Vehicle Network Toolbox™

User's Guide

MATLAB® & SIMULINK®

R2021b

MathWorks®

How to Contact MathWorks

Latest news:	www.mathworks.com
Sales and services:	www.mathworks.com/sales_and_services
User community:	www.mathworks.com/matlabcentral
Technical support:	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

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)
September 2019	Online only	Revised for Version 4.3 (Release 2019b)
March 2020	Online only	Revised for Version 4.4 (Release 2020a)
September 2020	Online only	Revised for Version 4.5 (Release 2020b)
March 2021	Online only	Revised for Version 5.0 (Release 2021a)
September 2021	Online only	Revised for Version 5.1 (Release 2021b)

Contents

1 Getting Started

2 Hardware Support Package Installation

3 CAN Communication Workflows

4 Using a CAN Database

5 XCP Communication Workflows

6 Universal Measurement & Calibration Protocol (XCP)

7 J1939

8 CAN Communications in Simulink

9 Hardware Limitations

10 XCP Communications in Simulink

11 Functions

12 Properties by Class

13 Blocks

14 Vehicle Network Toolbox Examples

Getting Started

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 for sending, receiving, encoding, and decoding 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 Explorer and CAN FD Explorer apps. 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 NI®.

Toolbox Characteristics and Capabilities

In this section...

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

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" 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

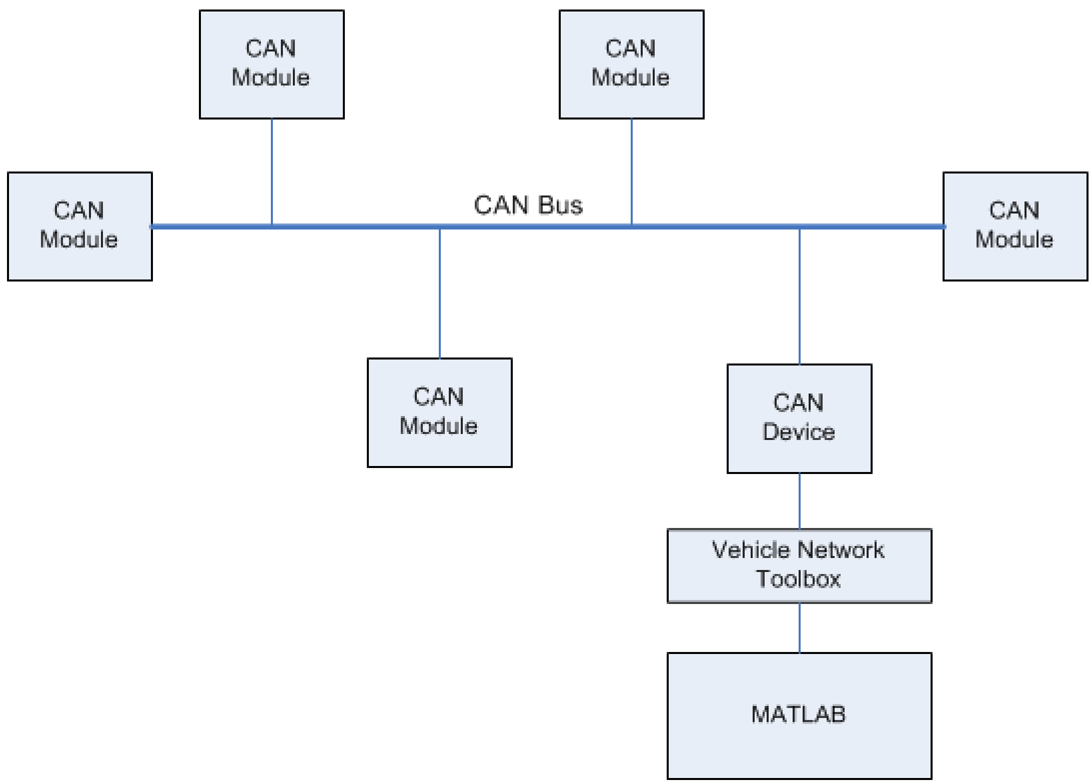
With Vehicle Network Toolbox block library and other blocks from the Simulink library, you can create sophisticated models to connect to a live network and to simulate message traffic on a CAN bus.

Visualize CAN Communication

Using the **CAN Explorer** or **CAN FD Explorer** app, 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 **CAN Explorer** or **CAN FD Explorer**.

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.

MathWorks Virtual Channels

Description

To facilitate code prototyping and model simulation without hardware, Vehicle Network Toolbox provides a MathWorks® virtual CAN device with two channels. These channels are identified with the vendor `"MathWorks"` and the device `"Virtual 1"`, and are accessible in both MATLAB and Simulink.

These virtual channels support CAN, CAN FD, and J1939 communication on Windows®, and support CAN and CAN FD on Linux®. Many examples throughout the documentation show how to use these virtual channels, so that you can run them on your own system.

The two virtual channels belong to a common device, so you could send a message on channel 1 and have that message received on channel 1 and channel 2. But because the virtual device is an application-level representation of a CAN/CAN FD bus without an actual bus, the following limitations apply:

- The virtual interface does not perform low level protocol activity like arbitration, error frames, acknowledgment, and so on.
- Although you can connect multiple channels of the same virtual device in the same MATLAB session or in Simulink models running in that MATLAB session, you cannot use virtual channels to communicate between different MATLAB sessions.

Examples

You can view the device and channels in MATLAB with the `canChannelList` function.

```
canChannelList
```

```
ans =

  2×6 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"
```

Create a virtual CAN channel.

```
canch = canChannel("MathWorks","Virtual 1",1);
```

Create a virtual CAN FD channel.

```
canfdch = canFDChannel("MathWorks","Virtual 1",2);
```

Create a virtual J1939 channel.

```
db = canDatabase([(matlabroot) '/examples/vnt/data/J1939.dbc']);
jch = j1939Channel(db,"MathWorks","Virtual 1",1);
```

See Also

Functions
`canChannelList` | `canChannel` | `j1939Channel`

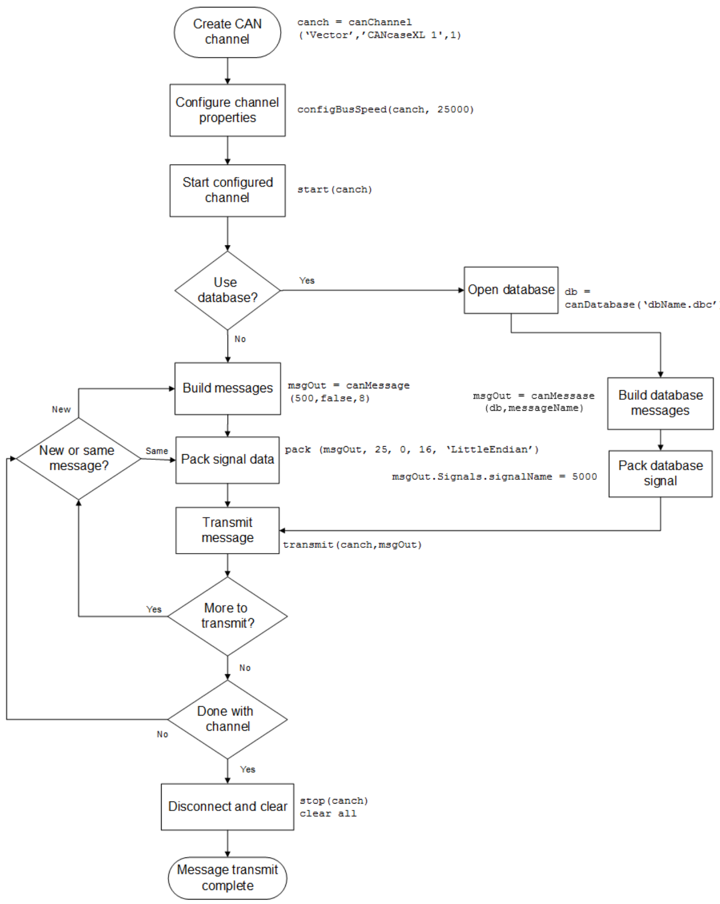
More About

- "Transmit and Receive CAN Messages" on page 1-10

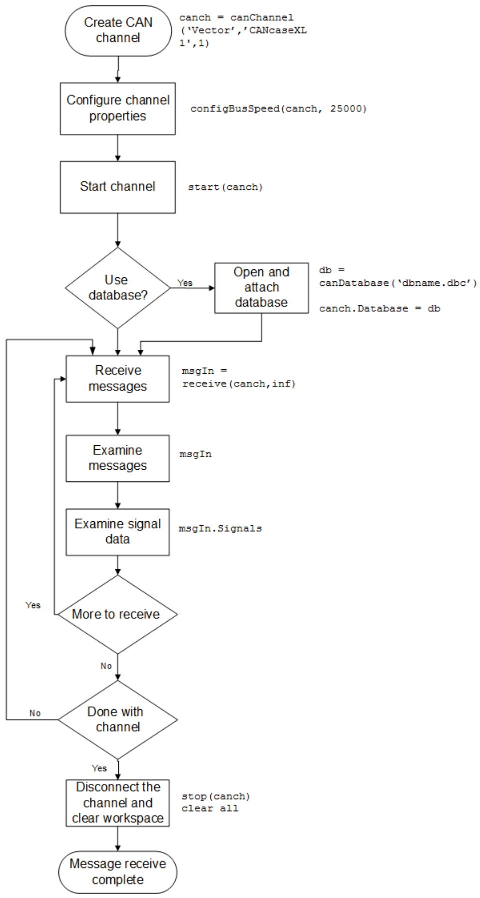
Vehicle Network Communication in MATLAB

Workflows in this section are sequential to help you understand how the communication works.

Transmit Workflow



Receive Workflow



See Also

More About

- "Transmit and Receive CAN Messages" on page 1-10

Transmit and Receive CAN Messages

In this section...

Discover Installed Hardware

In the example, you discover your system CAN devices with `canChannelList`, 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 = canChannelList
```

```
info =

  14×6 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"        "VN1610 1"                 1     "VN1610"               "CAN, CAN FD"    "18959"
 "Vector"        "VN1610 1"                 2     "VN1610"               "CAN, CAN FD"    "18959"
 "Vector"        "Virtual 1"                1     "Virtual"              "CAN, CAN FD"    "0"
 "Vector"        "Virtual 1"                2     "Virtual"              "CAN, CAN FD"    "0"
 "PEAK-System"   "PCAN-USB Pro"             1     "PCAN-USB Pro"         "CAN, CAN FD"    "0"
 "PEAK-System"   "PCAN-USB Pro"             2     "PCAN-USB Pro"         "CAN, CAN FD"    "0"
 "Kvaser"        "USBcan Professional 1"    1     "USBcan Professional"  "CAN"            "10680"
 "Kvaser"        "USBcan Professional 1"    1     "USBcan Professional"  "CAN"            "10680"
 "Kvaser"        "Virtual 1"                1     "Virtual"              "CAN, CAN FD"    "0"
 "Kvaser"        "Virtual 1"                2     "Virtual"              "CAN, CAN FD"    "0"
 "NI"            "9862 CAN/HS (CAN1)"       1     "9862"                 "CAN, CAN FD"    "17F5094"
 "NI"            "9862 CAN/HS (CAN2)"       1     "9862"                 "CAN, CAN FD"    "17F50B2"
```

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: [0×0 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: [0×0 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: [0×0 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 information 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: 23-May-2019 15:43:40
           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 Running property value is now 1 (true).

Create a Message

After you set all the property values as desired and your channels are running, 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 =

     1×8 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: 23-May-2019 15:43:40
           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 Information 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 in the `MessagesAvailable` property 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: 23-May-2019 15:43:40
           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 information indicates 1 for `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: 0.0312
            Data: [25 0 0 0 0 0 0 0]
         Signals: []
          Length: 8

   Protocol Flags
           Error: 0
```

```
          Remote: 0

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

Note the received 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: 23-May-2019 15:43:40
           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 information indicates 1 for `MessagesReceived`, and 0 for `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` which 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 1-19
- “Clean Up the MATLAB Workspace” on page 1-20

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 1-21
- “Multiplex Signals” on page 1-22
- “Configure Silent Mode” on page 1-25

More About

- “MathWorks Virtual Channels” on page 1-6

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.

See Also

Related Examples

- “Transmit and Receive CAN Messages” on page 1-10
- “Load .dbc Files and Create Messages” on page 4-2
- “View Message Information in a CAN Database” on page 4-5
- “Attach a CAN Database to Existing Messages” on page 4-8

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 1-10

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.

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

Functions
`canChannel`

Properties
can.Channel Properties

Related Examples

- "Transmit and Receive CAN Messages" on page 1-10

2

Hardware Support Package Installation

Install Hardware Support Package for Device Driver

In this section...

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

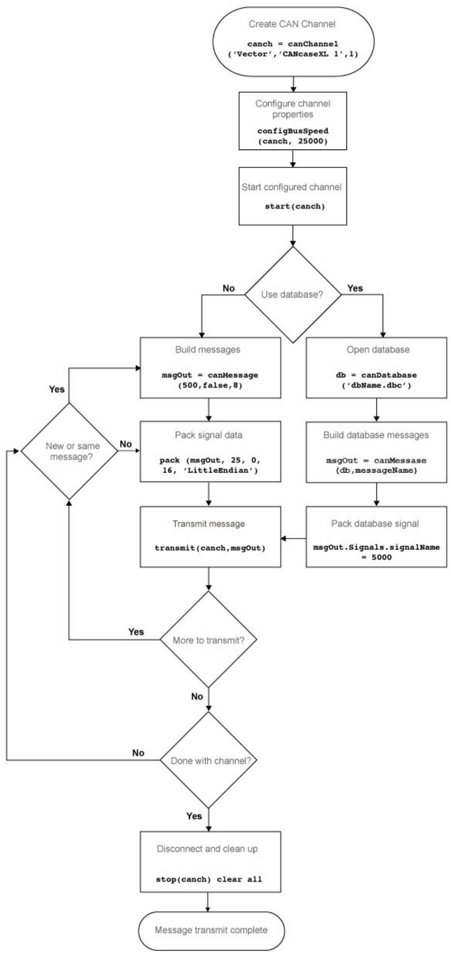
- “Get and Manage Add-Ons”

- “Vendor Limitations”

CAN Communication Workflows

CAN Transmit Workflow

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



See Also

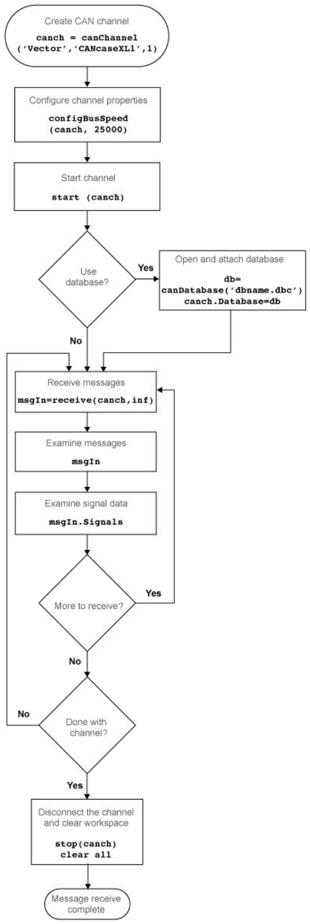
Functions
canChannel | configBusSpeed | start | canMessage | canDatabase | pack | transmit | stop | canMessageImport | transmitConfiguration | transmitEvent | transmitPeriodic

Properties
can.Channel Properties | can.Message Properties

Blocks
CAN Pack | CAN Transmit | CAN Replay

CAN Receive Workflow

Use this workflow to receive and unpack CAN messages.



See Also

Functions

`receive` | `configBusSpeed` | `attachDatabase` | `canDatabase` | `stop` | `unpack` | `extractAll` | `extractRecent` | `extractTime`

Properties

can.Channel Properties | can.Message Properties

Blocks

CAN Receive | CAN Unpack | CAN Log

Using a CAN Database

Load .dbc Files and Create Messages

In this section...

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')
```

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.

See Also

Functions
`canDatabase`

Properties
can.Database Properties

Related Examples

- “Use DBC-Files in CAN Communication” on page 14-21

More About

- “View Message Information in a CAN Database” on page 4-5
- “View Signal Information in a CAN Message” on page 4-7
- “Attach a CAN Database to Existing Messages” on page 4-8

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

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 4-5
- “Attach a CAN Database to Existing Messages” on page 4-8

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 4-5
- “View Signal Information in a CAN Message” on page 4-7

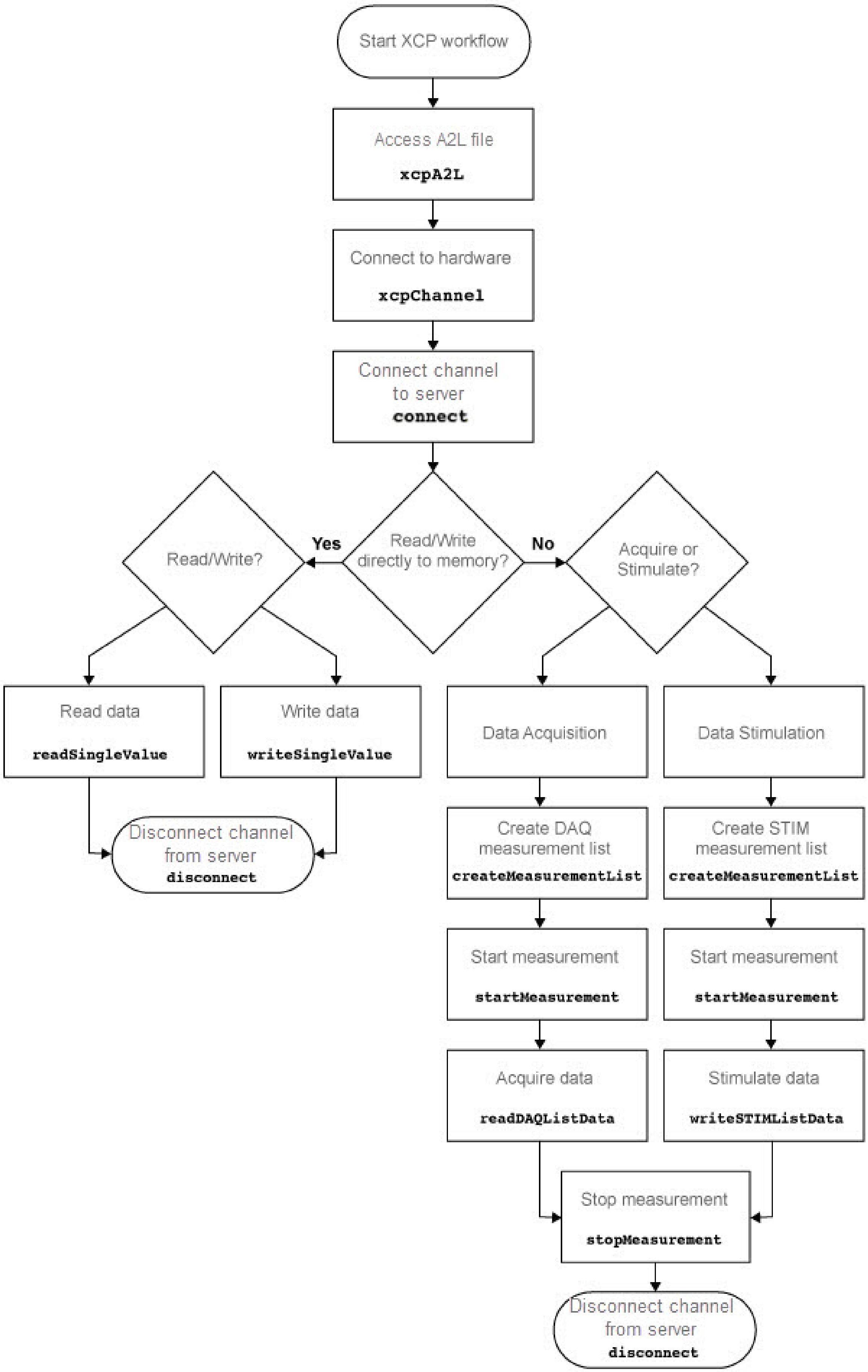
5

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


Start XCP workflow
Access A2L file
xcpA2L
Connect to hardware
xcpChannel
Connect channel to server
connect
Read/Write directly to memory?
Yes
No
Read/Write?
Acquire or Stimulate?
Read data
readSingleValue
Write data
writeSingleValue
Disconnect channel from server
disconnect
Data Acquisition
Data Stimulation
Create DAQ measurement list
createMeasurementList
Create STIM measurement list
createMeasurementList
Start measurement
startMeasurement
Start measurement
startMeasurement
Acquire data
readDAQListData
Stimulate data
writeSTIMListData
Stop measurement
stopMeasurement
Disconnect channel from server
disconnect

See Also

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

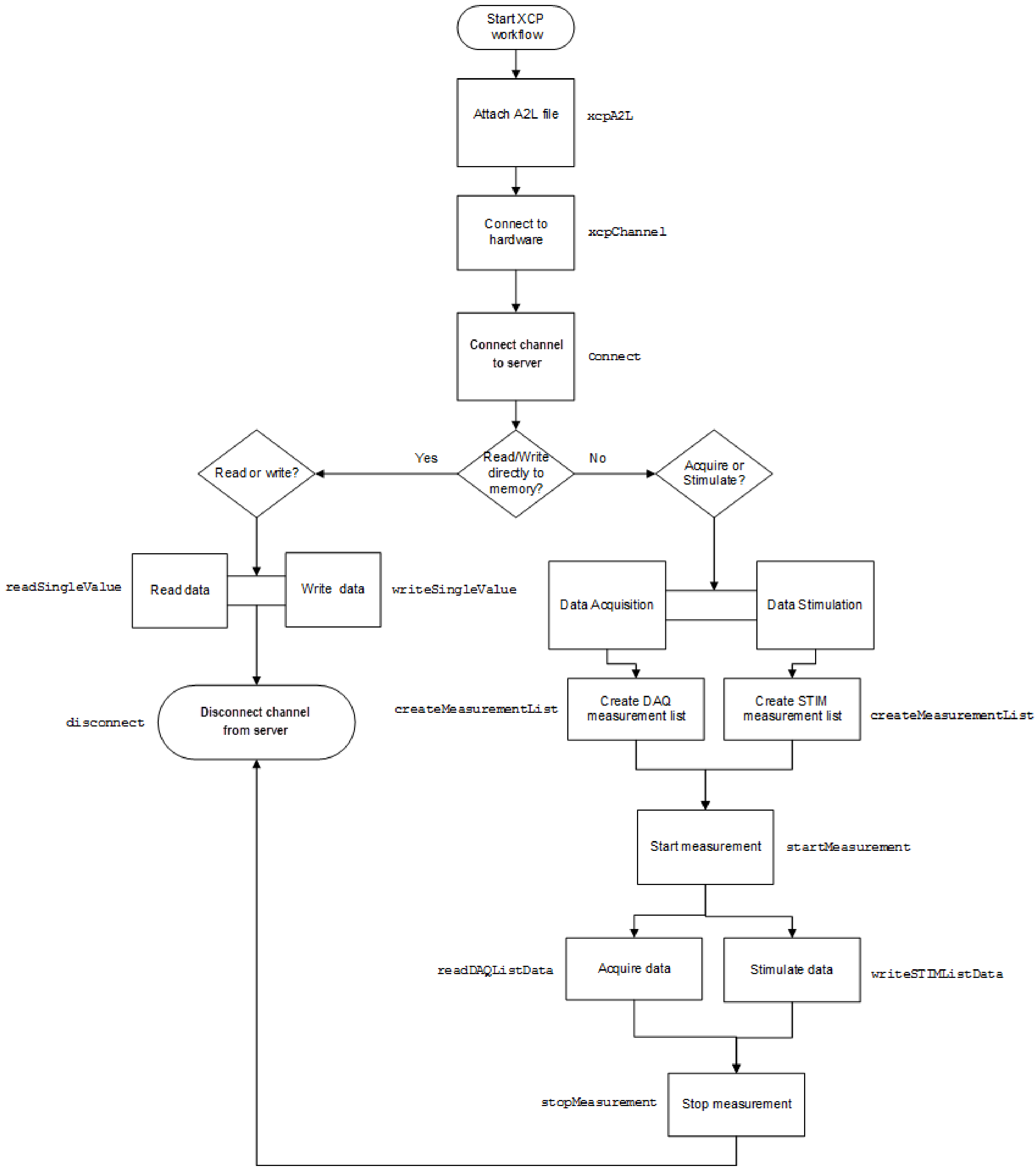
Properties
xcp.A2L Properties | xcp.Channel Properties

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

Universal Measurement & Calibration Protocol (XCP)

XCP Hardware Connection

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



Start XCP workflow
Attach A2L file
xcpA2L
Connect to hardware
xcpChannel
Connect channel to server
connect
Read/Write directly to memory?
Yes
No
Read or write?
Acquire or Stimulate?
readSingleValue
Read data
Write data
writeSingleValue
Data Acquisition
Data Stimulation
disconnect
Disconnect channel from server
createMeasurementList
Create DAQ measurement list
Create STIM measurement list
createMeasurementList
Start measurement
startMeasurement
readDAQListData
Acquire data
Stimulate data
writeSTIMListData
stopMeasurement
Stop measurement

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 server using seed key security.

Access an A2L file that describes the server module.

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

```
a2lfile = 

  A2L with properties:

   File Details
                 FileName: 'XCPServerSineWaveGenerator.a2l'
                 FilePath: 'C:\work\XCPServerSineWaveGenerator.a2l'
               ServerName: 'ModuleName'
                 Warnings: [0×0 string]

   Parameter Details
                   Events: {'100 ms'}
                EventInfo: [1×1 xcp.a2l.Event]
             Measurements: {1×6 cell}
          MeasurementInfo: [6×1 containers.Map]
          Characteristics: {'Gain'  'ydata'}
       CharacteristicInfo: [2×1 containers.Map]
                 AxisInfo: [1×1 containers.Map]
            RecordLayouts: [4×1 containers.Map]
             CompuMethods: [3×1 containers.Map]
                CompuTabs: [0×1 containers.Map]
               CompuVTabs: [0×1 containers.Map]

   XCP Protocol Details
        ProtocolLayerInfo: [1×1 xcp.a2l.ProtocolLayer]
                  DAQInfo: [1×1 xcp.a2l.DAQ]
    TransportLayerCANInfo: [0×0 xcp.a2l.XCPonCAN]
    TransportLayerUDPInfo: [0×0 xcp.a2l.XCPonIP]
    TransportLayerTCPInfo: [1×1 xcp.a2l.XCPonIP]
```

Create an XCP channel using MathWorks virtual CAN channel 1.

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

```
xcpch = 

  Channel with properties:

              ServerName: 'ModuleName'
             A2LFileName: 'XCPServerSineWaveGenerator.a2l'
          TransportLayer: 'CAN'
    TransportLayerDevice: [1×1 struct]
              SeedKeyDLL: []
```

Configure the Channel to Unlock the Server

This example shows how to configure the channel to unlock the server 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:
```

```
           ServerName: 'ModuleName'
          A2LFileName: 'XCPServerSineWaveGenerator.a2l'
       TransportLayer: 'CAN'
 TransportLayerDevice: [1×1 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 server.

```
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 server.

```
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.

```
writeCharacteric(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
readCharacteristic | writeCharacteristic | readMeasurement | writeMeasurement | readAxis | writeAxis

Acquire Measurement Data via Dynamic DAQ Lists

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

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

```
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 can 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 server.

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

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 7-3
- "J1939 Network Management" on page 7-4
- "J1939 Transport Protocols" on page 7-5
- "J1939 Channel Workflow" on page 7-6

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:

Parameter	**Priority**	**Reserved**	**Data Page**	**PDU Format**	**PDU Specific**	**Source Address**
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 7-2
- “J1939 Network Management” on page 7-4
- “J1939 Transport Protocols” on page 7-5
- “J1939 Channel Workflow” on page 7-6

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 7-2
- "J1939 Parameter Group Format" on page 7-3
- "J1939 Transport Protocols" on page 7-5
- "J1939 Channel Workflow" on page 7-6

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.

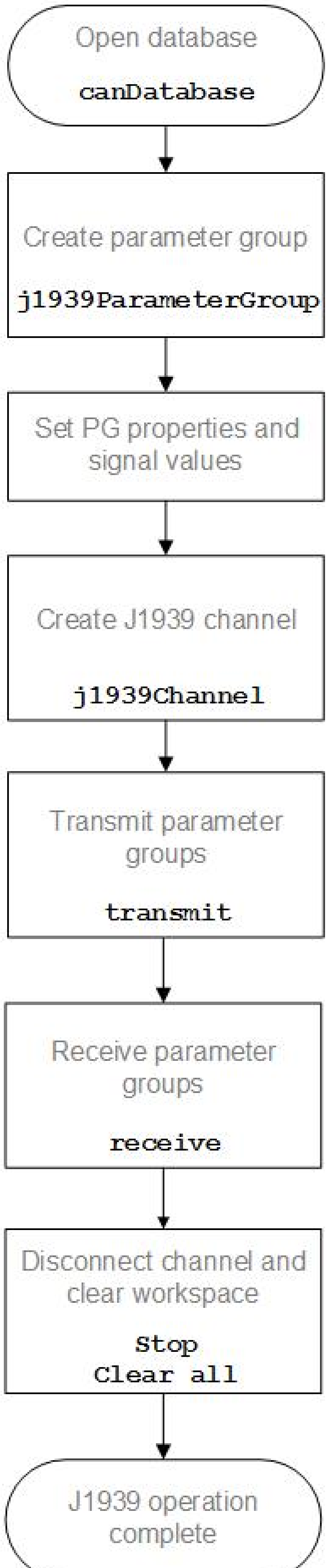
See Also

More About

- "J1939 Interface" on page 7-2
- "J1939 Parameter Group Format" on page 7-3
- "J1939 Network Management" on page 7-4
- "J1939 Transport Protocols" on page 7-5
- "J1939 Channel Workflow" on page 7-6

J1939 Channel Workflow

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


Open database
canDatabase
Create parameter group
j1939ParameterGroup
Set PG properties and signal values
Create J1939 channel
j1939Channel
Transmit parameter groups
transmit
Receive parameter groups
receive
Disconnect channel and clear workspace
Stop
Clear all
J1939 operation complete

See Also

More About

- "J1939 Interface" on page 7-2
- "J1939 Parameter Group Format" on page 7-3
- "J1939 Network Management" on page 7-4
- "J1939 Transport Protocols" on page 7-5

CAN Communications in Simulink

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.

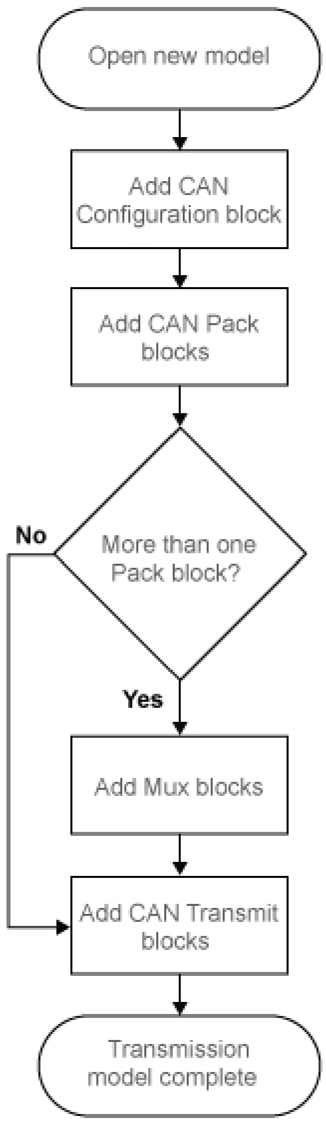
To use the Vehicle Network Toolbox block library, you require Simulink, a tool for simulating dynamic systems. Simulink is a model definition environment. Use Simulink blocks to create a block diagram that represents the computations of your system or application. Simulink is also a model simulation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

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

CAN Communication Workflows in Simulink

Message Transmission Workflow

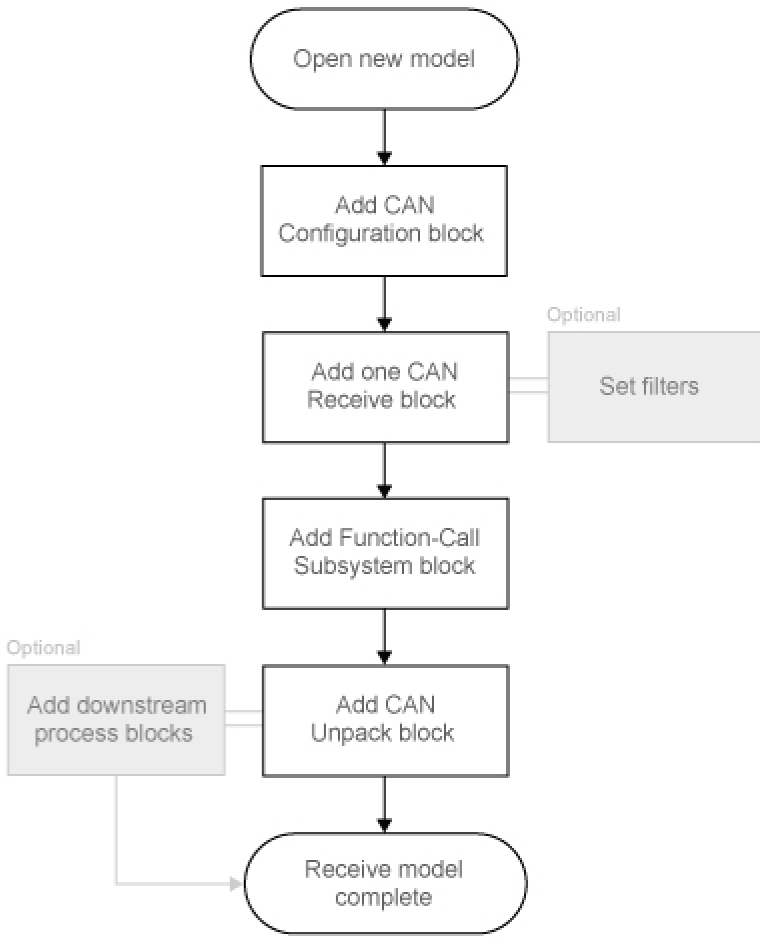
This workflow represents the most common CAN Transmit model. Adjust your model as needed. For more workflow examples, see "Build CAN Communication Simulink Models" on page 8-7 and the "Simulink Tutorials" in the Vehicle Network Toolbox examples.



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

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 toolstrip **Simulation** tab, click **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

Build the Message Transmit Part of the Model

This section shows how to build the part of the model to transmit CAN messages, using Vehicle Network Toolbox blocks with other blocks in the Simulink library.

Building a model to transmit CAN messages is detailed in the following steps:

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

For this portion of the example

- Use a MathWorks virtual CAN channel to transmit messages.
- Use the CAN Configuration block to configure your CAN channel.
- Use the Constant block to provide 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 the Message Receive Part of the Model" on page 8-9 and "Save and Run the Model" on page 8-13 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 toolstrip **Simulation** tab, click **Save > Save As** to assign a name to your new model.

Step 2: Open the Block Library

1. In the model Editor toolstrip **Simulation** tab, click **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 place a block into your model, click a block in the library and drag it into the editor. For this example, you need in your model one instance each of the following blocks:

- CAN Configuration

- CAN Pack
- CAN Transmit

Note The default configuration of each block in your model uses MathWorks Virtual 1 Channel 1. You can configure the blocks in your model to use virtual channels or hardware devices from other vendors.

Note By default, block names are not shown in the model. To display the block names while working in the model Editor, in the toolstrip **Format** tab click **Auto** and clear the **Hide Automatic Block Names** selection.

Step 4: Drag Other Blocks to Complete the Model

This example uses a Constant block as a source of data. From the Simulink > Commonly Used Blocks library, add a Constant block to your model.

Step 5: Connect the Blocks

Make a connection between the Constant block and the CAN Pack block input. 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 its CAN channel for communication.

Step 6: Specify the Block Parameter Values

You set parameters for each block 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. Verify or set the following parameters:

- **Device** to `MathWorks 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. Verify or set the following parameters:

- **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. Verify or set the following parameters:

- **Device** to MathWorks Virtual 1 (Channel 1).

  Click **OK**.

Configure the Constant Block

Double-click the Constant block to open its parameters dialog box.

On the **Main** tab, set:

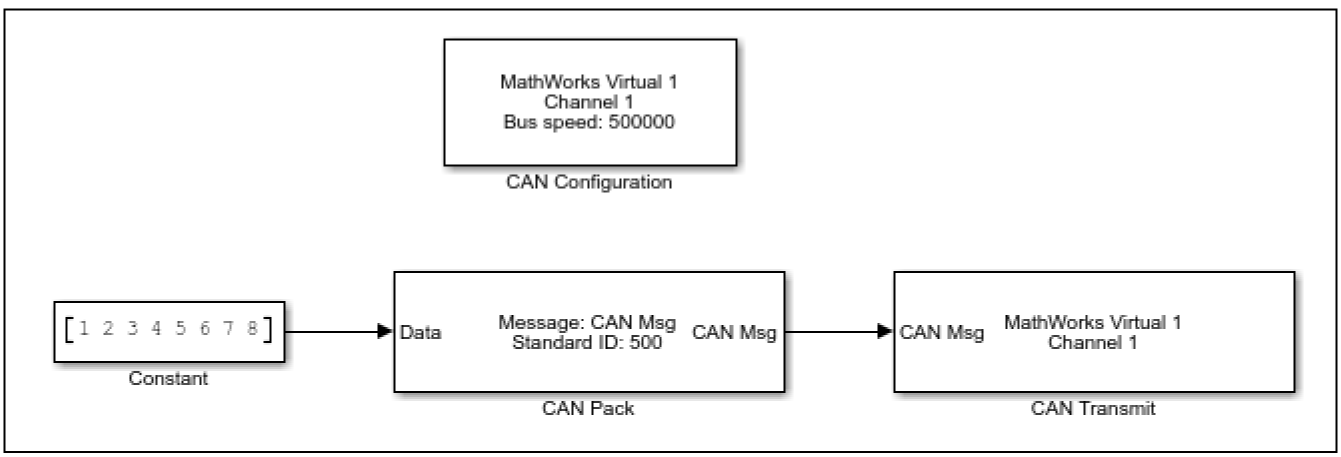
- **Constant value** to [1 2 3 4 5 6 7 8].
- **Sample time** to 0.01 seconds.

On the **Signal Attributes** tab, set:

- **Output data type** to uint8.

  Click **OK**.

Your model looks like this figure.



Build the Message Receive Part of the Model

This section shows how to build the part of the model to receive CAN messages, 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 8-10

- “Step 8: Drag Other Blocks to Complete the Model” on page 8-10
- “Step 9: Connect the Blocks” on page 8-11
- “Step 10: Specify the Block Parameter Values” on page 8-12

For this portion of the example

- Use a MathWorks virtual CAN channel to receive messages.
- Use a CAN Configuration block to configure your virtual CAN channel.
- Use a CAN Receive block to receive the message.
- 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 display the received data.

Step 7: Drag Vehicle Network Toolbox Blocks into the Model

For this part of the example, start with one instance each of the following blocks from the Vehicle Network Toolbox CAN Communication block library:

- CAN Configuration
- CAN Receive

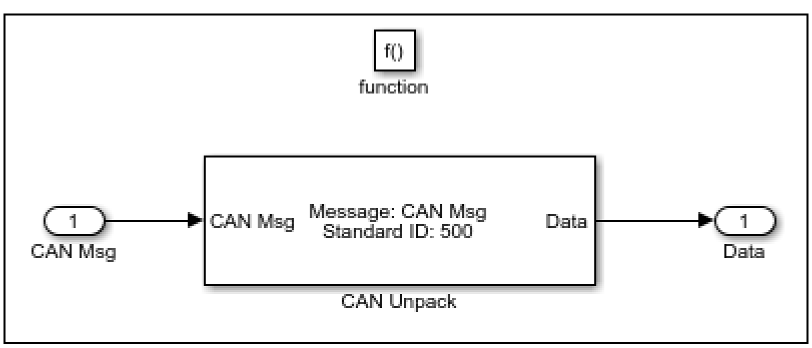
Tip Configure separate CAN channels for the CAN Receive and CAN Transmit blocks. Each channel needs its own CAN Configuration block.

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. Double-click the **In1** port label to rename it to **CAN Msg**.
4. Double-click the **Out1** port label to rename it to **Data**.
5. Drag and drop the CAN Unpack block from the Vehicle Network Toolbox block library into this subsystem. If placed between the input and output lines, they will automatically connect.

   The inside of your Function-Call Subsystem block should now look like this figure.



The reason to place the CAN Unpack inside a Function-Call Subsystem is so that it can capture all possible messages.

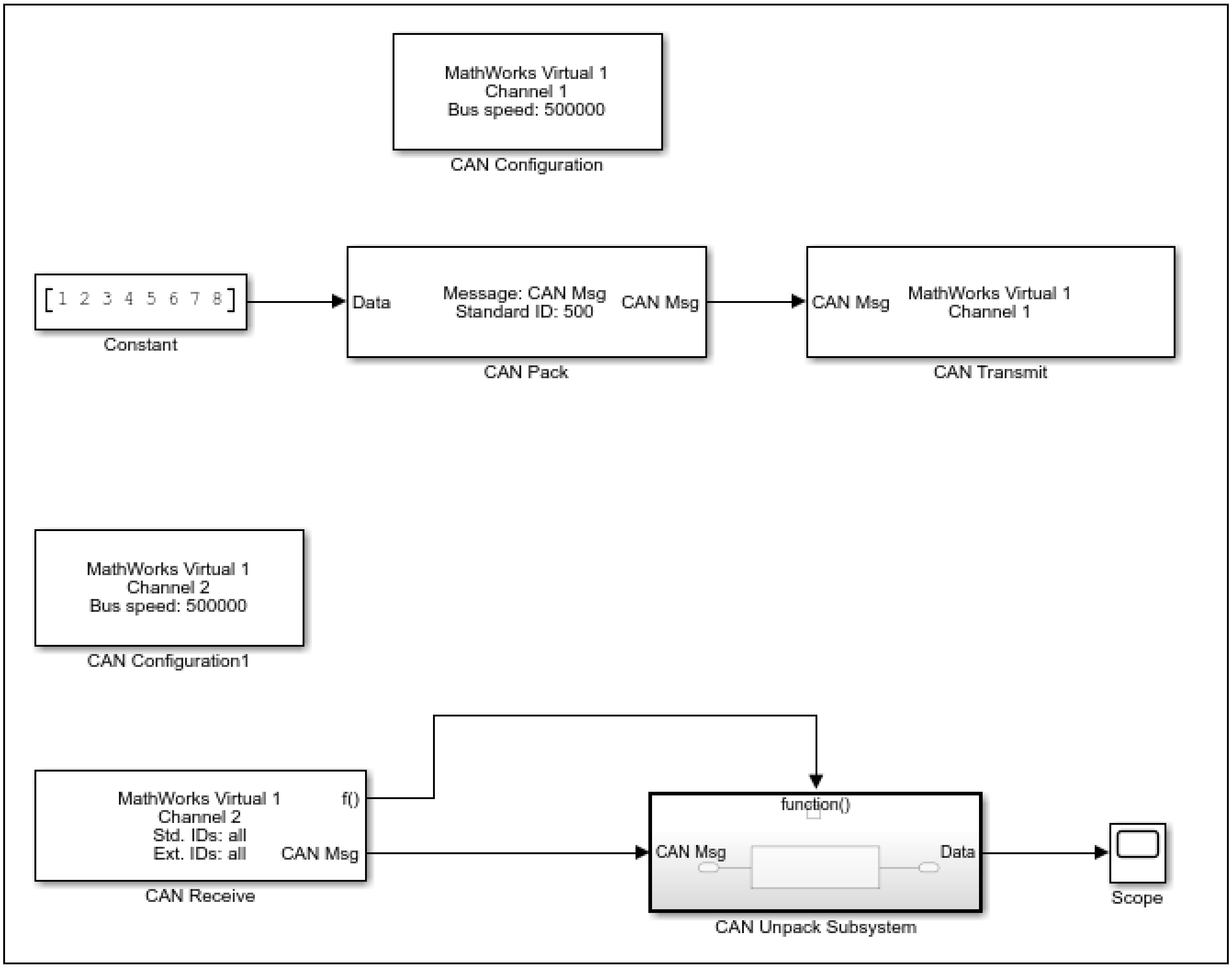
6. Click the back-arrow in the toolstrip to return to your model view.

Step 9: Connect the Blocks

1. Rename the Function-Call Subsystem block to **CAN Unpack Subsystem**.
2. Connect the **CAN Msg** output port of the CAN Receive block to the **In1** input port of the **CAN Unpack Subsystem** block.
3. Connect the **f()** output port of the CAN Receive block to the **function()** input port of the **CAN Unpack Subsystem** block.
4. For a visual display of the simulation results, drag the Scope block from the Simulink block library into your model.
5. Connect the **CAN Msg** output port of your **CAN Unpack Subsystem** block to the input port of the Scope block.

   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.

Your model looks like this figure.



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 `MathWorks 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 `MathWorks 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**.

Save and Run the Model

This section shows you how to save the model you built, "Build the Message Transmit Part of the Model" on page 8-7 and "Build the Message Receive Part of the Model" on page 8-9.

- "Step 11: Save the Model" on page 8-13
- "Step 12: Change Configuration Parameters" on page 8-13
- "Step 13: Run the Simulation" on page 8-13
- "Step 14: View the Results" on page 8-14

Step 11: Save the Model

Before you run the simulation, save your model by clicking the **Save** icon or selecting **Save** from the Editor toolstrip **Simulation** tab.

Step 12: Change Configuration Parameters

1. In your model Editor toolstrip **Modeling** tab, click **Model Settings**. 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

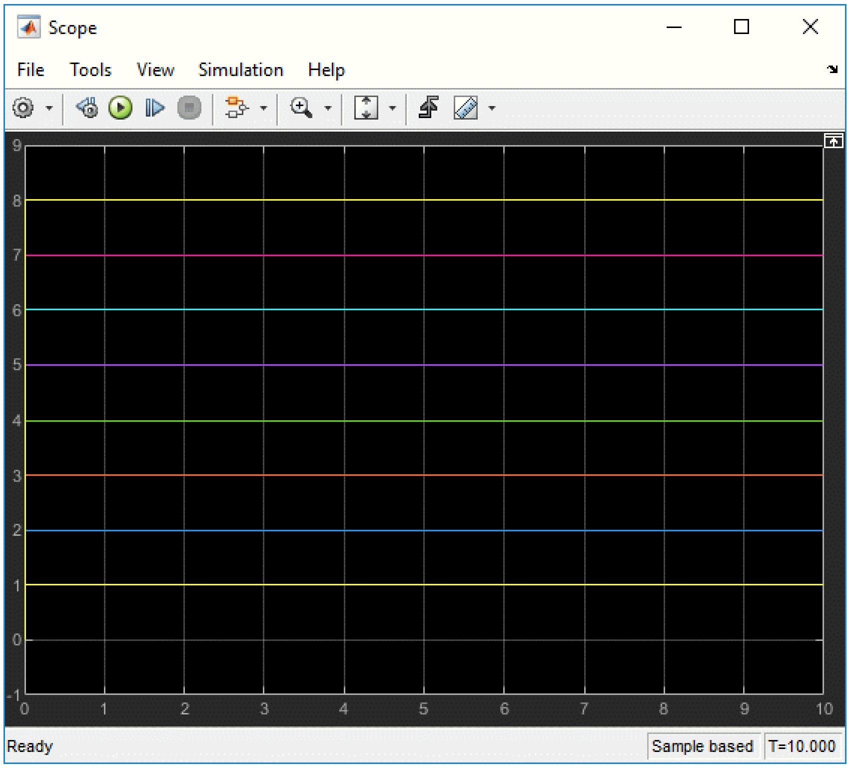
To run the simulation, click the **Run** button in the **Simulation** or **Modeling** tab of the Editor toolstrip.

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 Subsystem 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 the 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. You configured the model to pack and transmit an array of constant values, `[1 2 3 4 5 6 7 8]`, every 0.01 seconds of simulation time. 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 Interactively" (Simulink)

Create Custom CAN Blocks

In this section...

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 8-15 (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 8-16 — 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 “Use a Bus Signal with S-Function Builder to Create an S-Function” (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.

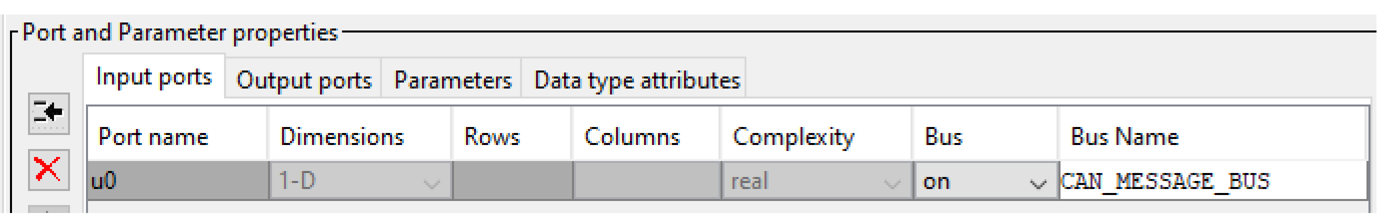
   ```
   canMessageBusType
   ```

   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`.

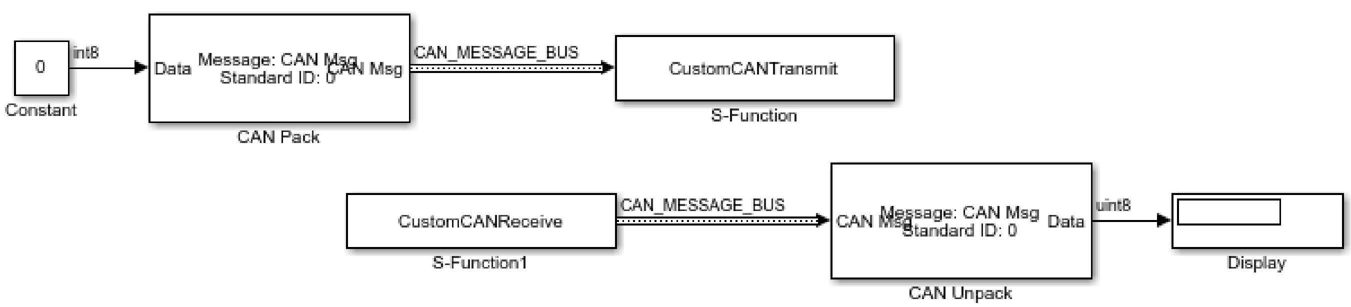
In your function and block building, use the other tabs in the dialog box to define the code for interaction with your device driver, and remove unnecessary ports.

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 or 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 8-15.

To create your own blocks for use with other Vehicle Network Toolbox blocks, 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 type to `CAN_MESSAGE`:

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

See Also

Functions
`canMessageBusType` | `canFDMessageBusType`

More About

- "C/C++ S-Function Basics" (Simulink)
- "Use a Bus Signal with S-Function Builder to Create an S-Function" (Simulink)

Supported Block Features

The blocks of the Vehicle Network Toolbox block library support the following features.

CAN Communication

Block	Platforms	Simulink Accelerator™ and Rapid Accelerator	Code Generation	Additional Supporting Products	Simulink Bus Objects
CAN Configuration	Windows, Linux	Yes	For host computer only		Not applicable
CAN Receive	Windows, Linux	Yes	For host computer only		Recommended
CAN Transmit	Windows, Linux	Yes	For host computer only		Recommended
CAN Pack	Windows, Linux	Yes	Portable for signal information up to 32-bit length	Simulink Real-Time™, Embedded Coder®	Recommended
CAN Unpack	Windows, Linux	Yes	Portable for signal information up to 32-bit length	Simulink Real-Time, Embedded Coder	Recommended
CAN Replay	Windows, Linux	Yes	For host computer only		Recommended
CAN Log	Windows, Linux	Yes	For host computer only		Recommended

CAN FD Communication

Block	Platforms	Simulink Accelerator and Rapid Accelerator	Code Generation	Additional Supporting Products	Simulink Bus Object
CAN FD Configuration	Windows, Linux	Yes	For host computer only		Not applicable
CAN FD Receive	Windows, Linux	Yes	For host computer only		Required
CAN FD Transmit	Windows, Linux	Yes	For host computer only		Required
CAN FD Pack	Windows, Linux	Yes	Portable for signal information up to 32-bit length	Simulink Real-Time	Required

Block	Platforms	Simulink Accelerator and Rapid Accelerator	Code Generation	Additional Supporting Products	Simulink Bus Object
CAN FD Unpack	Windows, Linux	Yes	Portable for signal information up to 32-bit length	Simulink Real-Time	Required
CAN FD Replay	Windows, Linux	Yes	For host computer only		Required
CAN FD Log	Windows, Linux	Yes	For host computer only		Required

XCP Communication

Block	Platforms	Simulink Accelerator and Rapid Accelerator	Code Generation	Additional Supporting Products	Simulink Bus Object
XCP CAN Configuration	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable
XCP CAN Transport Layer	Windows	Yes	For host computer only		Not applicable
XCP CAN Data Acquisition	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable
XCP CAN Data Stimulation	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable
XCP UDP Configuration	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable
XCP UDP Data Acquisition	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable
XCP UDP Data Stimulation	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable
XCP UDP Bypass	Windows	Yes	For host computer only	Simulink Real-Time	Not applicable

J1939 Communication

Block	Platforms	Simulink Accelerator and Rapid Accelerator	Code Generation	Additional Supporting Products	Simulink Bus Object
J1939 Network Configuration	Windows	Yes	For Windows host computer only	Simulink Real-Time	Not applicable

Block	Platforms	Simulink Accelerator and Rapid Accelerator	Code Generation	Additional Supporting Products	Simulink Bus Object
J1939 Node Configuration	Windows	Yes	For Windows host computer only	Simulink Real-Time	Not applicable
J1939 CAN Transport Layer	Windows	Yes	For Windows host computer only	Simulink Real-Time	Not applicable
J1939 Transmit	Windows	Yes	For Windows host computer only	Simulink Real-Time	Not applicable
J1939 Receive	Windows	Yes	For Windows host computer only	Simulink Real-Time	Not applicable

See Also

More About

- “Platform Support” on page 9-6
- “Communication Protocols” (Simulink Real-Time)
- “Blocks for Embedded Targets” (Embedded Coder)

Timing in Hardware Interface Models

Simulation Time

When blocks in your Simulink model must interface with hardware devices, you might have to consider how long the simulation takes to run in real time versus simulation time, and how often and how many times the hardware interface blocks execute during a simulation. Usually your hardware communication rates are relative to real-world or "wall clock" time. You can adjust the duration of a simulation, the execution rate of the blocks, and the pacing of the model to accommodate your hardware requirements. This topic discusses basic timing concepts in hardware interface models, using fixed steps for block execution.

A model simulation has a duration defined by a start time and a stop time. The default duration is 10 units of simulation time (or simulated seconds). These simulation seconds are not necessarily equivalent to a real-time second as measured by a wall clock.

To adjust the model duration, open the model Configuration Parameters by clicking the **Model Settings** icon in the Modeling tab of the model editor toolstrip. Select **Solver** in the left pane. The **Start time** and **Stop time** settings define the duration. In most cases, **Start time** should be `0.0`, and you can set **Stop time** to reflect the duration you want the model to have.

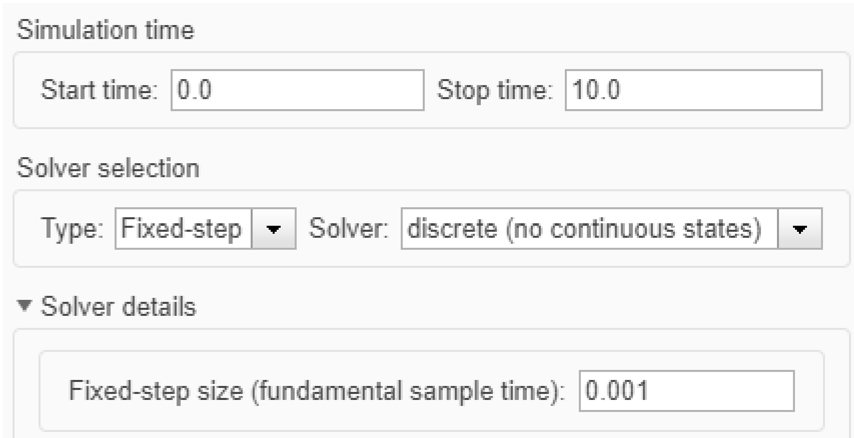
As a simulation runs, the clocking for block execution is performed by a series of timesteps. With a setting for an automatic solver with fixed timestep sizes, during compilation Simulink calculates the timestep frequency to accommodate the **Sample time** parameter settings of all the blocks in the model. For example, if all the timed blocks in the model have a Sample time setting of 0.01 or a multiple of that, then a timestep size of 0.01 works for the whole model.

Block Sample Time

For models that interface with hardware devices, you might prefer fixed timesteps of a specified rate. For example, you might need millisecond resolution to control the timing relationship of your blocks. Set the timing options as follows:

- **Start time**: `0.0`
- **Stop time**: `10.0`
- **Type**: `Fixed-step`
- **Solver**: `discrete`
- **Fixed-step size**: `0.001`

The dialog settings look like this figure:



In this model, a block with a default **Sample time** setting of `0.01` executes every tenth timestep, or 1001 times in a 10 second simulation. Another block that must run at twice the rate should have **Sample time** set to `0.005`.

Note In most cases, you can leave the **Fixed-step size** setting to `auto`, allowing Simulink to calculate the appropriate fundamental sample time based on all the block settings.

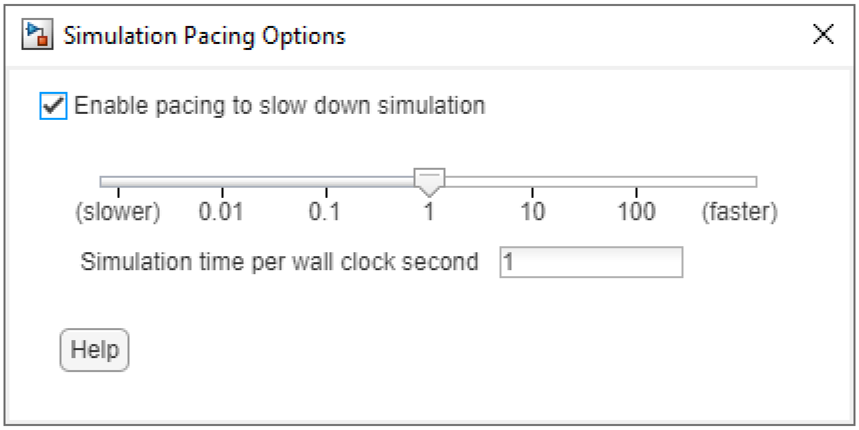
Because the simulation duration is 10 simulated seconds, and the **Sample time** period of the block is 0.01 simulated seconds, that block executes 1001 times in a complete simulation (including first and last step). The simulation runs as fast as its blocks can perform, and those 1001 executions might take significantly less than 10 seconds of wall clock time. So the simulation in real time is determined by how fast it can execute the blocks in the model for the required number of iterations. Often the purpose of simulation is to model behavior in a way that takes less time than it would in a real-world situation. In these cases, the sequence and repetition of block execution is important, while the actual span of real-world time might not be.

Pacing Model Simulation

You might have a requirement for a model to interact with a hardware device by repeating some operation at fixed intervals of real-world time. For example, a block might repeatedly read data from a thermometer or send triggers for an external signal generator to output a pulse train.

If you set the block **Sample time** to `0.1`, that would control the rate of block execution only in simulation time. To correlate simulation time to real time, you can use Simulation Pacing to slow down a simulation to run at the pace of real-world time. Access the Simulation Pacing Options dialog by clicking **Run** > **Simulation Pacing** in the Simulation tab of the model editor toolstrip

Check **Enable pacing to slow down simulation**, and set the slider ratio to `1` (the default). This causes simulation time to track as closely as possible with wall clock time, so 1 simulation second is approximately equal to 1 wall clock second.



With this pacing setting, a block **Sample time** of `0.1` is approximately equal to 0.1 wall clock seconds, resulting in ten block executions per second. So a block that generates a device output pulse every 0.1 simulation seconds, now puts out 10 pulses per wall clock second.

See Also

More About

- "What Is Sample Time?" (Simulink)
- "Simulation Pacing" (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

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.

National Instruments Hardware Limitations

Limited number of connections to an NI-XNET channel

When using NI-XNET for CAN or CAN FD communication, there is a limit to the total number of connections to the channel from MATLAB or Simulink.

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.

File Format Limitations

MDF-File

The following restrictions apply to MDF-file operations.

- The `mdfSort` function is not supported on Linux systems.
- `mdfVisualize` supports only integer and floating point data types in MDF-file channels.
- The following MDF-file functions do not support the full range of international characters that are supported by the other MDF functions:
  - `mdfSort`
  - `mdfVisualize`

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.
- DTD-based headers are not supported. 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">
```

BLF-File

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

See Also

Functions
`blfread` | `blfwrite` | `cdfx` | `mdf` | `mdfDatastore`

More About

- “Platform Support” on page 9-6
- “Troubleshooting MDF Applications” on page 9-7

Platform Support

The following tables indicate which toolbox features are available for each operating system platform.

Vendor	Windows	Linux
MathWorks virtual channels	✓	✓
Vector	✓	
PEAK-System	✓	✓
Kvaser	✓	✓
National Instruments	✓	

File Format	Windows	Linux
BLF	✓	✓
CDF	✓	✓
MDF	✓	✓

See Also

More About

- “Vendor Limitations”
- “File Format Limitations” on page 9-5
- “Supported Block Features” on page 8-18

Troubleshooting MDF Applications

Error When Creating mdf Object

Issue

You might see an error when you try to create an object for access to the MDF-file with the `mdf` function.

Possible Solutions

- A likely cause is an MDF-file that is improperly formatted or that includes unsupported elements. For checking an MDF-file, Vector provides an MDF Validator tool, which you can download from Tool Support MDF.

Error When Reading an MDF-File

Issue

You might see an error when you try to read data from an MDF-file with the `read` function.

Possible Solutions

- A possible cause is an MDF-file that is improperly formatted or that includes unsupported elements. For checking an MDF-file, Vector provides an MDF Validator tool, which you can download from Tool Support MDF.
- Another possible cause is an unsorted MDF-file. Beginning in R2019b, accessing an unsorted MDF-file generates a recognizable error, and you can sort the file using the `mdfSort` function.
- When unable to read the entire file, you can read data one channel at a time. Use the `read` function with the form `data = read(mdfObj,chanGroupIndex,chanName)`

Error When Reading an MDFDatastore

Issue

You might see an error when you try to read data from an MDFDatastore with the `read` function.

Possible Solutions

- Those channels targeted for reading must have the same name and belong to the same channel group in each file of the MDF datastore. Assure uniformity across the MDF-files in the database for the channels you are reading.

Unable to Find Specific Channel

Issue

You might be unable to find and read a channel of interest in the MDF-file.

Possible Solutions

- To identify channels in the MDF-file, use the `channelList` function.

Unable to Save MDF Attachments

Issue

The `saveAttachment` function fails to save a file attached to the MDF-file.

Possible Solutions

- The `saveAttachment` function works only with embedded attachments; external files are not saved because they are already on disk.
- If the attachment does not exist, check with the provider of the MDF-file.

Unable to Read Array Channel Structures

Issue

Vehicle Network Toolbox does not support array channel structures.

Possible Solutions

- To read these channels, you must write a composition function to repackage the data.

Unable to Read MIME and CANopen Data

Issue

Reading MDF-file channels with MIME or CANopen data generates an error.

Possible Solutions

- MIME and CANopen data are not supported by Vehicle Network Toolbox.

Table Column Names Do Not Match Channel Names

Issue

When reading an MDF-file, the column names of the output timetable correspond to the channel names in the file, but they might not be identical. Table column names must be compliant with MATLAB variable names, so they are altered to limit their size and characters. Most unsupported characters are converted to underscores.

Possible Solutions

- The returned timetable preserves the ordering of the channels. So you can access data in the table with numerical indexing.
- The original names of the channels are embedded in the timetable properties. For example:

```
m = mdf('File01.mf4');
tt = read(m);
```

```
t1 = tt{1};
t1.Properties.VariableDescriptions

ans =

  1×2 cell array

    {'Signed_Int16_LE_Offset_32'}    {'Unsigned_UInt32_LE_Primary_Offset_0'}
```

See Also

Functions
`mdf` | `mdfSort` | `channelList`

More About

- "Standard File Formats"
- "File Format Limitations" on page 9-5

External Websites

- Tool Support MDF

10

XCP Communications in Simulink

Vehicle Network Toolbox XCP Simulink Blocks

Vehicle Network Toolbox provides two sets of XCP block libraries, which provide blocks for handling XCP message traffic on a CAN network or by UDP. The CAN and UDP libraries contain the following blocks:

CAN:

- **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.

UDP:

- **XCP UPD Configuration** — Configure XCP settings for UDP.
- **XCP UDP Data Acquisition** — Acquire XCP data over UDP.
- **XCP UDP Data Stimulation** — Stimulate XCP data over UDP.

You can use these blocks with blocks from other Simulink libraries to create sophisticated models.

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

See Also

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

More About

- “Open the Vehicle Network Toolbox XCP Block Libraries” on page 10-3

Open the Vehicle Network Toolbox XCP Block Libraries

Using the MATLAB Command Window

To open the Vehicle Network Toolbox XCP block libraries, enter `vntxcplib` in the MATLAB Command window.

The Simulink Library Browser opens in a separate window and displays two libraries for XCP blocks. Double-click either `CAN` or `UDP` for the protocol you want.

Using the Simulink Library Browser

To open the Vehicle Network Toolbox XCP block libraries using Simulink windows and menus, use the following steps.

1. Click **Simulink** in the MATLAB toolstrip **Home** tab.
2. In the Simulink Start Page hover over **Blank Model** and click **Create Model**, or open one of your existing models.
3. In the model Editor toolstrip **Simulation** tab, click **Library Browser**.
4. The left pane of the browser lists all available block libraries. Expand the `Vehicle Network Toolbox` and `XCP Communication` trees, then select either `CAN` or `UDP` for the protocol you want.

See Also

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

More About

- “Vehicle Network Toolbox XCP Simulink Blocks” on page 10-2

11

Functions

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'
    'A1DEFSI'
```

```
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:

Field	Description
`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
`nodeInfo` | `messageInfo` | `signalInfo` | `canDatabase` | `valueTableText`

Properties
can.Database Properties

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: [2×3 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.

Note Support for BLF-files is limited to only CAN and CAN FD protocols on Windows operating systems. See "File Format Limitations" on page 9-5.

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, ...
                                              "CANExtendedFilter", 3:7)
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`

Topics
"File Format Limitations" on page 9-5

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.

Note Support for BLF-files is limited to only CAN and CAN FD protocols on Windows operating systems. See “File Format Limitations” on page 9-5.

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`

Topics
"File Format Limitations" on page 9-5

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.

`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: [0×0 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: [0×0 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 can.Channel Properties.

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

Properties
can.Channel Properties

Introduced in R2009a

CAN.ChannelInfo 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
canHWInfo | can.VendorInfo

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 can.Database Properties.

See Also

Functions
canMessage

Properties
can.Database Properties

Introduced in R2009a

CAN Explorer

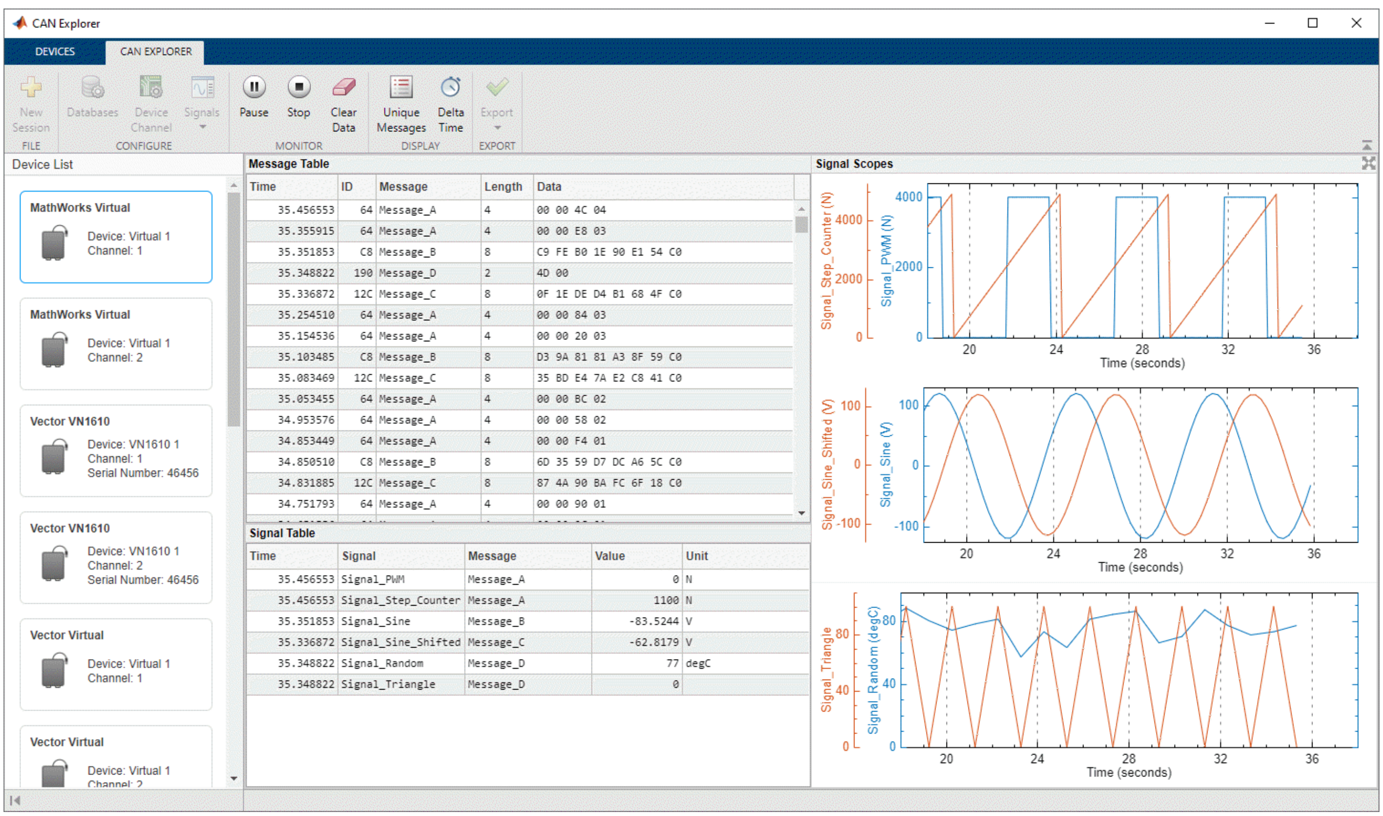
Acquire and visualize CAN data

Description

The **CAN Explorer** app allows you to acquire and visualize CAN data, filtering on specified signals and messages.

Using this app, you can:

- Configure device channels and acquisition properties.
- Apply CAN database configurations.
- Preview data.
- Export data to the MATLAB workspace
- Export the app setup to a MATLAB script.



Open the CAN Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app.
- MATLAB command prompt: Enter `canExplorer`.

Examples

- "Receive and Visualize CAN Data Using CAN Explorer" on page 14-192

Limitations

- For performance reasons, there are limitations on the number of messages saved or displayed in the app.
- Although the app configuration is saved for the next time the same user opens it, you cannot save or export the app configuration to share with other users.
- The **CAN Explorer** supports only the CAN protocol. For CAN FD protocol data, use the **CAN FD Explorer**.
- The apps does not support J1939 data.

See Also

Apps
CAN FD Explorer

Topics
"Receive and Visualize CAN Data Using CAN Explorer" on page 14-192

Introduced in R2021a

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 `devicenumber` is the interface number defined in the NI Measurement & Automation Explorer.

For PEAK-System devices `vendor` is the character vector `'PEAK-System'`, and `devicenumber` 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: [0×0 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:

CAN Device Properties:

Bit Timing Properties:

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 =

  2×6 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"
```

```
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

CAN FD Explorer

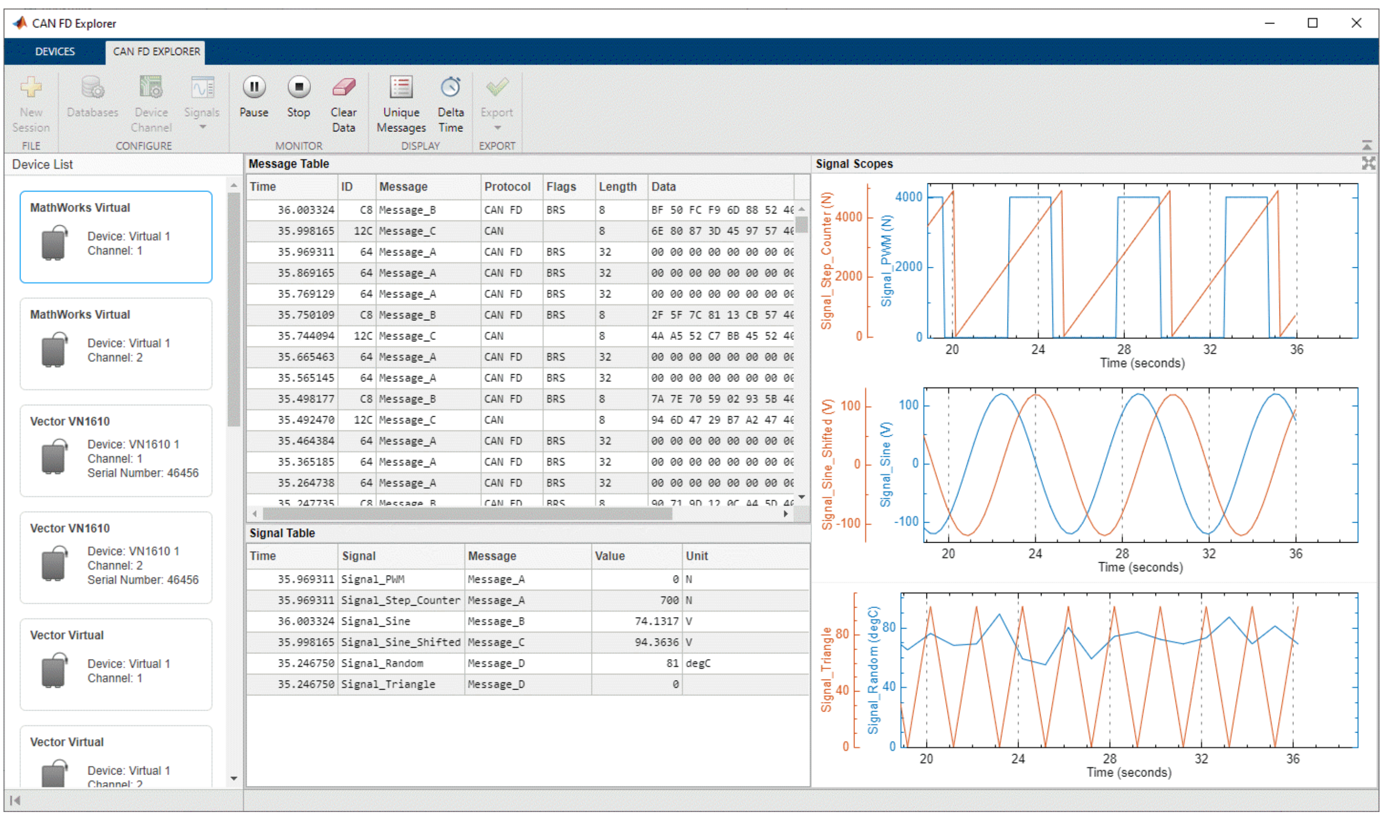
Acquire and visualize CAN FD data

Description

The **CAN FD Explorer** app allows you to acquire and visualize CAN FD data, filtering on specified signals and messages.

Using this app, you can:

- Configure device channels and acquisition properties.
- Apply CAN FD database configurations.
- Preview data.
- Export data to the MATLAB workspace
- Export the app setup to a MATLAB script.



Open the CAN FD Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app.
- MATLAB command prompt: Enter `canFDExplorer`.

Examples

- "Receive and Visualize CAN FD Data Using CAN FD Explorer" on page 14-198

Limitations

- For performance reasons, there are limitations on the number of messages saved or displayed in the app.
- Although the app configuration is saved for the next time the same user opens it, you cannot save or export the app configuration to share with other users.
- The **CAN FD Explorer** supports only the CAN FD protocol. For CAN protocol data, use the **CAN Explorer**.
- The apps does not support J1939 data.

See Also

Apps
CAN Explorer

Topics
"Receive and Visualize CAN FD Data Using CAN FD Explorer" on page 14-198

Introduced in R2021a

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: [1×1 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: [1×64 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 of 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 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: [12×1 Simulink.BusElement]
```

View the `Elements` properties of the bus.

```
CAN_FD_MESSAGE_BUS.Elements
```

```
ans = 

  12×1 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 8-15
"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    CAN_DataFrame_W
    ___________    ________________________    _____________________    _________________    ________________________    _______________

    0.30022 sec               1                     2.1095e+06                  1                       0                          0
    0.45025 sec               1                     2.0972e+06                  1                       0                          0
    0.60022 sec               1                     2.1095e+06                  1                       0                          0
    0.75013 sec               1                     2.1095e+06                  1                       0                          0
```

```
msgTimetable(1:4,1:8)
```

```
ans =

  4×8 timetable

       Time         ID     Extended    Name    ProtocolMode        Data        Length    DLC      Signals
    ___________    ____    ________    ____    ____________    ____________    ______    ___    ____________

    0.30022 sec     768     false       ''       'CAN FD'      [1×64 uint8]      64      15     [0×0 struct]
    0.45025 sec    1104     false       ''       'CAN'         [1×8  uint8]       8       8     [0×0 struct]
    0.60022 sec     768     false       ''       'CAN FD'      [1×64 uint8]      64      15     [0×0 struct]
    0.75013 sec    1872     false       ''       'CAN FD'      [1×24 uint8]      24      12     [0×0 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
`canSignalTimetable` | `canDatabase` | `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: [1×2 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
`canChannelList` | `canChannel`

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: 2500
        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: [1×1 struct]
         Length: 8

   Protocol Flags
          Error: 0
         Remote: 0

   Other Information
       Database: [1×1 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('CANdb.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 can.Message Properties.

See Also

Functions
`attachDatabase` | `canDatabase` | `extractAll` | `extractRecent` | `extractTime` | `pack` | `unpack`

Properties
can.Message Properties

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: [7×1 Simulink.BusElement]
```

View the `Elements` properties.

```
CAN_MESSAGE_BUS.Elements
```

```
ans = 

  7×1 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 8-15
"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 =

  4×6 timetable

       Time        CAN_DataFrame_DataLength    CAN_DataFrame_WakeUp    CAN_DataFrame_SingleWire    CAN_DataFrame_IDE    CAN_DataFrame_I
    ___________    ________________________    ____________________    ________________________    _________________    _______________

    0.019968 sec              4                         0                          0                       0                  100
    0.029964 sec              4                         0                          0                       0                  100
    0.039943 sec              4                         0                          0                       0                  100
    0.049949 sec              4                         0                          0                       0                  100
```

```
msgTimetable(1:4,1:6)
```

```
ans =

  4×6 timetable

       Time         ID    Extended    Name       Data        Length      Signals
    ____________    ___   ________    ____    ___________    ______    ____________

    0.019968 sec    100    false       ''     [1×4 uint8]      4       [0×0 struct]
    0.029964 sec    100    false       ''     [1×4 uint8]      4       [0×0 struct]
    0.039943 sec    100    false       ''     [1×4 uint8]      4       [0×0 struct]
    0.049949 sec    100    false       ''     [1×4 uint8]      4       [0×0 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
`canSignalTimetable` | `canDatabase` | `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
canMessageImport | canSignalTimetable | canDatabase

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

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
`canHWInfo` | `CAN.ChannelInfo`

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 9-5.

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
`instanceList` | `systemList` | `getValue` | `setValue` | `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_Primary_Offset_0"                2                     10000
    "Signed_Int16_LE_Offset_32"                   1                     10000
    "Unsigned_UInt32_LE_Primary_Offset_0"         1                     10000
```

View Specific MDF Channels

Filter on channel names.

```
chans = channelList(mdfObj,'Float')
```

```
chans =
```

```
  2×9 table

             ChannelName               ChannelGroupNumber    ChannelGroupNumSamples
    ______________________________    __________________    ______________________

    "Float_32_LE_Offset_64"                   2                     10000
    "Float_64_LE_Primary_Offset_0"            2                     10000
```

```
chans = channelList(mdfObj,'Float','ExactMatch',true)
```

```
chans =

  0×9 empty table
```

Input Arguments

mdfObj — 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,arbbusspeed,databusspeed)` 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,arbbusspeed,arbSJW,arbTSeg1,arbTSeg2,databusspeed,
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 timing on a Kvaser device.

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

Configure CAN FD on a 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 a 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 a 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`

`arbbusspeed` — 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: `5`

Data Types: `double`

`dataBRP` — Data clock prescalar for time quantum
double

Data clock prescalar for time quantum, specified as a double.

Example: 2

Data Types: `double`

See Also

Functions
`canChannel` | `canFDChannel` | `start`

Properties
can.Channel Properties

External Websites
Bit Timing

Introduced in R2009a

configBusSpeed

Package: 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 server module

Syntax

```
connect(xcpch)
```

Description

`connect(xcpch)` creates an active connection between the XCP channel and the server module, enabling active messaging between the channel and the server.

Examples

Connect to a Server 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 server module defined by the A2L file.

See Also

Functions
`xcpA2L` | `xcpChannel` | `readSingleValue` | `writeSingleValue`

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:

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

Connect the channel to the server module.

```
connect(xcpch)
```

Set up 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:

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

Connect the channel to the server module.

```
connect(xcpch)
```

Set up 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 server 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

`viewMeasurementLists` | `startMeasurement` | `freeMeasurementLists`

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

Package: 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 server module

Syntax

```
disconnect(xcpch)
```

Description

`disconnect(xcpch)` disconnects the specified XCP channel from the server module. Disconnecting the channel stops active messaging between the channel and the server module.

Examples

Disconnect an Active XCP Connection

Create an XCP channel using a CAN module, connect the channel and disconnect it from the specified server module.

Link an A2L file

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

Create an XCP channel using a Vector CAN module virtual channel. Check to see if channel is connected.

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

Connect the channel and validate its connection.

```
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 server module defined by the A2L file.

See Also

`xcpA2L` | `xcpChannel` | `connect` | `isConnected`

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

Package: `j1939`

Occurrences of specified J1939 parameter groups

Syntax

```
extractedPGs = extractAll(pgrp,pgname)
[extractedPGs,remainderPGs] = extractAll(pgrp,pgname)
```

Description

`extractedPGs = extractAll(pgrp,pgname)` returns all parameter groups whose name occurs in `pgname`.

`[extractedPGs,remainderPGs] = extractAll(pgrp,pgname)` also returns a parameter group array, `remainder`, containing all groups from the original array not matching the specified names in `pgname`.

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
`j1939ParameterGroup` | `extractRecent` | `extractTime`

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

Package: `j1939`

Occurrences of most recent J1939 parameter groups

Syntax

```
extractedPGs = extractRecent(pgrp)
extractedPGs = extractRecent(pgrp,pgname)
```

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,pgname)` returns the most recent instance of parameter groups whose names match any of those specified in `pgname`.

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.

pgname — 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
j1939ParameterGroup | extractAll | extractTime

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

Package: 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
`j1939ParameterGroup` | `extractAll` | `extractRecent`

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

Package: 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
`j1939Channel` | `filterAllowOnly` | `filterBlockOnly`

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

Package: 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
`j1939Channel` | `filterAllowAll` | `filterBlockOnly`

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

Package: 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
`j1939Channel` | `filterAllowAll` | `filterAllowOnly`

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 server module.

```
connect(xcpch)
```

Set up 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 server module defined by the A2L file.

See Also

`xcpA2L` | `xcpChannel` | `createMeasurementList` | `viewMeasurementLists`

Introduced in R2013a

getCharacteristicInfo

Get information about specific characteristic from A2L file

Syntax

`info = getCharacteristicInfo(a2lFile,characteristic)`

Description

`info = getCharacteristicInfo(a2lFile,characteristic)` returns information about the specified characteristic from the specified A2L file, and stores it in the xcp.Characteristic object, `info`.

Examples

Get XCP Characteristic Information

Create a handle to parse an A2L file and get information about the `curve1_8_uc` characteristic.

```
a2lfile = xcpA2L('C:\XCPSIM.a2l');
info = getCharacteristicInfo(a2lfile,'curve1_8_uc')
```

```
info = 

  Characteristic with properties:

     CharacteristicType: 'VAL_BLK'
                Deposit: [1×1 xcp.RecordLayout]
         AxisConversion: {}
                   Name: 'curve1_8_uc'
         LongIdentifier: '8 BYTE shared axis Curve2'
             ECUAddress: 1131912
    ECUAddressExtension: 0
             Conversion: [1×1 xcp.CompuMethodRational]
              Dimension: [8 1 1]
             LowerLimit: 0
             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

`xcpA2L` | `getEventInfo` | `getMeasurementInfo`

Topics
"Get Started with A2L-Files" on page 14-231
"XCP Database and Communication Workflow" on page 5-2

Introduced in R2018a

getEventInfo

Get event information about specific event from A2L file

Syntax

`info = getEventInfo(a2lFile,eventName)`

Description

`info = getEventInfo(a2lFile,eventName)` returns information about the specified event from the specified A2L file, and stores it in the xcp.Event object, `info`.

Examples

Get XCP Event Information

Create a handle to parse an A2L file and get information about the `'10 ms'` event.

```
a2lfile = xcpA2L('C:\XCPSIM.a2l')
info = getEventInfo(a2lfile,'10 ms')
```

```
info =
  Event with properties:
                    Name: '10 ms'
               Direction: 'DAQ_STIM'
              MaxDAQList: 255
           ChannelNumber: 1
        ChannelTimeCycle: 10
         ChannelTimeUnit: 6
         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
`xcpA2L` | `getCharacteristicInfo` | `getMeasurementInfo`

Topics
"Get Started with A2L-Files" on page 14-231
"XCP Database and Communication Workflow" on page 5-2

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: [1×1 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

`xcpA2L` | `getCharacteristicInfo` | `getEventInfo`

Topics
"Get Started with A2L-Files" on page 14-231
"XCP Database and Communication Workflow" on page 5-2

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` | `systemList` | `setValue` | `write`

Introduced in R2019a

hasdata

Package: `matlab.io.datastore`

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','data','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 =

  4×4 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 =

  1×6 table

      ShortName        System      Category      Value      Units    FeatureReference
    _______________    _________    ________    _________    _____    ________________

    "VALUE_NUMERIC"    "System1"    "VALUE"     [12.2400]     ""          "model1"
```

```
iList = instanceList(cdfxObj,"VALUE_NUMERIC","System1")
```

```
iList =
```

```
  1×6 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` | `systemList` | `getValue` | `setValue` | `write`

Introduced in R2019a

isConnected

Connection status

Syntax

`isConnected(xcpch)`

Description

`isConnected(xcpch)` returns a logical value to indicate active connection to the server.

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 server 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 logical 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 server module.

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

Set up a data acquisition measurement list with the `'10 ms'` event and `'Bitslice'` measurement and determine if the measurement is running.

```
createMeasurementList(xcpch,'DAQ','10 ms','BitSlice')
isMeasurementRunning(xcpch)
```

```
ans =

     0
```

Start your measurement and verify that the 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 server 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: [0×0 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 `j1939.Channel` object, with j1939.Channel Properties.

See Also

Functions
`canDatabase` | `j1939ParameterGroup` | `transmit` | `receive`

Properties
j1939.Channel Properties

Topics
"J1939 Channel Workflow" on page 7-6

Introduced in R2015b

j1939ParameterGroup

Create J1939 parameter group

Syntax

```
pg = j1939ParameterGroup(database,name)
pg = j1939ParameterGroup(database,j1939TimeTable)
```

Description

`pg = j1939ParameterGroup(database,name)` creates a parameter group using the `name` defined in the specified `database`.

`pg = j1939ParameterGroup(database,j1939TimeTable)` creates parameter groups from the specified J1939 parameter group timetable. This allows you to convert parameter group timetables into arrays of parameter group objects to be used in code from earlier versions of the toolbox. For performance reasons, it is recommended that you work with timetables instead of parameter group objects.

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`

Output Arguments

pg — J1939 parameter group
parameter group object

J1939 parameter group, returned as a parameter group object, with j1939.ParameterGroup Properties.

See Also

Functions
`canDatabase` | `j1939Channel`

Properties
j1939.ParameterGroup Properties

Topics
"J1939 Interface" on page 7-2

"J1939 Parameter Group Format" on page 7-3

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 assigned to the timetable `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
timetable of parameter groups

J1939 parameter groups, returned as a timetable of parameter groups.

See Also

Functions
`canDatabase` | `j1939ParameterGroupTimetable` | `j1939SignalTimetable`

Introduced in R2017a

j1939ParameterGroupTimetable

Convert CAN messages or J1939 parameter groups into timetable

Syntax

```
j1939PGTT = j1939ParameterGroupTimetable(msg)
j1939PGTT = j1939ParameterGroupTimetable(msg,database)
```

Description

Handling parameter group information in a timetable format allows significantly faster processing of J1939 network data across a wide array of workflows.

`j1939PGTT = j1939ParameterGroupTimetable(msg)` takes the input messages as an array of J1939 parameter group objects and returns a J1939 parameter group timetable. The timetable contains the decoded data (PGN, Priority, Data, etc.) from the input J1939 traffic. Use this function to convert J1939 information received as objects in earlier versions of the toolbox to the preferred timetable data type.

`j1939PGTT = j1939ParameterGroupTimetable(msg,database)` takes the input messages as either a CAN message timetable, an ASAM MDF CAN message timetable, an array of CAN message objects, a CAN message structure from the CAN Log block, an array of J1939 parameter group objects, or an existing J1939 parameter group timetable and returns a J1939 parameter group timetable. If CAN messages are input, the database is used to transform the CAN messages into J1939 parameter groups. If J1939 parameter groups are input, the database is used to re-decode the J1939 parameter group signals.

All CAN message information given as input must originate from a J1939 network. If the provided J1939 database does not contain the information needed to decode the input CAN messages, the output J1939 parameter group timetable is empty.

Examples

Convert Various Message Information to J1939 Parameter Group Timetable

Convert CAN and J1939 data from various formats.

Convert the output structure from a CAN Log block.

```
load LogBlockOutput.mat
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(canMsgs, db)
```

Convert an array of CAN message objects.

```
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(canMsgObjects, db)
```

Convert a timetable of CAN messages.

```
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(canMsgTimetable, db)
```

Convert ASAM MDF CAN messages.

```
m = mdf("LogFile.mf4")
mdfData = read(m, 2, m.ChannelNames{2})
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(mdfData, db)
```

Convert Vector BLF CAN messages.

```
blfData = blfread("LogFile.blf", 1)
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(blfData, db)
```

Repackage J1939 parameter group objects

```
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(j1939PGObjects, db)
```

Re-decode signals in an existing J1939 parameter group timetable.

```
db = canDatabase("Database.dbc")
j1939PGTT = j1939ParameterGroupTimetable(j1939PGTimetable, db)
```

Input Arguments

msg — Message data
timetable | array | structure

Message data, in one of the following formats:

- Array of J1939 parameter group objects
- Timetable of J1939 parameter groups
- Timetable of CAN messages
- Timetable of ASAM MDF CAN messages
- Array of CAN message objects
- Structure of CAN messages from a CAN Log block

database — CAN database
database handle

CAN database, specified as a database handle, created with the `canDatabase` function.

Output Arguments

j1939PGTT — Timetable of J1939 parameter groups
timetable

J1939 parameter groups, returned as a timetable.

See Also

Functions
`canDatabase` | `j1939ParameterGroup` | `j1939ParameterGroupImport` | `j1939SignalTimetable`

Introduced in R2021a

j1939SignalTimetable

Create J1939 signal timetable from J1939 parameter group timetable

Syntax

```
sigTables = j1939SignalTimetable(pgTable)
sigTables = j1939SignalTimetable(pgTable,"ParameterGroups",pgNames)
sigTables = j1939SignalTimetable( ___ ,"IncludeAddresses",true)
```

Description

`sigTables = j1939SignalTimetable(pgTable)` converts a timetable of J1939 parameter group information into individual timetables of signal values. The function returns a structure with a field for each unique parameter group in the timetable. Each field value is a timetable of all the signals in that parameter group. Use this form of syntax to convert an entire set of parameter groups in a single function call.

`sigTables = j1939SignalTimetable(pgTable,"ParameterGroups",pgNames)` returns signal timetables for only the parameter groups specified by `pgNames`, which can specify one or more parameter group names. Use this form of syntax to quickly convert only a subset of parameter groups into signal timetables.

`sigTables = j1939SignalTimetable( ___ ,"IncludeAddresses",true)` adds source and destination addresses to each J1939 signal timetable. The default argument value is `false`, in which case the J1939 signal timetables do not include addresses.

.

Examples

Create J1939 Signal Timetables from All Parameter Groups

Create J1939 signal timetables from all data in a J1939 parameter group timetable.

```
sigTables = j1939SignalTimetable(pgTable);
```

Create J1939 Signal Timetables from Specified Parameter Groups

Create J1939 signal timetables from only specified J1939 parameter groups in a timetable.

```
sigTable1 = j1939SignalTimetable(pgTable,"ParameterGroups","pgName");
sigTable2 = j1939SignalTimetable(pgTable,"ParameterGroups",{"pgName1","pgName2"});
```

Input Arguments

`pgTable` — J1939 parameter group timetable
timetable

J1939 parameter groups, specified as a timetable.

`pgNames` — Parameter group names
char | string | cell

J1939 parameter group names, specified as a character vector, string, or array.

Data Types: `char` | `string` | `cell`

Output Arguments

`sigTables` — J1939 signals
structure

J1939 signals, returned as a structure. The structure field names correspond to the parameter groups of the input, and each field value is a timetable of J1939 signals.

Data Types: `struct`

See Also

Functions
`j1939ParameterGroupImport` | `j1939ParameterGroupTimetable`

Introduced in R2021a

mdf

Access information contained in MDF-file

Syntax

```
mdfObj = mdf(mdfFileName)
```

Description

The `mdf` function creates an object for accessing a measurement data format (MDF) file. See “Measurement Data Format (MDF)” on page 11-138.

`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: 'mmddff.04'
              Creator: [1x1 struct]

   File Contents
           Attachment: [1×1 struct]
         ChannelNames: {6×1 cell}
         ChannelGroup: [1×6 struct]
```

```
   Options
          Conversion: Numeric
```

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
`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`

Property	Description
`Conversion`	Conversion option for data in the MDF-file. Supported values are: • `'Numeric'` (default) — Apply only numeric conversion rules (CC_Type 1-6). Data with non-numeric conversion rules is imported as raw, unconverted values. • `'None'` — Do not apply any conversion rules. All data is imported as raw data. • `'All'` — Apply all numeric and text conversion rules (CC_Type 1-10).

More About

Measurement Data Format (MDF)

Measurement data format (MDF) files are binary format files for storing measurement data. The format standard is defined by the Association for Standardization of Automation and Measuring Systems (ASAM), which you can read about at ASAM MDF.

