractTime`

**Introduced in R2009a**



## extractRecent (J1939)

Occurrences of most recent J1939 parameter groups

### Syntax

```
extractedPGs = extractRecent(pgrp)
extractedPGs = extractRecent(pgrp, pgrname)
```

### Description

`extractedPGs = extractRecent(pgrp)` returns the most recent instance of each unique parameter group found in the array `pgrp`, based on the parameter group timestamps.

`extractedPGs = extractRecent(pgrp, pgrname)` returns the most recent instance of parameter groups whose names match any of those specified in `pgrname`.

### Examples

#### Extract Most Recent Parameter Groups

Extract the most recent of each parameter group.

```
extractedPGs = extractRecent(pgrp)
```

#### Extract Most Recent Parameter Groups for Specific Names

Extract the most recent of each parameter group named 'PG1' or 'PG2'.

```
extractedPGs = extractRecent(pgrp,{'PG1' 'PG2'})
```

## Input Arguments

### **pgrp — J1939 parameter group**

array of ParameterGroup objects

J1939 parameter groups, specified as an array of ParameterGroup objects. Use the `j1939ParameterGroup` or `receive` function to create ParameterGroup objects.

### **pgrname — Names of J1939 parameter groups to extract**

char vector | string | array

Names of J1939 parameter groups to extract, specified as a character vector, string, or array of these.

Example: 'PG1'

Data Types: char | string | cell

## Output Arguments

### **extractedPGs — Extracted parameter groups**

array of ParameterGroup objects

Extracted parameter groups, returned as an array of ParameterGroup objects.

## See Also

### **Functions**

`extractAll` | `extractTime` | `j1939ParameterGroup`

**Introduced in R2015b**

# extractTime

Select CAN messages occurring within specified time range

## Syntax

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

## Description

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

## Examples

### Extract Messages Within Time Range

Extract messages in first 10 seconds of channel being on.

```
msgRange = extractTime(msgs, 0, 10);
```

## Input Arguments

### **message** — CAN messages to parse

array of CAN message objects

CAN messages to parse, specified as an array of CAN message objects. This is the collection from which you extract recent messages.

### **starttime, endtime** — Time range in seconds

numeric values

Time range in seconds, specified as numeric values. The function returns messages with timestamps that fall within the range defined by `starttime` and `endtime`, inclusive.

Specify the time range in increasing order from `starttime` to `endtime`. If you must specify the largest available time, set `endtime` to `Inf`. The earliest time you can specify for `starttime` is `0`.

Example: `0,10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **extracted** — Extracted CAN messages

array of CAN messages

Extracted CAN messages, returned as an array of CAN message objects. These are the messages within the specified time range.

## See Also

### **Functions**

`extractAll` | `extractRecent`

**Introduced in R2009a**

## extractTime (J1939)

Occurrences of J1939 parameter groups within time range

### Syntax

```
extractedPGs = extractTime(pgrp,starttime,endtime)
```

### Description

`extractedPGs = extractTime(pgrp,starttime,endtime)` returns the parameter groups found in the array `pgrp`, with timestamps between the specified `starttime` and `endtime`, inclusive.

### Examples

#### Extract Parameter Groups Within Specified Time Range

Extract the parameter groups according to start and stop timestamps.

Extract parameter groups between 5 and 10.5 seconds.

```
extractedPGs = extractTime(pgrp,5,10.5)
```

Extract all parameter groups within the first minute.

```
extractedPGs = extractTime(pgrp,0,60)
```

Extract all parameter groups after 150 seconds.

```
extractedPGs = extractTime(pgrp,150,Inf)
```

## Input Arguments

### **pgrp — J1939 parameter group**

array of ParameterGroup objects

J1939 parameter groups, specified as an array of ParameterGroup objects. Use the `j1939ParameterGroup` or `receive` function to create ParameterGroup objects.

### **starttime,endtime — Start time and end time**

numeric value

Start time and end time, specified as numeric values. These arguments define the range of time from which to extract parameter groups, inclusively. For the earliest possible `starttime` use `0`, for the latest possible `endtime` use `Inf`. The `endtime` value must be greater than the `starttime` value.

Data Types: `double` | `single`

## Output Arguments

### **extractedPGs — Extracted parameter groups**

array of ParameterGroup objects

Extracted parameter groups, returned as an array of ParameterGroup objects. These parameter groups fall within the specified time range, inclusively.

## See Also

### **Functions**

`extractAll` | `extractRecent` | `j1939ParameterGroup`

**Introduced in R2015b**

# filterAllowAll

Allow all CAN messages of specified identifier type

## Syntax

```
filterAllowAll(canch, type)
```

## Description

`filterAllowAll(canch, type)` opens the filter on the specified CAN channel to allow all messages matching the specified identifier type to pass the acceptance filter.

## Examples

### Allow Standard and Extended ID Messages

Allow all standard and extended ID messages to pass the filter.

```
canch = canChannel('Vector', 'CANCaseXL 1', 1);
filterAllowAll(canch, 'Standard')
filterAllowAll(canch, 'Extended')

canch.FilterHistory

'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

## Input Arguments

### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, on which to filter.

Example: `canch = canChannel('NI', 'CAN1')`

### **type — Identifier type**

'standard' | 'extended'

Identifier type by which to filter, specified as a character vector or string. CAN messages identifier types are 'Standard' and 'Extended'.

Example: 'Standard'

Data Types: char | string

## **See Also**

### **Functions**

canChannel | canMessage | filterAllowOnly | filterBlockAll

**Introduced in R2011b**



## filterAllowAll (J1939)

Open parameter group filters on J1939 channel

### Syntax

```
filterAllowAll(chan)
```

### Description

`filterAllowAll(chan)` opens all parameter group filters on the specified channel, making all parameter groups receivable.

### Examples

#### Allow All Parameter Groups to Be Received

Open the filter to allow all J1939 parameter groups on the channel.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);  
filterAllowAll(chan)
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

## See Also

### Functions

`filterAllowOnly` | `filterBlockOnly` | `j1939Channel`

**Introduced in R2015b**

# filterAllowOnly

Configure CAN message filter to allow only specified messages

## Syntax

```
filterAllowOnly(canch, name)  
filterAllowOnly(canch, IDs, type)
```

## Description

`filterAllowOnly(canch, name)` configures the filter on the channel `canch` to pass only messages with the specified name.

Set the channel object Database property to attach a database to allow filtering by message names.

`filterAllowOnly(canch, IDs, type)` configures the filter on the channel `canch` to pass only messages of the specified identifier type and values.

## Examples

### Filter by Message Name

Filter a database defined message with the name 'EngineMsg'

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

### Filter by Message IDs

Filter messages by identifiers.

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

## Input Arguments

### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, on which to filter.

Example: `canch = canChannel('NI', 'CAN1')`

### **name** — Name of CAN messages

char vector | string

Name of CAN messages that you want to allow, specified as a character vector, string, or supporting array of these types.

Example: `'EngineMsg'`

Data Types: char | string | cell

### **IDs** — CAN message IDs

numeric value

CAN message IDs that you want to allow, specified as a numeric value or vector.

Specify IDs as a decimal value. To convert a hexadecimal to a decimal value, use the `hex2dec` function.

Example: `600, [600, 610], [600:800], [200:400, 600:800]`

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **type** — Identifier type

'standard' | 'extended'

Identifier type by which to filter, specified as a character vector or string. CAN messages identifier types are 'Standard' and 'Extended'.

Example: `'Standard'`

Data Types: char | string

## See Also

### Functions

canChannel | canDatabase | filterAllowAll | filterBlockAll | hex2dec

**Introduced in R2011b**

## filterAllowOnly (J1939)

Allow only specified parameter groups to pass J1939 channel filter

### Syntax

```
filterAllowOnly(chan,pgname)
```

### Description

`filterAllowOnly(chan,pgname)` configures the filter on the channel `chan` to pass only the parameter groups specified by `pgname`.

### Examples

#### Allow Only Some Parameter Groups to Be Received

Configure the channel filter to allow only specified J1939 parameter groups to be received on the channel.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db,'Vector','CANCaseXL 1',1);  
filterAllowOnly(chan,{'PG1' 'PG2'})
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

#### **pgname** — Allowed J1939 parameter groups

char vector | string | array

Allowed J1939 parameter groups, specified as a character vector, string, or array of these.

Example: 'PG1'

Data Types: char | string | cell

## See Also

### Functions

filterAllowAll | filterBlockOnly | j1939Channel

**Introduced in R2015b**

## filterBlockAll

Configure filter to block CAN messages with specified identifier type

### Syntax

```
filterBlockAll(canch, type)
```

### Description

`filterBlockAll(canch, type)` configures the CAN message filter to block all messages matching the specified identifier type.

### Examples

#### Block All Standard ID Messages

Block all standard ID message types.

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

### Input Arguments

#### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, on which to filter.

Example: `canch = canChannel('NI', 'CAN1')`

#### **type** — Identifier type

'standard' | 'extended'



Identifier type by which to filter, specified as a character vector or string. CAN messages identifier types are 'Standard' and 'Extended'.

Example: 'Standard'

Data Types: char | string

## See Also

### Functions

canChannel | filterAllowAll | filterAllowOnly

**Introduced in R2011b**

## filterBlockOnly (J1939)

Block only specified parameter groups on J1939 channel filter

### Syntax

```
filterBlockOnly(chan,pgname)
```

### Description

`filterBlockOnly(chan,pgname)` configures the filter on the channel `chan` to block only the parameter groups specified by `pgname`.

### Examples

#### Block Only Some Parameter Groups on Channel

Configure the channel filter to block only specified J1939 parameter groups on the channel.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db,'Vector','CANCaseXL 1',1);  
filterBlockOnly(chan,{'PG1' 'PG2'})
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

#### **pgname** — Blocked J1939 parameter groups

char vector | string | array

Blocked J1939 parameter groups, specified as a character vector, string, or array of these.

Example: 'PG1'

Data Types: char | string | cell

## See Also

### Functions

filterAllowAll | filterAllowOnly | j1939Channel

**Introduced in R2015b**

## freeMeasurementLists

Remove all measurement lists from XCP channel

### Syntax

```
freeMeasurementLists(xcpch)
```

### Description

`freeMeasurementLists(xcpch)` removes all configured measurement lists from the specified XCP channel.

### Examples

#### Free DAQ Lists

Create two data acquisition lists and remove them.

Create an object to parse an A2L file and connect that to an XCP channel.

```
a2lfile = xcpA2L('XCPSIM.a2l')  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data acquisition measurement list with the '10 ms' event and 'PMW' measurement.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', {'BitSlice0', 'PWMFiltered', 'Triangle'})
```

Create another measurement list with the '100ms' event and 'PWMFiltered', and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'DAQ', '100ms', {'PWMFiltered', 'Triangle'})
```

view details of the measurement lists.

`viewMeasurementLists(xcpch)`

```
DAQ List #1 using the "10 ms" event @ 0.010000 seconds and the following measurements:  
  PWM
```

```
DAQ List #2 using the "100ms" event @ 0.100000 seconds and the following measurements:  
  PWMFiltered  
  Triangle
```

Free the measurement lists.

`freeMeasurementLists(xcpch)`

## Input Arguments

### **xcpch — XCP channel**

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

`createMeasurementList` | `viewMeasurementLists` | `xcpA2L` | `xcpChannel`

**Introduced in R2013a**



```
UpperLimit: 255  
BitMask: []
```

## Input Arguments

### **a2lFile** — A2L file

xcp.A2L object

A2L file, specified as an xcp.A2L object, used in this connection. You can create an A2L file object using `xcpA2L`.

### **characteristic** — XCP channel characteristic name

char vector | string

XCP channel characteristic name, specified as a character vector or string.

Example: 'curve1\_8\_uc'

Data Types: char | string

## Output Arguments

### **info** — XCP characteristic information

xcp.Characteristic object

XCP characteristic information, returned as an xcp.Characteristic object, containing characteristic details such as type, identifier, and conversion.

## See Also

`getEventInfo` | `getMeasurementInfo` | `xcpA2L`

## Topics

“Inspect the Contents of an A2L File” on page 7-2

“XCP Database and Communication Workflow” on page 6-2

**Introduced in R2018a**





```
ChannelPriority: 0  
ChannelTimeCycleInSeconds: 0.0100
```

## Input Arguments

### **a2lFile** — A2L file

xcp.A2L object

A2L file, specified as an xcp.A2L object, used in this connection. You can create an A2L file object using xcpA2L.

### **eventName** — XCP event name

character vector | string

XCP event name, specified as a character vector or string. Make sure `eventName` matches the corresponding event name defined in your A2L file.

## Output Arguments

### **info** — XCP event information

xcp.Event object

XCP event information, returned as xcp.Event object, containing event details such as timing and priority.

## See Also

### Functions

getCharacteristicInfo | getMeasurementInfo | xcpA2L

### Topics

“Inspect the Contents of an A2L File” on page 7-2

“XCP Database and Communication Workflow” on page 6-2

**Introduced in R2013a**

## getMeasurementInfo

Get information about specific measurement from A2L file

### Syntax

```
info = getMeasurementInfo(a2lFile,measurementName)
```

### Description

`info = getMeasurementInfo(a2lFile,measurementName)` returns information about the specified measurement from the specified A2L file, and stores it in the `xcp.Measurement` object, `info`.

### Examples

#### Get XCP Measurement Information

Create a handle to parse an A2L file and get information about the `channel1` measurement.

```
a2lfile = xcpA2L('C:\XCPSIM.a2l')  
info = getMeasurementInfo(a2lfile,'channel1')
```

```
info = Measurement with properties:  
    Resolution: 0  
    Accuracy: 0  
    LocDataType: 'FLOAT32_IEEE'  
        Name: 'channel1'  
    LongIdentifier: 'FLOAT demo signal (sine wave)'  
    ECUAddress: 1155080  
    ECUAddressExtension: 0  
    Conversion: [1x1 xcp.CompuMethodRational]  
    Dimension: 1  
    LowerLimit: -1.0000e+12
```

```
UpperLimit: 1.0000e+12  
BitMask: []
```

## Input Arguments

### **a2lFile** — A2L file

xcp.A2L object

A2L file, specified as an xcp.A2L object, used in this connection. You can create an A2L file object using xcpA2L.

### **measurementName** — Name of single XCP measurement

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure measurementName matches the corresponding measurement name defined in your A2L file.

Data Types: char | string

## Output Arguments

### **info** — XCP measurement information

xcp.Measurement object

XCP measurement information, returned as an xcp.Measurement object, containing measurement details such as memory address, identifier, and limits.

## See Also

getCharacteristicInfo | getEventInfo | xcpA2L

## Topics

“Inspect the Contents of an A2L File” on page 7-2

“XCP Database and Communication Workflow” on page 6-2

**Introduced in R2013a**

## getValue

Retrieve instance value from CDFX object

### Syntax

```
iVal = getValue(cdfxObj,instName)
iVal = getValue(cdfxObj,instName,sysName)
```

### Description

`iVal = getValue(cdfxObj,instName)` returns the value of the unique instance whose `ShortName` is specified by `instName`. If multiple instances share the same `ShortName`, the function returns an error.

`iVal = getValue(cdfxObj,instName,sysName)` returns the value of the instance whose `ShortName` is specified by `instName` and is contained in the system specified by `sysName`.

### Examples

#### Retrieve Value of Instance

Create an `asam.cdfx` object and read the value of its `VALUE_NUMERIC` instance.

```
cdfxObj = cdfx('c:\DataFiles\AllCategories_VCD.cdfx');
iVal = getValue(cdfxObj,'VALUE_NUMERIC')
```

```
iVal =  
    12.2400
```

## Input Arguments

**cdfxObj** — CDFX-file object  
asam.cdfx object

CDFX-file object, specified as an `asam.cdfx` object. Use the object to access the calibration data.

Example: `cdfx()`

**instName** — Instance name  
char | string

Instance name, specified as a character vector or string.

Example: `'NUMERIC_VALUE'`

Data Types: char | string

**sysName** — Parent system name  
char | string

Parent system name, specified as a character vector or string.

Example: `'System2'`

Data Types: char | string

## Output Arguments

**iVal** — Instance value  
instance type

Instance value, returned as the instance type.

## See Also

### Functions

`cdfx` | `instanceList` | `setValue` | `systemList` | `write`

**Introduced in R2019a**

## hasdata (MDFDatastore)

Determine if data is available to read from MDF datastore

### Syntax

```
tf = hasdata(mdfds)
```

### Description

`tf = hasdata(mdfds)` returns logical 1 (`true`) if there is data available to read from the MDF datastore specified by `mdfds`. Otherwise, it returns logical 0 (`false`).

### Examples

#### Check MDF Datastore for Readable Data

Use `hasdata` in a loop to control read iterations.

```
mdfds = mdfDatastore(fullfile(matlabroot, 'examples', 'vnt', 'CANape.MF4'));  
while hasdata(mdfds)  
    m = read(mdfds);  
end
```

### Input Arguments

#### **mdfds** — MDF datastore

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

## Output Arguments

**tf** — Indicator of data to read

1 | 0

Indicator of data to read, returned as a logical 1 (true) or 0 (false).

## See Also

### Functions

mdfDatastore | read | readall | reset

**Introduced in R2017b**



# instanceList

Parameter instances in the CDFX object

## Syntax

```
iList = instanceList(cdfxObj)
iList = instanceList(cdfxObj,instName)
iList = instanceList(cdfxObj,instName,sysName)
```

## Description

`iList = instanceList(cdfxObj)` returns a table of every parameter instance in the CDFX object.

`iList = instanceList(cdfxObj,instName)` returns a table of every parameter instance in the CDFX object whose `ShortName` matches `instName`.

`iList = instanceList(cdfxObj,instName,sysName)` returns a table of every parameter instance in the CDFX object whose `ShortName` matches `instName` and whose parent `System` matches `sysName`.

## Examples

### View CDFX Object Instances

Create an `asam.cdfx` object and view its parameter instances.

```
cdfxObj = cdfx('c:\DataFiles\AllCategories_VCD.cdfx');
iList = instanceList(cdfxObj);
iList(1:4,1:4)
```

ans =

4x4 table

ShortName	System	Category	Value
-----------	--------	----------	-------

"VALUE_NUMERIC"	"System1"	"VALUE"	[ 12.2400]
"VALUE_TEXT"	"System1"	"VALUE"	[ "Text_Value"
"BLOB_HEX"	"System1"	"BLOB"	[ "0102030405060708 090A0B0C0D0E0F10"]
"BOOLEAN_TEXT"	"System1"	"BOOLEAN"	[ 1]

```
iList = instanceList(cdfxObj, "VALUE_NUMERIC")
```

```
iList =
```

```
1x6 table
```

ShortName	System	Category	Value	Units	FeatureReference
"VALUE_NUMERIC"	"System1"	"VALUE"	[12.2400]	" "	"model1"

```
iList = instanceList(cdfxObj, "VALUE_NUMERIC", "System1")
```

```
iList =
```

```
1x6 table
```

ShortName	System	Category	Value	Units	FeatureReference
"VALUE_NUMERIC"	"System1"	"VALUE"	[12.2400]	" "	"model1"

## Input Arguments

### **cdfxObj** — CDFX-file object

asam.cdfx object

CDFX-file object, specified as an `asam.cdfx` object. Use the object to access the calibration data.

Example: `cdfx()`

### **instName** — Instance name

string

Instance name, specified as a string.

Example: "NUMERIC\_VALUE"

Data Types: string

### **sysName** — Parent system name

string

Parent system name, specified as a string.

Example: "System2"

Data Types: string

## Output Arguments

### **iList** – Instance list

table

Instance list, returned as a table.

## See Also

### **Functions**

`cdfx` | `getValue` | `setValue` | `systemList` | `write`

**Introduced in R2019a**

## isConnected

Return connection status

### Syntax

```
isConnected(xcpch)
```

### Description

`isConnected(xcpch)` returns a boolean value to indicate active connection to the slave.

### Examples

#### Verify if XCP Channel is Connected

Create a new XCP channel and see if it is connected.

```
a2l = xcpA2L('XCPSIM.a2l')
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1)
isConnected(xcpch)
```

```
ans =
```

```
0
```

### Input Arguments

#### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## **See Also**

xcpChannel

**Introduced in R2013a**

## isMeasurementRunning

Indicate if measurement is active

### Syntax

```
isMeasurementRunning(xcpch)
```

### Description

`isMeasurementRunning(xcpch)` returns a boolean indicating if the configured measurements are active and running.

### Examples

#### Verify if Configured Measurement List is Active

Set up a DAQ measurement list and start it. Verify if this list is running.

Create an XCP channel with a CAN slave module.

```
a2l = xcpA2L('XCPsim.a2l')  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Setup a data acquisition measurement list with the '10 ms' event and 'BitSlice' measurement and verify if measurement is running.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', 'BitSlice')  
isMeasurementRunning(xcpch)
```

```
ans =
```

```
0
```

Start your measurement and verify if measurement is running.

```
startMeasurement(xcpch)
isMeasurementRunning(xcpch)
```

```
ans =
```

```
    1
```

## Input Arguments

### **xcpch — XCP channel**

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

`startMeasurement`

**Introduced in R2013a**

## j1939Channel

Create J1939 CAN channel

### Syntax

```
j1939Ch = j1939Channel(database, 'vendor', 'device')  
j1939Ch = j1939Channel(database, 'vendor', 'device', chanIndex)
```

### Description

`j1939Ch = j1939Channel(database, 'vendor', 'device')` creates a J1939 channel connected to the specified CAN device. Use this syntax for National Instruments and PEAK-System devices, which do not require a channel index argument.

`j1939Ch = j1939Channel(database, 'vendor', 'device', chanIndex)` creates a J1939 CAN channel connected to the specified CAN device and channel index. Use this syntax for Vector and Kvaser devices that support a channel index specifier.

### Examples

#### Create a J1939 CAN Channel for a Vector Device

Specify a database.

```
db = canDatabase('C:\J1939DB.dbc');
```

Create the channel object.

```
j1939Ch = j1939Channel(db, 'Vector', 'Virtual 1', 1)
```

```
j1939Ch =
```

```
Channel with properties:
```

```
Device Information:
```



```

-----
                DeviceVendor: 'Vector'
                  Device: 'Virtual 1'
    DeviceChannelIndex: 1
    DeviceSerialNumber: 0

Data Details:
-----
    ParameterGroupsAvailable: 0
    ParameterGroupsReceived: 0
    ParameterGroupsTransmitted: 0
        FilterPassList: []
        FilterBlockList: []

Channel Information:
-----
                Running: 0
                BusStatus: 'N/A'
    InitializationAccess: 1
    InitialTimestamp: [0x0 datetime]
        SilentMode: 0
        TransceiverName: ''
        TransceiverState: 0
            BusSpeed: 500000
                SJW: 1
                TSEG1: 4
                TSEG2: 3
            NumOfSamples: 1

Other Information:
-----
                UserData: []

```

### Create a J1939 CAN Channel for a National Instruments Device

Specify a database.

```
db = canDatabase('C:\J1939DB.dbc');
```

Create the channel object.

```
j1939Ch = j1939Channel(db, 'NI', 'CAN1');
```

## Input Arguments

### **database** — CAN database

CAN database object

CAN database specified as a CAN database object. The specified database contains J1939 parameter group definitions.

Example: `database = canDatabase('C:\database.dbc')`

### **vendor** — Name of device vendor

'Vector' | 'NI' | 'Kvaser' | 'Peak-System'

Name of device vendor, specified as a character vector or string.

Example: `'Vector'`

Data Types: `char` | `string`

### **device** — Name of CAN device

`char` vector | `string`

Name of CAN device attached to the J1939 CAN channel, specified as a character vector or string.

For Kvaser and Vector products, `device` is a combination of the device type and a device index. For example, a Kvaser device might be `'USBcanProfessional 1'`; if you have two Vector CANcardXL devices, `device` can be `'CANcardXL 1'` or `'CANcardXL 2'`.

For National Instruments devices the `devicenumber` is the interface number defined in the NI Measurement & Automation Explorer.

For PEAK-System devices the `devicenumber` is the alphanumeric device number defined for the channel.

Example: `'Virtual 1'`

Data Types: `char` | `string`

### **chanIndex** — Channel number of CAN device

numeric

Channel number of the CAN device attached to the J1939 CAN channel, specified as a numeric value. Use this argument with Kvaser and Vector devices.

Example: 1

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

### **j1939Ch — J1939 CAN channel**

J1939 CAN channel object

J1939 CAN channel returned as a channel object.

## See Also

### **Functions**

canDatabase | j1939ParameterGroup | receive | transmit

### **Properties**

SilentMode | TransceiverState | UserData

## Topics

“J1939 Channel Workflow” on page 9-7

### **Introduced in R2015b**

## j1939ParameterGroup

Create J1939 parameter group

### Syntax

```
pg = j1939ParameterGroup(database, name)
```

### Description

`pg = j1939ParameterGroup(database, name)` creates a parameter group using the name defined in the specified database.

### Examples

#### Create a Parameter Group

This example shows how to attach a database to a parameter group name and view the signal information in the group.

Create a database handle.

```
db = canDatabase('C:\j1939Demo.dbc');
```

Create a parameter group.

```
pg = j1939ParameterGroup(db, 'PackedData')
```

```
pg =
```

```
ParameterGroup with properties:
```

```
Protocol Data Unit Details:
```

```
-----
```

```
      Name: 'PackedData'  
      PGN: 57344
```

```

        Priority: 6
        PDUFormatType: 'Peer-to-Peer (Type 1)'
        SourceAddress: 50
        DestinationAddress: 255

Data Details:
-----
        Timestamp: 0
        Data: [255 255 255 255 255 255 255 255]
        Signals: [1x1 struct]

Other Information:
-----
        UserData: []

```

Examine the signals in the parameter group.

```
pg.Signals
```

```
ans =
```

```

        ToggleSwitch: -1
        SliderSwitch: -1
        RockerSwitch: -1
        RepeatingStairs: 255
        PushButton: 1

```

## Input Arguments

### **database** — Handle to CAN database

CAN database object

Handle to CAN database, specified as a CAN database object. The specified database contains J1939 parameter group definitions.

Example: `db = canDatabase('C:\database.dbc')`

### **name** — Parameter group name

character vector | string

Parameter group name, specified as a character vector or string. The name must match the name specified in the attached CAN database.

Example: `'pgName'`

Data Types: `char` | `string`

### See Also

#### Functions

`canDatabase` | `j1939Channel`

#### Properties

`DestinationAddress` | `PDUFormatType` | `Priority` | `Signals` | `SourceAddress` | `UserData`

### Topics

“J1939 Interface” on page 9-2

“J1939 Parameter Group Format” on page 9-3

### Introduced in R2015b

# j1939ParameterGroupImport

Import J1939 log file

## Syntax

```
pgs = j1939ParameterGroupImport(file,vendor,database)
```

## Description

`pgs = j1939ParameterGroupImport(file,vendor,database)` reads the input file as a CAN message log file from the specified vendor. Using the specified CAN database, the CAN messages are converted into J1939 parameter groups, and assigns the output to the array `pgs`.

## Examples

### Import Log Data to J1939 Parameter Groups

Read a CAN message log file, and generate J1939 parameter groups according to a CAN database.

```
db = canDatabase('MyDatabase.dbc');  
pgs = j1939ParameterGroupImport('MsgLog.asc', 'Vector', db);
```

## Input Arguments

### **file** — CAN message log file

character vector | string

CAN message log file, specified as a character vector or string.

Example: 'MyDatabase.dbc'

Data Types: char | string

### **vendor — Vendor file format**

'Kvaser' | 'Vector'

Vendor file format, specified as a character vector or string. The supported file formats are those defined by Vector and Kvaser.

Example: 'Vector'

Data Types: char | string

### **database — CAN database**

database handle

CAN database, specified as a database handle.

## **Output Arguments**

### **pgs — J1939 parameter groups**

parameter group array

J1939 parameter groups, returned as a parameter group array.

## **See Also**

### **Functions**

canDatabase

**Introduced in R2017a**



# mdf

Access information contained in MDF file

## Syntax

```
mdfObj = mdf(mdfFileName)
```

## Description

`mdfObj = mdf(mdfFileName)` identifies a measurement data format (MDF) file and returns an MDF file object, which you can use to access information and data contained in the file. You can specify a full or partial path to the file.

## Examples

### Create MDF File Object for Specified MDF File

Create an MDF object for a given file, and view the object display.

```
mdfObj = mdf('MDFFile.mf4')
```

MDF with properties:

```
File Details
      Name: 'MDFFile.mf4'
      Path: 'c:\temp\MDFFile.mf4'
      Author: 'HOK'
      Department: 'Research'
      Project: 'MDF'
      Subject: 'CAN bus'
      Comment: 'This file contains CAN messages'
      Version: '4.10'
      DataSize: 32100
      InitialTimestamp: 2016-02-27 12:09:02
```

```

Creator Details
  ProgramIdentifier: 'mmdff.04'
    Creator: [1x1 struct]

File Contents
  Attachment: [1x1 struct]
  ChannelNames: {6x1 cell}
  ChannelGroup: [1x6 struct]

```

## Input Arguments

### **mdfFileName** — MDF file name

char vector | string

MDF file name, specified as a character vector or string, including the necessary full or relative path.

Example: 'MDFFile.mf4'

Data Types: char | string

## Output Arguments

### **mdfObj** — MDF file

MDF file object

MDF file, returned as an MDF file object. The object provides access to the MDF file information contained in the following properties.

Property	Description
Name	Name of the MDF file, including extension
Path	Full path to the MDF file, including file name
Author	Author who originated the MDF file
Department	Department that originated the MDF file
Project	Project that originated the MDF file
Subject	Subject matter in the MDF file

<b>Property</b>	<b>Description</b>
Comment	Open comment field from the MDF file
Version	MDF standard version of the file
DataSize	Total size of the data in the MDF file, in bytes
InitialTimestamp	Time when file data acquisition began in UTC or local time
ProgramIdentifier	Originating program of the MDF file
Creator	Structure containing details about creator of the MDF file, with these fields: VendorName, ToolName, ToolVersion, UserName, and Comment
Attachment	Structure of information about attachments contained within the MDF file, with these fields: Name, Path, Comment, Type, MIMETYPE, Size, EmbeddedSize, and MD5Checksum
ChannelNames	Cell array of the channel names in each channel group
ChannelGroup	Structure of information about channel groups contained within the MDF file, with these fields: AcquisitionName, Comment, NumSamples, DataSize, Sorted, and Channel

## See Also

### Functions

mdfVisualize | read | saveAttachment

### Topics

“Access MDF Files”

“Reading Data from MDF Files”

### Introduced in R2016b

## mdfDatastore

Datastore for collection of MDF files

### Description

Use the MDF datastore object to access data from a collection of MDF files.

### Creation

### Syntax

```
mdfds = mdfDatastore(location)
mdfds = mdfDatastore(__, 'Name1', Value1, 'Name2', Value2, ...)
```

### Description

`mdfds = mdfDatastore(location)` creates an `MDFDatastore` based on an MDF file or a collection of files in the folder specified by `location`. All files in the folder with extensions `.mdf`, `.dat`, or `.mf4` are included.

`mdfds = mdfDatastore(__, 'Name1', Value1, 'Name2', Value2, ...)` specifies function options and properties of `mdfds` using optional name-value pairs.

### Input Arguments

#### **location** — Location of MDF datastore files

character vector | cell array | `DsFileSet` object

Location of MDF datastore files, specified as a character vector, cell array of character vectors, or `matlab.io.datastore.DsFileSet` object identifying either files or folders. The path can be relative or absolute, and can contain the wildcard character `*`. If `location` specifies a folder, by default the datastore includes all files in that folder with the extensions `.mdf`, `.dat`, or `.mf4`.

Example: 'CANape.MF4'

Data Types: char | cell | DsFileSet

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments to set file information or object “Properties” on page 13-171. Allowed options are IncludeSubfolders, FileExtensions, and the properties ReadSize, SelectedChannelGroupNumber, and SelectedChannelNames.

Example: 'SelectedChannelNames','Counter\_B4'

### **IncludeSubfolders — Include files in subfolders**

false (default) | true

Include files in subfolders, specified as a logical. Specify true to include files in each folder and recursively in subfolders.

Example: 'IncludeSubfolders',true

Data Types: logical

### **FileExtensions — Custom extensions for filenames to include in MDF datastore**

{'.mdf','.dat','.mf4'} (default) | char | cell

Custom extensions for filenames to include in the MDF datastore, specified as a character vector or cell array of character vectors. By default, the supported extensions include .mdf, .dat, and .mf4. If your files have custom or nonstandard extensions, use this Name-Value setting to include files with those extensions.

Example: 'FileExtensions',{'myformat1','myformat2'}

Data Types: char | cell

## **Properties**

### **ChannelGroups — All channel groups present in first MDF file (read-only)**

table

All channel groups present in first MDF file, returned as a table.

Data Types: table

**Channels — All channels present in first MDF file (read-only)**

table

All channels present in first MDF file, returned as a table.

Data Types: table

**Files — Files included in datastore**

char | string | cell

Files included in the datastore, specified as a character vector, string, or cell array.

Example: {'file1.mf4', 'file2.mf4'}

Data Types: char | string | cell

**ReadSize — Size of data returned by read**

'file' (default) | numeric | duration

Size of data returned by the read function, specified as 'file', a numeric value, or a duration. A character vector value of 'file' causes the entire file to be read; a numeric double value specifies the number of records to read; and a duration value specifies a time range to read.

If you later change the ReadSize property value type, the datastore resets.

Example: 50

Data Types: double | char | duration

**SelectedChannelGroupNumber — Channel group to read**

numeric scalar

Channel group to read, specified as a numeric scalar value.

Example: 1

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**SelectedChannelNames — Names of channels to read**

char | string | cell

Names of channels to read, specified as a character vector, string, or cell array.

Example: 'Counter\_B4'

Data Types: char | string | cell

## Object Functions

read	Read data in MDF datastore
readall	Read all data in MDF datastore
preview	Subset of data from MDF datastore
reset	Reset MDF datastore to initial state
hasdata	Determine if data is available to read from MDF datastore
partition	Partition MDF datastore
numpartitions	Number of partitions for MDF datastore
combine (MATLAB)	Combine data from multiple datastores
transform (MATLAB)	Transform datastore

## Examples

### Create an MDF Datastore

Create an MDF datastore from the sample file `CANape.MF4`, and read it into a timetable.

```
mdfds = mdfDatastore(fullfile(matlabroot, 'examples', 'vnt', 'CANape.MF4'));  
while hasdata(mdfds)  
    m = read(mdfds);  
end
```

## See Also

### Topics

“Using MDF Files Via MDF Datastore”

**Introduced in R2017b**

## mdfVisualize

View channel data from MDF file

### Syntax

```
mdfVisualize(mdfFileName)
```

### Description

`mdfVisualize(mdfFileName)` opens an MDF file in the Simulation Data Inspector for viewing and interacting with channel data. `mdfFileName` is the name of the MDF file, specified as a full or partial path.

### Examples

#### View MDF Data

View the data from a specified MDF file in the Simulation Data Inspector.

```
mdfVisualize('File01.mf4')
```

### Input Arguments

#### **mdfFileName** — MDF file name

char vector | string

MDF file name, specified as a character vector or string, including the necessary full or relative path.

Example: 'MDFFile.mf4'

Data Types: char | string



## See Also

### Functions

mdf | read

### Topics

“View and Analyze Simulation Results” (Simulink)

**Introduced in R2019a**

## messageInfo

Information about CAN database messages

### Syntax

```
msgInfo = messageInfo(candb)
msgInfo = messageInfo(candb,msgName)
msgInfo = messageInfo(candb,id,msgIsExtended)
```

### Description

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

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

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

### Examples

#### Get All Messages

Get information from all messages in a CAN database.

```
candb = canDatabase('J1939DB.dbc');
msgInfo = messageInfo(candb)
```

```
msgInfo =
3x1 struct array with fields:
    Name
    Comment
    ID
```

```

Extended
J1939
Length
Signals
SignalInfo
TxNodes
Attributes
AttributeInfo

```

You can index into the structure for information on a particular message.

### Get One Message by Name

Get information from one message in a CAN database using the message name.

```

candb = canDatabase('J1939DB.dbc');
msgInfo = messageInfo(candb, 'A1')

msgInfo =
    Name: 'A1'
    Comment: 'This is an A1 message'
    ID: 419364350
    Extended: 1
    J1939: [1x1 struct]
    Length: 8
    Signals: {2x1 cell}
    SignalInfo: [2x1 struct]
    TxNodes: {'AerodynamicControl'}
    Attributes: {4x1 cell}
    AttributeInfo: [4x1 struct]

```

### Get One Message by ID

Get information from one message in a CAN database using the message ID.

```

candb = canDatabase('J1939DB.dbc');
msgInfo = messageInfo(candb, 419364350, true)

msgInfo =
    Name: 'A1'
    Comment: 'This is an A1 message'

```

```
        ID: 419364350
    Extended: 1
        J1939: [1x1 struct]
    Length: 8
    Signals: {2x1 cell}
    SignalInfo: [2x1 struct]
        TxNodes: {'AerodynamicControl'}
    Attributes: {4x1 cell}
    AttributeInfo: [4x1 struct]
```

## Input Arguments

### **candb** — CAN database

CAN database object

CAN database, specified as a CAN database object. **candb** identifies the database containing the CAN messages that you want information about.

Example: `candb = canDatabase(_____)`

### **msgName** — Message name

character vector | string

Message name, specified as a character vector or string. Provide the name of the message you want information about.

Example: `'A1'`

Data Types: `char` | `string`

### **id** — Message ID

numeric value

Message ID, specified as a numeric value. **id** is the numeric identifier of the specified message, in either extended or standard form.

Example: `419364350`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **msgIsExtended** — Message ID format

`true` | `false`

Message ID format, specified as a logical. Specify whether the message ID is in standard or extended type. Use the logical value `true` if extended, or `false` if standard. There is no default; you must provide this argument when using a message ID.

Example: `true`

Data Types: `logical`

## Output Arguments

### **msgInfo** — Message information

structure

Message information, returned as a structure or array of structures for the specified CAN database and messages.

## See Also

### **Functions**

`attributeInfo` | `canDatabase` | `canMessage` | `nodeInfo` | `signalInfo`

### **Properties**

`MessageInfo` | `Messages`

### **Introduced in R2009a**

## nodeInfo

Information about CAN database node

### Syntax

```
info = nodeInfo(db)
info = nodeInfo(db,NodeName)
```

### Description

`info = nodeInfo(db)` returns a structure containing information for all nodes found in the database `db`.

If no matches are found in the database, `nodeInfo` returns an empty node information structure.

`info = nodeInfo(db,NodeName)` returns a structure containing information for the specified node in the database `db`.

### Examples

#### View Information from All Nodes

Create a CAN database object, and view information about its nodes.

```
db = canDatabase('c:\Database.dbc')
info = nodeInfo(db)
```

```
info =
3x1 struct array with fields:
    Name
    Comment
    Attributes
    AttributeInfo
```

View name of first node.

```
n = info(1).Name

n =
AerodynamicControl
```

### View Information from One Node

Create a CAN database object, and view information about its first node, listed in the previous example.

```
db = canDatabase('c:\Database.dbc')
info = nodeInfo(db, 'AerodynamicControl')

info =
      Name: 'AerodynamicControl'
      Comment: 'This is an AerodynamicControl node'
      Attributes: {3x1 cell}
      AttributeInfo: [3x1 struct]
```

## Input Arguments

### db — CAN database

CAN database object

CAN database, specified as a CAN database object.

Example: db = canDatabase(\_\_\_\_\_) )

### nodeName — Node name

char vector | string

Node name, specified as a character vector or string.

Example: 'AerodynamicControl'

Data Types: char | string

## Output Arguments

### **info — Node information**

structure

Node information, returned as a structure with these fields:

<b>Field</b>	<b>Description</b>
Name	Node name
Comment	Text about node

## See Also

### **Functions**

[attributeInfo](#) | [canDatabase](#) | [messageInfo](#) | [signalInfo](#)

### **Properties**

[NodeInfo](#) | [Nodes](#)

**Introduced in R2015b**



## numpartitions (MDFDatastore)

Number of partitions for MDF datastore

### Syntax

```
N = numpartitions(mdfds)
N = numpartitions(mdfds,pool)
```

### Description

`N = numpartitions(mdfds)` returns the recommended number of partitions for the MDF datastore `mdfds`. Use the result as an input to the `partition` function.

`N = numpartitions(mdfds,pool)` returns a reasonable number of partitions to parallelize `mdfds` over the parallel pool, `pool`, based on the number of files in the datastore and the number of workers in the pool.

### Examples

#### Find Recommended Number of Partitions for MDF Datastore

Determine the number of partitions you should use for your MDF datastore.

```
mdfds = mdfDatastore(fullfile(matlabroot,'examples','vnt','CANape.MF4'));
N = numpartitions(mdfds);
```

### Input Arguments

#### **mdfds** — MDF datastore

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

**pool — Parallel pool**

parallel pool object

Parallel pool specified as a parallel pool object.

Example: `gcp`

## Output Arguments

**N — Number of partitions**

double

Number of partitions, returned as a double. This number is the calculated recommendation for the number of partitions for your MDF datastore. Use this when partitioning your datastore with the `partition` function.

## See Also

**Functions**

`mdfDatastore` | `partition` | `read` | `reset`

**Introduced in R2017b**

# pack

Pack signal data into CAN message

## Syntax

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

## Description

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

## Examples

### Pack a CAN Message

Pack a CAN message with a 16-bit integer value of 1000.

```
message = canMessage(500,false,8);  
pack(message,int16(1000),0,16,'LittleEndian')  
message.Data
```

1×8 uint8 row vector

```
232    3    0    0    0    0    0    0
```

Note that  $1000 = (3 \times 256) + 232$ .

Pack a CAN message with a double value of 3.14. A double requires 64 bits.

```
pack(message,3.14,0,64,'LittleEndian')
```

Pack a CAN message with a single value of -40. A single requires 32 bits.

```
pack(message, single(-40), 0, 32, 'LittleEndian')
```

## Input Arguments

### **message** — CAN message

CAN message object

CAN message, specified as a CAN message object.

Example: `canMessage`

### **value** — Value of signal to pack into message

numeric value

Value of signal to pack into message, specified as a numeric value. The value is assumed decimal, and distributed among the 8 bytes of the message Data property. You should convert the value into the data type expected for transmission.

Example: `int16(1000)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **startbit** — Signal starting bit in data

`single` | `double`

Signal starting bit in the data, specified as a single or double value. This is the least significant bit position in the signal data. Accepted values for `startbit` are from 0 through 63, inclusive.

Example: `0`

Data Types: `single` | `double`

### **signalsize** — Length of signal in bits

numeric value

Length of the signal in bits, specified as a numeric value. Accepted values for `signalsize` are from 1 through 64, inclusive.

Example: `16`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**byteorder — Signal byte order format**`'LittleEndian' | 'BigEndian'`

Signal byte order format, specified as 'LittleEndian' or 'BigEndian'.

Example: 'LittleEndian'

Data Types: char | string

## See Also

**Functions**

`canMessage` | `extractAll` | `extractRecent` | `extractTime` | `unpack`

**Introduced in R2009a**

## partition (MDFDatastore)

Partition MDF datastore

### Syntax

```
subds = partition(mdfds,N,index)

subds = partition(mdfds,'Files',index)
subds = partition(mdfds,'Files',filename)
```

### Description

`subds = partition(mdfds,N,index)` partitions the MDF datastore `mdfds` into the number of parts specified by `N`, and returns the partition corresponding to the index `index`.

`subds = partition(mdfds,'Files',index)` partitions the MDF datastore by files and returns the partition corresponding to the file of index `index` in the `Files` property.

`subds = partition(mdfds,'Files',filename)` partitions the datastore by files and returns the partition corresponding to the specified `filename`.

### Examples

#### Partition an MDF Datastore into Default Parts

Partition an MDF datastore from the sample file `CANape.MF4`, and return the first part.

```
mdfds = mdfDatastore(fullfile(matlabroot,'examples','vnt','CANape.MF4'));
N = numpartitions(mdfds);
subds1 = partition(mdfds,N,1);
```

## Partition an MDF Datastore by Its Files

Partition an MDF datastore according to its files, and return partitions by index and file name.

```
cd c:\temp
mdfds = mdfDatastore({'CANape1.MF4', 'CANape2.MF4', 'CANape3.MF4'});
mdfds.Files

ans =
    3x1 cell array
    'c:\temp\CANape1.MF4'
    'c:\temp\CANape2.MF4'
    'c:\temp\CANape3.MF4'

subds2 = partition(mdfds, 'files', 2);
subds3 = partition(mdfds, 'files', 'c:\temp\CANape3.MF4');
```

## Input Arguments

### **mdfds** — MDF datastore

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

### **N** — Number of partitions

positive integer

Number of partitions, specified as a double of positive integer value. Use the `numpartitions` function for the recommended number or partitions.

Example: `numpartitions(mdfds)`

Data Types: double

### **index** — Index

positive integer

Index, specified as a double of positive integer value. When using the 'files' partition scheme, this value corresponds to the index of the MDF datastore object `Files` property.

Example: 1

Data Types: double

### **filename — File name**

character vector

File name, specified as a character vector. The argument can specify a relative or absolute path.

Example: 'CANape.MF4'

Data Types: char

## Output Arguments

### **subds — MDF datastore partition**

MDF datastore object

MDF datastore partition, returned as an MDF datastore object. This output datastore is of the same type as the input datastore `mdfds`.

## See Also

### **Functions**

`mdfDatastore` | `numpartitions` | `read` | `reset`

**Introduced in R2017b**



# preview (MDFDatastore)

Subset of data from MDF datastore

## Syntax

```
data = preview(mdfds)
```

## Description

`data = preview(mdfds)` returns a subset of data from MDF datastore `mdfds` without changing the current position in the datastore.

## Examples

### Examine Preview of MDF Datastore

```
mdfds = mdfDatastore(fullfile(matlabroot,'examples','vnt','CANape.MF4'));
data = preview(mdfds)
```

```
data2 =
```

```
10×74 timetable
```

Time	Counter_B4	Counter_B5	Counter_B6	Counter_B7	PWM
0.00082554 sec	0	0	1	0	100
0.010826 sec	0	0	1	0	100
0.020826 sec	0	0	1	0	100
0.030826 sec	0	0	1	0	100
0.040826 sec	0	0	1	0	100
0.050826 sec	0	0	1	0	100

0.060826 sec	0	0	1	0	100
0.070826 sec	0	0	1	0	100

## Input Arguments

### **mdfds — MDF datastore**

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

## Output Arguments

### **data — Subset of data**

timetable

Subset of data, returned as a timetable of MDF records.

## See Also

### **Functions**

`hasdata` | `mdfDatastore` | `read`

### **Introduced in R2017b**

## read

Read channel data from MDF file

### Syntax

```
data = read(mdfObj)
data = read(mdfObj,chanGroupIndex,chanName)
data = read(mdfObj,chanGroupIndex,chanName,startPosition)
data = read(mdfObj,chanGroupIndex,chanName,startPosition,
endPosition)
data = read(mdfObj,chanGroupIndex,chanName,startPosition,
endPosition,'OutputFormat',fmtType)
[data,time] = read(mdfObj,chanGroupIndex,chanName,startPosition,
endPosition,'OutputFormat','Vector')
```

### Description

`data = read(mdfObj)` reads all data for all channels from the MDF file identified by the MDF file object `mdfObj`, and assigns the output to `data`. If the file data is one channel group, the output is a timetable; multiple channel groups are returned as a cell array of timetables, where the cell array index corresponds to the channel group number.

`data = read(mdfObj,chanGroupIndex,chanName)` reads all data for the specified channel from the MDF file identified by the MDF file object `mdfObj`.

`data = read(mdfObj,chanGroupIndex,chanName,startPosition)` reads data from the position specified by `startPosition`.

`data = read(mdfObj,chanGroupIndex,chanName,startPosition, endPosition)` reads data for the range specified from `startPosition` to `endPosition`.

`data = read(mdfObj,chanGroupIndex,chanName,startPosition, endPosition,'OutputFormat',fmtType)` returns data with the specified output format.

`[data,time] = read(mdfObj,chanGroupIndex,chanName,startPosition,endPosition,'OutputFormat','Vector')` returns two vectors of channel data and corresponding timestamps.