Vehicle Network Toolbox and Powertrain Blockset™ provide access to MDF-files through an object you create with the `mdf` function.

See Also

Functions
`saveAttachment` | `read` | `mdfVisualize` | `mdfInfo` | `mdfSort`

Topics
“Get Started with MDF-Files” on page 14-79
“Read Data from MDF-Files” on page 14-83
“Data Analytics Application with Many MDF-Files” on page 14-110
“File Format Limitations” on page 9-5
“Troubleshooting MDF Applications” on page 9-7

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 11-140. 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
table

This property is read-only.

All channel groups present in first MDF-file, returned as a table.

Data Types: `table`

`Channels` — All channels present in first MDF-file
table

This property is read-only.

All channels present in first MDF-file, returned as a table.

Those channels targeted for reading must have the same name and belong to the same channel group in each file of the MDF datastore.

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.

Those channels targeted for reading must have the same name and belong to the same channel group in each file of the MDF datastore.

Example: `'Counter_B4'`

Data Types: `char` | `string` | `cell`

`Conversion` — Conversion option for MDF-file data
`'Numeric'` (default) | `'All'` | `'None'`

Conversion option for MDF-file data, specified as `'Numeric'`, `'All'`, or `'None'`.

- `'Numeric'` (default) — Apply only numeric conversion rules (CC_Type 1-6). Data with non-numeric conversion rules is imported as raw, unconverted values.
- `'None'` — Do not apply any conversion rules. All data is imported as raw data.
- `'All'` — Apply all numeric and text conversion rules (CC_Type 1-10).

Example: `'All'`

Data Types: `char` | `string`

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
isPartitionable (MATLAB)	Determine whether datastore is partitionable
isShuffleable (MATLAB)	Determine whether datastore is shuffleable

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','data','CANape.MF4'));
while hasdata(mdfds)
    m = read(mdfds);
end
```

See Also

Topics

"Get Started with MDF Datastore" on page 14-88

Introduced in R2017b

mdfFinalize

Finalize MDF-file by ASAM standards

Syntax

```
mdfFinalize(UnfinalizedMDFFile)
mdfFinalize(UnfinalizedMDFFile,FinalizedMDFFile)
finalizedPath = mdfFinalize( ___ )
```

Description

`mdfFinalize(UnfinalizedMDFFile)` sorts and finalizes the specified MDF-file according to ASAM standards, and overwrites the original file.

`mdfFinalize(UnfinalizedMDFFile,FinalizedMDFFile)` creates a sorted, finalized copy of the MDF-file with the specified name, `FinalizedMDFFile`.

`finalizedPath = mdfFinalize( ___ )` returns an output argument, `finalizedPath`, indicating the full path to the sorted, finalized file, including the file name.

Note This function is supported only on 64-bit Windows operating systems.

Examples

Finalize an MDF-File in Place

Finalize an MDF-file, overwriting the original.

```
finalizedPath = mdfFinalize('MDFFile.mf4');
mdfObj = mdf(finalizedPath);
```

Finalize an MDF-File into a Copy

Finalize an MDF-file, creating a separate copy from the original.

```
finalizedPath = mdfFinalize('UnfinalizedMDFFile.mf4','FinalizedMDFFile.mf4');
mdfObj = mdf(finalizedPath);
```

Input Arguments

`UnfinalizedMDFFile` — Original unfinalized MDF-file

string | char

Original unfinalized MDF-file, specified as a string or character vector. Full and relative path names are allowed.

Example: `'UnfinalizedMDFFile.mf4'`

Data Types: char | string

FinalizedMDFFile — New finalized copy of MDF-file
string | char

New finalized copy of MDF-file, specified as a string or character vector. Full and relative path names are allowed.

Example: 'FinalizedMDFFile.mf4'

Data Types: char | string

Output Arguments

finalizedPath — Path to finalized file
char

Full path to finalized file, returned as a character vector. The path includes the file name.

See Also

Functions
mdf | read | mdfSort

Introduced in R2021b

mdfInfo

Information about MDF-file

Syntax

```
fileInfo = mdfInfo(mdfFileName)
```

Description

`fileInfo = mdfInfo(mdfFileName)` returns a struct that contains information about the specified MDF-file, including name, location, version, size, and initial timestamp of the data.

Examples

Access Information About MDF-File

Get the MDF-file information, and programmatically read its version.

```
fileInfo = mdfInfo('MDFFile.mdf');
fileInfo.Version
```

```
ans =

    '3.20'
```

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

`fileInfo` — MDF-file information
structure

MDF-file information, returned as a structure.

See Also

Functions
`mdf`

Introduced in R2019b

mdfSort

Sort MDF-file by ASAM standards

Syntax

```
mdfSort(UnsortedMDFFile)
mdfSort(UnsortedMDFFile,SortedMDFFile)
sortedPath = mdfSort( ___ )
```

Description

If you get an error when trying to read an unsorted MDF-file, sort the file with `mdfSort` and read from that instead.

`mdfSort(UnsortedMDFFile)` sorts the specified MDF-file according to ASAM standards for fast reading. The sorted result overwrites the original file.

`mdfSort(UnsortedMDFFile,SortedMDFFile)` creates a sorted copy of the MDF-file with the specified name, `SortedMDFFile`.

`sortedPath = mdfSort( ___ )` returns an output argument, `sortedPath`, indicating the full path to the sorted file, including the file name.

Note This function is supported only on 64-bit Windows operating systems.

Examples

Sort an MDF-File in Place

Sort an MDF-file, overwriting the original, and read its data.

```
sortedPath = mdfSort('MDFFile.mf4');
mdfObj = mdf(sortedPath);
data = read(mdfObj);
```

Sort an MDF-File into a Copy

Create a sorted copy of an MDF-file and read its data.

```
sortedPath = mdfSort('UnsortedMDFFile.mf4','SortedMDFFile.mf4');
mdfObj = mdf(sortedPath);
data = read(mdfObj);
```

Input Arguments

`UnsortedMDFFile` — Original MDF-file with unsorted data
string | char

Original MDF-file without sorted data, specified as a string or character vector. Full and relative path names are allowed.

Example: `'UnsortedMDFFile.mf4'`

Data Types: `char` | `string`

`SortedMDFFile` — New copy of MDF-file with sorted data
string | char

New copy of MDF-file with sorted data, specified as a string or character vector. Full and relative path names are allowed.

Example: `'SortedMDFFile.mf4'`

Data Types: `char` | `string`

Output Arguments

`sortedPath` — Path to sorted file
char

Full path to sorted file, returned as a character vector. The path includes the file name.

See Also

Functions
`mdf` | `read` | `mdfFinalize`

Introduced in R2019b

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.

Note `mdfVisualize` supports only integer and floating point data types in MDF-file channels.

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
`canDatabase` | `attributeInfo` | `nodeInfo` | `signalInfo` | `canMessage`

Properties
can.Database Properties

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:

Field	Description
`Name`	Node name
`Comment`	Text about node

See Also

Functions
`attributeInfo` | `messageInfo` | `signalInfo` | `canDatabase`

Properties
can.Database Properties

Introduced in R2015b

numpartitions

Package: `matlab.io.datastore`

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','data','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` | `read` | `reset` | `partition`

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 x 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

Package: `matlab.io.datastore`

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','data','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 =
  3×1 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` | `read` | `reset` | `numpartitions`

Introduced in R2017b

preview

Package: `matlab.io.datastore`

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','data','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
`mdfDatastore` | `read` | `hasdata`

Introduced in R2017b

read

Read channel data from MDF-file

Syntax

```
data = read(mdfObj)
data = read(mdfObj,chanList)
data = read(mdfObj,chanGroupIndex,chanName)
data = read(mdfObj,chanGroupIndex,chanName,startPosition)
data = read(mdfObj,chanGroupIndex,chanName,startPosition,endPosition)
data = read( ___ ,'Conversion',convOpt)
data = read( ___ ,'OutputFormat',fmtType)
[data,time] = read( ___ ,'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,chanList)` reads all data for all channels specified in the channel list table `chanList`.

`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( ___ ,'Conversion',convOpt)` applies the specified conversion option to the MDF data when reading it in. This option overrides the setting of the `Conversion` property of the `mdf` object.

`data = read( ___ ,'OutputFormat',fmtType)` returns data with the specified output format.

`[data,time] = read( ___ ,'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 Raw Data

Read raw data from a specified channel in the first channel group, without applying any conversion rules.

```
mdfObj = mdf('MDFFile.mf4');
data = read(mdfObj,1,'Unsigned_UInt32_LE_Primary_Offset_0','Conversion','None');
data(1:4,:)
```

```
ans =

  4×1 timetable

    Time     Unsigned_UInt32_LE_Primary_Offset_0
    _____    ___________________________________

    0 sec                    0
    1 sec                    1
    2 sec                    2
    3 sec                    3
```

Read All Data from Specified Channel List

Read all available data from the MDF-file for channels specified as part of a channel list.

```
mdfObj = mdf('MDFFile.mf4');
chanList = channelList(mdfObj) % Channel table
data = read(mdfObj,chanList(1:3,:)); % First 3 channels
```

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(1:2,1:2) % Display 2 channels, 2 columns
```

```
  2×2 table

             ChannelName                ChannelGroupNumber
    _______________________________     __________________

    "Float_32_LE_Offset_64"                     2
    "Float_64_LE_Primary_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')`

`chanList` — List of channels
table

List of channels, specified as a table in the format returned by the `channelList` function.

Example: `channelList()`

Data Types: `table`

`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'`	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.) 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 `ChannelNames` property. The column headers are derived from the property using the function `matlab.lang.makeValidName`. The original channel names are available in the `VariableDescriptions` property of the `timetable` object.
`'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`

`convOpt` — Conversion option for MDF-file data
`'Numeric'` (default) | `'All'` | `'None'`

Conversion option for MDF-file data, specified as `'Numeric'`, `'All'`, or `'None'`. The default uses the value specified in the `Conversion` property of the `mdf` object.

- `'Numeric'` — Apply only numeric conversion rules (CC_Type 1-6). Data with non-numeric conversion rules is imported as raw, unconverted values.
- `'None'` — Do not apply any conversion rules. All data is imported as raw data.
- `'All'` — Apply all numeric and text conversion rules (CC_Type 1-10).

Example: `'All'`

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` | `saveAttachment` | `mdfVisualize` | `mdfInfo` | `mdfSort` | `channelList`

Topics
“Get Started with MDF-Files” on page 14-79
“Read Data from MDF-Files” on page 14-83
“Time Series”
“Represent Dates and Times in MATLAB”
“Tables”

Introduced in R2016b

read

Package: `matlab.io.datastore`

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: [1×1 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
`mdfDatastore` | `readall` | `preview` | `reset` | `hasdata`

Topics
"Get Started with MDF Datastore" on page 14-88

Introduced in R2017b

readall

Package: `matlab.io.datastore`

Read all data in MDF datastore

Syntax

```
data = readall(mdfds)
data = readall(mdfds,"UseParallel",true)
```

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.

`data = readall(mdfds,"UseParallel",true)` specifies to use a parallel pool to read all of the data. By default, the `"UseParallel"` option is `false`. The choice of pool depends on the following conditions:

- If you already have a parallel pool running, that pool is used.
- If your parallel preference settings allow a pool to automatically start, this syntax will start one, using the default cluster.
- If no pool is running and one cannot automatically start, this syntax does not use parallel functionality.

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);
```

Read All Data in Datastore

Use a parallel pool to read all the data from the datastore into a timetable.

```
mdfds = mdfDatastore({'CANape1.MF4','CANape2.MF4','CANape3.MF4'});
data = readall(mdfds,"UseParallel",true);
```

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
`mdfDatastore` | `read` | `preview` | `reset` | `hasdata`

Topics
"Get Started with MDF Datastore" on page 14-88

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 server 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
`writeAxis` | `readCharacteristic` | `writeCharacteristic` | `readMeasurement` | `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 server 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` | `writeAxis` | `writeCharacteristic` | `readMeasurement` | `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 server 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 server.

```
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 server 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 6-9

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 server 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` | `writeAxis` | `readCharacteristic` | `writeCharacteristic` | `writeMeasurement`

Topics
"Read XCP Measurements with Direct Acquisition" on page 14-265

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'` event `'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 server 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 server 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

Package: 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
timetable of parameter groups

J1939 parameter groups, returned as a timetable.

See Also

Functions
`j1939Channel` | `start` | `transmit` | `j1939ParameterGroupTimetable`

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

Package: `matlab.io.datastore`

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 your 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','data','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
`mdfDatastore` | `read` | `hasdata`

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` | `instanceList` | `systemList` | `getValue` | `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

can.Database Properties

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` | `stop` | `configBusSpeed`

Introduced in R2009a

start

Package: 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 server module and STIM lists begin transmitting data values to the server module.

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:

              ServerName: 'CPP'
             A2LFileName: 'XCPSIM.a2l'
          TransportLayer: 'CAN'
    TransportLayerDevice: [1x1 struct]
      SeedKeyCallbackFcn: []
                KeyValue: []
```

Connect the channel to the server module.

```
connect(xcpch)
```

Set up 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:

              ServerName: 'CPP'
             A2LFileName: 'XCPSIM.a2l'
          TransportLayer: 'CAN'
    TransportLayerDevice: [1x1 struct]
      SeedKeyCallbackFcn: []
                KeyValue: []
```

Connect the channel to the server module.

```
connect(xcpch)
```

Set up a data stimulation measurement list with the ‘100ms’ event and `'Bitslice0'`, `'PWMFiletered'`, and `'Triangle'`measurements.

```
createMeasurementList(xcpch,'STIM','100ms',{'BitSlice0','PWMFiletered','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 server 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

Package: 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 server module and STIM lists stop transmitting data values to the server module.

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:

              ServerName: 'CPP'
             A2LFileName: 'XCPSIM.a2l'
          TransportLayer: 'CAN'
    TransportLayerDevice: [1x1 struct]
      SeedKeyCallbackFcn: []
                KeyValue: []
```

Connect the channel to the server module.

```
connect(xcpch)
```

Set up 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 server 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

    ShortName      Instances       Metadata
    _________    ______________    ________

    "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 | instanceList | getValue | 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

Package: `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` | `start` | `receive`

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,'On',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 specified 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 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]
```

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 a 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
`nodeInfo` | `messageInfo` | `signalInfo` | `attributeInfo` | `canDatabase`

Properties
can.Database Properties

Introduced in R2015b

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:

              ServerName: 'CPP'
             A2LFileName: 'XCPSIM.a2l'
          TransportLayer: 'CAN'
    TransportLayerDevice: [1x1 struct]
      SeedKeyCallbackFcn: []
                KeyValue: []
```

Connect the channel to the server module.

```
connect(xcpch)
```

Set up 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 server 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` | `instanceList` | `systemList` | `getValue` | `setValue`

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 server module.

Examples

Write Value to XCP Channel Axis

Write a value to an XCP axis and verify the value.

Read the 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 a new value.

```
newValue = 50;
writeAxis(chanObj,axisObj,newValue);
```

Read the 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` | `writeCharacteristic` | `readMeasurement` | `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 server 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 a new value.

```
newValue = 200;
writeCharacteristic(chanObj,charObj,newValue');
```

Read the value again to verify the 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` | `writeAxis` | `readCharacteristic` | `readMeasurement` | `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 server module.

Examples

Write Value to an XCP Channel Measurement

Write a value to an XCP measurement, and verify the value.

Read the 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` | `writeAxis` | `readCharacteristic` | `writeCharacteristic` | `readMeasurement`

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 server 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 server 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 server 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 server 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 server.

```
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 server 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')
```

```
a2lfile = 

  A2L with properties:

   File Details
                 FileName: 'XCPSIM.a2l'
                 FilePath: 'c:\XCPSIM.a2l'
               ServerName: 'CPP'
                 Warnings: [0×0 string]

   Parameter Details
                   Events: {1×6 cell}
                EventInfo: [1×6 xcp.a2l.Event]
             Measurements: {1×45 cell}
          MeasurementInfo: [45×1 containers.Map]
          Characteristics: {1×16 cell}
       CharacteristicInfo: [16×1 containers.Map]
                 AxisInfo: [1×1 containers.Map]
            RecordLayouts: [41×1 containers.Map]
             CompuMethods: [15×1 containers.Map]
                CompuTabs: [0×1 containers.Map]
               CompuVTabs: [2×1 containers.Map]

   XCP Protocol Details
        ProtocolLayerInfo: [1×1 xcp.a2l.ProtocolLayer]
                  DAQInfo: [1×1 xcp.a2l.DAQ]
    TransportLayerCANInfo: [1×1 xcp.a2l.XCPonCAN]
    TransportLayerUDPInfo: [1×1 xcp.a2l.XCPonIP]
    TransportLayerTCPInfo: [0×0 xcp.a2l.XCPonIP]
```

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`

Output Arguments

`a2lfile` — A2L file
`xcp.A2L` object

A2L file, returned as an `xcp.A2L` object, with xcp.A2L Properties.

See Also

Functions
`xcpChannel` | `getEventInfo` | `getMeasurementInfo`

Properties
xcp.A2L Properties

Topics
"Get Started with A2L-Files" on page 14-231
"XCP Database and Communication Workflow" on page 5-2

External Websites
ASAM MCD-2 MC Technical Content

Introduced in R2013a

xcpChannel

Create XCP channel

Syntax

```
xcpch = xcpChannel(a2lFile,CANProtocol,vendor,deviceID)
xcpch = xcpChannel(a2lFile,CANProtocol,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,CANProtocol,vendor,deviceID)` creates a channel connected to the CAN bus via the specified vendor and device, using the specified `CANProtocol` of `"CAN"` or `"CAN FD"`. The XCP channel accesses the server module via the CAN bus, parsing the attached A2L file.

Use this syntax for vendor `"PEAK-System"` or `"NI"`. With NI CAN devices, the `deviceID` argument must include the interface number defined for the channel in the NI Measurement & Automation Explorer.

Note: XCP over CAN FD is not supported for PEAK-System devices.

`xcpch = xcpChannel(a2lFile,CANProtocol,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.

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 Server 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:

              ServerName: '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:

              ServerName: '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`.

CANProtocol — CAN protocol mode
"CAN" | "CAN FD"

CAN protocol mode, specified as "CAN" or "CAN FD".

Example: "CAN"

Data Types: char | string

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 NI 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

portNmbr — 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 xcp.Channel object with xcp.Channel Properties.

See Also

Functions
xcpA2L | connect | disconnect | isConnected

Properties
xcp.Channel Properties

Introduced in R2013a

Properties by Class

can.Channel Properties

Properties of the `can.Channel` object

Description

Use the following properties to examine or configure CAN channel settings. Use `canChannel` to create a CAN channel object.

Properties

Device Information

`DeviceVendor` — Device vendor name
char vector

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

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `char`

`Device` — Channel device type
char vector

This property is read-only.

For National Instruments devices, the `Device` property displays the device number on the hardware.

For all other vendors, the `Device` property displays information about the device type to which the CAN or J1939 channel is connected.

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `char`

`DeviceChannelIndex` — Device channel index
double

This property is read-only.

The `DeviceChannelIndex` property indicates the channel index on which the specified CAN or J1939 channel is configured.

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `double`

`DeviceSerialNumber` — Device serial number
double | char

This property is read-only.

The DeviceSerialNumber property displays the serial number of the device connected to the CAN or J1939 channel.

Values are automatically defined when you configure the channel with the canChannel or j1939Channel function.

Data Types: double | char

ProtocolMode — Protocol mode of CAN channel
'CAN' (default) | 'CAN FD'

This property is read-only.

The ProtocolMode property indicates the communication protocol for which the CAN channel is configured, either CAN or CAN FD.

The value is defined when you configure the channel with the canChannel function.

Data Types: char

Status Information

Running — Indicate running status of channel
false (0) | true (1)

This property is read-only.

The Running property indicates the state of the CAN or J1939 channel, according to the following values:

- false (default) — The channel is offline.
- true — The channel is online.

Use the start function to set your channel online.

Data Types: logical

MessagesAvailable — Number of messages available to be received by CAN channel
double

This property is read-only.

The MessagesAvailable property displays the total number of messages available to be received by a CAN channel. The value is 0 when no messages are available.

Data Types: double

MessagesReceived — Number of messages received by CAN channel
double

This property is read-only.

The MessagesReceived property indicates the total number of messages received since the channel was last started. The value is 0 when no messages have been received, and increments based on the number of messages the channel receives.

Data Types: double

MessagesTransmitted — Number of messages transmitted by CAN channel
double

This property is read-only.

The MessagesTransmitted property indicates the total number of messages transmitted since the channel was last started. The default value is 0 when no messages have been sent, and increments based on the number of messages the channel transmits.

Data Types: double

MessageReceivedFcn — Callback function to run when messages available
function handle | char | string

Configure MessageReceivedFcn as a callback function to run, specified as a character vector, string, or a function handle, when a required number of messages are available.

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

For example, to specify the callback function to execute:

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

Data Types: char | string | function_handle

MessageReceivedFcnCount — Specify number of messages available before callback is triggered
numeric

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

The default value is 1. You can specify a positive integer for your MessageReceivedFcnCount. For example, to specify the message count required to trigger a callback:

```
canch.MessageReceivedFcnCount = 55;
```

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

InitializationAccess — Indicate control of device channel
true (1) | false (0)

This property is read-only.

The InitializationAccess property indicates if the configured CAN or J1939 channel object has full control of the device channel, according to the following values:

- true — Has full control of the hardware channel and can change the property values.
- false — Does not have full control and cannot change property values.

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.

Data Types: `logical`

`InitialTimestamp` — Indicate when channel started
datetime

This property is read-only.

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.

Data Types: `datetime`

`FilterHistory` — Indicate settings of message acceptance filters
char

This property is read-only.

Indicate settings of message acceptance filters, returned as a character vector. This property indicates the settings implemented by the functions `filterAllowOnly`, `filterAllowAll`, and `filterBlockAll`.

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

Data Types: `char`

Channel Information

`BusStatus` — Status of bus
char

This property is read-only.

The `BusStatus` property displays information about the state of the CAN bus or the J1939 bus.

- `'N/A'` — Property not supported by vendor.
- `'ErrorActive'` — Node transmits Active Error Flags when it detects errors. Note: This status does not necessarily indicate that an error actually exists, but indicates how an error is handled.
- `'ErrorPassive'` — Node transmits Passive Error Flags when it detects errors.
- `'BusOff'` — Node will not transmit anything on the bus.

Data Types: `char`

`SilentMode` — Specify if channel is active or silent
`false` (default) | `true`

Specify whether the channel operates silently, according to the following values:

- `false` (default) — The channel is in normal or active mode. 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.

- `true` — The channel is in silent mode. You can observe all message activity on the network and perform analysis without affecting the network state or behavior. In this mode, you can only receive messages and not transmit any.

Data Types: `logical`

`TransceiverName` — Name of device transceiver
char

This property is read-only.

`TransceiverName` indicates 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.

Data Types: `char`

`TransceiverState` — Specify state or mode of transceiver
numeric

If your CAN or J1939 transceiver allows you to control its mode, you can use the `TransceiverState` property to set the mode.

The numeric property value for each mode is 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`

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

`ReceiveErrorCount` — Number of errors during message reception
double

This property is read-only.

The `ReceiveErrorCount` property indicates the total number of errors during message reception since the channel was last started. The default value is `0`, and increments based on the number of errors.

Data Types: `double`

`TransmitErrorCount` — Number of errors during message transmission
double

This property is read-only.

The `TransmitErrorCount` property indicates the total number of errors during message transmission since the channel was last started. The default value is `0`, and increments based on the number of errors.

Data Types: `double`

BusSpeed — Bit rate of bus transmission
double

This property is read-only.

The BusSpeed property indicates the speed at which messages are transmitted in bits per second. The default value is assigned by the vendor driver.

You can set BusSpeed to a supported bit rate using the configBusSpeed function, specifying the channel name and the bit rate value as input parameters. For example, to change the bus speed of the CAN channel object canch to 250,000 bits per second, and view the result, type

```
configBusSpeed(canch,250000);
bs = canch.BusSpeed
```

Data Types: double

SJW — Synchronization jump width (SJW) of bit time segment
double

This property is read-only.

SJW displays the synchronization jump width of the bit time segment. To adjust the on-chip bus clock, the controller can 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.

Data Types: double

TSEG1,TSEG2 — Allowed number of bits segments to lengthen and shorten sample time
double

This property is read-only.

The TSEG1 and TSEG2 properties indicate the amount in bit time segments that the channel can lengthen and shorten the sample time, respectively, to resynchronize or compensate for delay times in the network. The value is inherited when you configure the bus speed of your CAN channel.

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

Data Types: double

NumOfSamples — Number of samples available to channel
double

This property is read-only.

The NumOfSamples property is a bit timing parameter that indicates the number of bit samples performed for a single bit read on the network. The value is a positive integer based on the driver settings for the channel.

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

Data Types: `double`

`BusLoad` — Load on CAN bus
double

This property is read-only.

The `BusLoad` property provides information about the load on the CAN network for message traffic on Kvaser devices. The current message traffic on a CAN network is represented as a percentage ranging from `0.00%` to `100.00%`.

Data Types: `double`

`OnboardTermination` — Configure bus termination on device
`true` (1) | `false` | (0)

The `OnboardTermination` property specifies whether the NI-XNET device uses its onboard termination of the CAN bus. For more information on the behavior and characteristics of a specific device, refer to its vendor documentation.

Data Types: `logical`

`StartTriggerTerminal` — Specify start trigger source terminal
char | string

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.

Examples

Configure an 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` values of the received CAN messages are relative to the trigger moment.

```
ch1.InitialTimestamp
messages = receive(ch1,Inf);
messages(1).Timestamp
```

Data Types: `char` | `string`

Other Information

DataBase — CAN database information
struct

The `Database` property stores information about an attached CAN database. If your channel message is not attached to a database, the property value is an empty structure, `[]`. You can edit the CAN channel `Database` property, but cannot edit the CAN message `Database` property.

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 in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

Data Types: `struct`

UserData — Custom data
any data

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.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

Functions
`canChannel` | `configBusSpeed` | `receive` | `transmit`

Topics
"CAN and CAN FD Communication"

Introduced in R2009a

can.Message Properties

Properties of the `can.Message` object

Description

Use the following properties to examine or configure CAN and CAN FD message settings. Use `canMessage` to create a CAN message.

Properties

Message Identification

`ProtocolMode` — Protocol mode of CAN channel
`'CAN'` (default) | `'CAN FD'`

This property is read-only.

The `ProtocolMode` property indicates the communication protocol for which the CAN message is configured, either `CAN` or `CAN FD`.

The value is defined when you configure the message with the `canChannel` function.

Data Types: `char`

`ID` — Identifier for CAN message
double

This property is read-only.

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

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

You can configure the message ID when constructing it. For example, to set a standard identifier of value 300 and a data length of eight bytes, type:

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

For hexadecimal values, convert using the `hex2dec` function.

Data Types: `double`

`Extended` — Identifier type for CAN message
`0` (`false`) (default) | `1` (`true`)

This property is read-only.

The `Extended` property is the identifier type for a CAN message. It can either be a standard identifier or an extended identifier, according to the following values:

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

You can configure the message extended property when constructing it. For example, 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)
```

Data Types: `logical`

Name — CAN message name
char

This property is read-only.

The `Name` property displays the name of the message, as 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.

Data Types: `char`

Data Details

`Timestamp` — Time when message received
double

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

You might want to set the value when constructing a message. For example, to set the time stamp of a message to 12, type:

```
message.Timestamp = 12
```

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

`Data` — CAN message raw data
uint8 array

Use the `Data` property to define the raw data in a CAN message. The data is an array of uint8 values, based on the data length you specify in the message.

For example, to create a CAN message and define its data:

```
message = canMessage(2500,true,8)
message.Data = [23 43 23 43 54 34 123 1]
```

If you are using a CAN database for your message definitions, you can directly specify values in the `Signals` property structure.

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

Data Types: `uint8`

`Signals` — Physical signals defined in CAN message
struct

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

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
```

Data Types: `struct`

`Length` — Length of CAN message
uint8

Length of the CAN message in bytes, specified as a uint8 value. This indicates the number of elements in the `Data` vector. For CAN messages this is limited to 8 bytes; for CAN FD messages the length can be 0-8, 12, 16, 20, 24, 32, 48, or 64 bytes.

Data Types: `uint8`

`DLC` — CAN message data length code
uint8

This property is read-only.

Length code of the CAN FD message data, returned as a uint8 value. This relates to the `Length` property: for sizes up to 8 bytes they are the same, but `DLC` values ranging from 9 (binary 1001) to 15 (binary 1110) are used to specify the data lengths of 12, 16, 20, 24, 32, 48, and 64 bytes. For more information, see CAN FD - Some Protocol Details.

Data Types: `uint8`

Protocol Flags

BRS — CAN FD message bit rate switch
`0` (`false`) | `1` (`true`)

The BRS property indicates that the CAN FD message bit rate switch is set. This determines whether the bit rate for the data phase of the message is faster (`true`) or the same (`false`) as the bit rate of the arbitration phase. For more information, see CAN FD - Some Protocol Details.

Data Types: `logical`

ESI — CAN FD message error state indicator
`0` (`false`) | `1` (`true`)

This property is read-only.

The ESI property indicates that the CAN FD message error state indicator flag is set. For more information, see CAN FD - Some Protocol Details.

Data Types: logical

Error — CAN message error frame indicator
0 (false) | 1 (true)

This property is read-only.

The Error property indicates if true that the CAN message is an error frame.

Data Types: logical

Remote — Specify CAN message remote frame
false (default) | true

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

- false (default) — The message is not a remote frame.
- true — The message is a remote frame.

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

```
message.Remote = true
```

Data Types: logical

Other Information

DataBase — CAN database information
struct

The Database property stores information about an attached CAN database. If your channel message is not attached to a database, the property value is an empty structure, []. You can edit the CAN channel Database property, but cannot edit the CAN message Database property.

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 in your database name. Rename any CAN database files with non-alphanumeric characters before you use them.

Data Types: struct

UserData — Custom data
any data

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.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

Functions
`canChannel` | `canMessage` | `receive` | `transmit` | `pack` | `unpack`

Topics
"CAN and CAN FD Communication"

External Websites
CAN FD - Some Protocol Details

Introduced in R2009a

can.Database Properties

Properties of the `can.Database` object

Description

Use the following properties to examine or configure CAN database settings. Use `canDatabase` to create a CAN database object.

Properties

can.Database

Name — CAN database name
char

This property is read-only.

The `Name` property displays the name of the database, as a character vector value. This value is acquired from the database file name.

Data Types: `char`

Path — Path to CAN database file
char

This property is read-only.

The `Path` property displays the path of the database including the DBC-file, as a character vector.

Data Types: `char`

Nodes — Node names from CAN database
cell

This property is read-only.

The `Nodes` property stores the names of all nodes defined in the specified CAN database, as a cell array of character vectors. For example, to examine and index into the database nodes:

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

```
3×1 cell array

    {'AerodynamicControl'        }
    {'Aftertreatment_1_GasIntake'}
    {'Aftertreatment_1_GasOutlet'}
```

```
db.Nodes{1}
```

```
'AerodynamicControl'
```

Data Types: `cell`

`NodeInfo` — Information on CAN database nodes

struct

This property is read-only.

The `NodeInfo` property is a structure with information about all nodes defined in the specified CAN database. The `NodeInfo` property is a read-only structure. Use indexing to access the information of each node. For example:

```
db  = canDatabase('CANex.dbc');
db.NodeInfo
```

```
3×1 struct array with fields:

    Name
    Comment
    Attributes
    AttributeInfo
```

```
db.NodeInfo(1).Name
```

```
'AerodynamicControl'
```

Data Types: `struct`

`Messages` — Message names from CAN database

cell

This property is read-only.

The `Messages` property stores the names of all messages defined in the specified CAN database, as a cell array of character vectors.

```
db  = canDatabase('CANex.dbc');
db.Messages
```

```
3×1 cell array

    {'A1'     }
    {'A1DEFI' }
    {'A1DEFSI'}
```

```
db.Messages{1}
```

```
    'A1'
```

Data Types: `cell`

`MessageInfo` — Information on CAN database messages

struct

This property is read-only.

The `MessageInfo` property is a structure with information about all messages defined in the specified CAN database.

Use indexing to access the information of each message. For example:

```
db  = canDatabase('CANFDex.dbc');
db.MessageInfo
```

```
3×1 struct array with fields:

             Name: 'CANFDMessage'
     ProtocolMode: 'CAN FD'
          Comment: ''
               ID: 1
         Extended: 0
            J1939: []
           Length: 48
              DLC: 14
              BRS: 1
          Signals: {2×1 cell}
       SignalInfo: [2×1 struct]
          TxNodes: {0×1 cell}
       Attributes: {2×1 cell}
    AttributeInfo: [2×1 struct]
```

```
db.MessageInfo(1).Name
```

```
    'CANFDMessage'
```

Data Types: `struct`

`Attributes` — Attribute names from CAN database
cell

This property is read-only.

The `Attributes` property stores the names of all attributes defined in the specified CAN database, as a cell array of character vectors.

Use indexing to access the information of each attribute. For example:

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

```
3×1 cell array

    {'BusType'        }
    {'DatabaseVersion'}
    {'ProtocolType'   }
```

```
db.Attributes{1}
```

```
    'BusType'
```

Data Types: `cell`

`AttributeInfo` — Information on CAN database attributes
struct

This property is read-only.

The `Attributeinfo` property is a structure with information about all attributes defined in the specified CAN database.

Use indexing to access the information of each attribute.

```
db  = canDatabase('CANex.dbc');
db.AttributeInfo
```

```
3×1 struct array with fields:

    Name
    ObjectType
    DataType
    DefaultValue
    Value
```

```
db.AttributeInfo(1).Name
```

```
    'BusType'
```

Data Types: `struct`

`UserData` — Custom data
any data

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.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

Functions
`canDatabase` | `attachDatabase` | `canMessage` | `j1939Channel` | `j1939ParameterGroup`

Introduced in R2009a

j1939.Channel Properties

Properties of the `j1939.Channel` object

Description

Use the following properties to examine or configure J1939 channel settings. Use `j1939Channel` to create a channel.

Properties

Device Information

`DeviceVendor` — Device vendor name
char vector

This property is read-only.

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

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `char`

`Device` — Channel device type
char vector

This property is read-only.

For National Instruments devices, the `Device` property displays the device number on the hardware.

For all other vendors, the `Device` property displays information about the device type to which the CAN or J1939 channel is connected.

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `char`

`DeviceChannelIndex` — Device channel index
double

This property is read-only.

The `DeviceChannelIndex` property indicates the channel index on which the specified CAN or J1939 channel is configured.

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `double`

`DeviceSerialNumber` — Device serial number
double | char

This property is read-only.

The `DeviceSerialNumber` property displays the serial number of the device connected to the CAN or J1939 channel.

Values are automatically defined when you configure the channel with the `canChannel` or `j1939Channel` function.

Data Types: `double` | `char`

Data Details

`ParameterGroupsAvailable` — Number of parameter groups available to be received
double

This property is read-only.

The `ParameterGroupsAvailable` property displays the total number of parameter groups available to be received by the channel.

Data Types: `double`

`ParameterGroupsReceived` — Number of parameter groups received by channel
double

This property is read-only.

The `ParameterGroupsReceived` property indicates the total number of parameter groups received since the channel was last started.

Data Types: `double`

`ParameterGroupsTransmitted` — Number of parameter groups transmitted by channel
double

This property is read-only.

The `ParameterGroupsTransmitted` property indicates the total number of parameter groups transmitted since the channel was last started.

Data Types: `double`

`FilterPassList` — List of parameter groups to pass
char | cell

This property is read-only.

`FilterPassList` displays a list of parameter group names and numbers that the channel can pass to the network. 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: `filterAllowOnly`, `filterAllowAll`, or `filterBlockAll`

Data Types: `char` | `cell`

FilterBlockList — List of parameter groups to block
char | cell

This property is read-only.

FilterBlockList displays a list of parameter group names and numbers blocked by the channel. 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.

To change the values, use one of the filtering functions: filterAllowOnly, filterAllowAll, or filterBlockAll

Data Types: char | cell

Channel Information

Running — Indicate running status of channel
false (0) | true (1)

This property is read-only.

The Running property indicates the state of the CAN or J1939 channel, according to the following values:

- false (default) — The channel is offline.
- true — The channel is online.

Use the start function to set your channel online.

Data Types: logical

BusStatus — Status of bus
char

This property is read-only.

The BusStatus property displays information about the state of the CAN bus or the J1939 bus.

- 'N/A' — Property not supported by vendor.
- 'ErrorActive' — Node transmits Active Error Flags when it detects errors. Note: This status does not necessarily indicate that an error actually exists, but indicates how an error is handled.
- 'ErrorPassive' — Node transmits Passive Error Flags when it detects errors.
- 'BusOff' — Node will not transmit anything on the bus.

Data Types: char

InitializationAccess — Indicate control of device channel
true (1) | false (0)

This property is read-only.

The InitializationAccess property indicates if the configured CAN or J1939 channel object has full control of the device channel, according to the following values:

- true — Has full control of the hardware channel and can change the property values.

- `false` — Does not have full control and cannot change property values.

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.

Data Types: `logical`

`InitialTimestamp` — Indicate when channel started
datetime

This property is read-only.

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.

Data Types: `datetime`

`SilentMode` — Specify if channel is active or silent
`false` (default) | `true`

Specify whether the channel operates silently, according to the following values:

- `false` (default) — The channel is in normal or active mode. 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.
- `true` — The channel is in silent mode. You can observe all message activity on the network and perform analysis without affecting the network state or behavior. In this mode, you can only receive messages and not transmit any.

Data Types: `logical`

`TransceiverName` — Name of device transceiver
char

This property is read-only.

`TransceiverName` indicates 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.

Data Types: `char`

`TransceiverState` — Specify state or mode of transceiver
numeric

If your CAN or J1939 transceiver allows you to control its mode, you can use the `TransceiverState` property to set the mode.

The numeric property value for each mode is 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`

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

BusSpeed — Bit rate of bus transmission
double

This property is read-only.

The `BusSpeed` property indicates the speed at which messages are transmitted in bits per second. The default value is assigned by the vendor driver.

You can set `BusSpeed` to a supported bit rate using the `configBusSpeed` function, specifying the channel name and the bit rate value as input parameters. For example, to change the bus speed of the CAN channel object `canch` to 250,000 bits per second, and view the result, type

```
configBusSpeed(canch,250000);
bs = canch.BusSpeed
```

Data Types: `double`

SJW — Synchronization jump width (SJW) of bit time segment
double

This property is read-only.

`SJW` displays the synchronization jump width of the bit time segment. To adjust the on-chip bus clock, the controller can 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.

Data Types: `double`

TSEG1,TSEG2 — Allowed number of bits segments to lengthen and shorten sample time
double

This property is read-only.

The `TSEG1` and `TSEG2` properties indicate the amount in bit time segments that the channel can lengthen and shorten the sample time, respectively, to resynchronize or compensate for delay times in the network. The value is inherited when you configure the bus speed of your CAN channel.

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

Data Types: `double`

`NumOfSamples` — Number of samples available to channel
double

This property is read-only.

The `NumOfSamples` property is a bit timing parameter that indicates the number of bit samples performed for a single bit read on the network. The value is a positive integer based on the driver settings for the channel.

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

Data Types: `double`

Other Information

`UserData` — Custom data
any data

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.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

Functions
`j1939Channel` | `j1939ParameterGroup` | `filterAllowOnly` | `filterAllowAll` | `filterBlockAll` | `configBusSpeed` | `receive` | `transmit`

Properties
j1939.ParameterGroup Properties

Topics
“J1939 Communication”

Introduced in R2015b

j1939.ParameterGroup Properties

Properties of the `j1939.ParameterGroup` object

Description

Use the following properties to examine or configure J1939 parameter group settings. Use `j1939ParameterGroup` to create a parameter group object.

Properties

Protocol Data Unit Details

`Name` — J1939 parameter group name
char

This property is read-only.

The `Name` property displays the name of the J1939 parameter group as a character vector. This value is acquired from the name you define when you create the parameter group.

Data Types: `char`

`PGN` — J1939 parameter group number
uint32

This property is read-only.

The `PGN` property displays the number of the parameter group as a uint32 value. This value is automatically assigned when you create the parameter group.

Data Types: `uint32`

`Priority` — Priority of parameter group
numeric

The `Priority` property specifies the precedence of the parameter group on the J1939 network. `Priority` takes a numeric value of 0 (highest priority) to 7 (lowest priority), which specifies the order of importance of the parameter group.

Data Types: `uint32`

`PDUFormatType` — J1939 parameter group PDU format
char

This property is read-only.

The `PDUFormatType` property displays the J1939 protocol data unit format of the parameter group, as a character vector. This value is automatically assigned when you create the parameter group.

Data Types: `char`

`SourceAddress` — Address of parameter group source
numeric

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.

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.

Data Types: `uint32`

`DestinationAddress` — Address of parameter group destination

numeric

Address of the J1939 parameter group destination. `DestinationAddress` identifies the parameter group destination on the J1939 network. The source uses the specified destination address to send parameter groups.

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.

Data Types: `uint32`

Data Details

`Timestamp` — Time when parameter group received

double

This property is read-only.

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.

Data Types: `double`

`Data` — CAN message raw data

uint8 array

Use the `Data` property to view or define the raw data in a J1939 parameter group. The data is an array of uint8 values.

For example, create a parameter group and specify data:

```
pg = j1939ParameterGroup(db,'PackedData')
pg.Data(1:2) = [50 0]
```

Data Types: `uint8`

`Signals` — Physical signals defined in parameter group

struct

The `Signals` property allows you to view and edit decoded signal values defined for a parameter group. The input values for this property depend on the signal type.

For example, create a parameter group.

```
pg = j1939ParameterGroup(db,'PackedData')
```

Display the 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
```

Data Types: `struct`

Other Information

`UserData` — Custom data
any data

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.

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

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

See Also

Functions
`canDatabase` | `j1939Channel` | `j1939ParameterGroup`

Properties
j1939.Channel Properties

Topics
"J1939 Communication"

External Websites
J1939 Standards Overview

Introduced in R2015b

xcp.A2L Properties

Properties of the `xcp.A2L` file object

Description

Use the following properties to examine `xcp.A2L` file object settings. Use `xcpA2L` to create an A2L-file object.

Properties

xcp.A2L

`FileName` — Name of referenced A2L file
char

The `FileName` property displays the name of the referenced A2L file as a character vector.

Data Types: `char`

`FilePath` — Path of A2L file
char

The `FilePath` property displays the full file path to the A2L file, including the A2L-file name, as a character vector.

Data Types: `char`

`ServerName` — Name of connected server
char

The `ServerName` property displays the name of the server node as specified in the A2L file, as a character vector.

Data Types: `char`

`Warnings` — Warnings from A2L file generation
string

Stores warnings thrown by the A2L file parser.

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

```
ans = 

  0×0 empty string array
```

Data Types: `string`

`Events` — Event names
cell

Event names, returned as a cell array of character vectors. For example:

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

```
ans =

  1×6 cell array

    {'Key T'}    {'10 ms'}    {'100ms'}    {'1ms'}    {'FilterBypassDaq'}    {'FilterBypassSt'}
```

Data Types: `cell`

`EventInfo` — Event information
array of `xcp.Event` object

Event information, returned as an array of `xcp.Event` objects. For example:

```
a2lfile = xcpA2L('XCPSIM.a2l');
ei = a2lfile.EventInfo(1)
```

```
ei =

  Event with properties:

                       Name: 'Key T'
                  Direction: 'DAQ'
                 MaxDAQList: 255
              ChannelNumber: 0
           ChannelTimeCycle: 0
            ChannelTimeUnit: 6
            ChannelPriority: 0
    ChannelTimeCycleInSeconds: 0
```

Data Types: `xcp.Event`

`Measurements` — Measurement names
cell

Measurement names, returned as a cell array of character vectors. For example:

```
a2lfile = xcpA2L('XCPSIM.a2l');
a2lfile.Measurements(10:15)
```

```
ans =

  1×6 cell array

    {'FW1'}    {'KL1Output'}    {'MaxChannel1'}    {'MinChannel1'}    {'PWM'}    {'PWMFiltered'}
```

Data Types: `cell`

`MeasurementInfo` — Measurement information
`containers.Map` object

Measurement information, returned as a Map object. For example:

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

```
mi =

  Map with properties:
```

```
        Count: 45
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

`Characteristics` — Names of characteristics
cell

Names of characteristics, returned as a cell array of character vectors. For example:

```
a2lfile = xcpA2L('XCPSIM.a2l');
a2lfile.Characteristics(10:15)
```

```
ans =

  1×6 cell array

    {'a0'}    {'b0'}    {'c0'}    {'map1'}    {'map1Counter'}    {'map4_80_uc'}
```

Data Types: `cell`

`CharacteristicInfo` — Characteristic information
`containers.Map` object

Characteristic information, returned as a `Map` object. For example:

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

```
ci =

  Map with properties:

        Count: 16
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

`AxisInfo` — Axis information
`containers.Map` object

Axis information, returned as a `Map` object. For example:

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

```
ai =

  Map with properties:

        Count: 1
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

`RecordLayouts` — Container for characteristic objects
`containers.Map` object

Container for characteristic objects, returned as a `containers.Map` object. For example:

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

rl = 

  Map with properties:

        Count: 41
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

CompuMethods — Container for computation method objects
`containers.Map` object

Container for computation method objects, returned as a `containers.Map` object. For example:

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

cm = 

  Map with properties:

        Count: 16
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

CompuTabs — Container for ComputationTAB method objects
`containers.Map` object

Container for ComputationTAB (conversion table) method objects used for `interp`, returned as a `containers.Map` object. For example:

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

ct = 

  Map with properties:

        Count: 0
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

CompuVTabs — Container for ComputationVTAB method objects
`containers.Map` object

Container for ComputationVTAB (verbal conversion table) method objects used for `enum`, returned as a `containers.Map` object. For example:

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

cvt = 

  Map with properties:

        Count: 2
      KeyType: char
    ValueType: any
```

Data Types: `containers.Map`

`ProtocolLayerInfo` — Protocol layer information
`xcp.ProtocolLayerInfo` object

The `ProtocolLayerInfo` property displays an `xcp.ProtocolLayerInfo` object containing general information about the XCP protocol implementation of the server as defined in the A2L file. For example:

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

pli = 

  ProtocolLayerInfo with properties:

    AddressGranularity: 'ADDRESS_GRANULARITY_BYTE'
             ByteOrder: 'BYTE_ORDER_MSB_LAST'
                MaxCTO: 8
                MaxDTO: 8
                    T1: 1000
                    T2: 200
                    T3: 0
                    T4: 0
                    T5: 0
                    T6: 0
                    T7: 0
```

Data Types: `xcp.ProtocolLayerInfo`

`DAQInfo` — DAQ related information
`xcp.DAQInfo` object

DAQ related information, returned as a `DAQInfo` object. For example:

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

di = 

  DAQInfo with properties:

              AddressExtension: 'ADDRESS_EXTENSION_FREE'
                    ConfigType: 'DYNAMIC'
    GranularityODTEntrySizeDAQ: 'GRANULARITY_ODT_ENTRY_SIZE_DAQ_BYTE'
       IdentificationFieldType: 'IDENTIFICATION_FIELD_TYPE_ABSOLUTE'
                        MaxDAQ: 0
```

```
            MaxEventChannels: 6
          MaxODTEntrySizeDAQ: 7
                      MinDAQ: 0
            OptimizationType: 'OPTIMISATION_TYPE_DEFAULT'
          OverloadIndication: 'OVERLOAD_INDICATION_PID'
                        STIM: [1×1 struct]
          PrescalerSupported: 'PRESCALER_SUPPORTED'
             ResumeSupported: 'RESUME_NOT_SUPPORTED'
                   Timestamp: [1×1 struct]
```

Data Types: `xcp.DAQInfo`

`TransportLayerCANInfo` — CAN specific transport layer information
`xcp.XCPonCAN` object

CAN specific transport layer information, returned as an `XCPonCAN` object. For example,

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

```
tlci = 

  XCPonCAN with properties:
           CommonParameters: [1×1 xcp.a2l.CommonParameters]
     TransportLayerInstance: ''
             CANIDBroadcast: []
                CANIDClient: 1
      CANIDClientIsExtended: 0
                CANIDServer: 2
      CANIDServerIsExtended: 0
                   BaudRate: 500000
                SamplePoint: 62
                 SampleRate: SINGLE
                  BTLCycles: 8
                        SJW: 1
                   SyncEdge: SINGLE
             MaxDLCRequired: []
                 MaxBusLoad: []
    MeasurementSplitAllowed: []
                      CANFD: [1×0 xcp.a2l.CANFD]
           OptionalTLSubCmd: [0×0 xcp.a2l.OptionalCANTLSubCmd]
```

Data Types: `xcp.XCPonCAN`

`TransportLayerUDPInfo` — UDP transport layer information
`xcp.XCPonIP` object

UDP transport layer information, returned as an `XCPonIP` object. For example:

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

```
tlui = 

  XCPonIP with properties:
          CommonParameters: [1×1 xcp.a2l.CommonParameters]
    TransportLayerInstance: ''
                      Port: 5555
```

```
                 Address: 2.1307e+09
           AddressString: '127.0.0.1'
```

Data Types: `xcp.XCPonIP`

`TransportLayerTCPInfo` — TCP transport layer information
`xcp.XCPonIP` object

TCP transport layer information, returned as a `XCPonIP` object.

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

```
tlti = 

  0×0 XCPonIP array with properties:

    CommonParameters
    TransportLayerInstance
    Port
    Address
    AddressString
```

Data Types: `xcp.XCPonIP`

See Also

Functions
`xcpA2L` | `getCharacteristicInfo` | `getMeasurementInfo` | `getEventInfo`

Properties
xcp.Channel Properties

Topics
"XCP Communication"
"Communication in MATLAB"

External Websites
ASAM MCD-2 MC Technical Content

Introduced in R2013a

xcp.Channel Properties

Properties of the `xcp.Channel` object

Description

Use the following properties to examine or configure `xcp.Channel` object settings. Use `xcpChannel` to create an XCP channel object.

Properties

xcp.Channel

`ServerName` — Name of connected server
char

This property is read-only.

Name of the server node as specified in the A2L file, returned as a character vector. For example:

```
xcpch = xcpChannel(a2lfile,'CAN','Vector','Virtual 1',1);
sn = xcpch.ServerName
```

```
sn =

    'CPP'
```

Data Types: `char`

`A2LFileName` — Name of referenced A2L file
char

This property is read-only.

Name of the referenced A2L file, returned as a character vector.

Data Types: `char`

`TransportLayer` — Type of transport layer used for XCP connection
char

This property is read-only.

Type of transport layer used for XCP connection, returned as a character vector. For example:

```
xcpch = xcpChannel(a2lfile,'CAN','Vector','Virtual 1',1);
tl = xcpch.TransportLayer
```

```
tl =

    'CAN'
```

Data Types: `char`

TransportLayerDevice — XCP transport layer connection details
struct

This property is read-only.

XCP transport layer connection details, including information about the device through which the channel communicates with the server, returned as a structure. For example:

```
xcpch = xcpChannel(a2lfile,'CAN','Vector','Virtual 1',1);
tld = xcpch.TransportLayerDevice
```

```
tld =

  struct with fields:

          Vendor: 'Vector'
          Device: 'Virtual 1'
    ChannelIndex: 1
```

Data Types: struct

SeedKeyDLL — DLL-file containing seed and key access algorithm
char

The SeedKeyDLL property indicates the name of the DLL-file that contains the seed and key security algorithm used to unlock an XCP server module. The file defines the algorithm for generating the access key from a given seed according to ASAM standard definitions. For information on the file format and API, see the Vector web page Steps to Use Seed&Key Option in CANape or "Seed and Key Algorithm" in National Instruments CAN ECU Measurement and Calibration Toolkit User Manual.
Note: The DLL must be the same bitness as MATLAB (64-bit).

Data Types: char

See Also

Functions
xcpA2L | xcpChannel

Properties
xcp.A2L Properties

Topics
“XCP Communication”
“Communication in MATLAB”

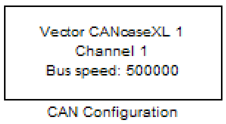
Introduced in R2013a

Blocks

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 requires a corresponding CAN Configuration block for 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.

Parameters

`Device` — CAN device and channel

list option

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

Programmatic Use

Block Parameter: `Device`

Type: character vector, string

`Bus speed` — CAN bus bit rate

integer

Set the `BusSpeed` property for the selected device, in bits per second. The default bus speed is the default assigned by the selected device.

Programmatic Use

Block Parameter: `BusSpeed`

Type: character vector, string

Values: integer

`Enable bit parameters manually` — Allow specifying individual bit parameters
off (default) | on

Note This option is available only for supporting vendors.

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**. For more information on these parameters, see Bit Timing. 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.

Programmatic Use
Block Parameter: `EnableBitParameters`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Synchronization jump width` — Maximum allowed time adjustment
integer

Specify the maximum limit of bit time adjustment in the case of resynchronization. The specified value must be a positive integer indicating a number of bit time quanta segments. 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.

Programmatic Use
Block Parameter: `SJW`
Type: character vector, string
Values: integer

`Time segment 1` — Number of time quanta before sample
integer

Specify the number of bit time quanta before the sampling point. The specified value must be a positive integer. Typically, an adjustment of this value is made with a corresponding inverse adjustment to **Time segment 2** so that their sum remains constant. 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.

Programmatic Use
Block Parameter: `TSEG1`
Type: character vector, string
Values: integer

`Time segment 2` — Number of time quanta after sample
integer

Specify the number of bit time quanta after the sampling point. The specified value must be a positive integer. Typically, an adjustment of this value is made with a corresponding inverse adjustment to

Time segment 1 so that their sum remains constant. 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.

Programmatic Use
Block Parameter: `TSEG2`
Type: character vector, string
Values: integer

`Number of samples` — Samples per bit
integer

Specify the number of samples per bit. 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.

Programmatic Use
Block Parameter: `NSamples`
Type: character vector, string
Values: integer

`Verify bit parameter settings validity` — Check validity of settings

If you have set the bit parameter settings manually, click this button to see if your settings are valid. The block runs a check to see if the combination of your bus speed and bit parameter values form a valid combination for the CAN device. If the current combination is not valid, the verification fails and displays an error message. This button is active only when the **Enable bit parameters manually** check box is selected.

Programmatic Use

None

`Acknowledge mode` — Control channel activity on CAN bus
`Normal` (default) | `Silent`

Specify whether the channel is in `Normal` or `Silent` mode. By default **Acknowledge mode** is `Normal`. In this mode, the channel can receive and transmit messages normally, and perform 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, the channel can only receive messages and not transmit.

Programmatic Use
Block Parameter: `AckMode`
Type: character vector, string
Values: `'Normal'` | `'Silent'`
Default: `'Normal'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

When used with the CAN Receive and CAN Transmit blocks, the CAN Configuration block supports code generation, but with limited portability that runs only on the host computer.

See Also

Blocks
CAN Receive | CAN Transmit

Properties
can.Channel Properties

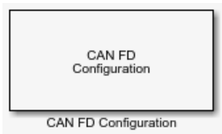
External Websites
Bit Timing

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.

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).

Parameters

`Device` — CAN device and channel

list option

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.

Programmatic Use

Block Parameter: `Device`

Type: character vector, string

`Arbitration Bus speed` — Arbitration bit rate

numeric

Set arbitration bus speed for the selected device, in bits per second. The default speed is assigned by the selected device.

Programmatic Use

Block Parameter: `ArbitrationBusSpeed`

Type: character vector, string
Values: integer

`Data Bus speed` — Data bit rate
numeric

Set data bus speed for the selected device, in bits per second. The default speed is assigned by the selected device.

Programmatic Use
Block Parameter: `DataBusSpeed`
Type: character vector, string
Values: integer

`Bus frequency` — PEAK-System bus rate
numeric

(PEAK-System only.) Set the bus frequency, in megahertz.

Programmatic Use
Block Parameter: `BusFrequency`
Type: character vector, string
Values: numeric

`Arbitration/Data bit rate prescaler` — Bit rate prescaler
integer

(PEAK-System only.) Set separate bit rate prescaler values for arbitration and data bit rates.

Programmatic Use
Block Parameter: `ArbitrationPrescaler`, `DataPrescaler`
Type: character vector, string
Values: integer

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. For more information on these parameters, see Bit Timing.

`Synchronization jump width` — Maximum allowed time adjustment
integer

Specify the maximum limit of bit time adjustment in the case of resynchronization. The specified value must be a positive integer indicating a number of bit time quanta segments. If you do not specify a value, the selected bus speed setting determines the default value.

Programmatic Use
Block Parameter: `ArbitrationSJW`, `DataSJW`
Type: character vector, string
Values: integer

`Time segment 1` — Number of time quanta before sample
integer

Specify the number of bit time quanta before the sampling point. The specified value must be a positive integer. Typically, an adjustment of this value is made with a corresponding inverse adjustment to **Time segment 2** so that their sum remains constant. If you do not specify a value, the selected bus speed setting determines the default value.

Programmatic Use
Block Parameter: `ArbitrationTSEG1`, `DataTSEG1`
Type: character vector, string
Values: integer

`Time segment 2` — Number of time quanta after sample
integer

Specify the number of bit time quanta after the sampling point. The specified value must be a positive integer. Typically, an adjustment of this value is made with a corresponding inverse adjustment to **Time segment 1** so that their sum remains constant. If you do not specify a value, the selected bus speed setting determines the default value.

Programmatic Use
Block Parameter: `ArbitrationTSEG2`, `DataTSEG2`
Type: character vector, string
Values: integer

`Verify bit parameter settings validity` — Check validity of settings

If you have altered the bit parameter settings, 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.

Programmatic Use

None

`Acknowledge mode` — Control channel activity on CAN bus
`Normal` (default) | `Silent`

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.

Programmatic Use
Block Parameter: `AckMode`
Type: character vector, string
Values: `'Normal'` | `'Silent'`
Default: `'Normal'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

When used with the CAN FD Receive and CAN FD Transmit blocks, the CAN FD Configuration block supports code generation, but with limited portability that runs only on the host computer.

See Also

Blocks
CAN FD Pack | CAN FD Unpack | CAN FD Receive | CAN FD Transmit

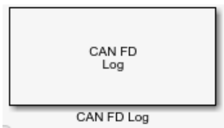
Properties
can.Channel Properties

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 block 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” on page 14-73.

The CAN FD 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.

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`.

Ports

Input

`CAN Msg` — CAN FD messages to log
`CAN_FD_MESSAGE_BUS`

The **CAN Msg** input port is available when the **Log messages from** parameter is set to `Input port`. Provide an input from another block as a Simulink signal bus of type `CAN_FD_MESSAGE_BUS`.

Data Types: `CAN_FD_MESSAGE_BUS`

Parameters

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

`File name` — Log file location and name
`untitled.mat` (default) | file name

Enter the path and name of the MAT-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 format `YYYY-MMM-DD_hhmmss`. Specify a unique name to differentiate between your files for repeated logging.

`Variable name` — Variable name for CAN FD messages in log file
`ans` (default) | variable name

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

Maximum number of messages to log — Limit quantity of messages
10000 (default) | numeric

Specify the maximum number of messages this block can log from the selected device or port. The specified value must be a positive integer. The default value is 10000 messages. The log file saves the most recent messages up to the specified maximum number.

Log messages from — Source of messages
CAN FD Bus (default) | Input port

Select the source of the messages logged by the block. To log messages from the CAN FD bus network, select CAN FD Bus, then specify a **Device**. To log messages from another block in the model, select Input port, which adds an inport port to the block.

Device — CAN device and channel
list option

Select the device on the CAN FD network that you want to log messages from. This field is available only if you select CAN FD Bus for the **Log messages from** parameter.

Sample time — Block sampling time in simulation
0.01 (default) | numeric

Specify the sampling time of the block during simulation. 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 the sample time. You can also specify a MATLAB variable for sample time. The default value is 0.01 simulation seconds. For more information, see “Timing in Hardware Interface Models” on page 8-21.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-10.

See Also

Blocks
CAN FD Configuration | CAN FD Replay

Functions
canFDMessage | canFDMessageBusType

Introduced in R2018b

CAN FD Pack

Pack individual signals into message for CAN FD bus

Library: Vehicle Network Toolbox / CAN FD Communication
Embedded Coder Support Package for Texas Instruments C2000 Processors / Target Communication
Simulink Real-Time / CAN / CAN-FD MSG blocks

Description

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

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

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 “Design Your Model for Effective Acceleration” (Simulink).

Tip

- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block. See “J1939 Communication”.

Ports

Input

`Data` — CAN FD message signal input
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

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 that you specify for the block. For example, if your message has four signals, the block can have four input ports.

Code generation to deploy models to targets. Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

Output

`Msg` — CAN FD message output
`CAN_FD_MESSAGE_BUS`

This block has one output port, Msg. The CAN FD Pack block takes the specified input signals and packs them into a CAN FD message, output as a Simulink `CAN_FD_MESSAGE_BUS` signal. For more information on Simulink bus objects, see "Composite Signals" (Simulink).

Parameters

`Data input as` — Select your data signal

`raw data` (default) | `manually specified signals` | `CANdb specified signals`

- `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.

  The conversion formula is:

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

  where `physical_value` is the original value of the signal and `raw_value` is the packed signal value.
- `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.

Programmatic Use
Block Parameter: `DataFormat`
Type: `string` | `character vector`
Values: `'raw data'` | `'manually specified signals'` | `'CANdb specified signals'`
Default: `'raw data'`

`CANdb file` — CAN database file

character vector

This option is available if you specify that your data is input through 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. File names that contain non-alphanumeric characters such as equal signs, ampersands, and so on are not valid CAN database file names. You can use periods in your database name. Before you use the CAN database files, rename them with non-alphanumeric characters.

Programmatic Use
Block Parameter: `CANdbFile`
Type: `string` | `character vector`

`Message list` — CAN message list

array of character vectors

This option is available if you specify that your data is input through 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.

Programmatic Use
Block Parameter: `MsgList`
Type: `string` | `character vector`

`Name` — CAN FD message name
`CAN Msg` (default) | character vector

Specify a name for your CAN FD message. The default is `CAN Msg`. This option is available if you choose to input raw data or manually specify signals. This option is not available if you choose to use signals from a CANdb file.

Programmatic Use
Block Parameter: `MsgName`
Type: `string` | `character vector`

`Protocol mode` — CAN FD message protocol
`CAN FD` (default) | `CAN`

Specify the message protocol mode.

Programmatic Use
Block Parameter: `ProtocolMode`
Type: `string` | `character vector`
Values: `'CAN FD'` | `'CAN'`
Default: `'CAN FD'`

`Identifier type` — CAN identifier type
`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

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.

Programmatic Use
Block Parameter: `MsgIDType`
Type: `string` | `character vector`
Values: `'Standard (11-bit identifier)'` | `'Extended (29-bit identifier)'`
Default: `'Standard (11-bit identifier)'`

`Identifier` — Message identifier
`0` (default) | `0 .. 536870911`

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 by using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`
Type: `string` | `character vector`
Values: `'0'` to `'536870911'`

`Length (bytes)` — CAN FD message length
`8` (default) | `0` to `64`

Specify the length of your message. For CAN messages the value can be 0 to 8 bytes; for CAN FD the value can be 0 to 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.

Programmatic Use
Block Parameter: `MsgLength`
Type: `string` | `character vector`
Values: `'0'` to `'8'`, `'12'`, `'16'`, `'20'`, `'24'`, `'32'`, `'48'`, `'64'`
Default: `'8'`

`Remote frame` — CAN message as remote frame
`off` (default) | `on`

(Disabled for CAN FD protocol mode.) Specify the CAN message as a remote frame.

Programmatic Use
Block Parameter: `Remote`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Bit Rate Switch (BRS)` — Enable bit rate switch
`off` (default) | `on`

(Disabled for CAN protocol mode.) Enable bit rate switch.

Programmatic Use
Block Parameter: `BRSSwitch`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Add signal` — Add CAN FD signal

Add a signal to the signal table.

Programmatic Use

None

`Delete signal` — Remove CAN FD signal

Remove the selected signal from the signal table.

Programmatic Use

None

`Signals` — Signals table
table

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

If you are using a CANdb file, the data in the file populates this table 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 in this table. Each signal that you create has these 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.)

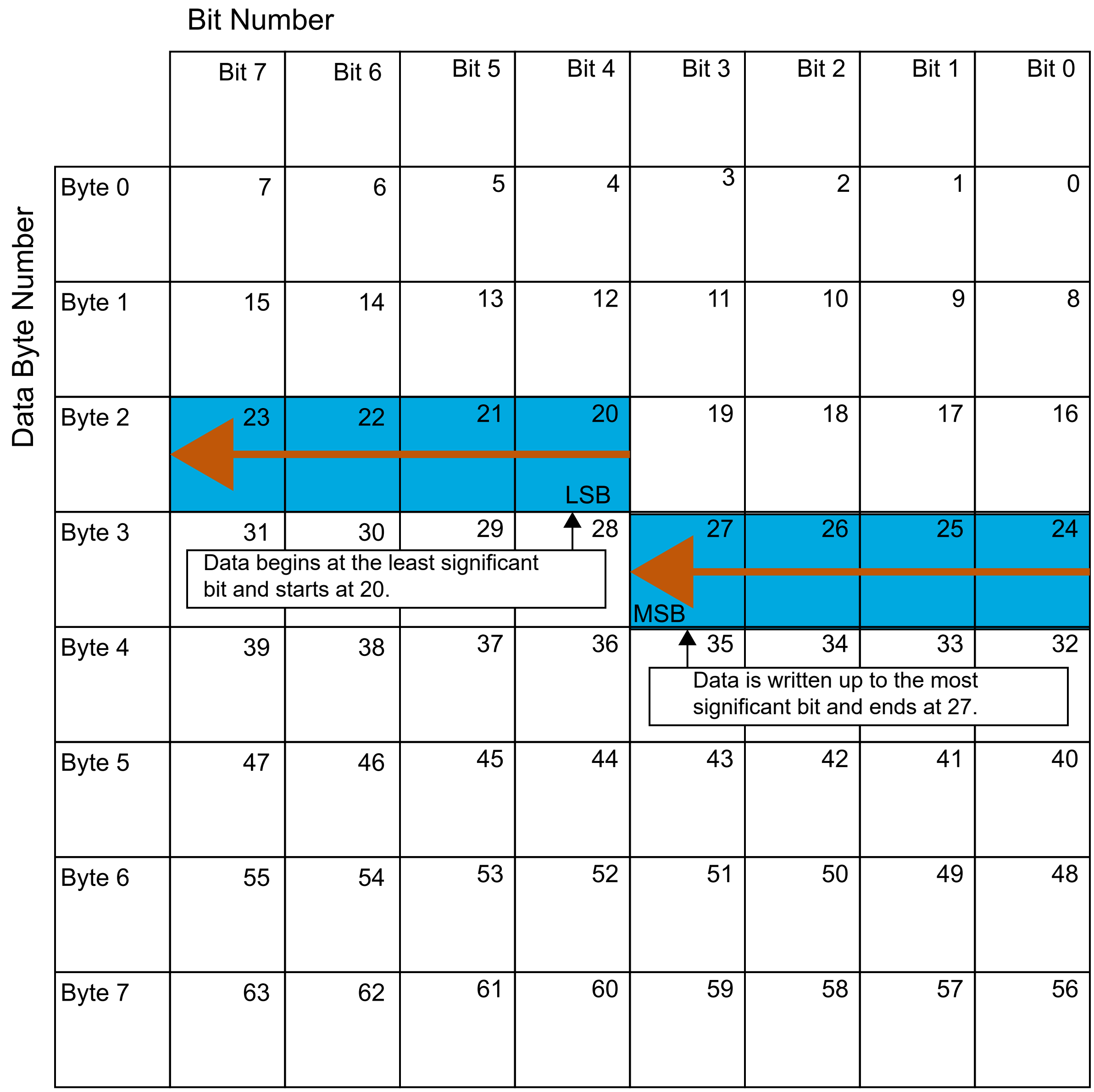
Length (bits)

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

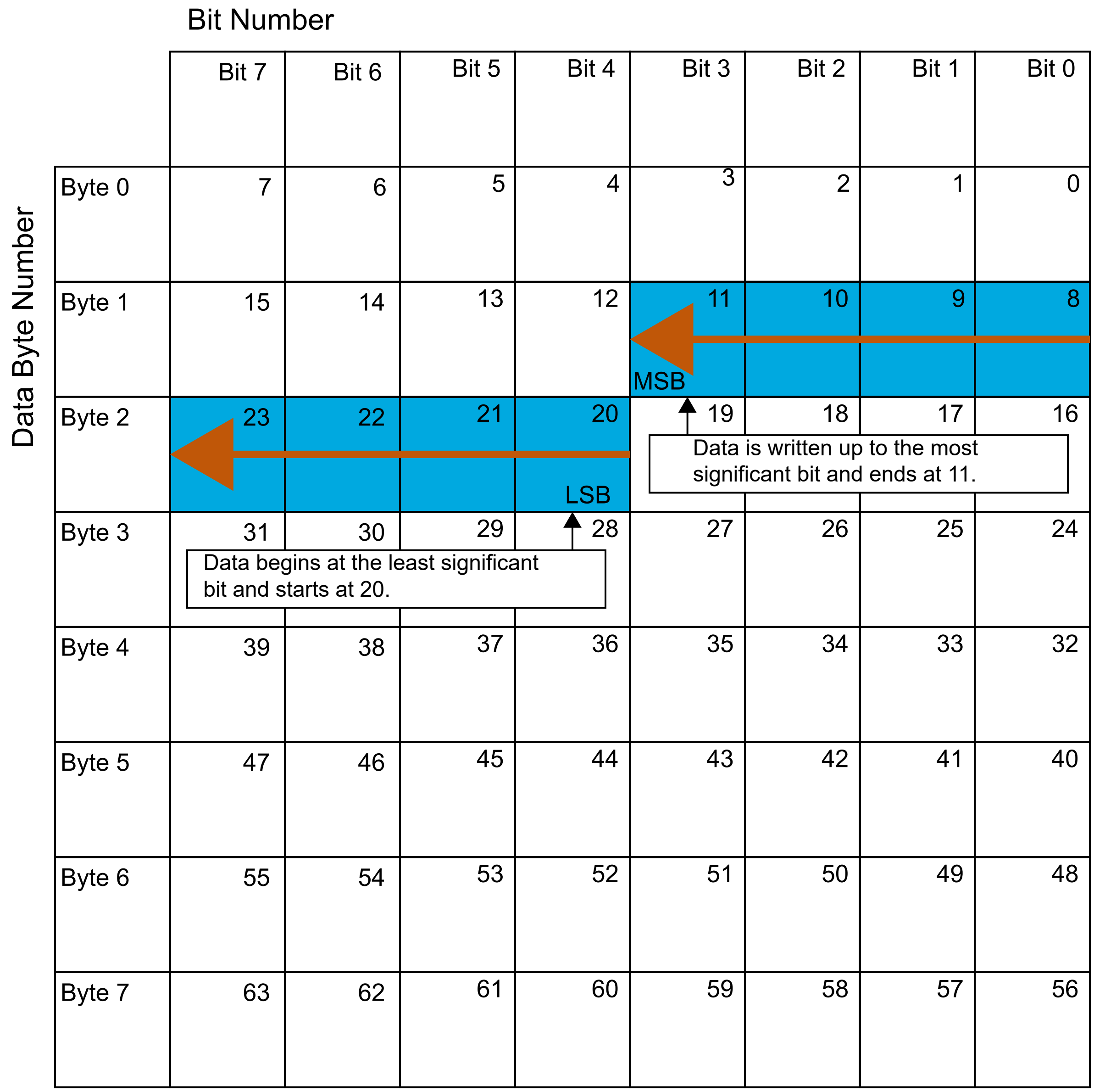
Select either of these options:

- `LE`: Where the byte order is in little-endian format (Intel®). In this format you count bits from the least significant bit, to the most significant bit. 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.



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 least-significant bit to the most-significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



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

Data type

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

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

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 time step:

- `Standard`: The signal is packed at each time step.
- `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 these types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

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

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 the **Data input as** parameter conversion formula 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 the **Data input as** parameter conversion formula 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.

Programmatic Use
Block Parameter: `SignalInfo`
Type: `string` | `character vector`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
CAN FD Unpack | CAN FD Configuration | CAN FD Transmit | CAN Pack

Functions
`canFDMessageBusType`

Topics
"Design Your Model for Effective Acceleration" (Simulink)
"Composite Signals" (Simulink)

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 the CAN messages 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.

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`.

Ports

Output

`CAN Msg` — Received CAN messages
CAN_FD_MESSAGE_BUS

The `CAN Msg` output port contains one or more packed CAN messages received at that particular timestep, output as a `CAN_FD_MESSAGE_BUS`. The output includes either one or all messages for that timestep, depending on the setting of **Number of messages received at each timestep**.

Data Types: `CAN_FD_MESSAGE_BUS`

`f()` — Function-call event output
function-call event

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.

Data Types: `function-call event`

Parameters

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

`Device` — CAN device and channel
list option

Select from the list 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.

Programmatic Use
Block Parameter: `Device`
Type: character vector, string

`Standard IDs Filter` — Limit or allow messages based on standard ID
`Allow all` (default) | `Allow only` | `Block all`

Select the filter for standard IDs. Choices are:

- `Allow all` (default): Allows all standard IDs to pass the filter.
- `Allow only`: Allows only the ID or range of IDs specified in the text field, specified as a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to allow IDs 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.

Programmatic Use
Block Parameter: `StdIDsCombo`
Type: character vector, string
Values: `'Allow all'` | `'Allow only'` | `'Block all'`
Default: `'Allow all'`

If using `'Allow only'`, set the filter values with the following:
Block Parameter: `StandardIDs`
Type: character vector, string
Values: integer scalar or row vector

`Extended IDs Filter` — Limit or allow messages based on extended ID
`Allow all` (default) | `Allow only` | `Block all`

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

- `Allow all` (default): Allows all extended IDs to pass the filter.
- `Allow only`: Allows only those IDs specified in the text field. Allows only the ID or range of IDs specified in the text field, specified as 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 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.

Programmatic Use
Block Parameter: `ExtIDsCombo`
Type: character vector, string
Values: `'Allow all'` | `'Allow only'` | `'Block all'`
Default: `'Allow all'`
Block Parameter: `ExtendedIDs`
Type: character vector, string
Values: integer scalar or row vector

`Sample time` — Block execution rate
`0.01` (default)

Specify the sampling time of the block, which defines the rate at which the block is executed during simulation. The default value is 0.01 simulation seconds. 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. For more information, see "Timing in Hardware Interface Models" on page 8-21.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector, string
Values: double
Default: `'0.01'`

`Number of messages received at each timestep` — Receive one or all messages
`1` (default) | `all`

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.

Programmatic Use
Block Parameter: `MsgsPerTimestep`
Type: character vector, string
Values: `'1'` | `'all'`
Default: `'1'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See "Code Generation" on page 13-22.

See Also

Blocks
CAN FD Configuration | CAN FD Unpack | CAN FD Transmit

Functions
`canFDMessageBusType`

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 `canMessageReplayBlockStruct` and save the result to a separate file. For more information, see “Log and Replay CAN Messages” on page 14-73.

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.

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`.

Ports

Output

`CAN Msg` — Replayed CAN FD messages
`CAN_FD_MESSAGE_BUS`

This output port contains a packed CAN FD messages logged at that particular timestep, output as a signal bus of type `CAN_FD_MESSAGE_BUS`.

Data Types: `CAN_FD_MESSAGE_BUS`

`f()` — Function-call event output
function-call event

This port provides a trigger to a Function-Call subsystem when the block receives a new message. You can connect it to a Function-Call Subsystem to unpack and process the message.

Data Types: `function-call event`

Parameters

Tip Configure the CAN FD Configuration block in the model before you configure the CAN FD Receive block parameters.

`File name` — Path and name of MAT-file with messages
`untitled.mat` (default) | file name

Specify the path and name of the MAT-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` — Variable in MAT-file holding messages
`ans` (default) | variable

Specify the variable saved in the MAT-file that holds the CAN FD messages.

`Number of times to replay messages` — Repeat value
`Inf` (default) | integer

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 output location
CAN FD Bus (default) | `Output port`

Specify if the model is replaying messages to the CAN FD network or an output port. For a network, you must also specify a **Device**.

`Device` — CAN FD device and channel
device list option

Select the device on the CAN FD network to replay messages to. This field is unavailable if you select `Output port` for the **Replay message to** parameter.

`Sample time` — Block execution rate
`0.01` (default) | numeric

Specify the sampling time of the block during simulation. 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 the sample time. You can also specify a MATLAB variable for sample time. The default value is `0.01` simulation seconds. For more information, see “Timing in Hardware Interface Models” on page 8-21.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-27.

See Also

Blocks
CAN FD Configuration | CAN FD Log

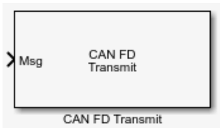
Functions
`canFDMessageBusType` | `canFDMessageReplayBlockStruct`

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.

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.

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`.

Ports

Input

`CAN Msg` — CAN FD messages to transmit
`CAN_FD_MESSAGE_BUS`

CAN FD messages to transmit, as packed by a CAN FD Pack block, input as a Simulink signal bus of type `CAN_FD_MESSAGE_BUS`.

Data Types: `CAN_FD_MESSAGE_BUS`

Parameters

Tip Configure the CAN FD Configuration block in the model before you configure the CAN FD Transmit block parameters.

`Device` — CAN FD device and channel
list option

Select the CAN device and channel for transmitting CAN FD 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.

Programmatic Use
Block Parameter: `Device`
Type: character vector, string

The following parameters define transmit options.

`On data change` — Enable event-based transmission when data changes
`off` (default) | `on`

When event-based transmission is enabled, messages are transmitted only at those time steps when a change in message data is detected. When the input data matches the most recent transmission for a given message ID, the message is not re-transmitted.

Event and periodic transmission can both be enabled to work together simultaneously. If neither is selected, the default behavior is to transmit the current input at each time step.

Programmatic Use
Block Parameter: `EnableEventTransmit`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Periodic` — Enable periodic transmission
`off` (default) | `on`

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.

Programmatic Use
Block Parameter: `EnablePerioicTransmit`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Transmit Options: Message period (in seconds)` — Period of message transmission rate
`1.000` (default) | positive numeric scalar

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.

Programmatic Use
Block Parameter: `MessagePeriod`
Type: character vector, string
Values: double
Default: `'1.000'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-30.

See Also

Blocks
CAN FD Configuration | CAN FD Pack | CAN FD Receive | Bus Creator | Vector Concatenate, Matrix Concatenate

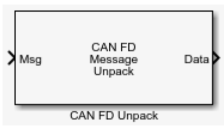
Functions
`canFDMessageBusType` | `transmitPeriodic`

Introduced in R2018a

CAN FD Unpack

Unpack individual signals from CAN FD messages

Library: Vehicle Network Toolbox / CAN FD Communication
Embedded Coder Support Package for Texas Instruments C2000 Processors / Target Communication
Simulink Real-Time / CAN / CAN-FD MSG blocks



Description

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

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

The CAN FD Unpack block supports:

- Simulink Accelerator mode. You can speed up the execution of Simulink models. For more information, see “Design Your Model for Effective Acceleration” (Simulink).

Tip

- To process every message coming through a channel, it is recommended that you use the CAN FD Unpack block in a function trigger subsystem. See “Using Triggered Subsystems” (Simulink).
- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block. See “J1939 Communication”.

Ports

Input

`Msg` — CAN FD message input
`CAN_FD_MESSAGE_BUS`

This block has one input port, Msg. The CAN FD Unpack block takes the specified input CAN messages and unpacks their signal data to separate outputs.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Code generation to deploy models to targets. Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

Output

`Data` — CAN message output
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

The CAN FD Unpack block has one output port by default. The number of data 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, labeled by signal name.

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.

Additional output ports can be added by the options in the parameters **Output ports** pane.

Parameters

`Data to output as` — Select your data signal
`raw data` (default) | `manually specify signals` | `CANdb specified signals`

- `raw data`: Output data as a uint8 vector array. If you select this option, you specify only the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.

  The conversion formula is:

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

  where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.
- `manually specified signals`: You can 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 that you specify. For example, if you specify four signals, your block has four output ports.
- `CANdb specified signals`: You can 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.

Programmatic Use
Block Parameter: `DataFormat`
Type: `string` | `character vector`
Values: `'raw data'` | `'manually specified signals'` | `'CANdb specified signals'`
Default: `'raw data'`

`CANdb file` — CAN database file
character vector

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 the **Signals** table. 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.

Programmatic Use
Block Parameter: `CANdbFile`
Type: `string` | `character vector`

`Message list` — Message list

array of character vectors

This option is available if you specify in the **Data to be output as** list that your data is to be output as a CANdb file 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.

Programmatic Use
Block Parameter: `MsgList`
Type: `string` | `character vector`

`Name` — Message name

`CAN Msg` (default) | character vector

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.

Programmatic Use
Block Parameter: `MsgName`
Type: `string` | `character vector`

`Identifier type` — Identifier type

`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

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.

Programmatic Use
Block Parameter: `MsgIDType`
Type: `string` | `character vector`
Values: `'Standard (11-bit identifier)'` | `'Extended (29-bit identifier)'`
Default: `'Standard (11-bit identifier)'`

`Identifier` — Message identifier

`0` (default) | `0 .. 536870911`

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.

Programmatic Use
Block Parameter: `MsgIdentifier`
Type: `string` | `character vector`
Values: `'0'` to `'536870911'`

`Length (bytes)` — CAN message length

8 (default) | `0 .. 8`

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.

Programmatic Use

Block Parameter: `MsgLength`
Type: `string` | `character vector`
Values: `'0'` to `'8'`, `'12'`, `'16'`, `'20'`, `'24'`, `'32'`, `'48'`, `'64'`
Default: `'8'`

`Add signal` — Add CAN signal

Add a signal to the signal table.

Programmatic Use

None

`Delete signal` — Remove CAN signal

Remove the selected signal from the signal table.

Programmatic Use

None

`Signals` — Signals table

table

If you choose to specify signals manually or define signals by using a CANdb file, this table appears.

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

If you have selected to specify signals manually, create your signals manually in this table. Each signal that you create has these 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.)

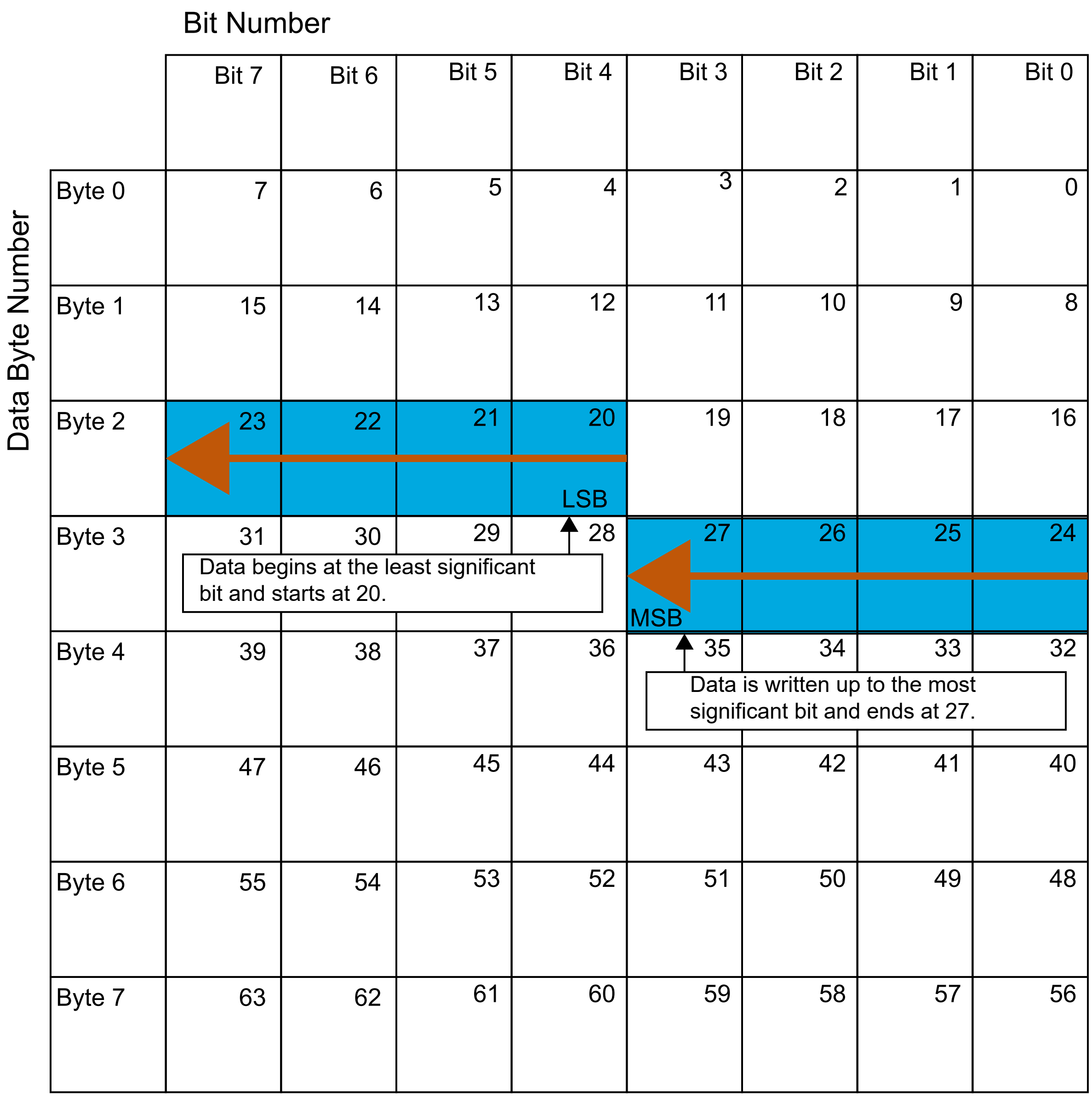
Length (bits)

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

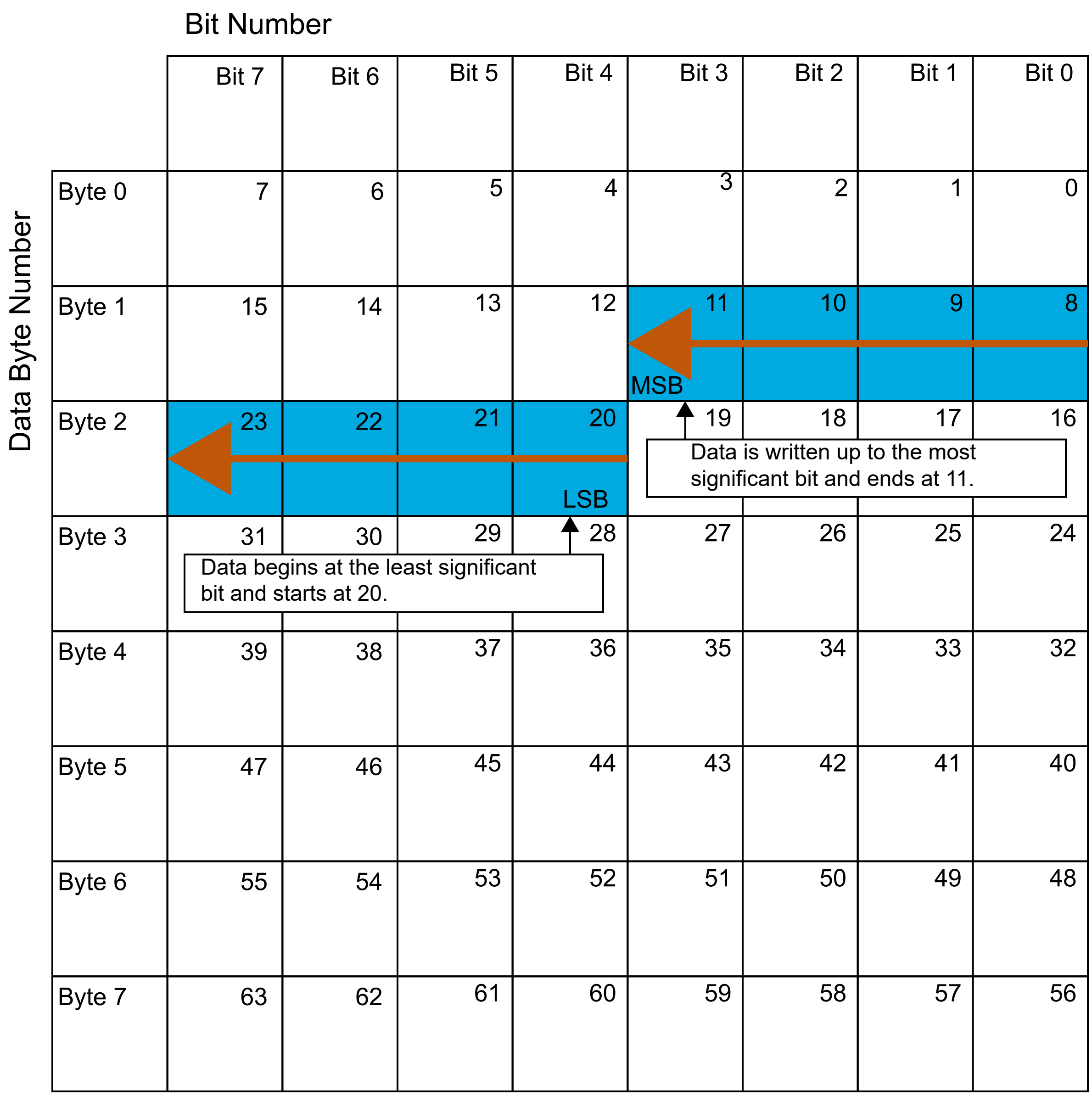
Select either of the following options:

- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the least-significant bit to the most-significant bit and 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.



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 least-significant bit to the most-significant bit and 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.



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 time step:

- `Standard`: The signal is unpacked at each time step.
- `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 these values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

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

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). For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. For more information, see the **Data input as** parameter conversion formula.

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.

Programmatic Use
Block Parameter: `SignalInfo`
Type: `string` | `character vector`

`Output identifier` — Add CAN ID output port
`off` (default) | `on`

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

Programmatic Use
Block Parameter: `IDPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output timestamp` — Add Timestamp output port
`off` (default) | `on`

Select this option to output the message timestamp. This value indicates when the message was received, measured as the number of seconds elapsed since the model simulation began. This option adds a new output port to the block. The data type of this port is `double`.

Programmatic Use
Block Parameter: `TimestampPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output error` — Add Error output port
`off` (default) | `on`

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. If the output value is `0`, there is no error. The data type of this port is `uint8`.

Programmatic Use
Block Parameter: `ErrorPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output remote` — Add Remote output port
`off` (default) | `on`

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`.

Programmatic Use
Block Parameter: `RemotePort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output length` — Add Length output port
`off` (default) | `on`

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`.

Programmatic Use
Block Parameter: `LengthPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output status` — Add Status output port
`off` (default) | `on`

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`.

Programmatic Use
Block Parameter: `StatusPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output Bit Rate Switch (BRS)` — Add BRS output port
`off` (default) | `on`

(Disabled for CAN protocol.) Select this option to output the message bit rate switch. This option adds a new output port to the block. The data type of this port is `boolean`.

Programmatic Use
Block Parameter: `BRSPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output Error Status Indicator (ESI)` — Add ESI output port
`off` (default) | `on`

(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`.

Programmatic Use
Block Parameter: `ESIPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output Data Length Code (DLC)` — Add DLC output port
`off` (default) | `on`

(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`.

Programmatic Use
Block Parameter: `DLCPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`

Default: `'off'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
CAN FD Configuration | CAN FD Pack | CAN FD Receive | CAN Unpack | CAN Unpack

Functions
`canFDMessageBusType`

Topics
“Design Your Model for Effective Acceleration” (Simulink)
“Composite Signals” (Simulink)

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 block 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" on page 14-73.

The CAN 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.

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.

Ports

Input

CAN Msg — CAN messages to log
CAN_MESSAGE | CAN_MESSAGE_BUS

The **CAN Msg** input port is available when the **Log messages from** parameter is set to Input port. Provide an input from another block as a CAN_MESSAGE or a Simulink signal bus of type CAN_MESSAGE_BUS.

Data Types: CAN_MESSAGE | CAN_MESSAGE_BUS

Parameters

File name — Log file location and name
untitled.mat (default) | file name

Enter 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 format YYYY-MMM-DD_hhmmss. Specify a unique name to differentiate between your files for repeated logging.

Variable name — Variable name for CAN message in log file
ans (default) | variable name

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

Maximum number of messages to log — Limit quantity of messages
10000 (default) | numeric

Specify the maximum number of messages this block can log from the selected device or port. The specified value must be a positive integer. The default value is `10000` messages. The log file saves the most recent messages up to the specified maximum number.

`Log messages from` — Source of messages
`CAN Bus` (default) | `Input port`

Select the source of the messages logged by the block. To log messages from the CAN bus network, select `CAN Bus`, then specify a **Device**. To log messages from another block in the model, select `Input port`, which adds an inport port to the block.

`Device` — CAN device and channel
list option

Select the device on the CAN network that you want to log messages from. This field is available only if you select `CAN Bus` for the **Log messages from** option.

`Sample time` — Block sampling time in simulation
`0.01` (default) | numeric

Specify the sampling time of the block during simulation. 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 the sample time. You can also specify a MATLAB variable for sample time. The default value is `0.01` simulation seconds. For more information, see “Timing in Hardware Interface Models” on page 8-21.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-44.

See Also

Blocks
CAN Configuration | CAN Replay

Introduced in R2011b

CAN Pack

Pack individual signals into CAN message

Library: Vehicle Network Toolbox / CAN Communication
Embedded Coder / Embedded Targets / Host Communication
Embedded Coder Support Package for Texas Instruments
C2000 Processors / Target Communication
Simulink Real-Time / CAN / CAN MSG blocks

Description

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

To use this block, you must have a license for Simulink software.

The CAN Pack block supports:

- Simulink Accelerator rapid accelerator mode. You can speed up the execution of Simulink models.
- Model referencing. Your model can include other Simulink models as modular components.

For more information, see “Design Your Model for Effective Acceleration” (Simulink).

Tip

- This block can be used to encode the signals of J1939 parameter groups up to 8 bytes. However, to work with J1939 messages, it is preferable to use the blocks in the J1939 Communication block library instead of this block. See “J1939 Communication”.

Ports

Input

Data — CAN message signal input
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

The 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 message has four signals, the block can have four input ports.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Code generation to deploy models to targets. If your signal information consists of signed or unsigned integers greater than 32 bits long, code generation is not supported.

Output

CAN Msg — CAN message output
CAN_MESSAGE | CAN_MESSAGE_BUS

This block has one output port, CAN Msg. The CAN Pack block takes the specified input signals and packs them into a CAN message. The output data type is determined by the **Output as bus** parameter setting.

Parameters

Data input as — Select your data signal
raw data (default) | manually specified signals | CANdb specified signals

- `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.

  The conversion formula is:

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

  where `physical_value` is the original value of the signal and `raw_value` is the packed signal value.
- `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.

Programmatic Use
Block Parameter: `DataFormat`
Type: `string` | `character vector`
Values: `'raw data'` | `'manually specified signals'` | `'CANdb specified signals'`
Default: `'raw data'`

CANdb file — CAN database file
character vector

This option is available if you specify that your data is input through 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.

File names that contain non-alphanumeric characters such as equal signs, ampersands, and so on are not valid CAN database file names. You can use periods in your database name. Before you use the CAN database files, rename them with non-alphanumeric characters.

Programmatic Use
Block Parameter: `CANdbFile`
Type: `string` | `character vector`

`Message list` — CAN message list
array of character vectors

This option is available if you specify that your data is input through 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.

Programmatic Use
Block Parameter: `MsgList`
Type: `string` | `character vector`

Name — CAN message name
`CAN Msg` (default) | character vector

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 not available if you choose to use signals from a CANdb file.

Programmatic Use
Block Parameter: `MsgName`
Type: `string` | `character vector`

`Identifier type` — CAN identifier type
`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

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.

Programmatic Use
Block Parameter: `MsgIDType`
Type: `string` | `character vector`
Values: `'Standard (11-bit identifier)'` | `'Extended (29-bit identifier)'`
Default: `'Standard (11-bit identifier)'`

`CAN Identifier` — CAN message ID
`0` (default) | `0` to `536870911`

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 by using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`
Type: `string` | `character vector`
Values: `'0'` to `'536870911'`

`Length (bytes)` — CAN message length
`8` (default) | `0` to `8`

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.

Programmatic Use
Block Parameter: `MsgLength`
Type: `string` | `character vector`
Values: `'0'` to `'8'`
Default: `'8'`

`Remote frame` — CAN message as remote frame

`off` (default) | `on`

Specify the CAN message as a remote frame.

Programmatic Use
Block Parameter: `Remote`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output as bus` — CAN message as bus

`off` (default) | `on`

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).

Programmatic Use
Block Parameter: `BusOutput`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Add signal` — Add CAN signal

Add a new signal to the signal table.

Programmatic Use

None

`Delete signal` — Remove CAN signal

Remove the selected signal from the signal table.

Programmatic Use

None

`Signals` — Signals table

table

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

If you are using a CANdb file, the data in the file populates this table 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 in this table. Each signal that you create has these 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.

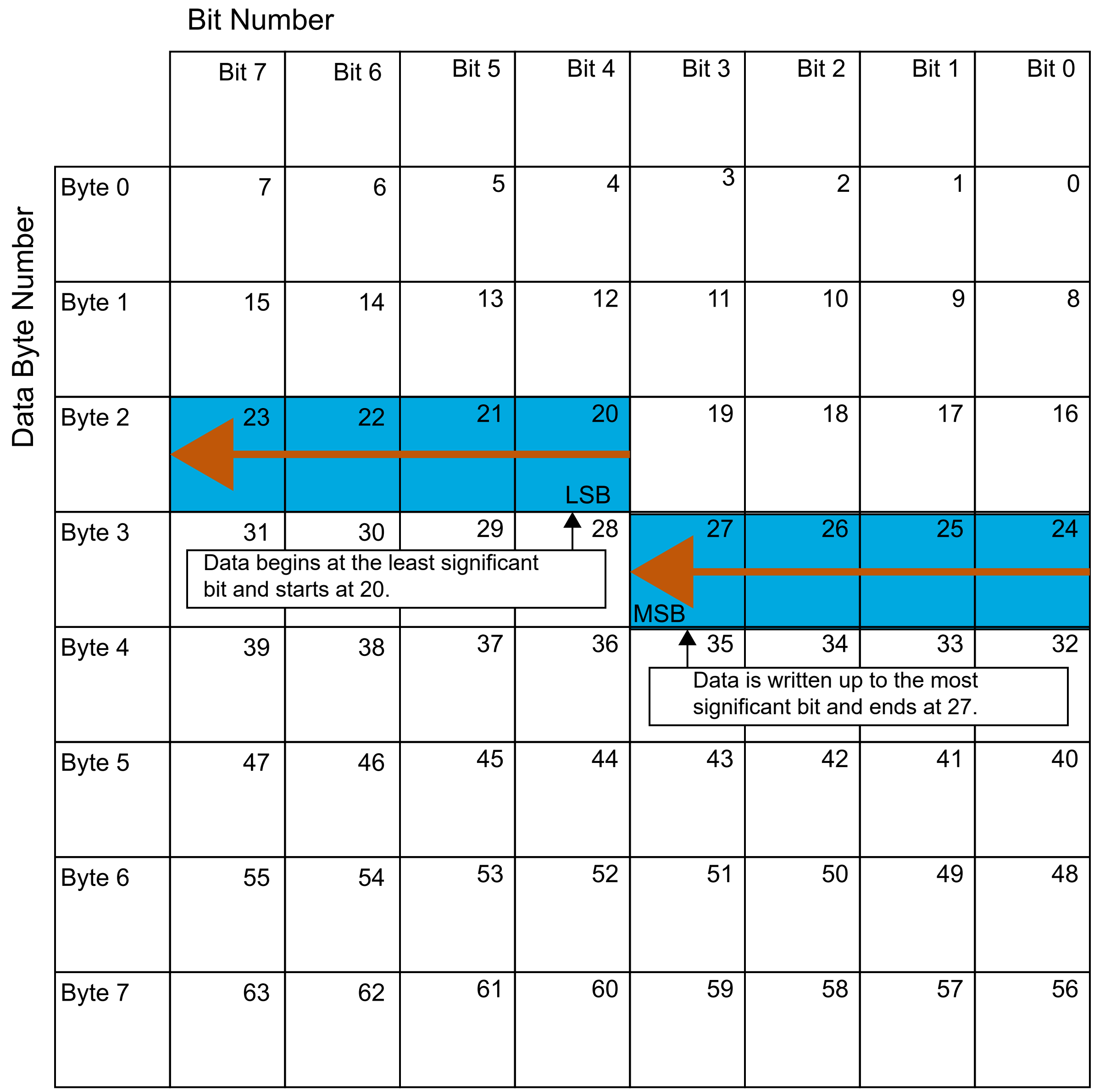
Length (bits)

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

Byte order

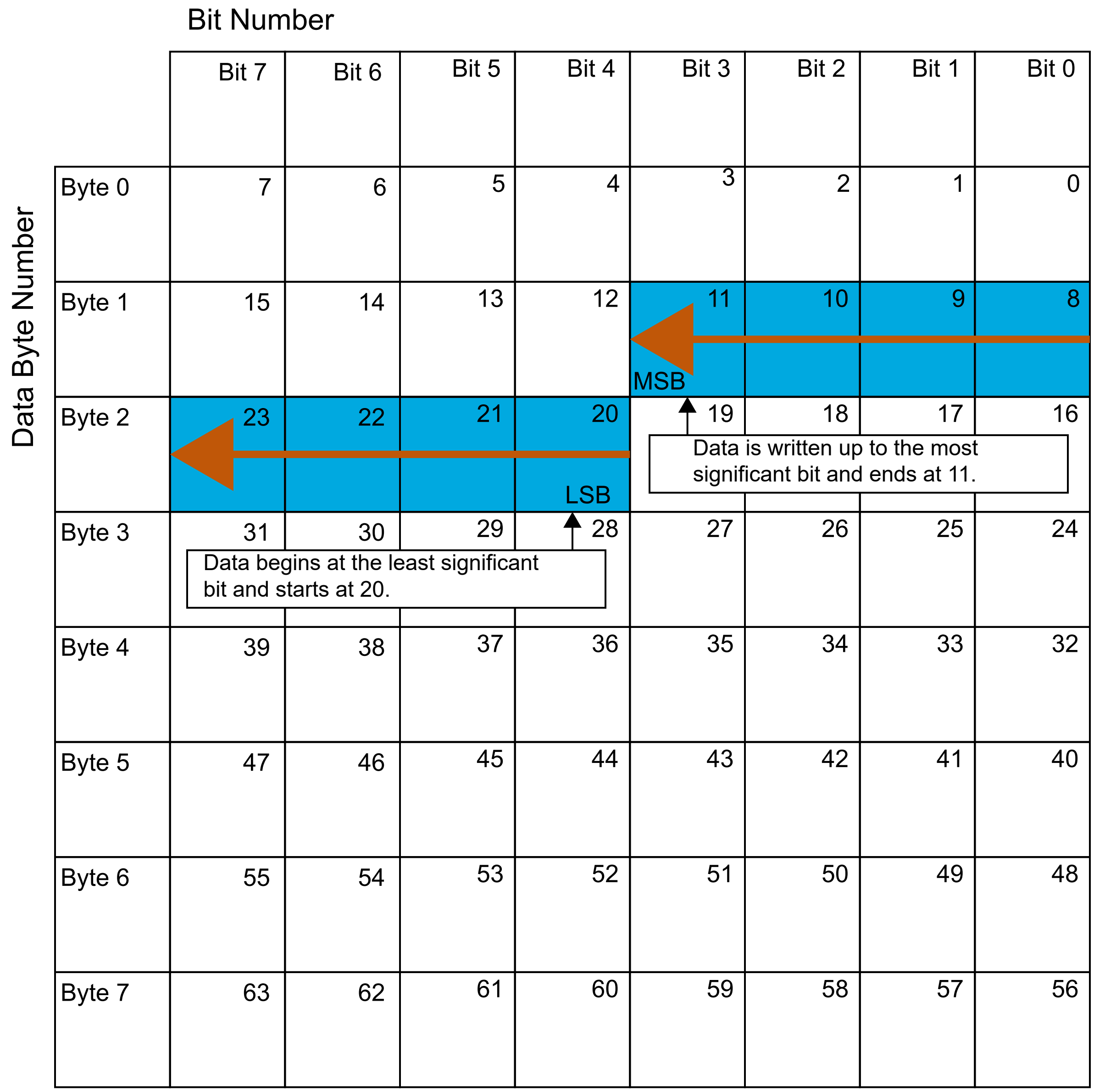
Select either of these options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the least significant bit, to the most significant bit. 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.



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 least-significant bit to the most-significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



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

Data type

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

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

Multiplex type

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

- `Standard`: The signal is packed at each time step.
- `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 these signals with the following types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

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

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide 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. For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. For more information, see the **Data input as** parameter conversion formula.

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 the settings.

Programmatic Use
Block Parameter: `SignalInfo`
Type: `string` | `character vector`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
CAN Unpack | CAN FD Pack

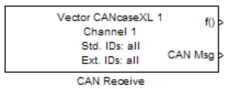
Topics
"Design Your Model for Effective Acceleration" (Simulink)

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 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.

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.

Ports

Output

CAN Msg — Received CAN messages
CAN_MESSAGE | bus

The CAN Msg output port contains one or more packed CAN messages received at that particular timestep, output as a signal bus or CAN_MESSAGE. The output includes either one or all messages for that timestep, depending on the setting of **Number of messages received at each timestep**.

Data Types: CAN_MESSAGE | bus

f() — Function-call event output
function-call event

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.

Data Types: function-call event

Parameters

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

Device — CAN device and channel
list option

Select from the list 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.

Programmatic Use
Block Parameter: Device
Type: character vector, string

Standard IDs Filter — Limit or allow messages based on standard ID
Allow all (default) | Allow only | Block all

Select the filter for standard IDs. Choices are:

- Allow all (default): Allows all standard IDs to pass the filter.
- Allow only: Allows only the ID or range of IDs specified in the text field, specified as a single ID or an array of IDs. You can also specify disjointed IDs or arrays separated by a comma. For example, to allow IDs 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.

Programmatic Use
Block Parameter: `StdIDsCombo`
Type: character vector, string
Values: `'Allow all'` | `'Allow only'` | `'Block all'`
Default: `'Allow all'`

If using `'Allow only'`, set the filter values with the following:
Block Parameter: `StandardIDs`
Type: character vector, string
Values: integer scalar or row vector

Extended IDs Filter — Limit or allow messages based on extended ID
Allow all (default) | Allow only | Block all

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

- Allow all (default): Allows all extended IDs to pass the filter.
- Allow only: Allows only those IDs specified in the text field. Allows only the ID or range of IDs specified in the text field, specified as 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 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.

Programmatic Use
Block Parameter: `ExtIDsCombo`
Type: character vector, string
Values: `'Allow all'` | `'Allow only'` | `'Block all'`
Default: `'Allow all'`
Block Parameter: `ExtendedIDs`
Type: character vector, string
Values: integer scalar or row vector

Sample time — Block execution rate
0.01 (default)

Specify the sample time of the block during the simulation. This is the rate at which the block is executed during simulation. The default value is 0.01 simulation seconds. For more information, see "Timing in Hardware Interface Models" on page 8-21.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector, string
Values: double
Default: `'0.01'`

`Number of messages received at each timestep` — Receive one or all messages
`1` (default) | `all`

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.

Programmatic Use
Block Parameter: `MsgsPerTimestep`
Type: character vector, string
Values: `'1'` | `'all'`
Default: `'1'`

`Output as bus` — Output Simulink bus signal
`off` (default) | `on`

Output a native Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

Programmatic Use
Block Parameter: `BusOutput`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-56.

See Also

Blocks
CAN Configuration | CAN Unpack

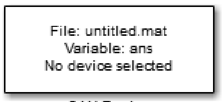
Functions
`canMessageBusType`

Introduced in R2009a

CAN Replay

Replay logged CAN messages

Library: Vehicle Network Toolbox / CAN Communication



CAN Replay

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 `canMessageReplayBlockStruct` and save the result to a separate file. For more information, see “Log and Replay CAN Messages” on page 14-73.

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 13-61.

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.

Ports

Output

CAN Msg — Replayed CAN messages

CAN_MESSAGE | CAN_MESSAGE_BUS

This output port contains a packed CAN message logged at that particular timestep, output as a CAN_MESSAGE or signal bus of type CAN_MESSAGE_BUS.

Data Types: CAN_MESSAGE | CAN_MESSAGE_BUS

f() — Function-call event output

function-call event

This port provides a trigger to a Function-Call subsystem when the block receives a new message. You can connect it to a Function-Call Subsystem to unpack and process the message.

Data Types: function-call event

Parameters

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

File name — Path and name of MAT-file with messages

untitled.mat (default) | 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` — Variable in MAT-file holding messages
`ans` (default) | variable

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

`Number of times to replay messages` — Repeat value
`Inf` (default) | integer

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 output location
CAN Bus (default) | `Output port`

Specify if the model is replaying messages to the CAN network or an output port. When replaying to the CAN network, you must also select a **Device**.

`Device` — CAN device and channel
device list option

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

`Sample time` — Block execution rate
`0.01` (default) | numeric

Specify the sampling time of the block during simulation. 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 the sample time. You can also specify a MATLAB variable for sample time. The default value is `0.01` simulation seconds. For more information, see “Timing in Hardware Interface Models” on page 8-21.

`Output as bus` — Enable signal bus output
off (default) | on

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).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-61.

See Also

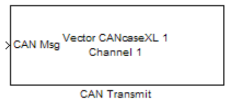
Blocks
CAN Log

Functions
`canMessageBusType` | `canMessageReplayBlockStruct`

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 that was 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.

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.

Ports

Input

CAN Msg — CAN message to transmit
packed CAN message

CAN message as packed by the CAN Pack block, input as a CAN_MESSAGE or a Simulink signal bus.

Data Types: CAN_MESSAGE | bus

Parameters

Device — CAN device and channel
list option

Select the CAN device and channel for transmitting CAN messages to the network. The list of options 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.

Programmatic Use
Block Parameter: Device
Type: character vector, string

Transmit Options: On data change — Enable event-based transmission when data changes
'off' (default) | 'on'

When event-based transmission is enabled, messages are transmitted only at those time steps when a change in message data is detected. When the input data matches the most recent transmission for a given message ID, the message is not re-transmitted.

Event and periodic transmission can both be enabled to work together simultaneously. If neither is selected, the default behavior is to transmit the current input at each time step.

Programmatic Use
Block Parameter: `EnableEventTransmit`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Transmit Options: Periodic` — Enable periodic transmission
`'off'` (default) | `'on'`

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.

Programmatic Use
Block Parameter: `EnablePerioicTransmit`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Transmit Options: Message period` — Period of message transmission rate
`1.000` (default) | positive numeric scalar

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

Programmatic Use
Block Parameter: `MessagePeriod`
Type: character vector, string
Values: double
Default: `'1.000'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates code with limited portability that runs only on the host computer. See “Code Generation” on page 13-65.

See Also

Blocks
CAN Configuration | CAN Pack

Introduced in R2009a

CAN Unpack

Unpack individual signals from CAN messages

Library: Vehicle Network Toolbox / CAN Communication
Embedded Coder / Embedded Targets / Host Communication
Embedded Coder Support Package for Texas Instruments C2000 Processors / Target Communication
Simulink Real-Time / CAN / CAN MSG blocks

CAN Unpack

Description

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

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

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.

For more information on these features, see “Design Your Model for Effective Acceleration” (Simulink).

Tip

- To process every message coming through a channel, it is recommended that you use the CAN Unpack block in a function trigger subsystem. See “Using Triggered Subsystems” (Simulink).
- This block can be used to decode the signals of J1939 parameter groups up to 8 bytes. However, to work with J1939 messages, it is preferable to use the blocks in the J1939 Communication block library instead of this block. See “J1939 Communication”.

Ports

Input

`CAN Msg` — CAN message input

`CAN_MESSAGE` | `CAN_MESSAGE_BUS`

This block has one input port, **CAN Msg**. The block takes the specified input CAN messages and unpacks their signal data to separate outputs.

The block supports the following signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Code generation to deploy models to targets. Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

Output

`Data` — CAN signal output
single | double | int8 | int16 | int32 | int64 | uint32 | uint64 | boolean

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

For signals specified manually or by a CANdb, the default output data type for CAN signals 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.

Additional output ports can be added by selecting the options in the parameters **Output ports** pane. For more information, see the parameters `Output identifier`, `Output timestamp`, `Output error`, `Output remote`, `Output length`, and `Output status`.

Parameters

`Data to output as` — Select your data signal
`raw data` (default) | `manually specify signals` | `CANdb specified signals`

- `raw data`: Output data as a uint8 vector array. If you select this option, you specify only the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.

  The conversion formula is:

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

  where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.
- `manually specified signals`: You can 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 that you specify. For example, if you specify four signals, your block has four output ports.
- `CANdb specified signals`: You can 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.

Programmatic Use
Block Parameter: `DataFormat`
Type: `string` | `character vector`
Values: `'raw data'` | `'manually specified signals'` | `'CANdb specified signals'`
Default: `'raw data'`

`CANdb file` — CAN database file
character vector

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. 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.

Programmatic Use
Block Parameter: `CANdbFile`
Type: `string` | `character vector`

`Message list` — CAN message list

array of character vectors

This option is available if you specify in the **Data to be output as** list that your data is to be output as a CANdb file 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.

Programmatic Use
Block Parameter: `MsgList`
Type: `string` | `character vector`

`Name` — CAN message name

`CAN Msg` (default) | character vector

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.

Programmatic Use
Block Parameter: `MsgName`
Type: `string` | `character vector`

`Identifier type` — CAN identifier type

`Standard (11-bit identifier)` (default) | `Extended (29-bit identifier)`

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.

Programmatic Use
Block Parameter: `MsgIDType`
Type: `string` | `character vector`
Values: `'Standard (11-bit identifier)'` | `'Extended (29-bit identifier)'`
Default: `'Standard (11-bit identifier)'`

`CAN Identifier` — CAN message identifier

`0` (default) | `0` to `536870911`

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 by using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use
Block Parameter: `MsgIdentifier`

Type: `string` | `character vector`
Values: `'0'` to `'536870911'`

`Length (bytes)` — CAN message length

8 (default) | `0` to `8`

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

Programmatic Use

Block Parameter: `MsgLength`
Type: `string` | `character vector`
Values: `'0'` to `'8'`
Default: `'8'`

`Add signal` — Add CAN signal

Add a signal to the signal table.

Programmatic Use

None

`Delete signal` — Remove CAN signal

Remove the selected signal from the signal table.

Programmatic Use

None

`Signals` — Signals table

table

If you choose to specify signals manually or define signals by using a CANdb file, this table appears.

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

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

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

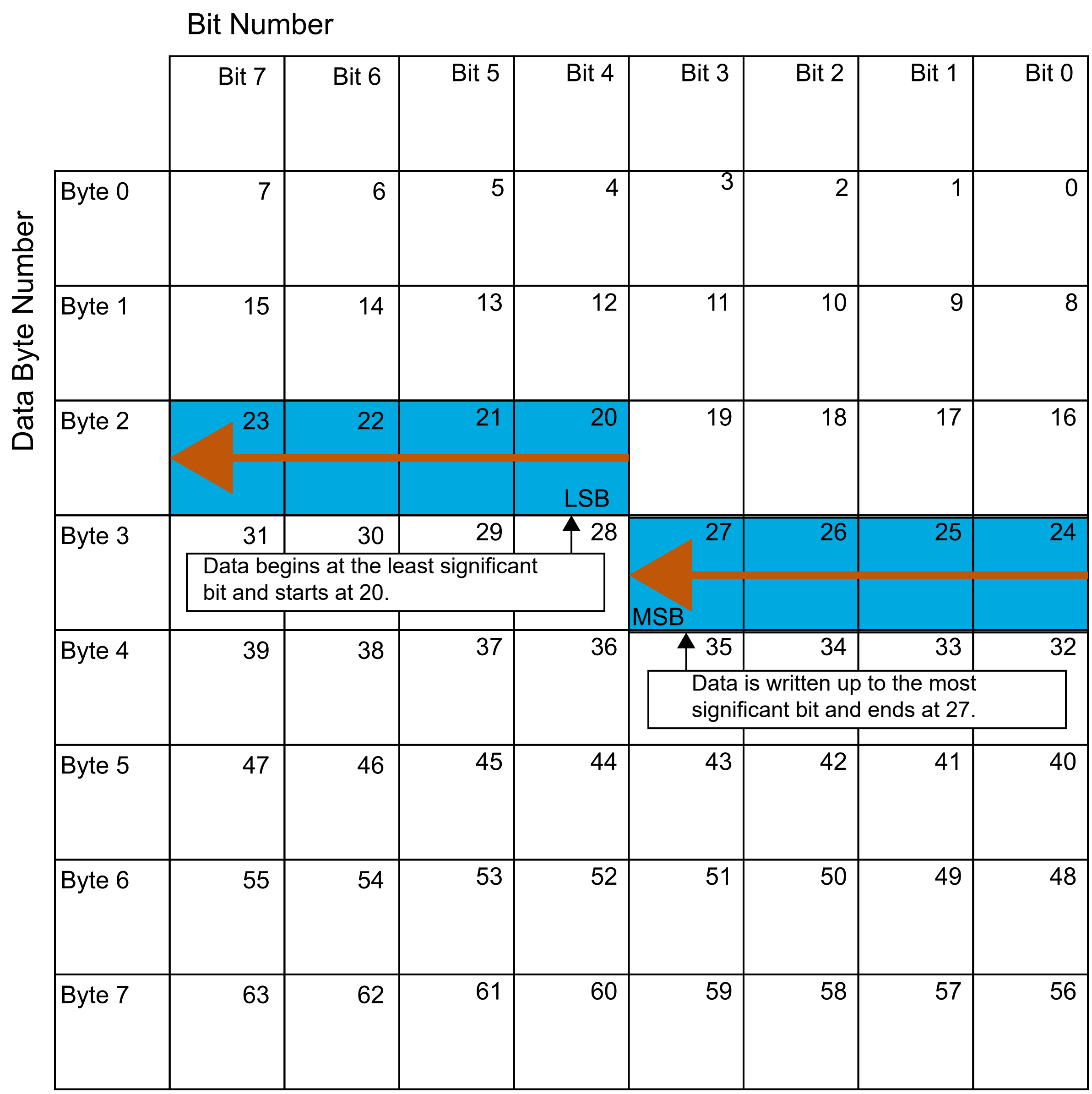
Start bit
: Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message. The start bit must be an integer from 0 through 63.

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

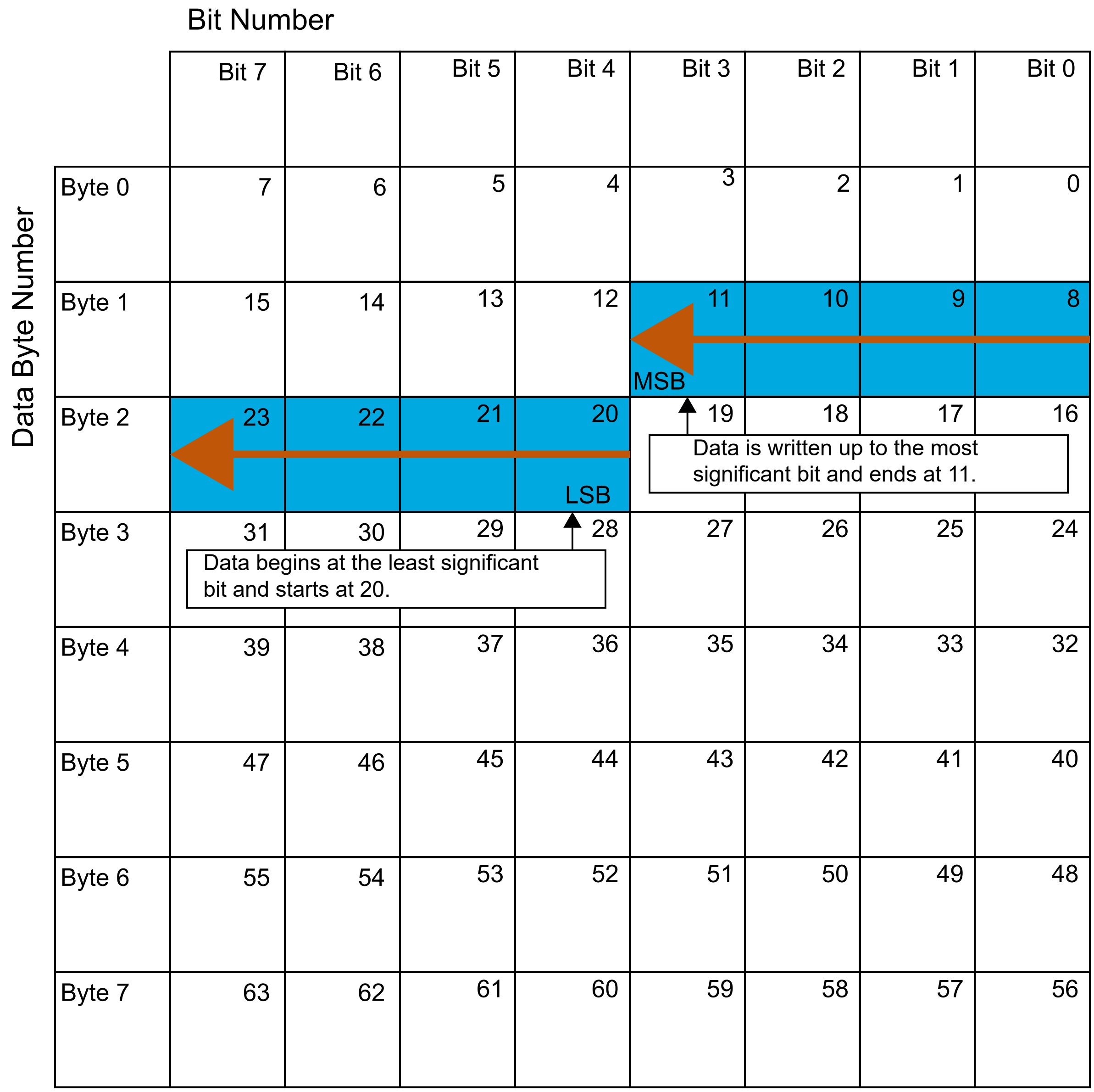
Select either of these options:

- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the least-significant bit to the most-significant bit. 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.



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 least-significant bit to the most-significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



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

Data type

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

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

Multiplex type

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

- `Standard`: The signal is unpacked at each time step.
- `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 these values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

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

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide 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). For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. For more information, see the **Data input as** parameter conversion formula.

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.

Programmatic Use

Block Parameter: `SignalInfo`
Type: `string` | `character vector`

`Output identifier` — Add CAN ID output port

`off` (default)

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

Programmatic Use
Block Parameter: `IDPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output timestamp` — Add Timestamp output port

`off` (default) | `on`

Select this option to output the message timestamp. This value indicates when the message was received, measured as the number of seconds elapsed since the model simulation began. This option adds a new output port to the block. The data type of this port is `double`.

Programmatic Use
Block Parameter: `TimestampPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output error` — Add Error output port

`off` (default) | `on`

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. If the output value is `0`, there is no error. The data type of this port is `uint8`.

Programmatic Use
Block Parameter: `ErrorPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output remote` — Add Remote output port

`off` (default) | `on`

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`.

Programmatic Use
Block Parameter: `RemotePort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output length` — Add Length output port

`off` (default) | `on`

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`.

Programmatic Use
Block Parameter: `LengthPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

`Output status` — Add Status output port
`off` (default) | `on`

Select this option to output the message received status. The status is `1` if the block receives a 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`.

Programmatic Use
Block Parameter: `StatusPort`
Type: `string` | `character vector`
Values: `'off'` | `'on'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
CAN Pack | CAN FD Unpack

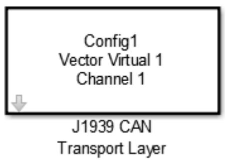
Topics
"Design Your Model for Effective Acceleration" (Simulink)

Introduced in R2009a

J1939 CAN Transport Layer

Transport J1939 messages via CAN

Library: Simulink Real-Time / J1939 Communication
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` — J1939 network configuration name
configuration list option

The name of the J1939 Network Configuration block to associate this transport layer block with.

`Device` — CAN device
device list option

The CAN device, chosen from all connected CAN devices.

`Bus speed` — Speed of CAN bus
`250000` (default) | `500000`

Speed of the CAN bus in bits per second, specified as one of the two rates supported by the J1939 protocol, 250000 or 500000. The default is 250000.

`Sample time` — Simulation refresh rate
`0.01` (default) | numeric

Simulation refresh rate, specified as 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, “Timing in Hardware Interface Models” on page 8-21. 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` simulation seconds.

See Also

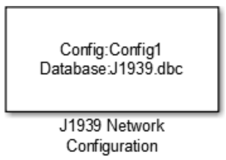
Blocks
J1939 Network Configuration | J1939 Node Configuration | J1939 Receive | J1939 Transmit

Introduced in R2015b

J1939 Network Configuration

Define J1939 network configuration name and database file

Library: Simulink Real-Time / J1939 Communication
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.

To use this block, you must have a license for both Vehicle Network Toolbox and Simulink software.

The J1939 communication blocks support the use of Simulink accelerator and rapid accelerator modes. You can speed up the execution of Simulink models by using these modes. For more information on these modes, see the Simulink documentation.

The J1939 communication blocks also support code generation that have limited deployment capabilities. Code generation requires a C++ compiler that is compatible with the code generation target. For the current list of supported compilers, see Supported and Compatible Compilers.

Parameters

`Configuration name` — Define a name for this J1939 network configuration
`ConfigX` (default) | character vector

The default value is `ConfigX`, where the number `X` increases from 1 based on the number of existing blocks.

`Database File` — Specify the J1939 database file name relative to the current folder
not set (default) | character vector

An example file name, 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. This file must be in the DBC file format defined by Vector Informatik GmbH.

See Also

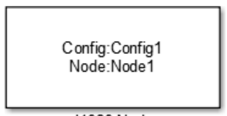
J1939 CAN Transport Layer | J1939 Receive | J1939 Transmit | J1939 Node Configuration

Introduced in R2015b

J1939 Node Configuration

Configure J1939 node with address and network management attributes

Library: Simulink Real-Time / J1939 Communication
Vehicle Network Toolbox / J1939 Communication



J1939 Node Configuration

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.

To use this block, you must have a license for both Vehicle Network Toolbox and Simulink software.

The J1939 communication blocks support the use of Simulink accelerator and rapid accelerator modes. You can speed up the execution of Simulink models by using these modes. For more information on these modes, see “Design Your Model for Effective Acceleration” (Simulink).

The J1939 communication blocks also support code generation that have limited deployment capabilities. Code generation requires a C++ compiler that is compatible with the code generation target. For the current list of supported compilers, see Supported and Compatible Compilers.

Ports

Output

`Address` — Returns the effective address of the node
int8

This optional output port exists when you select the **Output current node address** check box in the dialog box.

`AC Status` — Indicates the success (1) or failure (0) of the node’s address claim
0 | 1

This optional output port exists when you select the **Output address claim status** check box in the dialog box.

Parameters

`Config name` — ID of the J1939 network configuration to associate with this node
`ConfigX` (default) | character vector

To access the corresponding J1939 database, use this ID.

`Node name` — name of this J1939 node
`NodeX` (default) | character vector

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` — Nine network attributes as defined by the database file consistent with the J1939 protocol
vector array

Unless you are defining a custom node, these parameters are read-only:

- **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
0.01 (default) | double

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` simulation seconds. For information about simulation sample timing, see “Timing in Hardware Interface Models” on page 8-21.

`Output current node address` — Enable or disable the Address port display
`off` (default) | `on`

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 display
`off` (default) | `on`

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

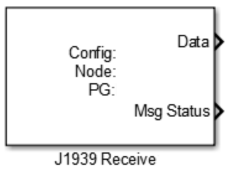
J1939 CAN Transport Layer | J1939 Receive | J1939 Transmit | J1939 Network Configuration

Introduced in R2015b

J1939 Receive

Receive J1939 parameter group messages

Library: Simulink Real-Time / J1939 Communication
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 by using the J1939 Network Configuration block.

To use this block, you must have a license for both Vehicle Network Toolbox and Simulink software.

The J1939 communication blocks support the use of Simulink accelerator and rapid accelerator modes. You can speed up the execution of Simulink models by using these modes. For more information on these modes, see "Design Your Model for Effective Acceleration" (Simulink).

The J1939 communication blocks also support code generation that have limited deployment capabilities. Code generation requires a C++ compiler that is compatible with the code generation target. For the current list of supported compilers, see Supported and Compatible Compilers.

Ports

Output

`Data` — Data output
double

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` — Message received status
0 | 1

When you select the **Output New Message Received status** check box in the parameters dialog, this port outputs `1` when a new message is received from the CAN bus. Otherwise, this port outputs `0`.

Parameters

`Config name` — Name of the J1939 network configuration to associate
`ConfigX` (default) | character vector

The name of the J1939 network configuration to associate. This value 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` — Name of the J1939 node
`NodeX` (default) | character vector

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` — Parameter group number (PGN) and name from database
character vector

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.

If you change any parameter group settings within your J1939 database file, open the J1939 Receive block dialog box and select the same **Parameter Group** and click **OK** or **Apply**.

`Signals` — Signals defined in the parameter group
array of character vectors

Signals that are 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 all` (default) | `Allow only`

Filter messages based on source address are:

- `Allow only` — Specify a single source address.
- `Allow all` — Accepts messages from any source address. This option is the default.

`Destination Address Filter` — Filter out message based on destination address
`global and node specific` (default) | `global only` | `node specific only`

Filter out a message based on the destination address:

- `global only` — Receive only broadcast messages.
- `node specific only` — Receive only messages addressed to this node.
- `global and node specific` — Receive all broadcast and node-addressed messages. This option is the default.

`Sample time` — Simulation refresh rate
`0.01` (default) | double

The simulation refresh rate. Specify the sampling time of the block during simulation. This value defines the frequency at which the J1939 Receive block 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` simulation seconds. For information about simulation sample timing, see "Timing in Hardware Interface Models" on page 8-21.

`Output New Message Received status` — Create a Msg Status output
0 (default) | 1

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

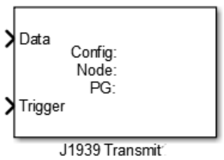
J1939 CAN Transport Layer | J1939 Transmit | J1939 Network Configuration | J1939 Node Configuration

Introduced in R2015b

J1939 Transmit

Transmit J1939 message

Library: Simulink Real-Time / J1939 Communication
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 by using the J1939 Network Configuration block.

To use this block, you must have a license for both Vehicle Network Toolbox and Simulink software.

The J1939 communication blocks support the use of Simulink accelerator and rapid accelerator modes. You can speed up the execution of Simulink models by using these modes. For more information on these modes, see “Design Your Model for Effective Acceleration” (Simulink).

The J1939 communication blocks also support code generation that have limited deployment capabilities. Code generation requires a C++ compiler that is compatible with the code generation target. For the current list of supported compilers, see Supported and Compatible Compilers.

Ports

Input

`Data` — Input data
signal

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 message
0 | 1

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` — Name of the J1939 network configuration to associate
`ConfigX` (default) | character vector

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` — Name of the J1939 node
`NodeX` (default) | character vector

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` — Group number (PGN) and name
int8

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.

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 parameter group
array of character vectors

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
int8

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 (higher value) will refrain from transmitting. The value can range from 0 (highest priority) to 7 (lowest).

`Destination Address` — Name of the destination node
int8

The 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

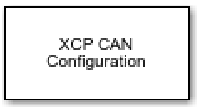
J1939 CAN Transport Layer | J1939 Receive | J1939 Network Configuration | J1939 Node Configuration

Introduced in R2015b

XCP CAN Configuration

Configure XCP server connection

Library: Vehicle Network Toolbox / XCP Communication / CAN
Simulink Real-Time / XCP / CAN



Description

The XCP CAN Configuration block uses the parameters specified in the A2L file and the ASAP2 database to establish an XCP server 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 server 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 you to add one.

The XCP CAN communication blocks support Simulink accelerator mode and rapid accelerator mode. You can speed up the execution of Simulink models by using these modes. For more information about these simulation modes, see “Design Your Model for Effective Acceleration” (Simulink).

Parameters

`Config name` — Specify XCP CAN session name

`'CAN_Config1'` (default)

Specify a unique name for your XCP CAN session.

`A2L File` — Select an A2L file

file name

Click **Browse** to select an A2L file for your XCP CAN session..

`Enable seed/key security` — Select that key required to establish connection

`'off'`

Select this option if your server requires a secure key to establish connection. Use the **File (*DLL)** parameter to specify the DLL file that contains the seed/key definition.

`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 the seed and key security algorithm that unlocks an XCP server module. This parameter is available in Windows Desktop Simulation for Vehicle Network Toolbox.

The **File (*.DLL)** parameter specifies the name of the DLL-file that contains the seed and key security algorithm used to unlock an XCP server module. The file defines the algorithm for generating the access key from a given seed according to ASAM standard definitions. For information on the file format and API, see the Vector web page Steps to Use Seed&Key Option in CANape or "Seed and Key Algorithm" in National Instruments CAN ECU Measurement and Calibration Toolkit User Manual. **Note:** The DLL must be the same bitness as MATLAB (64-bit).

`Output connection status` — Display connection status
`'off'`

Select this option to display the status of the connection to the server module. Selecting this option adds a new output port.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

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 server connection

Library: Vehicle Network Toolbox / XCP Communication / CAN
Simulink Real-Time / XCP / CAN

Description

The XCP CAN Data Acquisition block acquires data from the configured server connection based on measurements that you select. 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 server connection. The block displays the selected measurements as output ports.

The XCP communication blocks support the use of Simulink accelerator mode and rapid accelerator mode. You can speed up the execution of Simulink models by using these modes. For more information on these simulation modes, see “Design Your Model for Effective Acceleration” (Simulink).

Parameters

`Config name` — Specify XCP CAN session name

select from list

Select the name of the XCP configuration that you want to use. This 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.

`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.

`All Measurements` — List all measurements available for event

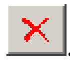
measurements list

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. On your keyboard, press the **Ctrl** key 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** list displays a list of all matching names. Click the x to clear your search.

`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.

`Block Output Settings` — Set the port output as Compu method conversion values or raw values

`Raw values as double (no Compu method conversion)` (default) | `Raw values (no Compu method conversion)` | `Physical values (apply Compu method conversion)`

This parameter enables support for XCP data types and dimensions as defined in the ASAP2 standard. The Block Output Settings parameter selects whether the port outputs Compu method conversion values or raw values. The options provide:

- `Physical values (apply Compu method conversion)` enables the raw-to-physical conversion of ECU measurement values. For this option, the block port settings are set either to `'double'` or `'string'`, based on the underlying Compu method conversion. For example, Compu method `IDENTICAL`, `LINEAR`, `RAT_FUNC`, `TAB_INTP`, and `TAB_NOINTP` port settings is `'double'` while Compu method `TAB_VERB` port settings is `'string'`. The maximum string length supported for Compu method conversion is 1024 as specified in the ASAM XIL specification.

  The FORM Compu method conversion is not supported. Simulink throws a warning for such a conversion and IDENTICAL conversion is applied to the underlying measurement. Also, only scalar measurement signals are supported for TAB_VERB conversion as Simulink only supports scalar strings.

  Selecting this option shows the physical units (if any) in front of the measurement name on the block mask. This physical unit is acquired from the A2L description of the measurement and Compu method. If the physical unit is not specified, only the measurement name is displayed.
- `Raw values (no Compu method conversion)` sets the port data type according to the type definition in the A2L file and supports up to three-dimensional XCP measurements in Simulink.
- `Raw values as double (no Compu method conversion)` sets the port data type as `double`, converting all internal measurement values. This selection supports up to three-dimensional XCP measurements in Simulink.

These ASAP2 data types are supported by corresponding Simulink port data types:

- `SBYTE`
- `UWORD`
- `SWORD`
- `ULONG`

- SLONG
- A_UINT64
- A_INT64
- FLOAT32_IEEE
- FLOAT64_IEEE

The dimension support in the block accommodates the different treatment of matrices by MATLAB and the ECU. The MATLAB default operation treats matrices as row-major matrices. An XCP measurement can have a LAYOUT as COLUMN_DIR or ROW_DIR . If a matrix measurement is COLUMN_DIR, the blocks rearrange the measurement in memory and ensure that the matrix (row X, col Y) in MATLAB refers to the same entry as (row X, col Y) on the ECU. The rearrangement causes matrix entries that are contiguous on the ECU to be noncontiguous in MATLAB and Simulink.

DAQ List Priority — Specify a priority value for server device driver
priority value

Specify a priority value as an integer from 0 to 255 for the server device driver to prioritize transmission of data packets. The server can accumulate XCP packets for lower priority DAQ lists before transmission to the client. A value of 255 has the highest priority. The SET_DAQ_LIST_MODE command communicates the **DAQ List Priority** value from client to server. This communication method differs from the specification of the Event Channel Priority property, which comes from the A2L file.

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 simulation seconds. For more information, see “Timing in Hardware Interface Models” on page 8-21.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

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: Vehicle Network Toolbox / XCP Communication / CAN
Simulink Real-Time / XCP / CAN

Description

The XCP CAN Data Stimulation block sends data to the selected server 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 server connection. The block displays the selected measurements as input ports.

The XCP communication blocks support Simulink accelerator mode and rapid accelerator mode. You can speed up the execution of Simulink models by using these modes. For more information about these simulation modes, see “Design Your Model for Effective Acceleration” (Simulink).

Parameters

`Config name` — Specify XCP CAN session name

select from list

Select the name of XCP configuration that you want to use. This 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. 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.

`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. The block is configured with the corresponding event number from the A2L.

The event time cycle does not control transmission of stimulation packets. The block stimulates each time it executes. For use in Simulink simulation, consider enabling simulation pacing to avoid free-running stimulation.

`All Measurements` — List all measurements available for event

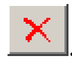
measurements list

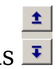
This list displays all measurements available for the selected event. Select the measurement that you want to use and click the add button,  to move it to the selected measurements. Hold the `Ctrl` key on your keyboard to select multiple measurements.

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 to clear your search.

`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.

`Block Input Settings` — Set the port input as Compu method conversion values or raw values

`Raw values as double (no Compu method conversion)` (default) | `Raw values (no Compu method conversion)` | `Physical values (apply Compu method conversion)`

This parameter enables support for XCP data types and dimensions as defined in the ASAP2 standard. The Block Input Settings parameter selects whether the port outputs Compu method conversion values or raw values. The options provide:

- `Physical values (apply Compu method conversion)` enables the physical-to-raw conversion of ECU measurement values. For this option, the block port settings are set either to `'double'` or `'string'`, based on the underlying Compu method conversion. For example, Compu method `IDENTICAL`, `LINEAR`, `RAT_FUNC`, `TAB_INTP`, and `TAB_NOINTP` port settings is `'double'` while Compu method `TAB_VERB` port settings is `'string'`. The maximum string length supported for Compu method conversion is 1024 as specified in the ASAM XIL specification.

  The FORM Compu method conversion is not supported. Simulink throws a warning for such a conversion and IDENTICAL conversion is applied to the underlying measurement. Also, only scalar measurement signals are supported for TAB_VERB conversion as Simulink only supports scalar strings.

  Selecting this option shows the physical units (if any) in front of the measurement name on the block mask. This physical unit is acquired from the A2L description of the measurement and Compu method. If the physical unit is not specified, only the measurement name is displayed.
- `Raw values (no Compu method conversion)` sets the port data type according to the type definition in the A2L file and supports up to three-dimensional XCP measurements in Simulink.
- `Raw values as double (no Compu method conversion)` sets the port data type as `double`, converting all internal measurement values. This selection supports up to three-dimensional XCP measurements in Simulink.

These ASAP2 data types are supported by corresponding Simulink port data types:

- `SBYTE`
- `UWORD`
- `SWORD`
- `ULONG`

- SLONG
- A_UINT64
- A_INT64
- FLOAT32_IEEE
- FLOAT64_IEEE

The dimension support in the block accommodates the different treatment of matrices by MATLAB and the ECU. The MATLAB default operation treats matrices as row-major matrices. An XCP measurement can have a LAYOUT as COLUMN_DIR or ROW_DIR . If a matrix measurement is COLUMN_DIR, the blocks rearrange the measurement in memory and ensure that the matrix (row X, col Y) in MATLAB refers to the same entry as (row X, col Y) on the ECU. The rearrangement causes matrix entries that are contiguous on the ECU to be noncontiguous in MATLAB and Simulink.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

See Also

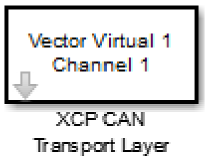
Blocks
XCP CAN Configuration | XCP CAN Data Acquisition | XCP CAN Transport Layer

Introduced in R2013a

XCP CAN Transport Layer

Transport XCP messages via CAN

Library: Vehicle Network Toolbox / XCP Communication / CAN
Simulink Real-Time / XCP / CAN



Description

The XCP CAN Transport Layer subsystem uses the specified device to transport and receive XCP messages.

Use this block with an XCP CAN Data Acquisition block to acquire and analyze specific XCP messages. Use this block with an XCP CAN 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.

Parameters

`Device` — CAN device
device list option

The CAN device, chosen from all connected CAN devices.

`Bus speed` — Speed of CAN bus
numeric

Speed of the CAN bus in bits per second. The default bus speed is the default assigned by the selected device.

`Sample time` — Simulation refresh rate
`0.01` (default) | numeric

Simulation refresh rate, specified as the sampling time of the block during simulation. This value defines the frequency at which the XCP CAN Transport Layer subsystem and the underlying blocks run during simulation. For information about simulation sample timing, see "Timing in Hardware Interface Models" on page 8-21. 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` simulation seconds.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

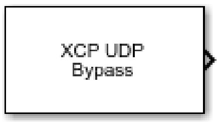
See Also

XCP CAN Configuration | XCP CAN Data Acquisition | XCP CAN Data Stimulation

Introduced in R2013a

XCP UDP Bypass

Connect the function-call outport to a function-call subsystem
Library: Vehicle Network Toolbox / XCP Communication / UDP
Simulink Real-Time / XCP / UDP



Description

The XCP UDP Bypass block connects the function-call outport to a function-call subsystem containing one data acquisition list. The block issues a function-call when the downstream data acquisition list has new data available.

Consider the downstream function-call subsystem as a bypass task:

In Simulink Real-Time, the bypass task is executed asynchronously with the assigned task priority.

In Simulink, the block checks for data acquisition data periodically at the assigned sample rate and executes the bypass task accordingly.

Ports

Output

`Function-call` — Function call for bypass
function call

Connects the function-call outport to a function-call subsystem containing one data acquistion list.

Parameters

`Task Priority` — Task priority in QNX Neutrino scheduler
191 (default) | integer

Select the task priority for the QNX Neutrino scheduler.

`Sample Time` — Sample time
-1 (default) | double

Select the sample time. For more information, see “Sample Times in Subsystems” (Simulink).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

See Also

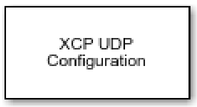
XCP UDP Configuration | XCP UDP Data Acquisition | XCP UDP Data Stimulation

Introduced in R2020b

XCP UDP Configuration

Configure XCP UDP server connection

Library: Vehicle Network Toolbox / XCP Communication / UDP
Simulink Real-Time / XCP / UDP



Description

The XCP UDP Configuration block uses the parameters specified in the A2L file and the ASAP2 database to establish an XCP server 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 server 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 UDP Configuration block, the model prompts you to add one.

The XCP UDP communication blocks support Simulink accelerator mode and rapid accelerator mode. You can speed up the execution of Simulink models by using these modes. For more information about these simulation modes, see “Design Your Model for Effective Acceleration” (Simulink).

Parameters

`Config name` — Specify XCP UDP session name

`'UDP_Config1'` (default)

Specify a unique name for your XCP session.

`A2L File` — Select an A2L file

file name

Click **Browse** to select an A2L file for your XCP session.

`Enable seed/key security` — Select that key required to establish connection

`'off'`

Select this option if your server requires a secure key to establish connection. Select a file that contains the seed/key definition to enable security.

`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 the seed and key security algorithm that unlocks an XCP server module. This parameter is available in Windows Desktop Simulation for Vehicle Network Toolbox.

`Output connection status` — Display connection status
`'off'`

Select this option to display the status of the connection to the server module. Selecting this option adds a new output port.

`Disable CTR error detection` — Disable CTR error detection scheme
`'on'` (default) | `'off'`

To detect missing packets, the block can check the counter value in each XCP packet header. When `'on'`, counter error detection for packet headers is disabled. When `'off'`, the counter **Error detection scheme** is enabled.

`Error detection scheme` — Select CTR error detection scheme
`One counter for all CTOs and DTOs` (default) | `Separate counters for (RES,ERR,EV,SERV) and (DAQ)` | `Separate counters for (RES,ERR), (EV,SERV) and (DAQ)`

To detect missing packets, the block can check the counter value in each XCP packet header and apply an error-detection scheme.

`Sample time` — Sample time of block
`-1` (default) | numeric

Enter the base sample time or a multiple of the base sample time. `-1` means that sample time is inherited. For information about simulation sample timing, see "Timing in Hardware Interface Models" on page 8-21.

`Local IP Address` — Maser IP address
`x.x.x.x`

Enter the IP address to which you want to connect.

`Local Port` — Client IP port
1–65535

The combination of **Local IP address** and **Local port** must be unique.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

See Also

Blocks
XCP UDP Data Acquisition | XCP UDP Data Stimulation | XCP UDP Bypass

Introduced in R2019a

XCP UDP Data Acquisition

Acquire selected measurements from configured server connection

Library: Vehicle Network Toolbox / XCP Communication / UDP
Simulink Real-Time / XCP / UDP

Description

The XCP UDP Data Acquisition block acquires data from the configured server connection based on the measurements that you select. 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 server connection. The block displays the selected measurements as output ports.

The XCP communication blocks support the use of Simulink accelerator mode and rapid accelerator mode. You can speed up the execution of Simulink models by using these modes. For more information on these simulation modes, see "Design Your Model for Effective Acceleration" (Simulink).

Parameters

`Config name` — Specify XCP UDP session name

select from list

Select the name of XCP configuration that you want to use. This 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. 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.

`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.

`All Measurements` — List all measurements available for event

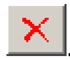
measurements list

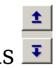
This list displays all measurements available for the selected event. Select the measurement that you want to use and click the add button,  to add it to the selected measurements. Hold the `Ctrl` key on your keyboard to select multiple measurements.

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 to clear your search.

`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.

`Block Output Settings` — Set the port output as Compu method conversion values or raw values

`Raw values as double (no Compu method conversion)` (default) | `Raw values (no Compu method conversion)` | `Physical values (apply Compu method conversion)`

This parameter enables support for XCP data types and dimensions as defined in the ASAP2 standard. The Block Output Settings parameter selects whether the port outputs Compu method conversion values or raw values. The options provide:

- `Physical values (apply Compu method conversion)` enables the raw-to-physical conversion of ECU measurement values. For this option, the block port settings are set either to `'double'` or `'string'`, based on the underlying Compu method conversion. For example, Compu method `IDENTICAL`, `LINEAR`, `RAT_FUNC`, `TAB_INTP`, and `TAB_NOINTP` port settings is `'double'` while Compu method `TAB_VERB` port settings is `'string'`. The maximum string length supported for Compu method conversion is 1024 as specified in the ASAM XIL specification.

  The FORM Compu method conversion is not supported. Simulink throws a warning for such a conversion and IDENTICAL conversion is applied to the underlying measurement. Also, only scalar measurement signals are supported for TAB_VERB conversion as Simulink only supports scalar strings.

  Selecting this option shows the physical units (if any) in front of the measurement name on the block mask. This physical unit is acquired from the A2L description of the measurement and Compu method. If the physical unit is not specified, only the measurement name is displayed.
- `Raw values (no Compu method conversion)` sets the port data type according to the type definition in the A2L file and supports up to three-dimensional XCP measurements in Simulink.
- `Raw values as double (no Compu method conversion)` sets the port data type as `double`, converting all internal measurement values. This selection supports up to three-dimensional XCP measurements in Simulink.

These ASAP2 data types are supported by corresponding Simulink port data types:

- `SBYTE`
- `UWORD`
- `SWORD`
- `ULONG`
- `SLONG`
- `A_UINT64`
- `A_INT64`

- FLOAT32_IEEE
- FLOAT64_IEEE

The dimension support in the block accommodates the different treatment of matrices by MATLAB and the ECU. The MATLAB default operation treats matrices as row-major matrices. An XCP measurement can have a LAYOUT as COLUMN_DIR or ROW_DIR . If a matrix measurement is COLUMN_DIR, the blocks rearrange the measurement in memory and ensure that the matrix (row X, col Y) in MATLAB refers to the same entry as (row X, col Y) on the ECU. The rearrangement causes matrix entries that are contiguous on the ECU to be noncontiguous in MATLAB and Simulink.

DAQ List Priority — Specify a priority value for server device driver
priority value

Specify a priority value as an integer from 0 to 255 for the server device driver to prioritize transmission of data packets. The server can accumulate XCP packets for lower priority DAQ lists before transmission to the client. A value of 255 has the highest priority. The SET_DAQ_LIST_MODE command communicates the **DAQ List Priority** value from client to server. This communication method differs from the specification of the Event Channel Priority property, which comes from the A2L file.

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 simulation seconds. For information about simulation sample timing, see “Timing in Hardware Interface Models” on page 8-21.

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

See Also

Blocks
XCP UDP Configuration | XCP UDP Data Stimulation | XCP UDP Bypass

Introduced in R2019a

XCP UDP Data Stimulation

Perform data stimulation on selected measurements

Library: Vehicle Network Toolbox / XCP Communication / UDP
Simulink Real-Time / XCP / UDP

Description

The XCP UDP Data Stimulation block sends data to the selected server connection for the event measurements that you select. 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 server connection. The block displays the selected measurements as input ports.

The XCP communication blocks support Simulink accelerator mode and rapid accelerator mode. You can speed up the execution of Simulink models by using these modes. For more information about these simulation modes, see “Design Your Model for Effective Acceleration” (Simulink).

Parameters

`Config name` — Specify XCP UDP session name

select from list

Select the name of XCP configuration that you want to use. This 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. 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.

`Event name` — Select an event

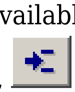
select from list

Select an event from the event list. The XCP UDP Configuration block uses the specified A2L file to populate the events list. The block is configured with the corresponding event number from the A2L.

The event time cycle does not control transmission of stimulation packets. The block stimulates each time it executes. For use in Simulink simulation, consider enabling simulation pacing to avoid free-running stimulation.

`All Measurements` — List all measurements available for event

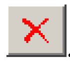
measurements list

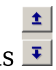
This list displays all measurements available for the selected event. Select the measurement that you want to use and click the add button,  to move it to the selected measurements. Hold the `Ctrl` key on your keyboard to select multiple measurements.

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.

`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.

`Block Input Settings` — Set the port input as Compu method conversion values or raw values

`Raw values as double (no Compu method conversion)` (default) | `Raw values (no Compu method conversion)` | `Physical values (apply Compu method conversion)`

This parameter enables support for XCP data types and dimensions as defined in the ASAP2 standard. The Block Input Settings parameter selects whether the port outputs Compu method conversion values or raw values. The options provide:

- `Physical values (apply Compu method conversion)` enables the physical-to-raw conversion of ECU measurement values. For this option, the block port settings are set either to `'double'` or `'string'`, based on the underlying Compu method conversion. For example, Compu method `IDENTICAL`, `LINEAR`, `RAT_FUNC`, `TAB_INTP`, and `TAB_NOINTP` port settings is `'double'` while Compu method `TAB_VERB` port settings is `'string'`. The maximum string length supported for Compu method conversion is 1024 as specified in the ASAM XIL specification.

  The FORM Compu method conversion is not supported. Simulink throws a warning for such a conversion and IDENTICAL conversion is applied to the underlying measurement. Also, only scalar measurement signals are supported for TAB_VERB conversion as Simulink only supports scalar strings.

  Selecting this option shows the physical units (if any) in front of the measurement name on the block mask. This physical unit is acquired from the A2L description of the measurement and Compu method. If the physical unit is not specified, only the measurement name is displayed.
- `Raw values (no Compu method conversion)` sets the port data type according to the type definition in the A2L file and supports up to three-dimensional XCP measurements in Simulink.
- `Raw values as double (no Compu method conversion)` sets the port data type as `double`, converting all internal measurement values. This selection supports up to three-dimensional XCP measurements in Simulink.

These ASAP2 data types are supported by corresponding Simulink port data types:

- `SBYTE`
- `UWORD`
- `SWORD`
- `ULONG`

- `SLONG`
- `A_UINT64`
- `A_INT64`
- `FLOAT32_IEEE`
- `FLOAT64_IEEE`

The dimension support in the block accommodates the different treatment of matrices by MATLAB and the ECU. The MATLAB default operation treats matrices as row-major matrices. An XCP measurement can have a `LAYOUT` as `COLUMN_DIR` or `ROW_DIR` . If a matrix measurement is `COLUMN_DIR`, the blocks rearrange the measurement in memory and ensure that the matrix (row X, col Y) in MATLAB refers to the same entry as (row X, col Y) on the ECU. The rearrangement causes matrix entries that are contiguous on the ECU to be noncontiguous in MATLAB and Simulink.

`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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The XCP communication blocks support code generation with limited deployment portability that runs only on the host computer or Simulink Real-Time targets.

Code generation requires a C++ compiler that is compatible with the code generation target. For more information, see Supported and Compatible Compilers.

See Also

Blocks
XCP UDP Configuration | XCP UDP Data Acquisition | XCP UDP Bypass

Introduced in R2019a

14

Vehicle Network Toolbox Examples

Get Started with CAN Communication in MATLAB

This example shows you how to use CAN channels to transmit and receive CAN messages. It uses MathWorks virtual CAN channels connected in a loopback configuration.

Create a Receiving Channel

Create a CAN channel using `canChannel` to receive messages by specifying the vendor name, device name, and device channel index.

```
rxCh = canChannel("MathWorks", "Virtual 1", 2);
```

Inspect the Channel

Use the `get` command to obtain more detailed information on all channel properties and their current values.

```
get(rxCh)
```

```
        ArbitrationBusSpeed: []
               DataBusSpeed: []
          ReceiveErrorCount: 0
         TransmitErrorCount: 0
       InitializationAccess: 1
           InitialTimestamp: [0x0 datetime]
                 SilentMode: 0
           TransceiverState: 'N/A'
                   BusSpeed: 500000
               NumOfSamples: []
                        SJW: []
                      TSEG1: []
                      TSEG2: []
                  BusStatus: 'N/A'
            TransceiverName: 'N/A'
                   Database: []
         MessageReceivedFcn: []
    MessageReceivedFcnCount: 1
                   UserData: []
              FilterHistory: 'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
           MessagesReceived: 0
        MessagesTransmitted: 0
                    Running: 0
                     Device: 'Virtual 1'
         DeviceChannelIndex: 2
         DeviceSerialNumber: 0
               DeviceVendor: 'MathWorks'
               ProtocolMode: 'CAN'
          MessagesAvailable: 0
```

Start the Channel

Use the `start` command to set the channel online.

```
start(rxCh);
```

Transmit Messages

The example function `generateMsgs` creates CAN messages using `canMessage` and transmits them using `transmit` at various periodic rates. It generates traffic on the CAN bus for demonstration purposes.

```
type generateMsgs
```

```
function generateMsgs()
% generateMsgs Creates and transmits CAN messages for demo purposes.
%
%   generateMsgs periodically transmits multiple CAN messages at various
%   periodic rates with changing message data.
%

% Copyright 2008-2016 The MathWorks, Inc.

    % Create the messages to send using the canMessage function. The
    % identifier, an indication of standard or extended type, and the data
    % length is given for each message.
    msgTx100 = canMessage(100, false, 0);
    msgTx200 = canMessage(200, false, 2);
    msgTx400 = canMessage(400, false, 4);
    msgTx600 = canMessage(600, false, 6);
    msgTx800 = canMessage(800, false, 8);

    % Create a CAN channel on which to transmit.
    txCh = canChannel('MathWorks', 'Virtual 1', 1);

    % Register each message on the channel at a specified periodic rate.
    transmitPeriodic(txCh, msgTx100, 'On', 0.500);
    transmitPeriodic(txCh, msgTx200, 'On', 0.250);
    transmitPeriodic(txCh, msgTx400, 'On', 0.125);
    transmitPeriodic(txCh, msgTx600, 'On', 0.050);
    transmitPeriodic(txCh, msgTx800, 'On', 0.025);

    % Start the CAN channel.
    start(txCh);

    % Run for several seconds incrementing the message data regularly.
    for ii = 1:50
        % Increment the message data bytes.
        msgTx200.Data = msgTx200.Data + 1;
        msgTx400.Data = msgTx400.Data + 1;
        msgTx600.Data = msgTx600.Data + 1;
        msgTx800.Data = msgTx800.Data + 1;

        % Wait for a time period.
        pause(0.100);
    end

    % Stop the CAN channel.
    stop(txCh);
end
```

Run the `generateMsgs` function to transmit messages for the example.

```
generateMsgs();
```

Receive Messages

Once `generateMsgs` completes, receive all available messages from the channel using the `receive` function.

```
rxMsg = receive(rxCh, Inf, "OutputFormat", "timetable");
```

Use `head` to extract the first few rows of received messages for preview.

```
head(rxMsg)
```

```
ans=8×8 timetable
        Time        ID     Extended       Name              Data            Length      Signals
    ___________    ___    ________    __________    ____________________    ______    ____________

    0.21832 sec    100     false      {0x0 char}    {1x0 uint8        }       0       {0x0 struct
    0.21832 sec    200     false      {0x0 char}    {[             0 0]}      2       {0x0 struct
    0.21833 sec    400     false      {0x0 char}    {[         0 0 0 0]}      4       {0x0 struct
    0.21834 sec    600     false      {0x0 char}    {[     0 0 0 0 0 0]}      6       {0x0 struct
    0.21834 sec    800     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}       8       {0x0 struct
    0.2484 sec     800     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}       8       {0x0 struct
    0.27821 sec    600     false      {0x0 char}    {[     1 1 1 1 1 1]}      6       {0x0 struct
    0.27822 sec    800     false      {0x0 char}    {[1 1 1 1 1 1 1 1]}       8       {0x0 struct
```

Stop the Channel

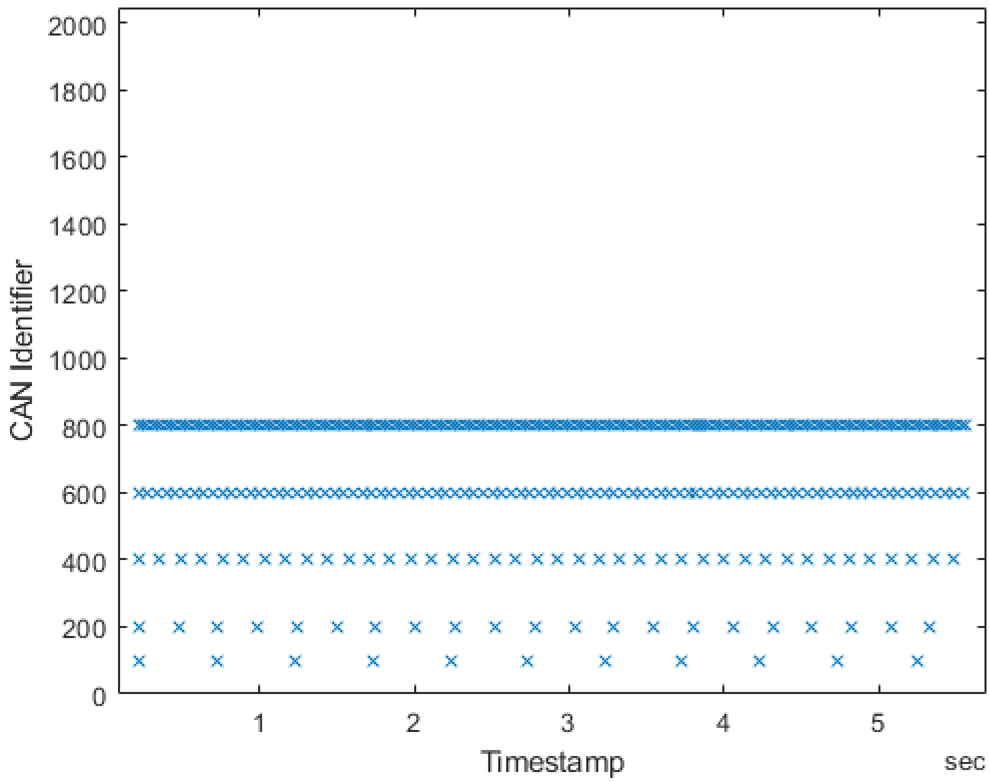
Use the `stop` command to set the channel offline.

```
stop(rxCh);
```

Analyze Received Messages

MATLAB provides a powerful environment for performing analysis on CAN messages. The `plot` command can create a scatter plot with message timestamps and identifiers to provide an overview of when certain messages occurred on the network.

```
plot(rxMsg.Time, rxMsg.ID, "x")
ylim([0 2047])
xlabel("Timestamp")
ylabel("CAN Identifier")
```


2000
1800
1600
1400
1200
1000
800
600
400
200
0
CAN Identifier
1
2
3
4
5
Timestamp
sec

Get Started with CAN FD Communication in MATLAB

This example shows you how to use CAN FD channels to transmit and receive CAN FD messages. It uses MathWorks virtual CAN FD channels connected in a loopback configuration.

View Available CAN FD Channels

Use `canFDChannelList` to see all available device channels supporting CAN FD.

```
canFDChannelList
```

```
ans=2×6 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"
```

Create Transmitting and Receiving Channels

Use `canFDChannel` with device details specified to create CAN FD channels for transmitting and receiving messages.

```
txCh = canFDChannel("MathWorks", "Virtual 1", 1)
```

```
txCh = 
  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: []
```

```
rxCh = canFDChannel("MathWorks", "Virtual 1", 2);
```

Configure Bus Speed

CAN FD channels require setting of bus speed before going online. Both the arbitration and data phase speeds are configured using `configBusSpeed`.

```
configBusSpeed(txCh, 500000, 1000000);
configBusSpeed(rxCh, 500000, 1000000);
```

Open the DBC-File

Use `canDatabase` to open the database file that contains definitions of CAN FD messages and signals.

```
db = canDatabase("CANFDExample.dbc")
```

```
db = 
  Database with properties:

             Name: 'CANFDExample'
             Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex36915890\CANFDExam
            Nodes: {}
         NodeInfo: [0x0 struct]
         Messages: {'CANFDMessage'}
      MessageInfo: [1x1 struct]
       Attributes: {2x1 cell}
    AttributeInfo: [2x1 struct]
         UserData: []
```

Attach the database directly to the receiving channel. Definitions from the DBC-files are applied automatically to decode incoming messages and signals.

```
rxCh.Database = db;
```

Start the Channels

Use the `start` command to set the channels online.

```
start(txCh);
start(rxCh);
```

Create CAN FD Messages

Create CAN FD messages using the `canFDMessage` function.

```
msg1 = canFDMessage(500, false, 12)
```

```
msg1 = 
  Message with properties:

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

```
          Name: ''

   Data Details
     Timestamp: 0
          Data: [0 0 0 0 0 0 0 0 0 0 0 0]
       Signals: []
        Length: 12
           DLC: 9

   Protocol Flags
           BRS: 0
           ESI: 0
         Error: 0

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

```
msg2 = canFDMessage(1000, false, 24);
msg3 = canFDMessage(1500, false, 64);
```

To engage the bit rate switch capability of CAN FD, set the BRS property of the messages.

```
msg1.BRS = true;
msg2.BRS = true;
msg3.BRS = true;
```

CAN FD messages can also be created using a database. The database defines if a message is CAN or CAN FD as well as the BRS status.

```
msg4 = canFDMessage(db, "CANFDMessage")
```

```
msg4 = 
  Message with properties:

   Message Identification
    ProtocolMode: 'CAN FD'
              ID: 1
        Extended: 0
            Name: 'CANFDMessage'

   Data Details
     Timestamp: 0
          Data: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... ]
       Signals: []
        Length: 48
           DLC: 14

   Protocol Flags
           BRS: 1
           ESI: 0
         Error: 0

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

Transmit Messages

Use `transmit` to send the created messages from the transmitting channel.

```
transmit(txCh, [msg1 msg2 msg3 msg4])
```

Receive Messages

Receive the messages from the receiving channel using the `receive` function. The default return type for CAN FD channels is a timetable containing information specific to the received CAN FD messages.

```
rxMsg = receive(rxCh, Inf)
```

```
rxMsg=4×12 timetable
       Time         ID     Extended          Name          ProtocolMode
    ___________    ____    ________    _________________    ____________    _______________________

    0.1969 sec      500     false      {0x0 char      }     {'CAN FD'}     {[
    0.19691 sec    1000     false      {0x0 char      }     {'CAN FD'}     {[
    0.19691 sec    1500     false      {0x0 char      }     {'CAN FD'}     {[0 0 0 0 0 0 0 0 0 0
    0.19691 sec       1     false      {'CANFDMessage'}     {'CAN FD'}     {[0 0 0 0 0 0 0 0 0 0
```

Stop the Channels

Use the `stop` command to set the channels offline.

```
stop(txCh);
stop(rxCh);
```

Close the DBC-File

Close access to the DBC-file by clearing its variable from the workspace.

```
clear db
```

Use Message Reception Callback Functions in CAN Communication

This example shows you how to use a callback function to process messages received from a CAN channel. It uses MathWorks virtual CAN channels connected in a loopback configuration. This example describes the workflow for a CAN network, but the concept demonstrated also applies to a CAN FD network.

Create a Receiving Channel

Create a CAN channel using `canChannel` to receive messages by specifying the vendor name, device name, and device channel index.

```
rxCh = canChannel("MathWorks", "Virtual 1", 2)
```

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

Configure the Callback Function

Set the callback function to run when a required number of messages are available on the channel.

```
rxCh.MessageReceivedFcn = @receivingFcn;
```

Configure the Message Received Count

Specify the number of messages required in the channel before the callback function is triggered.

```
rxCh.MessageReceivedFcnCount = 30;
```

Implement the Callback Function

The example callback function receives all available messages from the channel and plots the CAN identifiers against their timestamps on each execution.

```
type receivingFcn
```

```
function receivingFcn(rxCh)
% RECEIVINGFCN A CAN channel message receive callback function.
%
%   This is a callback function used to receive CAN message. It receives
%   messages from the channel RXCH and plots the result.
%

% Copyright 2009-2016 The MathWorks, Inc.

    % Receive all available messages.
    rxMsg = receive(rxCh, Inf, 'OutputFormat', 'timetable');

    % Plot the signal values against their message timestamps.
    plot(rxMsg.Time, rxMsg.ID, 'x');
    ylim([0 2047])
    xlabel('Timestamp');
    ylabel('CAN Identifier');
    hold all;
end
```

Start the Channel

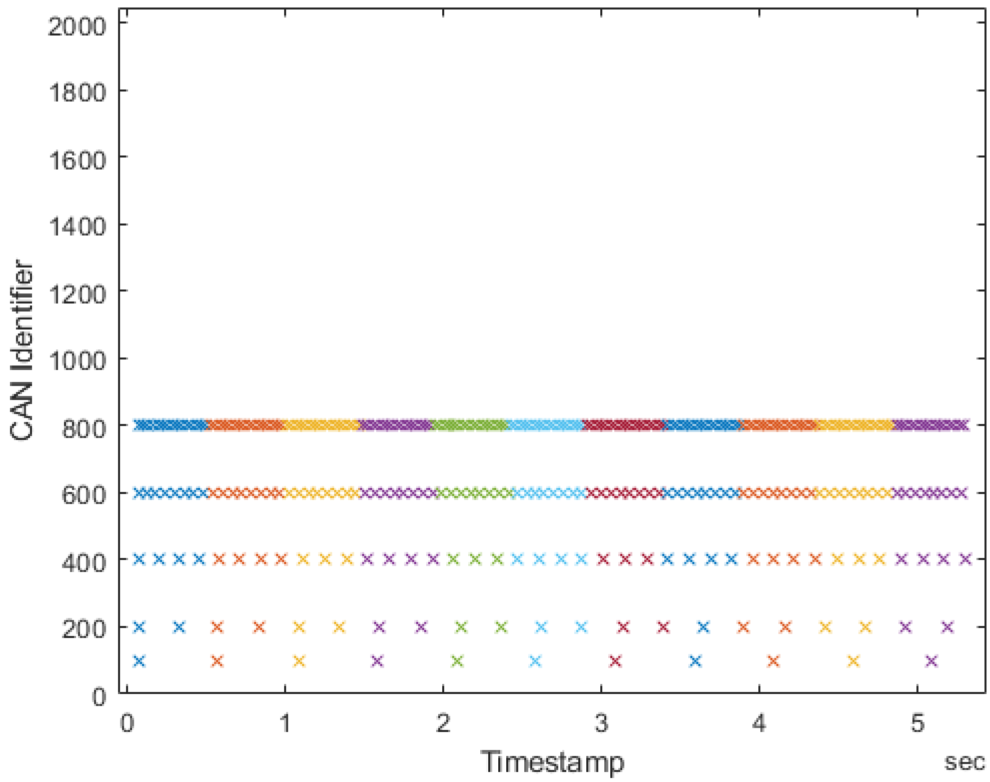
Use the `start` command to set the channel online.

```
start(rxCh);
```

Execute the Callback Function

The function `generateMsgs` creates CAN messages and transmits them at various periodic rates to create traffic on the CAN bus. As the messages are transmitted, the callback function executes each time the threshold specified by property `MessageReceivedFcnCount` is met.

```
generateMsgs();
```



Inspect the Remaining Messages

Display the `MessagesAvailable` property of the channel to see the number of remaining messages. Since the available message count is below the specified threshold, more messages are required to trigger the callback another time.

```
rxCh.MessagesAvailable
```

```
ans = 31
```

Stop the Channel

Use the `stop` command to set the channel offline.

```
stop(rxCh);
```

Use Message Filters in CAN Communication

This example shows you how to use CAN message filters to allow only messages that contain specified identifiers to pass through a channel. It uses MathWorks virtual CAN channels connected in a loopback configuration. This example describes the workflow for a CAN network, but the concept demonstrated also applies to a CAN FD network.

Create Transmitting and Receiving Channels

Create one channel for transmitting messages and another channel for receiving. Filters are set later on the receiving channel.

```
txCh = canChannel("MathWorks", "Virtual 1", 1)
```

```
txCh = 
  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: []
```

```
rxCh = canChannel("MathWorks", "Virtual 1", 2)
```

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

Create Messages

Create a few messages to be sent to the receiving channel multiple times throughout the example. Note that one message has an extended identifier.

```
txMsgs(1) = canMessage(250, false, 8);
txMsgs(2) = canMessage(500, false, 8);
txMsgs(3) = canMessage(1000, false, 8);
txMsgs(4) = canMessage(1500, true, 8);
txMsgs(5) = canMessage(2000, false, 8);
```

Receive Messages with No Filter

Set the channels online, transmit the messages on one channel, and receive on the other. Note that all messages sent are received. By default, a newly created channel with no filter configured receives all standard and extended identifiers.

```
start(rxCh);
start(txCh);
transmit(txCh, txMsgs);
pause(0.5);
rxMsgs1 = receive(rxCh, Inf, "OutputFormat", "timetable")
```

```
rxMsgs1=5×8 timetable
        Time          ID      Extended       Name               Data             Length      Signals
```

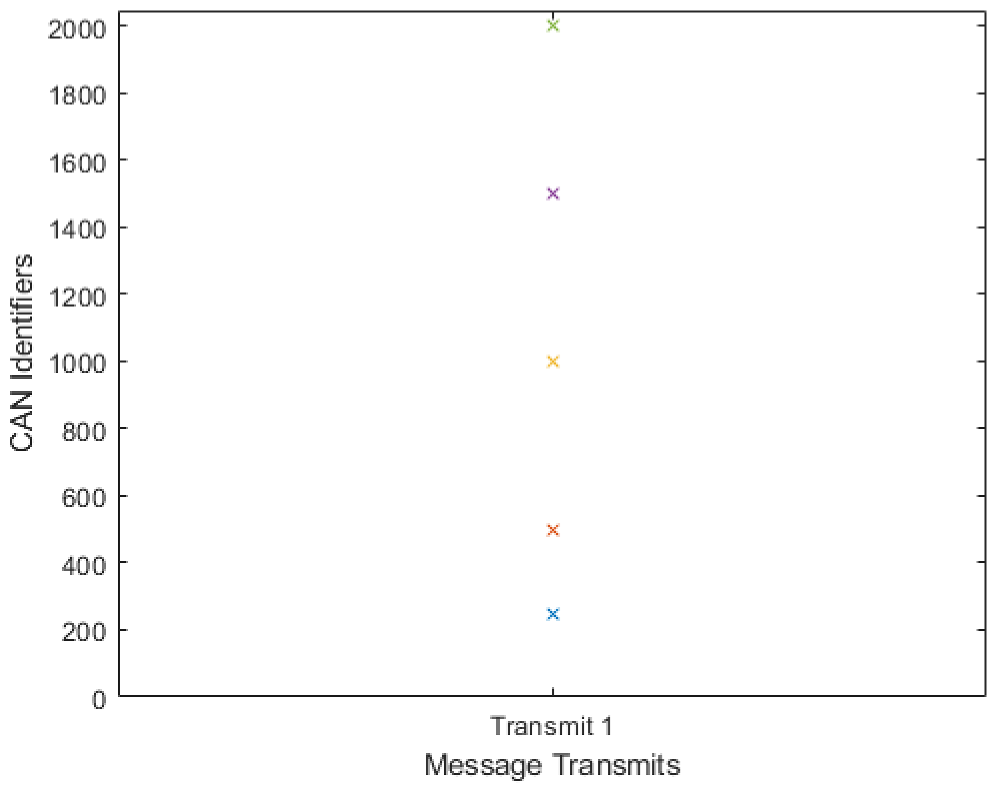
```
    ____________     ____     ________     __________     ___________________     ______     __________

    0.071294 sec      250     false       {0x0 char}     {[0 0 0 0 0 0 0 0]}        8        {0x0 struc
    0.071296 sec      500     false       {0x0 char}     {[0 0 0 0 0 0 0 0]}        8        {0x0 struc
    0.071298 sec     1000     false       {0x0 char}     {[0 0 0 0 0 0 0 0]}        8        {0x0 struc
    0.071301 sec     1500     true        {0x0 char}     {[0 0 0 0 0 0 0 0]}        8        {0x0 struc
    0.071303 sec     2000     false       {0x0 char}     {[0 0 0 0 0 0 0 0]}        8        {0x0 struc
```

Stop both receiving and transmitting channels.

```
stop(rxCh);
stop(txCh);
```

Plot identifiers of the received messages to see that all messages sent are received by the channel.

```
plot(1, rxMsgs1.ID, "x")
h_gca = gca;
h_gca.XTick = 0:1:2;
h_gca.XTickLabel = ["", "Transmit 1", ""];
axis([0 2 0 2047])
xlabel("Message Transmits")
ylabel("CAN Identifiers")
```



Receive Messages with Filters Configured by Identifier

Use the `filterAllowOnly` command to allow only specified messages by CAN identifier and identifier type. Configure the receiving channel to only receive messages with standard identifiers 500 and 2000.

```
filterAllowOnly(rxCh, [500 2000], "Standard");
```

Display the `FilterHistory` property of the channel to view the configured state of the message filters.

```
rxCh.FilterHistory
```

```
ans = 
'Standard ID Filter: Allow Only | Extended ID Filter: Allow All'
```

Transmit the messages again to the receiving channel. Note that fewer messages are received this time.

```
start(rxCh);
start(txCh);
transmit(txCh, txMsgs);
pause(0.5);
rxMsgs2 = receive(rxCh, Inf, "OutputFormat", "timetable")
```

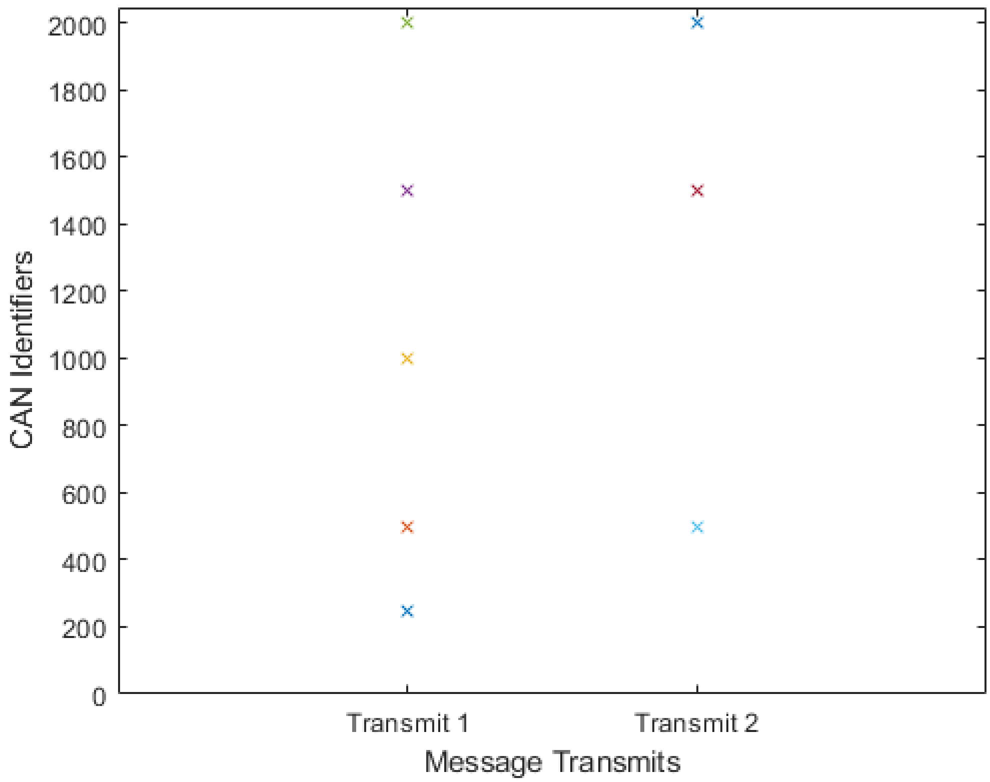
```
rxMsgs2=3×8 timetable
       Time         ID     Extended       Name              Data             Length      Signals
    ___________    ____    ________    __________    ___________________    ______    ____________

    0.10398 sec     500     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 struc
    0.10399 sec    1500     true       {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 struc
    0.10399 sec    2000     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 struc
```

Stop both receiving and transmitting channels.

```
stop(rxCh);
stop(txCh);
```

Add the newly received data to the plot to see which messages passed the filters. The message with extended identifier 1500 is not blocked by the filter because the filter was only configured for standard identifiers.

```
plot(1, rxMsgs1.ID, "x", 2, rxMsgs2.ID, "x");
h_gca = gca;
h_gca.XTick = 0:1:3;
h_gca.XTickLabel = ["", "Transmit 1", "Transmit 2", ""];
axis([0 3 0 2047])
xlabel("Message Transmits")
ylabel("CAN Identifiers")
```



Reset the Message Filters

Reset the message filters to the default states with the `filterAllowAll` command so that all standard identifiers are allowed.

```
filterAllowAll(rxCh, "Standard");
```

Display the `FilterHistory` property of the channel to view the configured state of the message filters.

```
rxCh.FilterHistory
```

```
ans =
'Standard ID Filter: Allow All | Extended ID Filter: Allow All'
```

Transmit and receive for a third time to see that all messages are once again passing through the filters and received by the receiving channel.

```
start(rxCh);
start(txCh);
transmit(txCh, txMsgs);
pause(0.5);
rxMsgs3 = receive(rxCh, Inf, "OutputFormat", "timetable")
```

```
rxMsgs3=5×8 timetable
        Time          ID      Extended       Name                  Data                Length      Signals
    ____________     ____     ________    __________    ______________________    ______      __________
```

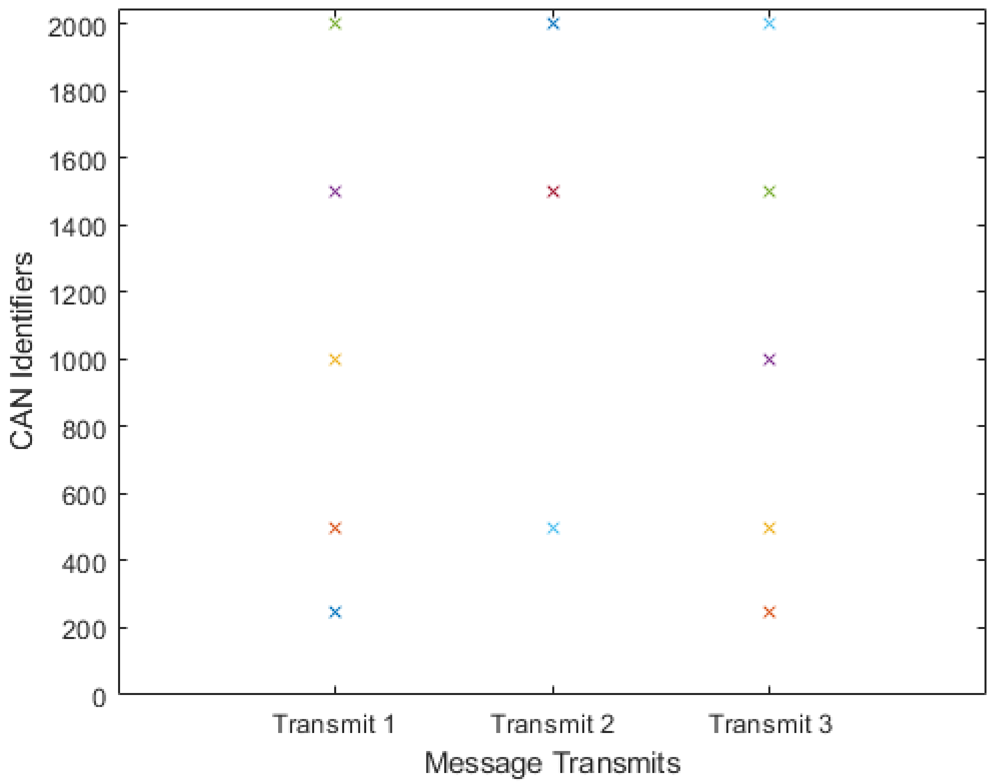
```
0.079855 sec     250     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 stru
0.079856 sec     500     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 stru
0.079857 sec    1000     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 stru
0.079861 sec    1500     true       {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 stru
0.079862 sec    2000     false      {0x0 char}    {[0 0 0 0 0 0 0 0]}      8       {0x0 stru
```

Stop both receiving and transmitting channels.

```
stop(rxCh);
stop(txCh);
```

With the new data added to the plot, observe that the first and third transmits are identical as the message filters are fully open in both cases.

```
plot(1, rxMsgs1.ID, "x", 2, rxMsgs2.ID, "x", 3, rxMsgs3.ID, "x")
h_gca = gca;
h_gca.XTick = 0:1:4;
h_gca.XTickLabel = ["", "Transmit 1", "Transmit 2", "Transmit 3", ""];
axis([0 4 0 2047])
xlabel("Message Transmits")
ylabel("CAN Identifiers")
```



Receive Messages with Filters Configured by Name

The `filterAllowOnly` command can also filter messages by name when using a DBC-file. Allow only messages with name `EngineMsg`.

```
db = canDatabase("demoVNT_CANdbFiles.dbc");
rxCh.Database = db;
filterAllowOnly(rxCh, "EngineMsg");
rxCh.FilterHistory
```

```
ans = 
'Standard ID Filter: Allow Only | Extended ID Filter: Allow All'
```

Block All Messages of a Specific Identifier Type

The `filterBlockAll` command allows you to easily set the filters to block all messages of either standard or extended identifier type. Block all messages with extended identifiers.

```
filterBlockAll(rxCh, "Extended");
rxCh.FilterHistory
```

```
ans = 
'Standard ID Filter: Allow Only | Extended ID Filter: Block All'
```

Stop the Channels

Stop both receiving and transmitting channels and clear them from the workspace.

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

Close the DBC-File

Close access to the DBC-file by clearing its variable from the workspace.

```
clear db
```

Use DBC-Files in CAN Communication

This example shows you how to create, receive and process messages using information stored in DBC-files. This example describes the workflow for a CAN network, but the concept demonstrated also applies to a CAN FD network.

Open the DBC-File

Open file `demoVNT_CANdbFiles.dbc` using `canDatabase`.

```
db = canDatabase("demoVNT_CANdbFiles.dbc")
```

```
db = 
  Database with properties:

             Name: 'demoVNT_CANdbFiles'
             Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex80654288\demoVNT_C
            Nodes: {}
         NodeInfo: [0x0 struct]
         Messages: {5x1 cell}
      MessageInfo: [5x1 struct]
       Attributes: {}
    AttributeInfo: [0x0 struct]
         UserData: []
```

Examine the `Messages` property to see the names of all messages defined in this file.

```
db.Messages
```

```
ans = 5x1 cell
    {'DoorControlMsg'   }
    {'EngineMsg'        }
    {'SunroofControlMsg'}
    {'TransmissionMsg'  }
    {'WindowControlMsg' }
```

View Message Information

Use `messageInfo` to view information for message `EngineMsg`, 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=5×1 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 information for signal `EngineRPM` in message `EngineMsg`, 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 message `EngineMsg` at once.

```
signalInfo(db, "EngineMsg")
```

```
ans=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
      ⋮
```

Create a Message Using Database Definitions

Create a new message by specifying the database and the message name `EngineMsg` to have the database definition applied. CAN signals in this message 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 and unpack data from the message.

```
msgEngineInfo.Signals
```

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

Change Signal Information

Write directly to the signal EngineRPM to change its value.

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

```
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: []
```

Read the current signal values back and note that EngineRPM has been updated with the written value.

```
msgEngineInfo.Signals
```

```
ans = struct with fields:
    VehicleSpeed: 0
       EngineRPM: 5.5003e+03
```

When a value is written directly to the signal, it is translated, scaled, and packed into the message data using the database definition. Note the value change in the Data property after a new value is written to the VehicleSpeed signal.

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

```
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: []
```

```
msgEngineInfo.Signals
```

```
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");
```

View the first few rows of received messages.

```
head(rxMsg)
```

```
ans=8×8 timetable
        Time         ID     Extended            Name                        Data               Leng
    _____________    ___    ________    _____________________    ________________________    _____

    0.087502 sec     100     false      {'EngineMsg'        }    {[     0 0 0 0 0 0 0 0]}      8
    0.087506 sec     200     false      {'TransmissionMsg'  }    {[     0 0 0 0 0 0 0 0]}      8
    0.08751 sec      400     false      {'DoorControlMsg'   }    {[     0 0 0 0 0 0 0 0]}      8
    0.087513 sec     600     false      {'WindowControlMsg' }    {[             0 0 0 0]}      4
    0.087516 sec     800     false      {'SunroofControlMsg'}    {[                 0 0]}      2
    0.11855 sec      100     false      {'EngineMsg'        }    {[172 129 0 0 50 0 0 0]}      8
    0.1485 sec       100     false      {'EngineMsg'        }    {[172 129 0 0 50 0 0 0]}      8
    0.14852 sec      200     false      {'TransmissionMsg'  }    {[     4 0 0 0 0 0 0 0]}      8
```

Stop the receiving 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, :)
```

```
ans=1×8 timetable
       Time         ID     Extended         Name                     Data                Length    S
    ___________    ___    ________    _____________    _________________________    ______    ___
```

```
    0.20849 sec    100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
```

```
rxMsg.Signals{10}
```

```
ans = struct with fields:
    VehicleSpeed: 50
       EngineRPM: 3.5696e+03
```

Extract All Instances of a Specified Message

Extract all instances of message `EngineMsg`.

```
allMsgEngine = rxMsg(strcmpi("EngineMsg", rxMsg.Name), :);
```

View the first few instances of this specific message.

```
head(allMsgEngine)
```

```
ans=8×8 timetable
       Time         ID     Extended        Name                   Data              Length      S
    ____________    ___    ________    _____________    ________________________    ______    ___

    0.087502 sec    100     false      {'EngineMsg'}    {[     0 0 0 0 0 0 0 0]}      8       {1x
    0.11855 sec     100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
    0.1485 sec      100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
    0.17844 sec     100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
    0.20849 sec     100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
    0.23845 sec     100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
    0.26846 sec     100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
    0.29837 sec     100     false      {'EngineMsg'}    {[172 129 0 0 50 0 0 0]}      8       {1x
```

Plot Physical Signal Values

Use `canSignalTimetable` to repackage signal data from message `EngineMsg` into a signal timetable.

```
signalTimetable = canSignalTimetable(rxMsg, "EngineMsg");
```

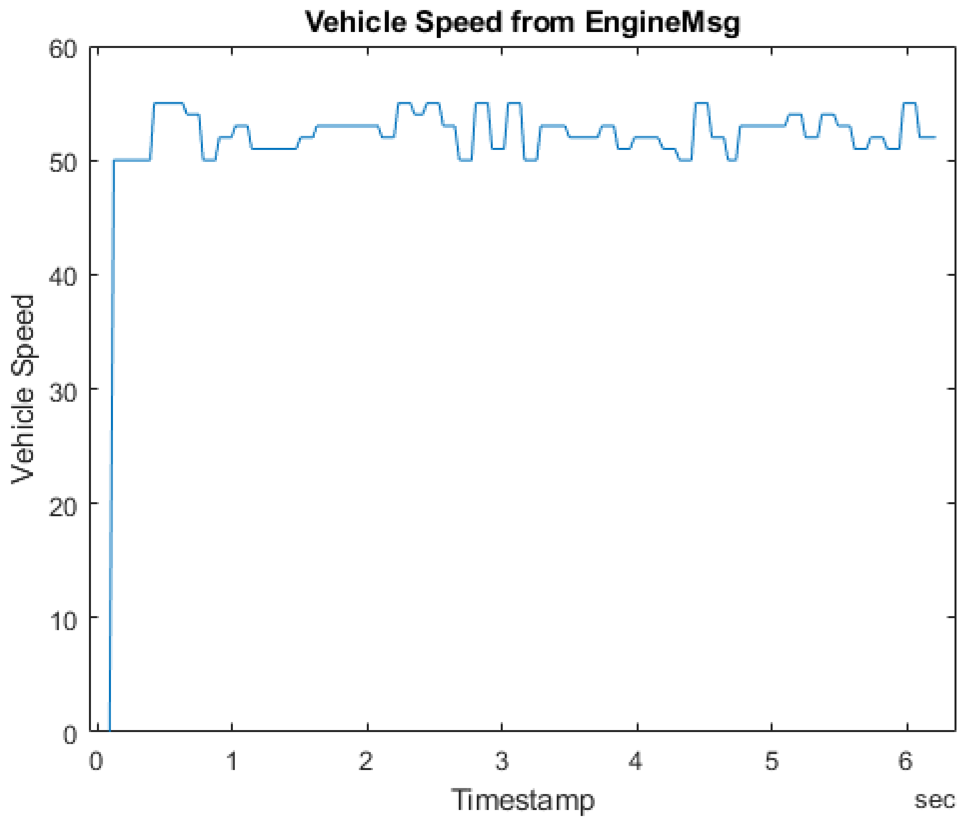
View the first few rows of the signal timetable.

```
head(signalTimetable)
```

```
ans=8×2 timetable
       Time         VehicleSpeed    EngineRPM
    ____________    ____________    _________

    0.087502 sec          0              250
    0.11855 sec          50           3569.6
    0.1485 sec           50           3569.6
    0.17844 sec          50           3569.6
    0.20849 sec          50           3569.6
    0.23845 sec          50           3569.6
    0.26846 sec          50           3569.6
    0.29837 sec          50           3569.6
```

Plot the values of signal `VehicleSpeed` over time.

```
plot(signalTimetable.Time, signalTimetable.VehicleSpeed)
title("Vehicle Speed from EngineMsg", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Vehicle Speed")
```



Close the DBC-File

Close access to the DBC-file by clearing its variable from the workspace.

```
clear db
```

Periodic CAN Communication in MATLAB

This example shows you how to how to configure CAN channels and messages for transmit messages periodically. It uses MathWorks virtual CAN channels connected in a loopback configuration.