## Examples

### Read All Data from MDF File

Read all available data from the MDF file.

```
mdfObj = mdf('MDFFile.mf4');  
data = read(mdfObj);
```

### Read All Data from Multiple Channels

Read all available data from the MDF file for specified channels.

```
mdfObj = mdf('MDFFile.mf4');  
data = read(mdfObj,1,{'Channel1','Channel2'});
```

### Read Range of Data from Specified Index Values

Read a range of data from the MDF file using indexing for `startPosition` and `endPosition` to specify the data range.

```
mdfObj = mdf('MDFFile.mf4');  
data = read(mdfObj,1,{'Channel1','Channel2'},1,10);
```

### Read Range of Data from Specified Time Values

Read a range of data from the MDF file using time values for `startPosition` and `endPosition` to specify the data range.

```
mdfObj = mdf('MDFFile.mf4');
data = read(mdfObj,1,{ 'Channel1' , 'Channel2'},seconds(5.5),seconds(7.3));
```

### Read All Data in Vector Format

Read all available data from the MDF file, returning data and time vectors.

```
mdfObj = mdf('MDFFile.mf4');
[data,time] = read(mdfObj,1,'Channel1','OutputFormat','Vector');
```

### Read All Data in Time Series Format

Read all available data from the MDF file, returning time series data.

```
mdfObj = mdf('MDFFile.mf4');
data = read(mdfObj,1,'Channel1','OutputFormat','TimeSeries');
```

### Read Data from Channel List Entry

Read data from a channel identified by the channelList function.

Get list of channels and display their names and group numbers.

```
mdfObj = mdf('File05.mf4');
chlist = channelList(mdfObj);
chlist(:,1:2)
```

4×2 table

ChannelName	ChannelGroupNumber
"Float_32_LE_Offset_64"	2
"Float_64_LE_Master_Offset_0"	2

Read data from the first channel in the list.

```
data = read(mdfObj,chlist{1,2},chlist{1,1});
data(1:5,:)
```

5×1 timetable

Time	Float_32_LE_Offset_64
0 sec	5
0.01 sec	5.1
0.02 sec	5.2
0.03 sec	5.3
0.04 sec	5.4

## Input Arguments

### **mdfObj** — MDF file

MDF file object

MDF file, specified as an MDF file object.

Example: `mdf('MDFFile.mf4')`

### **chanGroupIndex** — Index of the channel group

numeric value

Index of channel group, specified as a numeric value that identifies the channel group from which to read.

Example: 1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **chanName** — Name of channel

char vector | string

Name of channel, specified as a character vector, string, or array. `chanName` identifies the name of a channel in the channel group. Use a cell array of character vectors or array of string to identify multiple channels.

Example: `'Channel1'`

Data Types: `char` | `string` | `cell`

### **startPosition** — First position of channel data

numeric value | duration

First position of channel data, specified as a numeric value or duration. The `startPosition` option specifies the first position from which to read channel data. Provide a numeric value to specify an index position; use a duration to specify a time position. If only `startPosition` is provided without the `endPosition` option, the data value at that location is returned. When used with `endPosition` to specify a range, the function returns data from the `startPosition` (inclusive) to the `endPosition` (noninclusive).

Example: 1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

### **endPosition — Last position of channel data range**

numeric value | duration

Last position of channel data range, specified as a numeric value or duration. The `endPosition` option specifies the last position for reading a range of channel data. Provide both the `startPosition` and `endPosition` to specify retrieval of a range of data. The function returns up to but not including `endPosition` when reading a range. Provide a numeric value to specify an index position; use a duration to specify a time position.

Example: 1000

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

### **fmtType — Format for output data**

'Timetable' (default) | 'Vector' | 'TimeSeries'

Format for output data, specified as a character vector or string. This option formats the output according to the following table.

OutputFormat	Description
'Timetable'	<p>Return a timetable from one or more channels into one output variable. This is the only format allowed when reading from multiple channels at the same time. (Default.)</p> <p>Note: The timetable format includes columns for the MDF channels. Because the column titles must be valid MATLAB identifiers, they might not be exactly the same as those values in the MDF object <code>ChannelNames</code> property. The column headers are derived from the property using the function <code>matlab.lang.makeValidName</code>. The original channel names are available in the <code>VariableDescriptions</code> property of the <code>timetable</code> object.</p>
'Vector'	Return a vector of numeric data values, and optionally a vector of time values from one channel. Use one output variable to return only data, or two output variables to return both data and time vectors.
'TimeSeries'	Return a time series of data from one channel.

Example: 'Vector'

Data Types: char | string

## Output Arguments

### data — Channel data

timetable (default) | double | time series | cell array

Channel data, returned as vector of doubles, a time series, a timetable, or cell array of timetables, according to the 'OutputFormat' option setting and the number of channel groups.

### time — Channel data times

double

Channel data times, returned as a vector of double elements. The time vector is returned only when the 'OutputFormat' is set to 'Vector'.



## See Also

### Functions

`mdf` | `mdfVisualize` | `saveAttachment`

### Topics

["Access MDF Files"](#)

["Reading Data from MDF Files"](#)

["Time Series" \(MATLAB\)](#)

["Represent Dates and Times in MATLAB" \(MATLAB\)](#)

["Tables" \(MATLAB\)](#)

### Introduced in R2016b

## read (MDFDatastore)

Read data in MDF datastore

### Syntax

```
data = read(mdfds)
[data,info] = read(mdfds)
```

### Description

`data = read(mdfds)` returns data from the MDF datastore `mdfds` into the timetable `data`.

The `read` function returns a subset of data from the datastore. The size of the subset is determined by the `ReadSize` property of the datastore object. On the first call, `read` starts reading from the beginning of the datastore, and subsequent calls continue reading from the endpoint of the previous call. Use `reset` to read from the beginning again.

`[data,info] = read(mdfds)` also returns to the output argument `info` information, including metadata, about the extracted data.

### Examples

#### Read Datastore by Files

Read data from an MDF datastore one file at a time.

```
mdfds = mdfDatastore({'CANape1.MF4','CANape2.MF4','CANape3.MF4'});
mdfds.ReadSize = 'file';
data = read(mdfds);
```

Read the second file and view information about the data.

```
[data2,info2] = read(mdfds);
info2
```

```
struct with fields:
    Filename: 'CANape2.MF4'
    FileSize: 57592
    MDFFileProperties: [1x1 struct]
```

## Input Arguments

### **mdfds** — MDF datastore

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

## Output Arguments

### **data** — Output data

timetable

Output data, returned as a timetable of MDF records.

### **info** — Information about data

structure array

Information about data, returned as a structure array with the following fields:

```
Filename
FileSize
MDFFileProperties
```

## See Also

### **Functions**

`hasdata` | `mdfDatastore` | `preview` | `readall` | `reset`

### **Topics**

“Using MDF Files Via MDF Datastore”

**Introduced in R2017b**

## readall (MDFDatastore)

Read all data in MDF datastore

### Syntax

```
data = readall(mdfds)
```

### Description

`data = readall(mdfds)` reads all the data in the datastore specified by `mdfds` and returns it to timetable `data`.

After the `readall` function returns all the data, it resets `mdfds` to point to the beginning of the datastore.

If all the data in the datastore does not fit in memory, then `readall` returns an error.

### Examples

#### Read All Data in Datastore

Read all the data from a multiple file MDF datastore into a timetable.

```
mdfds = mdfDatastore({'CANape1.MF4', 'CANape2.MF4', 'CANape3.MF4'});  
data = readall(mdfds);
```

### Input Arguments

#### **mdfds** — MDF datastore

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

## Output Arguments

### **data — Output data**

timetable

Output data, returned as a timetable of MDF records.

## See Also

### **Functions**

hasdata | mdfDatastore | preview | read | reset

### **Topics**

“Using MDF Files Via MDF Datastore”

**Introduced in R2017b**

## readAxis

Read and scale specified axis value from direct memory

### Syntax

```
value = readAxis(chanObj,axis)
```

### Description

`value = readAxis(chanObj,axis)` reads and scales a value for the specified axis through the XCP channel object `chanObj`. This action performs a direct read from memory on the slave module.

### Examples

#### Read Value from XCP Channel Axis

Read the value from an XCP channel axis, identifying the axis by name.

```
a2lObj = xcpA2L('myA2Lfile.a2l');  
chanObj = xcpChannel(a2lObj,'CAN','Vector','Virtual 1',1);  
connect(chanObj);  
value = readAxis(chanObj,'pedal_position');
```

Alternatively, create an axis object and read its value.

```
axisObj = a2lObj.AxisXs('pedal_position');  
value = readAxis(chanObj,axisObj);
```

### Input Arguments

#### **chanObj** — XCP channel

channel object

XCP channel, specified as an XCP channel object.

Example: `xcpChannel()`

### **axis — XCP channel axis**

axis object | char

XCP channel axis, specified as a character vector or axis object.

Example: `'pedal_position'`

Data Types: char

## Output Arguments

### **value — Value from axis read**

axis value

Value from axis read, returned as type supported by the axis.

## See Also

### **Functions**

`readCharacteristic` | `readMeasurement` | `writeAxis` | `writeCharacteristic` | `writeMeasurement`

### **Introduced in R2018a**



# readCharacteristic

Read and scale specified axis value from direct memory

## Syntax

```
value = readCharacteristic(chanObj,characteristic)
```

## Description

`value = readCharacteristic(chanObj,characteristic)` reads and scales a value for the specified `characteristic` through the XCP channel object `chanObj`. This action performs a direct read from memory on the slave module.

## Examples

### Read Value from XCP Channel Characteristic

Read the value from an XCP channel characteristic, identifying the characteristic by name.

```
a2lObj = xcpA2L('myA2Lfile.a2l');  
chanObj = xcpChannel(a2lObj,'CAN','Vector','Virtual 1',1);  
connect(chanObj);  
value = readCharacteristic(chanObj,'torque_demand');
```

Alternatively, create a characteristic object and read its value.

```
charObj = a2lObj.CharacteristicInfo('torque_demand');  
value = readCharacteristic(chanObj, charObj);
```

## Input Arguments

### **chanObj** — XCP channel

channel object

XCP channel, specified as an XCP channel object.

Example: `xcpChannel()`

### **characteristic** — XCP channel characteristic

characteristic object | char

XCP channel characteristic, specified as a character vector or characteristic object.

Example: `'torque_demand'`

Data Types: char

## Output Arguments

### **value** — Value from characteristic read

characteristic value

Value from characteristic read, returned as a type supported by the characteristic.

## See Also

### **Functions**

`readAxis` | `readMeasurement` | `writeAxis` | `writeCharacteristic` | `writeMeasurement`

**Introduced in R2018a**

## readDAQ

Read scaled samples of specified measurement from DAQ list

### Syntax

```
value = readDAQ(xcpch,measurementName)
value = readDAQ(xcpch,measurementName,count)
```

### Description

`value = readDAQ(xcpch,measurementName)` reads and scales all acquired DAQ list data from the XCP channel object `xcpch`, for the specified `measurementName`, and stores the results in the variable `value`. If the measurement has no data, the function returns an empty value.

`value = readDAQ(xcpch,measurementName,count)` reads the quantity of data specified by `count`. If fewer than `count` samples are available, it returns only those.

### Examples

#### Acquire Data from DAQ List

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up a DAQ measurement list and acquire 10 data values, then all data.

```
a2lObj = xcpA2L('myFile.a2l');
channelObj = xcpChannel(a2lObj, 'CAN', 'Vector', 'CANcaseXL 1', 1);
connect(channelObj);
createMeasurementList(channelObj, 'DAQ', 'Event1', 'Measurement1');
startMeasurement(channelObj);
```

```
data = readDAQ(channelObj, 'Measurement1', 10);  
data_all = readDAQ(channelObj, 'Measurement1');
```

## Input Arguments

### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

### **measurementName** — Name of single XCP measurement

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure `measurementName` matches the corresponding measurement name defined in your A2L file.

Data Types: `char` | `string`

### **count** — Number of samples to read

numeric value

Number of samples to read, specified as a numeric value, for the specified measurement name. If the number of samples in the measurement is less than the specified count, only the available number of samples are returned.

## Output Arguments

### **value** — Values from specified measurement

numeric array

Values from the specified measurement, returned as a numeric array.

## See Also

`readSingleValue`

**Introduced in R2018b**

## readDAQListData

Read samples of specified measurement from DAQ list

### Syntax

```
value = readDAQListData(xcpch,measurementName)
value = readDAQListData(xcpch,measurementName,count)
```

### Description

`value = readDAQListData(xcpch,measurementName)` reads all acquired DAQ list data from the XCP channel object `xcpch`, for the specified `measurementName`, and stores the results in the variable `value`. If the measurement has no data, the function returns an empty value.

`value = readDAQListData(xcpch,measurementName,count)` reads the quantity of data specified by `count`. If fewer than `count` samples are available, it returns only those.

### Examples

#### Acquire Data for Triangle Measurement in a DAQ List

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up a DAQ measurement list and acquire data from a '100ms' events 'Triangle' measurement.

Create an object to parse an A2L file and connect that to an XCP channel.

```
a2lfile = xcp.A2L('XCPSIM.a2l')
xcpch = xcp.Channel(a2lfile,'CAN','Vector','Virtual 1',1);
```

Connect the channel to the slave.

```
connect(xcpch)
```

Create a measurement list with a '100ms' event and 'PMW', 'PwmFiltered', and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'DAQ', '100ms', {'PMW', 'PwmFiltered', 'Triangle'})
```

Start the measurement.

```
startMeasurement(xcpch)
```

Acquire data for the 'Triangle' measurement for 5 counts.

```
value = readDAQListData(xcpch, 'Triangle', 5)
```

```
value =
```

```
    -50    -50    -50    -50    -50
```

## Input Arguments

### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

### **measurementName** — Name of single XCP measurement

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure `measurementName` matches the corresponding measurement name defined in your A2L file.

Data Types: `char` | `string`

### **count** — Number of samples to read

numeric value

Number of samples to read, specified as a numeric value, for the specified measurement name. If the number of samples in the measurement is less than the specified count, only the available number of samples are returned.

## Output Arguments

**value** — Values from specified measurement

numeric array

Values from the specified measurement, returned as a numeric array.

## See Also

`readSingleValue`

## Topics

“Acquire Measurement Data via Dynamic DAQ Lists” on page 8-9

**Introduced in R2013a**



# readMeasurement

Read and scale specified measurement value from direct memory

## Syntax

```
value = readMeasurement(chanObj,measurement)
```

## Description

`value = readMeasurement(chanObj,measurement)` reads and scales a value for the specified measurement through the XCP channel object `chanObj`. This action performs a direct read from memory on the slave module.

## Examples

### Read Value from XCP Channel Measurement

Read the value from an XCP channel measurement, identifying the measurement by name.

```
a2lObj = xcpA2L('myA2Lfile.a2l');  
chanObj = xcpChannel(a2lObj, 'CAN', 'Vector', 'Virtual 1', 1);  
connect(chanObj);  
value = readMeasurement(chanObj, 'limit')
```

```
100
```

Alternatively, create a measurement object and read its value.

```
measObj = a2lObj.MeasurementInfo('limit');  
value = readMeasurement(chanObj, measObj)
```

100

## Input Arguments

**chanObj** — XCP channel

channel object

XCP channel, specified as an XCP channel object.

Example: `xcpChannel()`**measurement** — XCP channel measurement

measurement object | char

XCP channel measurement, specified as a character vector or measurement object.

Example: `'limit'`

Data Types: char

## Output Arguments

**value** — Value from measurement read

measurement value

Value from measurement read, returned as a type supported by the measurement.

## See Also

**Functions**`readAxis` | `readCharacteristic` | `writeAxis` | `writeCharacteristic` | `writeMeasurement`**Introduced in R2018a**

# readSingleValue

Read single sample of specified measurement from memory

## Syntax

```
value = readSingleValue(xcpch, 'measurementName')
```

## Description

`value = readSingleValue(xcpch, 'measurementName')` acquires a single value for the specified measurement through the configured XCP channel and stores it in a variable for later use. The values are read directly from memory.

## Examples

### Acquire a Single Value for Triangle Measurement

Read a single value from a '100ms' events 'Triangle' measurement.

Create an object to parse an A2L file and connect that to an XCP channel.

```
a2lfile = xcpA2L('XCPSIM.a2l')  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Acquire data for the 'Triangle' measurement.

```
value = readSingleValue(xcpch, 'Triangle')
```

value =

14

## Input Arguments

### **xcpch — XCP channel**

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

### **measurementName — Name of single XCP measurement**

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure `measurementName` matches the corresponding measurement name defined in your A2L file.

Data Types: `char` | `string`

## Output Arguments

### **value — Value of the measurement**

numeric value

Value of the selected measurement, returned as a numeric value.

## See Also

`readDAQListData`

**Introduced in R2013a**

## receive

Receive messages from CAN bus

### Syntax

```
message = receive(canch,  
messagesrequested, 'OutputFormat', 'timetable')  
message = receive(canch, messagesrequested)
```

### Description

`message = receive(canch, messagesrequested, 'OutputFormat', 'timetable')` returns a timetable of CAN messages 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 `Inf`, the function returns all available messages.

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

Specifying the `'OutputFormat'` option value of `'timetable'` results in a timetable of messages. This output format is recommended for optimal performance and representation of CAN messages within MATLAB.

`message = receive(canch, messagesrequested)` returns an array of CAN message objects instead of a timetable if the channel `ProtocolMode` is `'CAN'`.

---

**Note** If the channel `ProtocolMode` is `'CAN FD'` the `receive` function returns a timetable, whether you specify an `'OutputFormat'` or not.

---

### Examples

## Receive CAN Messages

You can receive CAN messages as a timetable or as an array of message objects.

Receive all available messages as a timetable.

```
canch = canChannel('Vector', 'CANCaseXL 1', 1)
start(canch)
message = receive(canch, Inf, 'OutputFormat', 'timetable')
```

Receive up to five messages as an array of message objects.

```
message = receive(canch, 5)
```

## Input Arguments

### **canch** — CAN channel

CAN channel object

CAN channel, specified as a CAN channel object. This is the channel by which you access the CAN bus.

Example: `canChannel`

### **messagesrequested** — Maximum number of messages to receive

numeric value | `Inf`

Maximum number of messages to receive, specified as a positive numeric value or `Inf`.

Example: `Inf`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **message** — CAN messages

timetable | CAN message object array

CAN messages from the channel, returned as a timetable of messages or an array of CAN message objects.

## See Also

### Functions

canChannel | canMessage | transmit

**Introduced in R2009a**

## receive (J1939)

Receive parameter groups from J1939 bus

### Syntax

```
pgrp = receive(chan,count)
```

### Description

`pgrp = receive(chan, count)` receives parameter groups from the bus via channel `chan`. The number of received parameter groups is limited to the value of `count`.

### Examples

#### Receive Parameter Groups from Bus

Receive all the available parameter groups from the bus by specifying a count of `Inf`.

```
db = canDatabase('MyDatabase.dbc')
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1)
start(chan)
pgrp = receive(chan, Inf)
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

#### **count** — Maximum number of parameter groups

double



Maximum number of parameter groups to receive, specified as a double. `count` must be a positive value, or `Inf` to specify all available parameter groups.

Data Types: `double`

## Output Arguments

**pgrp** — J1939 parameter groups

array of `ParameterGroup` objects

J1939 parameter groups, returned as an array of `ParameterGroup` objects.

## See Also

### Functions

`j1939Channel` | `start` | `transmit`

**Introduced in R2015b**

## replay

Retransmit messages from CAN bus

## Syntax

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

## Description

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

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

## Examples

### Replay Messages on CAN Channel

Use a loopback connection between two channels, so that:

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

The timestamp differentials between messages in the two receive arrays, `msgRx1` and `msgRx2`, 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);  
  
% The first channel transmits messages 2 seconds apart.
```

```
transmit(ch1,msgTx1)
pause(2)
transmit(ch1,msgTx2)
%The second channel receives them
msgRx1 = receive(ch2,Inf);

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

## Input Arguments

### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, on which to retransmit.

Example: `canChannel('NI','CAN1')`

### **message** — Messages to replay

array of message objects

Messages to replay, specified as an array of message objects.

## See Also

### Functions

`canMessage` | `transmit`

### Introduced in R2009a

## reset (MDFDatastore)

Reset MDF datastore to initial state

### Syntax

```
reset(mdfds)
```

### Description

`reset(mdfds)` resets the MDF datastore specified by `mdfds` to its initial read state, where no data has been read from it. Resetting allows you to reread from the same datastore.

### Examples

#### Reset MDF Datastore

Reset an MDF datastore so that you can read from it again.

```
mdfds = mdfDatastore(fullfile(matlabroot,'examples','vnt','CANape.MF4'));  
data = read(mdfds);  
reset(mdfds);  
data = read(mdfds);
```

### Input Arguments

#### **mdfds** — MDF datastore

MDF datastore object

MDF datastore, specified as an MDF datastore object.

Example: `mdfds = mdfDatastore('CANape.MF4')`

## See Also

### Functions

hasdata | mdfDatastore | read

**Introduced in R2017b**

## saveAttachment

Save attachment from MDF file

### Syntax

```
saveAttachment(mdfObj,AttachmentName)
saveAttachment(mdfObj,AttachmentName,DestFile)
```

### Description

`saveAttachment(mdfObj,AttachmentName)` saves the specified attachment from the MDF file to the current MATLAB working folder. The attachment is saved with its existing name.

`saveAttachment(mdfObj,AttachmentName,DestFile)` saves the specified attachment from the MDF file to the given destination. You can specify relative or absolute paths to place the attachment in a specific folder.

### Examples

#### Save Attachment with Original Name

Save an MDF file attachment with its original name in the current folder.

```
mdfObj = mdf('MDFFile.mf4');
saveAttachment(mdfObj,'AttachmentName.ext')
```

#### Save Attachment with New Name

Save an MDF file attachment with a new name in the current folder.

```
mdfObj = mdf('MDFFile.mf4');  
saveAttachment(mdfObj, 'AttachmentName.ext', 'MyFile.ext')
```

### Save Attachment in Parent Folder

Save an MDF file attachment in a folder specified with a relative path name, in this case in the parent of the current folder.

```
mdfObj = mdf('MDFFile.mf4');  
saveAttachment(mdfObj, 'AttachmentName.ext', '..\MyFile.ext')
```

### Save Attachment in Specified Folder

This example saves an MDF file attachment using an absolute path name.

```
mdfObj = mdf('MDFFile.mf4');  
saveAttachment(mdfObj, 'AttachmentName.ext', 'C:\MyDir\MyFile.ext')
```

## Input Arguments

### **mdfObj** — MDF file

MDF file object

MDF file, specified as an MDF file object.

Example: `mdf('MDFFile.mf4')`

### **AttachmentName** — MDF file attachment name

char vector | string

MDF file attachment name, specified as a character vector or string. The name of the attachment is available in the Name field of the MDF file object Attachment property.

Example: `'file1.dbc'`

Data Types: char | string

### **DestFile** — Destination file name for the saved attachment

existing attachment name (default) | char vector | string

Destination file name for the saved attachment, specified as a character vector or string. The specified destination can include an absolute or relative path, otherwise the attachment is saved in the current folder.

Example: `'MyFile.ext'`

Data Types: `char` | `string`

## See Also

### Functions

`mdf` | `read`

**Introduced in R2016b**



# setValue

Set instance value in CDFX object

## Syntax

```
setValue(cdfxObj, instName, iVal)  
setValue(cdfxObj, instName, sysName, iVal)
```

## Description

setValue(cdfxObj, instName, iVal) sets the value of the unique instance whose ShortName is specified by instName to iVal. If multiple instances share the same ShortName, the function returns an error.

setValue(cdfxObj, instName, sysName, iVal) sets the value of the instance whose ShortName is specified by instName and is contained in the system specified by sysName.

---

**Note** setValue does not write the instance value in the original CDFX-file. Use the write function to update the CDFX-file or to create a new file.

---

## Examples

### Set Value of Instance

Create an asam.cdfx object and set the value of its VALUE\_NUMERIC instance.

```
cdfxObj = cdfx('c:\DataFiles\AllCategories_VCD.cdfx');  
setValue(cdfxObj, 'VALUE_NUMERIC', 55)
```

Read back the value to verify it.