As this example is based on sending and receiving CAN messages on a virtual network, running CAN Explorer in conjunction may provide a more complete understanding of what the code is doing. To run CAN Explorer, open and configure it to use the same interface as the receiving channel of the example. Make sure to start CAN Explorer before beginning to run the example in order to see all of the messages as they occur.

This example describes the workflow for a CAN network, but the concept demonstrated also applies to a CAN FD network.

Create the CAN Channels

Create CAN channels for message transmission and reception.

```
txCh = canChannel("MathWorks", "Virtual 1", 1);
rxCh = canChannel("MathWorks", "Virtual 1", 2);
```

Open the DBC-file that contains message and signal definitions, and attach it to both CAN channels.

```
db = canDatabase("CANDatabasePeriodic.dbc");
txCh.Database = db;
rxCh.Database = db;
```

Create the CAN Messages

Create CAN messages `EngineMsg` and `TransmissionMsg` using the database information.

```
msgFast = canMessage(db, "EngineMsg")
```

```
msgFast = 
  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: []
```

```
msgSlow = canMessage(db, "TransmissionMsg")
```

```
msgSlow = 
  Message with properties:

   Message Identification
    ProtocolMode: 'CAN'
              ID: 200
        Extended: 0
            Name: 'TransmissionMsg'

   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: []
```

Configure Messages for Periodic Transmission

To enable a message for periodic transmission, use the `transmitPeriodic` command specifying the transmitting channel, the message to register on the channel, a state value, and the periodic rate.

```
transmitPeriodic(txCh, msgFast, "On", 0.100);
transmitPeriodic(txCh, msgSlow, "On", 0.500);
```

Start the Periodic Transmission

Start the receiving channel.

```
start(rxCh);
```

Start the transmitting channel with periodic transmission configured in the previous step. Period transmission begins immediately. Allow the channels to run for two seconds.

```
start(txCh);
pause(2);
```

Update Transmitted Data

To update the live messages or signal data transmitted onto the CAN bus, write new values directly to the `VehicleSpeed` signal in message `EngineMsg`.

```
msgFast.Signals.VehicleSpeed = 60;
pause(1);
msgFast.Signals.VehicleSpeed = 65;
pause(1);
msgFast.Signals.VehicleSpeed = 70;
pause(1);
```

Alternatively, you can write new values to the `Data` property of the created messages.

Receive the Messages

Stop the CAN channels and receive all periodically transmitted messages for analysis.

```
stop(txCh);
stop(rxCh);
msgRx = receive(rxCh, Inf, "OutputFormat", "timetable");
```

View the first few rows of the received messages using the `head` function.

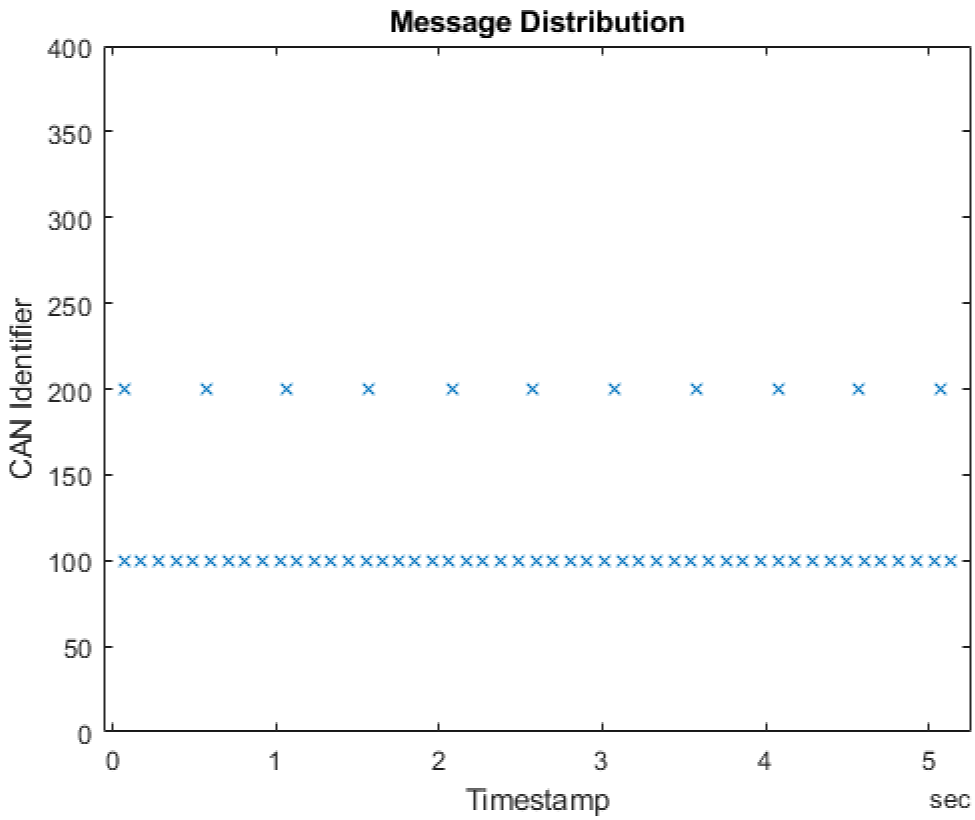
```
head(msgRx)
```

```
ans=8×8 timetable
        Time        ID     Extended          Name                   Data             Length
    ____________    ___    ________    ___________________    ___________________    ______    _

    0.070266 sec    100     false      {'EngineMsg'      }    {[0 0 0 0 0 0 0 0]}      8       {
    0.070272 sec    200     false      {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {
    0.17624 sec     100     false      {'EngineMsg'      }    {[0 0 0 0 0 0 0 0]}      8       {
    0.28219 sec     100     false      {'EngineMsg'      }    {[0 0 0 0 0 0 0 0]}      8       {
    0.38811 sec     100     false      {'EngineMsg'      }    {[0 0 0 0 0 0 0 0]}      8       {
    0.49409 sec     100     false      {'EngineMsg'      }    {[0 0 0 0 0 0 0 0]}      8       {
    0.56905 sec     200     false      {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {
    0.59903 sec     100     false      {'EngineMsg'      }    {[0 0 0 0 0 0 0 0]}      8       {
```

Analyze the Behavior of Periodic Transmission

Analyze the distribution of messages by plotting the identifiers of each received message against their timestamps. Note the difference between how often the two messages appear according to the configured periodic rates.

```
plot(msgRx.Time, msgRx.ID, "x")
ylim([0 400])
title("Message Distribution", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("CAN Identifier")
```



For further analysis, separate the two messages into individual timetables.

```
msgRxFast = msgRx(strcmpi("EngineMsg", msgRx.Name), :);
head(msgRxFast)
```

```
ans=8×8 timetable
        Time          ID     Extended        Name                 Data              Length      Signa
    _____________    ___    ________    _____________    ____________________    ______    _______

    0.070266 sec     100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.17624 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.28219 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.38811 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.49409 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.59903 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.70499 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
    0.80992 sec      100     false      {'EngineMsg'}    {[0 0 0 0 0 0 0 0]}       8        {1x1 str
```

```
msgRxSlow = msgRx(strcmpi("TransmissionMsg", msgRx.Name), :);
head(msgRxSlow)
```

```
ans=8×8 timetable
        Time          ID     Extended           Name                     Data              Length
    _____________    ___    ________    ___________________    ____________________    ______    _

    0.070272 sec     200     false      {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}       8       {
    0.56905 sec      200     false      {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}       8       {
```

```
    1.0668 sec      200     false     {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {1
    1.5646 sec      200     false     {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {1
    2.0763 sec      200     false     {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {1
    2.5751 sec      200     false     {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {1
    3.0739 sec      200     false     {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {1
    3.5747 sec      200     false     {'TransmissionMsg'}    {[0 0 0 0 0 0 0 0]}      8       {1
```

Analyze the timestamps of each set of messages to see how closely the average of the differences corresponds to the configured periodic rates.

```
avgPeriodFast = mean(diff(msgRxFast.Time))
```

```
avgPeriodFast = duration
   0.10549 sec
```

```
avgPeriodSlow = mean(diff(msgRxSlow.Time))
```

```
avgPeriodSlow = duration
   0.50036 sec
```

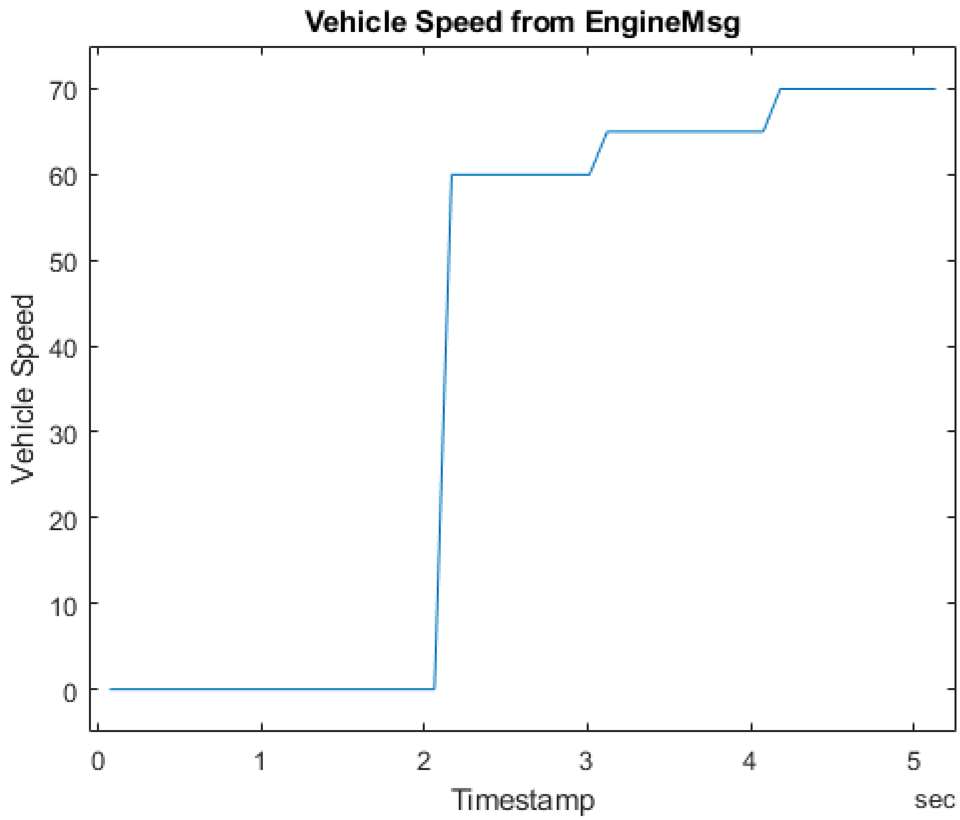
Use `canSignalTimetable` to repackage signal data from message `EngineMsg` into a signal timetable.

```
signalTimetable = canSignalTimetable(msgRx, "EngineMsg");
head(signalTimetable)
```

```
ans=8×2 timetable
        Time         VehicleSpeed    EngineRPM
    ____________     ____________    _________

    0.070266 sec          0             250
    0.17624 sec           0             250
    0.28219 sec           0             250
    0.38811 sec           0             250
    0.49409 sec           0             250
    0.59903 sec           0             250
    0.70499 sec           0             250
    0.80992 sec           0             250
```

Plot the received values of signal `VehicleSpeed` over time and note how it reflects the three updates in message data.

```
plot(signalTimetable.Time, signalTimetable.VehicleSpeed)
title("Vehicle Speed from EngineMsg", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Vehicle Speed")
ylim([-5 75])
```



View Messages Configured for Periodic Transmission

To see messages configured on the transmitting channel for automatic transmission, use the `transmitConfiguration` command.

```
transmitConfiguration(txCh)
```

```
Periodic Messages

ID  Extended      Name             Data          Rate (seconds)
--- -------- -------------- ---------------- --------------
100 false    EngineMsg      0 0 0 0 70 0 0 0  0.100000
200 false    TransmissionMsg 0 0 0 0 0 0 0 0   0.500000


Event Messages

None
```

Close the Channels and DBC-File

Close access to the channels and the DBC-file by clearing their variables from the workspace.

```
clear rxCh txCh
clear db
```

Event-Based CAN Communication in MATLAB

This example shows you how to configure CAN channels and messages for transmit messages on event. It uses MathWorks virtual CAN channels connected in a loopback configuration.

As this example is based on sending and receiving CAN messages on a virtual network, running CAN Explorer in conjunction may provide a more complete understanding of what the code is doing. To run CAN Explorer, open and configure it to use the same interface as the receiving channel of the example. Make sure to start CAN Explorer before beginning to run the example in order to see all of the messages as they occur.

This example describes the workflow for a CAN network, but the concept demonstrated also applies to a CAN FD network.

Create the CAN Channels

Create CAN channels for message transmission and reception.

```
txCh = canChannel("MathWorks", "Virtual 1", 1);
rxCh = canChannel("MathWorks", "Virtual 1", 2);
```

Open the DBC-file that contains message and signal definitions, and attach it to both CAN channels.

```
db = canDatabase("CANDatabaseEvent.dbc");
txCh.Database = db;
rxCh.Database = db;
```

Create the CAN Message

Create CAN message `EngineMsg` using the database information.

```
msgEngineMsg = canMessage(db, "EngineMsg")
```

```
msgEngineMsg = 
  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: []
```

Configure the Message for Event-Based Transmission

To enable a message for event-based transmission, use the `transmitEvent` command specifying the transmitting channel, the message to register on the channel, and a state value.

```
transmitEvent(txCh, msgEngineMsg, "On");
```

Start the Event-Based Transmission

Start the receiving and transmitting channels.

```
start(rxCh);
start(txCh);
```

Write new values to the `Data` property and directly to the `VehicleSpeed` signal to trigger automatic event-based transmission of the message onto the CAN bus.

```
msgEngineMsg.Data = [250 100 0 0 20 0 0 0];
pause(1);
msgEngineMsg.Signals.VehicleSpeed = 60;
pause(1);
```

Stop the transmitting and receiving channels.

```
stop(txCh);
stop(rxCh);
```

Analyze the Behavior of Event-Based Transmission

The receiving channel now has two messages available, corresponding to the two updates that resulted in two transmissions.

```
rxCh.MessagesAvailable
```

```
ans = 2
```

Receive the available messages. Inspect the messages and note that each has the data values set previously to the `Data` property.

```
msgRx = receive(rxCh, Inf, "OutputFormat", "timetable")
```

```
msgRx=2×8 timetable
        Time         ID     Extended        Name                   Data               Length      S
    ____________    ___    ________    _____________    ________________________    ______    __

    0.088211 sec    100     false      {'EngineMsg'}    {[250 100 0 0 20 0 0 0]}      8       {1x
    1.1006 sec      100     false      {'EngineMsg'}    {[250 100 0 0 60 0 0 0]}      8       {1x
```

Inspect the signals and note that the second instance of `VehicleSpeed` has the data value set previously to the `VehicleSpeed` signal.

```
signals = canSignalTimetable(msgRx)
```

```
signals=2×2 timetable
        Time        VehicleSpeed    EngineRPM
    ____________    ____________    _________

    0.088211 sec         20           2835
```

```
    1.1006 sec           60           2835
```

View Messages Configured for Event-Based Transmission

To see messages configured on the transmitting channel for automatic transmission, use the `transmitConfiguration` command.

```
transmitConfiguration(txCh)
```

```
Periodic Messages

None


Event Messages

ID  Extended   Name            Data
--- -------- --------- ---------------------
100 false    EngineMsg 250 100 0 0 60 0 0 0
```

Close the Channels and DBC-File

Close access to the channels and the DBC-file by clearing their variables from the workspace.

```
clear rxCh txCh
clear db
```

Use Relative and Absolute Timestamps in CAN Communication

This example shows you how to use the InitialTimestamp property of a CAN channel to work with relative and absolute timestamps for CAN messages. It uses MathWorks virtual CAN channels connected in a loopback configuration. This example describes the workflow for a CAN network, but the concept demonstrated also applies to a CAN FD network.

Open the DBC-File

Open the DBC-file to access the database definitions.

```
db = canDatabase("VehicleInfo.dbc")
```

```
db = 
  Database with properties:

             Name: 'VehicleInfo'
             Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex13648766\VehicleIn
            Nodes: {}
         NodeInfo: [0x0 struct]
         Messages: {'WheelSpeeds'}
      MessageInfo: [1x1 struct]
       Attributes: {'BusType'}
    AttributeInfo: [1x1 struct]
         UserData: []
```

Create the CAN Channels

Create CAN channels on which you can transmit and receive messages.

```
txCh = canChannel("MathWorks", "Virtual 1", 1)
```

```
txCh = 
  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: []
```

```
rxCh = canChannel("MathWorks", "Virtual 1", 2)
```

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

Attach the database directly to the receiving channel to apply database definitions to incoming messages automatically.

```
rxCh.Database = db;
```

Create the CAN Message

Create a new CAN message by specifying the database and the message name `WheelSpeeds` to have the database definition applied.

```
msg = canMessage(db, "WheelSpeeds")
```

```
msg = 
  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: []
```

Start the CAN Channels

Start the channels to begin using them for transmission and reception.

```
start(rxCh)
start(txCh)
```

Transmit CAN Messages

The `transmit` function sends messages onto the network. Use `pause` to add delays between the transmit operations. Update the `LF_WSpeed` signal value before each transmission.

```
msg.Signals.LF_WSpeed = 10;
transmit(txCh, msg)
pause(1);
msg.Signals.LF_WSpeed = 20;
transmit(txCh, msg)
pause(2);
msg.Signals.LF_WSpeed = 30;
transmit(txCh, msg)
pause(3);
msg.Signals.LF_WSpeed = 40;
transmit(txCh, msg)
pause(1);
msg.Signals.LF_WSpeed = 50;
transmit(txCh, msg)
```

Receive the CAN Messages

The `receive` function receives CAN messages that occurred on the network.

```
stop(rxCh)
stop(txCh)
msgRx = receive(rxCh, Inf, "OutputFormat", "timetable")
```

```
msgRx=5×8 timetable
       Time         ID     Extended         Name                   Data             Length
    ____________    ____    ________    _______________    ________________________    ______    _

    0.099503 sec    1200     false      {'WheelSpeeds'}    {[42 248 0 0 0 0 0 0]}       8       {
    1.126 sec       1200     false      {'WheelSpeeds'}    {[46 224 0 0 0 0 0 0]}       8       {
    3.1504 sec      1200     false      {'WheelSpeeds'}    {[50 200 0 0 0 0 0 0]}       8       {
    6.1818 sec      1200     false      {'WheelSpeeds'}    {[54 176 0 0 0 0 0 0]}       8       {
    7.1986 sec      1200     false      {'WheelSpeeds'}    {[58 152 0 0 0 0 0 0]}       8       {
```

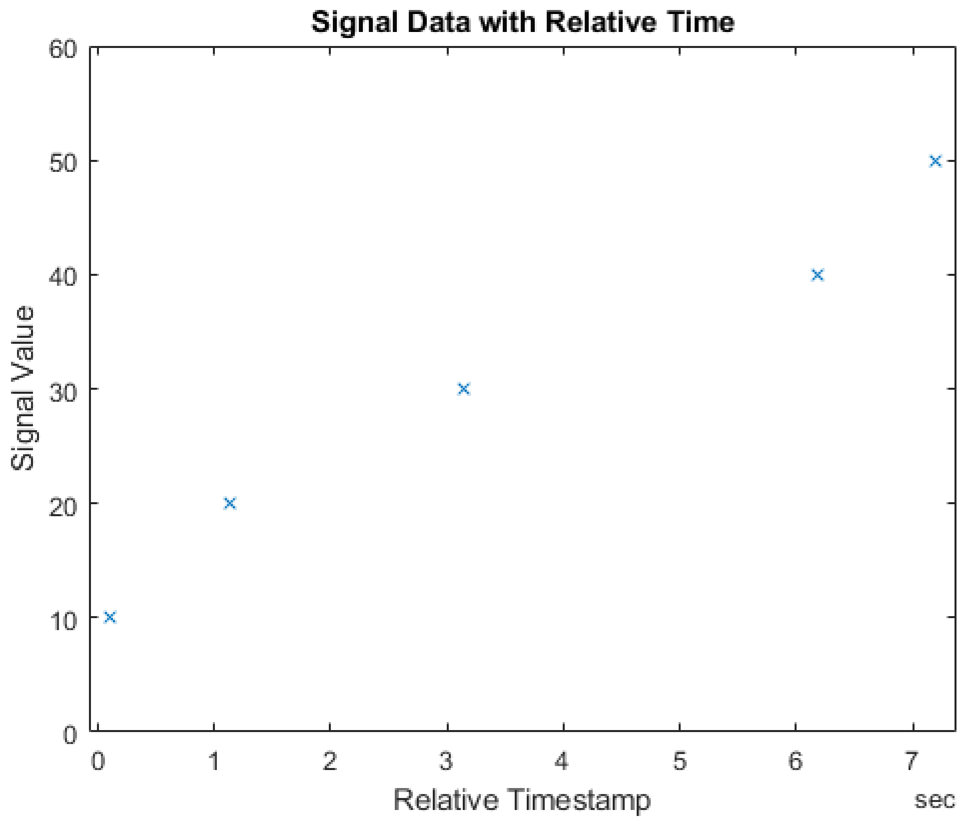
Inspect Signal Data

Use `canSignalTimetable` to repackage signal data from the received messages into a signal timetable. Note that timestamp values represent time elapsed from the start of the CAN channel.

```
signalTimetable = canSignalTimetable(msgRx)
```

```
signalTimetable=5×4 timetable
       Time         LR_WSpeed    RR_WSpeed    RF_WSpeed    LF_WSpeed
    ____________    _________    _________    _________    _________

    0.099503 sec      -100         -100         -100          10
    1.126 sec         -100         -100         -100          20
    3.1504 sec        -100         -100         -100          30
    6.1818 sec        -100         -100         -100          40
    7.1986 sec        -100         -100         -100          50
```

```
plot(signalTimetable.Time, signalTimetable.LF_WSpeed, "x")
title("Signal Data with Relative Time", "FontWeight", "bold")
xlabel("Relative Timestamp")
ylabel("Signal Value")
ylim([0 60])
```



Inspect `InitialTimestamp` Property

View the `InitialTimestamp` property of the receiving CAN channel. It is a `datetime` value that represents the absolute time of when the channel is started.

```
rxCh.InitialTimestamp
```

```
ans = datetime
   01-Sep-2021 12:39:51
```

Analyze Data with Absolute Timestamps

Combine the relative timestamp of each message and the `InitialTimestamp` property to obtain the absolute timestamp of each message. Set the absolute timestamps back into the message timetable as the time vector.

```
msgRx.Time = msgRx.Time + rxCh.InitialTimestamp
```

```
msgRx=5×8 timetable
            Time             ID     Extended        Name                  Data             Leng
    ____________________    ____    ________    _______________    ______________________    ____

    01-Sep-2021 12:39:51    1200     false      {'WheelSpeeds'}    {[42 248 0 0 0 0 0 0]}      8
    01-Sep-2021 12:39:52    1200     false      {'WheelSpeeds'}    {[46 224 0 0 0 0 0 0]}      8
    01-Sep-2021 12:39:54    1200     false      {'WheelSpeeds'}    {[50 200 0 0 0 0 0 0]}      8
    01-Sep-2021 12:39:57    1200     false      {'WheelSpeeds'}    {[54 176 0 0 0 0 0 0]}      8
```

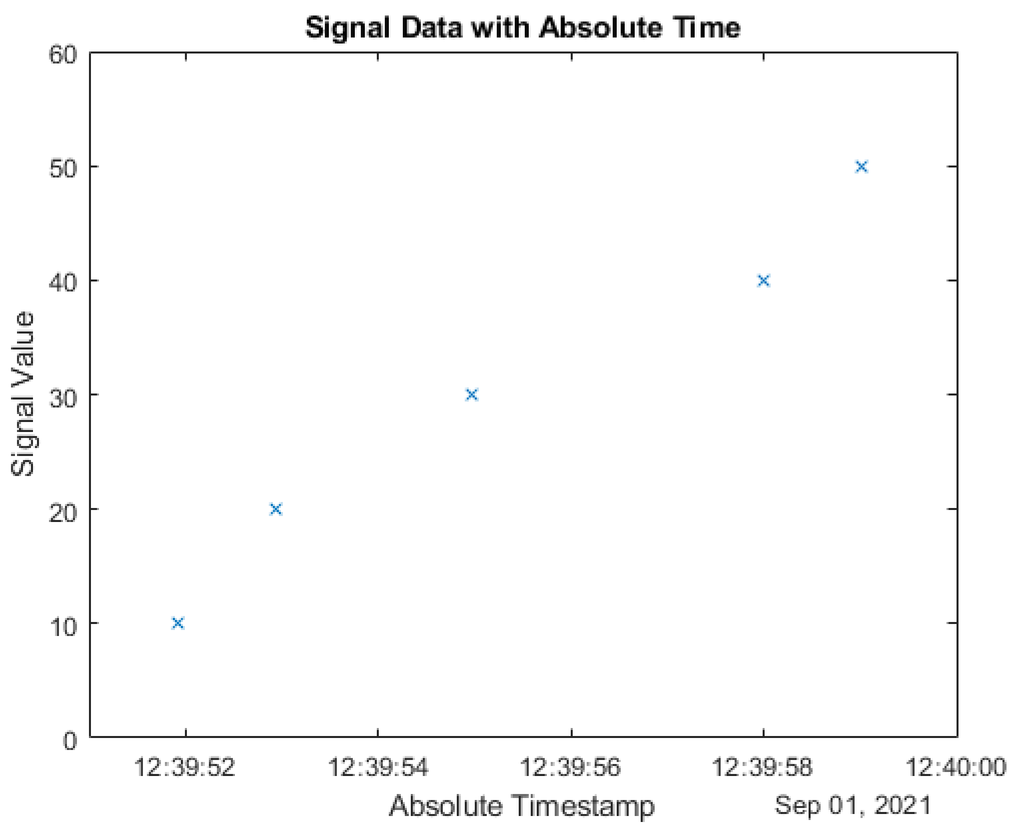
```
    01-Sep-2021 12:39:59    1200     false      {'WheelSpeeds'}    {[58 152 0 0 0 0 0 0]}      8
```

The signal timetable created from the updated message timetable now also has absolute timestamps.

```
signalTimetable = canSignalTimetable(msgRx)
```

```
signalTimetable=5×4 timetable
            Time            LR_WSpeed    RR_WSpeed    RF_WSpeed    LF_WSpeed
    ____________________    _________    _________    _________    _________

    01-Sep-2021 12:39:51      -100         -100         -100          10
    01-Sep-2021 12:39:52      -100         -100         -100          20
    01-Sep-2021 12:39:54      -100         -100         -100          30
    01-Sep-2021 12:39:57      -100         -100         -100          40
    01-Sep-2021 12:39:59      -100         -100         -100          50
```

```
figure
plot(signalTimetable.Time, signalTimetable.LF_WSpeed, "x")
title("Signal Data with Absolute Time", "FontWeight", "bold")
xlabel("Absolute Timestamp")
ylabel("Signal Value")
ylim([0 60])
```



Close the Channels and DBC-File

Close access to the channels and the DBC-file by clearing their variables from the workspace.

```
clear rxCh txCh
clear db
```

Get Started with J1939 Parameter Groups in MATLAB

This example shows you how to create and manage J1939 parameter groups using information stored in DBC-files. This example uses file `J1939.dbc`. Creating and using parameter groups this way is recommended when needing to transmit data to a J1939 network.

Open the DBC-File

Open the DBC-file using `canDatabase` to access the definitions.

```
db = canDatabase("J1939.dbc")
```

```
db = 
  Database with properties:

             Name: 'J1939'
             Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex46196345\J1939.dbc
            Nodes: {2x1 cell}
         NodeInfo: [2x1 struct]
         Messages: {2x1 cell}
      MessageInfo: [2x1 struct]
       Attributes: {3x1 cell}
    AttributeInfo: [3x1 struct]
         UserData: []
```

Create a Parameter Group

Use the `j1939ParameterGroup` function to create a parameter group using information contained within the database.

```
pg = j1939ParameterGroup(db, "VehicleDataSingle")
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataSingle'
                   PGN: 40192
              Priority: 6
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 254
    DestinationAddress: 254

   Data Details:
   -------------
             Timestamp: 0
                  Data: [255 255 255 255 255 255 255 255]
               Signals: [1x1 struct]

   Other Information:
   ------------------
              UserData: []
```

Set Source and Destination Addresses

To fully define the parameter group and determine the logistics of its transmission on a network, set the source and destination addresses.

```
pg.SourceAddress = 30
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataSingle'
                   PGN: 40192
              Priority: 6
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 30
    DestinationAddress: 254

   Data Details:
   -------------
             Timestamp: 0
                  Data: [255 255 255 255 255 255 255 255]
               Signals: [1x1 struct]

   Other Information:
   ------------------
              UserData: []
```

```
pg.DestinationAddress = 50
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataSingle'
                   PGN: 40192
              Priority: 6
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 30
    DestinationAddress: 50

   Data Details:
   -------------
             Timestamp: 0
                  Data: [255 255 255 255 255 255 255 255]
               Signals: [1x1 struct]

   Other Information:
   ------------------
              UserData: []
```

Set Priority

Set the `Priority` property to further customize the transmission.

```
pg.Priority = 5;
```

View Signal Information

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

```
pg.Signals
```

```
ans = struct with fields:
    VehicleSignal4: -1
    VehicleSignal3: -1
    VehicleSignal2: -1
    VehicleSignal1: -1
```

Change Signal Information

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

```
pg.Signals.VehicleSignal1 = 10
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                 Name: 'VehicleDataSingle'
                  PGN: 40192
             Priority: 5
        PDUFormatType: 'Peer-to-Peer (Type 1)'
        SourceAddress: 30
   DestinationAddress: 50

   Data Details:
   -------------
            Timestamp: 0
                 Data: [10 0 255 255 255 255 255 255]
              Signals: [1x1 struct]

   Other Information:
   ------------------
             UserData: []
```

```
pg.Signals.VehicleSignal2 = 100
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                 Name: 'VehicleDataSingle'
                  PGN: 40192
             Priority: 5
        PDUFormatType: 'Peer-to-Peer (Type 1)'
        SourceAddress: 30
   DestinationAddress: 50
```

```
   Data Details:
   -------------
             Timestamp: 0
                  Data: [10 0 100 0 255 255 255 255]
               Signals: [1x1 struct]

   Other Information:
   ------------------
              UserData: []
```

```
pg.Signals.VehicleSignal3 = 1000
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataSingle'
                   PGN: 40192
              Priority: 5
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 30
    DestinationAddress: 50

   Data Details:
   -------------
             Timestamp: 0
                  Data: [10 0 100 0 232 3 255 255]
               Signals: [1x1 struct]

   Other Information:
   ------------------
              UserData: []
```

```
pg.Signals.VehicleSignal4 = 10000
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataSingle'
                   PGN: 40192
              Priority: 5
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 30
    DestinationAddress: 50

   Data Details:
   -------------
             Timestamp: 0
                  Data: [10 0 100 0 232 3 16 39]
               Signals: [1x1 struct]

   Other Information:
   ------------------
```

```
            UserData: []
```

```
pg.Signals
```

```
ans = struct with fields:
    VehicleSignal4: 10000
    VehicleSignal3: 1000
    VehicleSignal2: 100
    VehicleSignal1: 10
```

Write New Direct Data

You can also write values directly into the `Data` property, although setting values through `Signals` is generally recommended and preferred.

```
pg.Data(1:2) = [50 0]
```

```
pg = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                 Name: 'VehicleDataSingle'
                  PGN: 40192
             Priority: 5
        PDUFormatType: 'Peer-to-Peer (Type 1)'
        SourceAddress: 30
   DestinationAddress: 50

   Data Details:
   -------------
            Timestamp: 0
                 Data: [50 0 100 0 232 3 16 39]
              Signals: [1x1 struct]

   Other Information:
   ------------------
             UserData: []
```

```
pg.Signals
```

```
ans = struct with fields:
    VehicleSignal4: 10000
    VehicleSignal3: 1000
    VehicleSignal2: 100
    VehicleSignal1: 50
```

Get Started with J1939 Communication in MATLAB

This example shows you how to create and use J1939 channels to transmit and receive parameter groups on a J1939 network. This example uses the database file `J1939.dbc` and MathWorks virtual CAN channels connected in a loopback configuration.

Open the DBC-File

Open the DBC-file using `canDatabase` to access the definitions.

```
db = canDatabase("J1939.dbc")
```

```
db = 
  Database with properties:

             Name: 'J1939'
             Path: 'C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex336052
            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: []
```

Create the J1939 Channels

Use the function `j1939Channel` to create J1939 channels on which you can send and receive information.

```
txCh = j1939Channel(db, "MathWorks", "Virtual 1", 1)
```

```
txCh = 
  Channel with properties:

   Device Information:
   -------------------
                 DeviceVendor: 'MathWorks'
                       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: [0×0 datetime]
                    SilentMode: 0
```

```
              TransceiverName: 'N/A'
             TransceiverState: 'N/A'
                     BusSpeed: 500000
                          SJW: []
                        TSEG1: []
                        TSEG2: []
                 NumOfSamples: []

   Other Information:
   ------------------
                     UserData: []
```

```
rxCh = j1939Channel(db, "MathWorks", "Virtual 1", 2)
```

```
rxCh = 
  Channel with properties:

   Device Information:
   -------------------
                 DeviceVendor: 'MathWorks'
                       Device: 'Virtual 1'
           DeviceChannelIndex: 2
           DeviceSerialNumber: 0

   Data Details:
   -------------
      ParameterGroupsAvailable: 0
       ParameterGroupsReceived: 0
    ParameterGroupsTransmitted: 0
                FilterPassList: []
               FilterBlockList: []

   Channel Information:
   --------------------
                      Running: 0
                    BusStatus: 'N/A'
         InitializationAccess: 1
             InitialTimestamp: [0×0 datetime]
                   SilentMode: 0
              TransceiverName: 'N/A'
             TransceiverState: 'N/A'
                     BusSpeed: 500000
                          SJW: []
                        TSEG1: []
                        TSEG2: []
                 NumOfSamples: []

   Other Information:
   ------------------
                     UserData: []
```

Create the J1939 Parameter Groups

Use the function `j1939ParameterGroup` to create a single-frame parameter group to send on the network.

```
pgSingleFrame = j1939ParameterGroup(db, "VehicleDataSingle")
```

```
pgSingleFrame = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataSingle'
                   PGN: 40192
              Priority: 6
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 254
    DestinationAddress: 254

   Data Details:
   -------------
             Timestamp: 0
                  Data: [255 255 255 255 255 255 255 255]
               Signals: [1×1 struct]

   Other Information:
   ------------------
              UserData: []
```

Set transmission details and signal data.

```
pgSingleFrame.SourceAddress = 30;
pgSingleFrame.DestinationAddress = 50;
pgSingleFrame.Signals.VehicleSignal1 = 25;
pgSingleFrame.Signals.VehicleSignal2 = 1000;
pgSingleFrame.Signals
```

```
ans = struct with fields:
    VehicleSignal4: -1
    VehicleSignal3: -1
    VehicleSignal2: 1000
    VehicleSignal1: 25
```

Using the same approach, create a multi-frame parameter group, then set transmission details and signal data.

```
pgMultiFrame = j1939ParameterGroup(db, "VehicleDataMulti")
```

```
pgMultiFrame = 
  ParameterGroup with properties:

   Protocol Data Unit Details:
   ---------------------------
                  Name: 'VehicleDataMulti'
                   PGN: 51200
              Priority: 6
         PDUFormatType: 'Peer-to-Peer (Type 1)'
         SourceAddress: 254
    DestinationAddress: 254

   Data Details:
   -------------
             Timestamp: 0
```

```
              Data: [255 255 255 255 255 255 255 255 255 255 255 255]
           Signals: [1×1 struct]

   Other Information:
   ------------------
          UserData: []
```

```
pgMultiFrame.SourceAddress = 30;
pgMultiFrame.DestinationAddress = 255;
pgMultiFrame.Signals.VehicleSignal1 = 5;
pgMultiFrame.Signals.VehicleSignal2 = 650;
pgMultiFrame.Signals.VehicleSignal3 = 5000;
pgMultiFrame.Signals
```

```
ans = struct with fields:
    VehicleSignal6: -1
    VehicleSignal5: -1
    VehicleSignal4: -1
    VehicleSignal3: 5000
    VehicleSignal2: 650
    VehicleSignal1: 5
```

Start the J1939 Channels

Use the function `start` to start the J1939 channels for transmit and receive operations.

```
start(rxCh);
start(txCh);
```

Send J1939 Parameter Groups

The `transmit` function sends parameter groups onto the network. The J1939 channel automatically sends parameter groups requiring multi-frame messaging via its transport protocol.

```
transmit(txCh, pgSingleFrame)
transmit(txCh, pgSingleFrame)
transmit(txCh, pgMultiFrame)
transmit(txCh, pgSingleFrame)
transmit(txCh, pgSingleFrame)
pause(2);
```

Receive the Parameter Groups

The `receive` function retrieves information from the channel which represents messaging that occurred on the network.

```
pgRx = receive(rxCh, Inf)
```

```
pgRx=5×8 timetable
       Time             Name            PGN     Priority       PDUFormatType        SourceAddre
    ___________    _________________    _____    ________    ______________________    ___________

    0.13955 sec    VehicleDataSingle    40192       6        Peer-to-Peer (Type 1)         30
    0.14347 sec    VehicleDataSingle    40192       6        Peer-to-Peer (Type 1)         30
    0.59386 sec    VehicleDataMulti     51200       6        Peer-to-Peer (Type 1)         30
    0.76564 sec    VehicleDataSingle    40192       6        Peer-to-Peer (Type 1)         30
```

```
    0.7702 sec     VehicleDataSingle    40192       6       Peer-to-Peer (Type 1)         30
```

Inspect Received Parameter Groups Signals

View details of the received signals for an instance of the single-frame and the multiframe parameter group.

```
pgRx.Signals{1}
```

```
ans = struct with fields:
    VehicleSignal4: -1
    VehicleSignal3: -1
    VehicleSignal2: 1000
    VehicleSignal1: 25
```

```
pgRx.Signals{3}
```

```
ans = struct with fields:
    VehicleSignal6: -1
    VehicleSignal5: -1
    VehicleSignal4: -1
    VehicleSignal3: 5000
    VehicleSignal2: 650
    VehicleSignal1: 5
```

Access Signal Values

The `j1939SignalTimetable` function allows you to easily extract and transform signal data from a timetable of parameter groups.

```
sigTT = j1939SignalTimetable(pgRx)
```

```
sigTT = struct with fields:
     VehicleDataMulti: [1×6 timetable]
    VehicleDataSingle: [4×4 timetable]
```

```
sigTT.VehicleDataSingle
```

```
ans=4×4 timetable
       Time       VehicleSignal4    VehicleSignal3    VehicleSignal2    VehicleSignal1
    ___________   ______________    ______________    ______________    ______________

    0.13955 sec         -1                -1               1000               25
    0.14347 sec         -1                -1               1000               25
    0.76564 sec         -1                -1               1000               25
    0.7702 sec          -1                -1               1000               25
```

```
sigTT.VehicleDataMulti
```

```
ans=1×6 timetable
       Time       VehicleSignal6    VehicleSignal5    VehicleSignal4    VehicleSignal3    Vehic
    ___________   ______________    ______________    ______________    ______________    _____

    0.59386 sec         -1                -1                -1               5000             6
```

Stop the J1939 Channels

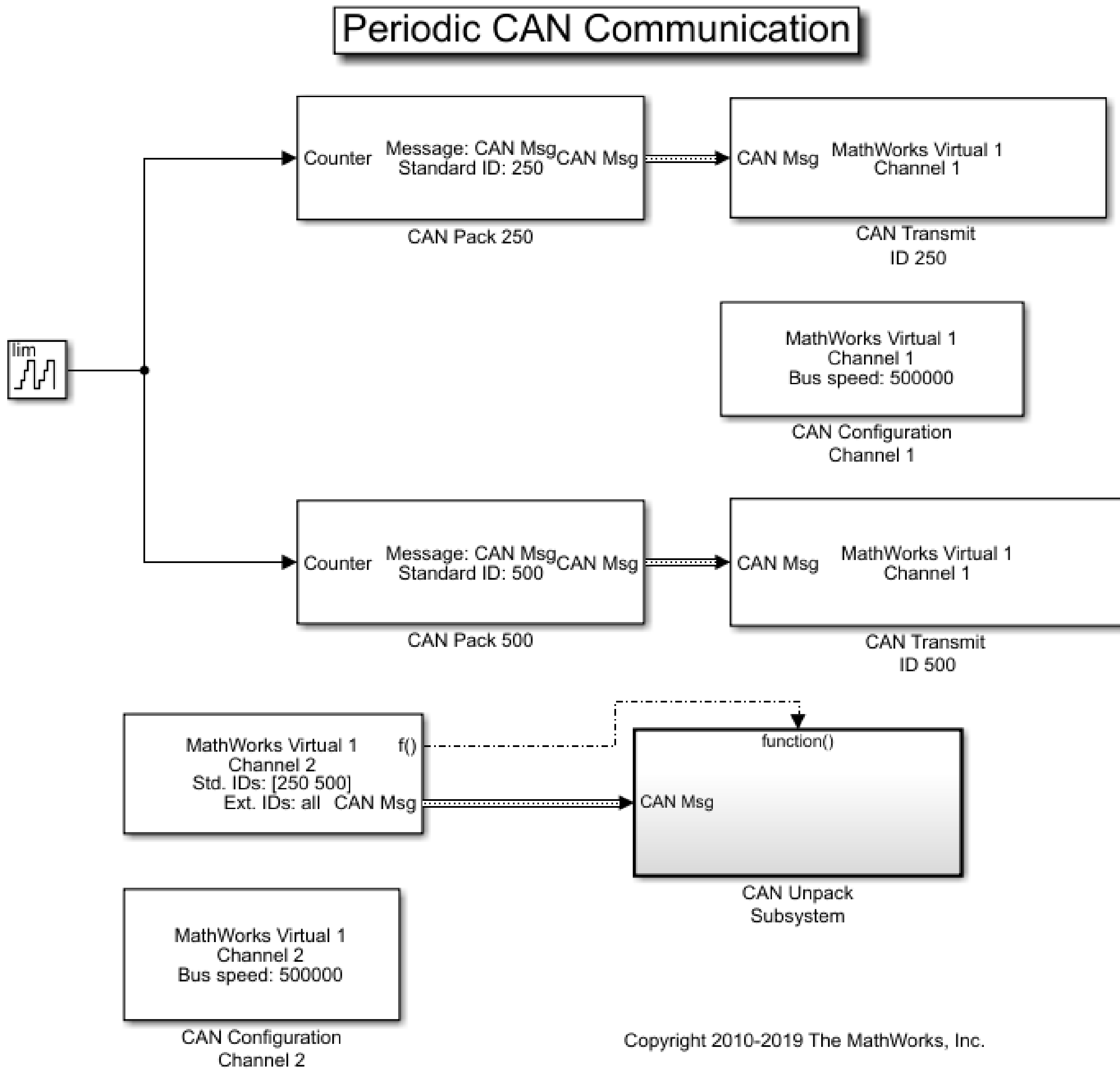
To stop receiving data from the network, stop the J1939 channels using the `stop` function.

```
stop(rxCh);
stop(txCh);
```

Periodic CAN Message Transmission Behavior in Simulink

This example shows how to set up periodic transmission and reception of CAN messages in Simulink using MathWorks virtual CAN channels. The virtual channels are connected in a loopback configuration.

Vehicle Network Toolbox™ provides Simulink blocks for transmitting and receiving live messages via Simulink models over Controller Area Networks (CAN). This example uses the CAN Configuration, CAN Pack, CAN Transmit, CAN Receive and CAN Unpack blocks to perform data transfer over a CAN bus.



Transmit and Receive CAN Messages

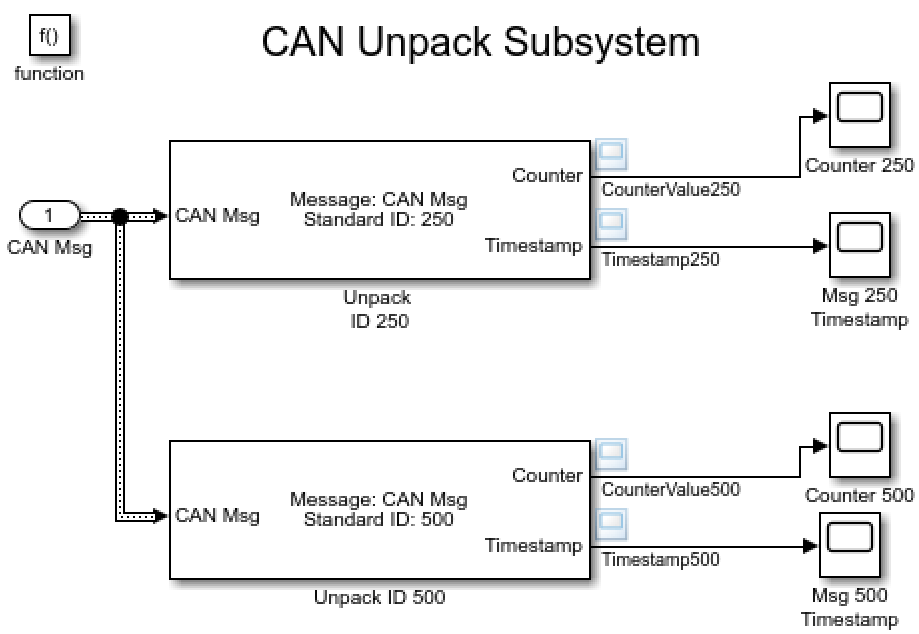
Create a model to transmit two messages at different periods, receive only specified messages and unpack the message with a specified ID.

- Use a CAN Transmit block to transmit the CAN message with ID 250 to transmit messages every 1 second.
- Use another CAN Transmit block to transmit the CAN message with ID 500 to transmit messages every 0.5 seconds.
- Input a signal to both CAN Pack blocks to an auto-incrementing counter with a limit of 50.
- Both CAN Transmit blocks are connected to MathWorks virtual channel 1.

Use a CAN Receive block to receive CAN messages from MathWorks virtual channel 2. Set the block to:

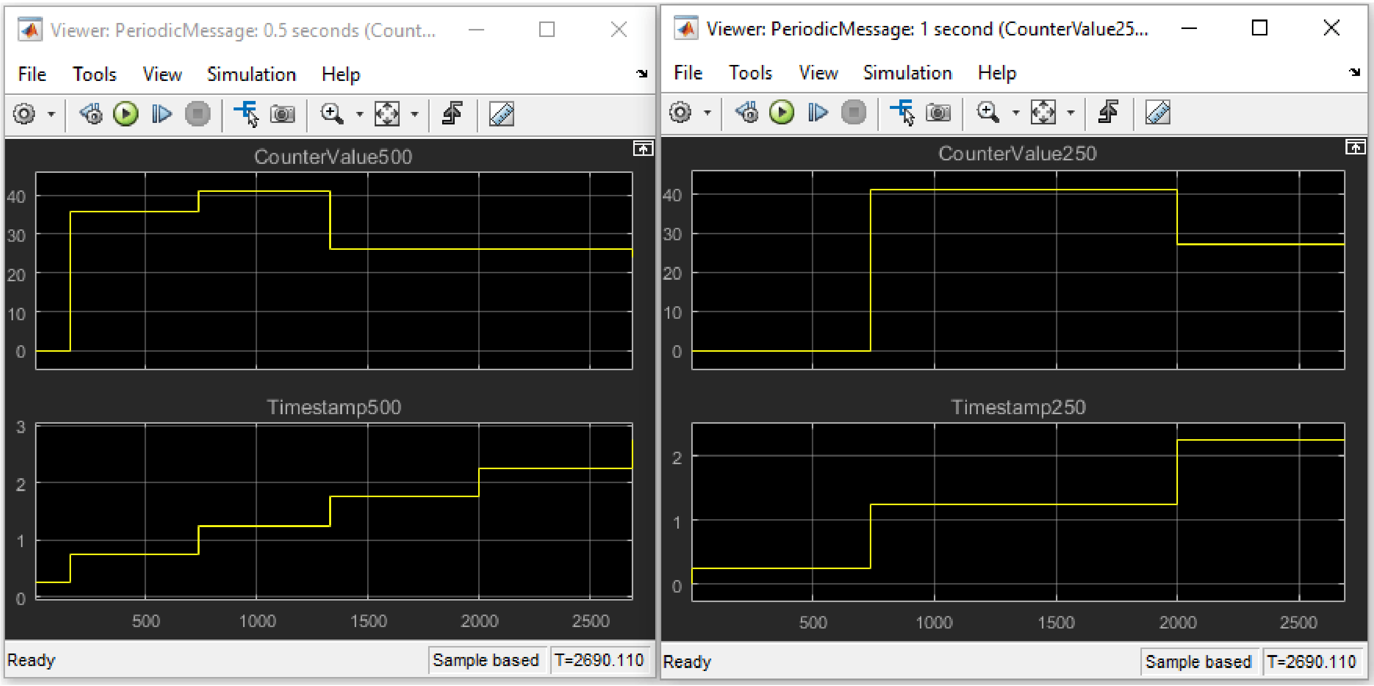
- Receive messages with ID 250 and 500 only.
- The Receive block generates a function call trigger if it receives a new message at any particular timestep.

The CAN Unpack blocks are in a Function-Call Subsystem. The subsystem is executed only when a new message is received by the CAN Receive block at a particular timestep.



Visualize Messages at Different Timestamps

Plot the results to see the counter value and timestamp for each unpacked message. The X-axis on the plot corresponds to the simulation timestep. The timestamp plots show that the messages are sent at the specified times. It can also be seen that the number of messages transmitted for ID 250 is half as much transmitted for ID 500 due to the different periodic rates specified for them.



Extend the Example

MathWorks virtual CAN channels were used for this example. You can however connect your models to other supported hardware. You can also modify the model to transmit at different rates or transmit a combination of periodic and non-periodic messages.

This example uses the CAN blocks, but the concept demonstrated also applies to the CAN FD blocks in Simulink.

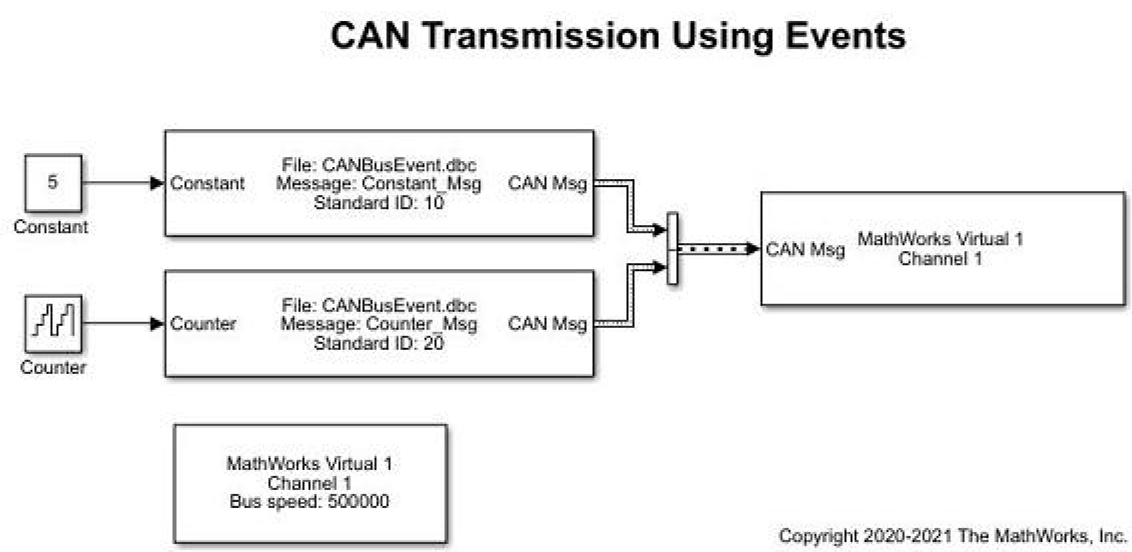
Event-Based CAN Message Transmission Behavior in Simulink

This example shows how to use event-based CAN message transmission in Simulink with Vehicle Network Toolbox. This feature allows for CAN and CAN FD message transmission when a change in data from one time step to the next is detected.

A configuration option available on the CAN and CAN FD Transmit blocks enables transmission on data change. When enabled, messages of particular CAN IDs transmit only when the data changes for that ID. Each message is independently processed in every time step based on its ID. When disabled, block operation and periodic transmit operation function normally. In addition, the event-based transmission can be enabled along with periodic transmission to have both work together simultaneously.

Prepare the Example Model

The included example model contains two CAN Pack blocks configured into a single CAN Transmit block. One message's data is a constant while the other is a counter that changes at every time step.



Open the example model.

```
open EventTransmit
```

Prepare the CAN Database File Access

You can access the contents of CAN DBC-files with the `canDatabase` function. Through this function, details about network nodes, messages, and signals are available. This DBC-file is used in the model and is used to decode information sent from the model.

```
db = canDatabase("CANBusEvent.dbc")
```

```
db = 
  Database with properties:

             Name: 'CANBusEvent'
```

```
            Path: 'C:\Users\jpyle\Documents\MATLAB\ExampleManager\jpyle.21bExampleBlitz\vnt-ex5
           Nodes: {'ECU'}
        NodeInfo: [1×1 struct]
        Messages: {2×1 cell}
     MessageInfo: [2×1 struct]
      Attributes: {}
   AttributeInfo: [0×0 struct]
        UserData: []
```

A test node is defined in the DBC-file.

```
node = nodeInfo(db,"ECU")
```

```
node = struct with fields:
            Name: 'ECU'
         Comment: ''
      Attributes: {}
   AttributeInfo: [0×0 struct]
```

The node transmits two CAN messages.

```
messageInfo(db,"Constant_Msg")
```

```
ans = struct with fields:
            Name: 'Constant_Msg'
    ProtocolMode: 'CAN'
         Comment: ''
              ID: 10
        Extended: 0
           J1939: []
          Length: 4
             DLC: 4
             BRS: 0
         Signals: {'Constant'}
      SignalInfo: [1×1 struct]
         TxNodes: {'ECU'}
      Attributes: {}
   AttributeInfo: [0×0 struct]
```

```
messageInfo(db,"Counter_Msg")
```

```
ans = struct with fields:
            Name: 'Counter_Msg'
    ProtocolMode: 'CAN'
         Comment: ''
              ID: 20
        Extended: 0
           J1939: []
          Length: 4
             DLC: 4
             BRS: 0
         Signals: {'Counter'}
      SignalInfo: [1×1 struct]
         TxNodes: {'ECU'}
      Attributes: {}
```

```
    AttributeInfo: [0×0 struct]
```

Execute the Model with Event-Based Transmission

Enable Event-Based Transmission Only

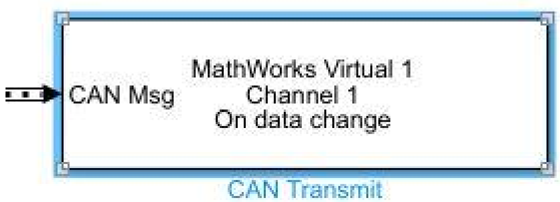
Enable the event-based transmission in the CAN Transmit block programmatically. Also, disable periodic transmission.

```
db = canDatabase("CANBusEvent.dbc")
```

```
db = 
  Database with properties:

             Name: 'CANBusEvent'
             Path: 'C:\Users\jpyle\Documents\MATLAB\ExampleManager\jpyle.21bExampleBlitz\vnt-ex5
            Nodes: {'ECU'}
         NodeInfo: [1×1 struct]
         Messages: {2×1 cell}
      MessageInfo: [2×1 struct]
       Attributes: {}
    AttributeInfo: [0×0 struct]
         UserData: []
```

```
set_param('EventTransmit/CAN Transmit', 'EnableEventTransmit', 'on');
set_param('EventTransmit/CAN Transmit', 'EnablePeriodicTransmit', 'off');
```

Note that the block display changes after applying the settings.



Configure a CAN Channel in MATLAB for Communication with the Model

Create a CAN channel using virtual device communication to interface with the Simulink model. Also, attach the CAN database to it to automatically decode incoming messages.

```
canCh = canChannel("Mathworks","Virtual 1",2)
```

```
canCh = 
  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: [0×0 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: []
```

```
canCh.Database = db;
```

Start the CAN channel to go online.

```
start(canCh);
```

Run the Model

Assign a simulation run time and start the model.

```
t = "10";
set_param("EventTransmit","StopTime",t)
set_param("EventTransmit","SimulationCommand","start");
```

Wait until the simulation starts.

```
while strcmp(get_param("EventTransmit","SimulationStatus"),"stopped")
end
```

Wait until the simulation ends.

```
pause(2)
```

Receive Messages in MATLAB

Receive all messages from the bus generated by the model.

```
msg = receive(canCh,inf,"OutputFormat","timetable")
```

```
msg=12×8 timetable
       Time        ID    Extended          Name              Data        Length      Signals
    ___________    __    ________    ________________    ____________    ______    ____________
```

```
    5.204 sec     10     false     {'Constant_Msg'}    {[ 5 0 0 0]}     4      {1×1 struct}
    5.204 sec     20     false     {'Counter_Msg' }    {[ 0 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 1 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 2 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 3 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 4 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 5 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 6 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 7 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 8 0 0 0]}     4      {1×1 struct}
    5.206 sec     20     false     {'Counter_Msg' }    {[ 9 0 0 0]}     4      {1×1 struct}
    5.2061 sec    20     false     {'Counter_Msg' }    {[10 0 0 0]}     4      {1×1 struct}
```

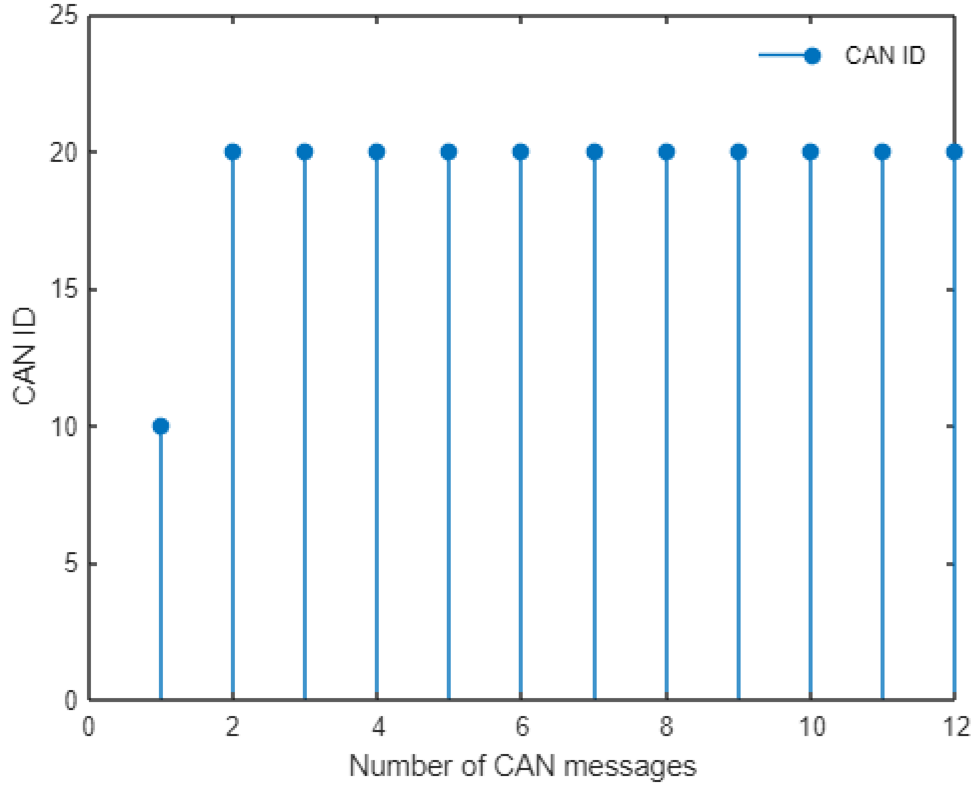
Stop the CAN channel in MATLAB.

```
stop(canCh);
```

Explore the Message and Signal Data Received

The number of times a CAN ID has been received is plotted below. The message "Constant_Msg" (CAN ID 10) is received only once because its data does not change after its initial setting. The message "Counter_Msg" (CAN ID 20) is received from every time step because the data changed continuously in it as the model ran.

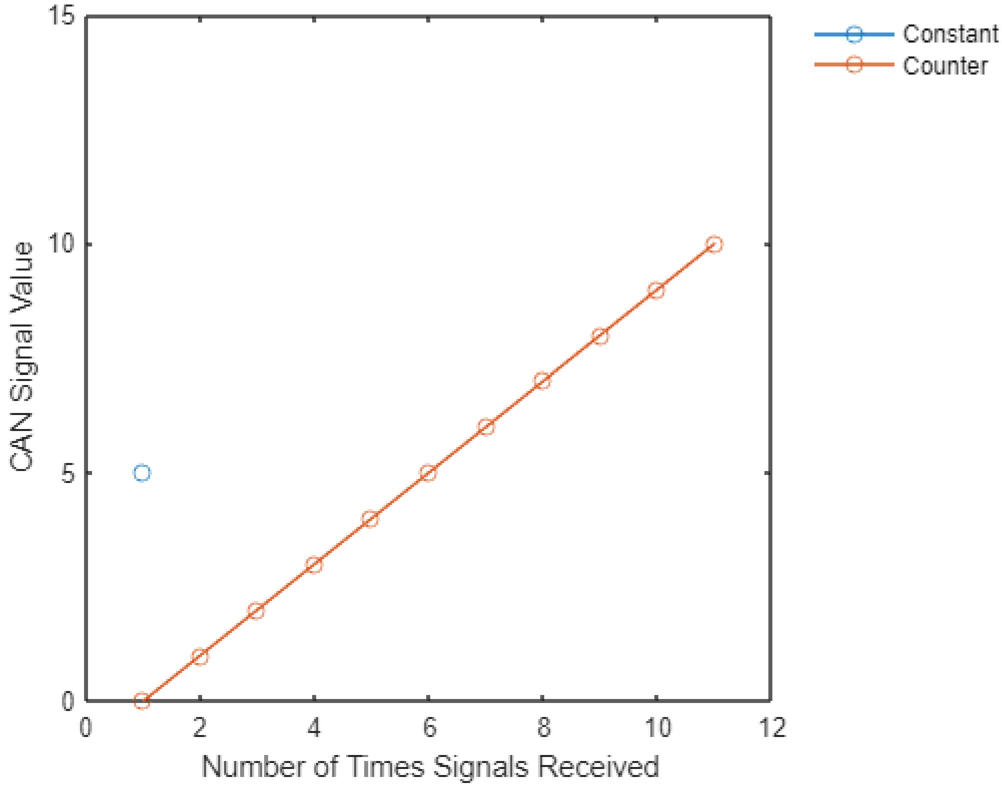
```
% Define X and Y axis.
x = 1:length(msg.ID);
y = msg.ID;

% Plot the graph for both the CAN IDs received.
stem(x,y,'filled')
hold on;
yMax = max(msg.ID)+5;
ylim([0 yMax])

% Label the graph.
xlabel("Number of CAN messages");
ylabel("CAN ID");
legend("CAN ID","Location","northeast");
legend("boxoff");
hold off;
```



Next, plot the signals received in each message over the same simulation run.

```
% Create a structure with signal details.
signalTimeTable = canSignalTimetable(msg);

% Plot the signal values of "Constant_Msg".
x1 = 1:height(signalTimeTable.Constant_Msg);
y1 = signalTimeTable.Constant_Msg.Constant;
plot(x1, y1,"Marker","o");
hold on

% Plot the signal values of "Counter_Msg".
x2 = 1:height(signalTimeTable.Counter_Msg);
y2 = signalTimeTable.Counter_Msg.Counter;
plot(x2, y2,"Marker","o");

% Determine the maximum value for y-axis for scaling of graph.
y1Max = max(signalTimeTable.Constant_Msg.Constant);
y2Max = max(signalTimeTable.Counter_Msg.Counter);
yMax = max(y1Max,y2Max)+5;
ylim([0 yMax]);

% Label the graph.
xlabel("Number of Times Signals Received");
ylabel("CAN Signal Value");
legend("Constant","Counter","Location","northeastoutside");
```

```
legend("boxoff");
hold off
```



The signal "Constant" (in message "Constant_Msg") is plotted only once, while the signal "Counter" (in message "Counter_Msg") is plotted for every time step. This is due to event-based transmission being enabled in the CAN Transmit block, which transmits a CAN message only if data has changed for that CAN ID compared with the previously received message.

As the signal in message "Counter_Msg" is a counter, which increments by 1 at every time step, a linear curve can be seen for it.

Each data points represents a transmission with event-based transmission enabled, hence signal "Counter" is received at every time step, but signal "Constant" is received only once.

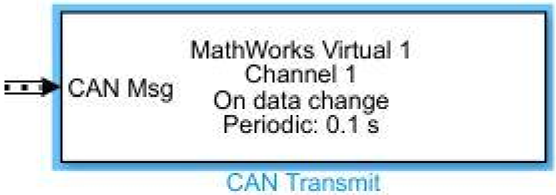
Execute the Model with Event-Based and Periodic Transmission

Enable Both Transmission Modes

Enable the event transmission in the CAN Transmit block programmatically. Also, enable the periodic transmission and set a message period.

```
set_param('EventTransmit/CAN Transmit', 'EnableEventTransmit', 'on');
set_param('EventTransmit/CAN Transmit', 'EnablePeriodicTransmit', 'on');
set_param('EventTransmit/CAN Transmit', 'MessagePeriod', '0.1');
```

Note that the block display changes after applying the settings.



Configure a CAN Channel in MATLAB for Communication with the Model

Create a CAN channel using virtual device communication to interface with the Simulink model. Also, attach the CAN database to it to automatically decode incoming messages.

```
canCh = canChannel("Mathworks","Virtual 1",2)
```

```
canCh = 
  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: [0×0 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: []
```

```
canCh.Database = db;
```

Start the CAN channel to go online.

```
start(canCh);
```

Run the Model

Assign a simulation run time and start the model.

```
t = "20";
set_param("EventTransmit","StopTime",t)
set_param("EventTransmit","SimulationCommand","start");
```

Wait until the simulation starts.

```
while strcmp(get_param("EventTransmit","SimulationStatus"),"stopped")
end
```

Wait until the simulation ends.

```
pause(5);
```

Receive Messages in MATLAB

Receive all messages from the bus generated by the model.

```
msg = receive(canCh,Inf,"OutputFormat","timetable")
```

```
msg=22×8 timetable
        Time        ID     Extended          Name               Data         Length       Signals
    ___________    __    ________    ________________    ______________    ______    ____________

    4.598 sec      10     false      {'Constant_Msg'}    {[ 5 0 0 0]}        4       {1×1 struct}
    4.598 sec      20     false      {'Counter_Msg' }    {[ 0 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 1 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 2 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 3 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 4 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 5 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 6 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 7 0 0 0]}        4       {1×1 struct}
    4.5987 sec     20     false      {'Counter_Msg' }    {[ 8 0 0 0]}        4       {1×1 struct}
    4.5988 sec     20     false      {'Counter_Msg' }    {[ 9 0 0 0]}        4       {1×1 struct}
    4.5988 sec     20     false      {'Counter_Msg' }    {[10 0 0 0]}        4       {1×1 struct}
    4.5988 sec     20     false      {'Counter_Msg' }    {[11 0 0 0]}        4       {1×1 struct}
    4.5988 sec     20     false      {'Counter_Msg' }    {[12 0 0 0]}        4       {1×1 struct}
    4.5988 sec     20     false      {'Counter_Msg' }    {[13 0 0 0]}        4       {1×1 struct}
    4.5988 sec     20     false      {'Counter_Msg' }    {[14 0 0 0]}        4       {1×1 struct}
      ⋮
```

Stop the CAN channel in MATLAB.

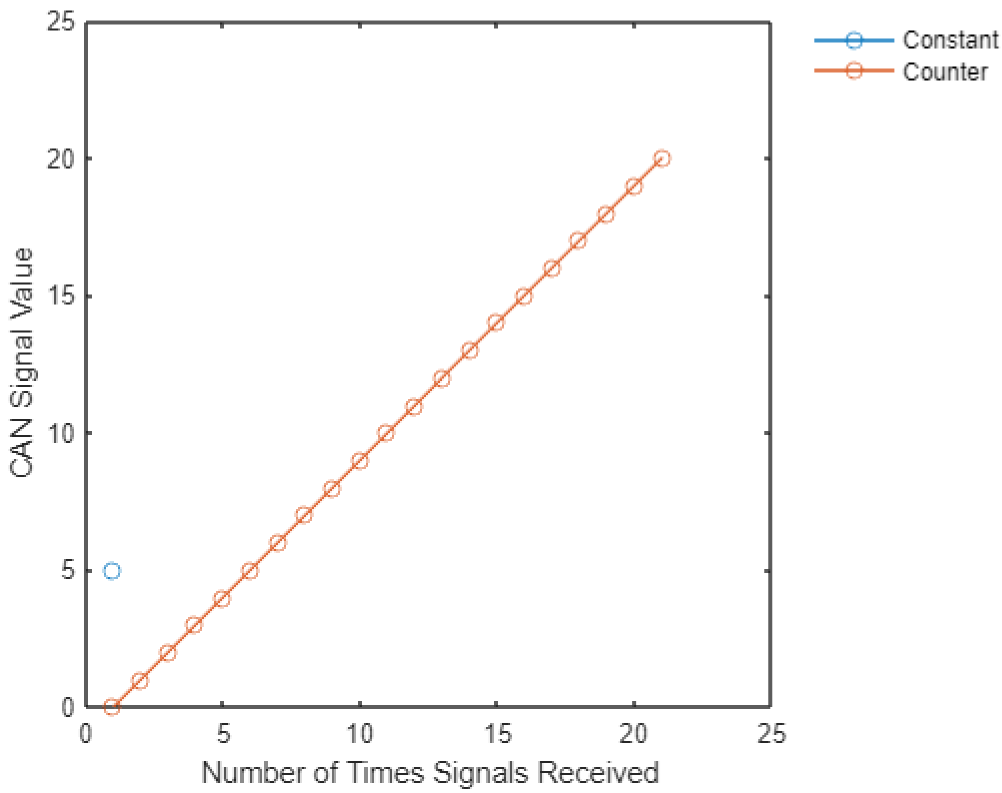
```
stop(canCh);
```

Explore the Data Received

Plot the data received in each message over the same period.

```
% Create a structure with signal details.
signalTimeTable = canSignalTimetable(msg);
```

```
% Plot the signal values of "Constant_Msg".
x3 = 1:height(signalTimeTable.Constant_Msg);
y3 = signalTimeTable.Constant_Msg.Constant;
plot(x3, y3,"Marker","o");
hold on

% Plot the signal values of "Counter_Msg".
x4 = 1:height(signalTimeTable.Counter_Msg);
y4 = signalTimeTable.Counter_Msg.Counter;
plot(x4, y4,"Marker","o");

% Determine the maximum value for y-axis for scaling of graph.
y3Max = max(signalTimeTable.Constant_Msg.Constant);
y4Max = max(signalTimeTable.Counter_Msg.Counter);
yMax = max(y3Max,y4Max)+5;
ylim([0 yMax]);

% Label the graph.
xlabel("Number of Times Signals Received");
ylabel("CAN Signal Value");
legend("Constant","Counter","Location","northeastoutside");
legend("boxoff");
hold off
```



The plot shows that the signal "Constant" in message "Constant_Msg" is received only a few times; once at the start due to the event-based transmission, and later due to the periodic nature of the transmission. This is because the input value to the signal is kept constant.

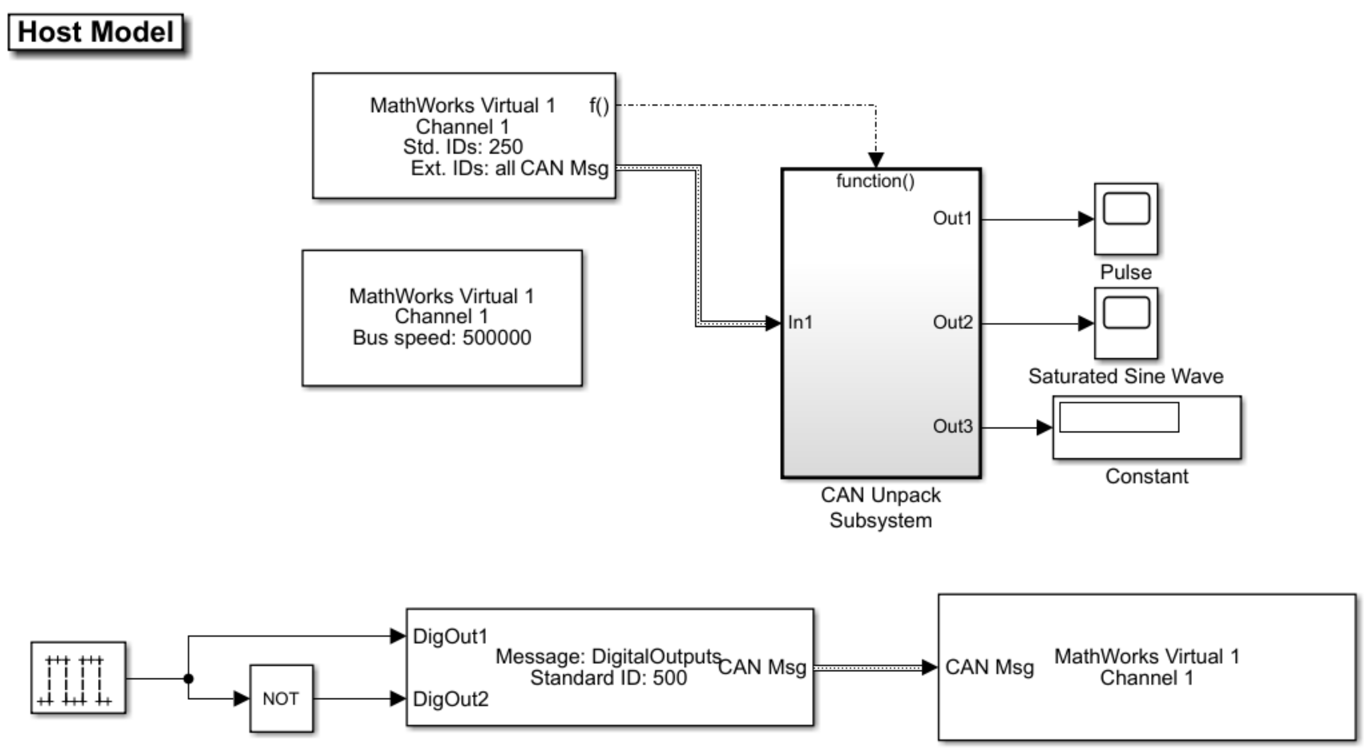
While the value for signal "Counter" changes at every time step in the message "Counter_Msg", it is received continuously due to the event-based transmission, and later there are a few more transmissions because periodic transmission is enabled.

Set up Communication Between Host and Target Models

This example shows you how to set up CAN communication between host-side CAN Vector blocks and target models. This example uses:

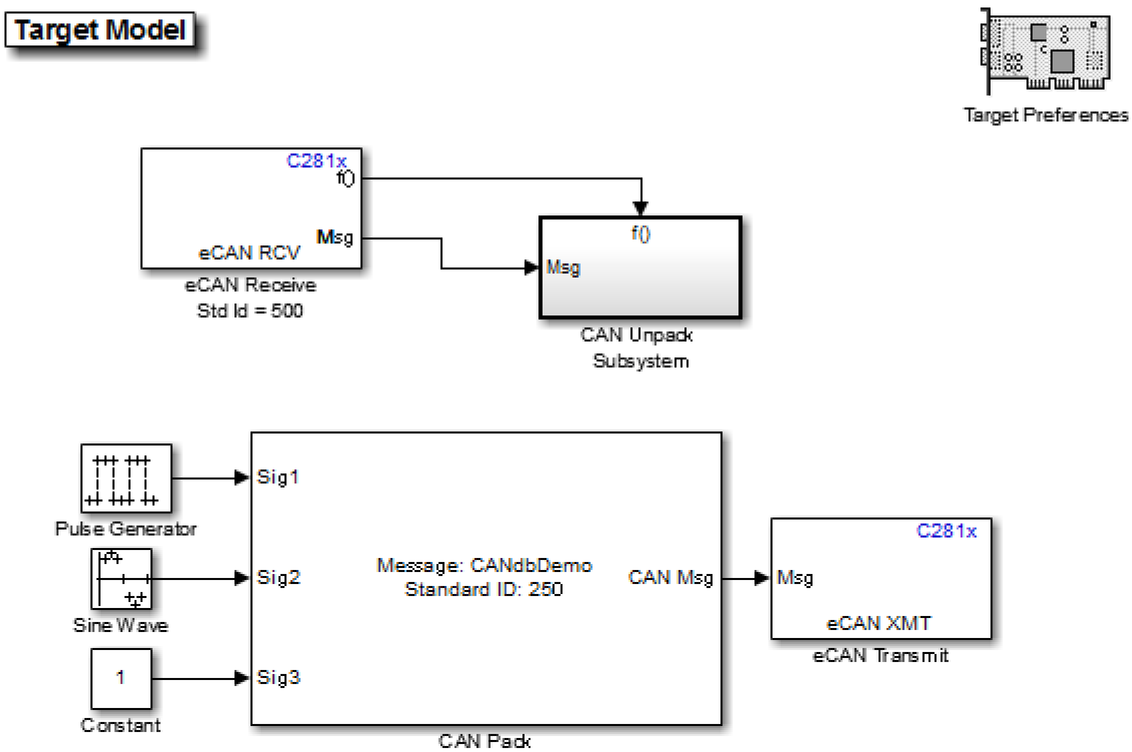
- The Embedded Coder™ product with CANcaseXL hardware to open and run the model.
- The Spectrum Digital F28335 eZdsp™ board to run the target model.

Transmit and Receive Using a Host Model



The host model receives CAN messages through Channel 1 of Vector CANcaseXL hardware. The model transmits CAN messages using Channel 1 of Vector hardware over the CAN bus.

Transmit and Receive Using a Target Model



This model illustrates how to use the CAN Pack and Unpack blocks to construct and inspect CAN messages. You can generate C281x specific code for this model.

If you have a CAN board and Vector CAN drivers installed, you can use the model demoVNTSL_CANMessaging_Host to exchange CAN messages with this model (running on C281x series hardware).

The target model contains the eCAN Receive and Transmit blocks that are packed and unpacked using the CAN Pack and Unpack blocks from Vehicle Network Toolbox™. To run this model successfully, the target model configuration settings done must match the host model configuration settings. The message that the target model receives controls the GPIO Digital outputs on the target DSP board.

Communication Between the Host and Target Models

Run the model demoVNTSL_CANMessaging_Target.slx on the target hardware.

Open the host side model demoVNTSL_CANMessaging_Host.slx.

Use the CAN Configuration block to configure a CAN channel on the Vector CAN hardware installed on your system.

Run the host communication model on your system.

CAN Messages are sent between the host model on your system and the target model running on your target hardware. The host receives, unpacks, and displays them using the display blocks and the scopes. The message transmitted by the host model controls the GPIO Digital outputs on the target hardware.

Vector CANcaseXL device was used for this example. You can however connect your models to other supported hardware.

Log and Replay CAN Messages

This example shows you how to log and replay CAN messages using MathWorks Virtual CAN channels in Simulink®. You can update this model to connect to supported hardware on your system.

Load the saved CAN message from `sourceMsgs.mat` file from the examples folder. The file contains CAN messages representing a 90 second drive cycle around a test track.

Convert these messages to a format compatible with the CAN Replay block and save it to a separate file.

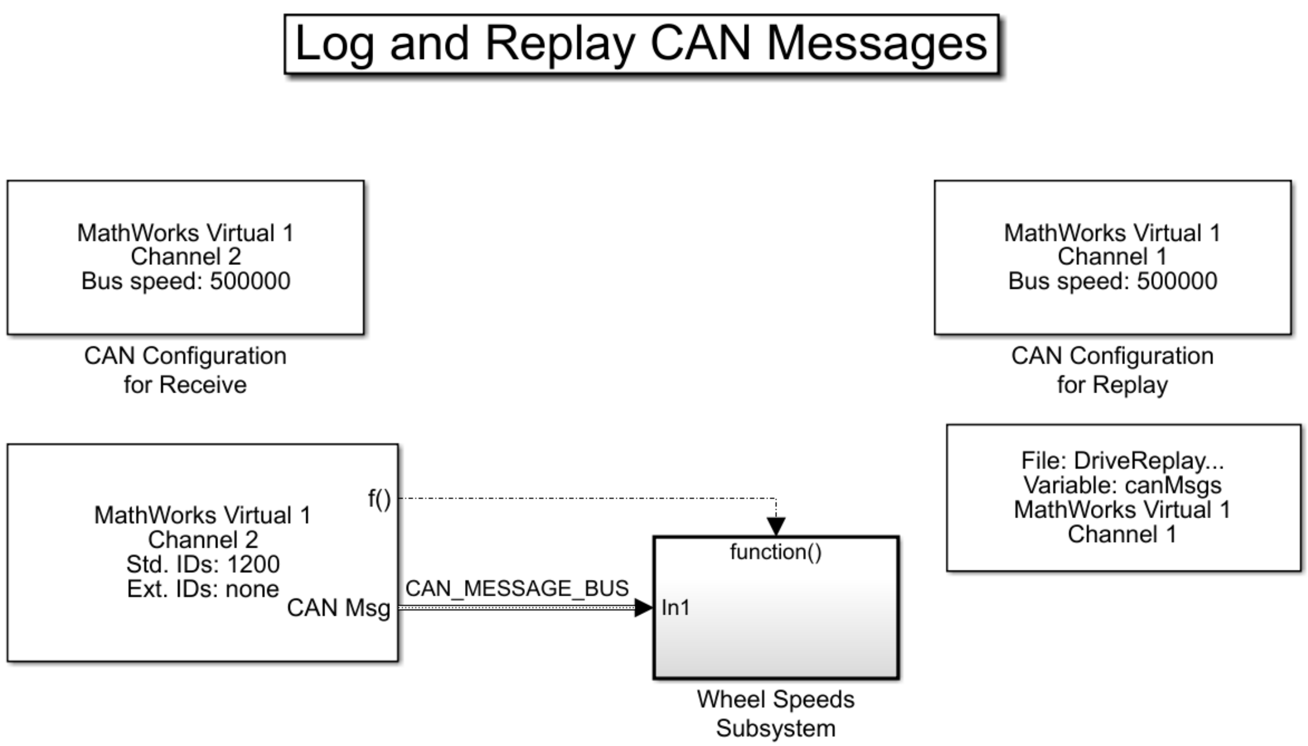
```
  Name                       Size                 Bytes  Class        Attributes

  canMsgTimetable      100000x8               33510851  timetable
  canMsgs                     1x1                2401176  struct
```

CAN Replay Model

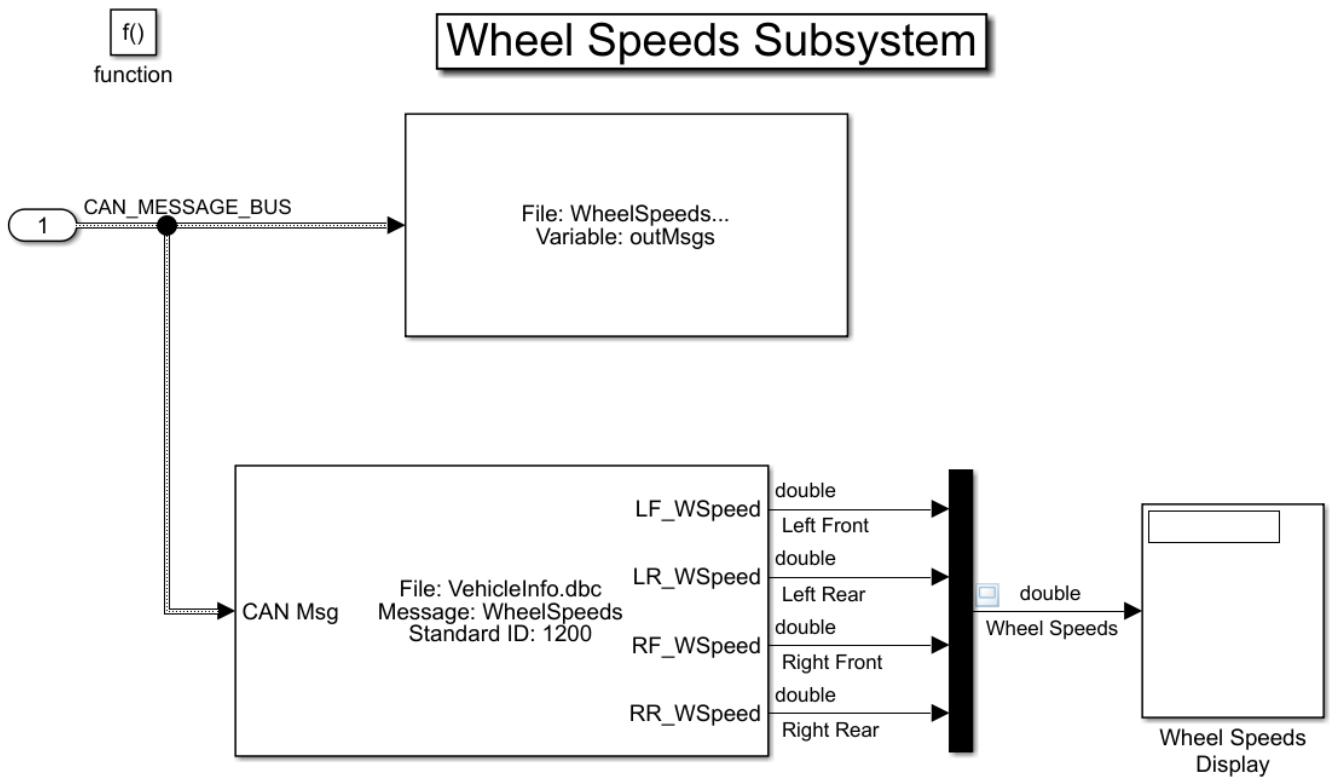
This model contains:

- A CAN Replay block that transmits to `MathWorks Virtual Channel 1`.
- A CAN Receive block that receives the messages on a CAN network, through `MathWorks Virtual Channel 2`.

The CAN Receive block is configured to block all extended IDs and allow only the `WheelSpeed` message with the standard ID `1200` to pass.

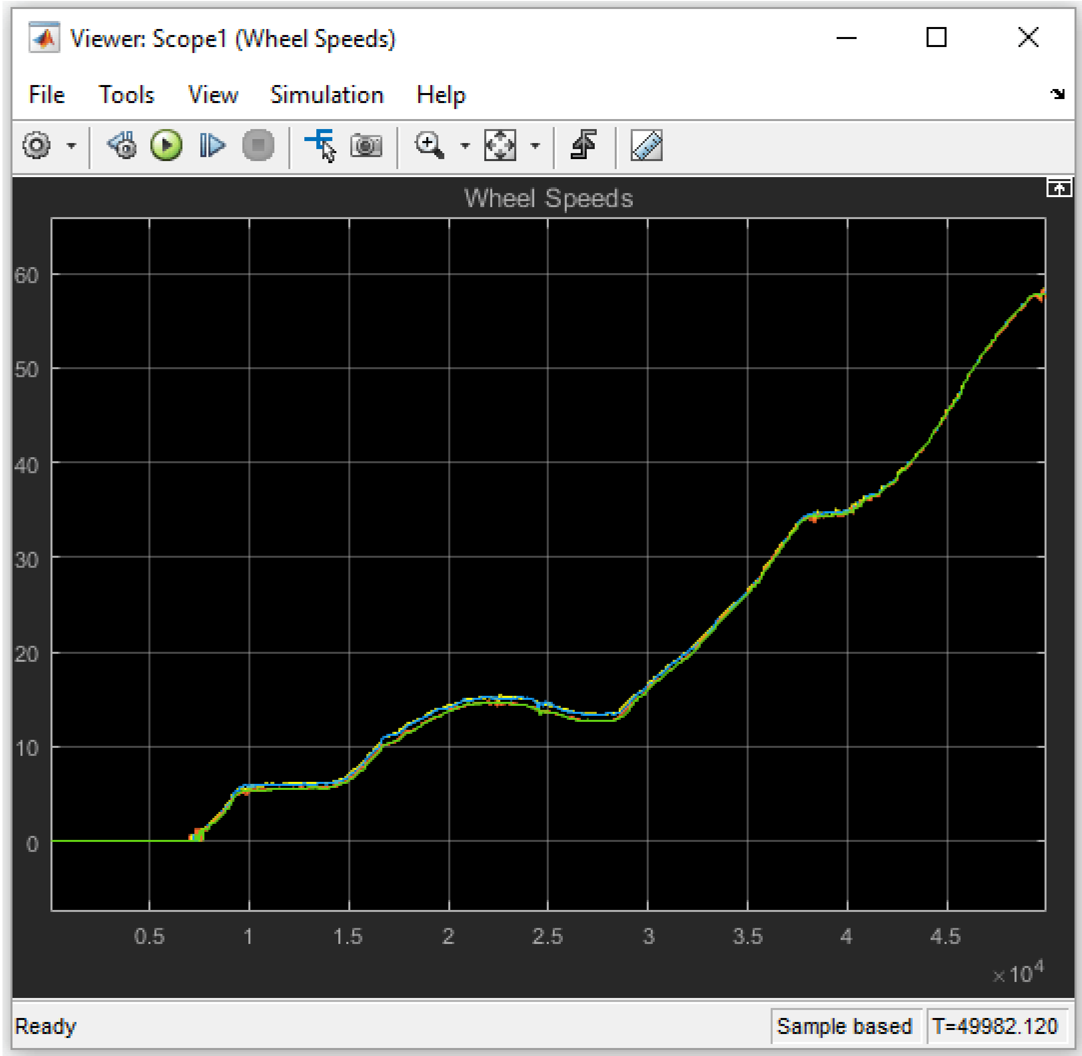


The Wheel Speeds subsystem unpacks the wheel speed information from the received CAN messages and plots them to a scope. The subsystem also logs the messages to a file.



Visualize Wheel Speed Information

The plot shows the wheel speed for all wheels for the duration of the test drive.



Load the Logged Message File

The CAN Log block creates a unique file each time you run the model. Use `dir` in the MATLAB Command Window to find the latest log file.

```
WheelSpeeds_2011-May-03_020634.mat

  Name                    Size                Bytes  Class        Attributes

  canMsgTimetable      100000x8            33510851  timetable
  canMsgs                   1x1             2401176  struct
  outMsgs                   1x1              154320  struct
```

Convert Logged Messages

Use `canMessageTimetable` to convert messages logged during the simulation to a timetable that you can use in the command window.

To access message signals directly, use the appropriate database file in the conversion along with `canSignalTimetable`.

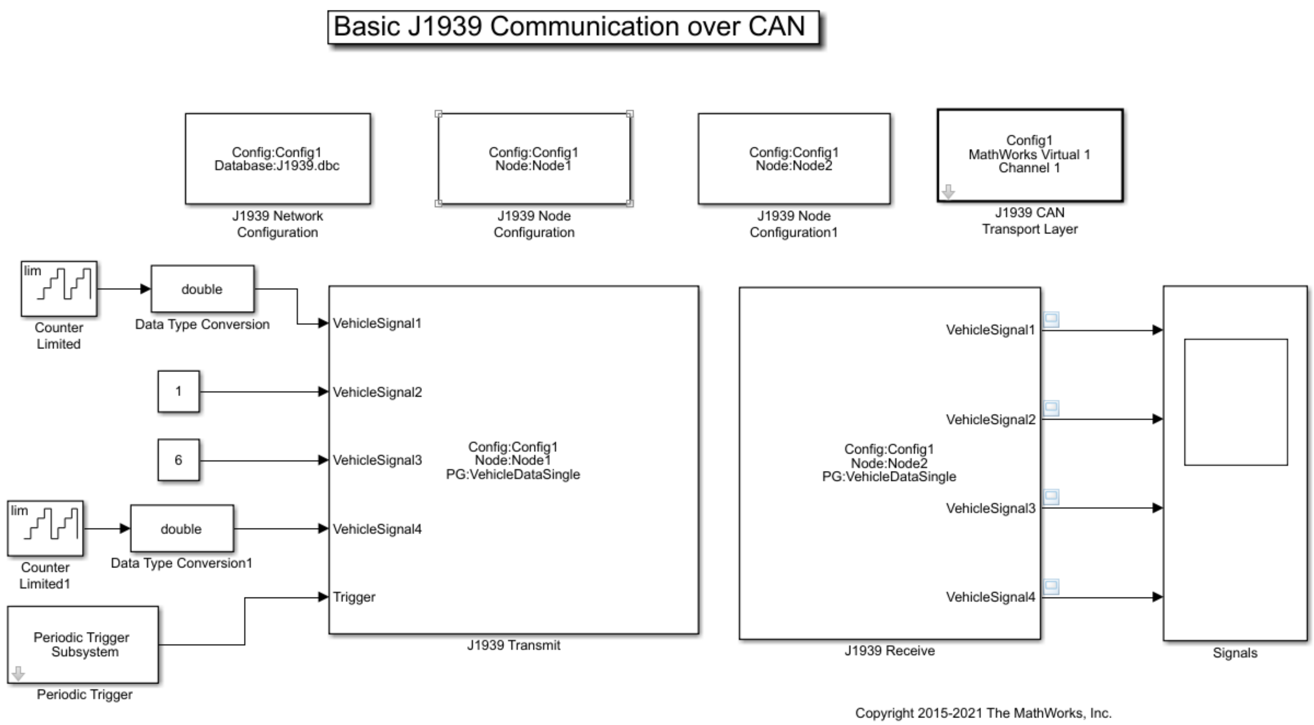
```
ans =

  15x8 timetable

       Time         ID     Extended         Name                      Data                Length
    ___________    ____    ________    _______________    ___________________________    ______

    0.10701 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.1153 sec     1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.12349 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.13178 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.13998 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.14826 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.15647 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.16475 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.17338 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.18122 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.18941 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.19768 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.20591 sec    1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.2142 sec     1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8
    0.2224 sec     1200     false      {'WheelSpeeds'}    {[39 16 39 16 39 16 39 16]}      8


ans =

  15x4 timetable

       Time        LR_WSpeed    RR_WSpeed    RF_WSpeed    LF_WSpeed
    ___________    _________    _________    _________    _________

    0.10701 sec        0            0            0            0
    0.1153 sec         0            0            0            0
    0.12349 sec        0            0            0            0
    0.13178 sec        0            0            0            0
    0.13998 sec        0            0            0            0
    0.14826 sec        0            0            0            0
    0.15647 sec        0            0            0            0
    0.16475 sec        0            0            0            0
    0.17338 sec        0            0            0            0
    0.18122 sec        0            0            0            0
    0.18941 sec        0            0            0            0
    0.19768 sec        0            0            0            0
    0.20591 sec        0            0            0            0
    0.2142 sec         0            0            0            0
    0.2224 sec         0            0            0            0
```

MathWorks CAN Virtual channels were used for this example. You can however connect your models to other supported hardware.

Get Started with J1939 Communication in Simulink

This example shows you how to use J1939 blocks to directly send and receive Parameter Group (PG) messages in Simulink.

Vehicle Network Toolbox provides J1939 Simulink blocks for receiving and transmitting Parameter Groups via Simulink models over Controller Area Networks (CAN). This example performs data transfer over a CAN bus using the J1939 Network Configuration, J1939 Node Configuration, J1939 CAN Transport Layer, J1939 Transmit and J1939 Receive blocks. It also uses MathWorks virtual CAN channels connected in a loopback configuration.



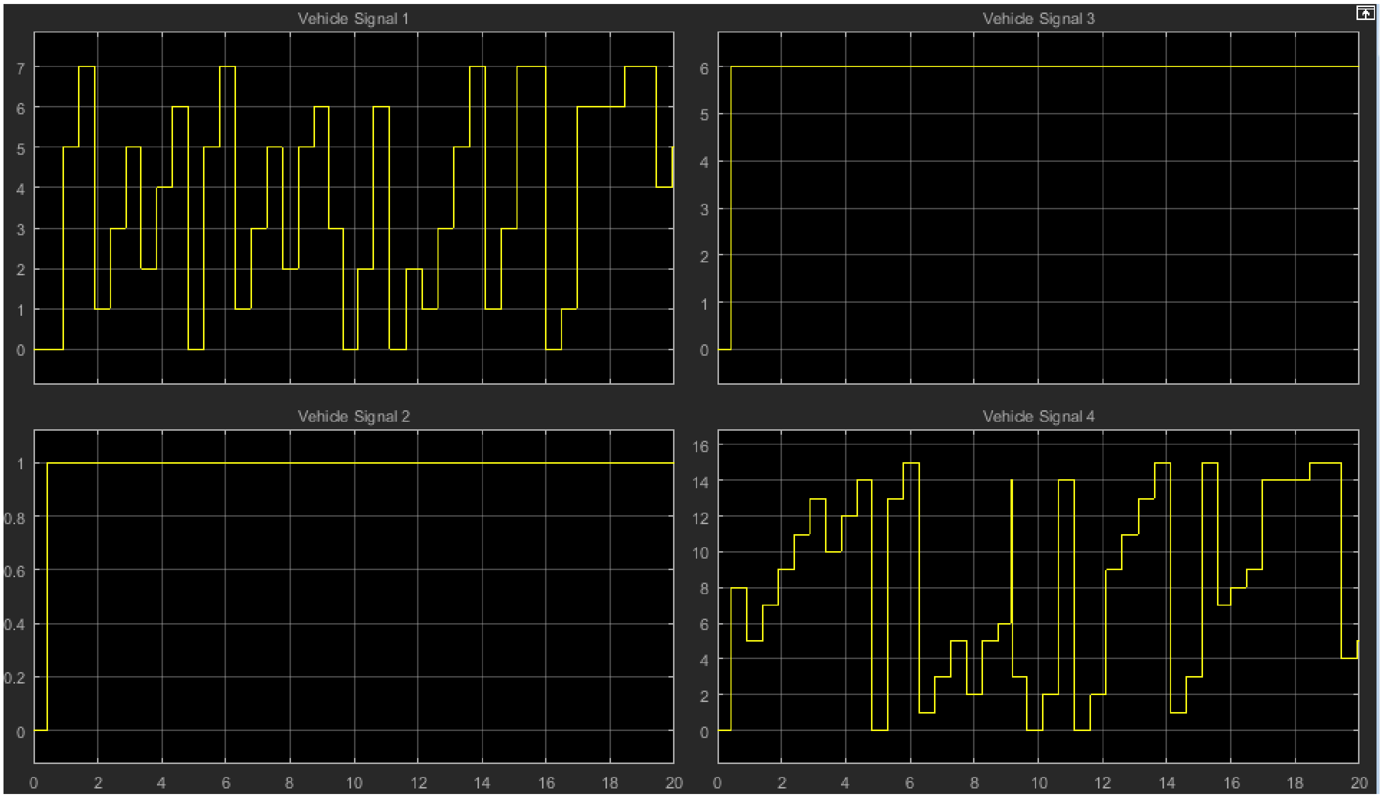
Set Up J1939 Block Parameters

Create a model to set up J1939 receive and transmit over the network. The model is configured to perform single frame transmission between two nodes defined in the J1939 DBC-file.

- Use a J1939 Network Configuration block and select file `J1939.dbc`. This J1939 database file consists of two nodes and a couple of single-frame and multiframe messages.
- Use a J1939 CAN Transport Layer block and set the Device to MathWorks virtual channel 1. The transport layer is configured to transfer J1939 messages over CAN via the specified virtual channel.
- Use basic Simulink source blocks to connect to a J1939 Transmit block. The J1939 Transmit block is set to queue data for transmit at each timestep when the Trigger port is enabled. For this example, a periodic trigger subsystem sends a high pulse every 50 milliseconds.
- Use the J1939 Receive block to receive the messages transmitted over the network.

Visualize Signals Received on the Network

Plot the results to see the vehicle signal values received over the network. The X-axis corresponds to the simulation timestep.



Get Started with MDF-Files

This example shows you how to open MDF-files and access information about the file and its contents.

Open an MDF-File

Open an MDF-file using `mdf` by specifying the name of the target file. Many basic details about the file are provided. This sample file was created using Vector CANape™.

```
m = mdf("CANapeBasic.MF4")
```

```
m = 
  MDF with properties:

   File Details
                 Name: 'CANapeBasic.MF4'
                 Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex51113426\CANape
               Author: 'Otmar Schneider'
           Department: 'PMC @ Vector Informatik GmbH'
              Project: 'Demo'
              Subject: 'XCPSim'
              Comment: 'Example file created with Vector CANape'
              Version: '4.10'
             DataSize: 176545
     InitialTimestamp: 2016-04-21 14:27:17.000010630

   Creator Details
    ProgramIdentifier: 'MCD14.02'
              Creator: [1x1 struct]

   File Contents
           Attachment: [0x1 struct]
         ChannelNames: {2x1 cell}
         ChannelGroup: [1x2 struct]

   Options
           Conversion: Numeric
```

View File Creation Details

Information about the originating tool of the MDF-file is found in the `Creator` property.

```
m.Creator
```

```
ans = struct with fields:
     VendorName: 'Vector Informatik GmbH'
       ToolName: 'CANape'
    ToolVersion: '14.0.20.2386'
       UserName: 'visosr'
        Comment: 'created'
```

View Channel Group Details

Data in an MDF-file is stored in channels contained within channel groups. This sample file contains two channel groups.

```
m.ChannelGroup(1)
```

```
ans = struct with fields:
    AcquisitionName: '10 ms'
            Comment: '10 ms'
         NumSamples: 1993
           DataSize: 153461
             Sorted: 1
            Channel: [74x1 struct]
```

```
m.ChannelGroup(2)
```

```
ans = struct with fields:
    AcquisitionName: '100ms'
            Comment: '100ms'
         NumSamples: 199
           DataSize: 23084
             Sorted: 1
            Channel: [46x1 struct]
```

View Channel Details

Within a channel group, details about each channel are stored.

```
m.ChannelGroup(1).Channel(1)
```

```
ans = struct with fields:
                  Name: 'Counter_B4'
           DisplayName: ''
    ExtendedNamePrefix: 'XCPsim'
           Description: 'Single bit demo signal (bit from a byte shifting)'
               Comment: 'Single bit demo signal (bit from a byte shifting)'
                  Unit: ''
                  Type: FixedLength
              DataType: IntegerUnsignedLittleEndian
               NumBits: 1
         ComponentType: None
       CompositionType: None
        ConversionType: ValueToText
```

Quickly Access Channels Names

The `ChannelNames` property allows quick access to find specific channels within the various channel groups.

```
m.ChannelNames
```

```
ans=2×1 cell array
    {74x1 cell}
    {46x1 cell}
```

```
m.ChannelNames{1}
```

```
ans = 74x1 cell
    {'Counter_B4'              }
```

```
{'Counter_B5'            }
{'Counter_B6'            }
{'Counter_B7'            }
{'PWM'                   }
{'PWM_Level'             }
{'PWMFiltered'           }
{'Triangle'              }
{'map1_8_8_uc_measure[0][0]'}
{'map1_8_8_uc_measure[0][1]'}
{'map1_8_8_uc_measure[0][2]'}
{'map1_8_8_uc_measure[0][3]'}
{'map1_8_8_uc_measure[0][4]'}
{'map1_8_8_uc_measure[0][5]'}
{'map1_8_8_uc_measure[0][6]'}
{'map1_8_8_uc_measure[0][7]'}
{'map1_8_8_uc_measure[1][0]'}
{'map1_8_8_uc_measure[1][1]'}
{'map1_8_8_uc_measure[1][2]'}
{'map1_8_8_uc_measure[1][3]'}
{'map1_8_8_uc_measure[1][4]'}
{'map1_8_8_uc_measure[1][5]'}
{'map1_8_8_uc_measure[1][6]'}
{'map1_8_8_uc_measure[1][7]'}
{'map1_8_8_uc_measure[2][0]'}
{'map1_8_8_uc_measure[2][1]'}
{'map1_8_8_uc_measure[2][2]'}
{'map1_8_8_uc_measure[2][3]'}
{'map1_8_8_uc_measure[2][4]'}
{'map1_8_8_uc_measure[2][5]'}
  ⋮
```

Find Channels in an MDF-File

The `channelList` function is available to quickly and easily query for channel details within an MDF-file. It returns a case-insensitive, partial match to the provided input by default, but an exact match can also be used.

```
channelList(m, "PWM")
```

ans=*3×9 table*

ChannelName	ChannelGroupNumber	ChannelGroupNumSamples	ChannelGroupAcquisitionName
"PWM"	1	1993	10 ms
"PWM_Level"	1	1993	10 ms
"PWMFiltered"	1	1993	10 ms

```
channelList(m, "PWM", "ExactMatch", true)
```

ans=*1×9 table*

ChannelName	ChannelGroupNumber	ChannelGroupNumSamples	ChannelGroupAcquisitionName
"PWM"	1	1993	10 ms

Close the File

Close access to the MDF-file by clearing its variable from the workspace.

```
clear m
```

Read Data from MDF-Files

This example shows you how to read channel data from an MDF-file.

Open the MDF-file

Before reading channel data from an MDF-file, open access to the file with the `mdf` command.

```
m = mdf("CANapeReadDemo.MF4")
```

```
m = 
  MDF with properties:

   File Details
                 Name: 'CANapeReadDemo.MF4'
                 Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex94427230\CANape
               Author: 'Otmar Schneider'
           Department: 'PMC @ Vector Informatik GmbH'
              Project: 'Demo'
              Subject: 'XCPSim'
              Comment: 'Example file created with Vector CANape'
              Version: '4.10'
             DataSize: 176545
     InitialTimestamp: 2016-04-21 14:27:17.000010630

   Creator Details
    ProgramIdentifier: 'MCD14.02'
              Creator: [1x1 struct]

   File Contents
           Attachment: [0x1 struct]
         ChannelNames: {2x1 cell}
         ChannelGroup: [1x2 struct]

   Options
           Conversion: Numeric
```

Specify Data to Read

The `read` command is used to retrieve data from the MDF-file with several variations. It requires two primary arguments. One is the numeric index of the channel group from which to read. Second is the name(s) of the channels in the channel group to read. Information about these items is available from the MDF-file.

```
m.ChannelGroup(1)
```

```
ans = struct with fields:
    AcquisitionName: '10 ms'
            Comment: '10 ms'
         NumSamples: 1993
           DataSize: 153461
             Sorted: 1
            Channel: [74x1 struct]
```

```
m.ChannelNames{1}
```

```
ans = 74x1 cell
    {'Counter_B4'            }
    {'Counter_B5'            }
    {'Counter_B6'            }
    {'Counter_B7'            }
    {'PWM'                   }
    {'PWM_Level'             }
    {'PWMFiltered'           }
    {'Triangle'              }
    {'map1_8_8_uc_measure[0][0]'}
    {'map1_8_8_uc_measure[0][1]'}
    {'map1_8_8_uc_measure[0][2]'}
    {'map1_8_8_uc_measure[0][3]'}
    {'map1_8_8_uc_measure[0][4]'}
    {'map1_8_8_uc_measure[0][5]'}
    {'map1_8_8_uc_measure[0][6]'}
    {'map1_8_8_uc_measure[0][7]'}
    {'map1_8_8_uc_measure[1][0]'}
    {'map1_8_8_uc_measure[1][1]'}
    {'map1_8_8_uc_measure[1][2]'}
    {'map1_8_8_uc_measure[1][3]'}
    {'map1_8_8_uc_measure[1][4]'}
    {'map1_8_8_uc_measure[1][5]'}
    {'map1_8_8_uc_measure[1][6]'}
    {'map1_8_8_uc_measure[1][7]'}
    {'map1_8_8_uc_measure[2][0]'}
    {'map1_8_8_uc_measure[2][1]'}
    {'map1_8_8_uc_measure[2][2]'}
    {'map1_8_8_uc_measure[2][3]'}
    {'map1_8_8_uc_measure[2][4]'}
    {'map1_8_8_uc_measure[2][5]'}
      ⋮
```

Read a Subset of Data by Index

To read just a subset of data by index, the index range is provided as input to the `read` command.

```
data = read(m, 1, m.ChannelNames{1}, 1, 10)
```

```
data=10×74 timetable
         Time          Counter_B4    Counter_B5    Counter_B6    Counter_B7    PWM    PWM_Level
    ______________    __________    __________    __________    __________    ___    _________

    0.00082554 sec        0             0             1             0         100        0
    0.010826 sec          0             0             1             0         100        0
    0.020826 sec          0             0             1             0         100        0
    0.030826 sec          0             0             1             0         100        0
    0.040826 sec          0             0             1             0         100        0
    0.050826 sec          0             0             1             0         100        0
    0.060826 sec          0             0             1             0         100        0
    0.070826 sec          0             0             1             0         100        0
    0.080826 sec          0             0             1             0         100        0
    0.090826 sec          0             0             1             0         100        0
```

Read a Specific Data Value by Index

Providing a single numeric index argument will retrieve the data values at that index.

```
data = read(m, 1, m.ChannelNames{1}, 5)
```

```
data=1×74 timetable
       Time         Counter_B4    Counter_B5    Counter_B6    Counter_B7    PWM    PWM_Level    P
    ____________    __________    __________    __________    __________    ___    _________    _

    0.040826 sec        0             0             1             0         100        0
```

Read a Subset of Data by Time

To read a subset of data by time, duration arguments are provided as input to the `read` command.

```
data = read(m, 1, m.ChannelNames{1}, seconds(0.50), seconds(0.60))
```

```
data=10×74 timetable
      Time        Counter_B4    Counter_B5    Counter_B6    Counter_B7    PWM    PWM_Level    PW
    ___________    __________    __________    __________    __________    ___    _________    __

    0.50083 sec        1             1             1             0          0         0
    0.51083 sec        1             1             1             0          0         0
    0.52083 sec        1             1             1             0          0         0
    0.53083 sec        1             1             1             0          0         0
    0.54083 sec        1             1             1             0          0         0
    0.55083 sec        1             1             1             0          0         0
    0.56083 sec        1             1             1             0          0         0
    0.57083 sec        1             1             1             0          0         0
    0.58083 sec        1             1             1             0          0         0
    0.59083 sec        1             1             1             0          0         0
```

Read a Specific Data Value by Time

Providing a single duration will retrieve the data values at or closest to that timestamp.

```
data = read(m, 1, m.ChannelNames{1}, seconds(0.55))
```

```
data=1×74 timetable
      Time        Counter_B4    Counter_B5    Counter_B6    Counter_B7    PWM    PWM_Level    PW
    ___________    __________    __________    __________    __________    ___    _________    __

    0.55083 sec        1             1             1             0          0         0
```

Output Format Defaults to Timetable

The default output format of the `read` command is a timetable. This option can also be controlled with the `OutputFormat` argument.

```
data = read(m, 1, "Triangle", 1, 10, "OutputFormat", "timetable")
```

```
data=10×1 timetable
         Time          Triangle
    ______________    ________

    0.00082554 sec       18
    0.010826 sec         17
    0.020826 sec         16
    0.030826 sec         15
```

```
    0.040826 sec         14
    0.050826 sec         13
    0.060826 sec         12
    0.070826 sec         11
    0.080826 sec         10
    0.090826 sec          9
```

Output Data as Timeseries

If a timeseries is desired as output, the `OutputFormat` can be specified to the `read` command. When outputting data as a timeseries, only a single channel may be read at a time.

```
data = read(m, 1, "Triangle", 1, 10, "OutputFormat", "timeseries")
```

```
  timeseries

  Common Properties:
            Name: 'Triangle'
            Time: [10x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [10x1 int8]
        DataInfo: tsdata.datametadata
```

Output Data as Vectors

Output from the `read` command can also be specified as vectors. When outputting data as a vector, only a single channel may be read at a time.

```
[data, time] = read(m, 1, "Triangle", 1, 10, "OutputFormat", "vector")
```

```
data = 10x1 int8 column vector

   18
   17
   16
   15
   14
   13
   12
   11
   10
    9


time = 10×1

    0.0008
    0.0108
    0.0208
    0.0308
    0.0408
    0.0508
    0.0608
    0.0708
    0.0808
    0.0908
```

Read an Entire Channel Group

To quickly read the data from an entire channel group in a single call, no additional arguments are specified to the `read` command.

```
data = read(m, 1, m.ChannelNames{1});
```

Close the File

Close access to the MDF-file by clearing its variable from the workspace.

```
clear m
```

Get Started with MDF Datastore

This example shows you how to use the MDF datastore feature of Vehicle Network Toolbox™ to quickly and efficiently process a data set spread across a collection of multiple MDF-files. This workflow is also valuable when there are too much data to fit into available memory.

Access MDF-Files in a Datastore

Find the collection of MDF-files representing logged information from multiple test sequences. Note that MDF-files to be used by MDF datastore as a set must have the same channel group and channel content structure.

```
dir("File*.mf4")
```

```
File01.mf4  File02.mf4  File03.mf4  File04.mf4  File05.mf4
```

Create an MDF Datastore

You create an MDF datastore by selecting a folder location containing a collection of MDF-files. In this case, target all files in the current working directory.

```
mds = mdfDatastore(pwd)
```

```
mds = 
  MDFDatastore with properties:

  DataStore Details
                       Files: {
                              ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                              ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                              ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                               ... and 2 more
                              }
               ChannelGroups: 
                                ChannelGroupNumber    AcquisitionName        Comment
                                __________________    _______________    _________________

                                        1               {0x0 char}       {'Integer Types'}
                                        2               {0x0 char}       {'Float Types'  }


                    Channels: 
                                ChannelGroupNumber              ChannelName
                                __________________    ______________________________________

                                        1             {'Signed_Int16_LE_Offset_32'         }
                                        1             {'Unsigned_UInt32_LE_Master_Offset_0'}
                                        2             {'Float_32_LE_Offset_64'             }

                              ... and 1 more rows

  Options
         SelectedChannelNames: {
                              'Signed_Int16_LE_Offset_32';
                              'Unsigned_UInt32_LE_Master_Offset_0'
                              }
    SelectedChannelGroupNumber: 1
```

```
                 ReadSize: 'file'
               Conversion: Numeric
```

Configure MDF Datastore

Multiple options allow control of what data are read from the MDF-files and how the reads are performed. In this case, the first channel group is used by default. Note that only one channel group may be selected by the datastore at a time. You can also specify certain channels within the selected channel group to read. In this case, all channels are read by default.

```
mds.SelectedChannelGroupNumber
```

```
ans = 1
```

```
mds.SelectedChannelNames
```

```
ans = 2x1 string
    "Signed_Int16_LE_Offset_32"
    "Unsigned_UInt32_LE_Master_Offset_0"
```

Preview MDF Datastore

Using the `preview` function, you can obtain a quick view of the data available in the file set. Preview always returns up to eight data points from the first file in the datastore.

```
preview(mds)
```

```
ans=8×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec                0                                0
    1 sec                1                                1
    2 sec                2                                2
    3 sec                3                                3
    4 sec                4                                4
    5 sec                5                                5
    6 sec                6                                6
    7 sec                7                                7
```

Read All Data in MDF Datastore

You can use the `readall` function to read the entire data in a single call. This is an efficient way to read from many files when the data set fits into available memory. After running `readall`, the datastore resets to the beginning of the data set.

```
data = readall(mds);
data(1:5,:)
```

```
ans=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec                0                                0
    1 sec                1                                1
    2 sec                2                                2
```

```
    3 sec              3                                 3
    4 sec              4                                 4
```

Read a Subset of Data in MDF Datastore

You can use the `read` function to obtain a subset of data from the datastore. The size of the subset is determined by the `ReadSize` property of the MDF datastore object. By default, data from an entire file are read in one call. The power of a datastore comes from reading through multiple files sequentially within the file set. As you read, the datastore automatically bridges from one file to the next until all data from all files are read.

```
for ii = 1:3
    data = read(mds);
    whos("data")
    data(1:5,:)
end
```

```
  Name          Size              Bytes  Class        Attributes

  data      10000x2              141839  timetable

ans=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec              0                                 0
    1 sec              1                                 1
    2 sec              2                                 2
    3 sec              3                                 3
    4 sec              4                                 4

  Name          Size              Bytes  Class        Attributes

  data      10000x2              141839  timetable

ans=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec              0                                 0
    1 sec              1                                 1
    2 sec              2                                 2
    3 sec              3                                 3
    4 sec              4                                 4

  Name          Size              Bytes  Class        Attributes

  data      10000x2              141839  timetable

ans=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec              0                                 0
    1 sec              1                                 1
    2 sec              2                                 2
```

```
    3 sec              3                                3
    4 sec              4                                4
```

Reset MDF Datastore

At any time, you can call the `reset` function to start over at the beginning of the data set.

```
reset(mds)
```

Configure Number of Records to Read from MDF Datastore

You can use the `ReadSize` property to specify how much data to read on each call. `ReadSize` can be specified as a numeric value to read a fixed number of data points. `ReadSize` lets you control how much data is loaded into memory when you have a data set larger than available memory. It is recommended to use custom read sizes that are small enough to fit in memory, but still as large as possible to reduce processing overhead and improve performance.

```
mds.ReadSize = 5
```

```
mds = 
  MDFDatastore with properties:

  DataStore Details
                         Files: {
                                ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                                ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                                ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                                 ... and 2 more
                                }
                 ChannelGroups: 
                                  ChannelGroupNumber    AcquisitionName        Comment
                                  __________________    _______________    _________________

                                          1               {0x0 char}       {'Integer Types'}
                                          2               {0x0 char}       {'Float Types'  }


                      Channels: 
                                  ChannelGroupNumber              ChannelName
                                  __________________    ______________________________________

                                          1             {'Signed_Int16_LE_Offset_32'          }
                                          1             {'Unsigned_UInt32_LE_Master_Offset_0'}
                                          2             {'Float_32_LE_Offset_64'              }

                                ... and 1 more rows

  Options
         SelectedChannelNames: {
                                'Signed_Int16_LE_Offset_32';
                                'Unsigned_UInt32_LE_Master_Offset_0'
                                }
    SelectedChannelGroupNumber: 1
                      ReadSize: 5
                    Conversion: Numeric
```

```
for ii = 1:3
    data = read(mds)
end
```

```
data=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec                0                                 0
    1 sec                1                                 1
    2 sec                2                                 2
    3 sec                3                                 3
    4 sec                4                                 4

data=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    5 sec                5                                 5
    6 sec                6                                 6
    7 sec                7                                 7
    8 sec                8                                 8
    9 sec                9                                 9

data=5×2 timetable
     Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    ______    _________________________    __________________________________

    10 sec               10                                 10
    11 sec               11                                 11
    12 sec               12                                 12
    13 sec               13                                 13
    14 sec               14                                 14
```

Configure a Time Range to Read from MDF Datastore

You can also specify `ReadSize` as a duration to read data points by elapsed time. Note that when the read type is changed, the datastore resets to the beginning of the data set.

```
mds.ReadSize = seconds(5)
```

```
mds = 
  MDFDatastore with properties:

  DataStore Details
                         Files: {
                                ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                                ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                                ' ...\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex10761765\
                                 ... and 2 more
                                }
                 ChannelGroups: 
                                  ChannelGroupNumber    AcquisitionName        Comment
                                  __________________    _______________    _________________

                                          1               {0x0 char}       {'Integer Types'}
```

```
                                        2              {0x0 char}       {'Float Types'  }    ·


                     Channels:
                                ChannelGroupNumber                 ChannelName
                                __________________    ______________________________________

                                        1             {'Signed_Int16_LE_Offset_32'          }
                                        1             {'Unsigned_UInt32_LE_Master_Offset_0'}
                                        2             {'Float_32_LE_Offset_64'              }

                               ... and 1 more rows

  Options
          SelectedChannelNames: {
                               'Signed_Int16_LE_Offset_32';
                               'Unsigned_UInt32_LE_Master_Offset_0'
                               }
    SelectedChannelGroupNumber: 1
                      ReadSize: 5 sec
                    Conversion: Numeric
```

```
for ii = 1:3
    data = read(mds)
end
```

```
data=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    0 sec                0                                0
    1 sec                1                                1
    2 sec                2                                2
    3 sec                3                                3
    4 sec                4                                4


data=5×2 timetable
    Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    _____    _________________________    __________________________________

    5 sec                5                                5
    6 sec                6                                6
    7 sec                7                                7
    8 sec                8                                8
    9 sec                9                                9


data=5×2 timetable
     Time     Signed_Int16_LE_Offset_32    Unsigned_UInt32_LE_Master_Offset_0
    ______    _________________________    __________________________________

    10 sec               10                               10
    11 sec               11                               11
    12 sec               12                               12
    13 sec               13                               13
    14 sec               14                               14
```

Close MDF-Files

Close access to the MDF-files by clearing the MDF datastore variable from workspace.

```
clear mds
```

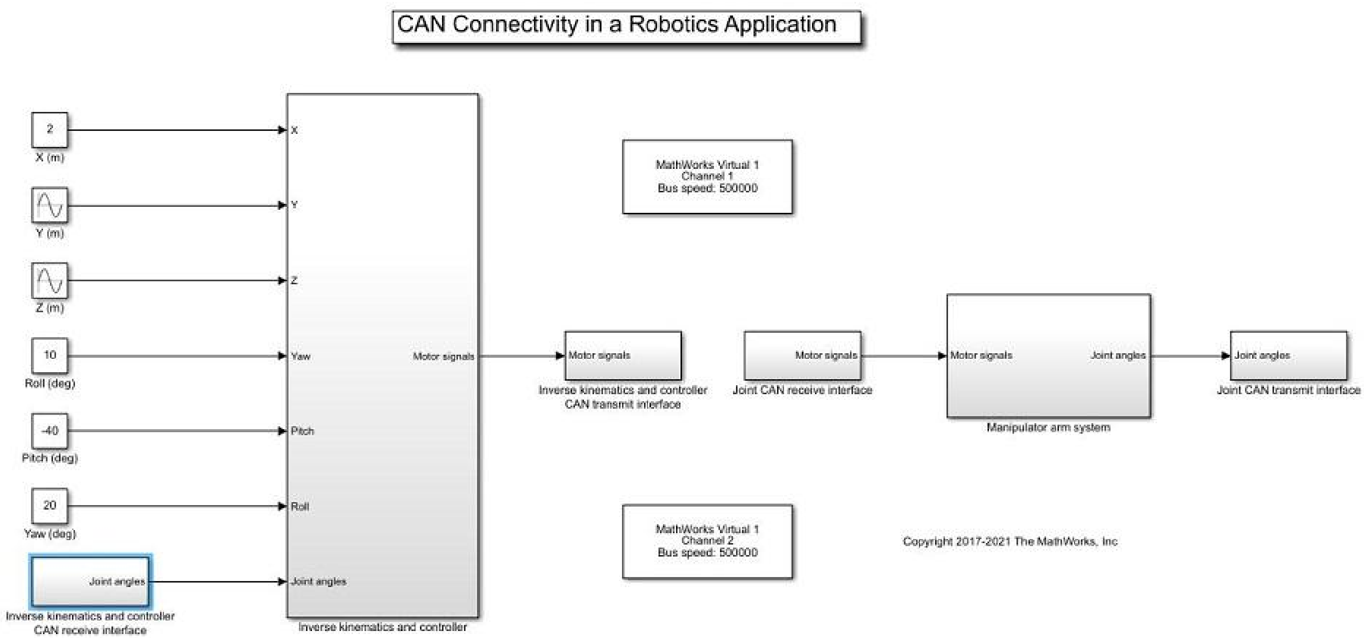
CAN Connectivity in a Robotics Application

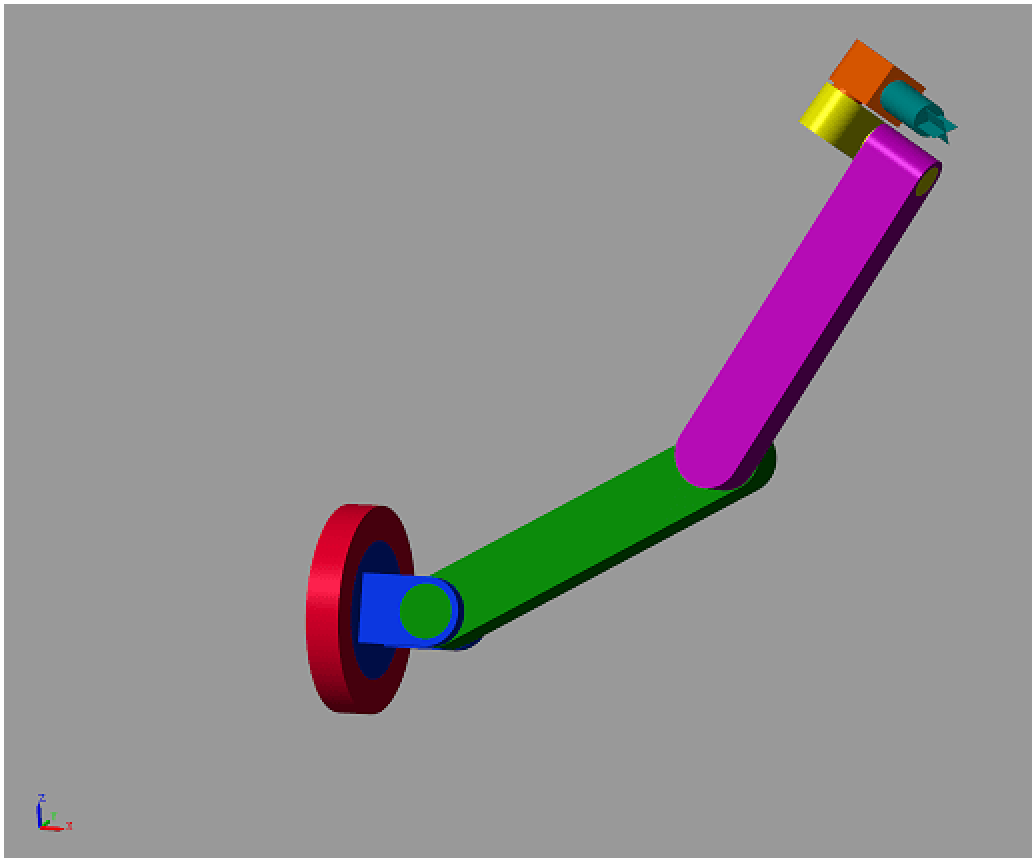
This example shows you how to use Vehicle Network Toolbox™ to implement a Controller Area Network (CAN) in a remote manipulator arm using Simulink®. The CAN messages used are defined in the CAN database file, `canDatabaseFor6DofRoboticArm.dbc`.

Vehicle Network Toolbox provides Simulink blocks for transmitting and receiving live messages via Simulink models over Controller Area Networks (CAN). This example uses the CAN Configuration, CAN Pack, CAN Transmit, CAN Receive, and CAN Unpack blocks to perform data transfer over a CAN bus.

MathWorks virtual CAN channels are used for this example. Alternatively, you can connect your models to other supported hardware.

Model Description





The model consists of the following subsystems: Manipulator arm system, Inverse kinematics and controller, Joint CAN transmit interface, Joint CAN receive interface, Inverse kinematics and controller CAN transmit interface, and Inverse kinematics and controller CAN receive interface. Each joint and the inverse kinematics and controller subsystem constitute a node in the CAN bus.

The user inputs the position coordinates (X, Y and Z in metres) and the orientation (roll, pitch and yaw angles in degrees, in body-3 2-3-1 sequence) of the end effector. The inverse kinematics and controller subsystem receives feedback from the joint angle sensors that are sent via the CAN bus, and sends appropriate commands to each joint motor via the CAN bus to drive the position anf the orientation of the end effector to the user-input values.

The remote manipulator arm is assumed to be attached to a spacecraft in orbit. As a result, gravity is neglected.

Manipulator Arm System

This subsystem consists of the rigid-body model of the remote manipulator arm, modeled using Simscape Multibody 2G. The arm has six joints. Each joint is actuated by a DC motor with a gearbox, and are modeled using Simscape Foundational Library. Each joint also has a joint angle sensor. The sensor data in sent into the CAN bus. Each motor is powered by a controlled voltage source. The voltage sources receive messages from the CAN bus and apply the DC voltage across their terminals corresponding to the information in the messages.

Inverse Kinematics and Controller

The Inverse kinematics and controller subsystem further implements the inverse kinematics and the control algorithm. The inverse kinematics computes the desired joint angles from the desired position (X, Y and Z) and orientation (roll, pitch and yaw angles) that are input by the user. The discrete PID controllers utilize the joint angle sensor values that are read from the CAN bus to determine the DC voltage that should be applied to each motor to drive the joint angles to the desired values. The DC voltage values are sent as messages in the CAN bus.

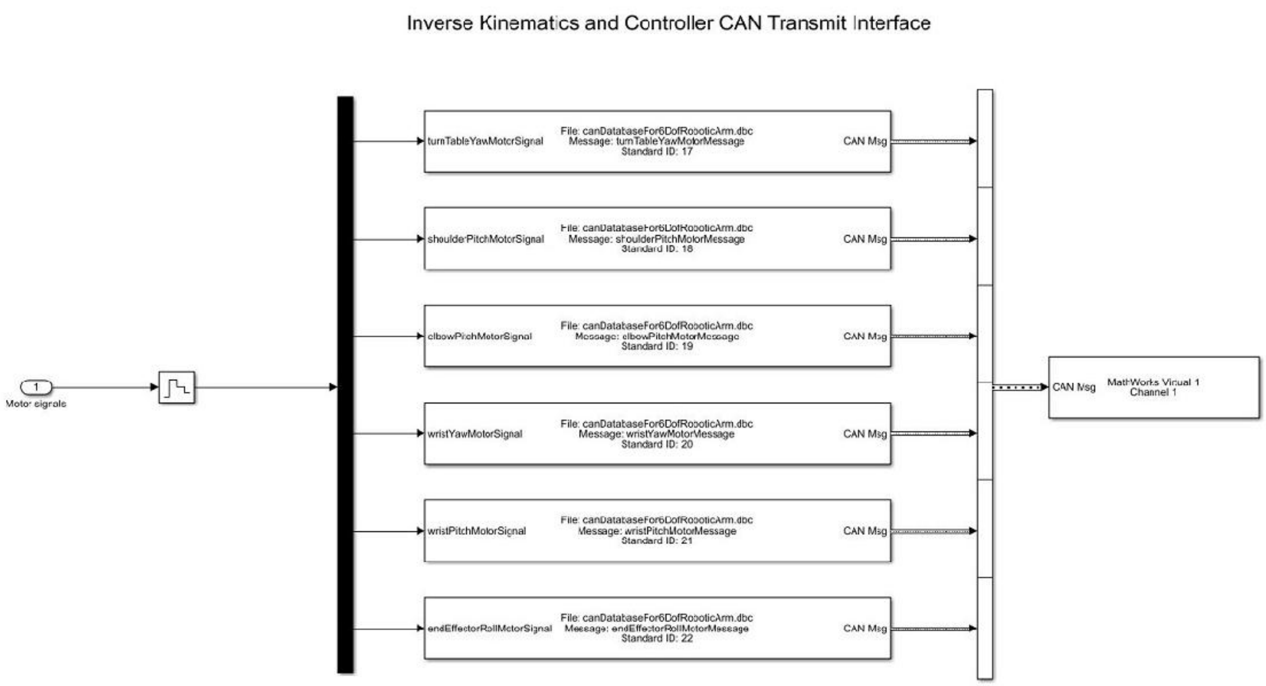
Joint CAN Transmit Interface

This subsystem consists of the VNT blocks that are necessary to transmit the joint angle values from the corresponding sensors into the CAN bus.

Joint CAN Receive Interface

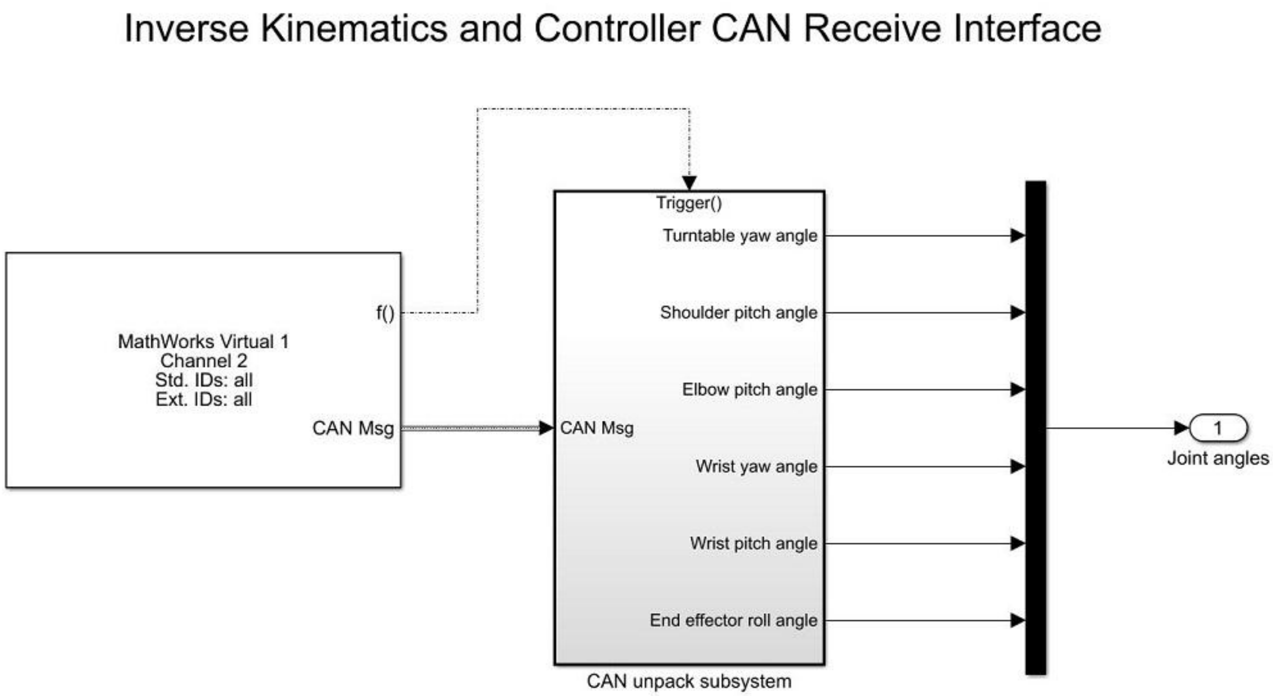
This subsystem consists of the VNT blocks that are necessary to receive and unpack the messages from the CAN bus that contain information about the DC voltages that need to be applied to the controlled voltage sources corresponding to each motor.

Inverse Kinematics and Controller CAN Transmit Interface



This subsystem consists of the VNT blocks that are necessary to transmit the motor signals (DC voltages that need to be applied across the controlled voltage sources) calculated by the Inverse Kinematics and Controller subsystem into the CAN bus.

Inverse Kinematics and Controller CAN Receive Interface



This subsystem consists of the VNT blocks that are necessary to receive the messages from the CAN bus that contain information about the joint angles that are sent by the joint angle sensors.

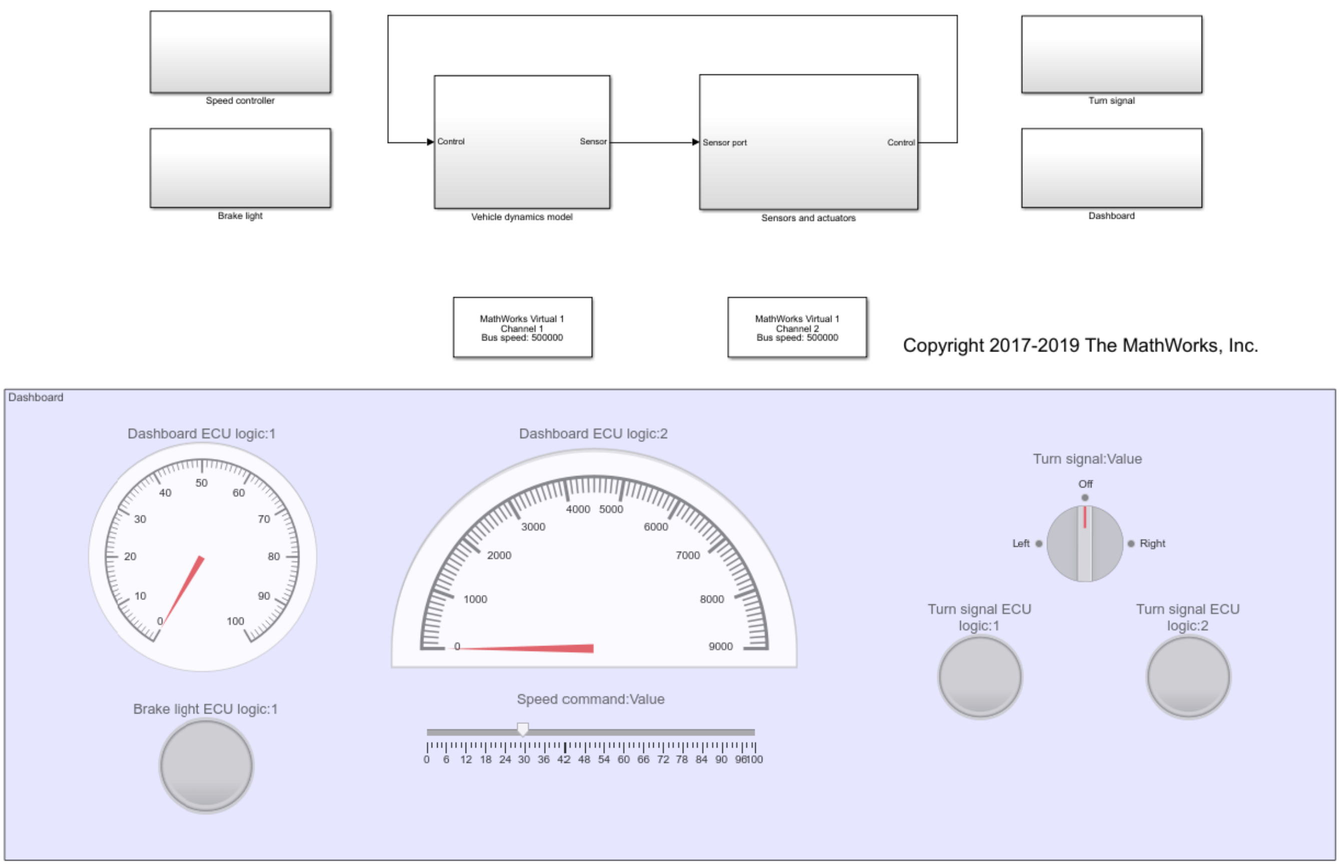
CAN Connectivity in an Automotive Application

This example uses Vehicle Network Toolbox to implement a distributed Electronic Control Unit (ECU) network on CAN for an automobile using Simulink®. The CAN messages used are defined in the CAN database file, `canConnectivityForVehicle.dbc`.

Vehicle Network Toolbox™ provides Simulink blocks for transmitting and receiving live messages via Simulink models over Controller Area Networks (CAN). This example uses the CAN Configuration, CAN Pack, CAN Transmit, CAN Receive and CAN Unpack blocks to perform data transfer over a CAN bus.

MathWorks virtual CAN channels were used for this example. You can however connect your models to other supported hardware.

Model Description

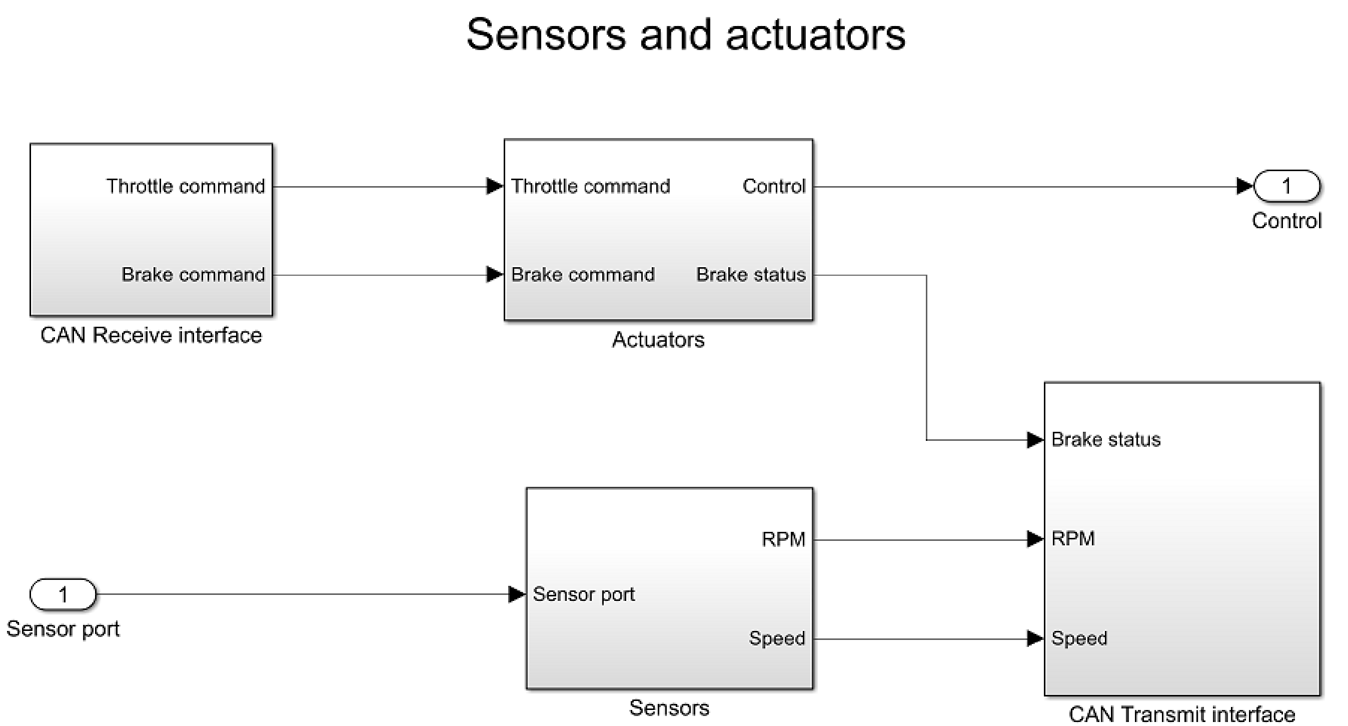


The model consists of the following subsystems: Vehicle dynamics model, Sensors and actuators, Turn signal, Dashboard, brake light and Speed controller. The Vehicle dynamics model represents the automobile (the environment), and the other subsystems represent the various nodes on the CAN bus.

Vehicle Dynamics Model

This subsystem defines the equations of motion of the automobile. The inputs are the throttle and brake actuator positions. The outputs are the engine RPM and vehicle speed, that are multiplexed into a single signal.

Sensors and Actuators



This subsystem consists of the throttle and brake actuators, and the RPM and vehicle speed sensors. The actuators receive the throttle and the brake commands via the CAN bus. The actuator outputs (control) are fed to the vehicle dynamics model.

The brake actuator also sends a signal that informs whether or not the brakes are actuated. This signal is sampled at 100 Hz and transmitted into the CAN bus. The engine RPM and vehicle speed signals from the vehicle dynamics model that are input to this subsystem and are also sampled at 100 Hz and transmitted into the CAN bus.

Dashboard

The dashboard is the interface between the vehicle and the driver. The commanded speed can be set by the user using the slider (Speed command:Value). The turn signal can be operated using the rotary switch (Turn signal:Value).

The speed command and turn signal status signals are transmitted into the CAN bus. The sampled vehicle speed and engine RPM are read from the CAN bus and displayed on the speedometer and the tachometer respectively.

Speed Controller

The speed controller sends commands to the actuators to drive the vehicle speed to the commanded value. The vehicle speed and the commanded speed are read from the CAN bus. The throttle and brake commands are calculated by the corresponding discrete Proportional - Integral controllers. The actuator commands are transmitted into the CAN bus.

Brake Light

The Brake light subsystem receives the brake actuator status signal from the CAN bus and appropriately operates the brake lights. Whenever the brakes are actuated, the brake lights are turned on.

Turn Signal

The turn signal subsystem receives the turn signal status message from the CAN bus and appropriately activates the turn signals. The left turn signal light blinks periodically when the rotary switch is set to the "Left" position, and the right turn signal light blinks periodically when the rotary switch is set to the "Right" position.

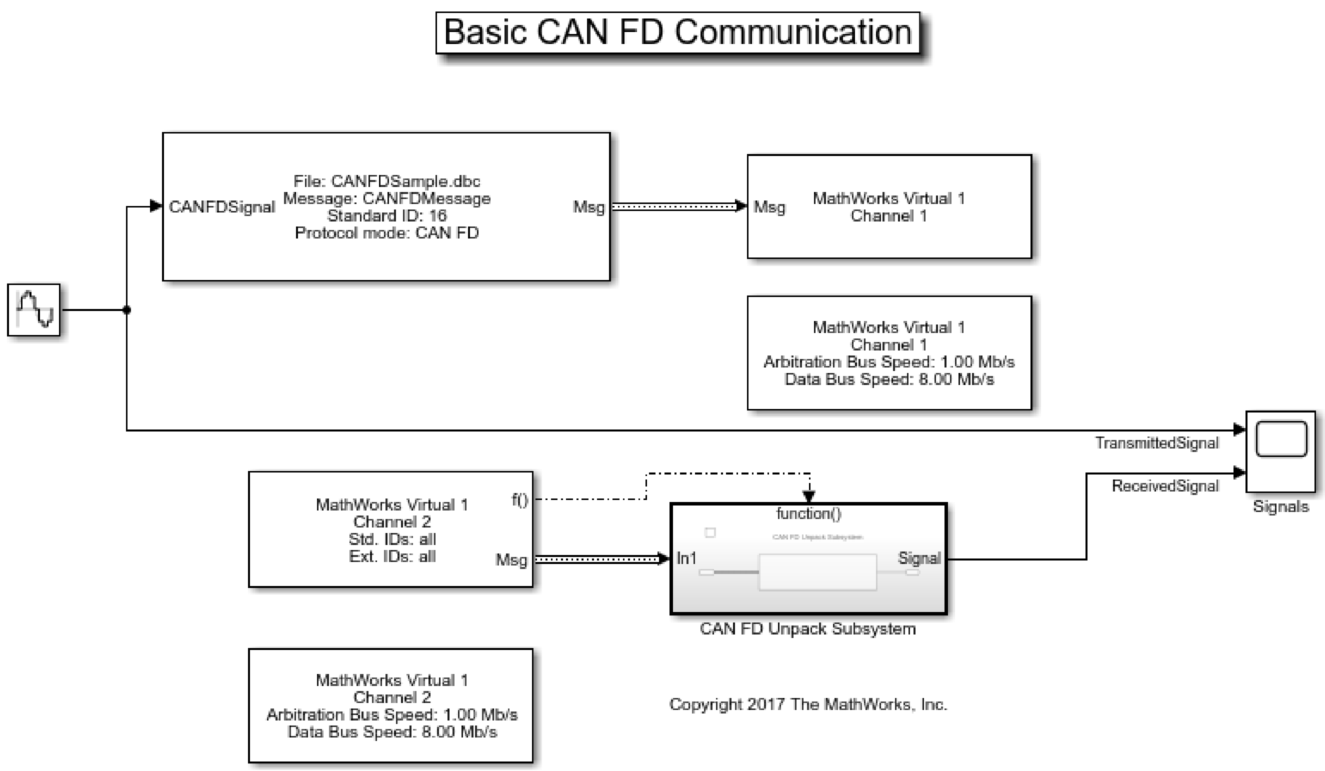
Get Started with CAN FD Communication in Simulink

This example shows how to use MathWorks virtual CAN FD channels to set up transmission and reception of CAN FD messages in Simulink. The virtual channels are connected in a loopback configuration.

Vehicle Network Toolbox provides Simulink blocks for transmitting and receiving live messages via Simulink models over networks utilizing the Controller Area Network Flexible Data (CAN FD) format. This example uses the CAN FD Configuration, CAN FD Pack, CAN FD Transmit, CAN FD Receive and CAN FD Unpack blocks to perform data transfer over a CAN FD bus. These blocks operate similarly to the CAN blocks, but are intended for use only on networks or devices that support the CAN FD protocol.

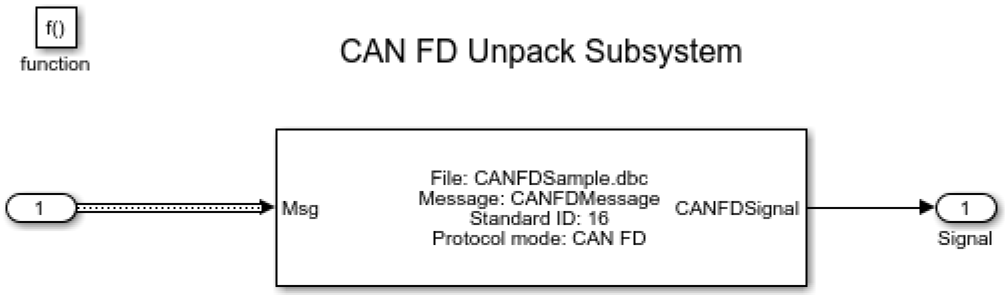
Transmit and Receive CAN FD Messages

Create a model to transmit and receive a CAN FD message carrying a sine wave data signal. The model transmits a single message per timestep. A DBC-file defines the message and signal used in the model.



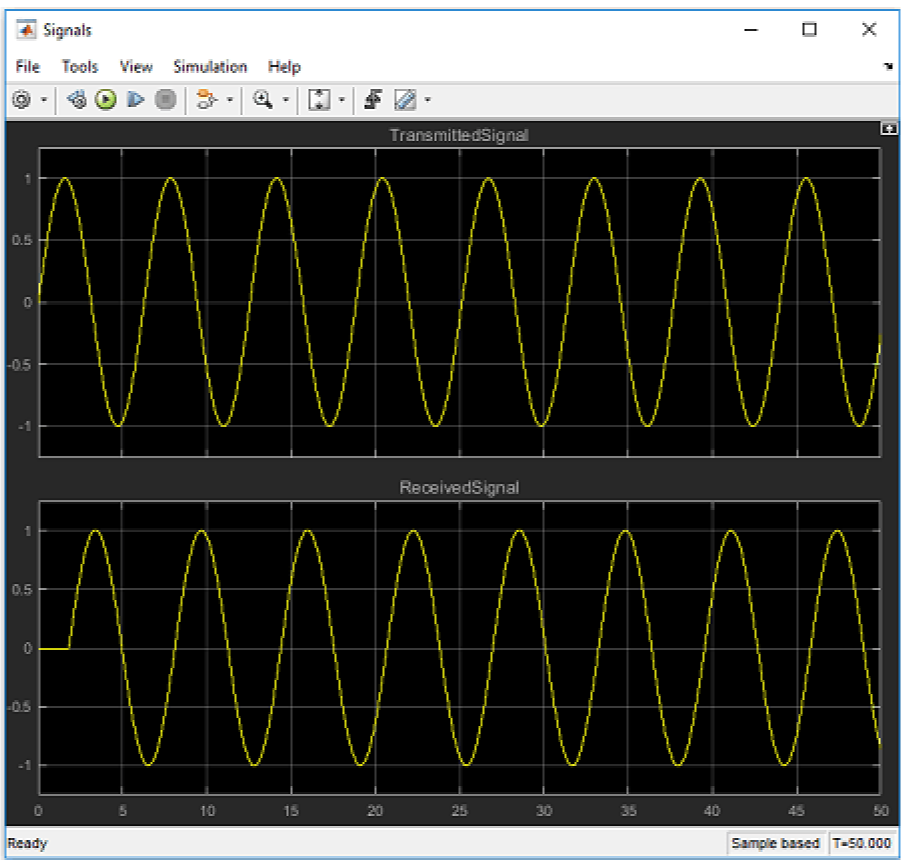
Process CAN FD Messages

The CAN FD Receive block generates a function-call trigger if it receives a new message at any particular timestep. This indicates to other blocks in the model that a message is available for decoding activities. Signal decoding and processing is performed inside the Function-Call Subsystem (Simulink).



Visualize Signal Data

Plot the sine wave values before and after transmission. The X-axis corresponds to the simulation timestep and the Y-axis corresponds to the value of the signal. Note that the phase shift between the two plots represents the propagation delay as the signal travels across the network.



Extend the Example

This example uses MathWorks virtual CAN FD channels. You can connect your models to other supported hardware. You can also modify the model to transmit at periodic rates.

Forward Collision Warning Application with CAN FD and TCP/IP

This example shows how to execute a forward collision warning (FCW) application with sensor and vision data replayed live via CAN FD and TCP/IP protocols. Recorded data from a sensor suite mounted on a test vehicle are replayed live as if they were coming through the network interfaces of the vehicle. Vehicle Network Toolbox™ and Instrument Control Toolbox™ provide these interfaces. This setup is used to test an FCW system developed using features from Automated Driving Toolbox™. For assistance with the design and development of actual FCW algorithms, refer to the example “Forward Collision Warning Using Sensor Fusion” (Automated Driving Toolbox).

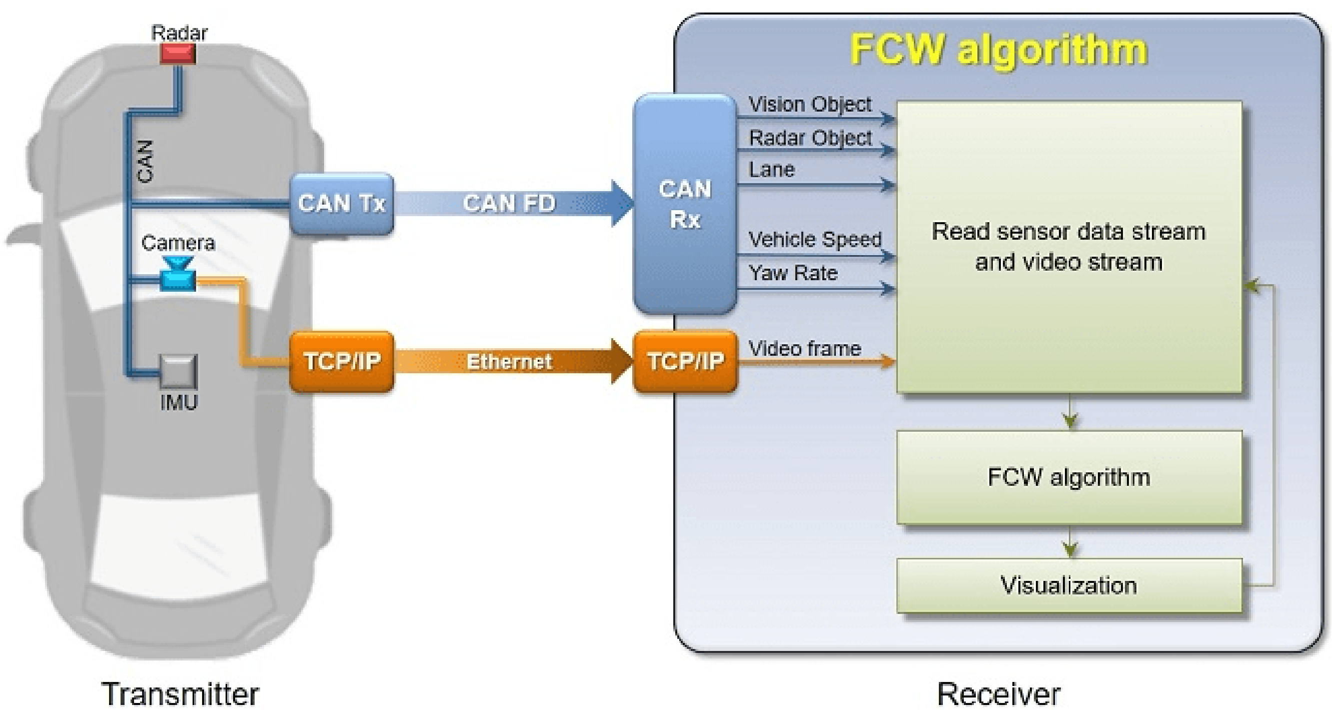
System Configuration

This example uses virtual CAN FD channels from Vector. These virtual device channels are available with the installation of the Vector Driver Setup package from www.vector.com.

This example has two primary components:

1. Transmitter: Sends the sensor and vision data via CAN FD and TCP/IP. This portion represents a sample vehicle environment. It replays prerecorded data as if it were a live vehicle.
2. Receiver: Collects all the data and executes the FCW algorithm and visualizations. This portion represents the application component.

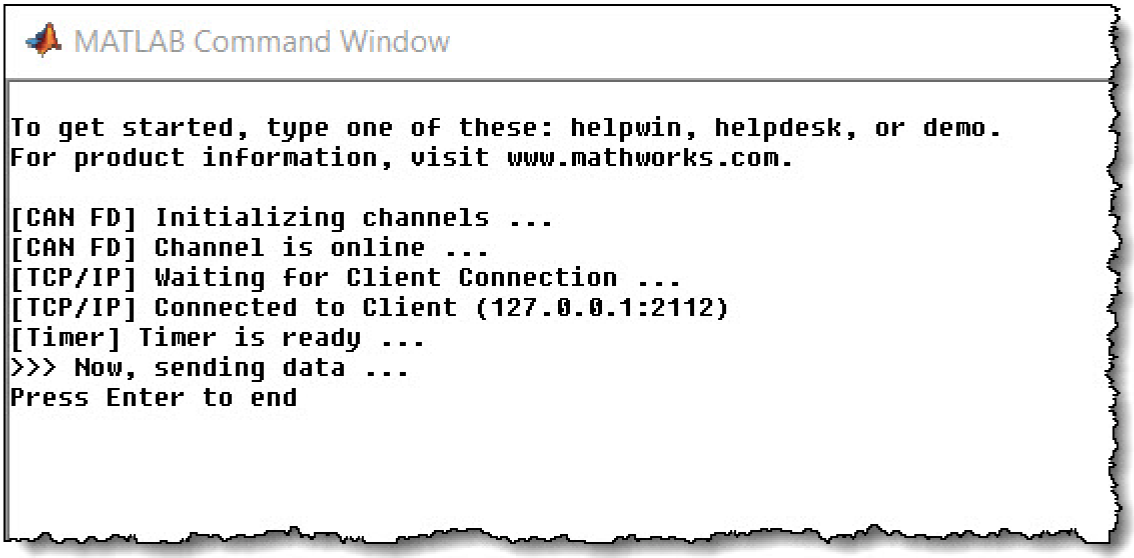
To execute the example, the transmitter and receiver portions run from separate sessions of MATLAB®. This replicates the data source existing outside the MATLAB session serving as the development tool. Furthermore, this example allows you to run the FCW application in multiple execution modes (interpreted and MEX) with different performance characteristics.



Generate Data

The transmitting application executes via the `helperStartTransmitter` function. It launches a separate MATLAB process to run outside of the current MATLAB session. The transmitter initializes itself and begins sending sensor and vision data automatically. To run the transmitter, use the `system` command.

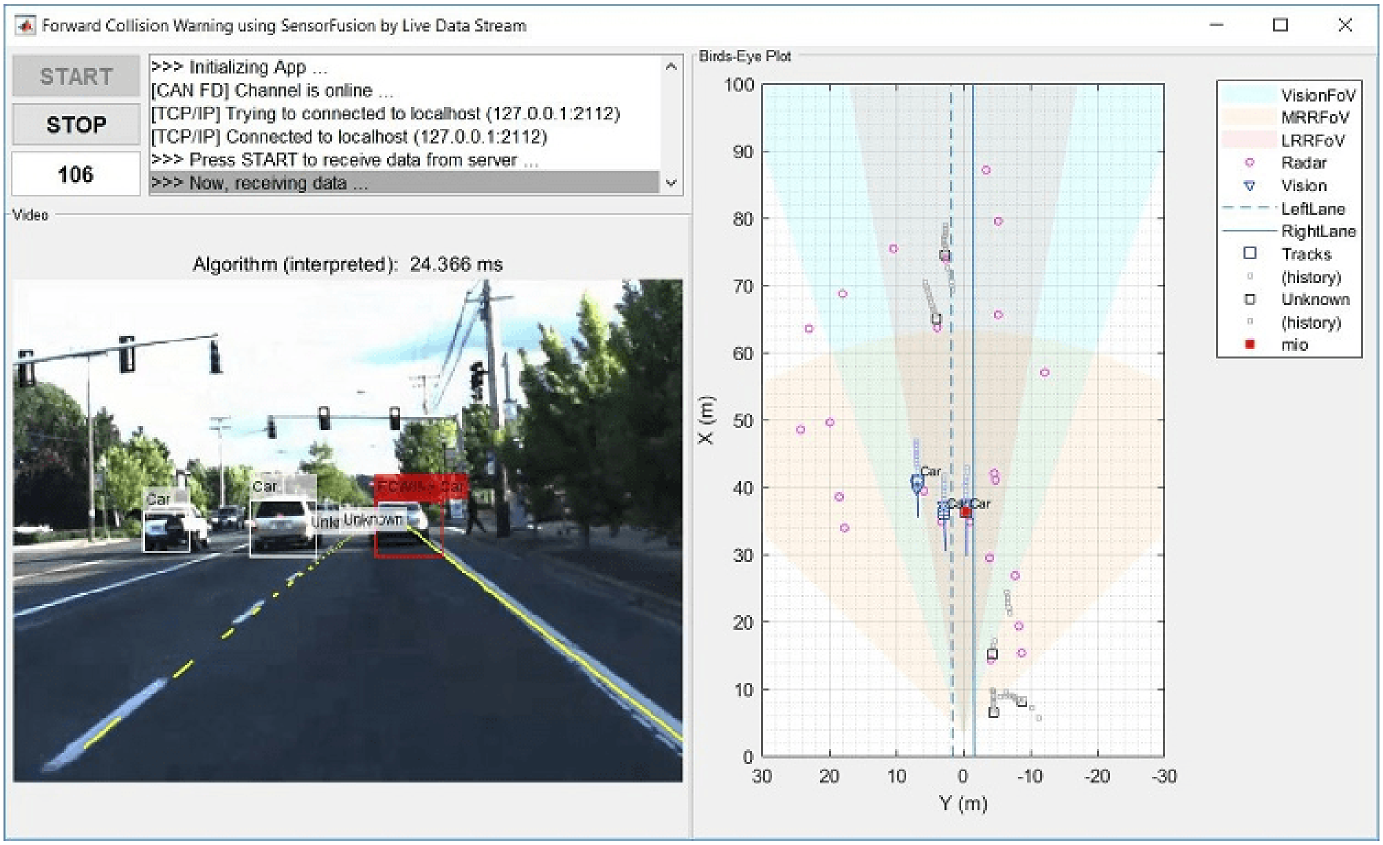
```
system('matlab -nodesktop -nosplash -r helperStartTransmitter &')
```



Execute Forward Collision Warning System (Interpreted Mode)

To open the receiving FCW application, execute the `helperStartReceiver` function. You can click **START** to begin data reception, processing, and visualization. You can explore the `helperStartReceiver` function to see how the Vehicle Network Toolbox CAN FD functions, Instrument Control Toolbox TCP/IP functions, and Automated Driving Toolbox capabilities are used in concert with one another.

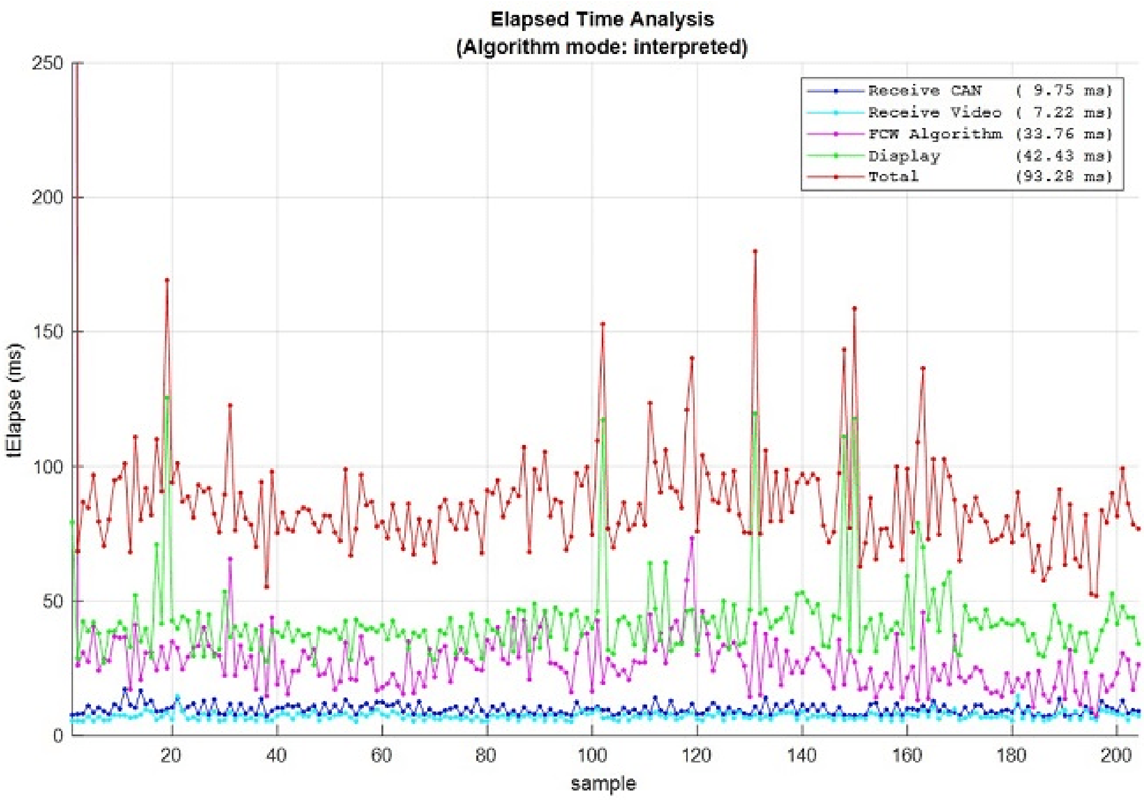
```
helperStartReceiver('interpreted')
```



Review Results

When ready, stop the transmitter application using the close window button on its command window. Click **STOP** on the receiving FCW application, and then close its window as well.

When the receiving FCW application is stopped, a plot appears detailing performance characteristics of the application. It shows time spent receiving data, processing the FCW algorithm, and performing visualizations. Benchmarking is useful to show parts of the setup that need performance improvement. It is clear that a significant portion of time is spent executing the FCW algorithm. In the next section, explore code generation as a strategy to improve performance.



Execute Forward Collision Warning System (MEX Mode)

If faster performance is a requirement in your workflow, you can use MATLAB Coder™ to generate and compile MATLAB code as MEX code. To build this example as MEX code, use the `helperGenerateCode` function. The build will compile the FCW application into a MEX function directly callable within MATLAB.

```
helperGenerateCode('mex')
```

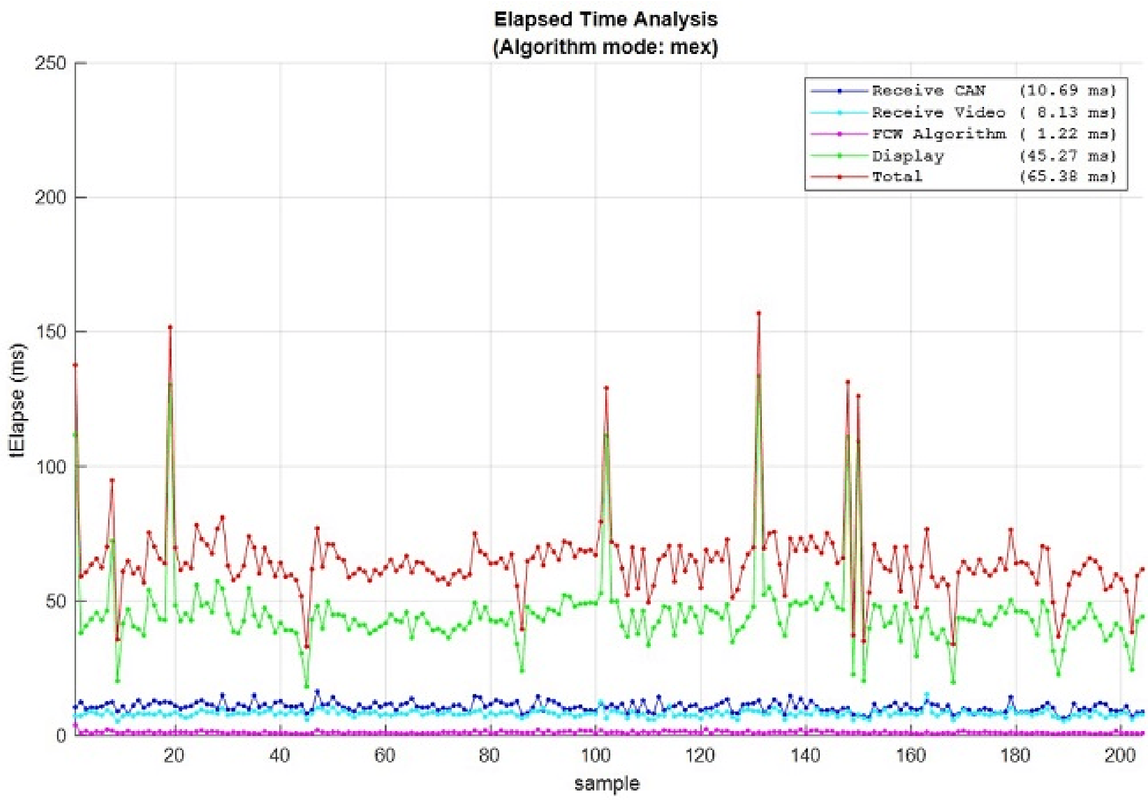
Restart the transmitter application.

```
system('matlab -nodesktop -nosplash -r helperStartTransmitter &')
```

The receiving FCW application can also be restarted. This time with an input argument to use the MEX compiled code built in the prior step.

```
helperStartReceiver('mex')
```

When ready, stop and close the transmitter and receiving FCW application. Comparing the time plot for MEX execution to the interpreted mode plot, you can see the performance improvement for the FCW algorithm.



Use Physical Hardware and Multiple Computers

The example uses a single computer to simulate the entire system with virtual connectivity. As such, its performance is meant as an approximation. You can also execute this example using two computers (one as transmitter, one as receiver). This would represent more of a real live data scenario. To achieve this, you can make simple modifications to the example code.

Changing the CAN FD communication from virtual to physical devices requires editing the transmission and reception code to invoke `canChannel` using a hardware device instead of the virtual channels. You may also need to modify the call to `configBusSpeed` depending on the capabilities of the hardware. These calls are found in the `helperStartReceiver` and `dataTransmitter` functions of the example.

Changing TCP/IP communication for multiple computers requires adjusting the TCP/IP address of the transmitter from local host (127.0.0.1) to a static value (192.168.1.2 recommended). This address is set first on the host transmitting computer. After, modify the `tcpipAddr` variable in the `helperStartReceiver` function to match.

Once configured and connected physically, you can run the transmitter application on one computer and the FCW application on the other.

Data Analytics Application with Many MDF-Files

This example shows you how to investigate vehicle battery power during discharge mode across various drive cycles. The data for this analysis are contained in a set of vehicle log files in MDF format. For this example, we need to build up a mechanism that can "detect" when the vehicle battery is in a given mode. What we are really doing is building a detector to determine when a signal of interest (battery power in this case) meets specific criteria. When the criteria is met, we will call that an "event". Each event will be subsequently "qualified" by imposing time bounds. That is to say an event is "qualified" if it persists for at least 5 seconds (such a qualification step can help limit noise and remove transients). The thresholds shown in this example are illustrative only.

Set Data Source Location

Define the location of the file set to analyze.

```
dataDir = '*.dat';
```

Obtain File Set Information

Get the names of all the MDF-files to analyze into a single cell array.

```
fileList = dir(dataDir);
fileName = {fileList(:).name}';
fileDir  = {fileList(:).folder}';
fullFilePath = fullfile(fileDir, fileName)
```

```
fullFilePath = 5x1 cell
    {'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex86857001\ADAC.dat' }
    {'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex86857001\ECE.dat'  }
    {'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex86857001\HWFET.dat'}
    {'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex86857001\SC03.dat' }
    {'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex86857001\US06.dat' }
```

Pre-allocate the Output Data Cell Array

Use a cell array to capture a collection of mini-tables which represent the event data of interest for each individual MDF-file.

```
numFiles = size(fullFilePath, 1);
eventSet = cell(numFiles, 1)
```

```
eventSet=5×1 cell array
    {0x0 double}
    {0x0 double}
    {0x0 double}
    {0x0 double}
    {0x0 double}
```

Define Event Detection and Channel Information Criteria

```
chName = 'Power';          % Name of the signal of interest in the MDF-files
thdValue = [5, 55];        % Threshold in KW
thdDuration = seconds(5); % Threshold for event qualification
```

Loop Through Each MDF-File and Apply the Event Detector Function

`eventSet` is a cell array which contains a summary table for each file that was analyzed. You can think of this cell array of tables as a set of mini-tables, all with the same format but the contents of each mini-table correspond to the individual MDF-files.

In this example, the event detector not only reports the event start and end times but also some descriptive statistics about the event itself. This kind of aggregation and reporting can be useful for discovery and troubleshooting activities. To understand the MDF-file interfacing and data handling in more detail, open and explore the `processMDF` function from this example.

Note that the data processing is written such that each MDF-file is parsed atomically and returns into its own index of the resulting cell array. This allows the processing function to leverage parallel computing capability with `parfor`. `parfor` and standard `for` are interchangeable in terms of outputs, but result in varying processing time needed to complete the analysis. To experiment with parallel computing, simply change the `for` call below to `parfor` and run this example.

```
for i = 1:numFiles
    eventSet{i} = processMDF(fullFilePath{i}, chName, thdValue, thdDuration);
end
eventSet{1}
```

```
ans=20×8 table
    FileName    EventNumber    EventDuration    EventStart    EventStop    MeanPower_KW    MaxPo
    ________    ___________    _____________    __________    _________    ____________    _____

    ADAC.dat         2           00:01:22       19.345 sec    101.79 sec      28.456           5
    ADAC.dat         3           00:00:08       107.82 sec    116.36 sec      21.295           5
    ADAC.dat         5           00:00:55        123.8 sec    179.67 sec      28.642           3
    ADAC.dat         6           00:00:10       189.83 sec    200.36 sec      11.192           5
    ADAC.dat         8           00:00:40        212.4 sec    252.79 sec      28.539           3
    ADAC.dat         9           00:00:08       258.76 sec    267.37 sec      21.289           5
    ADAC.dat        11           00:00:44       274.81 sec    319.79 sec      28.554           3
    ADAC.dat        12           00:00:08       325.75 sec    334.37 sec      21.279           5
    ADAC.dat        14           00:00:44       341.81 sec    386.79 sec      28.554           3
    ADAC.dat        15           00:00:08       392.75 sec    401.37 sec      21.278           5
    ADAC.dat        17           00:00:44       408.81 sec    453.67 sec      28.579           3
    ADAC.dat        18           00:00:07       463.77 sec    471.37 sec      11.895          54
    ADAC.dat        20           00:00:40       483.44 sec    523.79 sec      28.544          37
    ADAC.dat        21           00:00:08       529.75 sec    538.37 sec      21.279           5
    ADAC.dat        23           00:00:44       545.81 sec    590.79 sec      28.553           3
    ADAC.dat        24           00:00:08       596.75 sec    605.37 sec      21.279           5
      ⋮
```

Concatenate Results

Combine the contents of the cell array `eventSet` into a single table. We can now use the table `eventSummary` for subsequent analysis. The `head` function is used to display the first 5 rows of the table `eventSummary`.

```
eventSummary = vertcat(eventSet{:});
disp(head(eventSummary, 5))
```

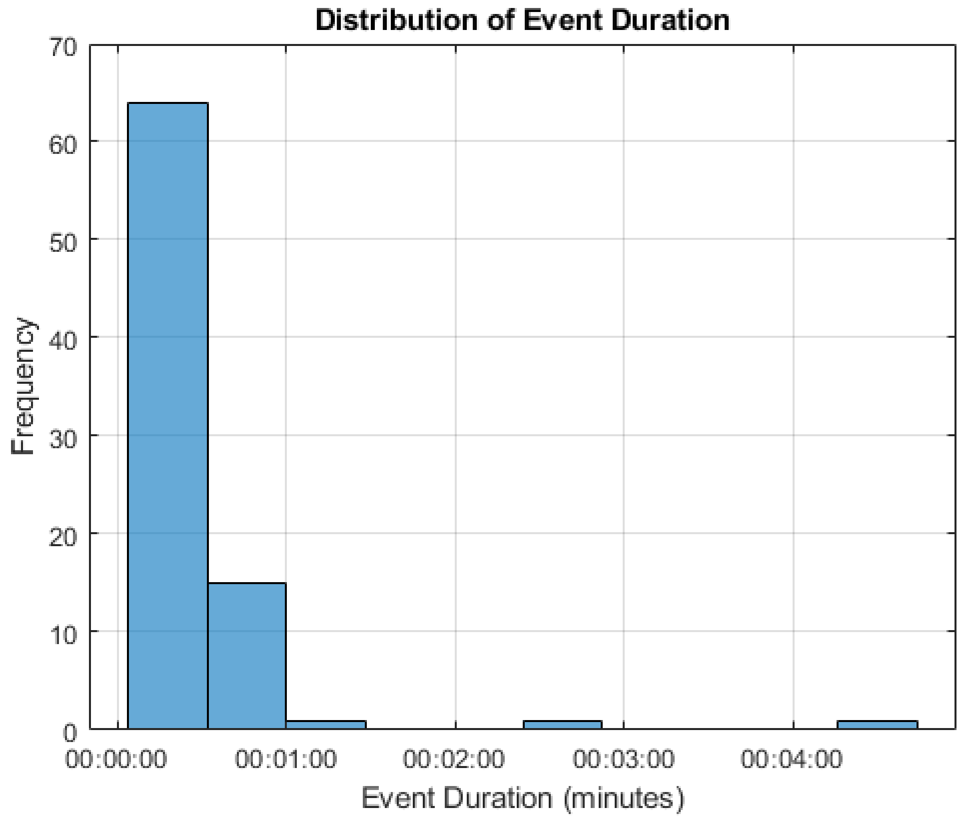
```
    FileName    EventNumber    EventDuration    EventStart    EventStop    MeanPower_KW    MaxPo
    ________    ___________    _____________    __________    _________    ____________    _____
```

```
    ADAC.dat        2          00:01:22       19.345 sec    101.79 sec       28.456      53
    ADAC.dat        3          00:00:08       107.82 sec    116.36 sec       21.295      53
    ADAC.dat        5          00:00:55        123.8 sec    179.67 sec       28.642      37
    ADAC.dat        6          00:00:10       189.83 sec    200.36 sec       11.192      54
    ADAC.dat        8          00:00:40        212.4 sec    252.79 sec       28.539      37
```

Visualize Summary Results to Determine Next Steps

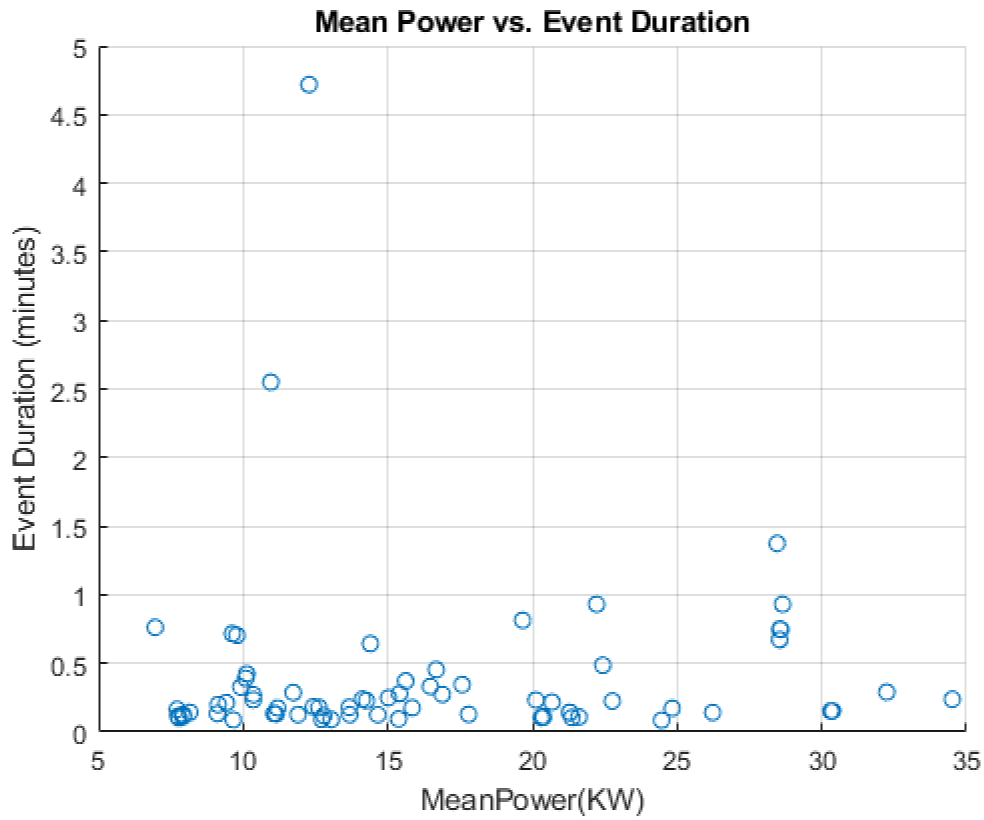
Look at an overview of the event durations.

```
histogram(eventSummary.EventDuration)
grid on
title 'Distribution of Event Duration'
xlabel 'Event Duration (minutes)'
ylabel 'Frequency'
```



Now look at Mean Power vs. Event Duration.

```
scatter(eventSummary.MeanPower_KW, minutes(eventSummary.EventDuration))
grid on
xlabel 'MeanPower(KW)'
ylabel 'Event Duration (minutes)'
title 'Mean Power vs. Event Duration'
```



Deep Dive an Event of Interest

Inspect the event that lasted for more than 4 minutes. First, create a mask to find the case of interest. `msk` is a logical index that shows which rows of the table `eventSummary` meet the specified criteria.

```
msk = eventSummary.EventDuration > minutes(4);
```

Pull out the rows of the table `eventSummary` that meet the criteria specified and display the results.

```
eventOfInterest = eventSummary(msk, :);
disp(eventOfInterest)
```

```
    FileName      EventNumber    EventDuration    EventStart    EventStop     MeanPower_KW    MaxP
    _________     ___________    _____________    __________    __________    ____________    ____

    HWFET.dat         18           00:04:43       297.22 sec    580.37 sec       12.275          3
```

Visualize This Event in the Context of the Entire Drive Cycle

We need the full file path and file name to read the data from the MDF-file. The table `eventOfInterest` has the filename because we kept track of that. It does not have the full file path to that file. To get this information we will apply a bit of set theory to our original list of filenames and paths. First, find the full file path of the file of interest.

```
fileMsk = find(ismember(fileName, eventOfInterest.FileName))
```

```
fileMsk = 3
```

Create an MDF object to read data from the MDF-file.

```
mdfObj = mdf(fullFilePath{fileMsk})
```

```
mdfObj = 
  MDF with properties:

   File Details
                 Name: 'HWFET.dat'
                 Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex86857001\HWFET
               Author: ''
           Department: ''
              Project: ''
              Subject: ''
              Comment: ''
              Version: '3.00'
             DataSize: 3167040
     InitialTimestamp: 2017-08-09 12:20:03.000000000

   Creator Details
    ProgramIdentifier: 'MDA v7.1'
              Creator: [1x1 struct]

   File Contents
           Attachment: [0x1 struct]
         ChannelNames: {{5x1 cell}}
         ChannelGroup: [1x1 struct]

   Options
           Conversion: Numeric
```

Identify the channel with `channelList` and read all the data from this file.

```
chInfo = channelList(mdfObj, chName)
```

chInfo=*1×9 table*

ChannelName	ChannelGroupNumber	ChannelGroupNumSamples	ChannelGroupAcquisitionName
"Power"	1	79176	<undefined>

```
data = read(mdfObj, chInfo)
```

```
data = 1x1 cell array
    {79176x1 timetable}
```

Note that reading with the output of `channelList` returns a cell array of results.

```
data{1}(1:10,:)
```

ans=*10×1 timetable*

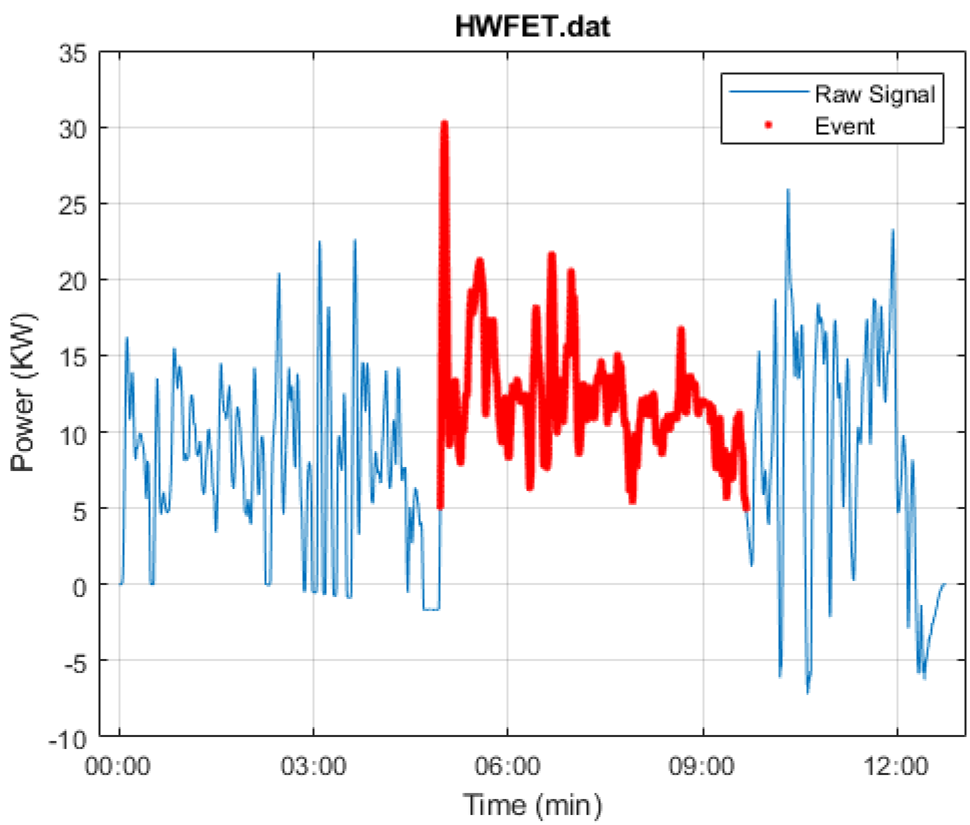
Time	Power
0.0048987 sec	0
0.0088729 sec	0

```
0.01 sec                   0
0.013223 sec               0
0.016446 sec               0
0.019668 sec               0
0.02 sec                   0
0.021658 sec        -2.4e-28
0.023878 sec       -3.42e-15
0.026098 sec       -1.04e-14
```

Visualize Using a Custom Plotting Function

Custom plotting functions are useful for encapsulation and reuse. Visualize the event in the context of the entire drive cycle. To understand how the visualization was created, open and explore the `eventPlotter` function from this example.

```
eventPlotter(data{1}, eventOfInterest)
```



Close the File

Close access to the MDF-file by clearing its variable from the workspace.

```
clear mdfObj
```

Log and Replay CAN FD Messages

This example shows you how to log and replay CAN FD messages using MathWorks virtual CAN FD channels in Simulink. You can update this model to connect to supported hardware on your system.

Load the saved CAN FD message from `sourceFDMsgs.mat` file from the examples folder. The file contains CAN FD messages representing a 90 second drive cycle around a test track.

Convert these messages to a format compatible with the CAN FD Replay block and save it to a separate file.