```
iVal = getValue(cdfxObj, 'VALUE_NUMERIC')
```

```
iVal =  
    55
```

## Input Arguments

### **cdfxObj** — CDFX-file object

asam.cdfx object

CDFX-file object, specified as an `asam.cdfx` object. Use the object to access the calibration data.

Example: `cdfx()`

### **instName** — Instance name

char | string

Instance name, specified as a character vector or string.

Example: `'NUMERIC_VALUE'`

Data Types: char | string

### **sysName** — Parent system name

char | string

Parent system name, specified as a character vector or string.

Example: `'System2'`

Data Types: char | string

### **iVal** — Instance value

instance type

Instance value, specified as the type supported by the instance.

Example: `55`

## See Also

### Functions

`cdfx` | `getValue` | `instanceList` | `systemList` | `write`

**Introduced in R2019a**

## signalInfo

Information about signals in CAN message

### Syntax

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

### 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

#### Use Message Name to Get Information

Get signal information from the message 'Battery\_Voltage'.

```
SigInfo = signalInfo(candb,'Battery_Voltage');
```

#### Use Message ID to Get Information

Get signal information from the message with ID 196608.

```
SigInfo = signalInfo(candb,196608,true);
```

### Use Signal Name to Get Information

Get information from the signal named 'BatVlt' from message 196608.

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

## Input Arguments

### **candb** — CAN database

CAN database object

CAN database, specified as a CAN database object, that contains the signals that you want information about.

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

### **msgName** — Message name

character vector | string

Message name, specified as a character vector or string. Provide the name of the message that contains the signals that you want information about.

Example: `'Battery_Voltage'`

Data Types: `char` | `string`

### **id** — Message identifier

numeric value

Message identifier, specified as a numeric value. Provide the numeric identifier of the specified message that contains the signals you want information about.

Example: `196608`

### **extended** — Extended message indicator

`true` | `false`

Extended message indicator, specified as `true` or `false`. Indicate whether the message ID is standard or extended type. Use the logical value `true` if extended, or `false` if standard.

Example: true

Data Types: logical

**signalName — Name of signal**

char vector | string

Name of the signal, specified as a character vector or string. Provide the name of the specific signal that you want information about.

Example: 'BatVlt'

Data Types: char | string

## Output Arguments

**SigInfo — Signal information**

struct or array of struct

Signal information, returned as a structure or array of structures.

Data Types: struct

## See Also

**Functions**

canDatabase | canMessage | messageInfo

**Properties**

MessageInfo | Messages

**Introduced in R2009a**

---

## start

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 until:

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

---

**Note** Before you can start a channel to transmit or receive CAN FD messages, you must configure its bus speed with `configBusSpeed`.

---

## Examples

### Start a CAN Channel

Start a virtual device CAN channel.

```
canch = canChannel('MathWorks','Virtual 1',1);  
start(canch)
```

## Input Arguments

### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, that you want to start.

Example: `canChannel('NI', 'CAN1')`

## See Also

### Functions

`canChannel` | `configBusSpeed` | `stop`

**Introduced in R2009a**



## start (J1939)

Start channel connection to J1939 bus

### Syntax

```
start(chan)
```

### Description

`start(chan)` activates the channel `chan` on a J1939 bus. The channel remains activated until `stop` is called or it is cleared from the memory.

### Examples

#### Start J1939 Channel

Activate a channel on a J1939 bus.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);  
start(chan)
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.

## See Also

### Functions

`j1939Channel` | `stop`

**Introduced in R2015b**

# startMeasurement

Start configured DAQ and STIM lists

## Syntax

```
startMeasurement(xcpch)
```

## Description

`startMeasurement(xcpch)` starts all configured data acquisition and stimulation lists on the specified XCP channel. When you start the measurement, configured DAQ lists begin acquiring data values from the slave module and STIM lists begin transmitting data values to the slave model.

## Examples

### Start a DAQ Measurement

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up a DAQ measurement list and start measuring data.

```
a2l = xcpA2L('XCPSIM.a2l')  
xcpch = xcpChannel(a2l, 'CAN', 'Vector', 'Virtual 1', 1),
```

```
xcpch =
```

```
Channel with properties:
```

```
    SlaveName: 'CPP'  
    A2LFileName: 'XCPSIM.a2l'  
    TransportLayer: 'CAN'  
    TransportLayerDevice: [1x1 struct]  
    SeedKeyCallbackFcn: []  
    KeyValue: []
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data acquisition measurement list with the '10 ms' event and 'BitSlice' measurement.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', 'BitSlice')
```

Start your measurement.

```
startMeasurement(xcpch);
```

### **Start a STIM Measurement**

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up a DAQ measurement list and start measuring data.

```
a2l = xcpA2L('XCPSIM.a2l')
xcpch = xcpChannel(a2l, 'CAN', 'Vector', 'Virtual 1', 1)
xcpch =
```

Channel with properties:

```
    SlaveName: 'CPP'
    A2LFileName: 'XCPSIM.a2l'
    TransportLayer: 'CAN'
    TransportLayerDevice: [1x1 struct]
    SeedKeyCallbackFcn: []
    KeyValue: []
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data stimulation measurement list with the '100ms' event and 'BitSlice0', 'PWMFiltered', and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'STIM', '100ms', {'BitSlice0', 'PWMFiltered', 'Triangle'})
```

Start your measurement.

```
startMeasurement(xcpch);
```

## Input Arguments

### **xcpch — XCP channel**

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

`stopMeasurement` | `xcpChannel`

**Introduced in R2013a**

## stop

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

### Stop a CAN Channel

Stop a virtual device CAN channel.

```
canch = canChannel('MathWorks','Virtual 1',1);  
start(canch)  
stop(canch)
```

## Input Arguments

### **canch** — CAN device channel

CAN channel object

CAN device channel, specified as a CAN channel object, that you want to stop.

Example: `canChannel('NI','CAN1')`

## **See Also**

canChannel | start

**Introduced in R2009a**

## stop (J1939)

Stop channel connection to J1939 bus

### Syntax

```
stop(chan)
```

### Description

`stop(chan)` deactivates the channel `chan` on a J1939 bus. The channel also deactivates when it is cleared from the memory.

### Examples

#### Stop J1939 Channel

Deactivate a channel on a J1939 bus.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);  
start(chan)
```

```
stop(chan)
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.



## See Also

### Functions

j1939Channel | start

**Introduced in R2015b**

## stopMeasurement

Stop configured DAQ and STIM lists

### Syntax

```
stopMeasurement(xcpch)
```

### Description

`stopMeasurement(xcpch)` stops all configured data acquisition and stimulation lists on the specified XCP channel. When you stop the measurement, configured DAQ lists stop acquiring data values from the slave module and STIM lists stop transmitting data values to the slave model.

### Examples

#### Stop a DAQ Measurement

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up a DAQ measurement list and start and stop measuring data.

```
a2l = xcp2L('XCPSIM.a2l')  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1)
```

```
xcpch =
```

```
Channel with properties:
```

```
    SlaveName: 'CPP'  
    A2LFileName: 'XCPSIM.a2l'  
    TransportLayer: 'CAN'  
    TransportLayerDevice: [1x1 struct]  
    SeedKeyCallbackFcn: []  
    KeyValue: []
```

Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data acquisition measurement list with the '10 ms' event and 'BitSlice' measurement and start your measurement.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', 'BitSlice')  
startMeasurement(xcpch);
```

Stop your measurement.

```
stopMeasurement(xcpch);
```

## Input Arguments

### **xcpch — XCP channel**

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

`startMeasurement` | `xcpChannel`

**Introduced in R2013a**

## systemList

ECU systems in the CDFX object

### Syntax

```
sList = systemList(cdfxObj)
sList = systemList(cdfxObj, sysName)
```

### Description

`sList = systemList(cdfxObj)` returns a table listing every electronic control unit (ECU) system in the CDFX object.

`sList = systemList(cdfxObj, sysName)` returns a table listing every ECU system in the CDFX object whose `ShortName` matches `SysName`.

### Examples

#### View CDFX Object Systems

Create an `asam.cdfx` object and view its ECU systems.

List all systems.

```
cdfxObj = cdfx('c:\DataFiles\AllCategories_VCD.cdfx');
sList = systemList(cdfxObj)
```

```
sList =
```

```
1×3 table
```

<u>ShortName</u>	<u>Instances</u>	<u>Metadata</u>
"System1"	[1×16 string]	" "

Match a specified system.

```
sList = systemList(cdfxObj, 'System1');
```

## Input Arguments

### **cdfxObj** — CDFX-file object

asam.cdfx object

CDFX-file object, specified as an `asam.cdfx` object. Use the object to access the calibration data.

Example: `cdfx()`

### **sysName** — Parent system name

string

Parent system name, specified as a string.

Example: "System2"

Data Types: string

## Output Arguments

### **sList** — ECU system list

table

ECU system list, returned as a table.

## See Also

### Functions

`cdfx` | `getValue` | `instanceList` | `setValue` | `write`

**Introduced in R2019a**

## transmit

Send CAN messages to CAN bus

### Syntax

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

### Description

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

For more information on the elements of a message, see `canMessage`.

---

**Note** The `transmit` function ignores the `Timestamp` property and the `Error` property.

---

CAN is a peer-to-peer network, so when transmitting messages on a physical bus at least one other node must be present to properly acknowledge the message. Without another node, the transmission will fail as an error frame, and the device will continually retry to transmit.

## Examples

### Transmit a CAN Message

Define a CAN message and transmit it to the CAN bus.

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

## Transmit an Array of Messages

Transmit an array of three CAN messages.

```
transmit(canch, [message0, message1, message2])
```

## Transmit Messages on a Remote Frame

Transmit a CAN message on a remote frame, using the message Remote property.

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

## Input Arguments

### **canch** — CAN channel

CAN channel object

CAN channel, specified as a CAN channel object. This is the channel by which you access the CAN bus.

### **message** — Message to transmit

CAN message object or array of objects

Message to transmit, specified as a CAN message object or array of message objects. These messages are transmitted via a CAN channel to the bus.

## See Also

### Functions

canChannel | canMessage | receive

**Introduced in R2009a**

## transmit (J1939)

Send parameter groups via channel to J1939 bus

### Syntax

```
transmit(chan, pgrp)
```

### Description

`transmit(chan, pgrp)` sends the specified parameter groups in the array `pgrp` onto the J1939 bus via the channel `chan`.

### Examples

#### Send Parameter Groups onto Bus

Send the parameter group 'MyParameterGroup' to the bus.

```
db = canDatabase('MyDatabase.dbc');  
chan = j1939Channel(db, 'Vector', 'CANCaseXL 1', 1);  
start(chan)  
pgrp = j1939ParameterGroup(db, 'MyParameterGroup')  
transmit(chan, pgrp)
```

### Input Arguments

#### **chan** — J1939 channel

channel object

J1939 channel, specified as a channel object. Use the `j1939Channel` function to create and define the channel.



**pgrp — J1939 parameter groups**

array of ParameterGroup objects

J1939 parameter groups, specified as an array of ParameterGroup objects. Use the `j1939ParameterGroup` function to create and define the ParameterGroup objects.

## See Also

**Functions**

`j1939Channel` | `j1939ParameterGroup` | `receive` | `start`

**Introduced in R2015b**

## transmitConfiguration

Display messages configured for automatic transmission

### Syntax

```
transmitConfiguration(canch)
```

### Description

`transmitConfiguration(canch)` displays information about all messages in the CAN channel, `canch`, configured for periodic transmit or event-based transmit.

For more information on periodic transmit of messages, refer to `transmitPeriodic`.

For more information on event-based transmit of messages, refer to `transmitEvent`.

### Examples

#### Configure and View Message Transmit Settings

Create two messages with different transmit settings, then view those settings.

Create a CAN channel with two messages.

```
canch = canChannel('Vector','Virtual 1',1);  
msg1 = canMessage(500,false,8);  
msg2 = canMessage(750,false,8);
```

Configure the transmit settings for `msg1` and `msg2`.

```
transmitEvent(canch,msg1,'On');  
transmitPeriodic(canch,msg2,'On',1);
```

Display the transmit configuration for the messages on `canch`.

```
transmitConfiguration(canch)
```

```
Periodic Messages
```

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

```
Event Messages
```

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

## Input Arguments

**canch — CAN channel**

CAN channel object

CAN channel, specified as a CAN channel object. This is the channel by which you access the CAN bus for periodic or event-based transmission.

## See Also

### Functions

canChannel | canMessage | transmitEvent | transmitPeriodic

**Introduced in R2010b**

## transmitEvent

Configure messages for event-based transmission

### Syntax

```
transmitEvent(canch,msg,state)
```

### Description

`transmitEvent(canch,msg,state)` enables or disables an event-based transmit of the CAN message, `msg`, on the channel, according to the `state` argument of 'On' or 'Off'. A typical event that triggers a transmission is a change to the message `Data` property.

### Examples

#### Enable an Event-Based Message

Configure a channel with an event-based message.

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

```
canch = canChannel('MathWorks','Virtual 1',1);  
msg = canMessage(200,false,4);
```

Enable the message for event-based transmit, start the channel, and pack the message to trigger the event-based transmit.

```
transmitEvent(canch,msg,'On');  
start(canch);  
pack(msg,int32(1000),0,32,'LittleEndian')
```

## Input Arguments

### **canch — CAN channel**

CAN channel object

CAN channel, specified as a CAN channel object. This is the channel by which you access the CAN bus, and the channel on which the specified message is enabled for event-based transmit.

### **msg — Message to transmit**

CAN message object or array of objects

Message to transmit, specified as a CAN message object or array of message objects. This is the message enabled for event-based transmission on the specified CAN channel.

### **state — Enable event-based transmission**

'On' | 'Off'

Enable event-based transmission, specified as 'On' or 'Off'.

Example: 'On'

Data Types: char | string

## See Also

### **Functions**

canChannel | canMessage | transmitConfiguration | transmitPeriodic

### **Introduced in R2010b**

## transmitPeriodic

Configure messages for periodic transmission

### Syntax

```
transmitPeriodic(canch,msg,'On',period)
transmitPeriodic(canch,msg,'Off')
```

### Description

`transmitPeriodic(canch,msg,'On',period)` enables periodic transmit of the message, `msg`, on the channel, `canch`, to transmit at the specified period, `period`.

You can enable and disable periodic transmit even when the channel is running, allowing you to make changes to the state of the channel without stopping it.

`transmitPeriodic(canch,msg,'Off')` disables periodic transmission of the message, `msg`.

### Examples

#### Transmit a Message Periodically

Configure a channel to transmit messages periodically.

Construct a CAN channel and message.

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

Enable the message for periodic transmission on the channel, with a period of 1 second. Start the channel, and pack the message you want to send periodically.

```
transmitPeriodic(canch,msg,'0n',1);  
start(canch);  
pack(msg,int32(1000),0,32,'LittleEndian')
```

## Input Arguments

### **canch — CAN channel**

CAN channel object

CAN channel, specified as a CAN channel object. This is the CAN channel for which you are controlling periodic transmission.

### **msg — Message to transmit**

CAN message object or array of objects

Message to transmit, specified as a CAN message object or array of message objects. This is the message enabled for periodic transmission on the specified CAN channel.

### **period — Period of transmissions**

0.500 (default) | numeric value

Period of transmissions, specified in seconds as a numeric value. This argument is optional, with a default of 0.5 seconds.

Example: 1.0

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |  
uint32 | uint64

## See Also

### **Functions**

canChannel | canMessage | transmitConfiguration | transmitEvent

### **Introduced in R2010b**

## unpack

Unpack signal data from CAN message

### Syntax

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

### 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

#### Unpack Data from a CAN Message

Unpack the data value from a CAN message.

Unpack a 16-bit integer value.

```
message = canMessage(500, false, 8);  
pack(message, int16(1000), 0, 16, 'LittleEndian')  
value = unpack(message, 0, 16, 'LittleEndian', 'int16')
```

```
value =
```

```
    int16
```

```
    1000
```

Unpack a 32-bit single value.

```
pack(message, single(-40), 0, 32, 'LittleEndian')  
value = unpack(message, 0, 32, "LittleEndian", 'single')
```



```
value =  
    single  
    -40
```

Unpack a 64-bit double value.

```
pack(message,3.14,0,64,'LittleEndian')  
value = unpack(message,0,64,'LittleEndian','double')
```

```
value =  
    3.1400
```

## Input Arguments

### **message** — CAN message

CAN message object

CAN message, specified as a CAN message object, from which to unpack the data.

Example: `canMessage`

### **startbit** — Signal starting bit in data

single | double

Signal starting bit in the data, specified as a single or double value. This is the least significant bit position in the signal data. Accepted values for `startbit` are from 0 through 63, inclusive.

Example: 0

Data Types: single | double

### **signalsize** — Length of signal in bits

numeric value

Length of the signal in bits, specified as a numeric value. Accepted values for `signalsize` are from 1 through 64, inclusive.

Example: 16

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**byteorder** — Signal byte order format

`'LittleEndian'` | `'BigEndian'`

Signal byte order format, specified as `'LittleEndian'` or `'BigEndian'`.

Example: `'LittleEndian'`

Data Types: `char` | `string`

**datatype** — Data type of unpacked value

`char vector` | `string`

Data type of unpacked value, specified as a character vector or string. The supported values are `'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`.

Example: `'double'`

Data Types: `char` | `string`

## Output Arguments

**value** — Value of message data

numeric value

Value of message data, returned as a numeric value of the specified data type.

## See Also

**Functions**

`canMessage` | `extractAll` | `extractRecent` | `extractTime` | `pack`

**Introduced in R2009a**

# valueTableText

Look up value of table text for signal

## Syntax

```
vtt = valueTableText(db,MsgName,SignalName,TableVal)
```

## Description

`vtt = valueTableText(db,MsgName,SignalName,TableVal)` returns the text from the value table for a specified message signal.

## Examples

### View Table Text for Signal

Create a CAN database object, and select a message and signal to retrieve their table text.

Identify a message.

```
db = canDatabase('J1939DB.dbc');  
m = db.MessageInfo(1)  
  
m =  
    Name: 'A1'  
    Comment: 'This is a A1message'  
    ID: 419364350  
    Extended: 1  
    J1939: [1x1 struct]  
    Length: 8  
    Signals: {2x1 cell}  
    SignalInfo: [2x1 struct]  
    TxNodes: {'AerodynamicControl'}
```

```
Attributes: {4x1 cell}
AttributeInfo: [4x1 struct]
```

Select one of the message signals.

```
s = m.signalInfo(2)
```

```
s =
    Name: 'EngGasSupplyPress'
    Comment: 'Gage pressure of gas supply to fuel metering device.'
    StartBit: 8
    SignalSize: 16
    ByteOrder: 'LittleEndian'
    Signed: 0
    ValueType: 'Integer'
    Class: 'uint16'
    Factor: 0.5000
    Offset: 0
    Minimum: 0
    Maximum: 3.2128e+04
    Units: 'kPa'
    ValueTable: [4x1 struct]
    Multiplexor: 0
    Multiplexed: 0
    MultiplexMode: 0
    RxNodes: {'Aftertreatment_1_GasIntake'}
    Attributes: {3x1 cell}
    AttributeInfo: [3x1 struct]
```

Retrieve second table text for specified signal.

```
vtt = valueTableText(db,m.Name,s.Name,2)
```

```
vtt =
pump error
```

## Input Arguments

### **db** — CAN database

CAN database object

CAN database, specified as a CAN database object.

Example: db = canDatabase(\_\_\_\_\_) )

### **MsgName** — Message name

char vector | string

Message name, specified as a character vector or string. You can view available message names from the db.Messages property.

Example: 'A1'

Data Types: char | string

### **SignalName** — Signal name

char vector | string

Signal name, specified as a character vector or string. You can view available signal names from the `db.MessageInfo(n).Signals` property.

Example: 'EngGasSupplyPress'

Data Types: char | string

### **TableVal** — Table value

numeric value

Table value, specified as a numeric value.

Example: 2

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## **Output Arguments**

### **vtt** — Table text

table text

Table text, returned as a character vector.

## **See Also**

### **Functions**

`attributeInfo` | `canDatabase` | `messageInfo` | `nodeInfo` | `signalInfo`

### **Properties**

`MessageInfo` | `Messages`

### **Introduced in R2015b**

## Vehicle CAN Bus Monitor

Monitor vehicle CAN bus message traffic

### Description

The Vehicle CAN Bus Monitor displays live CAN message traffic.

Using this app, you can:

- View message traffic for a specified CAN device and channel.
- Save CAN bus messages to a log file.

---

**Notes** The Vehicle CAN Bus Monitor does not support the CAN FD protocol.

You cannot programmatically configure the Vehicle CAN Bus Monitor. However, you can use it to independently visualize bus traffic generated on CAN channels by MATLAB or Simulink CAN blocks.

---

### Open the Vehicle CAN Bus Monitor App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app.
- MATLAB command prompt: Enter `canTool`.

### Examples

- “Using the Vehicle CAN Bus Monitor” on page 5-9

### See Also

#### Functions

`canTool`

## **Topics**

"Using the Vehicle CAN Bus Monitor" on page 5-9

"Vehicle CAN Bus Monitor" on page 5-2

**Introduced in R2009a**

## viewMeasurementLists

View configured measurement lists on XCP channel

### Syntax

```
viewMeasurementLists(xcpch)
```

### Description

`viewMeasurementLists(xcpch)` shows you all configured measurement list sets for this XCP channel.

### Examples

#### View DAQ Measurement Lists

Create an XCP channel and configure a data acquisition measurement list, then view the configured measurement list.

Create an object to parse an A2L file and connect that to an XCP channel.

```
a2lfile = xcpA2L('XCPSIM.a2l')  
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1)
```

```
xcpch =
```

```
Channel with properties:
```

```
    SlaveName: 'CPP'  
    A2LFileName: 'XCPSIM.a2l'  
    TransportLayer: 'CAN'  
    TransportLayerDevice: [1x1 struct]  
    SeedKeyCallbackFcn: []  
    KeyValue: []
```



Connect the channel to the slave module.

```
connect(xcpch)
```

Setup a data acquisition measurement list with the '10 ms' event and 'PMW' measurement.

```
createMeasurementList(xcpch, 'DAQ', '10 ms', {'BitSlice0', 'PWMFiltered', 'Triangle'});
```

Create another measurement list with the '100ms' event and 'PWMFiltered' and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'DAQ', '100ms', {'PWMFiltered', 'Triangle'});
```

view details of the measurement list.

```
viewMeasurementLists(xcpch)
```

```
DAQ List #1 using the "10 ms" event @ 0.010000 seconds and the following measurements:
  PMW
```

```
DAQ List #2 using the "100ms" event @ 0.100000 seconds and the following measurements:
  PWMFiltered
  Triangle
```

## Input Arguments

### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

## See Also

`createMeasurementList` | `freeMeasurementLists`

**Introduced in R2013a**

## write

Export data of CDFX object to file

### Syntax

```
write(cdfxObj)
write(cdfxObj, CDFXfile)
```

### Description

`write(cdfxObj)` exports the data contents of the `asam.cdfx` object to the file specified by the `Path` property of the object.

`write(cdfxObj, CDFXfile)` exports the contents of the `asam.cdfx` object to the CDFX-file specified by `CDFXfile`.

### Examples

#### Write Modified Data to New CDFX-File

Create an `asam.cdfx` object with data from a file, modify the data in the object, and write it out to a new file.

```
cdfxObj = cdfx('c:\DataFiles\AllCategories_VCD.cdfx');
setValue(cdfxObj, 'VALUE_NUMERIC', 55)
write(cdfxObj, 'c:\DataFiles\AllCategories_NEW_VCD.cdfx')
```

### Input Arguments

**cdfxObj** — CDFX-file object  
`asam.cdfx` object

CDFX-file object, specified as an `asam.cdfx` object. Use the object to access the calibration data.

Example: `cdfx()`

### **CDFXfile — Calibration data format CDFX-file location**

`char` | `string`

Calibration data format CDFX-file location, specified as a character vector or string. `CDFXfile` can specify the file name in the current folder, or the full or relative path to the CDFX-file.

Example: `'ASAMCDFExample.cdfx'`

Data Types: `char` | `string`

## **See Also**

### **Functions**

`cdfx` | `getValue` | `instanceList` | `setValue` | `systemList`

**Introduced in R2019a**

## writeAxis

Scale and write specified axis value to direct memory

### Syntax

```
writeAxis(chanObj,axis,value)
```

### Description

`writeAxis(chanObj,axis,value)` scales and writes a value for the specified `axis` through the XCP channel object `chanObj`. This action performs a direct write to memory on the slave module.

### Examples

#### Write Value to XCP Channel Axis

Write a value to an XCP axis and verify the value.

Read original value.