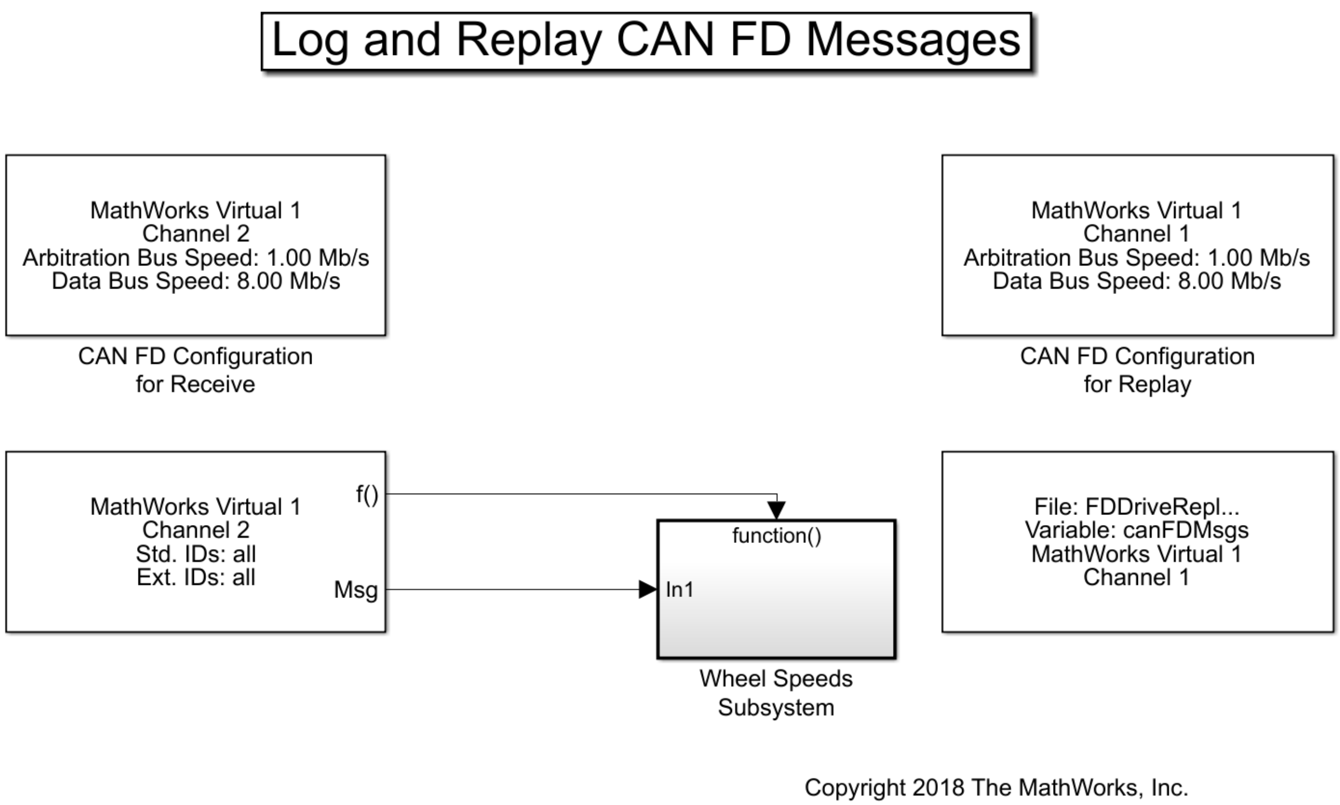
```
  Name                      Size                Bytes  Class        Attributes

  canFDMsgTimetable      100000x12           45411725  timetable
  canFDMsgs                   1x1             8401848  struct
```

CAN FD Replay Model

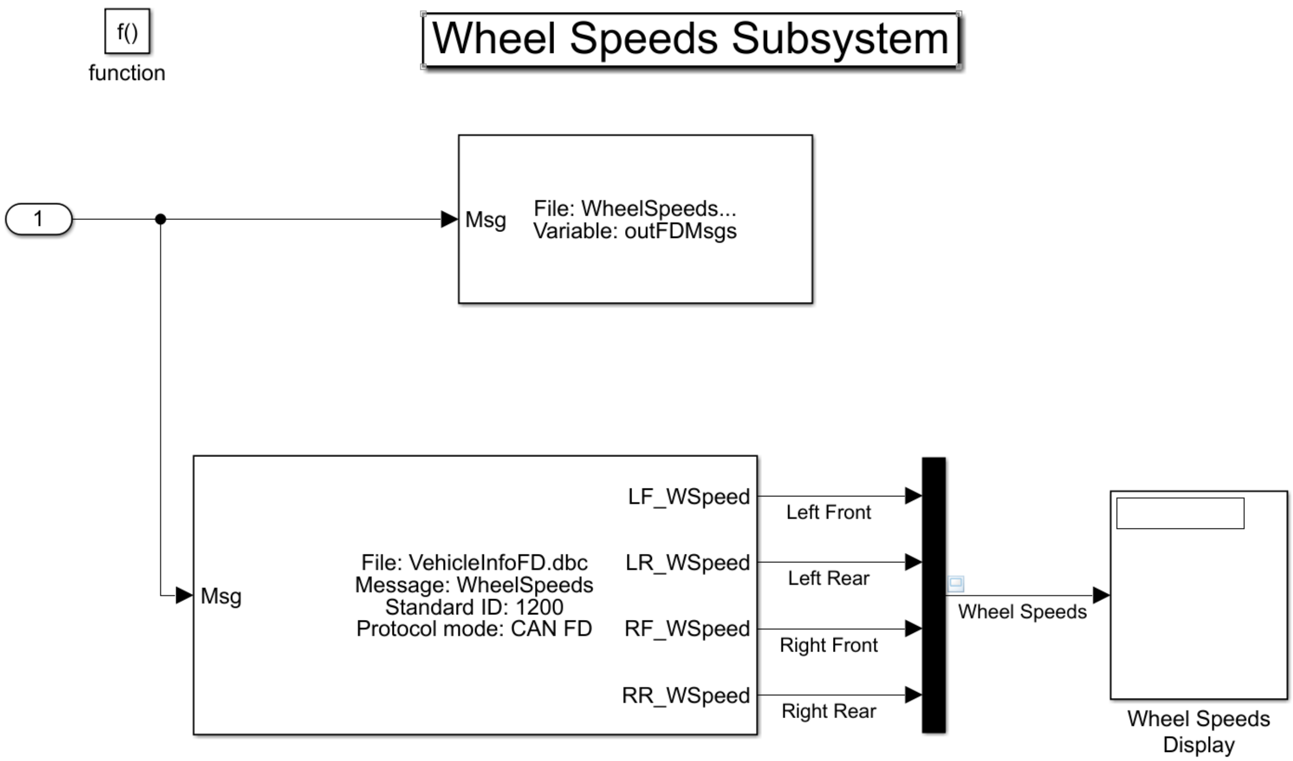
This model contains:

- A CAN FD Replay block that transmits to `MathWorks Virtual Channel 1`.
- A CAN FD Receive block that receives the messages on a CAN FD network, through `MathWorks Virtual Channel 2`.

The CAN FD Receive block is configured to block all extended IDs and allow only the `WheelSpeed` message with the standard ID `1200` to pass.

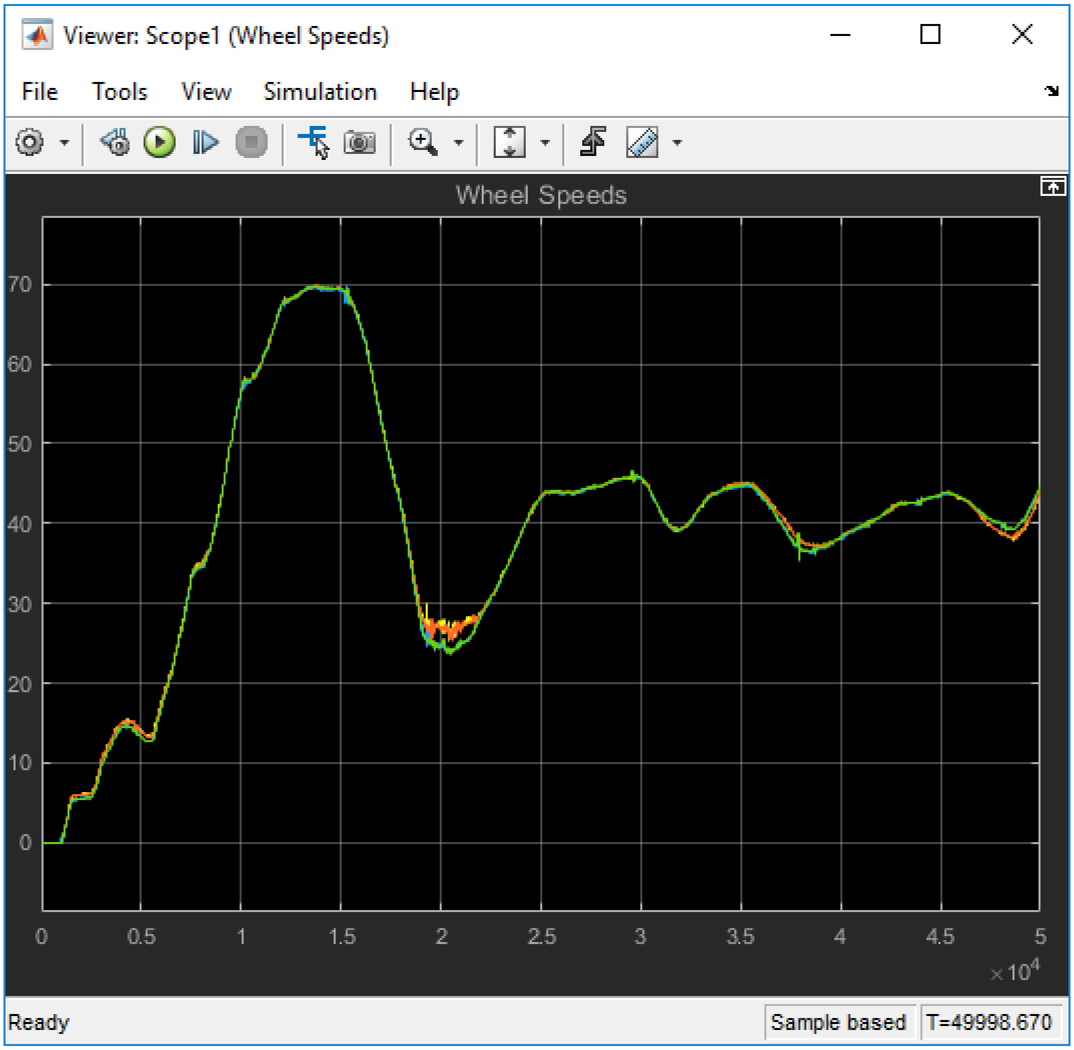


The Wheel Speeds subsystem unpacks the wheel speed information from the received CAN FD messages and plots them to a scope. The subsystem also logs the messages to a file.



Visualize Wheel Speed Information

The plot shows the wheel speed for all wheels for the duration of the test drive.



Load the Logged Message File

The CAN FD Log block creates a unique file each time you run the model. Use `dir` in the MATLAB Command Window to find the latest log file.

```
WheelSpeeds_2018-Apr-30_132033.mat

  Name                       Size                Bytes  Class        Attributes

  canFDMsgTimetable       100000x12           45411725  timetable
  canFDMsgs                    1x1             8401848  struct
  outFDMsgs                    1x1              841848  struct
```

Convert Logged Messages

Use `canFDMessageTimetable` to convert messages logged during the simulation to a timetable that you can use in the command window.

To access message signals directly, use the appropriate database file in the conversion along with `canSignalTimetable`.

```
ans =

  15x12 timetable

       Time        ID     Extended          Name          ProtocolMode                Data
    __________    ____    ________    _______________    ____________    ________________________

    75.393 sec     576     false      {0x0 char     }     {'CAN FD'}     {[                   79 13
    75.397 sec    1200     false      {'WheelSpeeds'}     {'CAN FD'}     {[     54 171 55 39 54 1
    75.398 sec     128     false      {0x0 char     }     {'CAN FD'}     {[         41 89 117 48
    75.398 sec     133     false      {0x0 char     }     {'CAN FD'}     {[             0 102 0 0
    75.398 sec     144     false      {0x0 char     }     {'CAN FD'}     {[   167 129 247 8 200 1
    75.398 sec     528     false      {0x0 char     }     {'CAN FD'}     {[           255 254 60
    75.399 sec     529     false      {0x0 char     }     {'CAN FD'}     {[255 255 255 255 255 25
    75.399 sec    1201     false      {0x0 char     }     {'CAN FD'}     {[     15 155 16 23 15 1
    75.399 sec     512     false      {0x0 char     }     {'CAN FD'}     {[     2 125 1 213 2 129
    75.399 sec     513     false      {0x0 char     }     {'CAN FD'}     {[  31 179 255 255 54 22
    75.399 sec     533     false      {0x0 char     }     {'CAN FD'}     {[        2 168 2 168 2
    75.4 sec      1312     false      {0x0 char     }     {'CAN FD'}     {[               250 0
    75.405 sec    1200     false      {'WheelSpeeds'}     {'CAN FD'}     {[     54 173 55 41 54 1
    75.406 sec    1201     false      {0x0 char     }     {'CAN FD'}     {[     15 157 16 25 15 1
    75.408 sec    1296     false      {0x0 char     }     {'CAN FD'}     {[


ans =

  15x4 timetable

       Time        RR_WSpeed    RF_WSpeed    LR_WSpeed    LF_WSpeed
    __________    _________    _________    _________    _________

    75.397 sec      41.19        40.04        41.19        39.95
    75.405 sec       41.2        40.04        41.21        39.97
    75.414 sec      41.22        40.05        41.26        40.03
    75.424 sec      41.25        40.13         41.3        40.05
    75.433 sec      41.19        40.14        41.28        40.08
    75.441 sec      41.17        40.18        41.31        40.14
    75.45 sec       41.31        40.27        41.31        40.17
    75.458 sec      41.37        40.25        41.31        40.19
    75.466 sec      41.39        40.22         41.3        40.19
    75.475 sec      41.39        40.25         41.3         40.2
    75.483 sec      41.37        40.26        41.33        40.21
    75.492 sec      41.44        40.35        41.33        40.19
    75.501 sec      41.51        40.44        41.36        40.22
    75.509 sec      41.58        40.47        41.44        40.29
    75.517 sec      41.63        40.45        41.44        40.31
```

MathWorks CAN FD virtual channels were used for this example. You can however connect your models to other supported hardware.

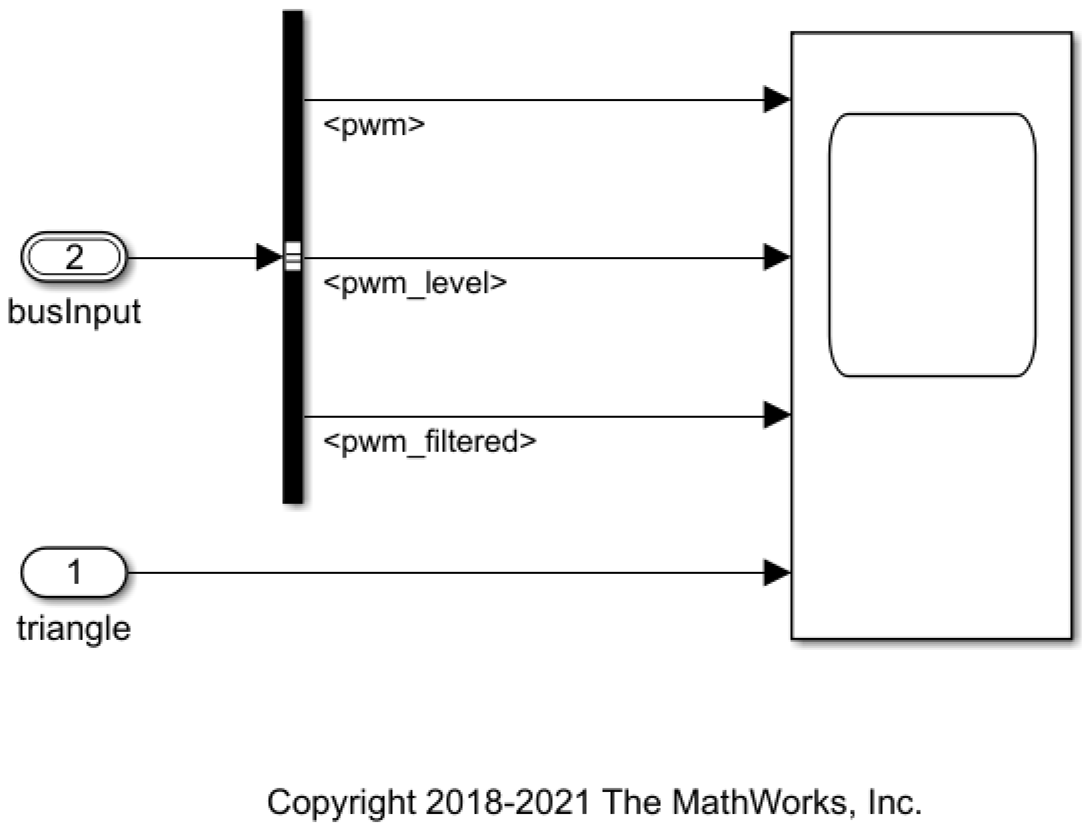
Map Channels from MDF-Files to Simulink Model Input Ports

This example shows you how to programmatically map channels from MDF-files and consume their data via input ports of a Simulink model. It performs the gathering of input port names of a Simulink model and correlates them to the content of a given MDF-file. A linkage between them is then created which consumes channel data sourced from the MDF-file when the model runs.

Acquire Model Details

Define the example model name and open it.

```
mdlName = "ModelForMDFInput";
open_system(mdlName);
```



Use the `createInputDataset` function to obtain overall information about the model and its inputs.

```
dsObj = createInputDataset(mdlName)
```

```
dsObj = 
Simulink.SimulationData.Dataset '' with 2 elements

                         Name      BlockPath
                         ________  _________

    1  [1x1 timeseries]  triangle  ''
    2  [1x1 struct    ]  busInput  ''
```

```
  - Use braces { } to access, modify, or add elements using index.
```

Obtain Model Input Port Names

This model has both a bus and an individual input port. The `helperGetMdlInputNames` function demonstrates how to get the name of all the model inputs regardless of how they are defined in the model.

```
mdlInputNames = helperGetMdlInputNames(mdlName)
```

```
mdlInputNames = 4x1 string
    "triangle"
    "pwm"
    "pwm_level"
    "pwm_filtered"
```

Investigate the MDF-File

Now that you have the input port names of the model, you can see what channels exist in the MDF-file so you can attempt to match them. The `channelList` function allows quick access to the available channels present in an MDF-file.

```
mdfName = "CANape.MF4";
mdfObj = mdf(mdfName);
mdfChannelInfo = channelList(mdfObj)
```

```
mdfChannelInfo=120×9 table
           ChannelName            ChannelGroupNumber    ChannelGroupNumSamples    ChannelGroupA
    ___________________________    __________________    ______________________    _____________

    "ampl"                                 2                      199                     10
    "channel1"                             2                      199                     10
    "Counter_B4"                           1                     1993                     10
    "Counter_B5"                           1                     1993                     10
    "Counter_B6"                           1                     1993                     10
    "Counter_B7"                           1                     1993                     10
    "map1_8_8_uc_measure"                  1                     1993                     10
    "map1_8_8_uc_measure[0][0]"            1                     1993                     10
    "map1_8_8_uc_measure[0][1]"            1                     1993                     10
    "map1_8_8_uc_measure[0][2]"            1                     1993                     10
    "map1_8_8_uc_measure[0][3]"            1                     1993                     10
    "map1_8_8_uc_measure[0][4]"            1                     1993                     10
    "map1_8_8_uc_measure[0][5]"            1                     1993                     10
    "map1_8_8_uc_measure[0][6]"            1                     1993                     10
    "map1_8_8_uc_measure[0][7]"            1                     1993                     10
    "map1_8_8_uc_measure[1][0]"            1                     1993                     10
      ⋮
```

Construct a Table to Manage Items of Interest

Use a table to map the model input ports to MDF channels.

```
channelTable = table();
channelTable.PortNames = mdlInputNames;
n = size(channelTable.PortNames,1);
```

```
channelTable.ChGrpNum = NaN(n,1);
channelTable.ChNameActual = strings(n,1);
channelTable
```

```
channelTable=4×3 table
      PortNames        ChGrpNum    ChNameActual
    _______________    ________    ____________

    "triangle"           NaN            ""
    "pwm"                NaN            ""
    "pwm_level"          NaN            ""
    "pwm_filtered"       NaN            ""
```

Perform Input Port to Channel Matching

The `helperReportChannelInfo` function searches the MDF-file for channel names that match the model input port names. When found, the details of the channel are recorded in the table. Specifically, the channel group number where the given channel is in the file and its actual defined name. Note that the actual channel names are not exact matches to the model port names. In this example, the channel name matching is performed case-insensitive and ignores the underscore characters. This algorithm can be adapted as needed based on application-specific matching criteria.

```
channelTable = helperReportChannelInfo(channelTable, mdfChannelInfo)
```

```
channelTable=4×3 table
      PortNames        ChGrpNum    ChNameActual
    _______________    ________    _____________

    "triangle"            1        "Triangle"
    "pwm"                 1        "PWM"
    "pwm_level"           1        "PWM_Level"
    "pwm_filtered"        1        "PWMFiltered"
```

Populate the Simulink Dataset Object with Channel Data

The dataset object created earlier contains both a single timeseries object and a structure of timeseries objects. This makes assigning data back to them somewhat challenging. Things to keep in mind include:

- When specifying `TimeSeries` as the return type from the MDF `read` function, you must call read separately for each channel.
- Because the dataset object has dissimilar elements (a scalar timeseries and a scalar structure of timeseries objects), you need to manually manage the collection and make sure you are writing to the correct location.

```
for ii = 1:dsObj.numElements
    switch ii
        case {1} % [1x1 timeseries], triangle
            % Read the input port data from the MDF-file one channel at a time.
            mdfData = read(mdfObj, channelTable.ChGrpNum(ii), channelTable.ChNameActual(ii), "Ou
            % Populate the dataset object.
            dsObj{ii} = mdfData;

        case {2} % [1x1 struct], busInput
            for jj = 1:numel(fieldnames(dsObj.getElement(ii)))
```

```
                % Read the input port data from the MDF-file one channel at a time.
                mdfData = read(mdfObj, channelTable.ChGrpNum(jj+1), channelTable.ChNameActual(jj+
                % Populate the dataset object.
                dsObj{ii}.(channelTable.PortNames{jj+1}) = mdfData;
            end
    end
end
dsObj
```

```
dsObj = 
Simulink.SimulationData.Dataset '' with 2 elements

                             Name      BlockPath
                             ________  _________
    1  [1x1 timeseries]      Triangle  ''
    2  [1x1 struct    ]      busInput  ''

  - Use braces { } to access, modify, or add elements using index.
```

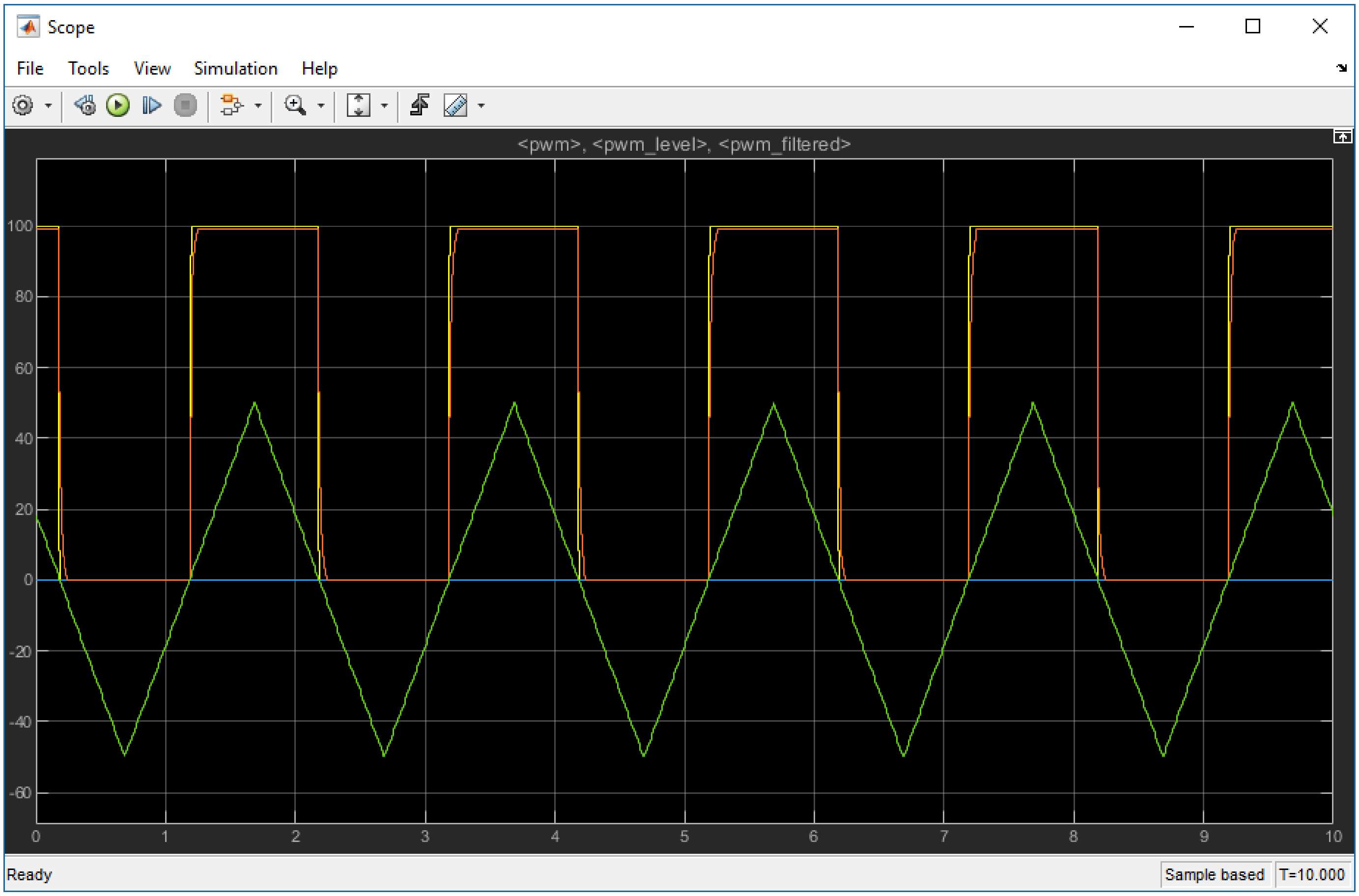
Enable the Dataset as Input to the Simulink Model

```
set_param(mdlName, "LoadExternalInput", "on");
set_param(mdlName, "ExternalInput", "dsObj");
```

Run the Model

Upon executing the model, note that channel data from the MDF-file properly maps to the designated input ports and plots through Simulink as expected.

```
open_system(mdlName);
bp = find_system(mdlName, "BlockType", "Scope");
open_system(bp);
pause(1)
sim(mdlName, "TimeOut", 10);
```



Close the File

Close access to the MDF-file by clearing its variable from the workspace.

```
clear mdfObj
```

Helper Functions

```
function mdlInputNames = helperGetMdlInputNames(mdlName)
% helperGetMdlInputNames Find input port names of a Simulink model.
%
% This function takes in the name of a Simulink model and returns the names of each model input.
% both a bus and a stand-alone input port going into it. To drive an input port that expects a bu
% the signals as timeseries objects in a struct that matches the structure of the bus object atta

% Test to see if the model is currently loaded in memory.
isLoaded = bdIsLoaded(matlab.lang.makeValidName(mdlName));

% If the model is not open then load it.
if ~isLoaded
    load_system(mdlName);
end

dsObj = createInputDataset(mdlName);
numElements = dsObj.numElements;
isStruct = zeros(1:numElements);
```

```
% Check to see if any of the elements in the returned dataset object are
% structs. If they are, assume they are for an input port that accepts a bus.
for elementIdx = 1:numElements
    isStruct(elementIdx) = isa(dsObj.getElement(elementIdx),"struct");
end

% For a port that accepts a bus, the data to be loaded must be arranged in a struct
% that matches the structure of the bus object attached to the input port.
busInportIdx = 1;
for idx = 1:numElements
    if isStruct(idx)
        % Get names of signals from a bus input port.
        inPortsBus(busInportIdx, :) = string(fieldnames(dsObj.getElement(idx)));
    else
        % Get signal name from a non-bus input port.
        inPorts(idx) = string(dsObj.getElement(idx).Name);
    end
end

mdlInputNames = [inPorts, inPortsBus]';
end


function channelTableOut = helperReportChannelInfo(channelTableIn, mdfChannelInfo)
% channelTableOut Reports if a channel is present in a set of channel names.

% Assign the output data.
channelTableOut = channelTableIn;

% Remove underscores and make everything lowercase for matching.
inPortChannelNames = lower(erase(channelTableIn.PortNames,"_"));
mdfChannelNames = lower(erase(mdfChannelInfo.ChannelName,"_"));

% Match the input channel names to the channel names in the MDF-file.
[~, inPortidx] = ismember(inPortChannelNames, mdfChannelNames);

% Assign the relevant information back to the channel table.
channelTableOut.ChGrpNum = mdfChannelInfo{(inPortidx), "ChannelGroupNumber"};
channelTableOut.ChNameActual = mdfChannelInfo{(inPortidx), "ChannelName"};
end
```

Get Started with CDFX-Files

This example shows how to import a calibration data file into MATLAB, examine and modify its contents, and export the changes back to a file on disk.

Import a CDFX-File

Import data from a CDFX-file using the `cdfx` function.

```
cdfxObj = cdfx("CDFXExampleFile.cdfx")
```

```
cdfxObj = 
  CDFX with properties:

       Name: "CDFXExampleFile.cdfx"
       Path: "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex38787800\CDFXExampleFile
    Version: "CDF20"
```

Visualize Calibration Data

CDFX-files contain information about vehicle ECUs (systems), and their parameters (instances). Use `instanceList` and `systemList` to visualize the calibration data in table form. These functions also allow filtering based on instance or system short names.

```
iList = instanceList(cdfxObj)
```

```
iList=4×6 table
             ShortName                    System          Category            Value            Units
    ____________________________    _______________    __________    _________________    ________

    "ASAM.C.SCALAR.GAIN"            "ExampleSystem"    "VALUE"       {[              3]}    "gain"
    "ASAM.C.SCALAR.BITMASK_0001"    "ExampleSystem"    "BOOLEAN"     {[              1]}    ""
    "ASAM.C.MAP"                    "ExampleSystem"    "MAP"         {1x1 struct      }    ""
    "ASAM.C.COM_AXIS"               "ExampleSystem"    "COM_AXIS"    {[-9 -8 -5 -3 0]}    "hours'
```

If you want to filter the table based on a desired short name, pass a string as a second argument.

```
iListArray = instanceList(cdfxObj, "ASAM.C.SCALAR")
```

```
iListArray=2×6 table
             ShortName                    System          Category      Value     Units     FeatureRe
    ____________________________    _______________    _________    _____    ______    _________

    "ASAM.C.SCALAR.GAIN"            "ExampleSystem"    "VALUE"      {[3]}    "gain"    "Function
    "ASAM.C.SCALAR.BITMASK_0001"    "ExampleSystem"    "BOOLEAN"    {[1]}    ""        "Function
```

The default querying behavior will return a table for all instances whose short names partially match the search string. To filter for an exact instance name match, use the `ExactMatch` name-value pair.

```
iListArrayExact = instanceList(cdfxObj, "ASAM.C.SCALAR.BITMASK_0001", "ExampleSystem", "ExactMat
```

```
iListArrayExact=1×6 table
             ShortName                    System          Category      Value     Units    FeatureRefe
    ____________________________    _______________    _________    _____    _____    __________
```

```
    "ASAM.C.SCALAR.BITMASK_0001"    "ExampleSystem"    "BOOLEAN"    {[1]}     ""      "FunctionSc
```

For CDFX-files that contain calibration data for more than one ECU system, `systemList` can be useful to view the contents of each system at a high level.

```
sList = systemList(cdfxObj)
```

```
sList=1×3 table
      ShortName                                    Instances                                   Me
    _______________    ____________________________________________________________________    ___

    "ExampleSystem"    {["ASAM.C.SCALAR.GAIN"    "ASAM.C.SCALAR.BITMASK_0001"    ...    ]}    "NO
```

Examine and Modify Simple Calibration Parameters

Use `getValue` to extract the value of an instance from the CDFX object. Use `setValue` to modify the value of the instance.

```
iValueScalar = getValue(cdfxObj, "ASAM.C.SCALAR.GAIN")
```

```
iValueScalar = 3
```

```
iValueScalarNew = iValueScalar + 20;
setValue(cdfxObj, "ASAM.C.SCALAR.GAIN", iValueScalarNew);
iValueScalarNew = getValue(cdfxObj, "ASAM.C.SCALAR.GAIN")
```

```
iValueScalarNew = 23
```

Work with More Complex Parameter Types

Certain instance categories contain more than just a physical value. These instances are often multi-dimensional arrays that are scaled according to an axis. Calling `getValue` on these instances returns a structure that contains each axis as a separate field, distinct from `PhysicalValue`.

To inspect the CUBOID instance, first call `getValue`, then examine the properties of the returned structure. Notice that there is additional data associated with each axis, including the type of axis, its physical values, and whether the axis values are referenced from another instance on the CDFX object.

```
iValueMap = getValue(cdfxObj, "ASAM.C.MAP")
```

```
iValueMap = struct with fields:
    PhysicalValue: [5x5 double]
            Axis1: [1x1 struct]
            Axis2: [1x1 struct]
```

```
disp(iValueMap.PhysicalValue)
```

```
     2    15    27    40    55
     5    17    30    42    57
     7    20    32    47    60
    10    22    35    50    62
    12    25    37    52    65
```

```
disp(iValueMap.Axis1)
```

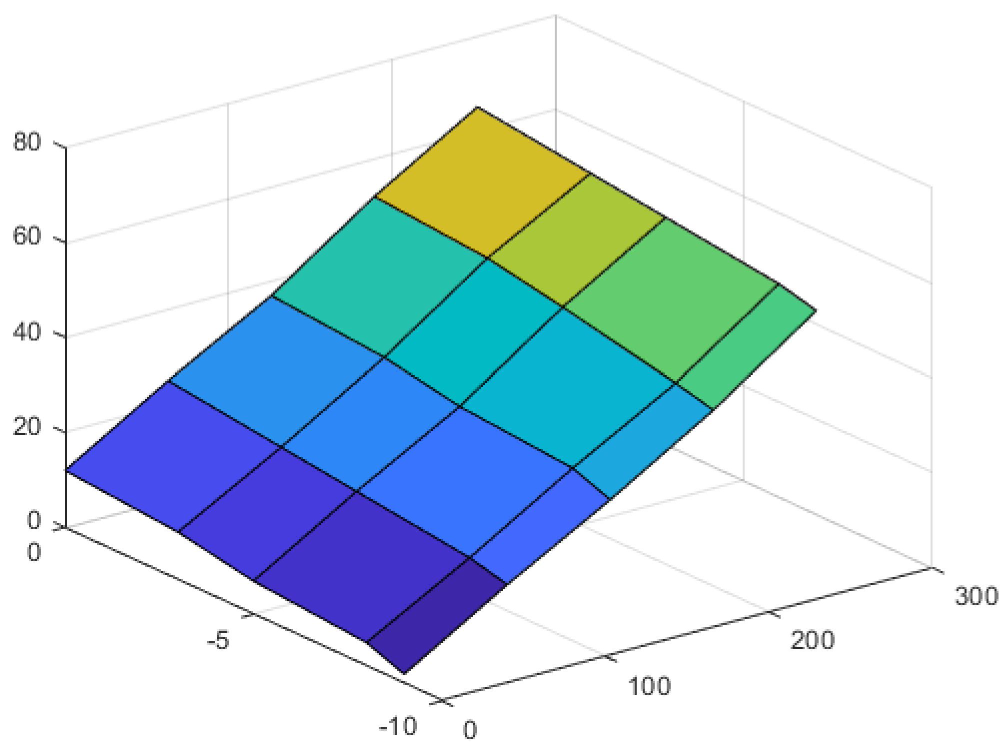
```
    ReferenceName: ""
         Category: "STD_AXIS"
    PhysicalValue: [0 63 126 189 252]
     IsReferenced: 0
```

```
disp(iValueMap.Axis2)
```

```
    ReferenceName: "ASAM.C.COM_AXIS"
         Category: "COM_AXIS"
    PhysicalValue: [-9 -8 -5 -3 0]
     IsReferenced: 1
```

We can also visualize the instance values using MATLAB plotting functions. For multidimensional arrays, use the physical values of the axes structures to define the axes on the plot.

```
surf("ZDataSource", "iValueMap.PhysicalValue", "XDataSource", "iValueMap.Axis1.PhysicalValue", "Y
refreshdata;
```



Modifying the physical value of this instance works the same as for scalars. Update the physical value field of the structure and pass it back to `setValue`.

```
iValueMap.PhysicalValue(:, 1) = iValueMap.PhysicalValue(:, 1)*2;
setValue(cdfxObj, "ASAM.C.MAP", iValueMap);
```

Now we can observe that the changes have been committed to the CDFX object in the workspace.

```
iValueMapNew = getValue(cdfxObj, "ASAM.C.MAP")
```

```
iValueMapNew = struct with fields:
    PhysicalValue: [5x5 double]
            Axis1: [1x1 struct]
            Axis2: [1x1 struct]
```

```
disp(iValueMapNew.PhysicalValue)
```

```
     4    15    27    40    55
    10    17    30    42    57
    14    20    32    47    60
    20    22    35    50    62
    24    25    37    52    65
```

To modify the axis values of this instance, we first need to know if the axis we want to modify is referenced or not. This can be determined by examining the `IsReferenced` field of each axis structure. If the axis is not referenced, we simply modify the `PhysicalValue` field of the axis structure and pass the top-level structure back to `setValue`.

```
disp(iValueMapNew.Axis1.PhysicalValue)
```

```
     0    63   126   189   252
```

```
iValueMapNew.Axis1.PhysicalValue = iValueMapNew.Axis1.PhysicalValue*10;
setValue(cdfxObj, "ASAM.C.MAP", iValueMapNew);
iValueMapNewAxis = getValue(cdfxObj, "ASAM.C.MAP");
disp(iValueMapNewAxis.Axis1.PhysicalValue)
```

```
           0         630        1260        1890        2520
```

However, some axes are not defined on the instance itself, and are instead referenced from another instance. There are specific instance categories for representing referenced axis values (COM_AXIS, RES_AXIS, and CURVE_AXIS). Attempting to modify a referenced axis from a referencing instance will result in an error. The solution is to update the values directly on the axis instance itself. Information on whether an axis is using referenced values, including the short name of the instance being referenced can be found on the axis fields of the top-level structure.

```
iValueCommonAxis = getValue(cdfxObj, iValueMapNewAxis.Axis2.ReferenceName)
```

```
iValueCommonAxis = 1×5

    -9    -8    -5    -3     0
```

```
iValueCommonAxis(:) = 1:5;
setValue(cdfxObj, iValueMapNewAxis.Axis2.ReferenceName, iValueCommonAxis);
```

Now that we have modified the original instance, we can observe that the changes are reflected in the referencing instance.

```
iValueMapNew = getValue(cdfxObj, "ASAM.C.MAP")
```

```
iValueMapNew = struct with fields:
    PhysicalValue: [5x5 double]
            Axis1: [1x1 struct]
            Axis2: [1x1 struct]
```

```
iValueMapNew.Axis2.PhysicalValue
```

```
ans = 1×5

     1     2     3     4     5
```

Export Calibration Data to a File

Using `write` function, you can write back to the same file or to a new file by specifying a filepath.

```
write(cdfxObj, "NewExampleFile.cdfx");
```

Use CDFX-Files with Simulink

This example shows how to use calibration data from a CDFX-file as inputs to a Simulink model.

Import Data

Import the calibration data using the `cdfx` function.

```
cdfxObj = cdfx("CDFXExampleFile.cdfx")
```

```
cdfxObj = 
  CDFX with properties:

       Name: "CDFXExampleFile.cdfx"
       Path: "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex88524458\CDFXExampleFile
    Version: "CDF20"
```

Instantiate Local Variables

Use `getValue` to extract the desired parameters into the MATLAB workspace.

```
gainParam = getValue(cdfxObj, "ASAM.C.SCALAR.GAIN")
```

```
gainParam = 3
```

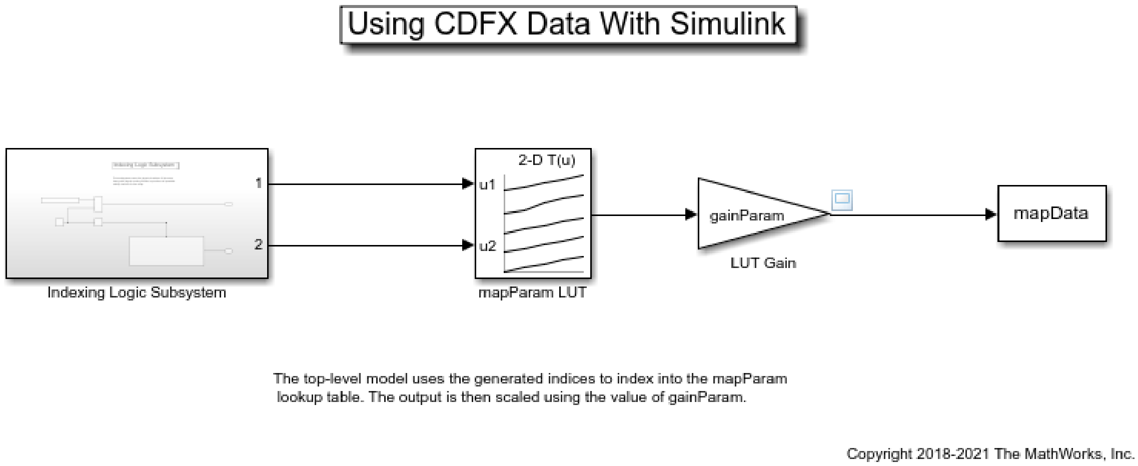
```
mapParam = getValue(cdfxObj, "ASAM.C.MAP")
```

```
mapParam = struct with fields:
    PhysicalValue: [5x5 double]
            Axis1: [1x1 struct]
            Axis2: [1x1 struct]
```

Lookup-Gain Model

```
open_system("CDFXSimulinkModel.slx");
cdfxMdl = gcs
```

```
cdfxMdl = 
'CDFXSimulinkModel'
```

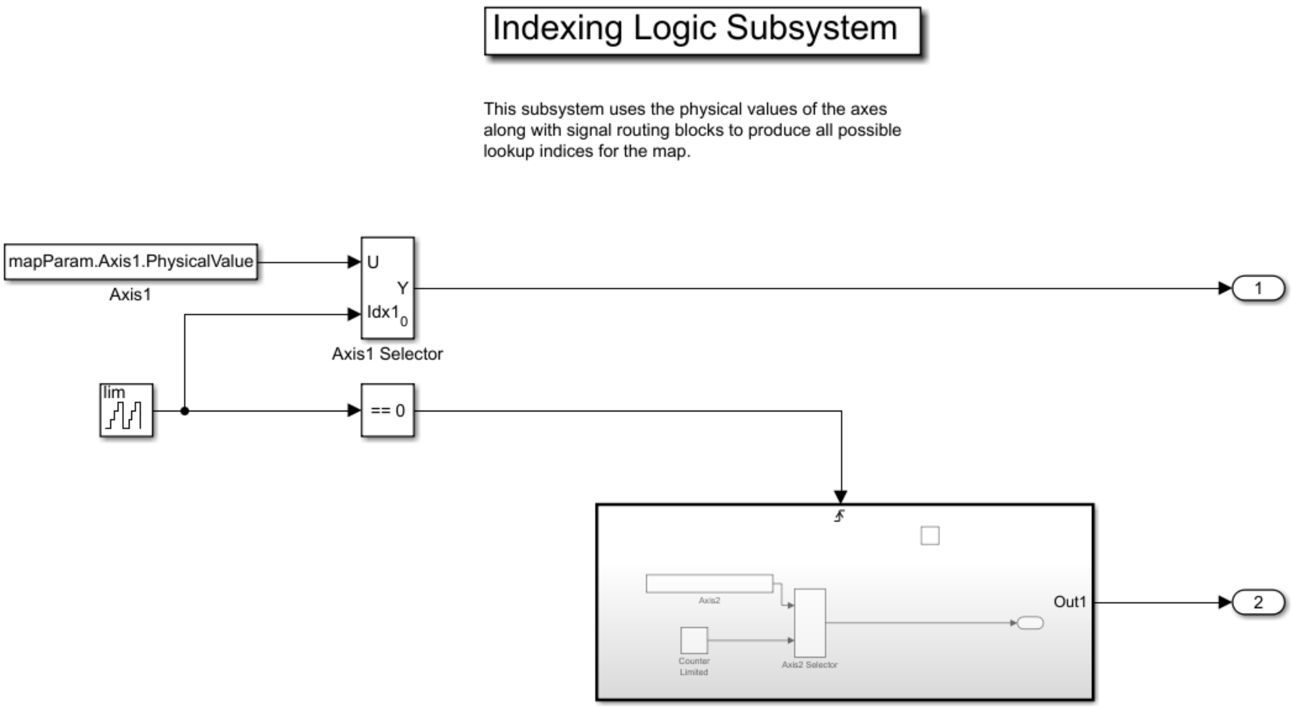


This model contains:

- 2-D Lookup Table block to represent the ASAM.C.MAP parameter from the CDFX-file. The "Table data" field represents the physical value of the instance, and the "Breakpoint" fields represent the physical values of the axes.
- Gain block to represent the ASAM.C.SCALAR.GAIN parameter from the CDFX-file.
- To Workspace block to log the simulation data.

Indexing Logic Subsystem

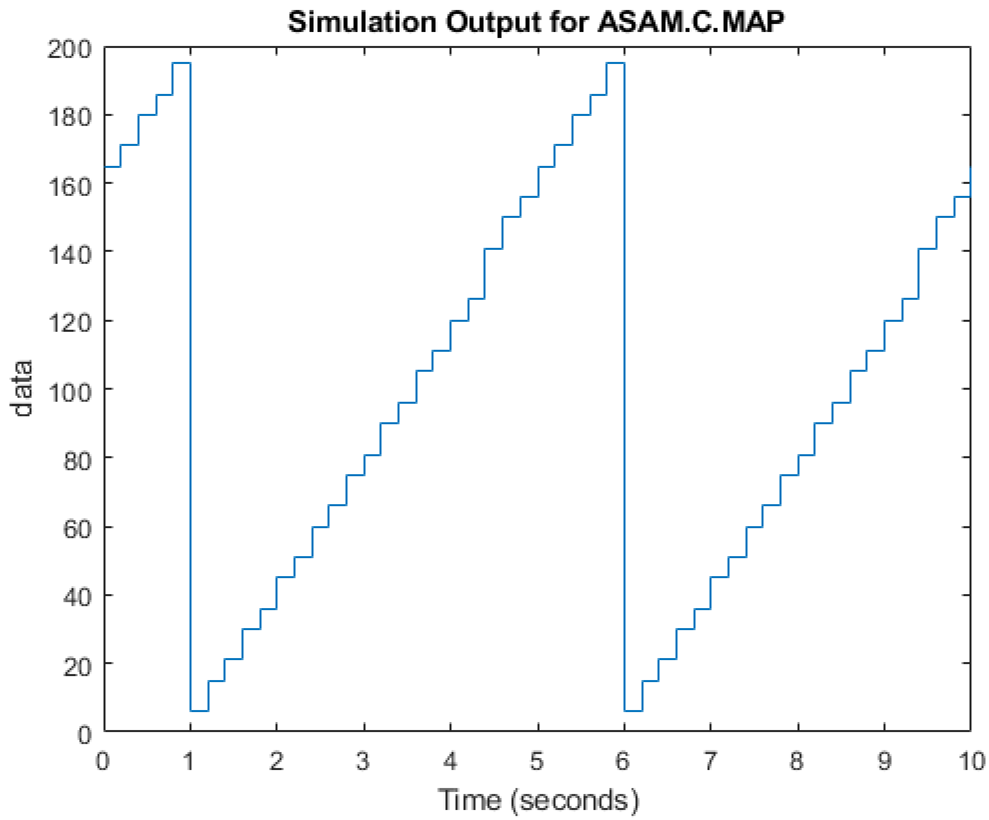
The Indexing Logic subsystem uses the physical values of the axes of the ASAM.C.MAP parameter, along with signal routing blocks and a triggered subsystem, to produce all valid combinations of lookup indices. This configuration can be useful if you need to test across the full range of possible input values of a calibration parameter.



Log Output Data in MATLAB

The output of the simulation is sent to MATLAB by the To Workspace block, where it is stored as a timeseries object called `mapData`. This data can now be inspected and visualized in the MATLAB workspace.

```
sim(cdfxMdl);
plot(mapData)
title("Simulation Output for ASAM.C.MAP")
```



% Copyright 2018-2021 The MathWorks, Inc.

Use CDFX-Files with Simulink Data Dictionary

This example shows how to store calibration data from an ASAM CDFX-file in a data dictionary and use these values as parameters to a Simulink model.

Import Data

Import the calibration data using the `cdfx` function.

```
cdfxObj = cdfx("CDFXExampleFile.cdfx")
```

```
cdfxObj = 
  CDFX with properties:

       Name: "CDFXExampleFile.cdfx"
       Path: "/mathworks/home/rollinb/Documents/MATLAB/Examples/vnt-ex73237310-20190405222527/CDI
    Version: "CDF20"
```

Create and Populate Data Dictionary with Calibration Data

Use `getValue` to extract the desired parameters into the MATLAB workspace.

```
dictName = "CDFXExampleDD.sldd"
```

```
dictName = 
"CDFXExampleDD.sldd"
```

Check if dictionary is already in the working folder.

```
if isfile(dictName)
    % If data dictionary exists, open it.
    dDict = Simulink.data.dictionary.open(dictName)
else
    % If dictionary does not exist, create it and populate with CDFX data.
    dDict = Simulink.data.dictionary.create(dictName)
    ddSection = getSection(dDict, "Design Data")

    addEntry(ddSection, "gainParam", getValue(cdfxObj, "ASAM.C.SCALAR.GAIN"))
    addEntry(ddSection, "mapParam", getValue(cdfxObj, "ASAM.C.MAP"))
end
```

```
dDict = 
  Dictionary with properties:

                  DataSources: {0×1 cell}
        HasAccessToBaseWorkspace: 0
    EnableAccessToBaseWorkspace: 0
               HasUnsavedChanges: 0
                 NumberOfEntries: 2
```

Display contents of the data dictionary.

```
listEntry(dDict)
```

```
  Section       Name        Status   DataSource           LastModified        LastModifiedBy   Cla
```

```
  Design Data    gainParam           CDFXExampleDD.sldd    2019-04-05 22:33    rollinb          dou
  Design Data    mapParam            CDFXExampleDD.sldd    2019-04-05 22:33    rollinb          str
```

Link Data Dictionary to Simulink Model

Open the Simulink model, then use `set_param` to link the existing data dictionary to your model. This will allow the model to access the values defined within the dictionary.

```
open_system("CDFXSLDDModel.slx");
cdfxMdl = gcs
```

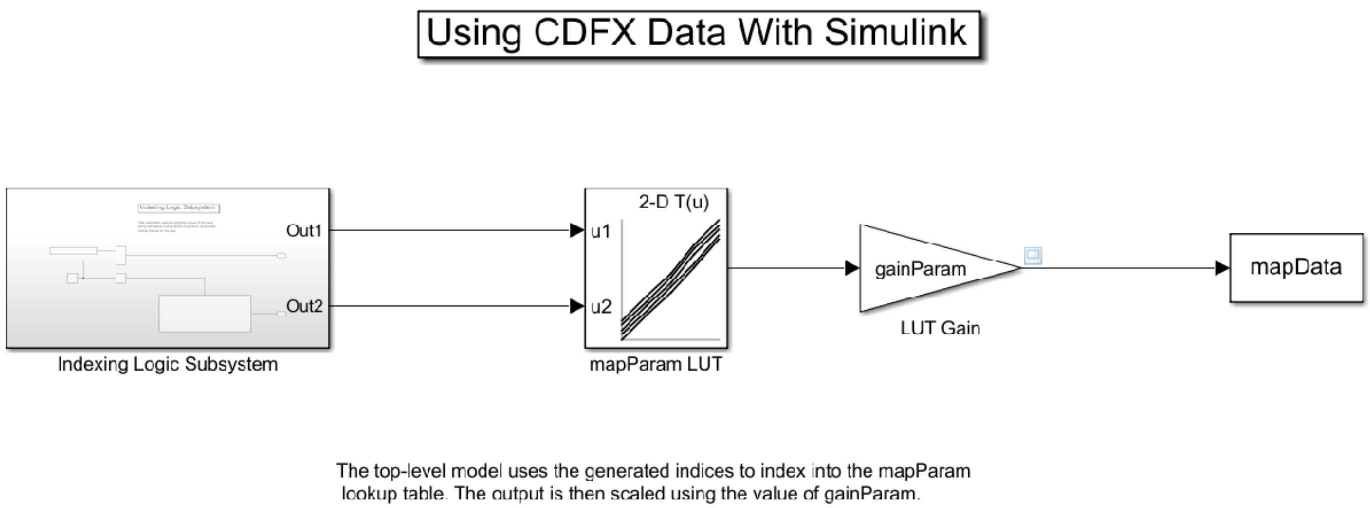
```
cdfxMdl = 
'CDFXSLDDModel'
```

```
set_param(gcs, "DataDictionary", dictName)
```

We can now close the connection to the data dictionary.

```
close(dDict)
```

Lookup-Gain Model

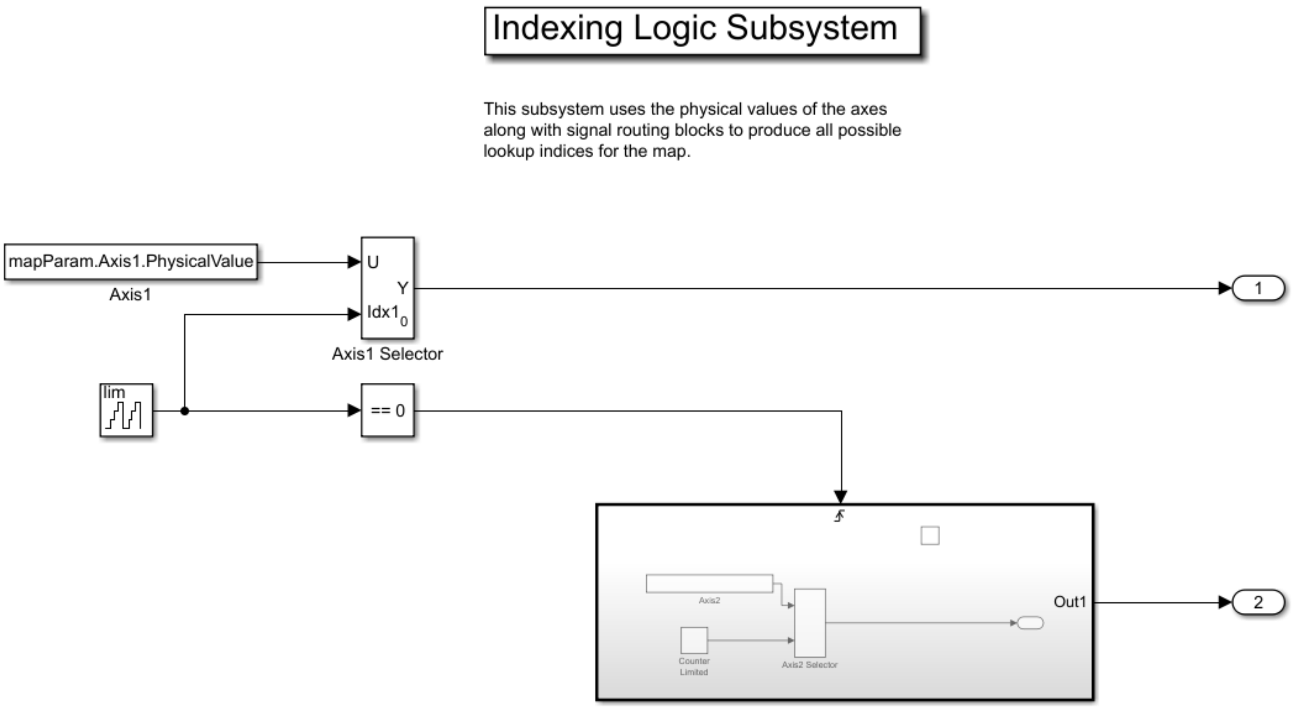


This model contains:

- 2-D Lookup Table block to represent the `ASAM.C.MAP` parameter from the CDFX-file. The "Table data" field represents the physical value of the instance, and the "Breakpoint" fields represent the physical values of the axes.
- Gain block to represent the `ASAM.C.SCALAR.GAIN` parameter from the CDFX-file.
- To Workspace block to log the simulation data.

Indexing Logic Subsystem

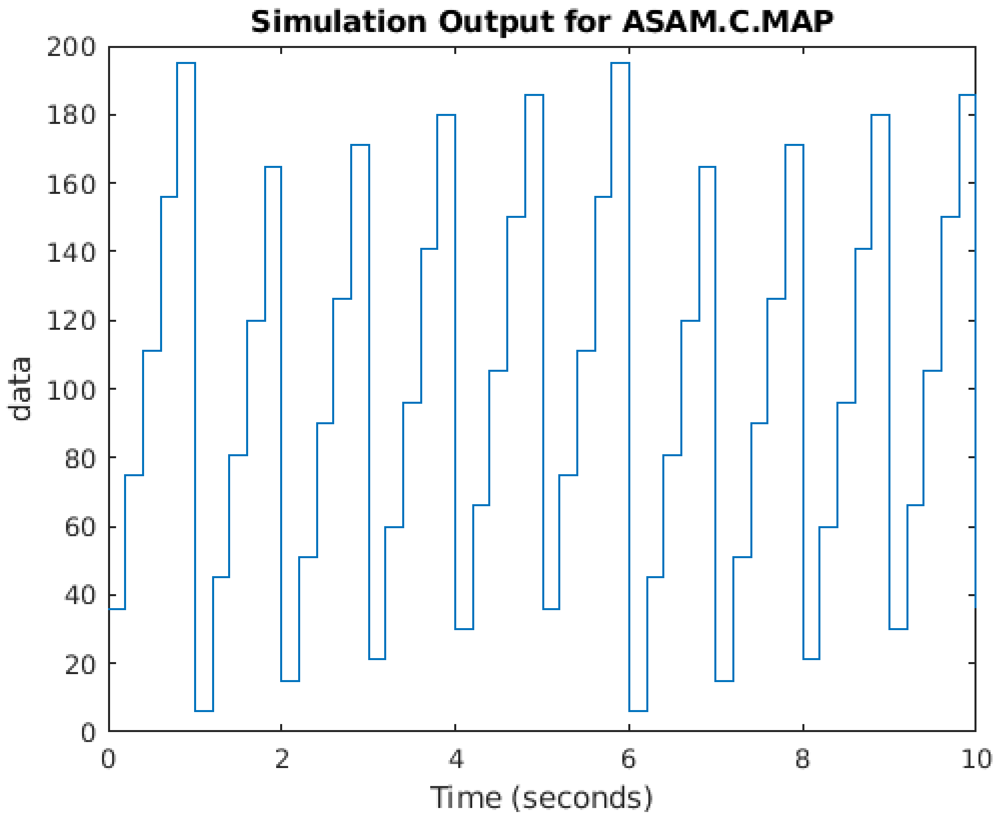
The Indexing Logic subsystem uses the physical values of the axes of the `ASAM.C.MAP` parameter, along with signal routing blocks and a triggered subsystem, to produce all valid combinations of lookup indices. This configuration can be useful if you need to test across the full range of possible input values of a calibration parameter.



Log Output Data in MATLAB

The output of the simulation is sent to MATLAB by the To Workspace block, where it is stored as a timeseries object called `mapData`. This data can now be inspected and visualized in the MATLAB workspace.

```
sim(cdfxMdl);
plot(mapData)
title("Simulation Output for ASAM.C.MAP")
```

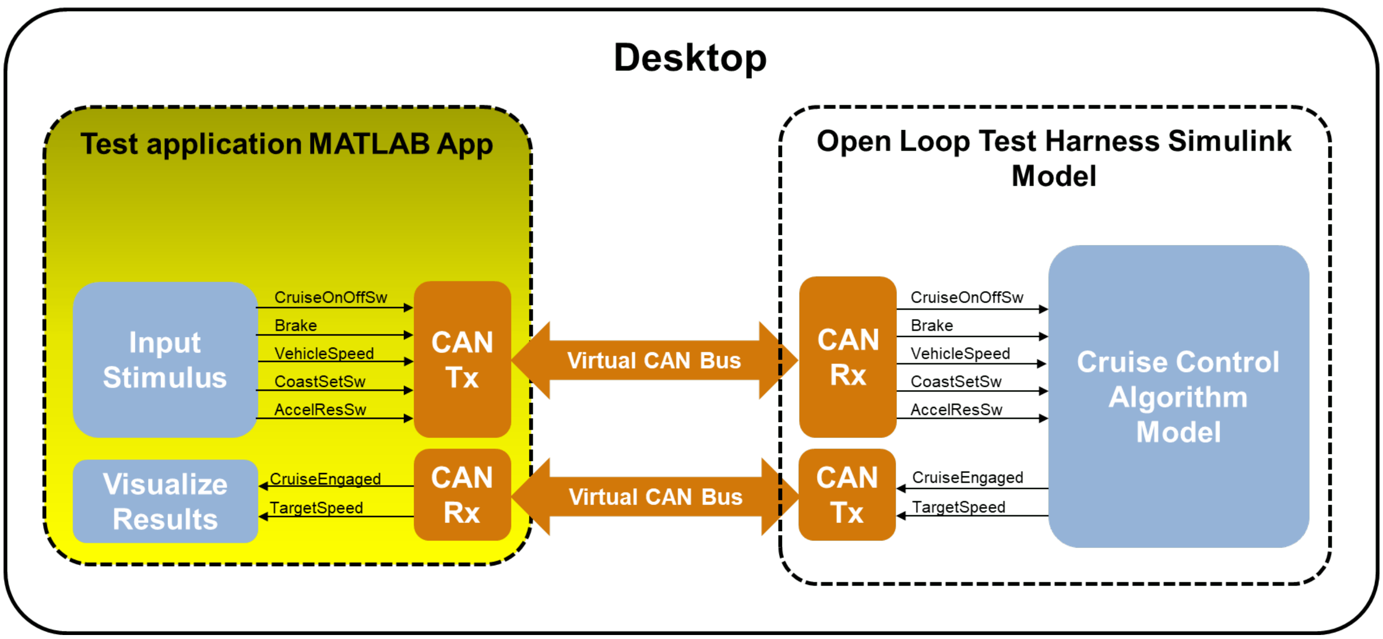


% Copyright 2018-2021 The MathWorks, Inc.

Develop an App Designer App for a Simulink Model Using CAN

This example shows how to construct a test application user interface (UI) and connect it to a Simulink model using virtual CAN channels. The test application UI is constructed using MATLAB App Designer™ along with several Vehicle Network Toolbox™ functions to provide a virtual CAN bus interface to a Simulink model of an automotive cruise control application. The test application UI allows a user to provide input stimulus to the cruise control algorithm model, observe results fed back from the model, log CAN messages to capture test stimuli, and replay logged CAN messages to debug and correct issues with the algorithm model. The example shows the key Vehicle Network Toolbox functions and blocks used to implement CAN communication in the following areas:

- The test application UI supporting communication with the Simulink algorithm model for testing via CAN
- The test application UI supporting logging and replaying of CAN data
- The Simulink algorithm model



Add Virtual CAN Channel Communication to the UI

In this section, we describe the key Vehicle Network Toolbox functions used to add a CAN channel interface to the Simulink Cruise Control algorithm test application model. This covers the following topics:

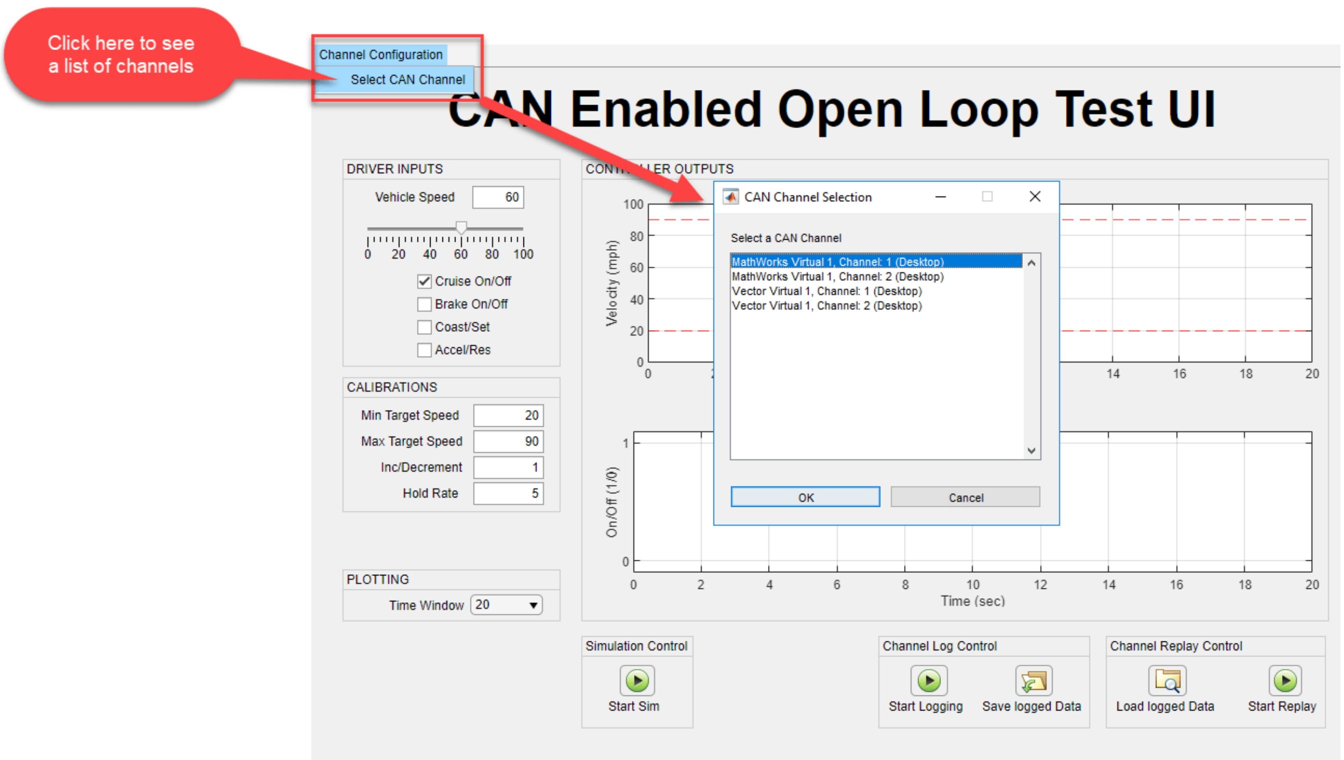
- Getting a list of available CAN channels
- Formatting the channel information for channel creation
- Creating the channel in the UI
- Configure the UI to transmit and receive CAN messages
- Starting and stopping the channel
- Extracting selected messages

Open App Designer

Open the test application UI in App Designer. With the test application UI open in App Designer you can alternate between the "Design" and "Code" views to follow along as you explore the controls and corresponding MATLAB code to communicate with the Simulink Cruise Control algorithm model via virtual CAN channels. Use the following command to open the example UI: `appdesigner('CruiseControlTestUI.mlapp')`.

List Available CAN Channels

First, implement a mechanism to find and present a list of the available CAN channels for a user to select. For this, we added a "Channel Configuration" menu item at the top left corner of the test application UI. It has a "Select CAN Channel" sub-menu.



When the user clicks on the "Select CAN Channel" sub-menu, the helper function `getAvailableCANChannelInfo(app)` is called via the sub-menu callback. `getAvailableCANChannelInfo()` uses the Vehicle Network Toolbox function `canChannelList` to detect the available CAN channels, as shown in the code fragment below:

```
function getAvailableCANChannelInfo(app)
    % Get a table containing all available CAN channels and devices.
    app.canChannelInfo = canChannelList;

    % Format CAN channel information for display on the UI.
    app.availableCANChannelsForDisplay = formatCANChannelEntryForDisplay(app);

    % Save the number of available constructors.
    app.numConstructors = numel(app.canChannelInfo.Vendor);
end
```

Run `canChannelList` to see how the available CAN channel information is stored.

```
canChannels = canChannelList
```

```
canChannels=12×6 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"       "VN1610 1"        1        "VN1610"      "CAN, CAN FD"      "46457"
    "Vector"       "VN1610 1"        2        "VN1610"      "CAN, CAN FD"      "46457"
    "Vector"       "VN1610 3"        1        "VN1610"      "CAN, CAN FD"      "46456"
    "Vector"       "VN1610 3"        2        "VN1610"      "CAN, CAN FD"      "46456"
    "Vector"       "VN1610 2"        1        "VN1610"      "CAN, CAN FD"      "48599"
    "Vector"       "VN1610 2"        2        "VN1610"      "CAN, CAN FD"      "48599"
    "Vector"       "Virtual 1"       1        "Virtual"     "CAN, CAN FD"      "0"
    "Vector"       "Virtual 1"       2        "Virtual"     "CAN, CAN FD"      "0"
    "Kvaser"       "Virtual 1"       1        "Virtual"     "CAN, CAN FD"      "0"
    "Kvaser"       "Virtual 1"       2        "Virtual"     "CAN, CAN FD"      "0"
```

The list of channels returned from `canChannelList` is stored in the UI property *`app.canChannelInfo`* and then displayed to the user in a "CAN Channel Selection" `listdlg` as shown in the screen shot above.

Format Channel List for Channel Configuration

The user selects a CAN channel from the "CAN Channel Selection" `listdlg`. The `listdlg` returns an index corresponding to the user's selection. This index is passed to the helper function `formatCANChannelConstructor`.

```
function canChannelConstructor = formatCANChannelConstructor(app, index)
    canChannelConstructor = "canChannel(" + "'" + app.canChannelInfo.Vendor(index) + "'" + ", " +
end
```

As shown in the code fragment above, `formatCANChannelConstructor` uses the strings stored in the table of CAN channels, *app.canChannelInfo*, to assemble the channel object constructor string corresponding to the channel the user selected from the channel selector list dialog box. To see an example of a CAN channel constructor string, execute the code shown below.

```
index = 1;
canChannelConstructor = "canChannel(" + "'" + canChannels.Vendor(index) + "'" + ", " + "'" + canC
```

```
canChannelConstructor = 
"canChannel('MathWorks', 'Virtual 1', 1)"
```

The CAN channel constructor string is stored in the app UI property *`app.canChannelConstructorSelected`* and will be used later to create the selected CAN channel object in the application UI as well as to update the Vehicle Network Toolbox Simulink blocks that implement the CAN channel interface in the Simulink Cruise Control algorithm model.

Create CAN Channel in the UI

When the UI is first opened and initialized, the formatted CAN channel constructor string stored in *app.canChannelConstructorSelected* is used by the helper function `setupCANChannel` to create an instance of a CAN channel object, connect a network configuration database (.DBC) file, and set the bus speed as shown in the code fragment below. The resulting channel object is stored in the UI property *app.canChannelObj*.

```
function setupCANChannel(app)
    % Open CAN database file.
    db = canDatabase('CruiseControl.dbc');

    % Create a CAN channel for sending and receiving messages.
    app.canChannelObj = eval(app.canChannelConstructorSelected);

    % Attach CAN database to channel for received message decoding.
    app.canChannelObj.Database = db;

    % Set the baud rate (can only do this if the UI has channel initialization access).
    if app.canChannelObj.InitializationAccess
        configBusSpeed(app.canChannelObj, 500000);
    end
end
```

To see an example CAN database object, execute the following:

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

```
db = 
  Database with properties:

             Name: 'CruiseControl'
             Path: '\\fs-01-mi\shome$\rollinb\Documents\MATLAB\Examples\vnt-ex00964061\CruiseCont
            Nodes: {2×1 cell}
         NodeInfo: [2×1 struct]
         Messages: {2×1 cell}
      MessageInfo: [2×1 struct]
       Attributes: {'BusType'}
    AttributeInfo: [1×1 struct]
         UserData: []
```

To see an example CAN channel object, execute the following:

```
% Instantiate the CAN channel object using the channel constructor string.
canChannelObj = eval(canChannelConstructor);

% Attach the CAN database to the channel object.
canChannelObj.Database = db
```

```
canChannelObj = 
  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: [0×0 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: [1×1 can.Database]
                UserData: []
```

`setupCANChannel` uses the following Vehicle Network Toolbox functions:

- `canChannel` to instantiate the channel object using the eval command and the CAN channel constructor string stored in the app UI property *`app.canChannelConstructorSelected`*. The resultant channel object is stored in the app UI property *`app.canChannelObj`*.
- `canDatabase` to create a CAN database (.DBC) object representing the DBC-file. This object is stored in the "*Database*" property of the channel object.

Setup to Transmit CAN Messages

After setting up the selected CAN channel object and storing it in the UI property *app.canChannelObj*, the next step is to call the helper function `setupCANTransmitMessages`, shown in the code fragment below. `setupCANTransmitMessages defines` the CAN message to transmit from the UI, populates the message payload with signals, assigns each signal a value, and queues the message to transmit periodically once the CAN channel is started.

```
function setupCANTransmitMessages(app)
    % Create a CAN message container.
    app.cruiseControlCmdMessage = canMessage(app.canChannelObj.Database, 'CruiseCtrlCmd');

    % Fill the message container with signals and assign values to each signal.
    app.cruiseControlCmdMessage.Signals.S01_CruiseOnOff = logical2Numeric(app, app.cruisePowerChe
    app.cruiseControlCmdMessage.Signals.S02_Brake =  logical2Numeric(app, app.brakeOnOffCheckBox
    app.cruiseControlCmdMessage.Signals.S03_VehicleSpeed =  app.vehicleSpeedSlider.Value;
    app.cruiseControlCmdMessage.Signals.S04_CoastSetSw =  logical2Numeric(app, app.cruiseCoastSet
    app.cruiseControlCmdMessage.Signals.S05_AccelResSw =  logical2Numeric(app, app.cruiseAccelRes

    % Set up periodic transmission of this CAN message.  Actual transmission starts/stops with CA
    transmitPeriodic(app.canChannelObj, app.cruiseControlCmdMessage, 'On', 0.1);
end
```

To see what the CAN message object looks like, execute the following:

```
cruiseControlCmdMessage = canMessage(canChannelObj.Database, 'CruiseCtrlCmd')
```

```
cruiseControlCmdMessage = 
  Message with properties:

   Message Identification
    ProtocolMode: 'CAN'
              ID: 256
        Extended: 0
            Name: 'CruiseCtrlCmd'

   Data Details
       Timestamp: 0
            Data: [0 0]
         Signals: [1×1 struct]
          Length: 2

   Protocol Flags
           Error: 0
          Remote: 0

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

```
cruiseControlCmdMessage.Signals
```

```
ans = struct with fields:
    S03_VehicleSpeed: 0
      S05_AccelResSw: 0
      S04_CoastSetSw: 0
           S02_Brake: 0
     S01_CruiseOnOff: 0
```

`setupCANTransmitMessages` uses the following Vehicle Network Toolbox functions:

- `canMessage` to build a CAN message based defined in the CAN database object.
- `transmitPeriodic` to queue the message stored in the UI property *app.cruiseControlCmdMessage* for periodic transmission on the channel defined by the channel object stored in the UI property *app.canChannelObj*, at the rate specified by the last argument, in this case every 0.1 seconds.

Setup to Receive CAN Messages

The UI needs to receive CAN messages on a periodic basis to update the plots with feedback from the Cruise Control algorithm within the Simulink model. To achieve this, we first create a MATLAB timer object as shown in the code fragment below.

```
% create a timer to receive CAN msgs
app.receiveCANmsgsTimer = timer('Period', 0.5,...
    'ExecutionMode', 'fixedSpacing', ...
    'TimerFcn', @(~,~)receiveCANmsgsTimerCallback(app));
```

The timer object will call the timer callback function `receiveCANmsgsTimerCallback` every 0.5 seconds. `receiveCANmsgsTimerCallback`, shown in the code fragment below, retrieves all the CAN messages from the bus, uses the helper function `getCruiseCtrlFBCANmessage` to extract the CAN messages fed back from the Cruise Control Algorithm model, and updates the UI plots with the extracted CAN message data.

```
% receiveCANmsgsTimerCallback Timer callback function for GUI updating
function receiveCANmsgsTimerCallback(app)
    try
        % Receive available CAN messages.
        msg = receive(app.canChannelObj, Inf, 'OutputFormat', 'timetable');

        % Update Cruise Control Feedback CAN message data.
        newFbData = getCruiseCtrlFBCANmessage(app, msg);

        if ~newFbData
            return;
        end

        % Update target speed and engaged plots with latest data from CAN bus.
        updatePlots(app);
    catch err
        disp(err.message)
    end
end
```

To see what the messages returned from the `receive` command look like, run the following code:

```
% Queue periodic transmission of a CAN message to generate some message data once the channel
% starts.
transmitPeriodic(canChannelObj, cruiseControlCmdMessage, 'On', 0.1);

% Start the channel.
start(canChannelObj);

% Wait 1 second to allow time for some messages to be generated on the bus.
pause(1);

% Retrieve all messages from the bus and output the results as a timetable.
msg = receive(canChannelObj, Inf, 'OutputFormat','timetable')
```

```
msg=31×8 timetable
        Time          ID     Extended          Name              Data        Length      Signals
    ____________     ___    ________    ________________    ____________    ______    ____________

    0.0066113 sec    256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.059438 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.16047 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.26045 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.36045 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.46046 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.56046 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.66046 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.75945 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.86044 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    0.95945 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    1.0594 sec       256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    1.1594 sec       256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    1.2594 sec       256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    1.3595 sec       256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
    1.4605 sec       256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struc
      ⋮
```

```
% Stop the channel.
stop(canChannelObj)
```

`receiveCANmsgsTimerCallback` uses the following Vehicle Network Toolbox functions:

- `receive` to retrieve CAN messages from the CAN bus. In this case, the function is configured to retrieve all messages since the previous invocation and output the results as a MATLAB timetable.

Extract Select CAN Messages

The helper function `getCruiseCtrlFBCANmessage` filters out the "CruiseCtrlFB" messages from all the retrieved CAN messages, extracts the *tspeedFb* and *engagedFb* signals from these messages, and concatenates these to MATLAB timeseries objects for the *tspeedFb* and *engagedFb* signals. These timeseries objects are stored in the UI properties *app.tspeedFb* and *app.engagedFb*, respectively. The stored timeseries signals are used to update the plots for each signal on the UI. Note the use of the `seconds` method to convert the time data stored in the timetable from a duration array into the equivalent numeric array in units of seconds in the timeseries objects for each signal.

```
function newFbData = getCruiseCtrlFBCANmessage(app, msg)
    % Exit if no messages were received as there is nothing to update.
    if isempty(msg)
        newFbData = false;
        return;
    end

    % Extract signals from all CruiseCtrlFB messages.
    cruiseCtrlFBSignals = canSignalTimetable(msg, "CruiseCtrlFB");

    % if no messages then just return as there is nothing to do
    if isempty(cruiseCtrlFBSignals)
        newFbData = false;
        return;
    end

    if ~isempty(cruiseCtrlFBSignals)
        % Received new Cruise Control Feedback messages, so create time series from CAN signal da
        % save the Target Speed feedback signal.
        if isempty(app.tspeedFb) % cCeck if target speed feedback property has been initialized.
            app.tspeedFb = cell(2,1);

            % It appears Simulink.SimulationData.Dataset class is not
            % compatible with MATLAB Compiler, so change the way we store data
            % from a Dataset format to cell array.

            % Save target speed actual data.
            app.tspeedFb = timeseries(cruiseCtrlFBSignals.F02_TargetSpeed, seconds(cruiseCtrlFBSi
                'Name','CruiseControlTargetSpeed');
        else  % Add to existing data.
            % Save target speed actual data.
            app.tspeedFb = timeseries([app.tspeedFb.Data; cruiseCtrlFBSignals.F02_TargetSpeed],
                [app.tspeedFb.Time; seconds(cruiseCtrlFBSignals.Time)],'Name','CruiseControlTarge
        end

        % Save the Cruise Control Engaged actual signal.
        % Check if Cruise engaged property has been initialized.
        if isempty(app.engagedFb)
            app.engagedFb = cell(2,1);
```

```
            % It appears Simulink.SimulationData.Dataset class is not
            % compatible with MATLAB Compiler, so change the way we store data
            % from a Dataset format to cell array.

            % Save cruise engaged command data.
            app.engagedFb = timeseries(cruiseCtrlFBSignals.F01_Engaged,seconds(cruiseCtrlFBSigna
                'Name','CruiseControlEngaged');
        else  % Add to existing logsout.
            % Save cruise engaged command data.
            app.engagedFb = timeseries([app.engagedFb.Data; cruiseCtrlFBSignals.F01_Engaged], ..
                [app.engagedFb.Time; seconds(cruiseCtrlFBSignals.Time)],'Name','CruiseControlEnga
        end

        newFbData = true;
    end
end
```

The helper function `getCruiseCtrlFBCANmessage` uses the following Vehicle Network Toolbox functions:

- `canSignalTimetable` to return a MATLAB timetable containing the signals from the CAN message *CruiseCtrlFB.*

Start the CAN Channel

Once the CAN channel object *app.canChannelObj* has been instantiated and messages have been set up to be transmitted and received, we can now start the channel. When the user clicks the start sim button on the UI, we want the channel to start just before we start running the Simulink model. To achieve this, the helper function `startSimApplication` is called. `startSimApplication`, shown in the code fragment below, checks to make sure we are using a virtual CAN channel, because this is the only type that makes sense if you are using only desktop simulation. Next, it checks to make sure the Simulink model we want to connect to is loaded in memory using the `bdIsLoaded` command. If the model is loaded and is not already running, the UI plots are cleared to accept new signal data, the CAN channel is started using the helper function startCANChannel, and the model is started.

```
function startSimApplication(app, index)
    % Start the model running on the desktop.

    % Check to see if hardware or virtual CAN channel is selected, otherwise do nothing.
    if app.canChannelInfo.DeviceModel(index) == "Virtual"
        % Check to see if the model is loaded before trying to run.
        if bdIsLoaded(app.mdl)
            % Model is loaded, now check to see if it is already running.
            if ~strcmp('running',get_param(app.mdl,'SimulationStatus'))
                % Model is not already running, so start it
                % flush the CAN Receive message buffers.
                app.tspeedFb = [];
                app.engagedFb = [];

                % Clear figure window.
                cla(app.tspeedPlot)
                cla(app.engagedPlot)

                % Start the CAN channels and update timer if it isn't already running.
                startCANChannel(app);
```

```
                % Start the model.
                set_param(app.mdl, 'SimulationCommand', 'start');

                % Set the sim start/stop button icon to the stop icon indicating the model has
                % been successfully started and is ready to be stopped at the next button press.
                app.SimStartStopButton.Icon = "IconEnd.png";
                app.StartSimLabel.Text = "Stop Sim";
            else
                % Model is already running, inform the user.
                warnStr = sprintf('Warning: Model %s is already running', app.mdl);
                warndlg(warnStr, 'Warning');
            end
        else
            % Model is not yet loaded, so warn the user.
            warnStr = sprintf('Warning: Model %s is not loaded\nPlease load the model and try aga
            warndlg(warnStr, 'Warning');
        end
    end
end
```

The helper function `startCANChannel` is shown in the code fragment below. This function checks to make sure the channel is not already running before it starts it. Next, it starts the MATLAB timer object so that the timer callback function `receiveCANmsgsTimerCallback`, described in the previous section, is called every 0.5 seconds to retrieve CAN message data from the bus.

```
function startCANChannel(app)
    % Start the CAN channel if it isn't already running.
    try
        if ~app.canChannelObj.Running
            start(app.canChannelObj);
        end
    catch
        % do nothing.
    end

    % Start the CAN receive processing timer - check to see if it is already running. This allows
    % with or without starting and stopping the model running on the real time target.
    if strcmpi(app.receiveCANmsgsTimer.Running, 'off')
        start(app.receiveCANmsgsTimer);
    end
end
```

`startCANchannel` uses the following Vehicle Network Toolbox function:

- `start` to start the CAN channel running. The channel will remain on-line until a *stop* command is issued.

Stop the CAN Channel

When the user clicks the stop sim button on the UI, we want to stop the Simulink model just before stopping the CAN channel. To achieve this, the helper function `stopSimApplication` is called. `stopSimApplication`, shown in the code fragment below, checks to make sure we are using a virtual CAN channel, because this is the only type that makes sense if you are using only desktop simulation. Next, it stops the Simulink model and calls the helper function `stopCANChannel` to stop the CAN channel.

```
function stopSimApplication(app, index)
    % Stop the model running on the desktop.
```

```
    try
        % Check to see if hardware or virtual CAN channel is selected.
        if app.canChannelInfo.DeviceModel(index) == "Virtual"
            % Virtual channel selected, so issue a stop command to the
            % the simulation, even if it is already stopped.
            set_param(app.mdl, 'SimulationCommand', 'stop')

            % Stop the CAN channels and update timer.
            stopCANChannel(app);
        end

        % Set the sim start/stop button text to Start indicating the model has
        % been successfully stopped and is ready to start again at the next
        % button press.
        app.SimStartStopButton.Icon = "IconPlay.png";
        app.StartSimLabel.Text = "Start Sim";
    catch
        % Do nothing at the moment.
    end
end
```

The helper function `stopCANChannel` is shown in the code fragment below. This function stops the CAN channel, then stops the MATLAB timer object so that the timer callback function `receiveCANmsgsTimerCallback`, described previously, is no longer called to retrieve messages from the bus.

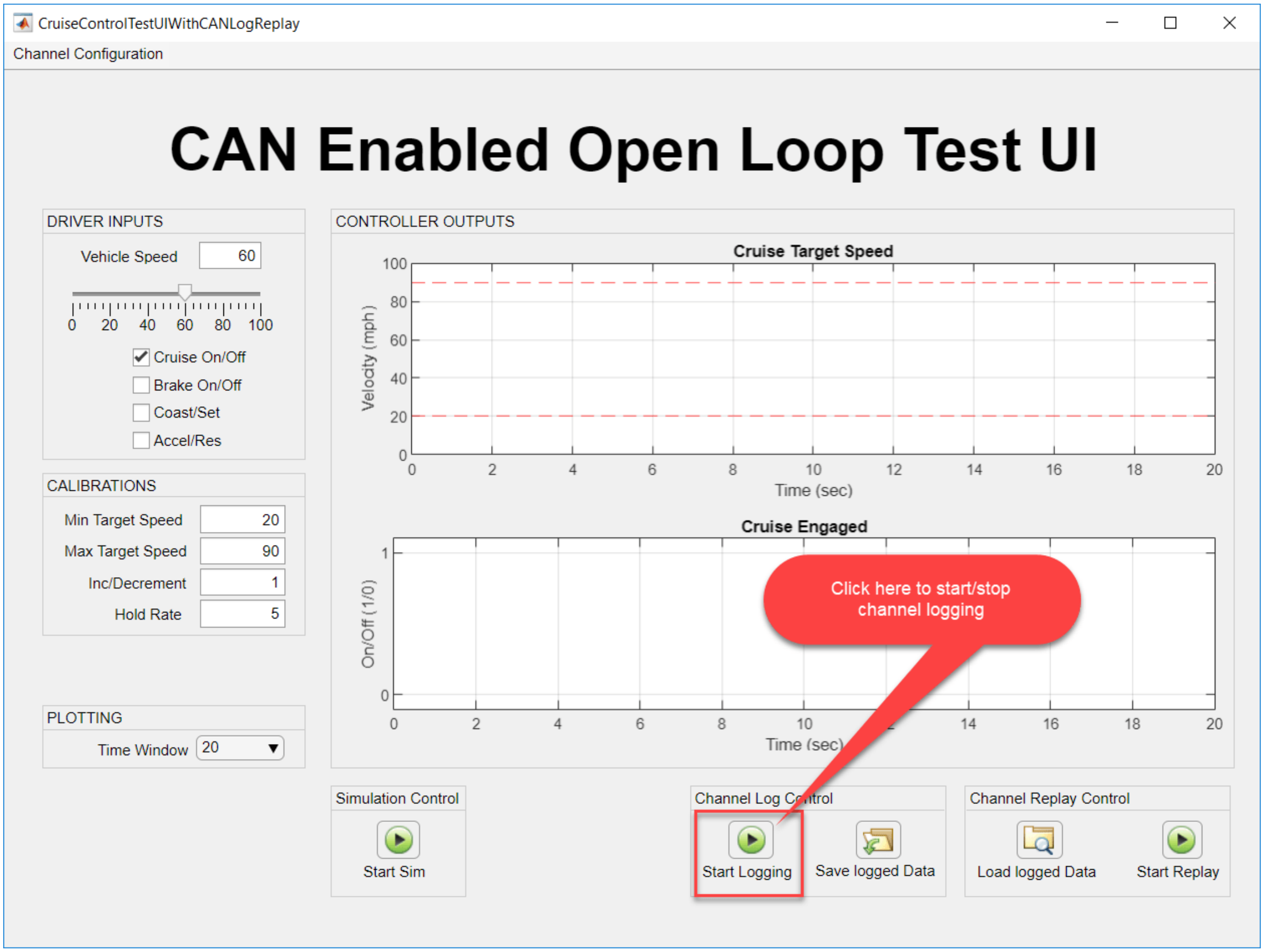
```
function stopCANChannel(app)
    % Stop the CAN channel.
    stop(app.canChannelObj);

    % Stop the CAN message processing timer.
    stop(app.receiveCANmsgsTimer);
end
```

`stopCANchannel` uses the following Vehicle Network Toolbox function:

- `stop` to stop the CAN channel. The channel will remain offline until another *start* command is issued.

Add CAN Log and Replay Capability

In this step, we will describe the key Vehicle Network Toolbox functions used to add the ability to log, save, and replay CAN messages. To implement this functionality, we instantiate a second CAN channel, identical to the first one created. Because the second CAN channel object is identical to the first one, it will see and collect the same messages as the first CAN channel object. The second CAN channel will be started when the user clicks the start logging button on the UI as shown in the UI screen shot below. The channel continues to run and collect messages until the user clicks the stop logging button on the UI. Once the user stops logging CAN messages, we will retrieve all the messages that have accumulated in the message buffer for the second CAN channel object, extract the messages we are interested in replaying, and save them to a MAT-file. Once the messages have been saved, we can replay them using the first CAN channel to provide an input stimulus to the Simulink Cruise Control algorithm model for debugging and algorithm verification purposes.



This description will cover the following topics:

- Setup a channel to log CAN messages
- Starting and Stopping the channel
- Retrieving and extracting logged CAN messages
- Saving the extracted messages to a file
- Loading saved messages from a file
- Start playback of logged CAN messages
- Stop playback of logged CAN messages

Setup a CAN Channel Object in the UI to Log CAN Messages

In a manner directly analogous to how the first CAN channel was instantiated, the formatted CAN channel constructor string stored in *app.canChannelConstructorSelected* is used by a new helper function `setupCANLogChannel` to create a second instance of a CAN channel object, connect the same network configuration database (.DBC) file as was used for the first channel, and set the bus speed as shown in the code fragment below. The resulting channel object is stored in the UI property *app.canLogChannelObj*.

```
function setupCANLogChannel(app)
    % Open CAN database file.
    db = canDatabase('CruiseControl.dbc');

    % Create a CAN channel for sending and receiving messages.
    app.canLogChannelObj = eval(app.canChannelConstructorSelected);

    % Attach CAN database to channel for received message decoding.
    app.canLogChannelObj.Database = db;

    % Set the baud rate (can only do this if the UI has channel initialization access).
    if app.canLogChannelObj.InitializationAccess
        configBusSpeed(app.canLogChannelObj, 500000);
    end
end
```

To see an example CAN database object, execute the following code:

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

```
db =
  Database with properties:

             Name: 'CruiseControl'
             Path: '\\fs-01-mi\shome$\rollinb\Documents\MATLAB\Examples\vnt-ex00964061\CruiseCont
            Nodes: {2×1 cell}
         NodeInfo: [2×1 struct]
         Messages: {2×1 cell}
      MessageInfo: [2×1 struct]
       Attributes: {'BusType'}
    AttributeInfo: [1×1 struct]
         UserData: []
```

To see an example CAN channel object, execute the following code:

```
% Instantiate the CAN channel object using the channel constructor string.
canLogChannelObj = eval(canChannelConstructor);

% Attach the CAN database to the channel object.
canLogChannelObj.Database = db
```

```
canLogChannelObj =
  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: 0
        InitialTimestamp: [0×0 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: [1×1 can.Database]
               UserData: []
```

`setupCANLogChannel` uses the following Vehicle Network Toolbox functions:

- `canChannel` to instantiate the channel object using the eval command and the CAN channel constructor string stored in the app UI property *`app.canChannelConstructorSelected.`* The resultant channel object is stored in the app UI property *`app.canLogChannelObj.`*
- `canDatabase` to create a CAN database (.DBC) object representing the DBC-file. This object is stored in the "*Database*" property of the channel object.

Start the CAN Log Channel

As with the first CAN channel, the CAN channel object *app.canLogChannelOb*, was instantiated when the test application UI is opened. When the user clicks the start Logging button on the UI as shown on the screen shot above, we call the helper function `startCANLogChannel`, shown in the code fragment below. This function checks to see if the second CAN channel is already running and starts it if isn't.

```
function startCANLogChannel(app)
    % Start the CAN Log channel if it isn't already running.
    try
        if ~app.canLogChannelObj.Running
            start(app.canLogChannelObj);
        end
    catch
        % Do nothing.
    end
end
```

`startCANLogChannel` uses the following Vehicle Network Toolbox function:

- `start` to start the CAN channel running. The channel will remain online until a *stop* command is issued.

Stop the CAN Log Channel

When the user clicks the "Stop logging" button on the UI, the button callback calls the helper function `stopCANLogging,` shown in the code fragment below. `stopCANLogging` stops the CAN channel and retrieves all the messages accumulated in the second channel buffer since the second CAN channel was started by the user clicking the "Start Logging" button.

```
function stopCANLogging(app)
    % Stop the CAN Log channel.
    stop(app.canLogChannelObj);

    % Get the messages from the CAN log message queue.
    retrieveLoggedCANMessages(app);

    % Update the button icon and label.
    app.canLoggingStartStopButton.Icon = 'IconPlay.png';
    app.StartLoggingLabel.Text = "Start Logging";
end
```

`stopCANLogging` uses the following Vehicle Network Toolbox function:

- `stop` to stop the CAN channel. The channel will remain offline until another *start* command is issued.

Retrieve and Extract Logged Messages

Once the logging CAN channel has been stopped, the helper function `retrieveLoggedCANMessages`, shown in the code fragment below, is called to retrieve all the CAN messages from the second channel bus. The CAN messages are acquired from the second channel bus using the `receive` command and logical indexing is used to extract the "*CruiseCtrlCmd*" messages from all the message timetable returned by the `receive` command.