```
a2lObj = xcpA2L('myA2Lfile.a2l');  
chanObj = xcpChannel(a2lObj,'CAN','Vector','Virtual 1',1);  
connect(chanObj);  
axisObj = a2lObj.AxisXs('pedal_position');  
value = readAxis(chanObj,axisObj)
```

25

Write new value.

```
newValue = 50;  
writeAxis(chanObj,axisObj,newValue);
```

Read value again to verify.

```
readAxis(chanObj,axisObj)
```

```
50
```

## Input Arguments

### **chanObj** — XCP channel

channel object

XCP channel, specified as an XCP channel object.

Example: `xcpChannel()`

### **axis** — XCP channel axis

axis object | char

XCP channel axis, specified as a character vector or axis object.

Example: `'pedal_position'`

Data Types: char

### **value** — Value for axis write

axis value

Value for axis write, specified as type supported by the axis.

## See Also

### Functions

`readAxis` | `readCharacteristic` | `readMeasurement` | `writeCharacteristic` | `writeMeasurement`

**Introduced in R2018a**

## writeCharacteristic

Scale and write specified characteristic value to direct memory

### Syntax

```
writeCharacteristic(chanObj,characteristic,value)
```

### Description

`writeCharacteristic(chanObj,characteristic,value)` scales and writes a value for the specified characteristic through the XCP channel object `chanObj`. This action performs a direct write to memory on the slave module.

### Examples

#### Write Value to an XCP Channel Characteristic

Write a value to an XCP characteristic and verify the value.

Read the original value.

```
a2lObj = xcpA2L('myA2Lfile.a2l');
chanObj = xcpChannel(a2lObj,'CAN','Vector','Virtual 1',1);
connect(chanObj);
charObj = a2lObj.CharacteristicInfo('torque_demand');
value = readCharacteristic(chanObj,charObj)'
```

```
100
```

Write new value.

```
newValue = 200;
writeCharacteristic(chanObj,charObj,newValue)';
```

Read value again to verify change.

```
readCharacteristic(chanObj, charObj)'
```

```
200
```

## Input Arguments

### **chanObj** — XCP channel

channel object

XCP channel, specified as an XCP channel object.

Example: `xcpChannel()`

### **characteristic** — XCP channel characteristic

characteristic object | char

XCP channel characteristic, specified as a character vector or characteristic object.

Example: `'torque_demand'`

Data Types: char

### **value** — Value for characteristic write

characteristic value

Value for characteristic write, specified as a type supported by the characteristic.

## See Also

### Functions

`readAxis` | `readCharacteristic` | `readMeasurement` | `writeAxis` | `writeMeasurement`

**Introduced in R2018a**

## writeMeasurement

Scale and write specified measurement value to direct memory

### Syntax

```
writeMeasurement(chanObj, measurement, value)
```

### Description

`writeMeasurement(chanObj, measurement, value)` scales and writes a value for the specified measurement through the XCP channel object `chanObj`. This action performs a direct write to memory on the slave module.

### Examples

#### Write Value to an XCP Channel Measurement

Write a value to an XCP measurement, and verify the value.

Read original value.

```
a2lObj = xcpA2L('myA2Lfile.a2l');  
chanObj = xcpChannel(a2lObj, 'CAN', 'Vector', 'Virtual 1', 1);  
connect(chanObj);  
measObj = a2lObj.MeasurementInfo('limit');  
value = readMeasurement(chanObj, measObj)
```

```
100
```

Write a new value.

```
newValue = 120;  
writeMeasurement(chanObj, measObj, newValue);
```

Read the value again to verify the change.



```
readMeasurement (chanObj, measObj)
```

```
120
```

## Input Arguments

### **chanObj** — XCP channel

channel object

XCP channel, specified as an XCP channel object.

Example: `xcpChannel()`

### **measurement** — XCP channel measurement

measurement object | char

XCP channel measurement, specified as a character vector or measurement object.

Example: `'curve1_8_uc'`

Data Types: char

### **value** — Value for measurement write

measurement value

Value for measurement write, specified as a data type supported by the measurement.

## See Also

### Functions

`readAxis` | `readCharacteristic` | `readMeasurement` | `writeAxis` | `writeCharacteristic`

**Introduced in R2018a**

## writeSingleValue

Write single sample to specified measurement

### Syntax

```
writeSingleValue(xcpch, measurementName, value)
```

### Description

`writeSingleValue(xcpch, measurementName, value)` writes a single value to the specified measurement through the configured XCP channel. The values are written directly to the memory on the slave module.

### Examples

#### Write a single value

Create an XCP channel and write a single value for the Triangle measurement directly to memory.

Link an A2L file to your session.

```
a2l = xcpA2L('XCPSIM.a2l')
```

Create an XCP channel and connect it to the slave module

```
xcpch = xcpChannel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);  
connect(xcpch)
```

Write the value 10 to the Triangle measurement.

```
writeSingleValue(xcpch, 'Triangle', 10)
```

## Input Arguments

### **xcpch — XCP channel**

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

### **measurementName — Name of single XCP measurement**

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure `measurementName` matches the corresponding measurement name defined in your A2L file.

Data Types: `char` | `string`

### **value — Value of the measurement**

numeric value

Value of the selected measurement, returned as a numeric value.

## See Also

`writeSTIMListData`

**Introduced in R2013a**

## writeSTIM

Write scaled value of specified measurement to STIM list

### Syntax

```
writeSTIM(xcpch,measurementName,value)
```

### Description

`writeSTIM(xcpch,measurementName,value)` writes the scaled value to the specified measurement on the XCP channel.

### Examples

#### Write Scaled Data to a Measurement in a Stimulation List

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up a data stimulation list and write to a specified measurement.

```
a2lObj = xcpA2L('myFile.a2l');  
channelObj = xcpChannel(a2lObj, 'CAN', 'Vector', 'CANcaseXL 1', 1);  
connect(channelObj);  
createMeasurementList(channelObj, 'STIM', 'Event1', 'Measurement1');  
startMeasurement(channelObj);  
writeSTIM(channelObj, 'Measurement1', newValue);
```

### Input Arguments

**xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

**measurementName — Name of single XCP measurement**

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure `measurementName` matches the corresponding measurement name defined in your A2L file.

Data Types: `char` | `string`

**value — Value of the measurement**

numeric value

Value of the measurement, specified as a numeric value.

## See Also

`writeSingleValue`

**Introduced in R2018b**

## writeSTIMListData

Write to specified measurement

### Syntax

```
writeSTIMListData(xcpch, measurementName, value)
```

### Description

`writeSTIMListData(xcpch, measurementName, value)` writes the specified value to the specified measurement on the XCP channel.

### Examples

#### Write Data to a Measurement in a Stimulation List

Create an XCP channel connected to a Vector CAN device on a virtual channel. Set up data stimulation list and write to a '100ms' event's 'Triangle' measurement.

Create an object to parse an A2L file and connect that to an XCP channel.

```
a2lfile = xcp.A2L('XCPSIM.a2l')  
xcpch = xcp.Channel(a2lfile, 'CAN', 'Vector', 'Virtual 1', 1);
```

Connect the channel to the slave.

```
connect(xcpch)
```

Create a measurement list with the '100ms' event and 'Bitslice0', 'PWMFiltered', and 'Triangle' measurements.

```
createMeasurementList(xcpch, 'STIM', '100ms', {'BitSlice0', 'PWMFiltered', 'Triangle'});
```

Start the measurement.

```
startMeasurement(xcpch)
```

Write data to the 'Triangle' measurement.

```
writeSTIMListData(xcpch, 'Triangle', 10)
```

## Input Arguments

### **xcpch** — XCP channel

XCP channel object

XCP channel, specified as an XCP channel object created using `xcpChannel`. The XCP channel object can then communicate with the specified slave module defined by the A2L file.

### **measurementName** — Name of single XCP measurement

character vector | string

Name of a single XCP measurement specified as a character vector or string. Make sure `measurementName` matches the corresponding measurement name defined in your A2L file.

Data Types: `char` | `string`

### **value** — Value of the measurement

numeric value

Value of the selected measurement, specified as a numeric value.

## See Also

`writeSingleValue`

**Introduced in R2013a**

## xcpA2L

Access A2L file

### Syntax

```
a2lfile = xcpA2L(filename)
```

### Description

`a2lfile = xcpA2L(filename)` creates an object that accesses an A2L file. The object can parse the contents of the file and view events and measurement information.

### Examples

#### Link to an A2L File

Create an A2L file object.

```
a2lfile = xcpA2L('XCPSIM.a2l')
```

### Input Arguments

#### **filename** — A2L file name

character vector | string

A2L file name, specified as a character vector or string. You must provide the file ending `.a2l` with the name. You can also provide a partial or full path to the file with the name.

Data Types: `char` | `string`



## See Also

### Functions

getEventInfo | getMeasurementInfo | xcpChannel

### Topics

“Inspect the Contents of an A2L File” on page 7-2

“XCP Database and Communication Workflow” on page 6-2

### Introduced in R2013a

## xcpChannel

Create XCP channel

### Syntax

```
xcpch = xcpChannel(a2lFile, 'CAN', vendor, deviceID)
xcpch = xcpChannel(a2lFile, 'CAN', vendor, deviceID, deviceChannelIndex)
xcpch = xcpChannel(a2lFile, 'TCP', IPAddr, portNmbr)
xcpch = xcpChannel(a2lFile, 'UDP', IPAddr, portNmbr)
xcpch = xcpChannel(a2lFile, 'TCP')
xcpch = xcpChannel(a2lFile, 'UDP')
```

### Description

`xcpch = xcpChannel(a2lFile, 'CAN', vendor, deviceID)` creates a channel connected to the CAN bus via the specified vendor and device. The XCP channel accesses the slave module via the CAN bus, parsing the attached A2L file.

Use this syntax for vendor 'PEAK-System' or 'NI'. With National Instruments CAN devices, the `deviceID` argument must include the interface number defined for the channel in the NI Measurement & Automation Explorer.

`xcpch = xcpChannel(a2lFile, 'CAN', vendor, deviceID, deviceChannelIndex)` creates a channel for the vendor 'Vector', 'Kvaser', or 'MathWorks'. Specify a numeric `deviceChannelIndex` for the channel.

`xcpch = xcpChannel(a2lFile, 'TCP', IPAddr, portNmbr)` or `xcpch = xcpChannel(a2lFile, 'UDP', IPAddr, portNmbr)` creates an XCP channel connected via Ethernet using TCP or UDP on the specified IP address and port.

---

**Note** XCP communication over UDP or TCP assumes a generic Ethernet adaptor. It is not supported on Ethernet connections of devices from specific vendors.

---

`xcpch = xcpChannel(a2lFile, 'TCP')` and `xcpch = xcpChannel(a2lFile, 'UDP')` use the IP address and port number defined in the A2L file.

## Examples

### Create an XCP Channel Using a CAN Slave Module

Create an XCP channel using a Vector CAN module virtual channel.

Link an A2L file to your session.

```
a2l = xcpA2L('XCPSIM.a2l');
```

Create an XCP channel.

```
xcpch = xcpChannel(a2l, 'CAN', 'Vector', 'Virtual 1', 1)
```

```
xcpch =
```

```
Channel with properties:
```

```
SlaveName: 'CPP'  
A2LFileName: 'XCPSIM.a2l'  
TransportLayer: 'CAN'  
TransportLayerDevice: [1x1 struct]  
SeedKeyDLL: []
```

### Create an XCP Channel for Ethernet

Create an XCP channel for TCP communication via Ethernet.

Link an A2L file to your session.

```
a2l = xcpA2L('XCPSIM.a2l');
```

Create an XCP channel.

```
xcpch = xcpChannel(a2l, 'TCP', '10.255.255.255', 80)
```

```
xcpch =
```

```
Channel with properties:
```

```
SlaveName: 'CPP'  
A2LFileName: 'XCPSIM.a2l'
```

```
TransportLayer: 'TCP'  
TransportLayerDevice: [1×1 struct]  
SeedKeyDLL: []
```

## Input Arguments

### **a2lFile** — A2L file

xcp.A2L object

A2L file, specified as an xcp.A2L object, used in this connection. You can create an A2L file object using xcpA2L.

### **vendor** — Device vendor

'NI' | 'Kvaser' | 'Vector' | 'PEAK-System' | 'MathWorks'

Device vendor name, specified as a character vector or string.

Example: 'Vector'

Data Types: char | string

### **deviceID** — Device to connect to

character vector | string

Device on the interface to connect to, specified as a character vector or string.

For National Instruments CAN devices, this must include the interface number for the device channel, defined in the NI Measurement & Automation Explorer.

Example: 'Virtual 1'

Data Types: char | string

### **deviceChannelIndex** — Index of channel on device

numeric value

Index of channel on the device, specified as a numeric value.

Example: 1

### **IPAddr** — IP address of device

char vector | string

IP address of the device, specified as a character vector or string

Example: '10.255.255.255'

Data Types: char | string

**portNbr — Port number for device connection**

numeric

Port number for device connection, specified as a numeric value.

Example: 80

## Output Arguments

**xcpch — XCP channel**

XCP channel object

XCP channel, returned as an object.

## See Also

**Functions**

connect | disconnect | isConnected | xcpA2L

**Introduced in R2013a**



# Properties – Alphabetical List

---

## AttributeInfo

Information on CAN database attributes

### Description

The `AttributeInfo` property is a structure with information about all attributes defined in the specified CAN database.

### Characteristics

Usage	CAN database
Read only	Always
Data type	Structure

### Values

The `AttributeInfo` property is a read-only structure. Use indexing to access the information of each attribute.

### Examples

#### Display Database Attribute Information

```
db = canDatabase('J1939DB.dbc');  
db.AttributeInfo
```

```
3x1 struct array with fields:  
    Name  
    ObjectType  
    DataType  
    DefaultValue  
    Value
```



```
db.AttributeInfo(1).Name
```

```
BusType
```

```
db.AttributeInfo(1).Value
```

```
CAN
```

## See Also

### Functions

[attributeInfo](#) | [canDatabase](#)

### Properties

[Attributes](#)

## Attributes

Attribute names from CAN database

## Description

The `Attributes` property stores the names of all attributes defined in the specified CAN database.

## Characteristics

Usage	CAN database
Read only	Always
Data type	Cell array of character vectors

## Values

The `Attributes` property displays a cell array of character vectors. You cannot edit this property.

## Examples

### Display Database Attributes

```
db = canDatabase('J1939DB.dbc');  
db.Attributes  
  
    'BusType'  
    'DatabaseVersion'  
    'ProtocolType'  
  
db.Attributes{1}  
  
BusType
```

## See Also

### Functions

`attributeInfo` | `canDatabase`

### Properties

`AttributeInfo`

## BusLoad

Load on CAN bus

### Description

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

### Characteristics

Usage	CAN channel
Read only	Always
Data type	Float

### Values

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

### See Also

#### Functions

canChannel

# BusSpeed

Bit rate of bus

## Description

The BusSpeed property indicates the speed at which messages are transmitted in bits per second. You can set BusSpeed to an acceptable bit rate using the configBusSpeed function.

## Characteristics

Usage	CAN channel, J1939 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 with the channel name and the value as input parameters.

## Examples

Change the bus speed of the CAN channel object canch to 250,000 bits per second, and view the result.

```
configBusSpeed(canch, 250000);  
bs = canch.BusSpeed
```

## **See Also**

### **Functions**

canChannel, j1939Channel, configBusSpeed

### **Properties**

NumOfSamples, SJW, TSEG1, TSEG2

# BusStatus

Determine status of bus

## Description

The `BusStatus` property displays information about the state of the CAN bus or the J1939 bus.

## Characteristics

Usage	CAN channel, J1939 channel
Read only	Always
Data type	Character vector

## Values

- `N/A` — Property not supported by vendor.
- `ErrorActive` — Node transmits Active Error Flags when it detects errors.
- `ErrorPassive` — Node transmits Passive Error Flags when it detects errors.
- `BusOff` — Node will not transmit anything on the bus.

## See Also

### Functions

`canChannel`, `j1939Channel`

## Data

CAN message or J1939 parameter group data

## Description

Use the `Data` property to define your message data in a CAN message or parameter group data in a J1939 parameter group.

## Characteristics

Usage	CAN message, J1939 parameter group
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

### Specify CAN Message Data

Create a CAN message and load data into a message.

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



## Specify J1939 Parameter Group Data

Create a parameter group and specify data.

```
pg = j1939ParameterGroup(db, 'PackedData')  
pg.Data(1:2) = [50 0]
```

## See Also

### Functions

canMessage, pack

## Database

Store CAN database information

## Description

The Database property stores information about an attached CAN database.

## Characteristics

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

## Values

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

## Examples

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

```
message.Database
```

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

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

---

**Tip** CAN database file names containing non-alphanumeric characters such as equal signs and ampersands are incompatible with Vehicle Network Toolbox. You can use a

period sign in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

---

## **See Also**

### **Functions**

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

## DestinationAddress

Address of parameter group destination

### Description

Address of the J1939 parameter group destination. `DestinationAddress` identifies the parameter group source on the J1939 network. The source uses the specified destination address to send parameter groups.

### Characteristics

Usage	J1939 parameter group
Read only	Never
Data type	Numeric

### Values

Specify `DestinationAddress` of the parameter group as a number from 0 through 253. 254 is a null value and is used by your application to detect available addresses on the J1939 network. To send a parameter group to all devices on the network, use 255, which is the global value.

### See Also

#### Functions

`j1939ParameterGroup`

## Device

Display channel device type

## Description

For National Instruments devices, the Device property displays the device number on the hardware.

For all other devices, the Device property displays information about the device type to which the CAN or J1939 channel is connected.

## Characteristics

Usage	CAN channel, J1939 channel
Read only	Always
Data type	Character vector

## Values

Values are automatically defined when you configure the channel with the `canChannel` or the `j1939Channel` function.

## See Also

### Functions

`canChannel`, `canHWInfo`, `j1939Channel`

## **Properties**

DeviceChannelIndex, DeviceSerialNumber, DeviceVendor

## Device(NI)

Display NI CAN channel device type

### Description

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

### Characteristics

Usage	CAN channel
Read only	Always
Data type	Character vector

### Values

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

### See Also

#### Functions

`canChannel`, `canHWInfo`

#### Properties

`DeviceChannelIndex`, `DeviceVendor`

## DeviceChannelIndex

Display device channel index

### Description

The DeviceChannelIndex property displays the channel index on which the selected CAN or J1939 channel is configured.

### Characteristics

Usage	CAN channel, J1939 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, j1939Channel

#### Properties

Device, DeviceVendor



# DeviceSerialNumber

Display device serial number

## Description

The `DeviceSerialNumber` property displays the serial number of the device connected to the CAN or J1939 channel.

## Characteristics

Usage	CAN channel, J1939 channel
Read only	Always
Data type	<ul style="list-style-type: none"><li>Numeric</li><li>Hexadecimal character vector (NI CAN devices only)</li></ul>

## Values

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

## See Also

### Functions

`canChannel`, `canHWInfo`, `j1939Channel`

### Properties

`Device`, `DeviceVendor`

## DeviceVendor

Display device vendor name

## Description

The DeviceVendor property displays the name of the device vendor.

## Characteristics

Usage	CAN channel, J1939 channel
Read only	Always
Data type	Character vector

## Values

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

## See Also

### Functions

`canChannel`, `canHWInfo`, `j1939Channel`

### Properties

`Device`, `DeviceChannelIndex`, `DeviceSerialNumber`

# Error

CAN message error frame

## Description

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

## Characteristics

Usage	CAN message
Read only	Always
Data type	Boolean

## Values

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

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

## See Also

### Functions

`canMessage`

## Extended

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 eight bytes, type:

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

You cannot edit this property after the initial configuration.

## **See Also**

### **Functions**

canMessage

### **Properties**

ID

## FilterBlockList

List of parameter groups to block

### Description

FilterBlockList displays a list of parameter group names and numbers blocked by the channel.

### Characteristics

Usage	J1939 channel
Read only	Always
Data type	Character vector, Cell array

### Values

The list displays parameter group names and numbers as character vectors or cell arrays of character vectors and numbers. To change the values, use one of the filtering functions.

### See Also

#### Functions

`j1939Channel`, `filterAllowOnly`, `filterAllowAll`, `filterBlockAll`

#### Properties

`FilterPassList`

# FilterPassList

List of parameter groups to pass

## Description

FilterPassList displays a list of parameter group names and numbers that the channel can pass to the network.

## Characteristics

Usage	J1939 channel
Read only	Always
Data type	Character vector, Cell array

## Values

The list displays parameter group names and numbers as character vectors or cell arrays of character vectors and numbers. To change the values, use one of the filtering functions.

## See Also

### Functions

`j1939Channel`, `filterAllowOnly`, `filterAllowAll`, `filterBlockAll`

### Properties

`FilterBlockList`

## ID

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 eight bytes, type:

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



## See Also

### Functions

canMessage

### Properties

Extended

## InitialTimestamp

Indicate when channel started

### Description

The `InitialTimestamp` property indicates when the channel was set online with the `start` function or when its start trigger was received. For National Instruments devices, the time is obtained from the device driver; for devices from other vendors the time is obtained from the operating system where MATLAB is running.

### Characteristics

Usage	CAN channel
Read only	Always
Data type	Datetime

### Examples

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

```
StrtTime = canch.InitialTimestamp
```

### See Also

#### Functions

`canChannel` | `start`

#### Properties

`StartTriggerTerminal`

**Introduced in R2016a**

## InitializationAccess

Determine control of device channel

### Description

The `InitializationAccess` property determines if the configured CAN or J1939 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, J1939 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`

# MessageInfo

Information on CAN database messages

## Description

The `MessageInfo` property is a structure with information about all messages defined in the specified CAN database.

## Characteristics

Usage	CAN database
Read only	Always
Data type	Structure

## Values

The `MessageInfo` property is a read-only structure. Use indexing to access the information of each message.

## Examples

### Display Database Message Information

```
db = canDatabase('J1939DB.dbc');  
db.MessageInfo
```

```
3×1 struct array with fields:  
    Name  
    Comment  
    ID  
    Extended  
    J1939
```

- Length
- Signals
- SignalInfo
- TxNodes
- Attributes
- AttributeInfo

`db.MessageInfo(1).Name`

A1

`db.MessageInfo(1).Signals`

- 'EngBlowerBypassValvePos'
- 'EngGasSupplyPress'

## See Also

### Functions

`canDatabase` | `messageInfo` | `signalInfo` | `valueTableText`

### Properties

`Messages` | `SignalInfo` | `Signals`

# MessageReceivedFcn

Specify function to run

## Description

Configure `MessageReceivedFcn` as a callback function to run a character vector 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 character vector. You can specify the name of a callback function that you want to run when the specified number of messages are available.

## Examples

### Specify Callback

Specify the callback function to execute.

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

## See Also

### Functions

canChannel

### Properties

MessageReceivedFcnCount | MessagesAvailable

### Topics

“CAN Message Reception Callback Functions”



# MessageReceivedFcnCount

Specify number of messages available before function is triggered

## Description

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

## Characteristics

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

## Values

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

## Examples

### Specify Message Count

Specify the message count to trigger a callback.