```
function retrieveLoggedCANMessages(app)
    try
        % Receive available CAN message
        % initialize buffer to make sure it is empty.
        app.canLogMsgBuffer = [];

        % Receive available CAN messages.
        msg = receive(app.canLogChannelObj, Inf, 'OutputFormat', 'timetable');

        % Fill the buffer with the logged Cruise Control Command CAN message data.
        app.canLogMsgBuffer = msg(msg.Name == "CruiseCtrlCmd", :);
    catch err
        disp(err.message)
    end
end
```

To see what the messages returned from the `receive` command look like in the timetable format, run the following code:

```
% Queue periodic transmission of CAN messages to generate sample message data once the channel s
cruiseControlFbMessage = canMessage(db, 'CruiseCtrlFB');
transmitPeriodic(canChannelObj, cruiseControlFbMessage, 'On', 0.1);
transmitPeriodic(canChannelObj, cruiseControlCmdMessage, 'On', 0.1);

% Start the first channel.
start(canChannelObj);

% Start the second (logging) channel.
start(canLogChannelObj);
```

```
% Wait 1 second to allow time for some messages to be generated on the bus.
pause(1);

% Stop the channels.
stop(canChannelObj)
stop(canLogChannelObj)

% Retrieve all messages from the logged message bus and output the results as a timetable.
msg = receive(canLogChannelObj, Inf, 'OutputFormat','timetable')
```

```
msg=20×8 timetable
        Time         ID     Extended          Name              Data        Length      Signals
    ____________    ___    ________    _________________    ___________    ______    ____________

    0.077716 sec    256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.07772 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.1777 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.17771 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.27673 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.27674 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.37673 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.37674 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.47773 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.47773 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.57674 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.57674 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.67673 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.67673 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
    0.77771 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.77771 sec     512     false      {'CruiseCtrlFB' }    {1×2 uint8}      2       {1×1 struct
      ⋮
```

```
% Extract only the Cruise Control Command CAN messages.
msgCmd = msg(msg.Name == "CruiseCtrlCmd", :)
```

```
msgCmd=10×8 timetable
        Time         ID     Extended          Name              Data        Length      Signals
    ____________    ___    ________    _________________    ___________    ______    ____________

    0.077716 sec    256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.1777 sec      256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.27673 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.37673 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.47773 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.57674 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.67673 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.77771 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.87776 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
    0.97769 sec     256     false      {'CruiseCtrlCmd'}    {1×2 uint8}      2       {1×1 struct
```

`retrieveLoggedCANMessages` uses the following Vehicle Network Toolbox functions:

- `receive` to retrieve CAN messages from the CAN bus. In this case, the function is configured to retrieve all messages since the previous invocation and output the results as a MATLAB timetable.

Save Messages to a File

When the user clicks the "Save Logged Data" button on the UI, the helper function `saveLoggedCANDataToFile` is called. This function opens a file browser window using the `uinputfile` function. `uinputfile` returns the filename and path selected by the user to store the logged CAN message data. The Vehicle Network Toolbox function `canMessageReplayBlockStruct` is used to convert the CAN messages from a MATLAB timetable into a form that the CAN Replay block can use. Once the logged CAN message data has been converted and saved to a file it can be recalled and replayed later using the Vehicle Network Toolbox "Replay" Simulink block. To replay messages in MATLAB with Vehicle Network Toolbox with the `replay` command, the timetable itself is passed as input to the function.

```
function savedLoggedCANDataToFile(app)
    % Raise dialog box to prompt user for a CAN log file to store the logged data.
    [FileName,PathName] = uiputfile('*.mat','Select a .MAT file to store logged CAN data');

    if FileName ~= 0
        % User did not cancel the file selection operation, so OK to save
        % convert the CAN log data from Timetable to struct of arrays so the data is compatible
        % with the VNT Simulink Replay block.
        canLogStructOfArrays = canMessageReplayBlockStruct(app.canLogMsgBuffer);
        save(fullfile(PathName, FileName), 'canLogStructOfArrays');

        % Clear the buffer after saving it.
        app.canLogMsgBuffer = [];
    end
end
```

`saveLoggedCANDataToFile` uses the following Vehicle Network Toolbox function:

- `canMessageReplayBlockStruct` to convert the CAN messages stored in the MATLAB timetable into a form that can be used by the CAN Message Replay Block.

Load Messages From a File

When the user clicks the "Load Logged Data" button on the UI the helper function `loadLoggedCANDataFromFile` is called. This function opens a file browser window using the `uigetfile` function, which returns the logged message filename selected by the user. The logged CAN message data is loaded from the file and converted back into a timetable representation for use with the Vehicle Network Toolbox `replay` command. Note that the same data file could be used directly with the Vehicle Network Toolbox "Replay" Simulink block if the user desired to replay the data from the Simulink model. You might choose to replay the data using the "Replay" block instead of from the UI using the `replay` command because the "Replay" block works with the Simulink debugger, pausing playback when the simulation halts on a breakpoint. The `replay` command, in contrast, does not recognize when the Simulink model halts on a breakpoint and would simply keep playing the stored message data from the file. For the purposes of the example, we will replay the data using the `replay` command.

```
function loadLoggedCANDataFromFile(app)
    % Raise dialog box to prompt user for a CAN log file to load.
    [FileName,PathName] = uigetfile('*.mat','Select a CAN log file to load');

    % Return focus to main UI after dlg closes.
    figure(app.UIFigure)

    if FileName ~= 0
```

```
            % User did not cancel the file selection operation, so OK to load
            % make sure the message buffer is empty before loading in the logged CAN data.
            app.canLogMsgBuffer = [];

            % Upload the saved message data from the selected file.
            canLogMsgStructOfArrays = load(fullfile(PathName, FileName), 'canLogStructOfArrays');

            % Convert the saved message data into timetables for the replay command.
            app.canLogMsgBuffer = canMessageTimetable(canLogMsgStructOfArrays.canLogStructOfArrays);
        end
    end
```

`loadLoggedCANDataFromFile` uses the following Vehicle Network Toolbox function:

- `canMessageTimetable` to convert the CAN messages stored in a form compatible with the CAN message "Replay" Simulink block back into a MATLAB timetable for use with the `replay` function.

Start Playback of Logged Messages

After the logged CAN message data has been loaded into the UI and reformatted it is ready for playback using the Vehicle Network Toolbox `replay` command. When the user clicks "Start Replay" button on the UI the helper function `startPlaybackOfLoggedCANData` is called. In order to replay the logged CAN message data over the first CAN channel, all activity associated with this channel must be halted and any buffered message data cleared. As shown in the code fragment below, `startPlaybackOfLoggedCANData` turns off periodic transmission of CAN messages, stops the CAN channel, clears any CAN message data buffered in the UI, and clears the plots displaying the signal data fed back from the Cruise Control algorithm model. The CAN channel is then restarted, and the logged CAN message data is replayed.

```
function startPlaybackOfLoggedCANData(app)
    % Turn off periodic transmission of CruiseCtrlCmd CAN message from UI controls.
    transmitPeriodic(app.canChannelObj, app.cruiseControlCmdMessage, 'Off');

    % Stop the UI CAN channel so we can instead use if for playback.
    stopCANChannel(app)

    % Flush the existing CAN messages stored for plotting.
    flushCANFbMsgQueue(app)

    % Clear the existing plots.
    cla(app.tspeedPlot)
    cla(app.engagedPlot)

    % Start the CAN Channel and replay the logged CAN message data.
    startCANChannel(app)

    % Replay the logged CAN data on the UI CAN Channel.
    replay(app.canChannelObj, app.canLogMsgBuffer);
end
```

startPlaybackOfLoggedCANData uses the following Vehicle Network Toolbox functions:

- `transmitPeriodic` to disable periodic transmission of the command signals sent to the Cruise Control algorithm model in the "CruiseCtrlCmd" message.
- `replay` to replay logged CAN message data on the first CAN channel.

Stop Playback of Logged Messages

When the user presses the "Stop Replay" button on the UI the helper function `stopPlaybackOfLoggedCANData` is called. In order to halt the playback of logged CAN message data the CAN channel where the data is being replayed must be stopped. Once that is done the periodic transmission of the "CruiseCtrlCmd" message can be re-enabled and the channel restarted so that the user will once again be able to inject test stimulus signals to the Cruise Control algorithm model interactively from the UI. As shown in the code fragment below, `stopPlaybackOfLoggedCANData` first stops the channel, which halts the replay of the logged message data. Logged message data is cleared from local buffers on the UI as well as the plots displaying the signal data fed back from the Cruise Control algorithm model. Periodic transmission of the "CruiseCtrlCmd" message is re-enabled, and the CAN channel restarted.

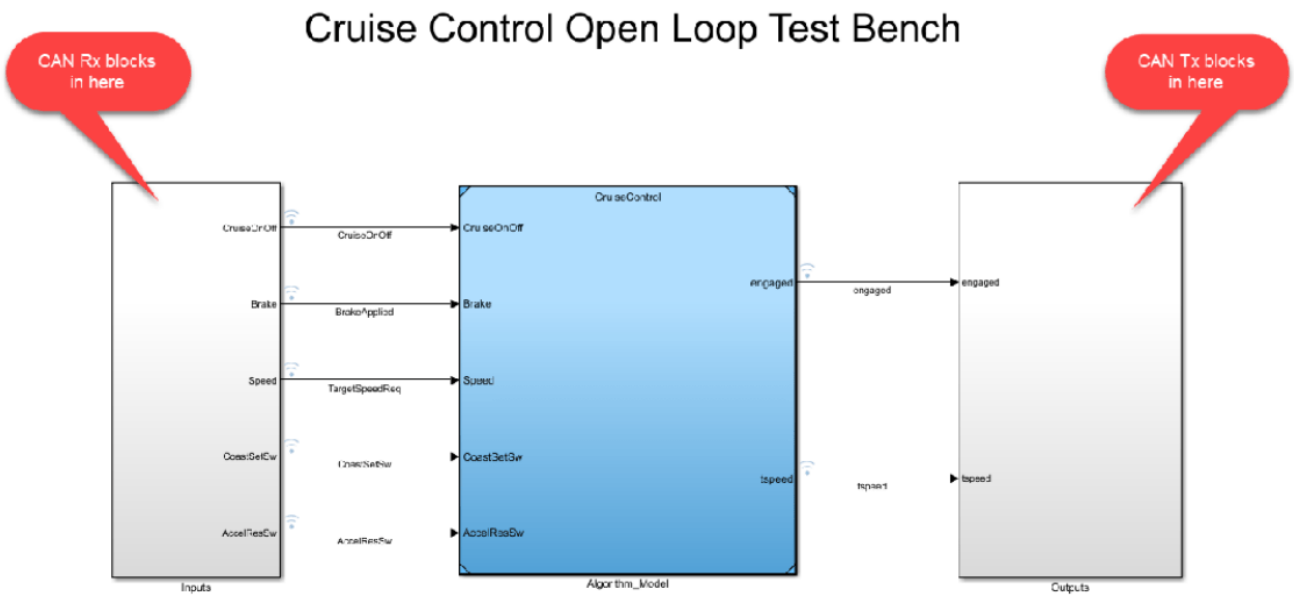
```
function stopPlaybackOfLoggedCANData(app)
    % Stop the playback CAN channel.
    stopCANChannel(app)

    % Flush the existing CAN messages stored for plotting.
    flushCANFbMsgQueue(app)

    % Clear the existing plots.
    cla(app.tspeedPlot)
    cla(app.engagedPlot)

    % Re-enable periodic transmission of CruiseCtrlCmd CAN message from UI controls.
    transmitPeriodic(app.canChannelObj, app.cruiseControlCmdMessage, 'On', 0.1);

    % Restart the CAN Channel from/To UI.
    startCANChannel(app)
end
```

`stopPlaybackOfLoggedCANData` uses the following Vehicle Network Toolbox functions:

- `stop` to stop the CAN channel to halt replay of the logged CAN message data.
- `transmitPeriodic` to re-enable periodic transmission of the command signals sent to the Cruise Control algorithm model in the "CruiseCtrlCmd" message.
- `start` to re-start the CAN channel so the user can once again inject test stimulus signals to the Cruise Control algorithm model interactively from the UI.

Add Virtual CAN Channel Communication to the Simulink Cruise Control Algorithm Model

In this step, we describe how Vehicle Network Toolbox Simulink blocks were used to add virtual CAN communication capability to the Simulink Cruise Control algorithm model.



This description will cover the following topics:

- Adding CAN message receive capability
- Adding CAN message transmit capability
- Pushing CAN channel configuration information from the UI to the Simulink model

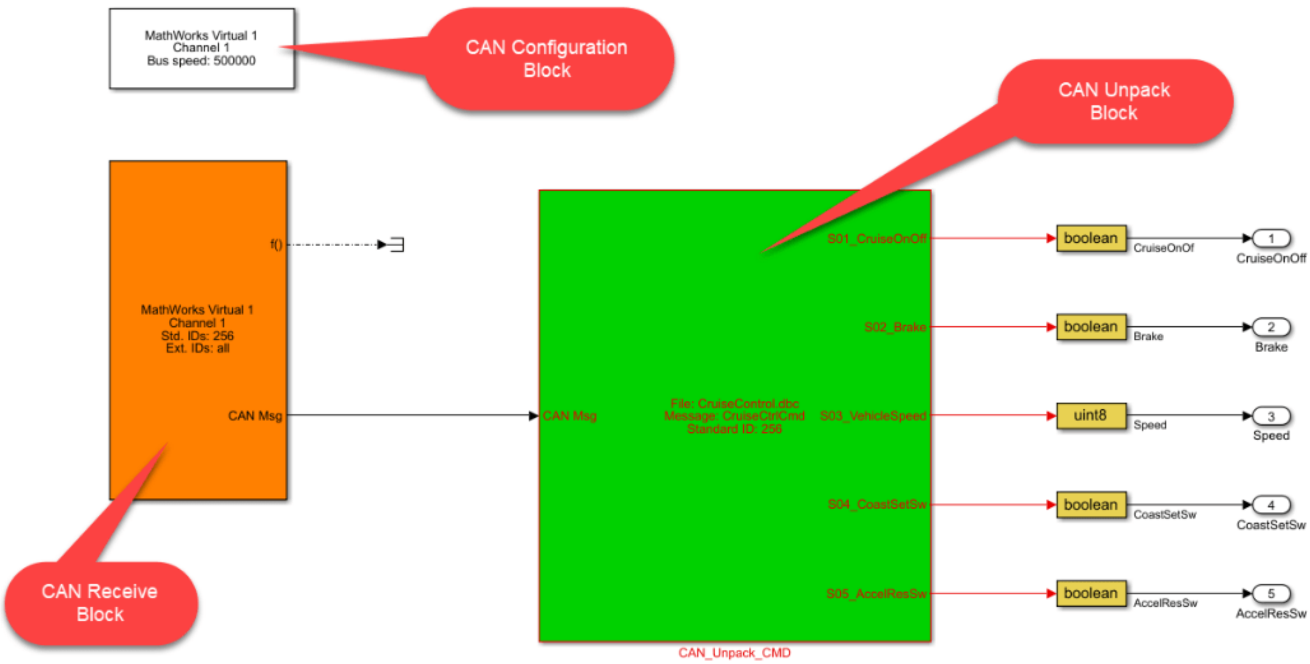
Open the Cruise Control Algorithm Simulink Model

Run the helper functions to configure the workspace with needed data parameters and then open the Cruise Control algorithm test harness model. With the Simulink model open, you can explore the portions of the model explained in the sections below. Execute `helperPrepareTestBenchParameterData` followed by `helperConfigureAndOpenTestBench`.

Add CAN Message Receive Capability

For the Cruise Control algorithm Simulink model to receive CAN data from the test UI requires a block to connect the Simulink model to a specific CAN device, a block to receive CAN messages from the selected device, and a block to unpack the data payload of the messages received into individual signals. To accomplish this, an "Inputs" subsystem is added to the Cruise Control algorithm Simulink model. The "Inputs" subsystem uses Vehicle Network Toolbox CAN Configuration, CAN Receive, and CAN Unpack blocks interconnected as shown in the screen shot below.

The CAN Configuration Block allows the user to determine which of the available CAN devices and channels to connect with the Simulink model. The CAN Receive block receives CAN messages from the CAN device and channel selected in the CAN Configuration block. It also allows the user to receive all messages on the bus or apply a filter to receive only select message(s). The CAN Unpack block has been configured to read a user defined network database (.DBC) file. This allows the user to determine the message name, message ID, and data payload to unpack signals with this block. Simulink input ports are automatically added to the block for each signal defined in the message in the network database file.



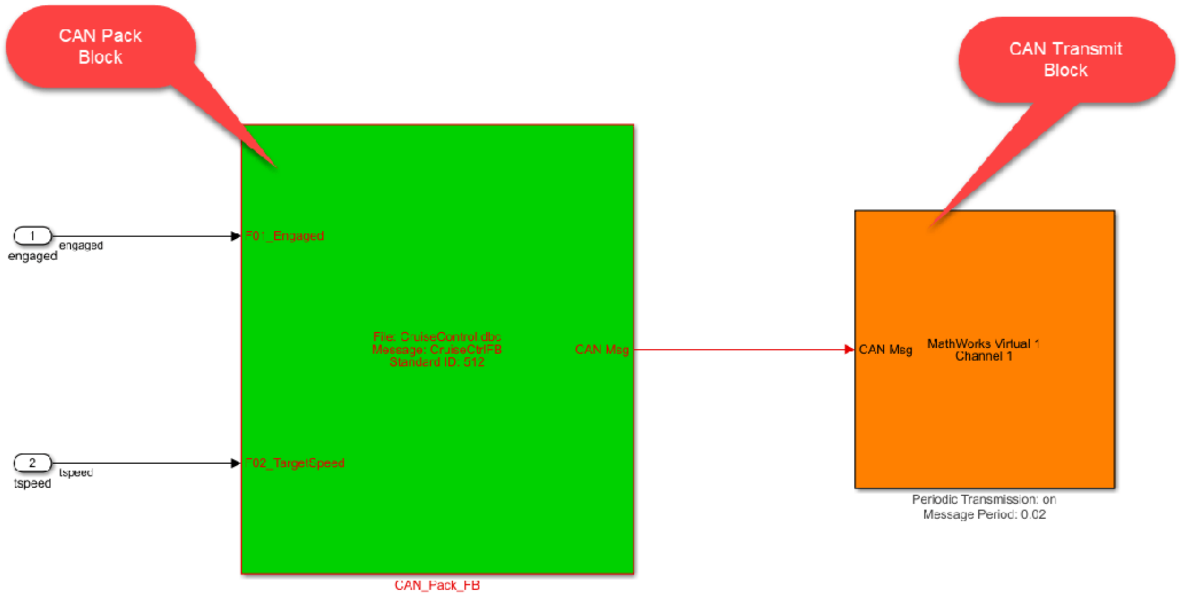
The "Inputs" subsystem of the Cruise Control algorithm Simulink model uses the following Vehicle Network Toolbox Simulink blocks to receive CAN messages:

- CAN Configuration Block to select which CAN channel device to connect to the Simulink model.
- CAN Receive Block to receive the CAN messages from the CAN device selected in the CAN Configuration block.
- CAN Unpack Block to unpack the payload of the received CAN message(s) into individual signals, one for each data item defined in the message.

Add CAN Message Transmit Capability

For the Cruise Control algorithm Simulink model to transmit CAN data to the test UI requires a block to connect the Simulink model to a specific CAN device, a block to pack Simulink signals into the data payload of one or more CAN messages, and a block to transmit the CAN messages from the selected device. To accomplish this, an "Outputs" subsystem is added to the Cruise Control algorithm Simulink model. The "Outputs" subsystem uses Vehicle Network Toolbox CAN pack and CAN Transmit blocks interconnected as shown in the screen shot below.

The CAN Pack block has been configured to read a user defined network database (.DBC) file. This allows the user to determine the message name, message ID, and data payload to pack signal with this block. Simulink output ports are automatically added to the block for each signal defined in the message in the network database file. The CAN Transmit block will transmit the message assembled by the Pack Block on the CAN channel and CAN device selected by the user with the Configuration block. Note that a second CAN Configuration block is not required since the "Outputs" subsystem is transmitting CAN messages on the same CAN channel and device used to receive CAN messages.



The "Outputs" subsystem of the Cruise Control algorithm Simulink model uses the following Vehicle Network Toolbox Simulink blocks to transmit CAN messages:

- CAN Pack Block to unpack the payload of the received CAN message(s) into individual signals, one for each data item defined in the message.
- CAN Transmit Block to receive the CAN messages from the CAN device selected in the CAN Configuration block.

Push CAN Channel Configuration from the UI to the Simulink Model

Because the user selects which of the available CAN devices and channels to use from the UI, this information needs to be sent to the Cruise Control algorithm Simulink model to keep the CAN device and channel configurations between the UI and the Simulink model in sync. In order to accomplish this, the CAN device and channel information used by the Vehicle Network Toolbox "CAN Configuration", "CAN Transmit" and "CAN Receive" blocks need to be configured programmatically from the UI. Every time a user selects a CAN device and CAN channel from the UI "Channel Configuration/Select CAN Channel" menu, the helper function `updateModelWithSelectedCANChannel` is called. As shown in the code fragment below, `updateModelWithSelectedCANChannel` finds the block path for the "CAN Configuration", "CAN Transmit", and "CAN Receive" blocks within the Cruise Control algorithm Simulink model. Using set_param commands, the "Device", "DeviceMenu", and "ObjConstructor" block properties for each of these three blocks are set to the corresponding properties from the CAN device and CAN channel selected by the user.

```
function updateModelWithSelectedCANChannel(app, index)
    % Check to see if we are using a virtual CAN channel and whether the model is loaded.
    if app.canChannelInfo.DeviceModel(index) == "Virtual" && bdIsLoaded(app.mdl)
        % Using a virtual channel.

        % Find path to CAN configuration block.
        canConfigPath = find_system(app.mdl,'Variants', 'AllVariants', 'LookUnderMasks', 'all',...
            'FollowLInks', 'on', 'Name', 'CAN Configuration');
```

```
        % Find path to CAN transmit block.
        canTransmitPath = find_system(app.mdl,'Variants', 'AllVariants', 'LookUnderMasks', 'all',
            'FollowLInks', 'on', 'Name', 'CAN Transmit');

        % Find path to CAN receive block.
        canReceivePath = find_system(app.mdl,'Variants', 'AllVariants', 'LookUnderMasks', 'all',
            'FollowLInks', 'on', 'Name', 'CAN Receive');

        % Push the selected CAN channel into the simulation model CAN Configuration block.
        set_param(canConfigPath{1}, 'Device', app.canChannelDeviceSelected);
        set_param(canConfigPath{1}, 'DeviceMenu', app.canChannelDeviceSelected);
        set_param(canConfigPath{1}, 'ObjConstructor', app.canChannelConstructorSelected);

        % Push the selected CAN channel into the simulation model CAN Receive block.
        set_param(canReceivePath{1}, 'Device', app.canChannelDeviceSelected);
        set_param(canReceivePath{1}, 'DeviceMenu', app.canChannelDeviceSelected);
        set_param(canReceivePath{1}, 'ObjConstructor', app.canChannelConstructorSelected);

        % Push the selected CAN channel into the simulation model CAN Transmit block.
        set_param(canTransmitPath{1}, 'Device', app.canChannelDeviceSelected);
        set_param(canTransmitPath{1}, 'DeviceMenu', app.canChannelDeviceSelected);
        set_param(canTransmitPath{1}, 'ObjConstructor', app.canChannelConstructorSelected);
    end
end
```

Use the UI and Model Together

With both the model and UI open, you can explore interacting with the model in the UI. Press "Start Sim" to put the model and UI online. Experiment with the "Driver Inputs" and "Calibrations" sections of the UI to control the model and actuate the cruise control algorithm. You will see the cruise control engagement and speed values plotted in the UI. You can also use the logging and replay features described previously via the UI controls.

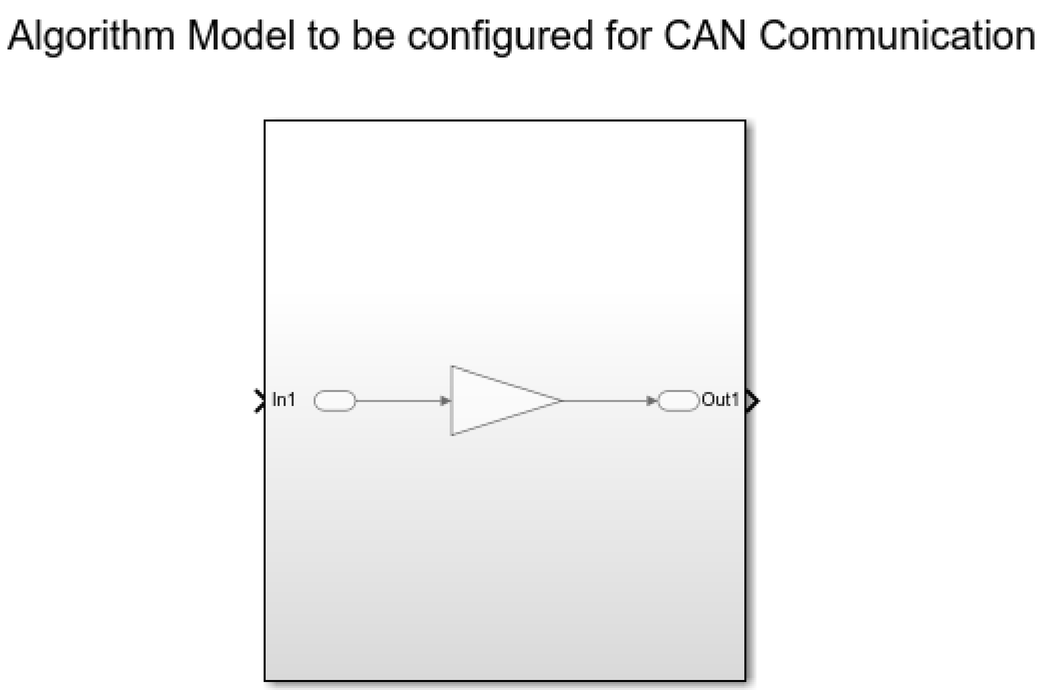
Creating a UI in this manner, gives you a powerful and flexible test interface customizable to your application. It is valuable when debugging and optimizing your algorithm in simulation. By changing the selected CAN device from virtual channels to physical channels, you can continue to use the UI to interact with the algorithm running in a rapid prototyping platform or target controller.

Programmatically Build Simulink Models for CAN Communication

This example shows how to programmatically construct a Simulink model to introduce CAN or CAN FD communication using a CAN DBC-file. With the add_block and set_param functions of Simulink, one can add and fully configure Vehicle Network Toolbox blocks to add network communication to a basic algorithm. The DBC-file contains the CAN messages and signal details. The primary focus is to programmatically configure CAN and CAN FD Pack and Unpack block parameters. This can significantly increase model construction efficiency.

Algorithm Model

The example model `AlgorithmModel.slx` contains a subsystem block called "Algorithm". This block represents any given application algorithm developed in Simulink. A Gain block with value of 2 is inside this subsystem for demonstration purposes. This subsystem has a CAN signal input named "In1". This input value is scaled by the gain value. The scaled value is given as the output of this subsystem named "Out1". For experimentation, the gain value can be changed and the Gain block can be replaced by a different algorithm.



CAN Database File Access

You can access the contents of CAN DBC-files with the `canDatabase` function. Through this function, details about network nodes, messages, and signals are available.

```
db = canDatabase("CANBus.dbc")
```

```
db = 
  Database with properties:
```

```
             Name: 'CANBus'
             Path: 'C:\Users\jpyle\Documents\MATLAB\Examples\vnt-ex60686316\CANBus.dbc'
            Nodes: {'ECU'}
         NodeInfo: [1×1 struct]
         Messages: {2×1 cell}
      MessageInfo: [2×1 struct]
       Attributes: {}
    AttributeInfo: [0×0 struct]
         UserData: []
```

A node "ECU" is defined in the example CAN DBC-file as shown below.

```
node = nodeInfo(db,"ECU")
```

```
node = struct with fields:
             Name: 'ECU'
          Comment: ''
       Attributes: {}
    AttributeInfo: [0×0 struct]
```

The node receives a CAN message "AlgInput" containing a signal "InitialValue". The signal "InitialValue" is the input to the algorithm.

```
messageInfo(db,"AlgInput")
```

```
ans = struct with fields:
             Name: 'AlgInput'
     ProtocolMode: 'CAN'
          Comment: ''
               ID: 100
         Extended: 0
            J1939: []
           Length: 1
              DLC: 1
              BRS: 0
          Signals: {'InitialValue'}
       SignalInfo: [1×1 struct]
          TxNodes: {0×1 cell}
       Attributes: {}
    AttributeInfo: [0×0 struct]
```

The node transmits a CAN message "AlgOutput" containing a signal "ScaledValue". The signal "ScaledValue" is the output of the algorithm.

```
messageInfo(db,"AlgOutput")
```

```
ans = struct with fields:
             Name: 'AlgOutput'
     ProtocolMode: 'CAN'
          Comment: ''
               ID: 200
         Extended: 0
            J1939: []
           Length: 2
              DLC: 2
```

```
            BRS: 0
        Signals: {'ScaledValue'}
     SignalInfo: [1×1 struct]
        TxNodes: {'ECU'}
     Attributes: {}
  AttributeInfo: [0×0 struct]
```

Programmatically Build the Model

Open the Example Model

Open the example model to be configured.

```
open AlgorithmModel
```

Add and Configure CAN Configuration Block

Add and position a CAN Configuration block in the model.

```
add_block("canlib/CAN Configuration","AlgorithmModel/CAN Configuration")
set_param("AlgorithmModel/CAN Configuration","position",[50,330,250,410])
```

Set the "Device" parameter to have the model use the MathWorks virtual CAN device.

```
set_param("AlgorithmModel/CAN Configuration","Device","MathWorks Virtual 1 (Channel 1)")
```

Add and Configure CAN Receive Block

Add and position a CAN Receive block in the model.

```
add_block("canlib/CAN Receive","AlgorithmModel/CAN Receive")
set_param("AlgorithmModel/CAN Receive","position",[50,200,250,280])
```

Add a Terminator block and position it. This is used to connect the function port of the CAN Receive block. In this example, simple message reception is performed. In general, placing a CAN Receive inside a Function-Call Subsystem is the preferred approach to modeling with CAN blocks.

```
add_block("simulink/Sinks/Terminator","AlgorithmModel/Terminator")
set_param("AlgorithmModel/Terminator","position",[310,210,330,230])
```

Set the "Device" parameter to have the model use the MathWorks virtual CAN device.

```
set_param("AlgorithmModel/CAN Receive","Device","MathWorks Virtual 1 (Channel 1)")
```

Add and Configure CAN Unpack Block

Add and position a CAN Unpack block in the model. By default, the block is in "Raw Data" mode.

```
add_block("canlib/CAN Unpack","AlgorithmModel/CAN Unpack")
set_param("AlgorithmModel/CAN Unpack","position",[350,220,600,300])
```

Set the following parameters in the CAN Unpack block in a single function call:

- DataFormat
- CANdbFile
- MsgList

```
set_param("AlgorithmModel/CAN Unpack","DataFormat","CANdb specified signals","CANdbFile",db.Path
```

If the "DataFormat" and "CANdbFile" parameters are already set on a block, the chosen message is changeable by only including the "MsgList" parameter.

Add and Configure CAN Pack Block

Add and position a CAN Pack block in the model.

```
add_block("canlib/CAN Pack","AlgorithmModel/CAN Pack")
set_param("AlgorithmModel/CAN Pack","position",[1000,220,1250,300])
```

Set the following parameters in the CAN Pack block in a single function call:

- DataFormat
- CANdbFile
- MsgList

```
set_param("AlgorithmModel/CAN Pack","DataFormat","CANdb specified signals","CANdbFile",db.Path,"
```

Add and Configure CAN Transmit Block

Add and position a CAN Transmit block in the model.

```
add_block("canlib/CAN Transmit","AlgorithmModel/CAN Transmit")
set_param("AlgorithmModel/CAN Transmit","position",[1350,220,1550,300])
```

Set the "Device" parameter to have the model use the MathWorks virtual CAN device. Also, periodic transmission is enabled with the default timing.

```
set_param("AlgorithmModel/CAN Transmit","Device","MathWorks Virtual 1 (Channel 1)")
set_param("AlgorithmModel/CAN Transmit", "EnablePeriodicTransmit", "on")
```

Make Connections Between the Blocks

The CAN blocks and the algorithm block added in the model must now be connected. The port coordinates for all CAN blocks are required.

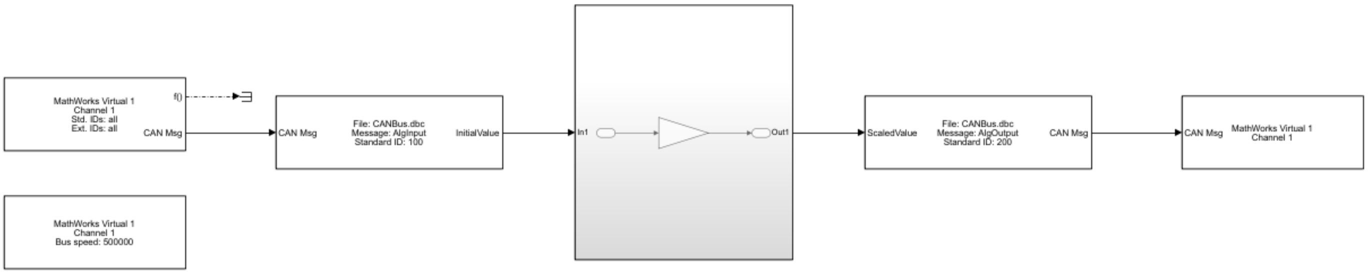
```
canRxPort = get_param("AlgorithmModel/CAN Receive","PortConnectivity");
canUnpackPort = get_param("AlgorithmModel/CAN Unpack","PortConnectivity");
subSystemPort = get_param("AlgorithmModel/Subsystem","PortConnectivity");
canPackPort = get_param("AlgorithmModel/CAN Pack","PortConnectivity");
canTxPort = get_param("AlgorithmModel/CAN Transmit","PortConnectivity");
terminatorPort = get_param("AlgorithmModel/Terminator","PortConnectivity");

[canRxPortFunc,canRxPortMsg] = canRxPort.Position;
[canUnpackPortIn,canUnpackPortOut] = canUnpackPort.Position;
[subSystemPortIn,subSystemPortOut] = subSystemPort.Position;
[canPackPortIn,canPackPortOut] = canPackPort.Position;
canTxPortMsg = canTxPort.Position;
terminatorPortIn = terminatorPort.Position;
```

Add lines to connect all of the blocks in the appropriate order.

```
add_line("AlgorithmModel",[canRxPortMsg ; canUnpackPortIn])
add_line("AlgorithmModel",[canUnpackPortOut ; subSystemPortIn])
add_line("AlgorithmModel",[subSystemPortOut ; canPackPortIn])
add_line("AlgorithmModel",[canPackPortOut ; canTxPortMsg])
add_line("AlgorithmModel",[canRxPortFunc ; terminatorPortIn])
```

Completed Model

This is how the model looks after construction and configuration.



Test the Built Model

Configure a CAN Channel in MATLAB for Communication with the Algorithm Model

Create a CAN channel in MATLAB using channel 2 of the MathWorks virtual CAN device. It will communicate with the CAN channel in the model. Also, attach the CAN database to the MATLAB channel to have it automatically decode incoming CAN data.

```
canCh = canChannel("MathWorks","Virtual 1",2);
canCh.Database = db;
```

For transmission from MATLAB to the model, use the CAN database to prepare a CAN message as input to the algorithm.

```
algInputMsg = canMessage(canCh.Database,"AlgInput");
```

Run the Algorithm Model

Assign the simulation time and start the simulation

```
set_param("AlgorithmModel","StopTime","inf")
set_param("AlgorithmModel","SimulationCommand","start")
```

Pause until the simulation is fully started.

```
while strcmp(get_param("AlgorithmModel","SimulationStatus"),"stopped")
end
```

Run the MATLAB Code

Start the MATLAB CAN channel.

```
start(canCh);
```

Transmit multiple CAN messages with different signal data as input to the model.

```
for value = 1:5
    algInputMsg.Signals.InitialValue = value*value;
    transmit(canCh,algInputMsg)
    pause(1)
end
```

Receive all messages from the bus. Note the instances of the "AlgInput" and "AlgOutput" messages, their timing, and signal values.

```
msg = receive(canCh,Inf,"OutputFormat","timetable")
```

```
msg=10×8 timetable
        Time         ID     Extended        Name            Data        Length      Signals
    ____________    ___    ________    _____________    ___________    ______    ____________

    0.009728 sec    100     false      {'AlgInput' }    {[      1]}      1       {1×1 struct}
    0.15737 sec     200     false      {'AlgOutput'}    {1×2 uint8}      2       {1×1 struct}
    1.0121 sec      100     false      {'AlgInput' }    {[      4]}      1       {1×1 struct}
    1.1574 sec      200     false      {'AlgOutput'}    {1×2 uint8}      2       {1×1 struct}
    2.0146 sec      100     false      {'AlgInput' }    {[      9]}      1       {1×1 struct}
    2.1574 sec      200     false      {'AlgOutput'}    {1×2 uint8}      2       {1×1 struct}
    3.0177 sec      100     false      {'AlgInput' }    {[     16]}      1       {1×1 struct}
    3.1574 sec      200     false      {'AlgOutput'}    {1×2 uint8}      2       {1×1 struct}
    4.0219 sec      100     false      {'AlgInput' }    {[     25]}      1       {1×1 struct}
    4.1574 sec      200     false      {'AlgOutput'}    {1×2 uint8}      2       {1×1 struct}
```

The `canSignalTimetable` function provides an efficient way to separate and organize the signal values of CAN messages into individual timetables for each.

```
signalTimeTable = canSignalTimetable(msg)
```

```
signalTimeTable = struct with fields:
     AlgInput: [5×1 timetable]
    AlgOutput: [5×1 timetable]
```

```
signalTimeTable.AlgInput
```

```
ans=5×1 timetable
        Time        InitialValue
    ____________    ____________

    0.009728 sec          1
    1.0121 sec            4
    2.0146 sec            9
    3.0177 sec           16
    4.0219 sec           25
```

```
signalTimeTable.AlgOutput
```

```
ans=5×1 timetable
       Time        ScaledValue
    ___________    ___________

    0.15737 sec         2
    1.1574 sec          8
    2.1574 sec         18
    3.1574 sec         32
    4.1574 sec         50
```

Stop the CAN channel.

```
stop(canCh)
```

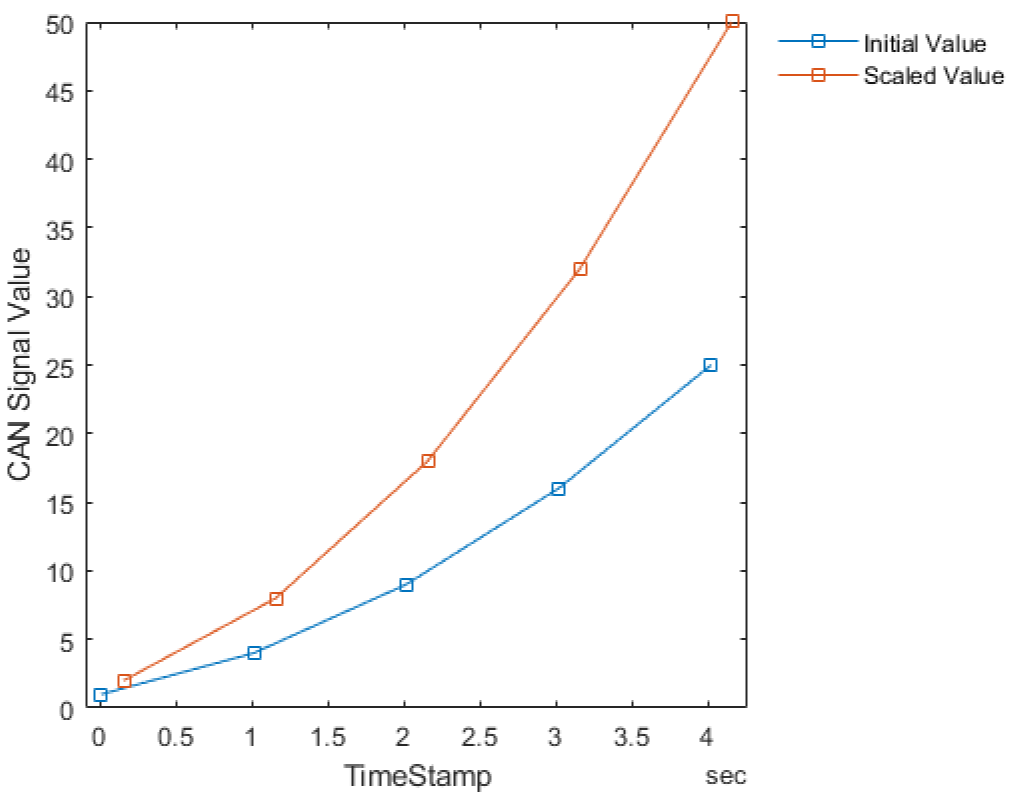
Stop the Algorithm Model

```
set_param("AlgorithmModel","SimulationCommand","stop")
```

Plot the Signal Data

Plot the initial and scaled signal values of the CAN messages against the timestamps as they occurred on the virtual bus. Note the change in values as transmitted by MATLAB and the scaling of the data as performed by the model.

```
plot(signalTimeTable.AlgInput.Time,signalTimeTable.AlgInput.InitialValue,"Marker","square","Marke
hold on
plot(signalTimeTable.AlgOutput.Time,signalTimeTable.AlgOutput.ScaledValue,"Marker","square","Mark
hold off
xlabel("TimeStamp");
ylabel("CAN Signal Value");
legend("Initial Value","Scaled Value","Location","northeastoutside");
legend("boxoff");
```



Class-Based Unit Testing of Automotive Algorithms via CAN

This example shows you how to validate the output of a cruise control algorithm using the Vehicle Network Toolbox and MATLAB class-based unit testing framework.

It uses the MATLAB unit test-class `tCruiseControlAlgorithmVerifier.m` to provide input commands via Controller Area Network (CAN) to a Simulink model of a cruise control algorithm to trigger the functional behavior of the algorithm, and then receives feedback from the model through CAN and validates the expected behavior of the algorithm. It also generates a PDF report of the test results, which can be used for analysis. For more information on how to write the test-class, see the `tCruiseControlAlgorithmVerifier.m` file. The dialog in that class helps you understand the method of setting up a test-class and what each individual test does.

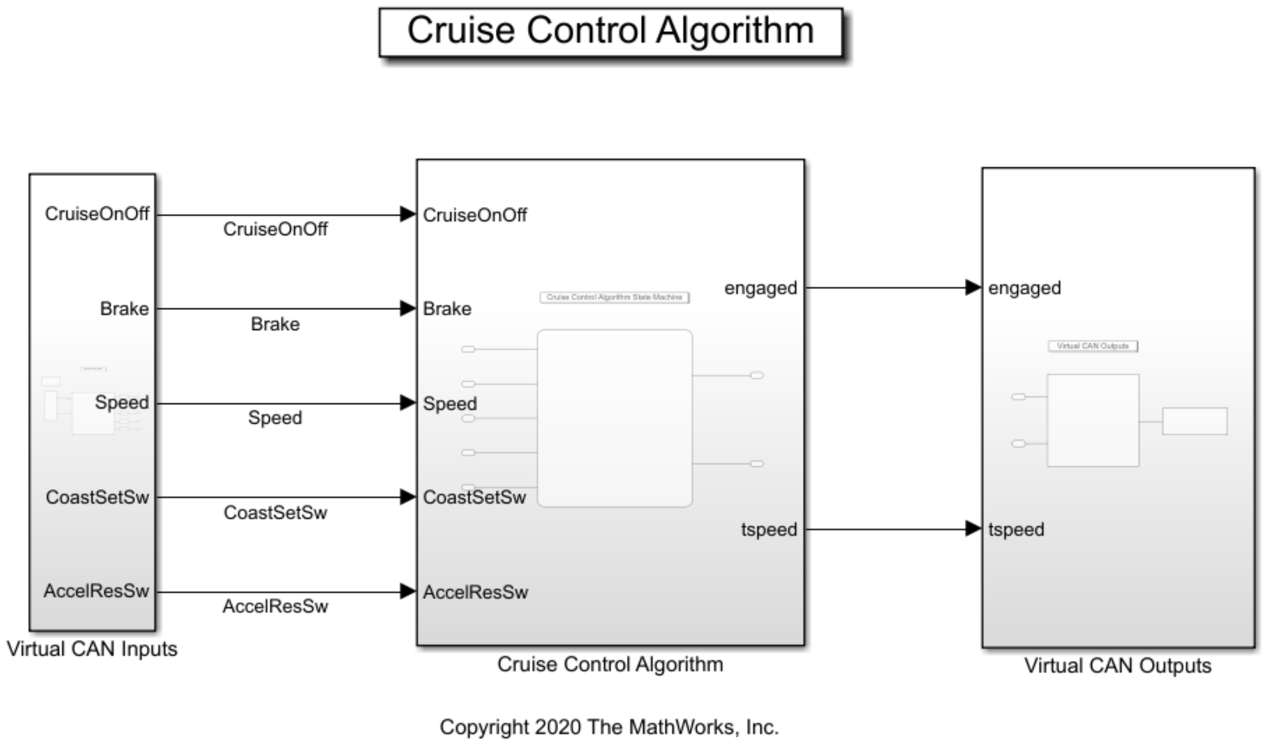
This example uses MathWorks virtual CAN Channels to communicate with the algorithm.

Simulink Model Overview

The cruise control algorithm has a Virtual CAN Inputs block, which houses the setup of the CAN channel using the CAN Configuration block, and receives the message commanded from the MATLAB test-class using the CAN Receive block. It then uses the CAN Unpack block to separate the individual signals from the received CAN message, which are then converted into their appropriate data types and transmitted to the actual cruise control algorithm.

The Cruise Control Algorithm block houses the Cruise Control Algorithm State Machine, which is a Stateflow chart. This algorithm works based on the inputs received from the Virtual CAN Inputs block and is set-up to trigger when the input conditions have reached a certain condition. The outputs of the Stateflow chart are the expected vehicle cruising speed, and algorithm engagement state.

The Virtual CAN Outputs block uses a CAN Pack block to load individual signals into a single CAN message, which are then transmitted onto the CAN bus using the CAN Transmit block. This feedback message is used for verification in the MATLAB test-class.



Create a Test Suite

Create a suite of test classes to run. In this example, the `tCruiseControlAlgorithmVerifier.m` is the only test in the suite. You can add additional tests in the same test suite. The 1xN Test array lists the number of tests, not the number of test-classes.

```
suite = testsuite("tCruiseControlAlgorithmVerifier")
```

```
suite = 
  1x3 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a Test Runner

Create a test runner to execute a set of tests in the test suite. This defines the runner with no special plugins.

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

```
runner = 
  TestRunner with properties:

    ArtifactsRootFolder: "C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23"
       PrebuiltFixtures: [1x0 matlab.unittest.fixtures.Fixture]
```

Create a PDF Report Output

Set-up the name of the PDF file in which you want your output to be captured.

```
pdfFile = "CruiseControlAlgorithmTestReport.pdf"
```

```
pdfFile = 
"CruiseControlAlgorithmTestReport.pdf"
```

Add a PDF creating plugin to your test runner. Firstly, construct the plugin.

```
plugin =  matlab.unittest.plugins.TestReportPlugin.producingPDF(pdfFile)
```

```
plugin = 
  PDFTestReportPlugin with properties:

     IncludeCommandWindowText: 0
    IncludePassingDiagnostics: 0
                 LoggingLevel: Terse
              PageOrientation: 'portrait'
```

Associate this plugin with the test runner to generate the PDF report in the working directory.

```
runner.addPlugin(plugin)
```

Run Tests

Run the test suite using the test runner.

```
result = runner.run(suite)
```

```
Generating test report. Please wait.
    Preparing content for the test report.

    Adding content to the test report.
    Writing test report to file.
Test report has been saved to:
 C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex21299704\CruiseControlAlgorithmTestRep

result = 
  1x3 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
```

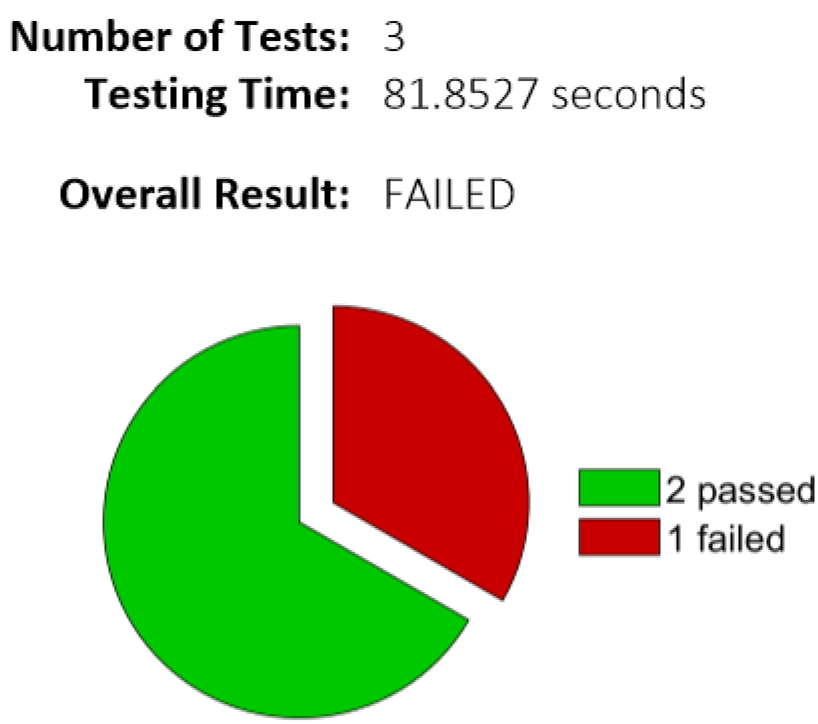
```
2 Passed, 1 Failed, 0 Incomplete.
123.9442 seconds testing time.
```

Analyze the PDF Report

Open the PDF file created with the defined name. The default setting creates it in the current working directory. There are two things to observe in the PDF report:

Overall Result and Result Pie Chart

The overall result states the final outcome based on whether all the tests passed or not. The pie chart shows how many tests passed and failed out of the total number of tests defined.



Failure Summary Details

The failure summary shows which test failed and for what reason.


Failure Summary

1 test failed.

Name of Failing Test	Failure Reasons	
tCruiseControlAlgorithmVerifier/verifyNoCoastingEngagement	Failed by verification.	(Details)

Clicking the details tab on the right hand side of the failure name provides a detailed reason for failure, along with the diagnostic message you configured in the tests.

verifyNoCoastingEngagement

The test failed.
Duration: 38.7588 seconds

Event:

Verification failed.

Test Diagnostic:

```
Actual and Expected Vehicle Speed are not the same
```

Framework Diagnostic:

```
Eventually failed.
--> The constraint never passed with a timeout of 20 second(s).
--> IsTrue failed.
    --> The value must evaluate to "true".

    Actual Value:
      logical

       0

Evaluated Function:
  function_handle with value:

    @()isequal(feedbackMessageSignalTable.F02_TargetSpeed(end,1),testCase.TestCommandMsg.Signals.S03_VehicleSpeed)
```

Event Location: tCruiseControlAlgorithmVerifier/verifyNoCoastingEngagement

Stack:

```
In C:\Users\SDange\Desktop\Demo\tCruiseControlAlgorithmVerifier.m (tCruiseControlAlgorithmVerifier.verifyNoCoastingEngagement) at
 328
```

Decode CAN Data from BLF-Files

This example shows you how to import and decode CAN data from BLF-files in MATLAB for analysis. The BLF-file used in this example was generated from Vector CANoe™ using the "CAN - General System Configuration (CAN)" sample. This example also uses the CAN database file, `PowerTrain_BLF.dbc`, provided with the Vector sample configuration.

Open the DBC-File

Open the database file describing the source CAN network using the `canDatabase` function.

```
canDB = canDatabase("PowerTrain_BLF.dbc")
```

```
canDB = 
  Database with properties:

             Name: 'PowerTrain_BLF'
             Path: 'C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex062025
            Nodes: {2×1 cell}
         NodeInfo: [2×1 struct]
         Messages: {12×1 cell}
      MessageInfo: [12×1 struct]
       Attributes: {11×1 cell}
    AttributeInfo: [11×1 struct]
         UserData: []
```

Investigate the BLF-File

Retrieve and view information about the BLF-File. The `blfinfo` function parses general information about the format and contents of the Vector Binary Logging Format BLF-file and returns the information as a structure.

```
binf = blfinfo("Logging_BLF.blf")
```

```
binf = struct with fields:
                  Name: "Logging_BLF.blf"
                  Path: "C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex0
           Application: "CANoe"
    ApplicationVersion: "11.0.55"
               Objects: 43344
             StartTime: 01-Jul-2020 14:47:34.427
               EndTime: 01-Jul-2020 14:48:33.487
           ChannelList: [2×3 table]
```

```
binf.ChannelList
```

ans=*2×3 table*

ChannelID	Protocol	Objects
1	"CAN"	8801
2	"CAN"	7575

Read Data from BLF-File

The data of interest was logged from the powertrain bus which is stored in channel 2 of the BLF-file. Read the CAN data using the `blfread` function. You can also provide the DBC-file to the function call which will enable message name lookup and signal value decoding.

```
blfData = blfread("Logging_BLF.blf", 2, "Database", canDB)
```

```
blfData=7575×8 timetable
       Time         ID     Extended           Name                         Data                    Le
    ___________    ____    ________    __________________    ______________________________    __

    2.2601 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.2801 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.3002 sec      100     false      {'EngineData'    }    {[      238 2 25 1 0 0 238 2]}
    2.3005 sec      102     false      {'EngineDataIEEE'}    {[       0 128 59 68 0 0 0 0]}
    2.3006 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.3008 sec      201     false      {'ABSdata'       }    {[             0 0 0 0 172 38]}
    2.3009 sec     1020     false      {'GearBoxInfo'   }    {[                          1]}
    2.3201 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.3401 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.3502 sec      100     false      {'EngineData'    }    {[      4 0 25 2 119 1 238 2]}
    2.3505 sec      102     false      {'EngineDataIEEE'}    {[53 127 119 64 0 128 187 67]}
    2.3507 sec      201     false      {'ABSdata'       }    {[              0 0 0 0 35 40]}
    2.3508 sec     1020     false      {'GearBoxInfo'   }    {[                          1]}
    2.3601 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.3801 sec      103     false      {'Ignition_Info' }    {[                        1 0]}
    2.4002 sec      100     false      {'EngineData'    }    {[     10 0 25 3 119 1 238 2]}
      ⋮
```

View signals from an "EngineData" message.

```
blfData.Signals{3}
```

```
ans = struct with fields:
    PetrolLevel: 1
       EngPower: 7.5000
       EngForce: 0
    IdleRunning: 0
        EngTemp: 0
       EngSpeed: 750
```

Repackage and Visualize Signal Values of Interest

Use the `canSignalTimetable` function to repackage signal data from each unique message on the bus into a signal timetable. This example creates three individual signal timetables for the three messages of interest, "ABSdata", "EngineData" and "GearBoxInfo", from the CAN message timetable.

```
signalTimetable1 = canSignalTimetable(blfData, "ABSdata")
```

```
signalTimetable1=1136×4 timetable
       Time        AccelerationForce    Diagnostics    GearLock    CarSpeed
    ___________    _________________    ___________    ________    ________

    2.3008 sec           -100                0            0           0
    2.3507 sec            275                0            0           0
```

```
    2.4008 sec          275                0            0            0
    2.4507 sec          275                0            0            0
    2.5008 sec          275                0            0            0
    2.5507 sec          275                0            0            0
    2.6008 sec          275                0            0            0
    2.6507 sec          275                0            0            0
    2.7008 sec          350                0            0            0
    2.7507 sec          425                0            0          0.5
    2.8008 sec          425                0            0          0.5
    2.8507 sec          500                0            0          0.5
    2.9008 sec          575                0            0          0.5
    2.9507 sec          575                0            0          0.5
    3.0008 sec          650                0            0          0.5
    3.0507 sec          725                0            0          0.5
       ⋮
```

```
signalTimetable2 = canSignalTimetable(blfData, "EngineData")
```

```
signalTimetable2=1136×6 timetable
       Time        PetrolLevel    EngPower    EngForce    IdleRunning    EngTemp    EngSpeed
    __________    ___________    ________    ________    ___________    _______    ________

    2.3002 sec          1           7.5           0           0            0          750
    2.3502 sec          2           7.5         375           0            0            4
    2.4002 sec          3           7.5         375           0            0           10
    2.4502 sec          4           7.5         375           0            0           17
    2.5002 sec          5           7.5         375           0            0           23
    2.5502 sec          6           7.5         375           0            0           30
    2.6002 sec          7           7.5         375           0            0           36
    2.6502 sec          8           7.5         375           0            0           43
    2.7002 sec          9             9         450           0            0           50
    2.7502 sec         10          10.5         525           0            0           59
    2.8002 sec         10          10.5         525           0            0           69
    2.8502 sec         11            12         600           0            0           80
    2.9002 sec         11          13.5         675           0            0           92
    2.9502 sec         12          13.5         675           0            0          106
    3.0002 sec         13            15         750           0            0          121
    3.0502 sec         13          16.5         825           0            0          136
       ⋮
```

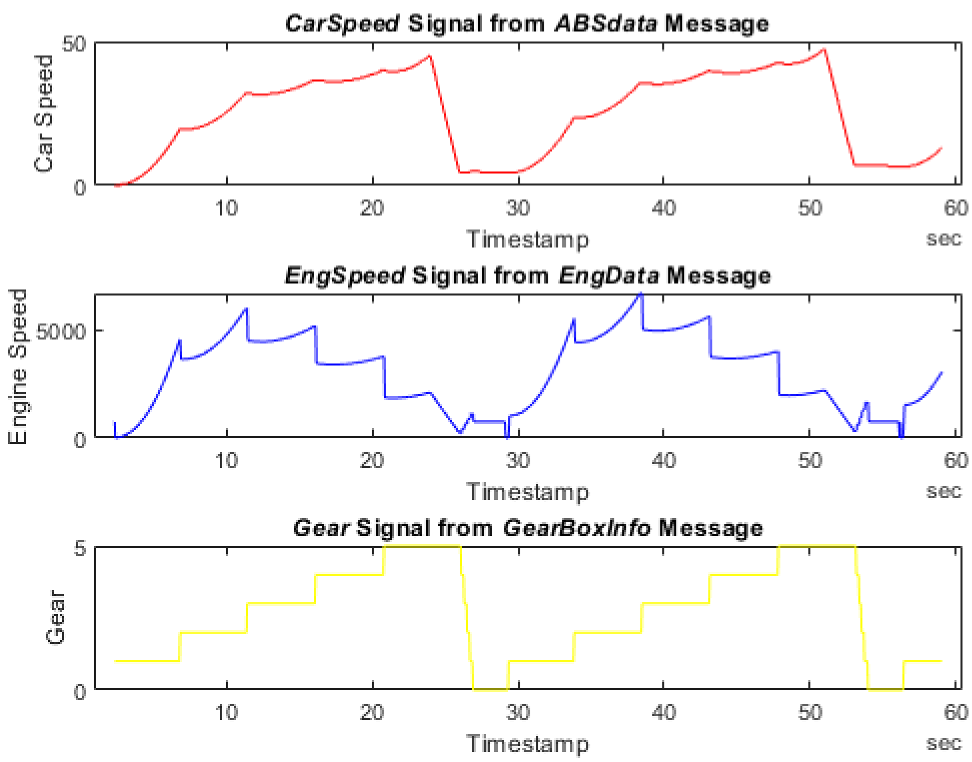
```
signalTimetable3 = canSignalTimetable(blfData, "GearBoxInfo")
```

```
signalTimetable3=1136×3 timetable
       Time        EcoMode    ShiftRequest    Gear
    __________    _______    ____________    ____

    2.3009 sec       0             0           1
    2.3508 sec       0             0           1
    2.4009 sec       0             0           1
    2.4508 sec       0             0           1
    2.5009 sec       0             0           1
    2.5508 sec       0             0           1
    2.6009 sec       0             0           1
    2.6508 sec       0             0           1
    2.7009 sec       0             0           1
    2.7508 sec       0             0           1
    2.8009 sec       0             0           1
```

```
    2.8508 sec        0             0            1
    2.9009 sec        0             0            1
    2.9508 sec        0             0            1
    3.0009 sec        0             0            1
    3.0508 sec        0             0            1
        ⋮
```

To visualize the signals of interest, columns from the signal timetables can be plotted over time for further analysis.

```
subplot(3, 1, 1)
plot(signalTimetable1.Time, signalTimetable1.CarSpeed, "r")
title("{\itCarSpeed} Signal from {\itABSdata} Message", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Car Speed")
subplot(3, 1, 2)
plot(signalTimetable2.Time, signalTimetable2.EngSpeed, "b")
title("{\itEngSpeed} Signal from {\itEngData} Message", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Engine Speed")
subplot(3, 1, 3)
plot(signalTimetable3.Time, signalTimetable3.Gear, "y")
title("{\itGear} Signal from {\itGearBoxInfo} Message", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Gear")
```



Decode CAN Data from MDF-Files

This example shows you how to import and decode CAN data from MDF-files in MATLAB for analysis. The MDF-file used in this example was generated from Vector CANoe™ using the "CAN - General System Configuration (CAN)" sample. This example also uses the CAN database file, `PowerTrain.dbc`, provided with the Vector sample configuration.

Open the MDF-File

Open access to the MDF-file using the `mdf` function.

```
m = mdf("Logging_MDF.mf4")
```

```
m = 
  MDF with properties:

   File Details
                  Name: 'Logging_MDF.mf4'
                  Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex42187575\Loggi
                Author: ''
            Department: ''
               Project: ''
               Subject: ''
               Comment: ''
               Version: '4.10'
              DataSize: 1542223
      InitialTimestamp: 2020-06-25 20:41:13.133000000

   Creator Details
     ProgramIdentifier: 'MDF4Lib'
               Creator: [1x1 struct]

   File Contents
            Attachment: [5x1 struct]
          ChannelNames: {62x1 cell}
          ChannelGroup: [1x62 struct]

   Options
            Conversion: Numeric
```

Identify CAN Data Frames

According to the ASAM MDF associated standard for bus logging, the event types defined for a CAN bus system can be "CAN_DataFrame", "CAN_RemoteFrame", "CAN_ErrorFrame" or "CAN_OverloadFrame". This example focuses on extracting the CAN data frames, so the bus logging standard will be discussed using "CAN_DataFrame" event type as example. Additionally, note that a standard CAN data frame has up to 8 bytes for its payload and is used to transfer signal values.

The standard specifies that the channel names of the event structure should be prefixed by the event type name, for instance, "CAN_DataFrame". Typically a dot is used as separator character to specify the member channels, for instance, "CAN_DataFrame.ID" or "CAN_DataFrame.DataLength".

Use the `channelList` function to filter on channel names exactly matching "CAN_DataFrame". A table with information on matched channels is returned.

```
channelList(m, "CAN_DataFrame", "ExactMatch", true)
```

```
ans=2×9 table
      ChannelName      ChannelGroupNumber    ChannelGroupNumSamples    ChannelGroupAcquisitionNar
    _______________    __________________    ______________________    __________________________

    "CAN_DataFrame"            17                     8889                            CAN1
    "CAN_DataFrame"            29                     7648                            CAN2
```

The powetrain data of interest was logged from the CAN 2 network. The `channelList` output above shows that the data from CAN 2 network has been stored in channel group 29 of the MDF-file. View the channel group details using the `ChannelGroup` property.

```
m.ChannelGroup(29)
```

```
ans = struct with fields:
    AcquisitionName: 'CAN2'
            Comment: ''
         NumSamples: 7648
           DataSize: 206496
             Sorted: 1
            Channel: [14x1 struct]
```

Within a channel group, details about each channel are stored. View details about channel 2 within channel group 29.

```
m.ChannelGroup(29).Channel(2)
```

```
ans = struct with fields:
                  Name: 'CAN_DataFrame.Flags'
           DisplayName: 'Flags'
    ExtendedNamePrefix: 'CAN2'
           Description: 'Combination of bit flags for the message.'
               Comment: 'Combination of bit flags for the message.'
                  Unit: ''
                  Type: FixedLength
              DataType: IntegerUnsignedLittleEndian
               NumBits: 8
         ComponentType: StructureMember
       CompositionType: None
        ConversionType: None
```

Read CAN Data Frames From the MDF-File

Read all data from all channels in channel group 29 into a timetable using the `read` function. The timetable is structured to follow the ASAM MDF standard logging format. Every row represents one raw CAN frame from the bus, while each column represents a channel within the specified channel group. The channels, such as "CAN_DataFrame.Dir", are named to follow the bus logging standard. However, because timetable column names must be valid MATLAB variable names, they may not be identical to the channel names. Most unsupported characters are converted to underscores. Since "." is not supported in a MATLAB variable name, "CAN_DataFrame.Dir" is altered to "CAN_DataFrame_Dir" in the table.

```
canData = read(m, 29, m.ChannelNames{29})
```

```
canData=7648×14 timetable
       Time       CAN_DataFrame_BusChannel    CAN_DataFrame_Flags    CAN_DataFrame_Dir    CAN_Da
```

```
    ___________    ___________________________    _____________________    ___________________    _______

    2.2601 sec              2                            1                        1
    2.2801 sec              2                            1                        1
    2.3002 sec              2                            1                        1
    2.3005 sec              2                            1                        1
    2.3006 sec              2                            1                        1
    2.3008 sec              2                            1                        1
    2.3009 sec              2                            1                        1
    2.3201 sec              2                            1                        1
    2.3401 sec              2                            1                        1
    2.3502 sec              2                            1                        1
    2.3505 sec              2                            1                        1
    2.3507 sec              2                            1                        1
    2.3508 sec              2                            1                        1
    2.3601 sec              2                            1                        1
    2.3801 sec              2                            1                        1
    2.4002 sec              2                            1                        1
        ⋮
```

Decode CAN Messages Using the DBC-File

Open the database file using the `canDatabase` function.

```
canDB = canDatabase("PowerTrain_MDF.dbc")
```

```
canDB = 
  Database with properties:

             Name: 'PowerTrain_MDF'
             Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex42187575\PowerTrai
            Nodes: {2x1 cell}
         NodeInfo: [2x1 struct]
         Messages: {12x1 cell}
      MessageInfo: [12x1 struct]
       Attributes: {11x1 cell}
    AttributeInfo: [11x1 struct]
         UserData: []
```

The `canMessageTimetable` function uses the database to decode the message names and signals. The timetable of ASAM standard logging format data is converted into a Vehicle Network Toolbox™ CAN message timetable.

```
msgTimetable = canMessageTimetable(canData, canDB)
```

```
msgTimetable=7648×8 timetable
       Time        ID     Extended          Name                        Data                 Le
    ___________    ____    ________    __________________    ______________________________    __

    2.2601 sec     103      false      {'Ignition_Info' }    {[                      1 0]}      
    2.2801 sec     103      false      {'Ignition_Info' }    {[                      1 0]}      
    2.3002 sec     100      false      {'EngineData'    }    {[       238 2 25 1 0 0 238 2]}    
    2.3005 sec     102      false      {'EngineDataIEEE'}    {[       0 128 59 68 0 0 0 0]}     
    2.3006 sec     103      false      {'Ignition_Info' }    {[                      1 0]}      
    2.3008 sec     201      false      {'ABSdata'       }    {[           0 0 0 0 172 38]}      
    2.3009 sec    1020      false      {'GearBoxInfo'   }    {[                        1]}      
```

```
    2.3201 sec     103     false      {'Ignition_Info' }    {[                       1 0]}       2
    2.3401 sec     103     false      {'Ignition_Info' }    {[                       1 0]}       2
    2.3502 sec     100     false      {'EngineData'    }    {[      4 0 25 2 119 1 238 2]}       8
    2.3505 sec     102     false      {'EngineDataIEEE'}    {[53 127 119 64 0 128 187 67]}       8
    2.3507 sec     201     false      {'ABSdata'       }    {[             0 0 0 0 35 40]}       6
    2.3508 sec    1020     false      {'GearBoxInfo'   }    {[                         1]}       1
    2.3601 sec     103     false      {'Ignition_Info' }    {[                       1 0]}       2
    2.3801 sec     103     false      {'Ignition_Info' }    {[                       1 0]}       2
    2.4002 sec     100     false      {'EngineData'    }    {[     10 0 25 3 119 1 238 2]}       8
      ⋮
```

View the signals stored in the "EngineData" message.

```
msgTimetable.Signals{3}
```

```
ans = struct with fields:
    PetrolLevel: 1
       EngPower: 7.5000
       EngForce: 0
    IdleRunning: 0
        EngTemp: 0
       EngSpeed: 750
```

Repackage and Visualize Signal Values of Interest

Use the `canSignalTimetable` function to repackage signal data from each unique message on the bus into a signal timetable. This example creates three individual signal timetables for the three messages of interest, "ABSdata", "EngineData" and "GearBoxInfo", from the CAN message timetable.

```
signalTimetable1 = canSignalTimetable(msgTimetable, "ABSdata")
```

```
signalTimetable1=1147×4 timetable
       Time        AccelerationForce    Diagnostics    GearLock    CarSpeed
    __________    _________________    ___________    ________    ________

    2.3008 sec          -100                0            0            0
    2.3507 sec           275                0            0            0
    2.4008 sec           275                0            0            0
    2.4507 sec           275                0            0            0
    2.5008 sec           275                0            0            0
    2.5507 sec           275                0            0            0
    2.6008 sec           275                0            0            0
    2.6507 sec           275                0            0            0
    2.7008 sec           350                0            0            0
    2.7507 sec           425                0            0          0.5
    2.8008 sec           425                0            0          0.5
    2.8507 sec           500                0            0          0.5
    2.9008 sec           575                0            0          0.5
    2.9507 sec           575                0            0          0.5
    3.0008 sec           650                0            0          0.5
    3.0507 sec           725                0            0          0.5
      ⋮
```

```
signalTimetable2 = canSignalTimetable(msgTimetable, "EngineData")
```

```
signalTimetable2=1147×6 timetable
       Time        PetrolLevel    EngPower    EngForce    IdleRunning    EngTemp    EngSpeed
```

```
    ___________    ____________    _________    _________    ____________    ________    _________

    2.3002 sec          1             7.5            0            0               0          750
    2.3502 sec          2             7.5          375            0               0            4
    2.4002 sec          3             7.5          375            0               0           10
    2.4502 sec          4             7.5          375            0               0           17
    2.5002 sec          5             7.5          375            0               0           23
    2.5502 sec          6             7.5          375            0               0           30
    2.6002 sec          7             7.5          375            0               0           36
    2.6502 sec          8             7.5          375            0               0           43
    2.7002 sec          9               9          450            0               0           50
    2.7502 sec         10            10.5          525            0               0           59
    2.8002 sec         10            10.5          525            0               0           69
    2.8502 sec         11              12          600            0               0           80
    2.9002 sec         11            13.5          675            0               0           92
    2.9502 sec         12            13.5          675            0               0          106
    3.0002 sec         13              15          750            0               0          121
    3.0502 sec         13            16.5          825            0               0          136
       ⋮
```

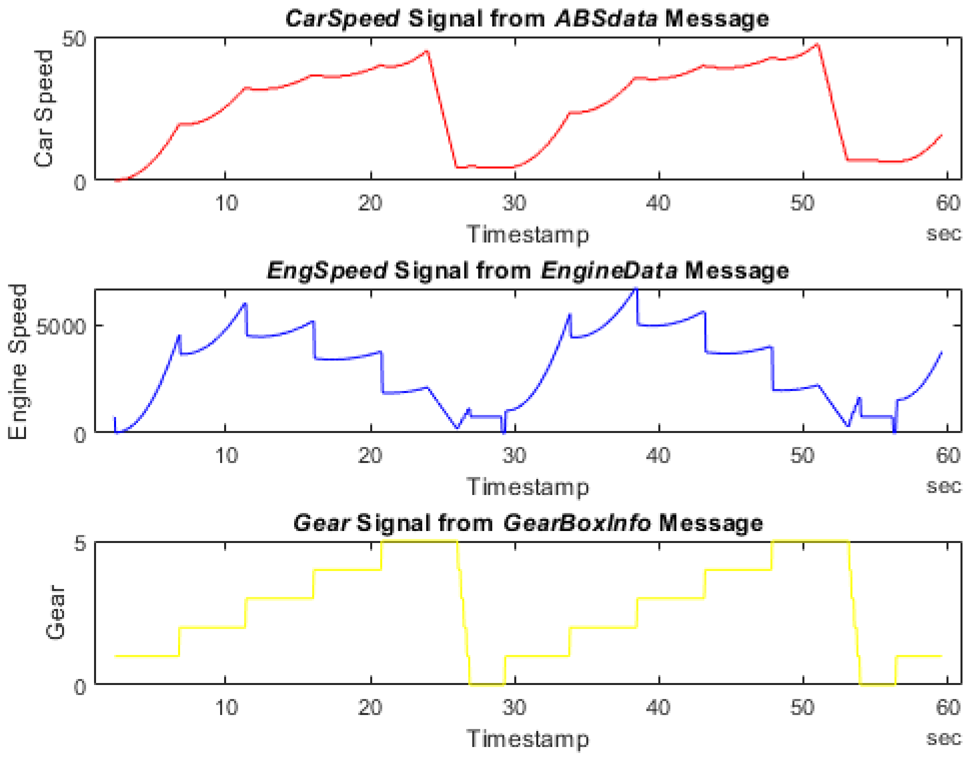
```
signalTimetable3 = canSignalTimetable(msgTimetable, "GearBoxInfo")
```

```
signalTimetable3=1147×3 timetable
       Time        EcoMode    ShiftRequest    Gear
    ___________    _______    ____________    ____

    2.3009 sec        0            0           1
    2.3508 sec        0            0           1
    2.4009 sec        0            0           1
    2.4508 sec        0            0           1
    2.5009 sec        0            0           1
    2.5508 sec        0            0           1
    2.6009 sec        0            0           1
    2.6508 sec        0            0           1
    2.7009 sec        0            0           1
    2.7508 sec        0            0           1
    2.8009 sec        0            0           1
    2.8508 sec        0            0           1
    2.9009 sec        0            0           1
    2.9508 sec        0            0           1
    3.0009 sec        0            0           1
    3.0508 sec        0            0           1
       ⋮
```

To visualize the signals of interest, columns from the signal timetables can be plotted over time for further analysis.

```
subplot(3, 1, 1)
plot(signalTimetable1.Time, signalTimetable1.CarSpeed, "r")
title("{\itCarSpeed} Signal from {\itABSdata} Message", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Car Speed")
subplot(3, 1, 2)
plot(signalTimetable2.Time, signalTimetable2.EngSpeed, "b")
title("{\itEngSpeed} Signal from {\itEngineData} Message", "FontWeight", "bold")
xlabel("Timestamp")
```

```
ylabel("Engine Speed")
subplot(3, 1, 3)
plot(signalTimetable3.Time, signalTimetable3.Gear, "y")
title("{\itGear} Signal from {\itGearBoxInfo} Message", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Gear")
```



Close the Files

Close access to the MDF-file and the DBC-file by clearing their variables from the workspace.

```
clear m
clear canDB
```

Read Data from MDF-Files with Applied Conversion Rules

This example shows you how to read channel data applying conversion rules from an MDF-file and configure different reading options in MATLAB.

Introduction to ASAM MDF Conversion Rules

According to the ASAM MDF standard, a data value encoded in the MDF channel is denoted as a raw value. It can be converted to a physical, engineering unit value using a conversion rule that describes the data. Conversion rules are the methods defined at the channel level to convert raw values to physical values.

ASAM MDF V4.2.0 supports the following conversion rules:

No Conversion

- CC_Type 0: Identity (“1:1”) conversion

Numeric to Numeric Conversions

- CC_Type 1: Linear conversion
- CC_Type 2: Rational conversion formula
- CC_Type 3: Algebraic conversion
- CC_Type 4: Value to value tabular look-up with interpolation
- CC_Type 5: Value to value tabular look-up without interpolation
- CC_Type 6: Value range to value tabular look-up

Numeric to Text Conversions

- CC_Type 7: Value to text/scale conversion tabular look-up
- CC_Type 8: Value range to text/scale conversion tabular look-up

Text to Numeric Conversion

- CC_Type 9: Text to value tabular look-up

Text to Text Conversion

- CC_Type 10: Text to text tabular look-up

Other Conversion

- CC_Type 11: Bitfield text table

Vehicle Network Toolbox™ provides the functionality to read your desired data from the MDF-file with different `Conversion` options. The allowed options are:

- `Numeric` — Apply only numeric to numeric conversions (CC_Type 1-6). Data with other conversion rules are read as raw values.
- `None` — Do not apply any conversion. All data are read as raw values.
- `All` — Apply all numeric and text conversions (CC_Type 1-10). All data are read as physical values.

Note that if there is an identity conversion (CC_Type 0), or a none conversion (no conversion rule) in the channel, the data are read as raw values regardless of which `Conversion` option is specified.

Open the MDF-File

Open access to an MDF-file using the `mdf` function. The object `mdfObj` has the property `Conversion` with the default value `Numeric`.

```
mdfObj = mdf("MDF_Conversion_Example.mf4")
```

```
mdfObj = 
  MDF with properties:

   File Details
                 Name: 'MDF_Conversion_Example.mf4'
                 Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex96016136\MDF_Co
               Author: ''
           Department: ''
              Project: ''
              Subject: ''
              Comment: ''
              Version: '4.10'
             DataSize: 185
     InitialTimestamp: 1980-01-01 05:00:00.000000000

   Creator Details
    ProgramIdentifier: 'amdf5206'
              Creator: [1x1 struct]

   File Contents
           Attachment: [0x1 struct]
         ChannelNames: {{6x1 cell}}
         ChannelGroup: [1x1 struct]

   Options
           Conversion: Numeric
```

Use the `channelList` function to view the list of channels available in `mdfObj`.

```
channelList(mdfObj)
```

```
ans=6×9 table
             ChannelName              ChannelGroupNumber    ChannelGroupNumSamples    ChannelGrou
    ______________________________    __________________    ______________________    ___________

    "Ambient temperature"                     1                       5                 Signal wi
    "Engine temperature"                      1                       5                 Signal wi
    "Fault code"                              1                       5                 Signal wi
    "Gear position"                           1                       5                 Signal wi
    "time"                                    1                       5                 Signal wi
    "Windshield wiper speed level"            1                       5                 Signal wi
```

`Conversion` Property and `Conversion` Name-Value Pair

You can choose a `Conversion` option to apply when reading data from an MDF-file in MATLAB. You specify the option in either of the following ways:

- Set the `Conversion` property of the MDF object and call the `read` function.
- Specify a `Conversion` name-value pair when calling the `read` function.

View the details about the channel `Engine temperature` in channel group 1. The output shows that it has `Linear` conversion (CC_Type 1).

```
mdfObj.ChannelGroup(1).Channel(2)
```

```
ans = struct with fields:
                  Name: 'Engine temperature'
           DisplayName: ''
    ExtendedNamePrefix: ''
           Description: ''
               Comment: ''
                  Unit: '°C'
                  Type: FixedLength
              DataType: IntegerSignedLittleEndian
               NumBits: 32
         ComponentType: None
       CompositionType: None
        ConversionType: Linear
```

You can set the `Conversion` property of the `mdfObj` to be `Numeric` and read data from channel `Engine temperature` in channel group 1.

```
mdfObj.Conversion = "Numeric"
```

```
mdfObj = 
  MDF with properties:

   File Details
                 Name: 'MDF_Conversion_Example.mf4'
                 Path: 'C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\23\tp1aa883b7\vnt-ex96016136\MDF_Co
               Author: ''
           Department: ''
              Project: ''
              Subject: ''
              Comment: ''
              Version: '4.10'
             DataSize: 185
     InitialTimestamp: 1980-01-01 05:00:00.000000000

   Creator Details
    ProgramIdentifier: 'amdf5206'
              Creator: [1x1 struct]

   File Contents
           Attachment: [0x1 struct]
         ChannelNames: {{6x1 cell}}
         ChannelGroup: [1x1 struct]

   Options
           Conversion: Numeric
```

```
dataPropNum = read(mdfObj, 1, "Engine temperature")
```

```
dataPropNum=5×1 timetable
      Time      EngineTemperature
    ________    _________________

    0 sec                35
    0.25 sec         35.556
    0.5 sec          36.111
    0.75 sec         36.667
    1 sec            37.222
```

You can also read data with the name-value pair `Conversion, None`. Note that the `Conversion` name-value pair has a higher priority than the `Conversion` property, which means the name-value pair is applied when the values are different.

```
dataNameValueNone = read(mdfObj, 1, "Engine temperature", "Conversion", "None")
```

```
dataNameValueNone=5×1 timetable
      Time      EngineTemperature
    ________    _________________

    0 sec              95
    0.25 sec           96
    0.5 sec            97
    0.75 sec           98
    1 sec              99
```

Using the `Conversion` name-value pair does not change the `Conversion` property of `mdfObj`. The `Conversion` property value of `mdfObj` after reading data with a name-value pair is still `Numeric`.

```
mdfObj.Conversion
```

```
ans = 
  Conversion enumeration

    Numeric
```

Read Data with Different `Conversion` Options on Numeric to Numeric Conversions

The following code shows how to read your desired data from a channel with a numeric to numeric conversion. The channel `Engine temperature` has `Linear` conversion (CC_Type 1) and it is chosen to represent the reading behavior of numeric to numeric conversions (CC_Type 1-6).

Read data with the name-value pair `Conversion`, `Numeric`. In this case, `Linear` conversion is applied when reading data because the `Numeric` option supports numeric to numeric conversions. The physical data are returned and the physical numeric data have data type `double`.

```
dataLinearNum = read(mdfObj, 1, "Engine temperature", "Conversion", "Numeric")
```

```
dataLinearNum=5×1 timetable
      Time      EngineTemperature
    ________    _________________

    0 sec                35
    0.25 sec         35.556
    0.5 sec          36.111
```

```
    0.75 sec         36.667
    1 sec            37.222
```

```
class(dataLinearNum.EngineTemperature)
```

```
ans = 
'double'
```

Read data with the name-value pair `Conversion`, `None`. In this case, `Linear` conversion is not applied when reading data because the `None` option does not apply any conversion. Raw data are returned with the original data type, which is an `int32`.

```
dataLinearNone = read(mdfObj, 1, "Engine temperature", "Conversion", "None")
```

```
dataLinearNone=5×1 timetable
      Time      EngineTemperature
    ________    _________________

    0 sec               95       
    0.25 sec            96       
    0.5 sec             97       
    0.75 sec            98       
    1 sec               99       
```

```
class(dataLinearNone.EngineTemperature)
```

```
ans = 
'int32'
```

Read data with the name-value pair `Conversion`, `All`. In this case, `Linear` conversion is applied when reading data because the `All` option supports all numeric and text conversions. The physical data are returned and the physical numeric data have data type `double`.

```
dataLinearAll = read(mdfObj, 1, "Engine temperature", "Conversion", "All")
```

```
dataLinearAll=5×1 timetable
      Time      EngineTemperature
    ________    _________________

    0 sec                 35     
    0.25 sec          35.556     
    0.5 sec           36.111     
    0.75 sec          36.667     
    1 sec             37.222     
```

```
class(dataLinearAll.EngineTemperature)
```

```
ans = 
'double'
```

Read Data with Different `Conversion` Options on Numeric to Text Conversions

The following code shows how to read your desired data from a channel with a numeric to text conversion. The channel `Gear position` has `ValueToText` conversion (CC_Type 7) and it is chosen to represent the reading behavior of numeric to text conversions (CC_Type 7-8).

View the details about the channel `Gear position` in channel group 1. The output shows that it has `ValueToText` conversion.

```
mdfObj.ChannelGroup(1).Channel(3)
```

```
ans = struct with fields:
                  Name: 'Gear position'
           DisplayName: ''
    ExtendedNamePrefix: ''
           Description: ''
               Comment: ''
                  Unit: ''
                  Type: FixedLength
              DataType: IntegerUnsignedLittleEndian
               NumBits: 8
         ComponentType: None
       CompositionType: None
        ConversionType: ValueToText
```

Read data with the name-value pair `Conversion`, `Numeric`. In this case, `ValueToText` conversion is not applied when reading data because the `Numeric` option supports only numeric to numeric conversions. The raw data are returned with the original data type, which is an `int8`.

```
dataV2TNum = read(mdfObj, 1, "Gear position", "Conversion", "Numeric")
```

```
dataV2TNum=5×1 timetable
      Time      GearPosition
    ________    ____________

    0 sec            2
    0.25 sec         3
    0.5 sec          0
    0.75 sec         2
    1 sec            1
```

```
class(dataV2TNum.GearPosition)
```

```
ans = 
'uint8'
```

Read data with the name-value pair `Conversion`, `None`. In this case, `ValueToText` conversion is not applied when reading data because the `None` option does not apply any conversion. Raw data are returned with the original data type, which is an `int8`.

```
dataV2TNone = read(mdfObj, 1, "Gear position", "Conversion", "None")
```

```
dataV2TNone=5×1 timetable
      Time      GearPosition
    ________    ____________

    0 sec            2
    0.25 sec         3
    0.5 sec          0
    0.75 sec         2
    1 sec            1
```

```
class(dataV2TNone.GearPosition)
```

```
ans = 
'uint8'
```

Read data with the name-value pair `Conversion`, `All`. In this case, `ValueToText` conversion is applied when reading data because the `All` option supports all numeric and text conversions. The physical data are returned and the physical text data have data type `char`.

```
dataV2TAll = read(mdfObj, 1, "Gear position", "Conversion", "All")
```

```
dataV2TAll=5×1 timetable
      Time         GearPosition    
    ________    ___________________

    0 sec       {'Gear position 2'}
    0.25 sec    {'Gear position 3'}
    0.5 sec     {'Invalid'        }
    0.75 sec    {'Gear position 2'}
    1 sec       {'Gear position 1'}
```

```
class(dataV2TAll.GearPosition{1})
```

```
ans = 
'char'
```

Other Conversion Examples

There are some other channels in the `mdfObj`. The channels `Ambient temperature`, `Windshield wiper speed level`, and `Fault code` have conversions `None`, `TextToValue`, and `TextToText`, respectively. You can try to read these channels with different `Conversion` options.

View the details about the channel `Ambient temperature` in channel group 1. The output shows that it has `None` conversion.

```
mdfObj.ChannelGroup(1).Channel(1)
```

```
ans = struct with fields:
                  Name: 'Ambient temperature'
           DisplayName: ''
    ExtendedNamePrefix: ''
           Description: ''
               Comment: ''
                  Unit: '°F'
                  Type: FixedLength
              DataType: RealLittleEndian
               NumBits: 64
         ComponentType: None
       CompositionType: None
        ConversionType: None
```

View the details about the channel `Windshield wiper speed level` in channel group 1. The output shows that it has `TextToValue` conversion.

```
mdfObj.ChannelGroup(1).Channel(4)
```

```
ans = struct with fields:
                  Name: 'Windshield wiper speed level'
```

```
          DisplayName: ''
    ExtendedNamePrefix: ''
          Description: ''
              Comment: ''
                 Unit: ''
                 Type: VariableLength
             DataType: StringUTF8
              NumBits: 64
        ComponentType: None
      CompositionType: None
       ConversionType: TextToValue
```

View the details about the channel `Fault code` in channel group 1. The output shows that it has `TextToText` conversion.

```
mdfObj.ChannelGroup(1).Channel(5)
```

```
ans = struct with fields:
                 Name: 'Fault code'
          DisplayName: ''
    ExtendedNamePrefix: ''
          Description: ''
              Comment: ''
                 Unit: ''
                 Type: VariableLength
             DataType: StringUTF8
              NumBits: 64
        ComponentType: None
      CompositionType: None
       ConversionType: TextToText
```

Close the File

Close access to the MDF-file by clearing the variable from the workspace.

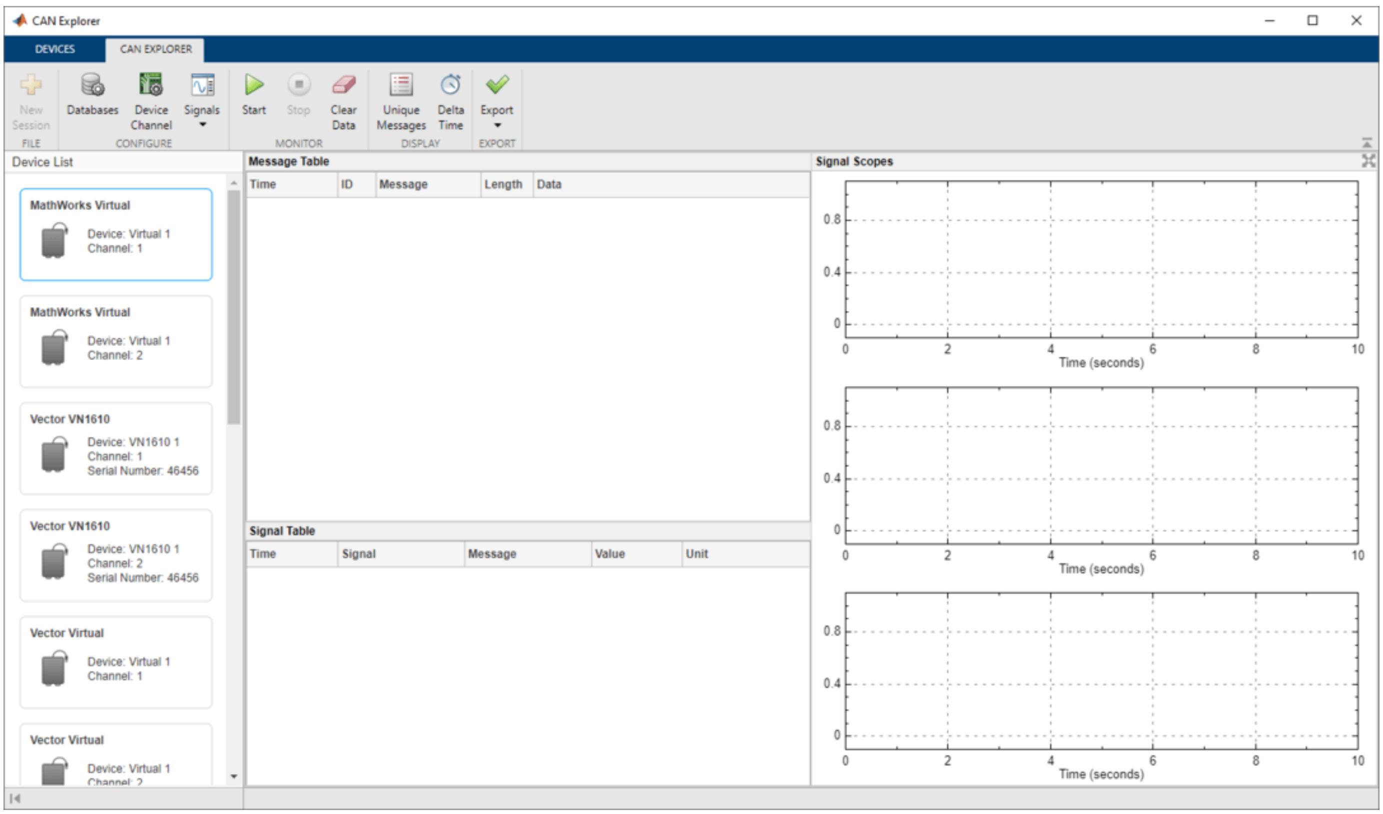
```
clear mdfObj
```

Receive and Visualize CAN Data Using CAN Explorer

This example shows how to use the **CAN Explorer** app to receive and visualize CAN data. It uses MathWorks® Virtual channels which are connected in a loopback configuration. **CAN Explorer** is configured to receive data using MathWorks Virtual 1 Channel 1. Pre-recorded data is provided in a MAT-file and replayed onto MathWorks Virtual 1 Channel 2 to emulate CAN traffic generated from connecting to an actual vehicle system.

Open the CAN Explorer

Open the **CAN Explorer** app using command `canExplorer`. Alternatively, you can find **CAN Explorer** in the MATLAB® **Apps** tab.



Select the Device Channel

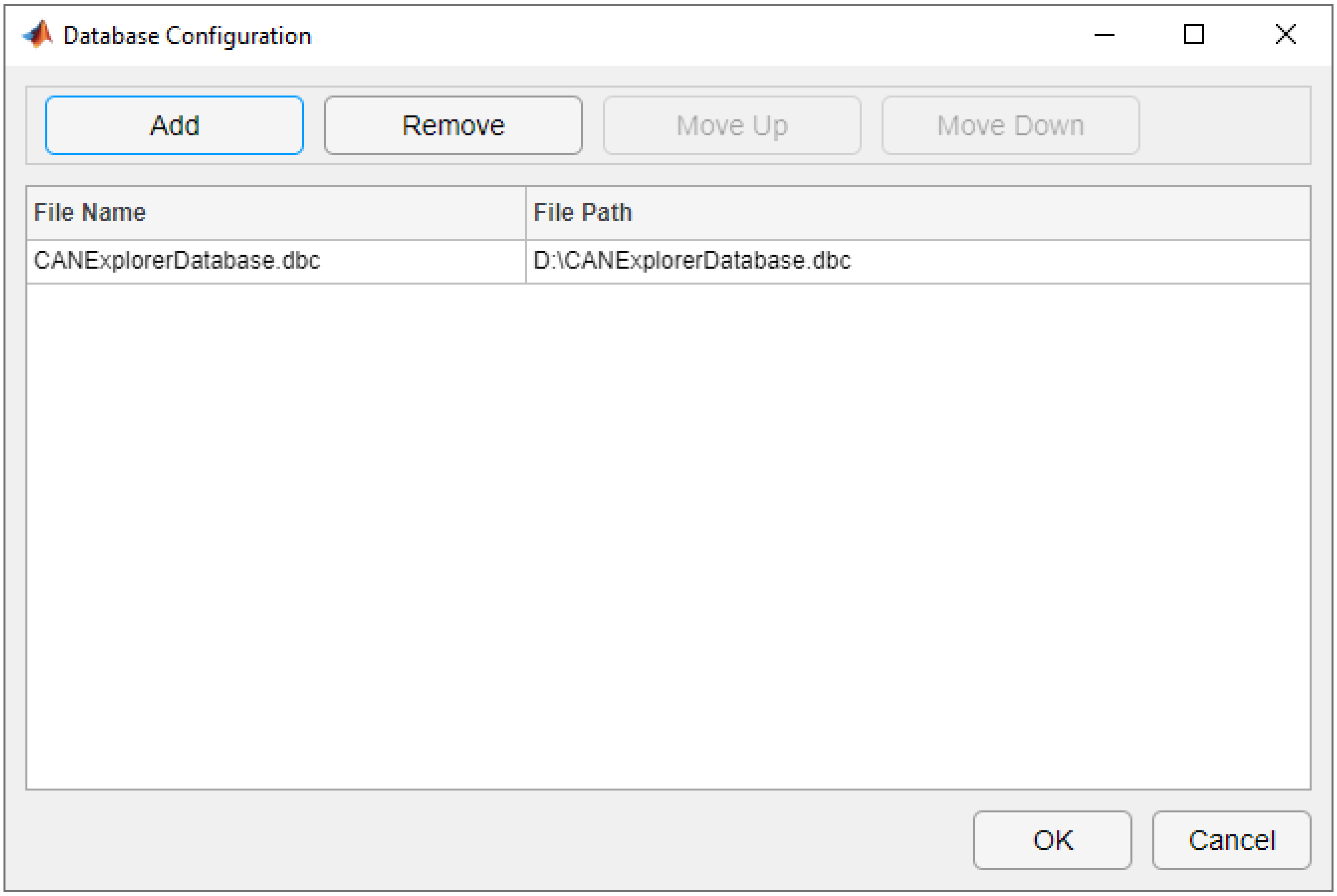
The **Device List** shows all the accessible CAN channels from devices connected to the system, and the current device channel in use is highlighted by a blue outline. Each time you start **CAN Explorer**, the first device channel in the list is automatically selected by default. Select MathWorks Virtual 1 Channel 1 from the **Device List** if it is not selected by default.