```
canch.MessageReceivedFcnCount = 55;
```

## See Also

### Functions

canChannel

### Properties

MessageReceivedFcn | MessagesAvailable

### Topics

“CAN Message Reception Callback Functions”

# Messages

Message names from CAN database

## Description

The `Messages` property stores the names of all messages defined in the specified CAN database.

## Characteristics

Usage	CAN database
Read only	Always
Data type	Cell array of character vectors

## Values

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

## Examples

### Display database message information

```
db = canDatabase('J1939DB.dbc');  
db.Messages
```

```
    'A1'  
    'A1DEFI'  
    'A1DEFSI'
```

```
db.Messages{1}
```

```
A1
```

## See Also

### Functions

[canDatabase](#) | [messageInfo](#) | [signalInfo](#) | [valueTableText](#)

### Properties

[MessageInfo](#) | [SignalInfo](#) | [Signals](#)

# MessagesAvailable

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

Display number of messages received by CAN channel

### Description

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

### Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

### Values

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

### See Also

#### Functions

canChannel, canHWInfo

#### Properties

MessagesAvailable, MessagesTransmitted

# MessagesTransmitted

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)**

CAN database name

### **Description**

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

### **Characteristics**

Usage	CAN database
Read only	Always
Data type	Character vector

### **Values**

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

### **See Also**

#### **Functions**

canDatabase

#### **Properties**

Extended, ID



## Name (CAN)

CAN message name

## Description

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

## Characteristics

Usage	CAN message
Read only	Always
Data type	Character vector

## Values

Name is a character vector 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

## **Name (J1939)**

J1939 parameter group name

## **Description**

The Name property displays the name of the J1939 parameter group.

## **Characteristics**

Usage	J1939 parameter group
Read only	Never
Data type	Character vector

## **Values**

Name is a character vector value. This value is acquired from the name you define when you create the parameter group.

## **See Also**

### **Functions**

`j1939ParameterGroup`

# NodeInfo

Information on CAN database nodes

## Description

The `NodeInfo` property is a structure with information about all nodes defined in the specified CAN database.

## Characteristics

Usage	CAN database
Read only	Always
Data type	Structure

## Values

The `NodeInfo` property is a read-only structure. Use indexing to access the information of each node.

## Examples

### Display Database Node Information

```
db = canDatabase('J1939DB.dbc');  
db.NodeInfo
```

```
3×1 struct array with fields:  
    Name  
    Comment  
    Attributes  
    AttributeInfo
```

```
db.NodeInfo(1).Name
```

AerodynamicControl

## **See Also**

### **Functions**

canDatabase | nodeInfo

### **Properties**

Nodes

# Nodes

Node names from CAN database

## Description

The Nodes property stores the names of all nodes defined in the specified CAN database.

## Characteristics

Usage	CAN database
Read only	Always
Data type	Cell array of character vectors

## Values

The Nodes property displays a cell array of character vectors. You cannot edit this property.

## Examples

### Display Database Attributes

```
db = canDatabase('J1939DB.dbc');  
db.Nodes
```

```
    'AerodynamicControl'  
    'Aftertreatment_1_GasIntake'  
    'Aftertreatment_1_GasOutlet'
```

```
db.Nodes{1}
```

```
AerodynamicControl
```

## See Also

### Functions

canDatabase | nodeInfo

### Properties

NodeInfo

# NumOfSamples

Display number of samples available to channel

## Description

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

---

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

---

## Characteristics

Usage	CAN channel, J1939 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, j1939ChannelconfigBusSpeed

## **Properties**

BusSpeed, SJW, TSEG1, TSEG2



# OnboardTermination

Configure bus termination on device

## Description

The `OnboardTermination` property specifies the device to use its onboard termination of the CAN bus. For more information on the behavior and characteristics of a specific device, refer to its vendor's documentation.

## Characteristics

Usage	NI-XNET CAN channel
Read only	When online
Data type	Logical

## Examples

```
canch = canChannel('NI', 'CAN1');  
canch.OnboardTermination = true
```

## See Also

### Functions

`canChannel`

**Introduced in R2016a**

## ParameterGroupsAvailable

Number of parameter groups available

### Description

ParameterGroupsAvailable displays the total number of parameter groups available to the channel.

### Characteristics

Usage	J1939 channel
Read only	Always
Data type	Double

### Values

The property displays the number of available parameters to the channel.

### See Also

#### Functions

j1939Channel

#### Properties

ParameterGroupsReceived, ParameterGroupsTransmitted

# ParameterGroupsReceived

Number of parameter groups received

## Description

ParameterGroupsTransmitted displays the total number of parameter groups transmitted since the channel was started.

## Characteristics

Usage	J1939 channel
Read only	Always
Data type	Double

## Values

The property displays the number of received parameters through the channel.

## See Also

### Functions

j1939Channel

### Properties

ParameterGroupsTransmitted, ParameterGroupsAvailable

## ParameterGroupsTransmitted

Number of parameter groups transmitted

### Description

ParameterGroupsTransmitted displays the total number of parameter groups transmitted since the channel was started.

### Characteristics

Usage	J1939 channel
Read only	Always
Data type	Double

### Values

The property displays the number of transmitted parameter through the channel.

### See Also

#### Functions

j1939Channel

#### Properties

ParameterGroupsReceived, ParameterGroupsAvailable

## Path

CAN database folder path

## Description

The Path property displays the path to the CAN database.

## Characteristics

Usage	CAN database
Read only	Always
Data type	Character vector

## Values

The path name is a character vector value, pointing to the CAN database in your folder structure.

## See Also

### Functions

canDatabase

## **PDUFormatType**

J1939 parameter group PDU format

### **Description**

The `PDUFormatType` property displays the J1939 protocol data unit format of the parameter group.

### **Characteristics**

Usage	J1939 parameter group
Read only	Always
Data type	Character vector

### **Values**

`PDUFormatType` is displayed as a character vector. This value is automatically assigned when you create the parameter group.

### **See Also**

#### **Functions**

`j1939ParameterGroup`

# PGN

J1939 parameter group number

## Description

The PGN property displays the number of the parameter group.

## Characteristics

Usage	J1939 parameter group
Read only	Never
Data type	Number

## Values

PGN is represented as a number. This value is automatically assigned when you create the parameter group.

## See Also

### Functions

`j1939ParameterGroup`

## Priority

Priority of parameter group

## Description

The `Priority` property specifies the precedence of the parameter group on the J1939 network.

## Values

`Priority` takes a numeric value of 0 to 7, which specifies the order of importance of the parameter group.

## See Also

### Functions

`j1939ParameterGroup`

## Characteristics

Usage	J1939 parameter group
Read only	Never
Data type	Numeric



# ReceiveErrorCount

Display number of received errors detected by channel

## Description

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

## Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

## Values

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

## See Also

### Functions

`canChannel`, `receive`

### Properties

`TransmitErrorCount`

## Remote

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

Determine status of channel

## Description

The `Running` property displays information about the state of the CAN or J1939 channel.

## Characteristics

Usage	CAN channel, J1939 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`, `j1939Channel`, `start`

## SilentMode

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, J1939 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
```

## **See Also**

### **Functions**

canChannel1939Channel

## SignalInfo

Information on CAN database message signals

### Description

The `SignalInfo` property is a structure with information about all signals defined in the specified CAN database message.

### Characteristics

Usage	CAN database
Read only	Always
Data type	Structure

### Values

The `SignalInfo` property is a read-only structure. Use indexing to access the information of each signal.

### Examples

#### Display Database Signal Information

```
db = canDatabase('J1939DB.dbc');  
db.MessageInfo
```

```
3x1 struct array with fields:  
    Name  
    Comment  
    ID  
    Extended  
    J1939
```

```
Length  
Signals  
SignalInfo  
TxNodes  
Attributes  
AttributeInfo
```

```
s = db.MessageInfo(1).SignalInfo
```

```
2×1 struct array with fields:
```

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

```
s(2).Name
```

```
EngGasSupplyPress
```

## See Also

### Functions

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

### Properties

[MessageInfo](#) | [Messages](#) | [Signals](#)

## Signals

Physical signals defined in CAN message or J1939 parameter group

### Description

The `Signals` property allows you to view and edit signal values defined for a CAN message or a J1939 parameter group. This property displays an empty structure if the message has no defined signals or a database is not attached to the message or parameter group. The input values for this property depends on the signal type.

### Characteristics

Usage	CAN message, J1939 parameter group
Read only	Sometimes
Data type	Structure

### Examples

#### Display CAN Message Signals

Create a CAN message.

```
message = canMessage(canDb, 'messageName');
```

Display message signals.

```
message.Signals  
    VehicleSpeed: 0  
    EngineRPM: 250
```

Change the value of a signal.

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



## Display J1939 Parameter Group Signals

Create a parameter group.

```
pg = j1939ParameterGroup(db, 'PackedData')
```

Display parameter group signals

```
pg.Signals
```

```
ToggleSwitch: -1
SliderSwitch: -1
RockerSwitch: -1
RepeatingStairs: 255
PushButton: 1
```

Change the value of the repeating stairs.

```
pg.Signals.RepeatingStairs = 200
```

```
ToggleSwitch: -1
SliderSwitch: -1
RockerSwitch: -1
RepeatingStairs: 200
PushButton: 1
```

## See Also

### Functions

[canDatabase](#) | [canMessage](#) | [j1939ParameterGroup](#) | [messageInfo](#) | [signalInfo](#)

### Properties

[MessageInfo](#) | [Messages](#) | [SignalInfo](#)

## SJW

Synchronization jump width (SJW) of bit time segment

### Description

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

---

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

---

### Characteristics

Usage	CAN channel, J1939 channel
Read only	Always
Data type	Numeric

### Values

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

### See Also

### Functions

canChannel, j1939Channel, configBusSpeed

## **Properties**

BusSpeed, NumOfSamples, TSEG1, TSEG2

## SourceAddress

Address of parameter group source

### Description

Address of the J1939 parameter group source. `SourceAddress` identifies the parameter group source on the J1939 network. This allows the destinations to identify the sender and respond appropriately.

### Characteristics

Usage	J1939 parameter group
Read only	Never
Data type	Numeric

### Values

Specify `SourceAddress` of the parameter group as a number between 0 and 253. 254 is a null value and is used by your application to detect available addresses on the J1939 network. To send a parameter group to all devices on the network, use 255, which is the global value.

### See Also

#### Functions

`j1939ParameterGroup`

# StartTriggerTerminal

Specify start trigger source terminal

## Description

The `StartTriggerTerminal` property specifies a synchronization trigger connection to start the NI-XNET channel on the connected source terminal.

To configure an NI-XNET CAN module (such as NI 9862) to start acquisition on an external signal triggering event provided at an external chassis terminal, set the CAN channel `StartTriggerTerminal` property to the appropriate terminal name. Form the property value character vector by combining the chassis name from the NI MAX utility and the trigger terminal name; for example, `'/cDAQ3/PFI0'`.

---

**Note** This property can be configured only once. After it is assigned, the property is read-only and cannot be changed. The only way to set a different value is to `clear` the channel object, recreate the channel with `canChannel`, and configure its properties.

---

## Characteristics

Usage	NI-XNET CAN channel
Read only	After assigned
Data type	Character vector

## Examples

Configure a NI-XNET CAN module start trigger on terminal `/cDAQ3/PFI0`.

```
ch1 = canChannel('NI','CAN1')
ch1.StartTriggerTerminal = '/cDAQ3/PFI0'
start(ch1) % Acquisition begins on hardware trigger
```

With a hardware triggering configuration, the `InitialTimestamp` value represents the absolute time the CAN channel acquisition was triggered. The `Timestamp` of the received CAN messages is relative to the trigger moment.

```
ch1.InitialTimestamp  
messages = receive(ch1,Inf);  
messages(1).Timestamp
```

## See Also

### Functions

`canChannel`

### Properties

`InitialTimestamp`

**Introduced in R2016a**

## Timestamp (CAN)

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



## Timestamp (J1939)

Display parameter received timestamp

### Description

The `Timestamp` property displays the time at which the parameter group was received on a J1939 channel. This time is based on the hardware log.

### Characteristics

Usage	J1939 parameter group
Read only	Never
Data type	Double

### Values

`Timestamp` displays a numeric value indicating the time the parameter group was received, based on the logged time on the hardware.

### See Also

#### Functions

`j1939ParameterGroup`

## TransceiverName

Name of device transceiver

### Description

TransceiverName displays the name of the device transceiver. The device transceiver translates the digital bit stream going to and coming from the bus into the real electrical signals present on the bus.

### Characteristics

Usage	CAN channel, J1939 channel
Read only	Always
Data type	Character vector

### Values

Values are automatically defined when you configure the channel with the `canChannel` or the `j1939Channel` function.

### See Also

#### Functions

`canChannel`, `j1939Channel`

#### Properties

`TransceiverState`

# TransceiverState

Display state or mode of transceiver

## Description

If your CAN or J1939 transceiver allows you to control its mode, you can use the TransceiverState property to set the mode.

## Characteristics

Usage	CAN channel, J1939 CAN channel
Read only	Never
Data type	Numeric

## Values

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

- high speed
- high voltage
- sleep
- wake up

## See Also

### Functions

canChannel

## **Properties**

TransceiverName

# TransmitErrorCount

Display number of transmitted errors by channel

## Description

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

## Characteristics

Usage	CAN channel
Read only	Always
Data type	Double

## Values

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

## See Also

### Functions

`canChannel`, `transmit`

### Properties

`ReceiveErrorCount`

## TSEG1

Display amount that channel can lengthen sample time

### Description

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

---

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

---

### Characteristics

Usage	CAN channel, J1939 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`, `j1939Channel`, `configBusSpeed`

## **Properties**

BusSpeed, NumOfSamples, SJW, TSEG2

## TSEG2

Display amount that channel can shorten sample time

### Description

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

---

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

---

### Characteristics

Usage	CAN channel, J1939 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`, `j1939Channel`, `configBusSpeed`



## **Properties**

BusSpeed, NumOfSamples, SJW, TSEG1

## UserData

Enter custom data

## Description

Enter custom data to be stored in your CAN message or a J1939 parameter group, channel, or database object using the `UserData` property. When you save an object with `UserData` specified, you automatically save the custom data. When you load an object with `UserData` specified, you automatically load the custom data.

---

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

---

## Characteristics

Usage	CAN channel, J1939 channel, CAN message, J1939 parameter group, CAN database
Read only	Never
Data type	User defined

## See Also

### Functions

`canChannel`, `canMessage`, `canDatabase`, `j1939ParameterGroup`, `j1939Channel`

## Events

Display A2L events list

## Description

The `Events` property displays events available in the selected A2L description file. This property contains a cell array of character vectors that correspond to the names of events in the A2L file. To use the A2L file events, see “Access Event Information” on page 7-4.

## **Measurements**

Display A2L measurements list

### **Description**

The `Measurements` property displays measurements available in the selected A2L description file. This property contains a cell array of character vectors that correspond to the names of measurements in the A2L file. To use the A2L file measurements see “Access Measurement Information” on page 7-2.

# DAQInfo

Data acquisition information in A2L file

## Description

The `DAQInfo` property displays data acquisition information in the A2L description file. This property contains a structure with values corresponding to the DAQ features in the slave.

## **SlaveName**

Name of connected slave

### **Description**

The `SlaveName` property displays the name of the slave node as specified in the A2L file. The name is specified as a character vector.

## FileName

Name of referenced A2L file

## Description

The `FileName` property displays the name of the referenced A2L file as a character vector.

## **FilePath**

Path of A2L file

## **Description**

The `FileName` property displays the full file path to the A2L file as a character vector.



# ProtocolLayerInfo

Protocol layer information

## Description

The `ProtocolLayerInfo` property displays a structure containing general information about the XCP protocol implementation of the slave as defined in the A2L file.

## **TransportLayerCANInfo**

CAN transport layer information

### **Description**

The `TransportLayerCANInfo` property displays a structure containing general information about the CAN transport layer for the XCP connection to the slave as defined in the A2L file.

## **A2LFileName**

Name of the A2L file

### **Description**

The `A2LFileName` property displays the name of the A2L file contains information about the slave that an XCP channel can access.

## **SeedKeyDLL**

Name of seed and key security access dll

### **Description**

The SeedKeyDLL property displays the name of the dll file that contains the seed and key security algorithm used to unlock an XCP slave module.

# TransportLayer

Transport layer type

## Description

The TransportLayer property displays the type of transport layer used in the XCP connection.

## **TransportLayerDevice**

XCP transport layer connection

### **Description**

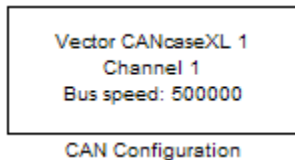
The `TransportLayerDevice` property contains a structure with XCP transport layer connection details, including information about the device through which the channel communicates with the slave.

# Block Reference

---

## CAN Configuration

Configure parameters for specified CAN device



## Library

Vehicle Network Toolbox: CAN Communication

## Description

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

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

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

---

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

---

## Other Supported Features

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

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



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

## Parameters

### Device

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

### Bus speed

Set the BusSpeed property for the selected device, in bits per second. The default bus speed is the default assigned by the selected device.

### Enable bit parameters manually

---

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

---

Select this check box to specify bit parameter settings manually. The bit parameter settings include:

**Synchronization jump width, Time segment 1, Time segment 2, and Number of samples.** If you do not select this option, the device automatically assigns the bit parameters depending on the bus speed setting.

---

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

---

### Synchronization jump width

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

### Time segment 1

Specify the amount of bit time segments that the channel can lengthen the sample time. The specified value must be a positive integer. If you do not specify a value, the

selected bus speed setting determines the default value. To change this value, select the **Enable bit parameters manually** check box first. Refer to the TSEG1 property for more information.

### **Time segment 2**

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

### **Number of samples**

Specify the total number of samples available to this channel. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determines the default value. To change this value, select the **Enable bit parameters manually** check box first. Refer to the NumOfSamples property for more information.

### **Verify bit parameter settings validity**

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

### **Acknowledge mode**

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

---

### **Notes**

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

## See Also

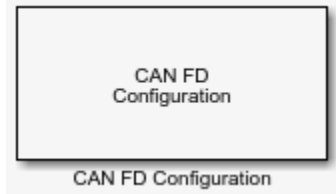
### Blocks

CAN Receive | CAN Transmit

**Introduced in R2009a**

## CAN FD Configuration

Configure parameters for specified CAN FD device



### Library

Vehicle Network Toolbox: CAN FD Communication

### Description

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

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

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

---

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

---

## Other Supported Features

The CAN FD Configuration block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration” (Simulink).

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

## Parameters

### Device

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

### Arbitration Bus speed

Set arbitration bus speed for the selected device, in bits per second. The default speed is assigned by the selected device.

### Data Bus speed

Set data bus speed for the selected device, in bits per second. The default speed is assigned by the selected device.

### Bus frequency

(PEAK-System only.) Set the bus frequency, in megahertz.

### Arbitration/Data bit rate prescaler

(PEAK-System only.) Set separate prescaler values for arbitration and data bit rates.

For Vector and PEAK-System devices, the next three parameters are available in two sets for manually setting bit parameters for data and arbitration bus speeds.

### Synchronization jump width

Specify the maximum value of the bit time adjustments. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determine the default value.

**Time segment 1**

Specify the amount of bit time segments that the channel can lengthen the sample time. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determines the default value.

**Time segment 2**

Specify the amount of bit time segments that the channel can shorten the sample time to resynchronize. The specified value must be a positive integer. If you do not specify a value, the selected bus speed setting determines the default value.

**Verify bit parameter settings validity**

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

**Acknowledge mode**

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

---

**Notes**

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

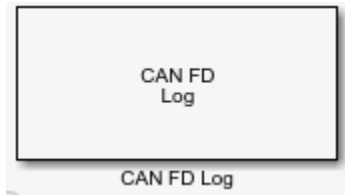
**See Also****Blocks**

CAN FD Receive | CAN FD Transmit | CAN FD Unpack | CAN FD Pack

**Introduced in R2018a**

## CAN FD Log

Log received CAN FD messages



## Library

Vehicle Network Toolbox: CAN FD Communication

## Description

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

Configure your CAN FD Log block to log from the Simulink input port. For more information, see “Log and Replay CAN Messages”.

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

If you want to use messages logged using Simulink blocks in the MATLAB Command window, use `canFDMessage` to convert messages to the correct format.

---

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

---

---

**Note** You cannot have more than one CAN FD Log block in a model using the same PEAK-System device channel.

---

## Other Supported Features

- The CAN FD Log block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see “Acceleration” (Simulink).
- The CAN FD Log block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see “Code Generation” on page 15-10.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder™, and Embedded Coder® software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

### Shared Library Dependencies

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

To set up packNGo:

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

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



run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux® platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### File name

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

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

### Variable name

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

### Maximum number of messages to log

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

### Log messages from

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

### Device

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

**Sample time**

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the CAN FD Log block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

**See Also****Blocks**

CAN FD Replay

**Functions**

canFDMessage

**Introduced in R2018b**

# CAN FD Pack

Pack individual signals into message for CAN FD bus



## Library

Vehicle Network Toolbox: CAN FD Communication

## Description

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

---

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

---

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

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

The block outputs CAN FD messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

## Other Supported Features

The CAN FD Pack block supports:

- The use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration” (Simulink).
- Code generation to deploy models to targets.

---

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

---

## Dialog Box

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

### Parameters

#### Data is input as

Select your data signal:

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

---

**Note** The block supports the following input signals data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

---

#### CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The

message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

---

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

---

### Message list

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

## Message

### Name

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

### Protocol mode

Specify the message protocol mode as **CAN FD** or **CAN**.

### Identifier type

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

### Identifier

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

### Length (bytes)

Specify the length of your message. For **CAN** messages the value can be 0-8 bytes; for **CAN FD** the value can be 0-8, 12, 16, 20, 24, 32, 48, or 64 bytes. If you are using

CANdb specified signals for your data input, the CANdb file defines the length of your message. This option is available if you choose to input raw data or manually specify signals.

**Remote frame**

(Disabled for CAN FD protocol mode.) Specify the CAN message as a remote frame.

**Bit Rate Switch (BRS)**

(Disabled for CAN protocol mode.) Enable bitrate switch.

## Signals Table

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

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

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

**Name**

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

**Start bit**

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. For CAN the start bit must be an integer from 0 through 63, for CAN FD 0 through 511, within the number of bits in the message. (Note that message length is specified in bytes.)

**Length (bits)**

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64. The sum of all the signal lengths in a message is limited to the number of bits in the message length; that is, all signals must cumulatively fit within the length of the message. (Note that message length is specified in bytes and signal length in bits.)

**Byte order**

Select either of the following options:

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

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

Diagram illustrating Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address. The table shows bit numbers for Data Bytes 0 through 7. A red arrow points from bit 20 (labeled LSB) to bit 27 (labeled MSB), indicating the direction of data writing. A callout box states: "Data begins at the least significant bit and starts at 20". Another callout box states: "Data is written up to the most significant bit and ends at 27".

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

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

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

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

**Data type**

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

- signed (default)
- unsigned
- single
- double

Note: If you have a **double** signal that does not align exactly to the message byte boundaries, to generate code with Embedded Coder you must check **Support**



**long long** under **Device Details** in the **Hardware Implementation** pane of the Configuration Parameters dialog.

### Multiplex type

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

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

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

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

In this example:

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

### Multiplex value

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

**Factor**

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 15-20 to understand how physical values are converted to raw values packed into a message.

**Offset**

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 15-20 to understand how physical values are converted to raw values packed into a message.

**Min, Max**

Define a range of signal values. The default settings are -Inf (negative infinity) and Inf, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

**Conversion Formula**

The conversion formula is

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

where `physical_value` is the original value of the signal, and `raw_value` is the packed signal value.

**See Also****Blocks**

CAN FD Configuration | CAN FD Transmit | CAN FD Unpack

**Functions**

canFDMessageBusType

**Introduced in R2018a**

## CAN FD Receive

Receive CAN FD messages from specified CAN FD device



## Library

Vehicle Network Toolbox: CAN FD Communication

## Description

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

---

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

---

The CAN FD Receive block has two output ports:

- The `f()` output port is a trigger to a Function-Call subsystem. If the block receives a new message, it triggers a Function-Call from this port. You can then connect to a Function-Call Subsystem to unpack and process a message.
- The `Msg` output port contains a CAN message received at that particular timestep. The block outputs messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

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

---

**Note** You cannot have more than one CAN FD Receive block in a model using the same PEAK-System device channel.

---

## Other Supported Features

The CAN FD Receive block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration” (Simulink).

The CAN FD Receive block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see “Code Generation” on page 15-22.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

### Shared Library Dependencies

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

To set up packNGo:

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

In this example, `gcs` is the current model that you want to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and there build the source code in the zip file to create an executable which can run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### Device

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

### Standard IDs Filter

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

- **Allow all** (default): Allows all standard IDs to pass the filter.
- **Allow only**: Allows only ID or range of IDs specified in the text field. You can specify a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to accept IDs from 400 through 500, and 600 through 650, enter `[[400:500] [600:650]]`. Standard IDs must be positive integers from 0 to 2047. You can also specify hexadecimal values with the `hex2dec` function.
- **Block all**: Blocks all standard IDs from passing the filter.

### Extended IDs Filter

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

- **Allow all** (default): Allows all extended IDs to pass the filter.
- **Allow only**: Allows only those IDs specified in the text field. You can specify a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to accept IDs from 3000 through 3500, and 3600 through 3620, enter `[[3000:3500] [3600:3620]]`. Extended IDs must be positive integers from 0 to 536870911. You can also specify hexadecimal values using the `hex2dec` function.
- **Block all**: Blocks all extended IDs from passing the filter.

### Sample time

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the CAN FD Receive block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify `-1` as your sample time. You can also specify a MATLAB variable for sample time. The default value is `0.01` (in seconds).

### Number of messages received at each timestep

Select how many messages the block receives at each specified timestep. Valid choices are:

- **all** (default): The CAN FD Receive block delivers all available messages in the FIFO buffer to the model during a specific timestep. The block generates one function call for each delivered message. The output port always contains one CAN message at a time.
- **1**: The CAN FD Receive block delivers one message per timestep from the FIFO buffer to the model.

If the block does not receive any messages before the next timestep, it outputs the last received message.

## See Also

### Blocks

CAN FD Configuration | CAN FD Transmit | CAN FD Unpack

### Functions

`canFDMessageType`

**Introduced in R2018a**

## CAN FD Replay

Replay logged CAN FD messages



### Library

Vehicle Network Toolbox: CAN FD Communication

### Description

The CAN FD Replay block replays logged messages from a `.mat` file to a CAN network or to Simulink as a bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink). You need a CAN FD Configuration block to replay to the network.

To replay messages logged in the MATLAB Command window in your Simulink model, convert them into a compatible format using `vntslgate` and save it to a separate file. For more information, see “Log and Replay CAN Messages”.

---

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

---

### Replay Timing

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



because simulation time and real time will be ahead of the timestamps in the replayed messages.

## Other Supported Features

- The CAN FD Replay block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see “Acceleration” (Simulink).
- The CAN FD Replay block supports the use of code generation along with the `packNGo` function to group required source code and dependent shared libraries. For more information, see “Code Generation” on page 15-27.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

### Shared Library Dependencies

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

To set up `packNGo`:

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

In this example, `gcs` is the current model that you want to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and there build the source code in the zip file to create an executable which can run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### File name

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

### Variable name

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

### Number of times to replay messages

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

### Replay messages to

Specify if the model is replaying messages to the CAN network or an output port. If replaying to the CAN network, you must also specify a device. If replaying to the model through an output port, the output is a Simulink bus signal.

### Device

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

**Sample time**

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the CAN FD Replay block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

**See Also****Functions**

canFDMessageBusType | canFDMessageReplayBlockStruct

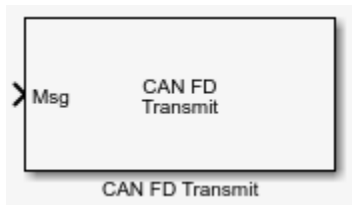
**Blocks**

CAN FD Log

**Introduced in R2018b**

## CAN FD Transmit

Transmit CAN FD message to selected CAN FD device



## Library

Vehicle Network Toolbox: CAN FD Communication

## Description

The CAN FD Transmit block transmits messages to the CAN network using the specified CAN device. The CAN FD Transmit block can transmit a single message or an array of messages during a given timestep. To transmit an array of messages from a signal bus, use a Bus Creator or Vector Concatenate, Matrix Concatenate block from the Simulink block library.

---

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

---

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

CAN is a peer-to-peer network, so when transmitting messages on a physical bus at least one other node must be present to properly acknowledge the message. Without another node, the transmission will fail as an error frame, and the device will continually retry to transmit.

## Other Supported Features

The CAN FD Transmit block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration” (Simulink).

The CAN FD Transmit block supports the use of code generation along with the `packNGo` function to group required source code and dependent shared libraries. For more information, see Code Generation on page 15-31.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

## Shared Library Dependencies

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

To set up `packNGo`:

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

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

run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### Device

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

Note: When using PEAK-System devices, CAN FD Transmit blocks in multiple enabled subsystems might skip some messages. If possible, replace the enabled subsystems with a different type of conditional subsystem, such as an if-action, switch-case-action, or triggered subsystem; or redesign your model so that all the CAN FD Transmit blocks are contained within a single enabled subsystem.

### Transmit messages periodically

Select this option to enable periodic transmission of the message on the configured channel at the specified message period. The period references real time, regardless of the Simulink model time step size (fundamental sample time) or block execution sample time. This is equivalent to the MATLAB function `transmitPeriodic`.

The periodic transmission is a nonbuffered operation. Only the last CAN message or set of messages present at the input of the CAN FD Transmit block is sent when the time period occurs.

### Message period (in seconds)

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

## See Also

### Blocks

[CAN FD Configuration](#) | [CAN FD Receive](#) | [CAN FD Pack](#)

**Introduced in R2018a**

## CAN FD Unpack

Unpack individual signals from CAN FD messages



## Library

Vehicle Network Toolbox: CAN FD Communication

## Description

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

---

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

---

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

## Other Supported Features

The CAN FD Unpack block supports

- The use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration” (Simulink).
- Code generation to deploy models to targets.



---

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

---

## Dialog Box

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

### Parameters

#### Data to be output as

Select your data signal:

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

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

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

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

---

**Note** For manually or CANdb specified signals, the default output signal data type is double. To specify other types, use a Signal Specification block. This allows the block to support the following output signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point types.

---

#### CANdb file

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

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

---

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

---

### **Message list**

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

## **Message**

### **Name**

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

### **Protocol mode**

Specify the message protocol mode as `CAN FD` or `CAN`.

### **Identifier type**

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

### **Identifier**

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

**Length (bytes)**

Specify the length of your message. For CAN messages the value can be 0-8 bytes; for CAN FD the value can be 0-8, 12, 16, 20, 24, 32, 48, or 64 bytes. If you are using CANdb specified signals for your output data, the CANdb file defines the length of your message. This option is available if you choose to output raw data or manually specify signals.

**Signals Table**

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

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

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

**Name**

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

**Start bit**

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. For CAN the start bit must be an integer from 0 through 63, for CAN FD 0 through 511, within the number of bits in the message. (Note that message length is specified in bytes.)

**Length (bits)**

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64. The sum of all the signal lengths in a message is limited to the number of bits in the message length; that is, all signals must cumulatively fit within the length of the message. (Note that message length is specified in bytes and signal length in bits.)

**Byte order**

Select either of the following options:

- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, proceeding to the next higher byte as you cross a byte boundary. For example, if

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

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

LSB

MSB

Data begins at the least significant bit and starts at 20

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

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

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

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

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

#### Data type

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

- signed (default)
- unsigned
- single
- double

Note: If you have a **double** signal that does not align exactly to the message byte boundaries, to generate code with Embedded Coder you must check **Support**

**long long** under **Device Details** in the **Hardware Implementation** pane of the Configuration Parameters dialog.

### Multiplex type

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

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

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

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

In this example:

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

### Multiplex value

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

**Factor**

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). See “Conversion Formula” on page 15-42 to understand how unpacked raw values are converted to physical values.

**Offset**

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. See “Conversion Formula” on page 15-42 to understand how unpacked raw values are converted to physical values.

**Min, Max**

Define a range of raw signal values. The default settings are -Inf (negative infinity) and Inf, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

**Output Ports**

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

**Output identifier**

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

**Output remote**

(Disabled for CAN FD protocol.) Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

**Output timestamp**

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

**Output length**

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

**Output error**

Select this option to output the message error status. This option adds a new output port to the block. An output value of 1 on this port indicates that the incoming message is an error frame; otherwise the output value is 0. The data type of this port is **uint8**.

**Output status**

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

**Output Bit Rate Switch (BRS)**

(Disabled for CAN protocol.) Select this option to output the message bitrate switch. This option adds a new output port to the block. The data type of this port is **boolean**.

**Output Error Status Indicator (ESI)**

(Disabled for CAN protocol.) Select this option to output the message error status. This option adds a new output port to the block. The data type of this port is **boolean**.

**Output Data Length Code (DLC)**

(Disabled for CAN protocol.) Select this option to output the message data length. This option adds a new output port to the block. The data type of this port is **double**.

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

**Conversion Formula**

The conversion formula is

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

where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.

**See Also****Blocks**

CAN FD Configuration | CAN FD Receive | CAN FD Pack

**Introduced in R2018a**



## CAN Log

Log received CAN messages



## Library

Vehicle Network Toolbox: CAN Communication

## Description

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

Configure your CAN Log block to log from the Simulink input port. For more information, see “Log and Replay CAN Messages”.

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

If you want to use messages logged using Simulink blocks in the MATLAB Command window, use `canMessage` to convert messages to the correct format.

---

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

---

---

**Note** You cannot have more than one CAN Log block in a model using the same PEAK-System device channel.

---

## Other Supported Features

The CAN Log block supports the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The CAN Log block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see “Code Generation” on page 15-44.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

## Shared Library Dependencies

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

To set up packNGo:

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

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

run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### File name

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

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

### Variable name

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

### Maximum number of messages to log

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

### Log messages from

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

### Device

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

**Sample time**

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the CAN Log block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

**See Also****Blocks**

CAN Replay

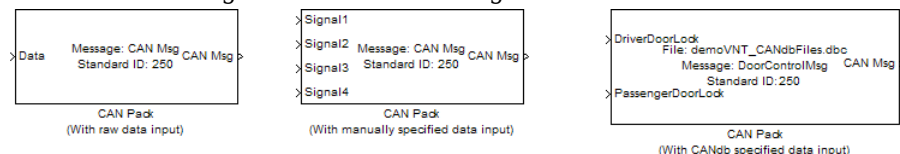
**Topics**

“Log and Replay CAN Messages”

**Introduced in R2011b**

# CAN Pack

Pack individual signals into CAN message



## Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

## Description

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

---

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

---

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

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

## Other Supported Features

The CAN Pack block supports:

- The use of Simulink Accelerator Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation to deploy models to targets.

---

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

---

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

## Dialog Box

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

### Parameters

#### Data is input as

Select your data signal:

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

---

**Note** The block supports the following input signals data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

---

#### CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The

message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

---

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

---

### Message list

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

## Message

### Name

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

### Identifier type

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

### Identifier

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

### Length (bytes)

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

**Remote frame**

Specify the CAN message as a remote frame.

Output as bus

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

**Signals Table**

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

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

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

**Name**

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

**Start bit**

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

**Length (bits)**

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

**Byte order**

Select either of the following options:

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



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

Diagram illustrating bit numbering for a CAN Pack. The table shows bit positions for each byte (Byte 0 to Byte 7). A red arrow points from bit 20 (labeled LSB) to bit 27 (labeled MSB), indicating that data is written up to the most significant bit and ends at 27. A text box notes: "Data begins at the least significant bit and starts at 20".

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

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

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

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

**Data type**

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

- signed (default)
- unsigned
- single
- double

**Multiplex type**

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

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

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

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

In this example:

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

### Multiplex value

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

### Factor

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 15-54 to understand how physical values are converted to raw values packed into a message.

**Offset**

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 15-54 to understand how physical values are converted to raw values packed into a message.

**Min, Max**

Define a range of signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

**Conversion Formula**

The conversion formula is

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

where `physical_value` is the original value of the signal, and `raw_value` is the packed signal value.

**See Also****Blocks**

CAN Unpack

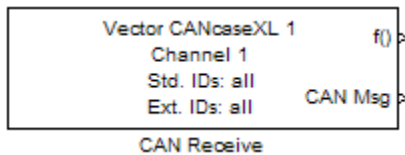
**Functions**

`canMessageBusType`

**Introduced in R2009a**

# CAN Receive

Receive CAN messages from specified CAN device



## Library

Vehicle Network Toolbox: CAN Communication

## Description

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

---

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

---

The CAN Receive block has two output ports:

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

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

---

**Note** You cannot have more than one CAN Receive block in a model using the same PEAK-System device channel.

---

## Other Supported Features

The CAN Receive block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The CAN Receive block supports the use of code generation along with the `packNGo` function to group required source code and dependent shared libraries. For more information, see “Code Generation” on page 15-56.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

### Shared Library Dependencies

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

To set up `packNGo`:

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

In this example, `gcs` is the current model that you want to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another machine and there build the source code in the zip file to create an executable which can run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### Device

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

### Standard IDs Filter

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

- **Allow all** (default): Allows all standard IDs to pass the filter.
- **Allow only**: Allows only ID or range of IDs specified in the text field. You can specify a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to accept IDs from 400 through 500, and 600 through 650, enter `[[400:500] [600:650]]`. Standard IDs must be positive integers from 0 to 2047. You can also specify hexadecimal values with the `hex2dec` function.
- **Block all**: Blocks all standard IDs from passing the filter.

### Extended IDs Filter

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

- **Allow all** (default): Allows all extended IDs to pass the filter.
- **Allow only**: Allows only those IDs specified in the text field. You can specify a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to accept IDs from 3000 through 3500, and 3600 through 3620, enter `[[3000:3500] [3600:3620]]`. Extended IDs must be positive integers from 0 to 536870911. You can also specify hexadecimal values using the `hex2dec` function.
- **Block all**: Blocks all extended IDs from passing the filter.

### **Sample time**

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the CAN Receive block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify `-1` as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

### **Number of messages received at each timestep**

Select how many messages the block receives at each specified timestep. Valid choices are:

- **all** (default): The CAN Receive block delivers all available messages in the FIFO buffer to the model during a specific timestep. The block generates one function call for each delivered message. The output port always contains one CAN message at a time.
- **1**: The CAN Receive block delivers one message per timestep from the FIFO buffer to the model.

If the block does not receive any messages before the next timestep, it outputs the last received message.

### **Output as bus**

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

## **See Also**

### **Blocks**

CAN Configuration | CAN Unpack



**Functions**

canMessageBusType

**Introduced in R2009a**

## CAN Replay

Replay logged CAN messages



## Library

Vehicle Network Toolbox: CAN Communication

## Description

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

To replay messages logged in the MATLAB Command window in your Simulink model, convert them into a compatible format using `vntslgate` and save it to a separate file. For more information, see “Log and Replay CAN Messages”.

---

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

---

## Replay Timing

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

## Other Supported Features

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

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

The CAN Replay block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see “Code Generation” on page 15-61.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

### Shared Library Dependencies

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

To set up packNGo:

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

In this example, gcs is the current model that you want to build. Building the model creates a zip file with the same name as model name. You can move this zip file to another

machine and there build the source code in the zip file to create an executable which can run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### File name

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

### Variable name

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

### Number of times to replay messages

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

### Replay messages to

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

### Device

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

### Sample time

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the CAN Replay block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify `-1` as your sample time. You can

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

#### Output as bus

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

## See Also

### Blocks

CAN Log

### Functions

canMessageBusType | canMessageReplayBlockStruct

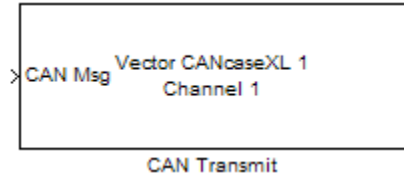
### Topics

“Log and Replay CAN Messages”

### Introduced in R2011b

## CAN Transmit

Transmit CAN message to selected CAN device



## Library

Vehicle Network Toolbox: CAN Communication

## Description

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

---

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

---

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

CAN is a peer-to-peer network, so when transmitting messages on a physical bus at least one other node must be present to properly acknowledge the message. Without another node, the transmission will fail as an error frame, and the device will continually retry to transmit.

## Other Supported Features

The CAN Transmit block supports the use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The CAN Transmit block supports the use of code generation along with the packNGo function to group required source code and dependent shared libraries. For more information, see Code Generation on page 15-65.

## Code Generation

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

### Code Generation with Simulink Coder

You can use Vehicle Network Toolbox, Simulink Coder, and Embedded Coder software together to generate code on the host end that you can use to implement your model. For more information on code generation, see “Build Process” (Simulink Coder).

## Shared Library Dependencies

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

To set up packNGo:

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

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

run independent of MATLAB and Simulink. The generated code compiles with both C and C++ compilers. For more information, see “Build Process Customization” (Simulink Coder).

---

**Note** On Linux platforms, you need to add the folder where you unzip the libraries to the environment variable `LD_LIBRARY_PATH`.

---

## Parameters

---

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

---

### Device

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

Note: When using PEAK-System devices, CAN Transmit blocks in multiple enabled subsystems might skip some messages. If possible, replace the enabled subsystems with a different type of conditional subsystem, such as an if-action, switch-case-action, or triggered subsystem; or redesign your model so that all the CAN Transmit blocks are contained within a single enabled subsystem.

### Transmit messages periodically

Select this option to enable periodic transmission of the message on the configured channel at the specified message period. The period references real time, regardless of the Simulink model time step size (fundamental sample time) or block execution sample time. This is equivalent to the MATLAB function `transmitPeriodic`.

The periodic transmission is a nonbuffered operation. Only the last CAN message or set of muxed messages present at the input of the CAN Transmit block is sent when the time period occurs.

### Message period (in seconds)

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



## See Also

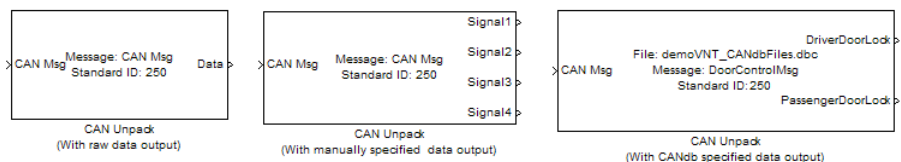
### Blocks

CAN Configuration | CAN Pack

**Introduced in R2009a**

## CAN Unpack

Unpack individual signals from CAN messages



## Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

## Description

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

---

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

---

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

## Other Supported Features

The CAN Unpack block supports:

- The use of Simulink Accelerator Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation to deploy models to targets.

---

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

---

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

## Dialog Box

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

### Parameters

#### Data to be output as

Select your data signal:

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

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

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

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

---

**Note** For manually or CANdb specified signals, the default output signal data type is double. To specify other types, use a Signal Specification block. This allows the block to

support the following output signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point types.

---

### **CANdb file**

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

---

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

---

### **Message list**

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

## **Message**

### **Name**

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

### **Identifier type**

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

### **Identifier**

Specify your CAN message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If

you specify `-1`, the block unpacks the messages that match the length specified for the message. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

**Length (bytes)**

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

**Signals Table**

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

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

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

**Name**

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

**Start bit**

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

**Length (bits)**

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

**Byte order**

Select either of the following options:

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

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

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

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

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

Diagram illustrating Big-Endian Byte Order. The table shows bit numbers for each byte. Red arrows indicate data flow: from bit 11 to bit 8 (MSB), and from bit 20 to bit 11 (LSB). Annotations state: "Data is written up to the most significant bit and ends at 11" and "Data begins at the least significant bit and starts at 20".

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

#### Data type

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

- signed (default)
- unsigned
- single
- double

#### Multiplex type

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

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

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

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

In this example:

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

### Multiplex value

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

### Factor

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). See “Conversion Formula” on page 15-76 to understand how unpacked raw values are converted to physical values.



**Offset**

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. See “Conversion Formula” on page 15-76 to understand how unpacked raw values are converted to physical values.

**Min, Max**

Define a range of raw signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

**Output Ports**

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

**Output identifier**

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

**Output remote**

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

**Output timestamp**

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

**Output length**

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

**Output error**

Select this option to output the message error status. This option adds a new output port to the block. An output value of 1 on this port indicates that the incoming message is an error frame; otherwise the output value is 0. The data type of this port is **uint8**.

**Output status**

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

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

## Conversion Formula

The conversion formula is

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

where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.

## See Also

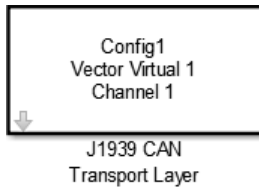
### Blocks

CAN Pack

**Introduced in R2009a**

# J1939 CAN Transport Layer

Transport J1939 messages via CAN



## Library

Vehicle Network Toolbox: J1939 Communication

## Description

The J1939 CAN Transport Layer block allows J1939 communication via a CAN bus. This block associates a user-defined J1939 network configuration with a connected CAN device. Use one block for each J1939 Network Configuration block in your model.

---

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

---

## Other Supported Features

The J1939 communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The J1939 communication blocks also support code generation with limited deployment capabilities. Code generation requires the Microsoft® C++ compiler.

## Parameters

### Config name

The name of the J1939 Network Configuration block to associate with.

### Device

The CAN device, chosen from all connected CAN devices.

### Bus speed

Speed of the CAN bus. The J1939 protocol specifies two rates of 250k and 500k. The default is 250000.

### Sample time

Simulation refresh rate. Specify the sampling time of the block during simulation. This value defines the frequency at which the J1939 CAN Transport Layer block runs during simulation. For information about simulation sample timing, see “What Is Sample Time?” (Simulink) If the block is inside a triggered subsystem or inherits a sample time, specify a value of -1. You can also specify a MATLAB variable for sample time. The default value is 0.01 seconds.

## See Also

### Blocks

J1939 CAN Transport Layer | J1939 Node Configuration | J1939 Receive | J1939 Transmit

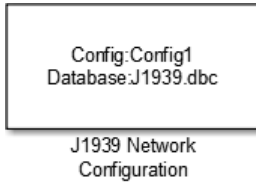
### Topics

“Basic J1939 Communication over CAN”

### Introduced in R2015b

# J1939 Network Configuration

Define J1939 network configuration name and database file



## Library

Vehicle Network Toolbox: J1939 Communication

## Description

The J1939 Network Configuration block is where you define a configuration name and specify the associated user-supplied J1939 database. You can include more than one block per model, each corresponding to a unique configuration on the CAN bus.

---

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

---

## Other Supported Features

The J1939 communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The J1939 communication blocks also support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Parameters

### Configuration name

Define a name for this J1939 network configuration. The default is `ConfigX`, where the number `X` automatically increases from 1 based on the number of existing blocks.

### Database File

Specify the J1939 database file name relative to the current folder. For example, enter `J1939.dbc` if the file is in the current folder; otherwise enter the full path with the file name, such as `C:\work\J1939.dbc`.

The database file defines the J1939 parameter groups and nodes, and must be in the `.dbc` format defined by Vector Informatik GmbH.

## See Also

### Blocks

J1939 CAN Transport Layer | J1939 Node Configuration | J1939 Receive | J1939 Transmit

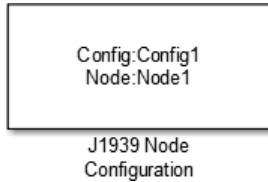
### Topics

“Basic J1939 Communication over CAN”

### Introduced in R2015b

# J1939 Node Configuration

Configure J1939 node with address and network management attributes



## Library

Vehicle Network Toolbox: J1939 Communication

## Description

The J1939 Node Configuration block is where you define a node and associate it with a specific network configuration. Its Message information is read from the database for that configuration, unless you are creating and configuring a custom node.

---

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

---

## Other Supported Features

The J1939 communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The J1939 communication blocks also support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Block Outputs

### Address (optional)

This output port exists when you check **Output current node address** in the dialog box. It returns the effective address of the node.

### AC Status (optional)

This output port exists when you check **Output address claim status** in the dialog box. It indicates the success (1) or failure (0) of the node's address claim.

## Parameters

### Config name

The ID of the J1939 network configuration to associate this node with. This is used to access the corresponding J1939 database.

### Node name

The name of this J1939 node. The available list shows **none** if no J1939 network configuration is found or no node is defined in the associated database. If you are creating a custom node, the node name must be unique within its J1939 network configuration.

### Message

These are the nine network attributes as defined by the database file consistent with the J1939 protocol. These parameters are read-only unless you are defining a custom node.

- **Allow arbitrary address** — Allow/disallow the node to switch to an arbitrary address if the station address is not available. If this option is off and the node loses its address claim, the node goes silent.
  - Node Address** — Station address, decimal, 8-bit.
- **Industry Group** — Decimal, 3-bit.
- **Vehicle System** — Decimal, 7-bit.
- **Vehicle System Instance** — Identifies one particular occurrence of a given vehicle system in a given network. If only one instance of a certain vehicle system exists in a network, then this field must be set to 0 to define it as the first instance. Decimal, 4-bit.



- **Function ID** — Decimal, 8-bit.
- **Function Instance** — Identifies the particular occurrence of a given function in a vehicle system and given network. If only one instance of a certain function exists in a network, then this field must be set to 0 to define it as the first instance.  
Decimal, 5-bit.
- **ECU Instance** — This 3-bit field is used when multiple electronic control units (ECU) are involved in performing a single function. If only one ECU is used for a particular controller application (CA), then this field must be set to 0 to define it as the first instance.
- **Manufacturer Code** — Decimal, 11-bit.
- **Identity Number** — Decimal, 21-bit.

### Sample time

Simulation refresh rate. Specify the sampling time of the block during simulation. This value defines the frequency at which the J1939 Node Configuration updates its optional output ports. If the block is inside a triggered subsystem or inherits a sample time, specify a value of -1. You can also specify a MATLAB variable for sample time. The default value is 0.01 seconds. For information about simulation sample timing, see “What Is Sample Time?” (Simulink)

### Output current node address

Enable or disable the **Address** output port to show the effective address. The effective address is different from the predefined station address if Allow arbitrary address is selected, a name conflict occurs, and the current node has lower priority. The output signal is a double value from 0 to 253. This port is disabled by default.

### Output address claim status

Enable or disable the address claim **AC Status** output port to show the success of an address claim. The output value is binary, 1 for success or 0 for failure. This port is disabled by default.

## See Also

### Blocks

J1939 CAN Transport Layer | J1939 Network Configuration | J1939 Receive | J1939 Transmit

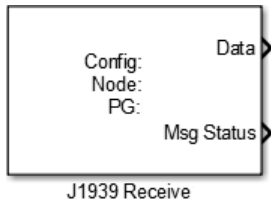
**Topics**

“Basic J1939 Communication over CAN”

**Introduced in R2015b**

# J1939 Receive

Receive J1939 parameter group messages



## Library

Vehicle Network Toolbox: J1939 Communication

## Description

The J1939 Receive block receives a J1939 message from the configured CAN device. The J1939 database file defines the nodes and parameter groups. You specify the J1939 database with the J1939 Network Configuration block.

---

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

---

## Other Supported Features

The J1939 communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The J1939 communication blocks also support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Block Outputs

### Data

Depending on the J1939 parameter group defined in the J1939 database file, the block can have multiple data output signal ports. The block output data type is double.

### Msg Status

When **Output New Message Received status** is checked in the dialog box, this port outputs 1 when a new message is received from the CAN bus; otherwise, outputs 0.

## Parameters

### Config name

The name of the J1939 network configuration to associate with. This is used to access the corresponding J1939 database. Only the nodes defined in the model and associated with the specified J1939 network configuration appear in the Node name list. The option shows none if no J1939 network configuration is found.

### Node name

The name of the J1939 node. The drop-down list includes all the nodes in the model, both custom nodes and nodes from the database.

### Parameter Group

The parameter group number (PGN) and name from the database. The contents of this list vary depending on the parameter groups that the J1939 database file specifies. The default is the first parameter group for the selected node.

---

**Note** If you change any parameter group settings within your J1939 database file, you must then open the J1939 Receive block dialog box and select the same **Parameter Group**, then click **OK** or **Apply** to update the parameter group information in the block.

---

### Signals

Signals defined in the parameter group. The **Min** and **Max** settings are read from the database, but by default the block does not clip signal values that exceed this range.

### Source Address Filter

Filter messages based on source address:

- `Allow only` — Lets you specify a single source address of interest.
- `Allow all` — Accepts messages from any source address. This is the default.

### **Destination Address Filter**

Filter out message based on destination address:

- `global only` — Receive only broadcast messages.
- `node specific only` — Receive only messages addressed to this node.
- `global and specific` — Receive all broadcast and node-addressed messages. This is the default.

### **Sample time**

Simulation refresh rate. Specify the sampling time of the block during simulation. This value defines the frequency at which the J1939 Receive updates its output ports. If the block is inside a triggered subsystem or inherits a sample time, specify a value of `-1`. You can also specify a MATLAB variable for sample time. The default value is `0.01` seconds. For information about simulation sample timing, see “What Is Sample Time?” (Simulink)

### **Output New Message Received status**

Select this check box to create a **Msg Status** output port. Its output signal indicates a new incoming message, showing 1 for a new message received, or 0 when there is no new message.

## **See Also**

### **Blocks**

J1939 CAN Transport Layer | J1939 Network Configuration | J1939 Node Configuration | J1939 Transmit

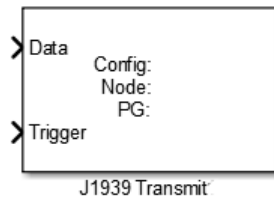
### **Topics**

“Basic J1939 Communication over CAN”

### **Introduced in R2015b**

## J1939 Transmit

Transmit J1939 message



## Library

Vehicle Network Toolbox: J1939 Communication

## Description

The J1939 Transmit block transmits a J1939 message. The J1939 database file defines the nodes and parameter groups. You specify the J1939 database with the J1939 Network Configuration block.

---

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

---

## Other Supported Features

The J1939 communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The J1939 communication blocks also support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Block Inputs

- **Data**

Depending on the J1939 parameter group and signals defined in the J1939 database file, the block can have multiple data input ports.

- **Trigger**

Enables the transmission of the message for that sample. A value of 1 specifies to send, a value of 0 specifies not to send.

## Parameters

### Config name

The name of the J1939 network configuration to associate with. This is used to access the corresponding J1939 database. Only the nodes defined in the model and associated with the specified J1939 network configuration appear in the Node name list. The option shows none if no J1939 network configuration is found.

### Node name

The name of the J1939 node. The drop-down list includes all the nodes in the model, both custom nodes and nodes from the database.

### Parameter Group

The parameter group number (PGN) and name from the database. The contents of this list vary depending on the parameter groups that the J1939 database file specifies. The default is the first parameter group for the selected node.

---

**Note** If you change any parameter group settings within your J1939 database file, you must then open the J1939 Transmit block dialog box and select the same **Parameter Group**, then click **OK** or **Apply** to update the parameter group information in the block.

---

### Signals

Signals defined in the parameter group. The **Min** and **Max** settings are read from the database, but by default the block does not clip signal values that exceed this range.

**PG Priority**

Priority of the parameter group, read from the database. This priority setting resolves clashes of multiple parameter groups transmitting on the same bus at the same time. If a conflict occurs, the priority group with lower priority (i.e., higher value) will refrain from transmitting. The value can range from 0 (highest priority) to 7 (lowest).

**Destination Address**

Name of the destination node. The default is the first node defined in the database, otherwise **Custom**.

For a custom destination address, you can specify 0-253 for the address of the destination node. For broadcasting to all nodes, use the **Custom Destination Address** setting with an address of 255.

**See Also****Blocks**

J1939 CAN Transport Layer | J1939 Network Configuration | J1939 Node Configuration | J1939 Receive

**Topics**

“Basic J1939 Communication over CAN”


**Introduced in R2015b**



# XCP CAN Configuration

Configure XCP slave connection

**Library:** Simulink Real-Time / XCP / CAN  
Vehicle Network Toolbox / XCP Communication / CAN

A rectangular icon with a black border containing the text "XCP CAN Configuration" in a sans-serif font.

## Description

The XCP CAN Configuration block uses the parameters specified in the A2L file and the ASAP2 database to establish an XCP slave connection.

Before you acquire or stimulate data, specify the A2L file to use in your XCP CAN Configuration . Use one XCP CAN Configuration to configure one slave connection for data acquisition or stimulation. If you add XCP CAN Data Acquisition and XCP CAN Data Stimulation blocks, your model checks to see if there is a corresponding XCP CAN Configuration block. If there is no corresponding XCP CAN Configuration block, the model prompts to add one.

## Other Supported Features

The XCP CAN communication blocks support Simulink accelerator mode and rapid accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information about these simulation modes, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Parameters

**Config name — Specify XCP CAN session name**  
'CAN\_Config1' (default)

Specify a unique name for your XCP CAN session.

**Programmatic Use**

SlaveName

**A2L File — Select an A2L file**

file name

Click **Browse** to select an A2L file for your XCP CAN session.

**Programmatic Use**

A2LFile

**Enable seed/key security — Select that key required to establish connection**

'off'

Select this option if your slave requires a secure key to establish connection. Select a file that contains the seed/key definition to enable the security.

**Programmatic Use**

EnableSecurity

**File (\*.DLL) — Select file for seed and key security**

file name

If you select **Enable seed/key security** (EnableSecurity), this field is enabled. Click **Browse** to select the file that contains seed and key security algorithm that unlocks an XCP slave module.

**Programmatic Use**

SeedKeyLib

**Output connection status — Display connection status**

'off'

Select this option to display the status of the connection to the slave module. Selecting this option adds a new output port.

**Programmatic Use**

EnableStatus

## See Also

### Blocks

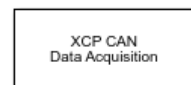
[XCP CAN Data Acquisition](#) | [XCP CAN Data Stimulation](#) | [XCP CAN Transport Layer](#)

**Introduced in R2013a**

## XCP CAN Data Acquisition

Acquire selected measurements from configured slave connection

**Library:** Simulink Real-Time / XCP / CAN  
Vehicle Network Toolbox / XCP Communication / CAN



### Description

The XCP CAN Data Acquisition block acquires data from the configured slave connection based on the selected measurements. The block uses the XCP CAN transport layer to obtain raw data for the selected measurements at the specified simulation time step. Configure your XCP connection and use the XCP CAN Data Acquisition block to select your event and measurements for the configured slave connection. The block displays the selected measurements as output ports.

### Other Supported Features

The XCP communication blocks support the use of Simulink accelerator mode and rapid accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on these simulation modes, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

### Parameters

#### **Config name — Specify XCP CAN session name**

select from list

Select the name of XCP configuration you want to use. The list displays all available names specified in the XCP CAN Configuration blocks in the model. Selecting a configuration displays events and measurements available in the A2L file of this configuration.

---

**Note** You can acquire measurements for only one event by using an XCP CAN Data Acquisition block. Use one block for each event whose measurements you want to acquire.

---

### Programmatic Use

SlaveName

### Event name — Select an event

select from list

Select an event from the available list of events. The XCP CAN Configuration block uses the specified A2L file to populate the events list.


### Programmatic Use

EventName

### All Measurements — List all measurements available for event

measurements list

This list displays all measurements available for the selected event. Select the

measurement that you want to use and click the add button,  to add it to the selected measurements. Hold the **Ctrl** key on your keyboard to select multiple measurements.

In the **Block Parameters** dialog box, type the name of the measurement you want to use in the **Search** box. The **All Measurements** lists displays a list of all matching names. Click the x

x


to clear your search.


### Programmatic Use

AllMeasurements

### Selected Measurements — List selected measurements

measurement names

This list displays selected measurements. To remove a measurement from this list, select the measurement and click the remove button, .

In the **Block Parameters** dialog box, use the toggle buttons  to reorder the selected measurements.

### **Programmatic Use**

SelectedMeasurements

### **DAQ List Priority — Specify a priority value for slave device driver**

priority value

Specify a priority value as an integer from 0 to 255 for the slave device driver to prioritize transmission of data packets. The slave can accumulate XCP packets for lower priority DAQ lists before transmission to the master. A value of 255 has the highest priority. The SET\_DAQ\_LIST\_MODE command communicates the **DAQ List Priority** value from master to slave. This communication method differs from the specification of the Event Channel Priority property, which comes from the A2L file.

### **Programmatic Use**

DAQPriority

### **Sample time — Specify sampling time of block**

0.01 (default)

Specify the sampling time of the block during simulation, which is the simulation time. This value defines the frequency at which the XCP CAN Data Acquisition block runs during simulation. If the block is inside a triggered subsystem or is to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

### **Programmatic Use**

SampleTime

### **Enable Timestamp — Enable reading timestamp from incoming DTO packets**

off (default) | on

When the Timestamp is enabled, the block reads the timestamp from incoming DTO packets and outputs the timestamp to Simulink. The **Enable Timestamp** check box appears in the block parameters dialog box when the parameter is supported in the A2L file.

### **Programmatic Use**

EnableTimestamp

## **See Also**

### **Blocks**

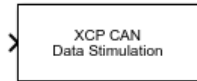
XCP CAN Configuration | XCP CAN Data Stimulation | XCP CAN Transport Layer

**Introduced in R2013a**

## XCP CAN Data Stimulation

Perform data stimulation on selected measurements

**Library:** Simulink Real-Time / XCP / CAN  
Vehicle Network Toolbox / XCP Communication / CAN



### Description

The XCP CAN Data Stimulation block sends data to the selected slave connection for the selected event measurements. The block uses the XCP CAN transport layer to output raw data for the selected measurements at the specified stimulation time step. Configure your XCP session and use the XCP CAN Data Stimulation block to select your event and measurements on the configured slave connection. The block displays the selected measurements as input ports.

### Other Supported Features

The XCP communication blocks support Simulink accelerator mode and rapid accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information about these simulation modes, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

### Parameters

**Config name — Specify XCP CAN session name**

select from list

Select the name of XCP configuration that you want to use. The list displays all available names specified in the available XCP CAN Configuration blocks in the model. Selecting a configuration displays events and measurements available in the A2L file of this configuration.



---

**Note** You can stimulate measurements for only one event by using an XCP CAN Data Stimulation block. Use one block for each event whose measurements you want to stimulate.

---

### Programmatic Use

SlaveName

### Event name — Select an event

select from list

Select an event from the event list. The XCP CAN Configuration block uses the specified A2L file to populate the events list.


### Programmatic Use

EventName

### All Measurements — List all measurements available for event

measurements list

This list displays all measurements available for the selected event. Select the

measurement that you want to use and click the add button,  to move it to the selected measurements. Hold the **Ctrl** key on your keyboard to select multiple measurements.

In the block parameters dialog box, type the name of the measurement you want to use in the **Search** box. The **All Measurements** lists displays a list of all matching names. Click the **x**

x


to clear your search.



### Programmatic Use

AllMeasurements

### Selected Measurements — List selected measurements

measurement names

This list displays your selected measurements. To remove a measurement from this list, select the measurement and click the remove button, .

In the **Block Parameters** dialog box, use the toggle buttons   to reorder the selected measurements.

**Programmatic Use**

SelectedMeasurements

**Enable Timestamp – Enable sending Simulink timestamp in STIM DTO packets**  
off (default) | on

When the Timestamp is enabled, the block inputs a timestamp from Simulink and sends the timestamp in the STIM DTO packets. The **Enable Timestamp** check box appears in the block parameters dialog box when the parameter is supported in the A2L file.

**Programmatic Use**

EnableTimestamp

## See Also

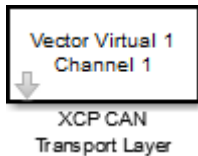
**Blocks**

XCP CAN Configuration | XCP CAN Data Acquisition | XCP CAN Transport Layer

**Introduced in R2013a**

# XCP CAN Transport Layer

Transport XCP messages via CAN



## Library

Vehicle Network Toolbox: CAN Communication

Vehicle Network Toolbox: XCP Communication

## Description

The XCP CAN Transport Layer subsystem uses the specified device to transport and receive XCP messages.

Use this block with an XCP Data Acquisition block to acquire and analyze specific XCP messages. Use this block with an XCP Data Stimulation block to send specific information to modules.

## Other Supported Features

The XCP communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Parameters

### Device

+

Select a CAN device from the list of devices available to your system.

### Bus speed

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

### Sample time

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the XCP CAN Transport Layer subsystem and the underlying blocks run during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

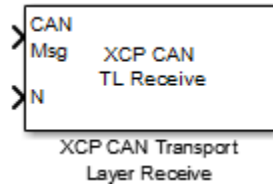
### See Also

[XCP CAN Configuration](#) | [XCP CAN Data Acquisition](#) | [XCP CAN Data Stimulation](#)

**Introduced in R2013a**

# XCP CAN Transport Layer Receive

Receive XCP messages via CAN device



## Description

The XCP CAN Transport Layer Receive block receives XCP messages from a CAN Receive block.

## Other Supported Features

The XCP communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Parameters

### Sample time

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the XCP CAN Transport Layer Receive block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is -1 (in seconds).

## **See Also**

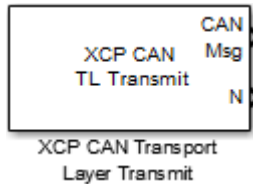
### **Blocks**

CAN Receive | XCP CAN Transport Layer | XCP CAN Transport Layer Transmit

**Introduced in R2013a**

# XCP CAN Transport Layer Transmit

Transmit queued XCP messages



## Description

The XCP CAN Transport Layer Transmit block connects to a CAN Transmit block to transmit queued XCP messages.

## Other Supported Features

The XCP communication blocks support the use of Simulink Accelerator and Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on this feature, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Parameters

### Maximum number of messages

Enter the maximum number of messages the block can transmit. Value must be a positive integer.

### Sample time

Specify the sampling time of the block during simulation, which is the simulation time as described by the Simulink documentation. This value defines the frequency at which the XCP CAN Transport Layer block runs during simulation. If the block is inside a triggered subsystem or to inherit sample time, you can specify -1 as your

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

## **See Also**

### **Blocks**

[CAN Transmit](#) | [XCP CAN Transport Layer](#) | [XCP CAN Transport Layer Receive](#)


**Introduced in R2013a**



# XCP UDP Configuration

Configure XCP UDP slave connection

**Library:** Simulink Real-Time / XCP / UDP  
Vehicle Network Toolbox / XCP Communication / UDP

A rectangular icon with a black border containing the text "XCP UDP Configuration" in a sans-serif font.

## Description

The XCP UDP Configuration block uses the parameters specified in the A2L file and the ASAP2 database to establish an XCP slave connection.

Before you acquire or stimulate data, specify the A2L file to use in your XCP UDP Configuration . Use one XCP UDP Configuration to configure one slave connection for data acquisition or stimulation. If you add XCP UDP Data Acquisition and XCP UDP Data Stimulation blocks, your model checks to see if there is a corresponding XCP UDP Configuration block. If there is no corresponding XCP CAN Configuration block, the model prompts to add one.

## Other Supported Features

The XCP UDP communication blocks support Simulink accelerator mode and rapid accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information about these simulation modes, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

## Parameters

**Config name — Specify XCP UDP session name**

'UDP\_Config1' (default)

Specify a unique name for your XCP session.

**Programmatic Use**

SlaveName

**A2L File — Select an A2L file**

file name

Click **Browse** to select an A2L file for your XCP session.

**Programmatic Use**

A2LFile

**Enable seed/key security — Select that key required to establish connection**

'off'

Select this option if your slave requires a secure key to establish connection. Select a file that contains the seed/key definition to enable the security.

**Programmatic Use**

EnableSecurity

**File (\*.DLL) — Select file for seed and key security**

file name

If you select **Enable seed/key security**, this field is enabled. Click **Browse** to select the file that contains seed and key security algorithm that unlocks an XCP slave module.

**Programmatic Use**

SeedKeyLib

**Output connection status — Display connection status**

'off'

Select this option to display the status of the connection to the slave module. Selecting this option adds a new output port.

**Programmatic Use**

EnableStatus

## See Also

### Blocks

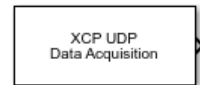
XCP UDP Data Acquisition | XCP UDP Data Stimulation

**Introduced in R2019a**

## XCP UDP Data Acquisition

Acquire selected measurements from configured slave connection

**Library:** Simulink Real-Time / XCP / UDP  
Vehicle Network Toolbox / XCP Communication / UDP



### Description

The XCP UDP Data Acquisition block acquires data from the configured slave connection based on the selected measurements. The block uses the XCP UDP transport layer to obtain raw data for the selected measurements at the specified simulation time step. Configure your XCP connection and use the XCP UDP Data Acquisition block to select your event and measurements for the configured slave connection. The block displays the selected measurements as output ports.

### Other Supported Features

The XCP communication blocks support the use of Simulink accelerator mode and rapid accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information on these simulation modes, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

### Parameters

**Config name — Specify XCP UDP session name**

select from list

Select the name of XCP configuration you want to use. The list displays all available names specified in the XCP UDP Configuration blocks in the model. Selecting a configuration displays events and measurements available in the A2L file of this configuration .

---

**Note** You can acquire measurements for only one event by using an XCP UDP Data Acquisition block. Use one block for each event whose measurements you want to acquire.

---

### Programmatic Use

SlaveName

### Event name — Select an event

select from list

Select an event from the available list of events. The XCP UDP Configuration block uses the specified A2L file to populate the events list.


### Programmatic Use

EventName

### All Measurements — List all measurements available for event

measurements list

This list displays all measurements available for the selected event. Select the

measurement that you want to use and click the add button,  to add it to the selected measurements. Hold the **Ctrl** key on your keyboard to select multiple measurements.

In the **Block Parameters** dialog box, type the name of the measurement you want to use in the **Search** box. The **All Measurements** lists displays a list of all matching names. Click the x

x


to clear your search.


### Programmatic Use

AllMeasurements

### Selected Measurements — List selected measurements

measurement names

This list displays selected measurements. To remove a measurement from this list, select the measurement and click the remove button, .

In the **Block Parameters** dialog box, use the toggle buttons  to reorder the selected measurements.

### **Programmatic Use**

SelectedMeasurements

### **DAQ List Priority — Specify a priority value for slave device driver**

priority value

Specify a priority value as an integer from 0 to 255 for the slave device driver to prioritize transmission of data packets. The slave can accumulate XCP packets for lower priority DAQ lists before transmission to the master. A value of 255 has the highest priority. The SET\_DAQ\_LIST\_MODE command communicates the **DAQ List Priority** value from master to slave. This communication method differs from the specification of the Event Channel Priority property, which comes from the A2L file.

### **Programmatic Use**

DAQPriority

### **Sample time — Specify sampling time of block**

0.01 (default)

Specify the sampling time of the block during simulation, which is the simulation time. This value defines the frequency at which the XCP UDP Data Acquisition block runs during simulation. If the block is inside a triggered subsystem or is to inherit sample time, you can specify -1 as your sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 (in seconds).

### **Programmatic Use**

SampleTime

### **Enable Timestamp — Enable reading timestamp from incoming DTO packets**

off (default) | on

When the Timestamp is enabled, the block reads the timestamp from incoming DTO packets and outputs the timestamp to Simulink. The **Enable Timestamp** check box appears in the block parameters dialog box when the parameter is supported in the A2L file.

### **Programmatic Use**

EnableTimestamp

## **See Also**

### **Blocks**

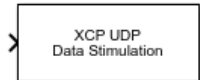
XCP UDP Configuration | XCP UDP Data Stimulation

**Introduced in R2019a**

## XCP UDP Data Stimulation

Perform data stimulation on selected measurements

**Library:** Simulink Real-Time / XCP / UDP  
Vehicle Network Toolbox / XCP Communication / UDP



### Description

The XCP UDP Data Stimulation block sends data to the selected slave connection for the selected event measurements. The block uses the XCP UDP transport layer to output raw data for the selected measurements at the specified stimulation time step. Configure your XCP session and use the XCP UDP Data Stimulation block to select your event and measurements on the configured slave connection. The block displays the selected measurements as input ports.

### Other Supported Features

The XCP communication blocks support Simulink accelerator mode and rapid accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information about these simulation modes, see the Simulink documentation.

The XCP communication blocks support code generation with limited deployment capabilities. Code generation requires the Microsoft C++ compiler.

### Parameters

**Config name — Specify XCP UDP session name**

select from list

Select the name of XCP configuration that you want to use. The list displays all available names specified in the available XCP UDP Configuration blocks in the model. Selecting a configuration displays events and measurements available in the A2L file of this configuration.



---

**Note** You can stimulate measurements for only one event by using an XCP UDP Data Stimulation block. Use one block for each event whose measurements you want to stimulate.

---

### Programmatic Use

SlaveName

### Event name — Select an event

select from list

Select an event from the event list. The XCP UDP Configuration block uses the specified A2L file to populate the events list.


### Programmatic Use

EventName

### All Measurements — List all measurements available for event

measurements list

This list displays all measurements available for the selected event. Select the

measurement that you want to use and click the add button,  to move it to the selected measurements. Hold the Ctrl key on your keyboard to select multiple measurements.

In the block parameters dialog box, type the name of the measurement you want to use. The **All Measurements** lists displays a list of all matching names. Click the x


to clear your search.



### Programmatic Use

AllMeasurements

### Selected Measurements — List selected measurements

measurement names

This list displays your selected measurements. To remove a measurement from this list, select the measurement and click the remove button, .

In the **Block Parameters** dialog box, use the toggle buttons   to reorder the selected measurements.

**Programmatic Use**

SelectedMeasurements

**Enable Timestamp — Enable sending Simulink timestamp in STIM DTO packets**  
off (default) | on

When the Timestamp is enabled, the block inputs a timestamp from Simulink and sends the timestamp in the STIM DTO packets. The **Enable Timestamp** check box appears in the block parameters dialog box when the parameter is supported in the A2L file.

**Programmatic Use**

EnableTimestamp

## See Also

**Blocks**

XCP UDP Configuration | XCP UDP Data Acquisition

**Introduced in R2019a**