Configure the Database Files

Add database files to **CAN Explorer** to decode incoming messages and signals.

1. To open the Database Configuration dialog, select **Databases** in the toolstrip.
2. Click **Add** to open the file selection dialog. Select the `CANExplorerDatabase.dbc` file provided with the example.

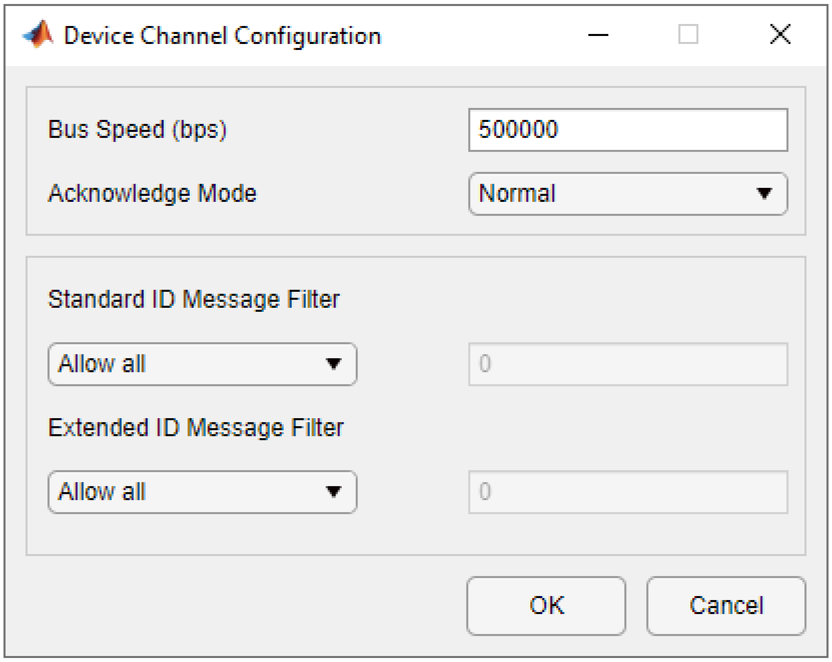
3 Click **OK** to save the database configuration and close the dialog.



Configure the Channel Bus Speed

Configure the channel bus speed if the desired network speed differs from the default value.

1 To open the Device Channel Configuration dialog, select **Device Channel** in the toolstrip.
2 This example uses the default bus speed at 500000 bits per second. Confirm the current device channel configuration and click **OK**.

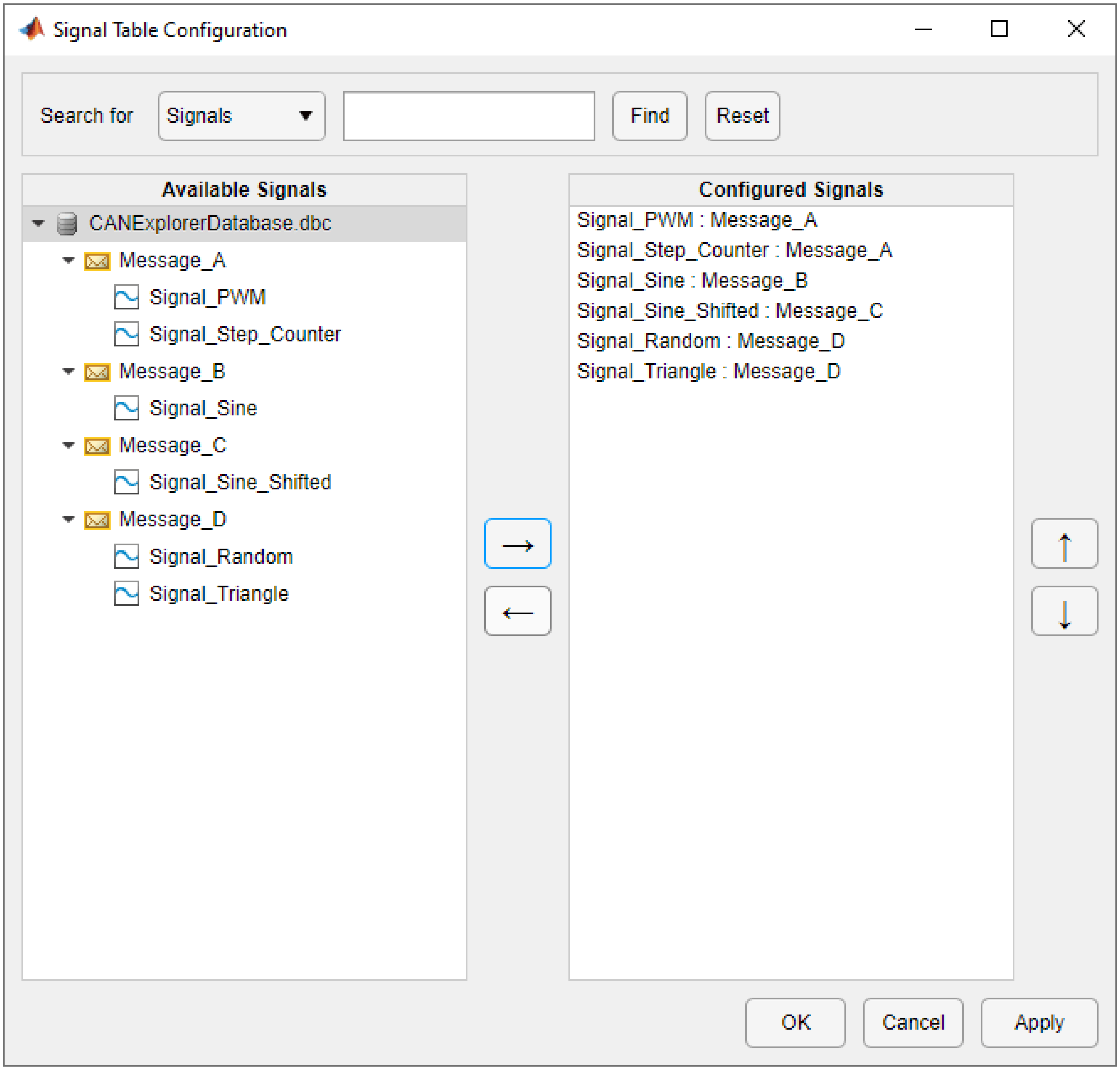


In the same dialog, you can configure message filters respectively for standard ID and extended ID to control which messages pass through the channel. By default, both filter options are set to allow all messages to pass, but you can also specify certain IDs to be allowed or blocked.

Configure the Signal Table

Add signals of interest to view on the Signal Table. In this example, you view all signals defined in the `CANExplorerDatabase.dbc` file.

1. To open the Signal Table Configuration dialog, select **Signals > Configure Signal Table** in the toolstrip.
2. Add signals from the **Available Signals** pane to the **Configured Signals** pane using the → button. You can add individual signals, add all signals in a message by adding the message, or add all signals in a database by adding the database. For this example, select `CANExplorerDatabase.dbc` in the **Available Signals** pane and click → to add all signals in the database to view.
3. Click **OK** to save the signal table configuration and close the dialog.



If you provide a search text for signals or messages and click **Find**, the **Available Signals** pane is updated to display search results that are case-insensitive partial matches to the search text.

Configure the Signal Scopes

Add signals of interest to view on the Signal Scopes. **CAN Explorer** provides 3 scopes that can each be configured to visualize signals of selection. The number of scopes is fixed and cannot be customized. In this example, you view all signals from `Message_A` in the top signal scope, all signals from `Message_B` and `Message_C` in the middle signal scope, and all signals from `Message_D` in the bottom signal scope.

1. To open the Top Signal Scope Configuration dialog, select **Signals > Configure Top Signal Scope** in the toolstrip.
2. Select `Message_A` in the **Available Signals** pane and click → to add all signals in this message to view on the top signal scope.
3. Click **OK** to save the top signal scope configuration and close the dialog.
4. Using a similar approach, add signals from `Message_B` and `Message_C` to view on the middle signal scope, and add signals from `Message_D` to view on the bottom signal scope.

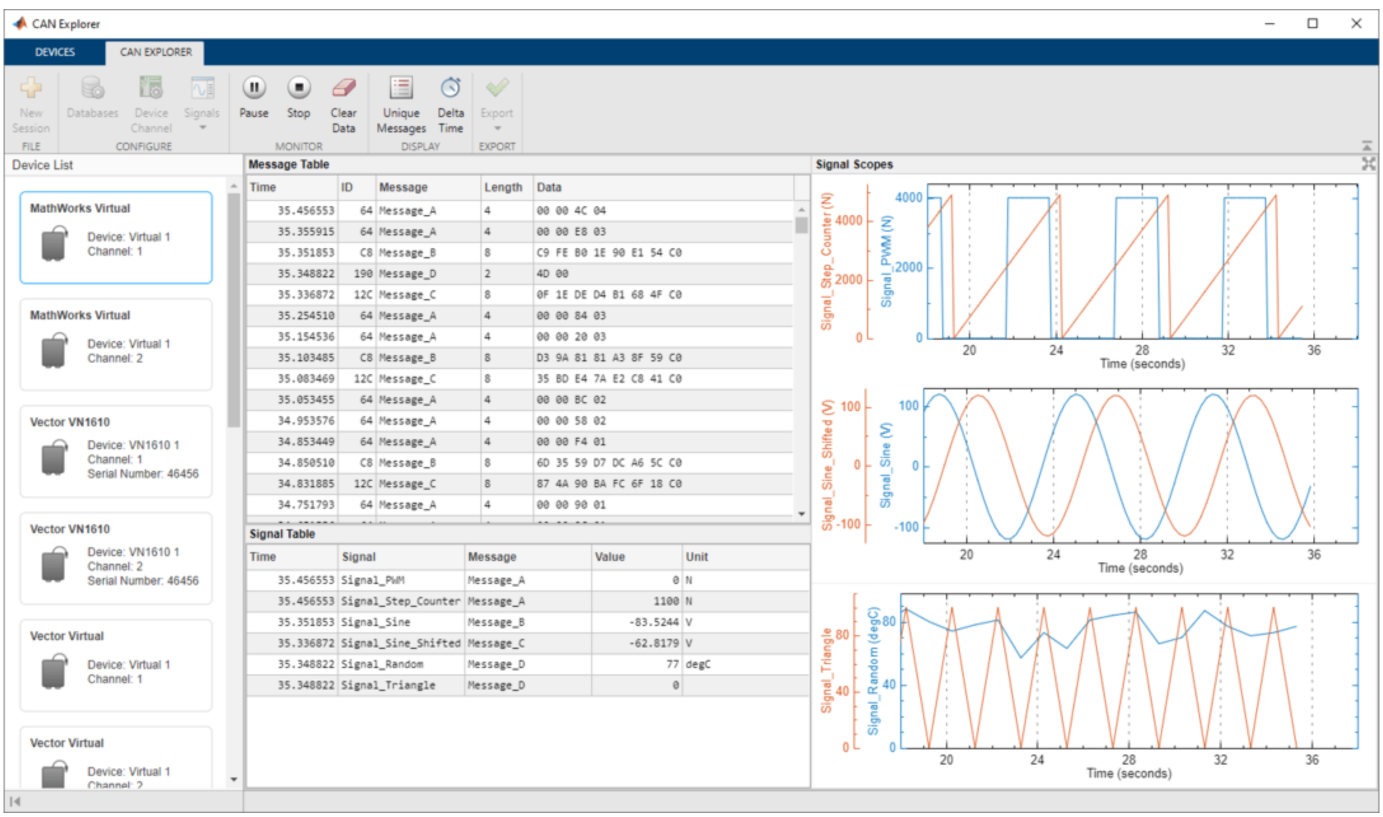
Start Monitoring

Start monitoring in **CAN Explorer** before starting the replay to avoid losing any data. Click **Start** in the toolstrip.

Replay Pre-Recorded CAN Data

Data logged from a CAN network is provided in the file `CANExplorerData.mat`. The data is saved in timetable format and the time range spans about 60 seconds.

Replay the CAN data onto MathWorks Virtual 1 Channel 2 for **CAN Explorer** to receive on MathWorks Virtual 1 Channel 1 in the same MATLAB instance. To start the data replay, execute the script `replayCANData.m`. You can also execute the script sequentially multiple times to generate CAN data beyond 60 seconds for additional experiments.



Explore the Monitor and Display Options

While **CAN Explorer** continues to receive data, you can experiment with controls in the **Monitor** and **Display** sections of the toolstrip.

1. Click **Pause** to temporarily suspend **CAN Explorer** from visually updating. While paused **CAN Explorer** continues accumulating and processing data in the background.
2. Click **Continue** to resume the visual updates in **CAN Explorer**.

For further exploration:

1. If you click **Clear Data**, all accumulated data is completely cleared from **CAN Explorer**.
2. By default, the Message Table displays all CAN messages in chronological order. To view the latest instance of each unique message, toggle **Unique Messages**.
3. By default, both the Message Table and the Signal Table display time since the start of monitoring. To view the delta time since the last message or signal in each table, toggle **Delta Time**.

Stop Monitoring

When you have completed your live acquisition activity, click **Stop** in the toolstrip to take the device channel offline.

Clean up for the Data Replay

Clean up by executing the script `replayCANDataCleanup.m`, which stops the MathWorks Virtual 1 Channel 2 used for replay and clears the unneeded variables.

Export Data for Additional Use

In the toolstrip, click the top half of the **Export** button to export the received data into the MATLAB workspace in a timetable format.

If you would like to retain the exported variable for future use:

- To save the variable to a MAT-file, use the `save` function.
- To save the variable to a BLF-file, use the `blfwrite` function.

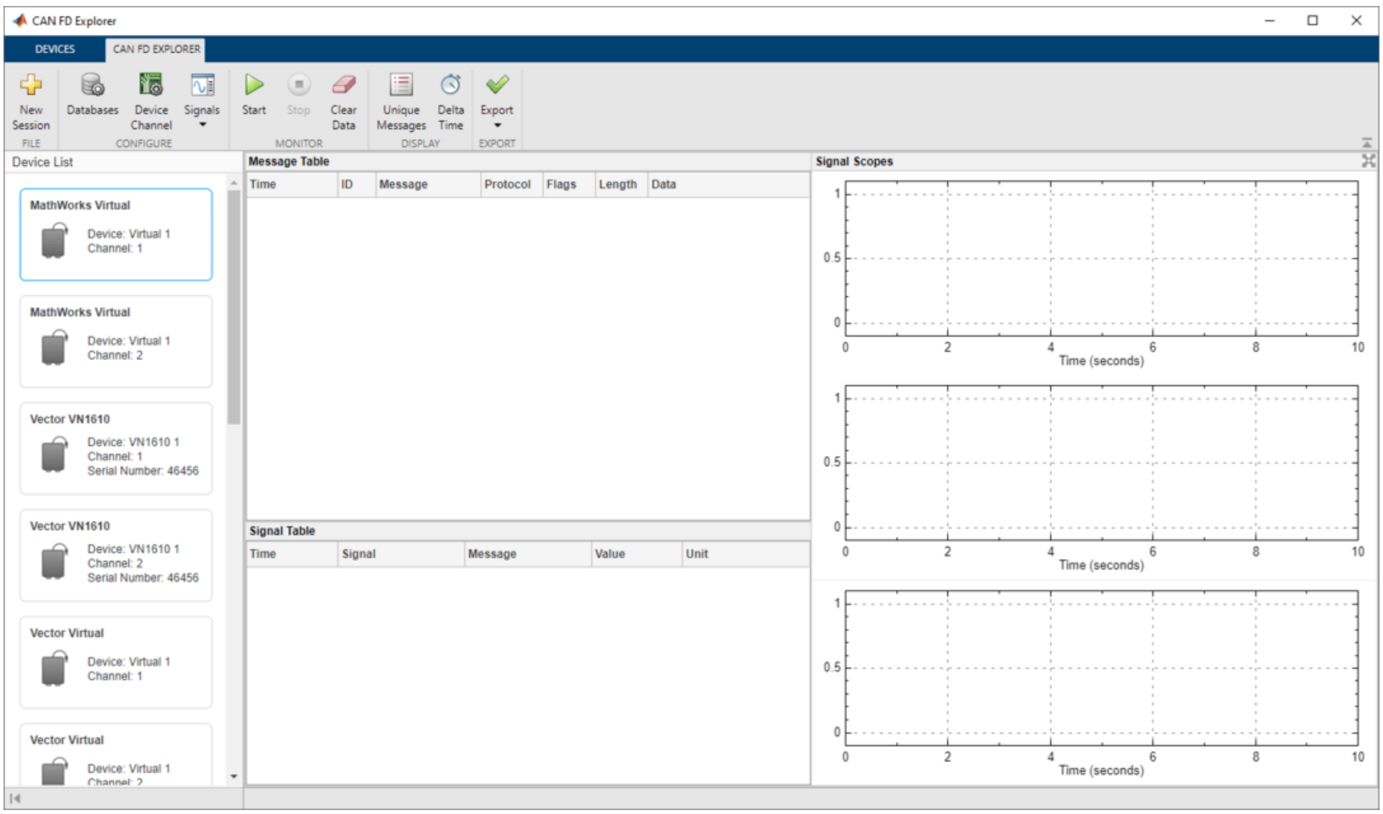
The exported timetable of messages is also convertible into individual timetables of signal data. The `canSignalTimetable` function returns a structure with one field for each unique message in the timetable. Each field value is a timetable of all the signals defined in that message.

Receive and Visualize CAN FD Data Using CAN FD Explorer

This example shows how to use the **CAN FD Explorer** app to receive and visualize CAN FD data. It uses MathWorks® Virtual channels which are connected in a loopback configuration. **CAN FD Explorer** is configured to receive data using MathWorks Virtual 1 Channel 1. Pre-recorded data is provided in a MAT-file and replayed onto MathWorks Virtual 1 Channel 2 to emulate CAN FD traffic generated from connecting to an actual vehicle system.

Open the CAN FD Explorer

Open the **CAN FD Explorer** app using command `canFDExplorer`. Alternatively, you can find **CAN FD Explorer** in the MATLAB® **Apps** tab.



Select the Device Channel

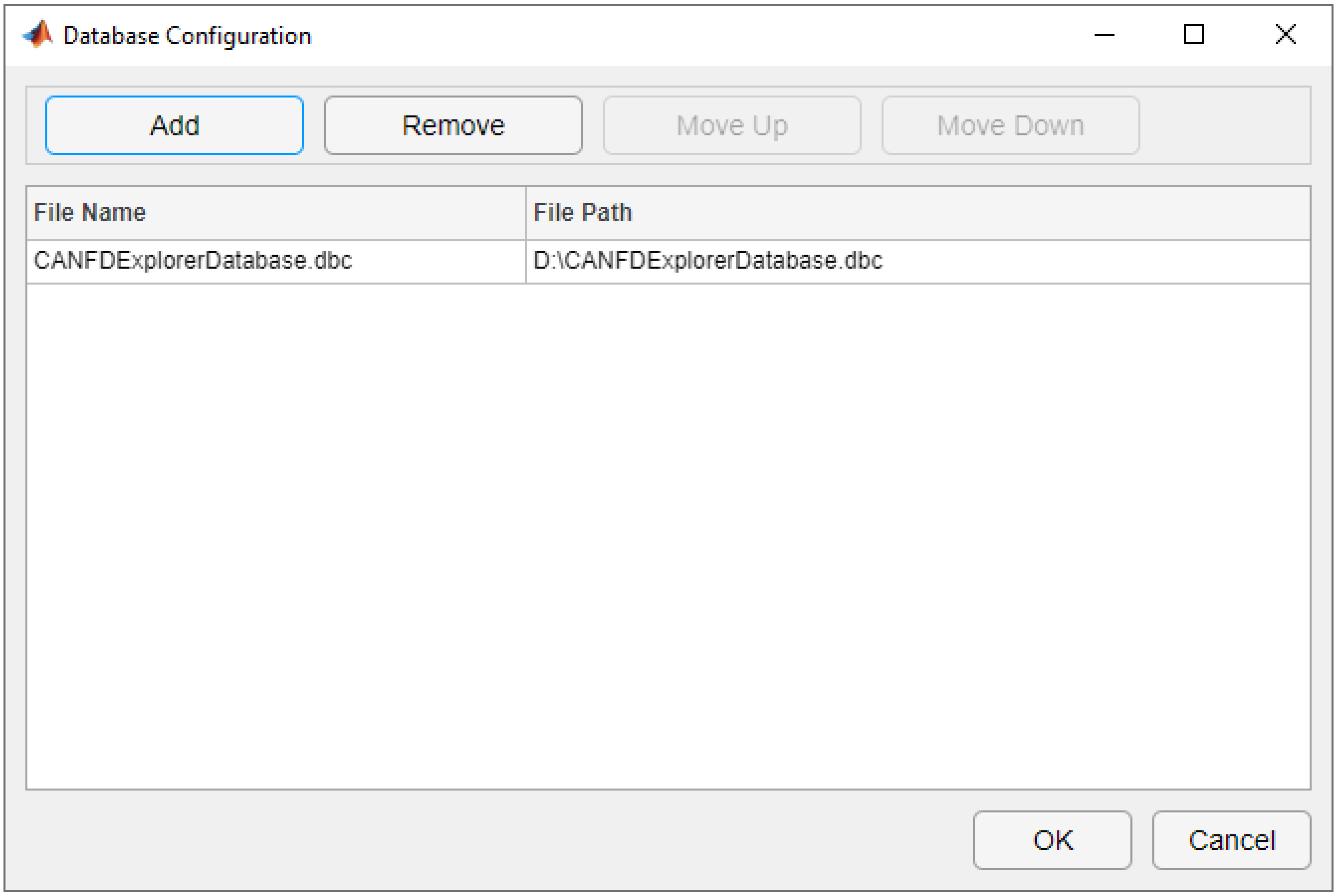
The **Device List** shows all the accessible CAN FD channels from devices connected to the system, and the current device channel in use is highlighted by a blue outline. Each time you start **CAN FD Explorer**, the first device channel in the list is automatically selected by default. Select MathWorks Virtual 1 Channel 1 from the **Device List** if it is not selected by default.

Configure the Database Files

Add database files to **CAN FD Explorer** to decode incoming messages and signals.

1. To open the Database Configuration dialog, select **Databases** in the toolstrip.
2. Click **Add** to open the file selection dialog. Select the `CANFDExplorerDatabase.dbc` file provided with the example.

3 Click **OK** to save the database configuration and close the dialog.

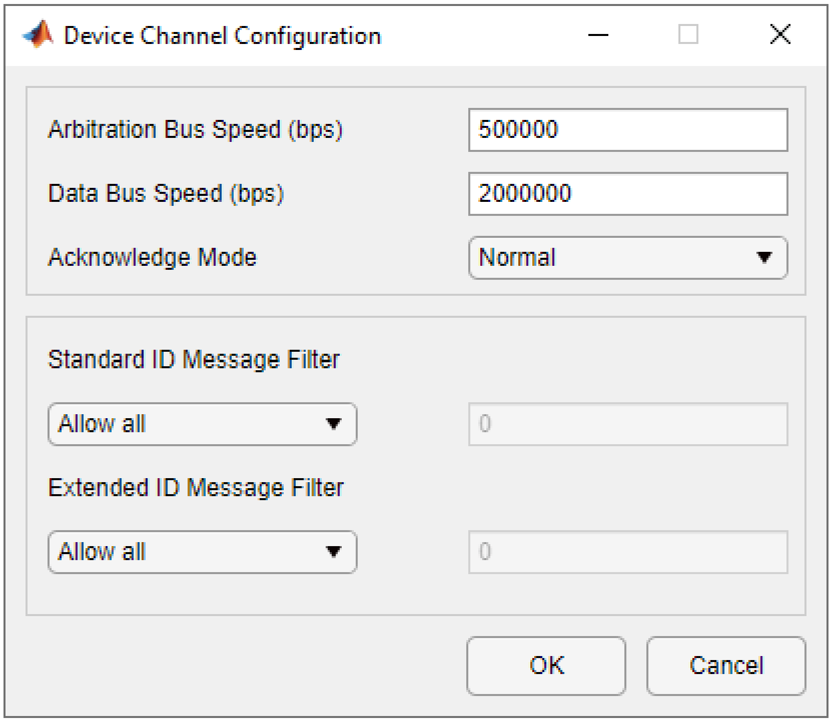


Configure the Channel Bus Speed

Configure the channel bus speed if the desired network speed differs from the default value.

1 To open the Device Channel Configuration dialog, select **Device Channel** in the toolstrip.

2 This example uses the default arbitration bus speed at 500000 bits per second and data bus speed at 2000000 bits per second. Confirm the current device channel configuration and click **OK**.

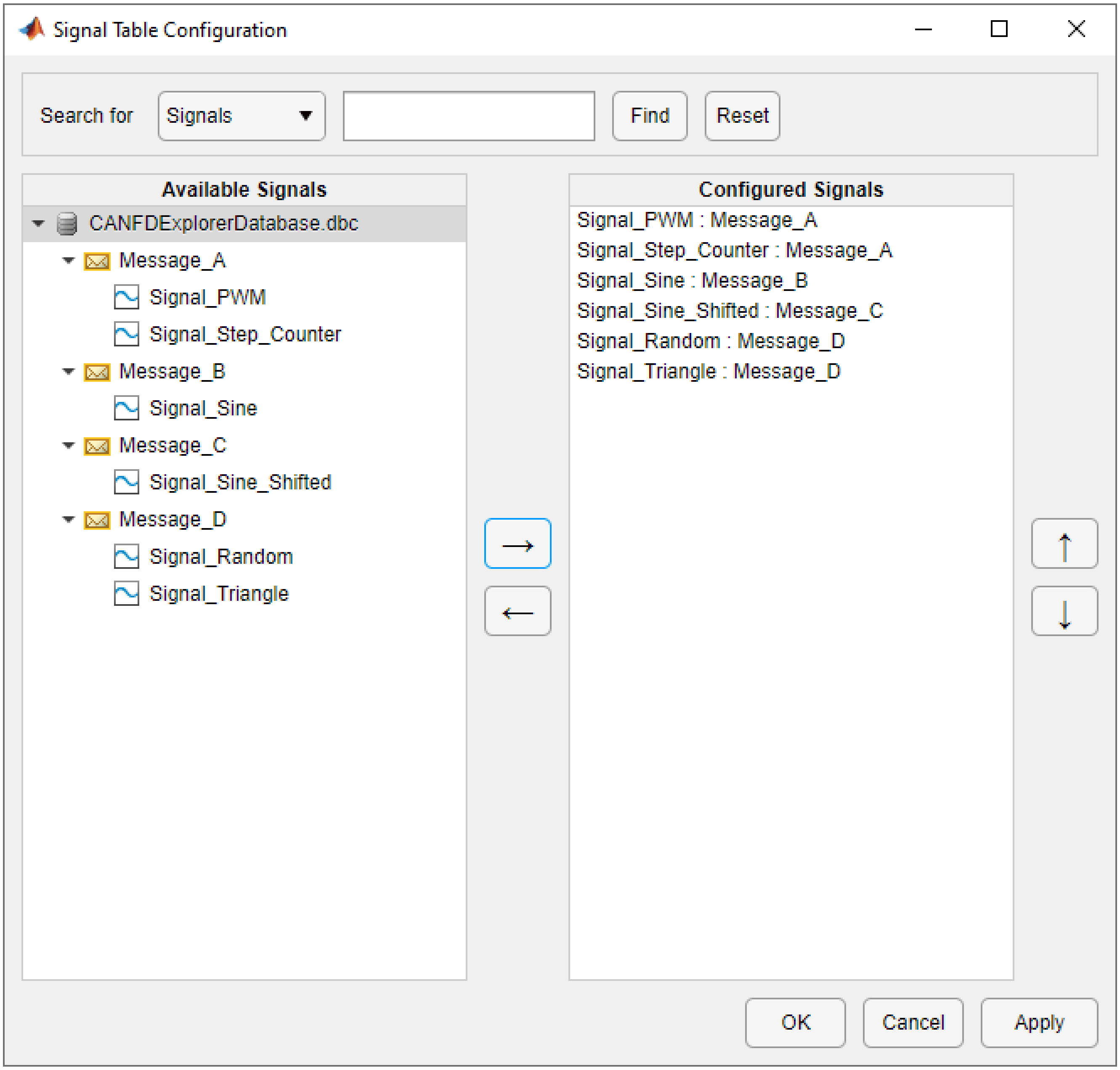


In the same dialog, you can configure message filters respectively for standard ID and extended ID to control which messages pass through the channel. By default, both filter options are set to allow all messages to pass, but you can also specify certain IDs to be allowed or blocked.

Configure the Signal Table

Add signals of interest to view on the Signal Table. In this example, you view all signals defined in the `CANFDExplorerDatabase.dbc` file.

1. To open the Signal Table Configuration dialog, select **Signals > Configure Signal Table** in the toolstrip.
2. Add signals from the **Available Signals** pane to the **Configured Signals** pane using the → button. You can add individual signals, add all signals in a message by adding the message, or add all signals in a database by adding the database. For this example, select `CANFDExplorerDatabase.dbc` in the **Available Signals** pane and click → to add all signals in the database to view.
3. Click **OK** to save the signal table configuration and close the dialog.



If you provide a search text for signals or messages and click **Find**, the **Available Signals** pane is updated to display search results that are case-insensitive partial matches to the search text.

Configure the Signal Scopes

Add signals of interest to view on the Signal Scopes. **CAN FD Explorer** provides 3 scopes that can each be configured to visualize signals of selection. The number of scopes is fixed and cannot be customized. In this example, you view all signals from `Message_A` in the top signal scope, all signals from `Message_B` and `Message_C` in the middle signal scope, and all signals from `Message_D` in the bottom signal scope.

1. To open the Top Signal Scope Configuration dialog, select **Signals > Configure Top Signal Scope** in the toolstrip.
2. Select `Message_A` in the **Available Signals** pane and click → to add all signals in this message to view on the top signal scope.
3. Click **OK** to save the top signal scope configuration and close the dialog.
4. Using a similar approach, add signals from `Message_B` and `Message_C` to view on the middle signal scope, and add signals from `Message_D` to view on the bottom signal scope.

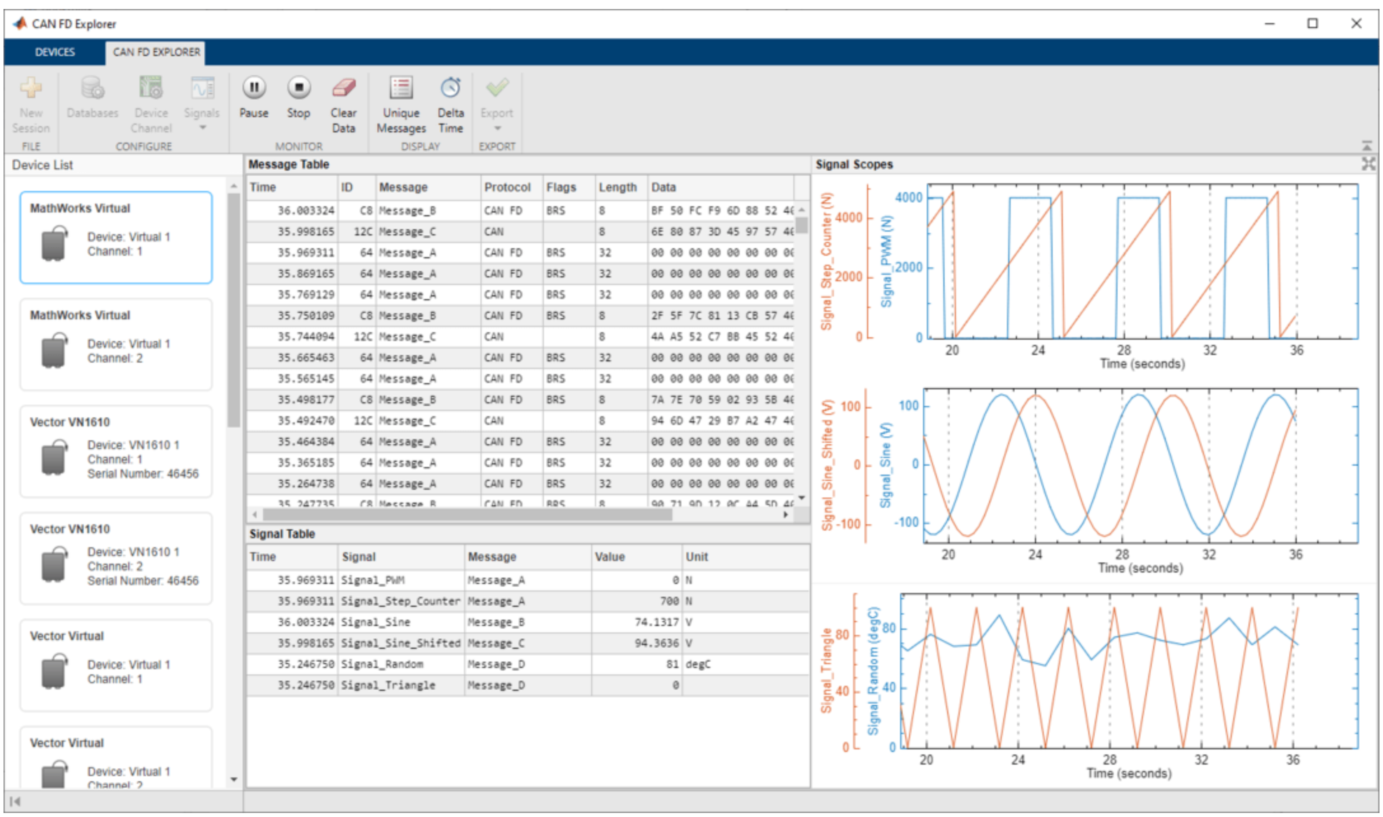
Start Monitoring

Start monitoring in **CAN FD Explorer** before starting the replay to avoid losing any data. Click **Start** in the toolstrip.

Replay Pre-Recorded CAN FD Data

Data logged from a CAN FD network is provided in the file `CANFDExplorerData.mat`. The data is saved in timetable format and the time range spans about 60 seconds.

Replay the CAN FD data onto MathWorks Virtual 1 Channel 2 for **CAN FD Explorer** to receive on MathWorks Virtual 1 Channel 1 in the same MATLAB instance. To start the data replay, execute the script `replayCANFDData.m`. You can also execute the script sequentially multiple times to generate CAN FD data beyond 60 seconds for additional experiments.



Explore the Monitor and Display Options

While **CAN FD Explorer** continues to receive data, you can experiment with controls in the **Monitor** and **Display** sections of the toolstrip.

1. Click **Pause** to temporarily suspend **CAN FD Explorer** from visually updating. While paused **CAN FD Explorer** continues accumulating and processing data in the background.
2. Click **Continue** to resume the visual updates in **CAN FD Explorer**.

For further exploration:

1. If you click **Clear Data**, all accumulated data is completely cleared from **CAN FD Explorer**.
2. By default, the Message Table displays all CAN FD messages in chronological order. To view the latest instance of each unique message, toggle **Unique Messages**.
3. By default, both the Message Table and the Signal Table display time since the start of monitoring. To view the delta time since the last message or signal in each table, toggle **Delta Time**.

Stop Monitoring

When you have completed your live acquisition activity, click **Stop** in the toolstrip to take the device channel offline.

Clean up for the Data Replay

Clean up by executing the script `replayCANFDDataCleanup.m`, which stops the MathWorks Virtual 1 Channel 2 used for replay and clears the unneeded variables.

Export Data for Additional Use

In the toolstrip, click the top half of the **Export** button to export the received data into the MATLAB workspace in a timetable format.

If you would like to retain the exported variable for future use:

- To save the variable to a MAT-file, use the `save` function.
- To save the variable to a BLF-file, use the `blfwrite` function.

The exported timetable of messages is also convertible into individual timetables of signal data. The `canSignalTimetable` function returns a structure with one field for each unique message in the timetable. Each field value is a timetable of all the signals defined in that message.

Decode J1939 Data from BLF-Files

This example shows you how to import and decode J1939 data from BLF-files in MATLAB for analysis. The BLF-file used in this example was generated from Vector CANoe using the "System Configuration (J1939)" sample. This example also uses the CAN database file, `Powertrain_J1939_BLF.dbc`, provided with the Vector sample configuration.

Investigate the BLF-File

Retrieve and view information about the BLF-file. The `blfinfo` function parses general information about the format and contents of the Vector Binary Logging Format BLF-file and returns the information as a structure.

```
binf = blfinfo("LoggingBLF_J1939.blf")
```

```
binf = struct with fields:
                  Name: "LoggingBLF_J1939.blf"
                  Path: "C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex5
           Application: "CANoe"
    ApplicationVersion: "12.0.167"
               Objects: 131119
             StartTime: 21-Apr-2021 10:05:13.232
               EndTime: 21-Apr-2021 10:09:14.344
           ChannelList: [2×3 table]
```

Notice the ChannelList property indicates there are 2 channels referenced in the BLF-file with ChannelID values of 1 and 2. The J1939 powertrain data of interest was logged from the CAN2 network, so this example focuses on ChannelID 2.

```
binf.ChannelList
```

```
ans=2×3 table
    ChannelID    Protocol    Objects
    _________    ________    _______

        1         "CAN"       92720
        2         "CAN"       26054
```

J1939 is a protocol built on top of the CAN protocol. A parameter group (PG) is a set of parameters belonging to the same topic and sharing the same transmission rate e.g. EngCoolantTemp, EngFuelTemp, EngTurboOilTemp, etc. of the ET1_EMS PG (see the ET1_EMS PG in signalTimetables below). Each parameter group is addressed via a unique number called the parameter group number (PGN). J1939 PGs are transmitted as CAN frames.

Read J1939 CAN Data Frames From the BLF-File

Read all data from channel 2 into a timetable using the `blfread` function. Each row of the timetable represents one raw CAN frame from the bus.

```
canData = blfread("LoggingBLF_J1939.blf", 2)
```

```
canData=26054×8 timetable
        Time            ID         Extended       Name                      Data
    ____________    _________    ________    __________    ___________________________________
```

```
    0.000568 sec    418316262     true       {0×0 char}    {[      105 52 169 232 0 131 0 16]}
    0.27057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.29057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.30058 sec     418382822     true       {0×0 char}    {[  255 0 255 255 255 255 255 255]}
    0.30116 sec     419327206     true       {0×0 char}    {[255 255 255 255 255 255 255 255]}
    0.31057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.33057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.35058 sec     418382822     true       {0×0 char}    {[  255 0 255 255 255 255 255 255]}
    0.35115 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.35173 sec     419327206     true       {0×0 char}    {[255 255 255 255 255 255 255 255]}
    0.3523 sec      419361254     true       {0×0 char}    {[     255 0 0 12 255 255 224 255]}
    0.37057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.39057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
    0.40058 sec     418382822     true       {0×0 char}    {[  255 0 255 255 255 255 255 255]}
    0.40116 sec     419327206     true       {0×0 char}    {[255 255 255 255 255 255 255 255]}
    0.41057 sec     418383078     true       {0×0 char}    {[  255 255 255 208 7 255 255 255]}
      ⋮
```

Decode J1939 Parameter Groups Using the DBC-File

Open the database file using the `canDatabase` function.

```
canDB = canDatabase("Powertrain_J1939_BLF.dbc")
```

```
canDB = 
  Database with properties:

             Name: 'Powertrain_J1939_BLF'
             Path: 'C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex52809
            Nodes: {12×1 cell}
         NodeInfo: [12×1 struct]
         Messages: {93×1 cell}
      MessageInfo: [93×1 struct]
       Attributes: {3×1 cell}
    AttributeInfo: [3×1 struct]
         UserData: []
```

The `j1939ParameterGroupTimetable` function uses the database to decode the raw CAN Data into PGs, PGNs and signals. The timetable of binary logging format data is converted into a Vehicle Network Toolbox™ J1939 parameter group timetable.

```
j1939PGTimetable = j1939ParameterGroupTimetable(canData, canDB)
```

```
j1939PGTimetable=26030×8 timetable
        Time           Name        PGN     Priority       PDUFormatType        SourceAddress    De
    ____________    ________    _____    ________    ____________________    _____________    __

    0.000568 sec    ACL         60928       6        Peer-to-Peer (Type 1)        230
    0.27057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)           230
    0.29057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)           230
    0.30058 sec     EEC2_EMS    61443       6        Broadcast (Type 2)           230
    0.30116 sec     TCO1_TCO    65132       6        Broadcast (Type 2)           230
    0.31057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)           230
    0.33057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)           230
    0.35058 sec     EEC2_EMS    61443       6        Broadcast (Type 2)           230
    0.35115 sec     EEC1_EMS    61444       6        Broadcast (Type 2)           230
```

```
    0.35173 sec     TCO1_TCO    65132       6        Broadcast (Type 2)          230
    0.3523 sec      CCVS_EMS    65265       6        Broadcast (Type 2)          230
    0.37057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)          230
    0.39057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)          230
    0.40058 sec     EEC2_EMS    61443       6        Broadcast (Type 2)          230
    0.40116 sec     TCO1_TCO    65132       6        Broadcast (Type 2)          230
    0.41057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)          230
      ⋮
```

View the signal data stored in the third PG of the timetable, which is one instance of the "EEC1_EMS" PG.

```
signalData = j1939PGTimetable.Signals{3}
```

```
signalData = struct with fields:
             EngDemandPercentTorque: 130
                     EngStarterMode: 15
    SrcAddrssOfCtrllngDvcForEngCtrl: 255
                           EngSpeed: 250
             ActualEngPercentTorque: 130
      DriversDemandEngPercentTorque: 130
                      EngTorqueMode: 15
```

Repackage and Visualize Signal Values of Interest

Use the `j1939SignalTimetable` function to repackage signal data from each unique PGN on the bus into a signal timetable. This example creates two individual signal timetables for the two PGs of interest, "EEC1_EMS" and "TCO1_TCO", from the J1939 PG timetable.

```
signalTimetable1 = j1939SignalTimetable(j1939PGTimetable, "ParameterGroups", "EEC1_EMS")
```

```
signalTimetable1=12043×7 timetable
       Time        EngDemandPercentTorque    EngStarterMode    SrcAddrssOfCtrllngDvcForEngCtrl
    ___________    ______________________    ______________    _______________________________

    0.27057 sec             130                    15                        255
    0.29057 sec             130                    15                        255
    0.31057 sec             130                    15                        255
    0.33057 sec             130                    15                        255
    0.35115 sec             130                    15                        255
    0.37057 sec             130                    15                        255
    0.39057 sec             130                    15                        255
    0.41057 sec             130                    15                        255
    0.43057 sec             130                    15                        255
    0.45115 sec             130                    15                        255
    0.47057 sec             130                    15                        255
    0.49057 sec             130                    15                        255
    0.51057 sec             130                    15                        255
    0.53057 sec             130                    15                        255
    0.55115 sec             130                    15                        255
    0.57057 sec             130                    15                        255
      ⋮
```

```
signalTimetable2 = j1939SignalTimetable(j1939PGTimetable, "ParameterGroups", "TCO1_TCO")
```

```
signalTimetable2=4817×14 timetable
       Time        TachographVehicleSpeed    TachographOutputShaftSpeed    DirectionIndicator
```

```
    ___________    ______________________    __________________________    __________________

    0.30116 sec             256                        8191.9                         3
    0.35173 sec             256                        8191.9                         3
    0.40116 sec             256                        8191.9                         3
    0.45173 sec             256                        8191.9                         3
    0.50116 sec             256                        8191.9                         3
    0.55173 sec             256                        8191.9                         3
    0.60116 sec             256                        8191.9                         3
    0.65173 sec             256                        8191.9                         3
    0.70116 sec             256                        8191.9                         3
    0.75173 sec             256                        8191.9                         3
    0.80116 sec             256                        8191.9                         3
    0.85173 sec             256                        8191.9                         3
    0.90116 sec             256                        8191.9                         3
    0.95173 sec             256                        8191.9                         3
    1.0012 sec              256                        8191.9                         3
    1.0517 sec              256                        8191.9                         3
      ⋮
```

You can alternatively choose to convert the whole J1939 PG timetable into a struct containing multiple J1939 signal timetables for each individual PG, and index into it to get data for a particular PG.

```
signalTimetables = j1939SignalTimetable(j1939PGTimetable)
```

```
signalTimetables = struct with fields:
          ACL: [1×14 timetable]
     CCVS_EMS: [2408×19 timetable]
           DD: [240×5 timetable]
     EEC1_EMS: [12043×7 timetable]
     EEC2_EMS: [4817×10 timetable]
      ET1_EMS: [240×6 timetable]
    HOURS_EMS: [240×2 timetable]
      LFC_EMS: [480×2 timetable]
         SERV: [240×6 timetable]
     TCO1_TCO: [4817×14 timetable]
     VDHR_EMS: [240×2 timetable]
       VI_EMS: [24×1 timetable]
       VW_SSC: [240×4 timetable]
```

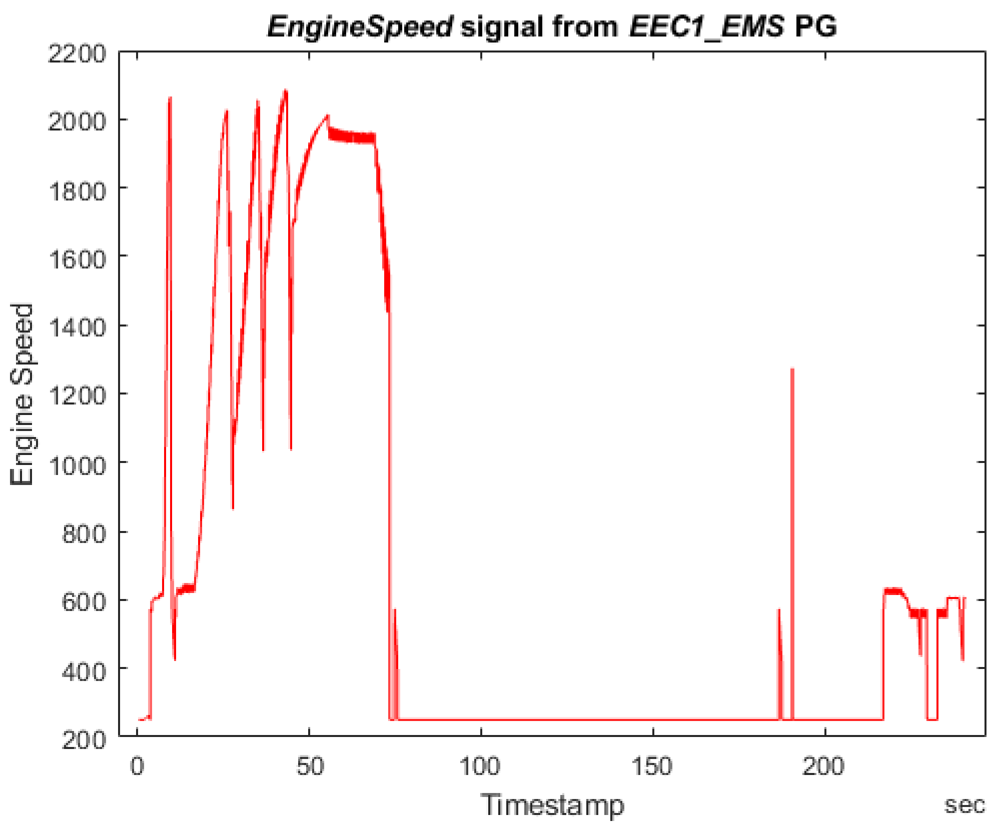
```
signalTimetables.EEC1_EMS
```

```
ans=12043×7 timetable
       Time        EngDemandPercentTorque    EngStarterMode    SrcAddrssOfCtrllngDvcForEngCtrl
    ___________    ______________________    ______________    _______________________________

    0.27057 sec             130                    15                        255
    0.29057 sec             130                    15                        255
    0.31057 sec             130                    15                        255
    0.33057 sec             130                    15                        255
    0.35115 sec             130                    15                        255
    0.37057 sec             130                    15                        255
    0.39057 sec             130                    15                        255
    0.41057 sec             130                    15                        255
    0.43057 sec             130                    15                        255
```

```
    0.45115 sec        130                15                  255
    0.47057 sec        130                15                  255
    0.49057 sec        130                15                  255
    0.51057 sec        130                15                  255
    0.53057 sec        130                15                  255
    0.55115 sec        130                15                  255
    0.57057 sec        130                15                  255
      ⋮
```

To visualize a signal of interest, variables from the signal timetables can be plotted over time for further analysis. For this example, look at the "EngineSpeed" signal from the "EEC1_EMS" PG.

```
plot(signalTimetable1.Time, signalTimetable1.EngSpeed, "r")
title("{\itEngineSpeed} signal from {\itEEC1\_EMS} PG", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Engine Speed")
```



Close the File

Close access to the DBC-file by clearing its variable from the workspace.

```
clear canDB
```

Decode J1939 Data from MDF-Files

This example shows you how to import and decode J1939 data from MDF-files in MATLAB for analysis. The MDF-file used in this example was generated from Vector CANoe using the "System Configuration (J1939)" sample. This example also uses the CAN database file, `Powertrain_J1939_MDF.dbc`, provided with the Vector sample configuration.

Open the MDF-File

Open access to the MDF-file using the `mdf` function.

```
m = mdf("LoggingMDF_J1939.mf4")
```

```
m = 
  MDF with properties:

   File Details
                Name: 'LoggingMDF_J1939.mf4'
                Path: 'C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex76
              Author: ''
          Department: ''
             Project: ''
             Subject: ''
             Comment: ''
             Version: '4.10'
            DataSize: 3743994
    InitialTimestamp: 2021-04-21 14:05:13.232000000

   Creator Details
   ProgramIdentifier: 'MDF4Lib'
             Creator: [1×1 struct]

   File Contents
          Attachment: [5×1 struct]
        ChannelNames: {43×1 cell}
        ChannelGroup: [1×43 struct]

   Options
          Conversion: Numeric
```

Identify J1939 CAN Data Frames

According to the ASAM MDF associated standard for bus logging, the event types defined for a CAN bus system can be "CAN_DataFrame", "CAN_RemoteFrame", "CAN_ErrorFrame", or "CAN_OverloadFrame". J1939 is a protocol built on top of the CAN protocol. A J1939 parameter group (PG) is a set of parameters belonging to the same topic and sharing the same transmission rate. For example, the Electronic Engine Controller 1 (EEC1) PG contains engine speed, engine torque demand percent, actual engine torque percent, etc. Each parameter group is addressed via a unique number called the parameter group number (PGN). J1939 PGs are transmitted as CAN frames, and the MDF-file reflects that a J1939 PG is logged as a "CAN_DataFrame".

The standard specifies that the channel names of the event structure should be prefixed by the event type name, for instance, "CAN_DataFrame". Typically a dot is used as a separator character to specify the member channels, for instance, "CAN_DataFrame.ID" or "CAN_DataFrame.DataLength".

Use the `channelList` function to filter on channel names exactly matching "CAN_DataFrame". A table with information on matched channels is returned.

```
channelList(m, "CAN_DataFrame", "ExactMatch", true)
```

```
ans=2×9 table
      ChannelName      ChannelGroupNumber    ChannelGroupNumSamples    ChannelGroupAcquisitionNa
    _______________    __________________    ______________________    _________________________

    "CAN_DataFrame"            13                    26054                                CAN2
    "CAN_DataFrame"            14                    92720                                CAN1
```

The J1939 powertrain data of interest was logged from the CAN 2 network. The `channelList` output above shows that the data from CAN 2 network has been stored in channel group 13 of the MDF-file. View the channel group details using the `ChannelGroup` property.

```
m.ChannelGroup(13)
```

```
ans = struct with fields:
    AcquisitionName: 'CAN2'
            Comment: ''
         NumSamples: 26054
           DataSize: 703458
             Sorted: 1
            Channel: [14×1 struct]
```

Read J1939 CAN Data Frames From the MDF-File

Read all data from all channels in channel group 13 into a timetable using the `read` function. The timetable is structured to follow the ASAM MDF standard logging format. Each row represents one raw CAN frame from the bus, while each column represents a channel within the specified channel group. The channels, such as "CAN_DataFrame.Dir", are named to follow the bus logging standard. However, because timetable column names must be valid MATLAB variable names, they might not be identical to the channel names. Most unsupported characters are converted to underscores. Because "." is not supported in a MATLAB variable name, "CAN_DataFrame.Dir" is altered to "CAN_DataFrame_Dir" in the table.

```
canData = read(m, 13, m.ChannelNames{13})
```

```
canData=26054×14 timetable
        Time        CAN_DataFrame_BusChannel    CAN_DataFrame_Flags    CAN_DataFrame_Dir    CAN_D
    ____________    ________________________    ___________________    _________________    _____

    0.000568 sec               2                         1                     1
    0.27057 sec                2                         1                     1
    0.29057 sec                2                         1                     1
    0.30058 sec                2                         1                     1
    0.30116 sec                2                         1                     1
    0.31057 sec                2                         1                     1
    0.33057 sec                2                         1                     1
    0.35058 sec                2                         1                     1
    0.35115 sec                2                         1                     1
    0.35173 sec                2                         1                     1
    0.3523 sec                 2                         1                     1
    0.37057 sec                2                         1                     1
    0.39057 sec                2                         1                     1
```

```
    0.40058 sec               2                         1                     1
    0.40116 sec               2                         1                     1
    0.41057 sec               2                         1                     1
      ⋮
```

Decode J1939 Parameter Groups Using the DBC-File

Open the database file using the `canDatabase` function.

```
canDB = canDatabase("Powertrain_J1939_MDF.dbc")
```

```
canDB = 
  Database with properties:

             Name: 'Powertrain_J1939_MDF'
             Path: 'C:\Users\michellw\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-ex76385
            Nodes: {12×1 cell}
         NodeInfo: [12×1 struct]
         Messages: {93×1 cell}
      MessageInfo: [93×1 struct]
       Attributes: {3×1 cell}
    AttributeInfo: [3×1 struct]
         UserData: []
```

The `j1939ParameterGroupTimetable` function uses the database to decode the raw CAN Data into PGs, PGNs and signals. The timetable of ASAM standard logging format data is converted into a Vehicle Network Toolbox J1939 parameter group timetable.

```
j1939PGTimetable = j1939ParameterGroupTimetable(canData, canDB)
```

```
j1939PGTimetable=26030×8 timetable
        Time           Name        PGN     Priority        PDUFormatType         SourceAddress     De
    ____________    ________    _____    ________    ______________________    _____________    _

    0.000568 sec    ACL         60928       6        Peer-to-Peer (Type 1)          230
    0.27057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.29057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.30058 sec     EEC2_EMS    61443       6        Broadcast (Type 2)             230
    0.30116 sec     TCO1_TCO    65132       6        Broadcast (Type 2)             230
    0.31057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.33057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.35058 sec     EEC2_EMS    61443       6        Broadcast (Type 2)             230
    0.35115 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.35173 sec     TCO1_TCO    65132       6        Broadcast (Type 2)             230
    0.3523 sec      CCVS_EMS    65265       6        Broadcast (Type 2)             230
    0.37057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.39057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
    0.40058 sec     EEC2_EMS    61443       6        Broadcast (Type 2)             230
    0.40116 sec     TCO1_TCO    65132       6        Broadcast (Type 2)             230
    0.41057 sec     EEC1_EMS    61444       6        Broadcast (Type 2)             230
      ⋮
```

View the signal data stored in the third PG of the timetable, which is one instance of the "EEC1_EMS" PG.

```
signalData = j1939PGTimetable.Signals{3}
```

```
signalData = struct with fields:
             EngDemandPercentTorque: 130
                     EngStarterMode: 15
    SrcAddrssOfCtrllngDvcForEngCtrl: 255
                           EngSpeed: 250
             ActualEngPercentTorque: 130
      DriversDemandEngPercentTorque: 130
                      EngTorqueMode: 15
```

Repackage and Visualize Signal Values of Interest

Use the `j1939SignalTimetable` function to repackage signal data from each unique PGN on the bus into a signal timetable. This example creates two individual signal timetables for the two PGs of interest, "EEC1_EMS" and "TCO1_TCO", from the J1939 PG timetable.

```
signalTimetable1 = j1939SignalTimetable(j1939PGTimetable, "ParameterGroups", "EEC1_EMS")
```

signalTimetable1=*12043×7 timetable*

Time	EngDemandPercentTorque	EngStarterMode	SrcAddrssOfCtrllngDvcForEngCtrl
0.27057 sec	130	15	255
0.29057 sec	130	15	255
0.31057 sec	130	15	255
0.33057 sec	130	15	255
0.35115 sec	130	15	255
0.37057 sec	130	15	255
0.39057 sec	130	15	255
0.41057 sec	130	15	255
0.43057 sec	130	15	255
0.45115 sec	130	15	255
0.47057 sec	130	15	255
0.49057 sec	130	15	255
0.51057 sec	130	15	255
0.53057 sec	130	15	255
0.55115 sec	130	15	255
0.57057 sec	130	15	255
⋮			

```
signalTimetable2 = j1939SignalTimetable(j1939PGTimetable, "ParameterGroups", "TCO1_TCO")
```

signalTimetable2=*4817×14 timetable*

Time	TachographVehicleSpeed	TachographOutputShaftSpeed	DirectionIndicator
0.30116 sec	256	8191.9	3
0.35173 sec	256	8191.9	3
0.40116 sec	256	8191.9	3
0.45173 sec	256	8191.9	3
0.50116 sec	256	8191.9	3
0.55173 sec	256	8191.9	3
0.60116 sec	256	8191.9	3
0.65173 sec	256	8191.9	3
0.70116 sec	256	8191.9	3
0.75173 sec	256	8191.9	3
0.80116 sec	256	8191.9	3

```
    0.85173 sec          256                      8191.9                       3
    0.90116 sec          256                      8191.9                       3
    0.95173 sec          256                      8191.9                       3
    1.0012 sec           256                      8191.9                       3
    1.0517 sec           256                      8191.9                       3
      ⋮
```

You can alternatively choose to convert the whole J1939 PG timetable into a struct containing multiple J1939 signal timetables for each individual PG, and index into it to get data for a particular PG.

```
signalTimetables = j1939SignalTimetable(j1939PGTimetable)
```

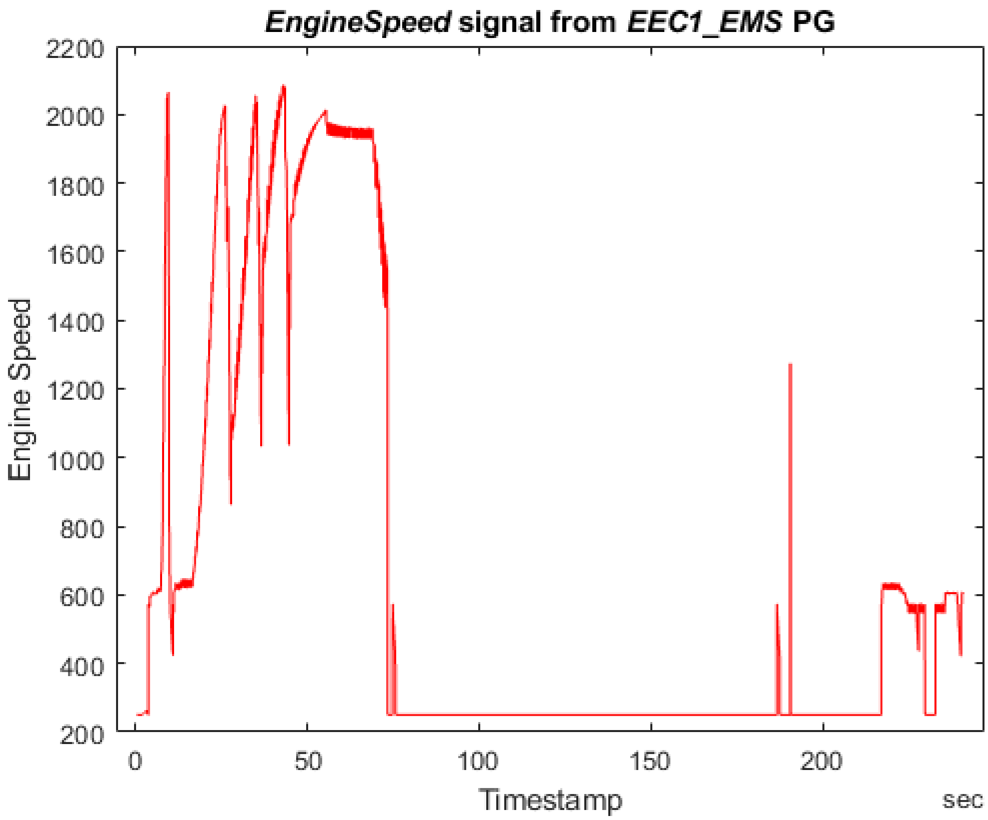
```
signalTimetables = struct with fields:
          ACL: [1×14 timetable]
     CCVS_EMS: [2408×19 timetable]
           DD: [240×5 timetable]
     EEC1_EMS: [12043×7 timetable]
     EEC2_EMS: [4817×10 timetable]
      ET1_EMS: [240×6 timetable]
    HOURS_EMS: [240×2 timetable]
      LFC_EMS: [480×2 timetable]
         SERV: [240×6 timetable]
     TCO1_TCO: [4817×14 timetable]
     VDHR_EMS: [240×2 timetable]
       VI_EMS: [24×1 timetable]
       VW_SSC: [240×4 timetable]
```

```
signalTimetables.EEC1_EMS
```

```
ans=12043×7 timetable
       Time        EngDemandPercentTorque    EngStarterMode    SrcAddrssOfCtrllngDvcForEngCtrl
    ___________    ______________________    ______________    _______________________________

    0.27057 sec             130                    15                        255
    0.29057 sec             130                    15                        255
    0.31057 sec             130                    15                        255
    0.33057 sec             130                    15                        255
    0.35115 sec             130                    15                        255
    0.37057 sec             130                    15                        255
    0.39057 sec             130                    15                        255
    0.41057 sec             130                    15                        255
    0.43057 sec             130                    15                        255
    0.45115 sec             130                    15                        255
    0.47057 sec             130                    15                        255
    0.49057 sec             130                    15                        255
    0.51057 sec             130                    15                        255
    0.53057 sec             130                    15                        255
    0.55115 sec             130                    15                        255
    0.57057 sec             130                    15                        255
      ⋮
```

To visualize a signal of interest, variables from the signal timetables can be plotted over time for further analysis. For this example, look at the "EngineSpeed" signal from the "EEC1_EMS" PG.

```
plot(signalTimetable1.Time, signalTimetable1.EngSpeed, "r")
title("{\itEngineSpeed} signal from {\itEEC1\_EMS} PG", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Engine Speed")
```



Close the Files

Close access to the MDF-file and the DBC-file by clearing their variables from the workspace.

```
clear m
clear canDB
```

Replay J1939 Logged Field Data to a Simulation

This example shows how to replay J1939 data from a BLF-file acquired from a J1939 system in a real-world application, such as a vehicle running in the field. The Simulink model runs a simple horsepower estimator algorithm to trigger a fault that might have occurred in the field. The example takes you through a part of the model-based workflow using field data to recreate a fault that was present in the Simulink algorithm before it was deployed onto an ECU, and can be extended to test any algorithm model to debug faults.

J1939 is a higher-layer protocol that uses the Controller Area Network (CAN) bus technology as a physical layer. Since CAN is the basis of data transfer in a J1939 system, the tool used in the field by default logs J1939 data as CAN frames. This example performs data replay of the originally logged CAN frames over a CAN bus from MATLAB and receives in a Simulink model using the J1939 Network Configuration, J1939 Node Configuration, J1939 CAN Transport Layer, and J1939 Receive blocks.

The BLF-file used in this example was generated from Vector CANoe using the "System Configuration (J1939)" sample configuration, and modified using MATLAB and Vehicle Network Toolbox. This example also uses the J1939 DBC-file `PowerTrain_J1939.dbc`, provided with the Vector sample configuration. Vehicle Network Toolbox provides J1939 Simulink blocks for receiving and transmitting parameter groups (PG) via Simulink models over CAN. The example uses MathWorks virtual CAN channels connected in a loopback configuration.

Read the BLF-File Data

Using the `blfread` function, read the data from channel 1 of the BLF-file that was acquired in the field.

```
canData = blfread("LoggingBLF_J1939Replay.blf",1)
```

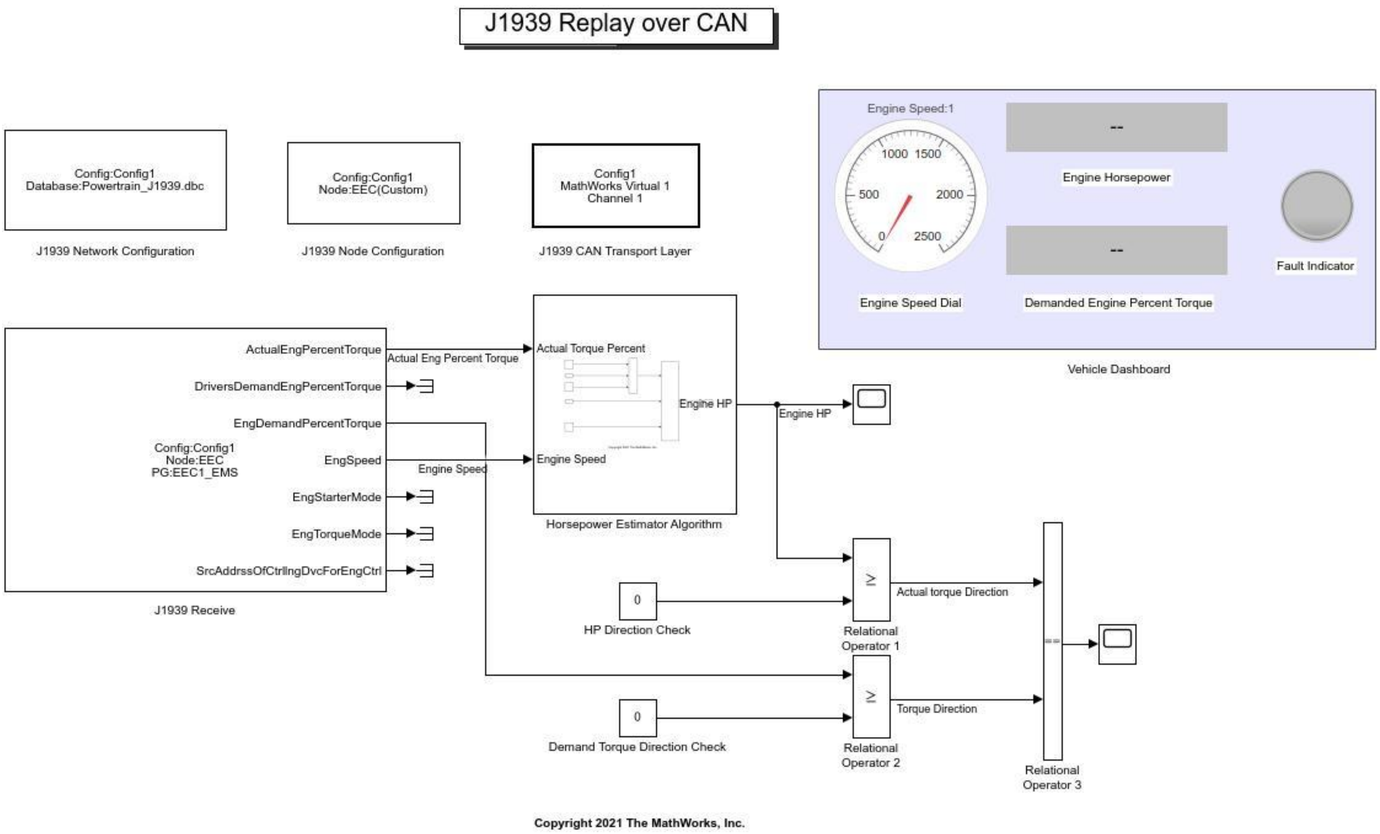
```
canData=15000×8 timetable
        Time           ID        Extended       Name                     Data
    ____________    _________    ________    __________    ____________________________________

    0.000568 sec    418316032     true       {0×0 char}    {[          76 52 169 232 0 0 0 0]}
    0.001128 sec    418316035     true       {0×0 char}    {[          78 52 169 232 0 3 0 0]}
    0.001688 sec    418316043     true       {0×0 char}    {[          75 52 169 232 0 9 0 0]}
    0.002244 sec    418316055     true       {0×0 char}    {[         77 52 169 232 0 19 0 0]}
    0.002796 sec    418316083     true       {0×0 char}    {[         79 52 169 232 0 38 0 0]}
    0.003364 sec    418316262     true       {0×0 char}    {[      105 52 169 232 0 131 0 16]}
    0.003932 sec    418316262     true       {0×0 char}    {[      105 52 169 232 0 131 0 16]}
    0.25158 sec     201326595     true       {0×0 char}    {[252 255 255 255 248 255 255 255]}
    0.25216 sec     201326603     true       {0×0 char}    {[252 255 255 255 248 255 255 255]}
    0.25272 sec     217055747     true       {0×0 char}    {[        192 0 0 250 240 240 7 3]}
    0.2533 sec      217056000     true       {0×0 char}    {[            1 0 0 0 0 252 0 255]}
    0.25386 sec     217056256     true       {0×0 char}    {[        240 0 125 208 7 0 241 0]}
    0.25444 sec     418382091     true       {0×0 char}    {[               0 0 0 0 0 1 11 3]}
    0.25501 sec     418383107     true       {0×0 char}    {[            125 0 0 125 0 0 0 0]}
    0.2556 sec      418384139     true       {0×0 char}    {[                0 0 0 0 0 0 0 0]}
    0.25618 sec     419283979     true       {0×0 char}    {[      3 0 0 255 255 255 255 255]}
      ⋮
```

This data contains one PG of interest for this example called `EEC1_EMS`. The PG contains data coming from the Engine Electronic Controller module. This example manipulates the dataset from the BLF-file to deliberately trigger a failure mode for demonstration purposes. The Simulink model recreates this failure using the modified dataset.

Open the Simulink Model

Open the Simulink model that contains your algorithm. The model contained in this example uses a basic J1939 network setup. For more details on this setup and the J1939 blocks, see the example "Get Started with J1939 Communication in Simulink" on page 14-77.

```
open demoVNTSL_J1939ReplayExample
```



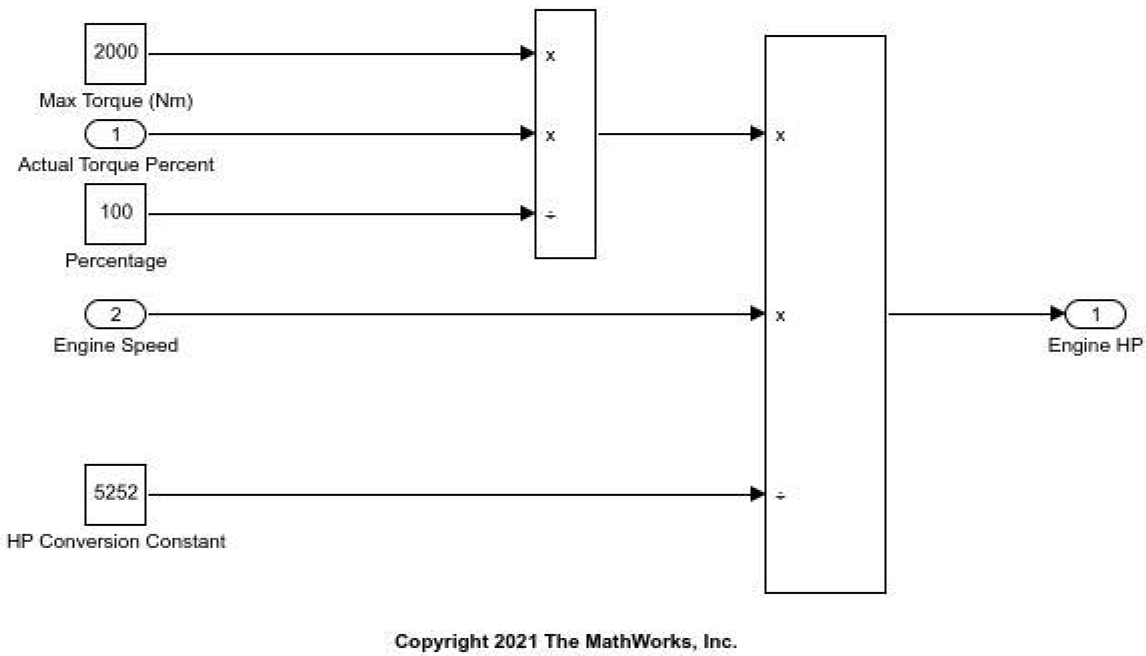
Model Overview

The example model is configured to perform a receive operation for the `EEC1_EMS` PG over the MathWorks virtual device 1 channel 1.

- The J1939 Network Configuration block is configured with the database `Powertrain_J1939.dbc`.
- The J1939 CAN Transport Layer block sets the Device to MathWorks virtual channel 1. The transport layer is configured to transfer J1939 messages over CAN via the specified virtual channel.
- The J1939 Receive block receives the messages transmitted over the network. The J1939 Receive is configured to receive the `EEC1_EMS` PG and pass on the required inputs (Actual Engine Percentage Torque (%) and Engine Speed (RPM)) to the Horsepower Estimator Algorithm. It is also configured to pass the Engine Demanded Percent Torque (%) to a relational operator block. The rest of the outputs have been terminated for simplicity.

Horsepower Estimator Algorithm

The Horsepower Estimator Algorithm is a simple calculation which takes the actual engine torque percentage and speed values and computes engine horsepower from them.



Relational Operators

There are three relational operator blocks in the model:

- Relational Operator 1 compares the value of computed horsepower to zero and outputs a Boolean.
- Relational Operator 2 compares the value of engine demanded torque percentage to zero and outputs a Boolean.
- Relational Operator 3 compares the value of the outputs from Relational Operators 1 and 2 and outputs a Boolean to trigger the state of the Fault Indicator lamp.

Vehicle Dashboard

The Vehicle Dashboard consists of the speed dial showing the engine RPM, the two gauges showing the computed value of horsepower and the engine percent demanded torque, and the Fault Indicator lamp.

Create the Channel for Replay

Create the CAN channel to replay the messages using the `canChannel` function.

```
replayChannel = canChannel("MathWorks","Virtual 1",2);
```

Set Model Parameters and Start the Simulation

Assign the simulation time and start the simulation.

```
set_param("demoVNTSL_J1939ReplayExample","StopTime","inf");
set_param("demoVNTSL_J1939ReplayExample","SimulationCommand","start");
```

Pause until the simulation is fully started.

```
while strcmp(get_param("demoVNTSL_J1939ReplayExample","SimulationStatus"),"stopped")
end
```

Start the CAN Channel and Replay the Data

Start the MATLAB CAN channel.

```
start(replayChannel);
pause(2);
```

Replay the data acquired from the BLF-file. The `replay` operation runs for approximately 45 seconds.

```
replay(replayChannel,canData);
```

Simulation Overview

During the running of this example, observe the Simulink model. There will be changes in value in the gauges and the red-green light transition of the Fault Indicator lamp in the Vehicle Dashboard section.

The J1939 Receive block receives the `EEC1_EMS` PG from MATLAB, decodes the signals of interest, and passes them to the Horsepower Estimator Algorithm. After the horsepower is computed, Relational Operator 1 compares its values to zero to determine the direction. The J1939 Receive block also passes the Engine Demanded Percent Torque to Relational Operator 2. Relational Operator 2 compares its values to zero to determine the direction.

The output is a Boolean 1 if the value is greater than or equal to zero, or 0 if it is less than zero (negative).

Relational Operator 3 takes the outputs of the earlier two relational operators and equates them. If the value for both the blocks is 0 or 1, i.e., positive horsepower and positive torque (1), or negative horsepower and negative torque (0), it provides an output of 1, which in turn triggers the green light of the Fault Indicator lamp. However, if the value for either of the earlier relational operator blocks is opposite to the other one, i.e., positive horsepower (1) and negative torque (0), or negative horsepower (0) and positive torque (1), it provides an output of 0, which in turn triggers the red light of the Fault Indicator lamp. These observations are helpful in determining whether the algorithm is faulty based on the field data, and you can further analyze the algorithm.

Stop the CAN Channel

```
stop(replayChannel);
```

Stop the Simulation

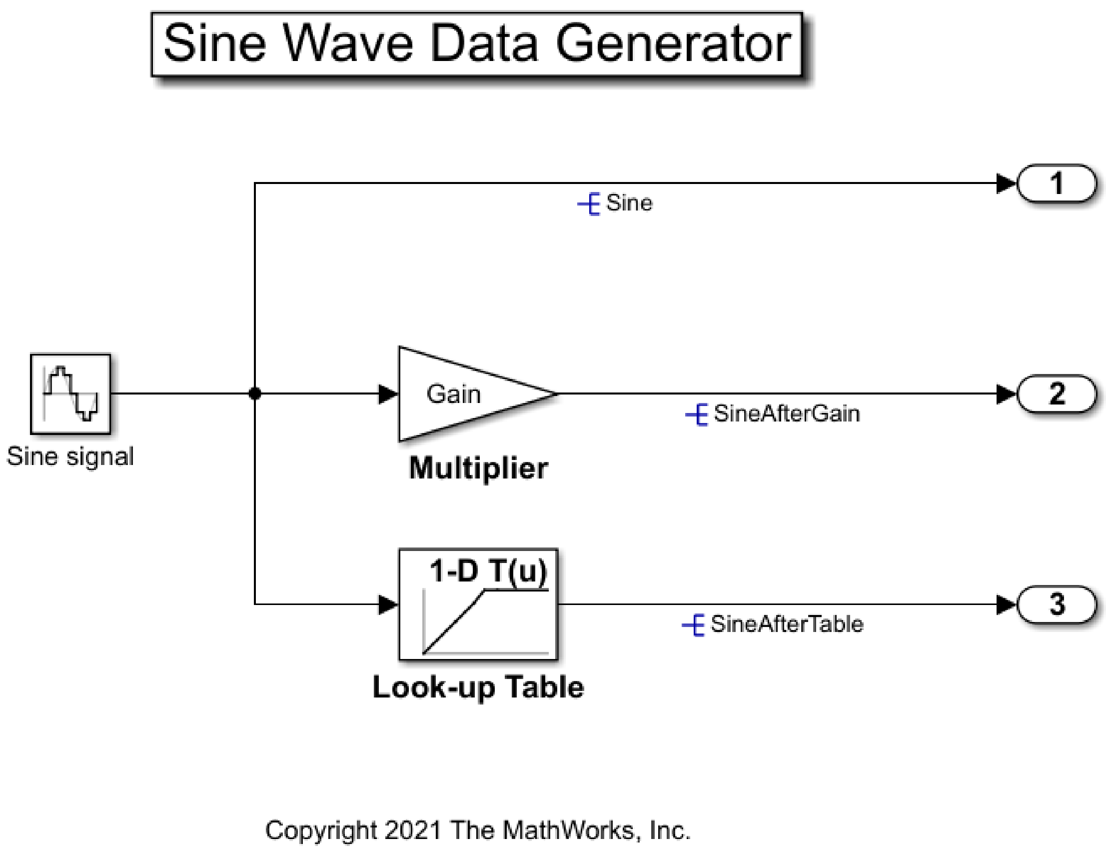
```
set_param("demoVNTSL_J1939ReplayExample","SimulationCommand","stop");
```

Calibrate XCP Characteristics

This example shows how to use the XCP protocol capability to connect and calibrate available characteristic data from a Simulink model deployed to a Windows executable. The example writes to modify the parameters of the model using TCP and direct memory access, and compares the measurements before and after calibration. XCP is a high-level protocol used for accessing and modifying internal parameters and variables of a model, algorithm, or ECU. For more information, refer to the ASAM standards.

Algorithm Overview

The algorithm used in this example is a Simulink model built and deployed as an XCP server. The model has already been compiled and is available to run in the file `XCPServerSineWaveGenerator.exe`. Additionally, the A2L-file `XCPServerSineWaveGenerator.a2l` is provided as an output of that build process. The model contains three measurements and two characteristics, accessible via XCP. Because the model is already deployed, Simulink is not required to run this example. The following image illustrates the model.



The signal `SineAfterGain` is obtained by using the multiplier `Gain` to scale the source signal `Sine`, and the signal `SineAfterTable` is obtained by using the 1-D look-up table to modify the source signal `Sine`. Calibrating the parameter `Gain` and the 1-D look-up table produces different `SineAfterGain` and `SineAfterTable` waveforms, correspondingly.

For details about how to build a Simulink model, including an XCP server and generating an A2L-file, see “Export ASAP2 File for Data Measurement and Calibration” (Simulink Coder).

Run the XCP Server Model

To communicate with the XCP server, the deployed model must be run. By using the `system` function, you can execute the `XCPServer.exe` from inside MATLAB. The function requires constructing an argument list pointing to the executable. A separate command window opens and shows running outputs from the server.

```
sysCommand = ['"', fullfile(pwd, 'XCPServerSineWaveGenerator.exe'),'"', ' &'];
system(sysCommand);
```

Open the A2L-File

An A2L-file is required to establish a connection to the XCP server. The A2L-file describes all the functionality and capability that the XCP server provides, as well as the details of how to connect to the server. Use the `xcpA2L` function to open the A2L-file that describes the server model.

```
a2lInfo = xcpA2L("XCPServerSineWaveGenerator.a2l")
```

```
a2lInfo = 
  A2L with properties:

   File Details
                 FileName: 'XCPServerSineWaveGenerator.a2l'
                 FilePath: 'C:\Users\siyingl\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-e
               ServerName: 'ModuleName'
                 Warnings: [0×0 string]

   Parameter Details
                   Events: {'100 ms'}
                EventInfo: [1×1 xcp.a2l.Event]
             Measurements: {'Sine'  'SineAfterGain'  'SineAfterTable'  'XCPServer_DW.lastCos'  'X
          MeasurementInfo: [6×1 containers.Map]
          Characteristics: {'Gain'  'ydata'}
       CharacteristicInfo: [2×1 containers.Map]
                 AxisInfo: [1×1 containers.Map]
            RecordLayouts: [4×1 containers.Map]
             CompuMethods: [3×1 containers.Map]
                CompuTabs: [0×1 containers.Map]
               CompuVTabs: [0×1 containers.Map]

   XCP Protocol Details
        ProtocolLayerInfo: [1×1 xcp.a2l.ProtocolLayer]
                  DAQInfo: [1×1 xcp.a2l.DAQ]
    TransportLayerCANInfo: [0×0 xcp.a2l.XCPonCAN]
    TransportLayerUDPInfo: [0×0 xcp.a2l.XCPonIP]
    TransportLayerTCPInfo: [1×1 xcp.a2l.XCPonIP]
```

TCP is the transport protocol used to communicate with the XCP server. Details for the TCP connection, such as the IP address and port number, are contained in the `TransportLayerTCPInfo` property.

```
a2lInfo.TransportLayerTCPInfo
```

```
ans = 
  XCPonIP with properties:
          CommonParameters: [1×1 xcp.a2l.CommonParameters]
    TransportLayerInstance: ''
                      Port: 17725
                   Address: 2.1307e+09
             AddressString: '127.0.0.1'
```

Create an XCP Channel

To create an active XCP connection to the server, use the `xcpChannel` function. The function requires a reference to the server A2L-file and the type of transport protocol to use for messaging with the server.

```
xcpCh = xcpChannel(a2lInfo, "TCP")
```

```
xcpCh = 
  Channel with properties:

              ServerName: 'ModuleName'
             A2LFileName: 'XCPServerSineWaveGenerator.a2l'
          TransportLayer: 'TCP'
    TransportLayerDevice: [1×1 struct]
              SeedKeyDLL: []
```

Connect to the Server

To make communication with the server active, use the `connect` function.

```
connect(xcpCh)
```

View Available Characteristics from A2L-File

A characteristic in XCP represents a tunable parameter in the memory of the model. Characteristics available for calibration are defined in the A2L-file and can be found in the `Characteristics` property. Note that the parameter `Gain` is the multiplier and `ydata` specifies the output data points of the 1-D look-up table.

```
a2lInfo.Characteristics
```

```
ans = 1×2 cell
    {'Gain'}    {'ydata'}
```

```
a2lInfo.CharacteristicInfo("Gain")
```

```
ans = 
  Characteristic with properties:
                 Name: 'Gain'
       LongIdentifier: ''
   CharacteristicType: VALUE
           ECUAddress: 549960
              Deposit: [1×1 xcp.a2l.RecordLayout]
              MaxDiff: 0
           Conversion: [1×1 xcp.a2l.CompuMethod]
```

```
              LowerLimit: -5
              UpperLimit: 5
               Dimension: 1
          AxisConversion: {1×0 cell}
                 BitMask: []
               ByteOrder: MSB_LAST
                Discrete: []
     ECUAddressExtension: 0
                  Format: ''
                  Number: []
                PhysUnit: ''
```

```
a2lInfo.CharacteristicInfo("ydata")
```

```
ans = 
  Characteristic with properties:
                     Name: 'ydata'
           LongIdentifier: 'Y data'
       CharacteristicType: CURVE
               ECUAddress: 550024
                  Deposit: [1×1 xcp.a2l.RecordLayout]
                  MaxDiff: 0
               Conversion: [1×1 xcp.a2l.CompuMethod]
               LowerLimit: -2
               UpperLimit: 2
                Dimension: 7
           AxisConversion: {[1×1 xcp.a2l.CompuMethod]}
                  BitMask: []
                ByteOrder: MSB_LAST
                 Discrete: []
      ECUAddressExtension: 0
                   Format: ''
                   Number: []
                 PhysUnit: ''
```

Inspect Preloaded Characteristic Values

Read the current value of the characteristic `Gain`. The `readCharacteristic` function performs a direct read from the server for a given characteristic.

```
initialGain = readCharacteristic(xcpCh, "Gain")
```

```
initialGain = 2
```

Read the current 1-D look-up table characteristic using `readAxis` and `readCharacteristic`, then plot the mapping. This table effectively maps any positive input value to zero output.

```
inputBreakpoints = readAxis(xcpCh, "xdata")
```

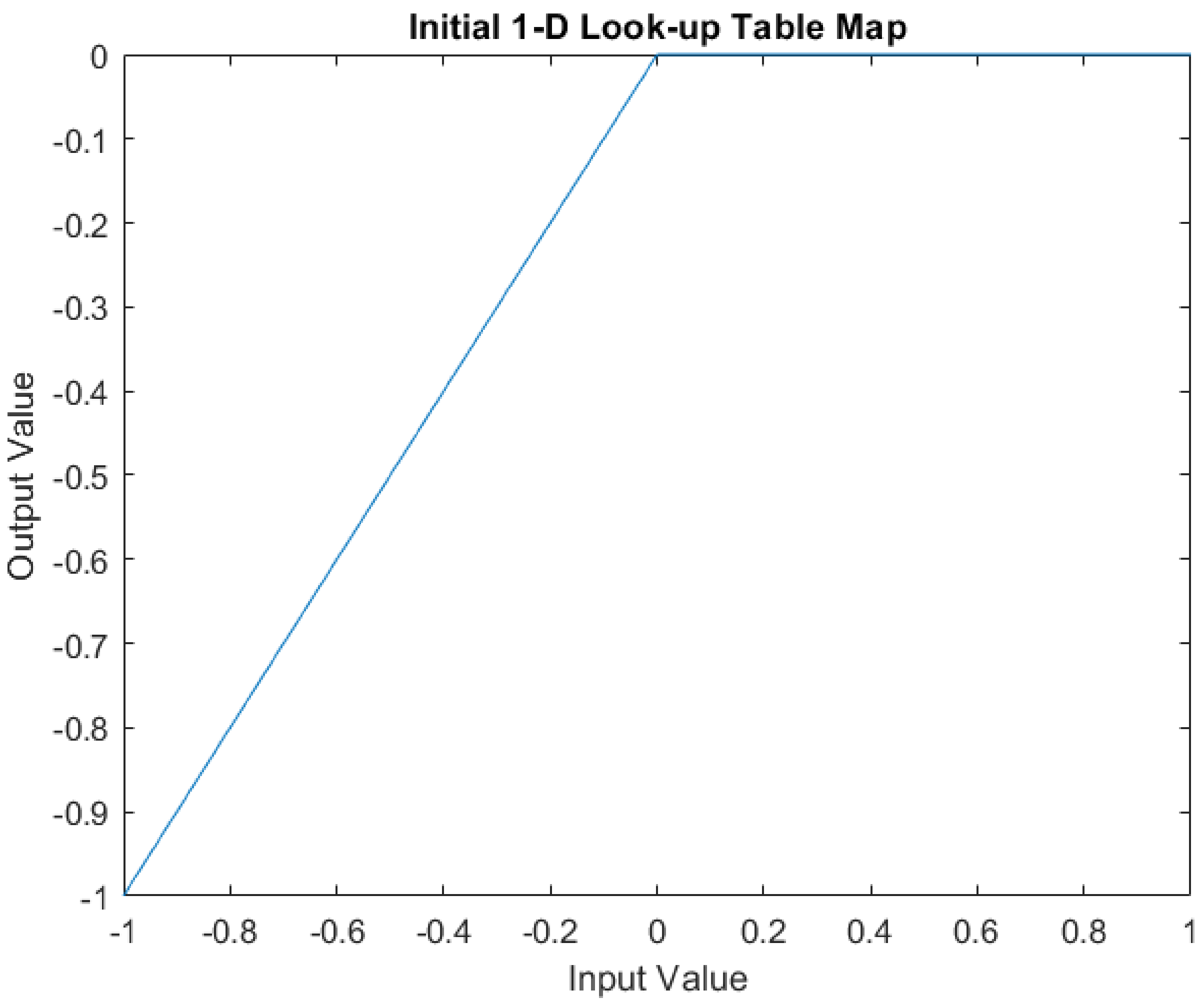
```
inputBreakpoints = 1×7

   -1.0000   -0.5000   -0.2000         0    0.2000    0.5000    1.0000
```

```
outputPoints = readCharacteristic(xcpCh, "ydata")
```

```
outputPoints = 1×7
```

```
   -1.0000   -0.5000   -0.2000         0         0         0         0
```

```
plot(inputBreakpoints, outputPoints);
title("Initial 1-D Look-up Table Map");
xlabel("Input Value");
ylabel("Output Value");
```



Create a Measurement List

This example explores the value of the measurement `Sine`, unmodified and modified by the two characteristics. To visualize the continuously changing value of `Sine` pre- and post-calibration, acquire measurement data values using a DAQ list. Use the `createMeasurementList` function to create a DAQ list containing all `Sine`-based measurements available from the server.

```
createMeasurementList(xcpCh, "DAQ", "100 ms", ["Sine", "SineAfterGain", "SineAfterTable"])
```

Obtain Measurements Before Calibration

Use the `startMeasurement` function and `stopMeasurement` function to run the DAQ list for a short period of time.

```
startMeasurement(xcpCh);
pause(3);
stopMeasurement(xcpCh);
```

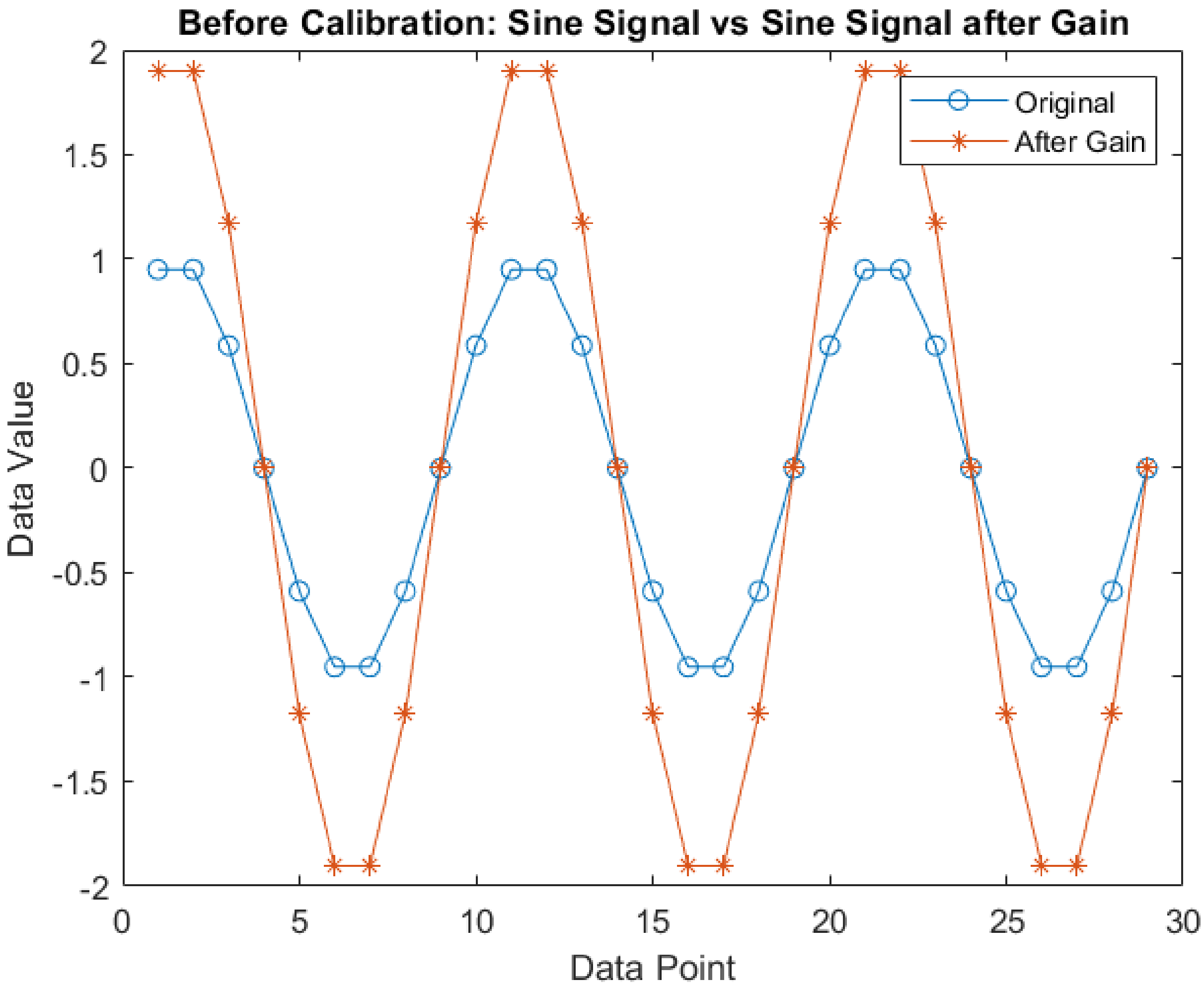
To retrieve the data acquired by the DAQ list for all the Sine-based measurements, use the `readDAQ` function. The number of retrieved samples during 3 seconds at 100 ms event is expected to be 30, but because the XCP server runs on Windows, which is not a real-time operating system, the actual number of retrieved samples might be less than 30, depending on how occupied the operating system is.

```
sine = readDAQ(xcpCh, "Sine");
sineAfterGain = readDAQ(xcpCh, "SineAfterGain");
sineAfterTable = readDAQ(xcpCh, "SineAfterTable");
```

Inspect Measurements Before Calibration

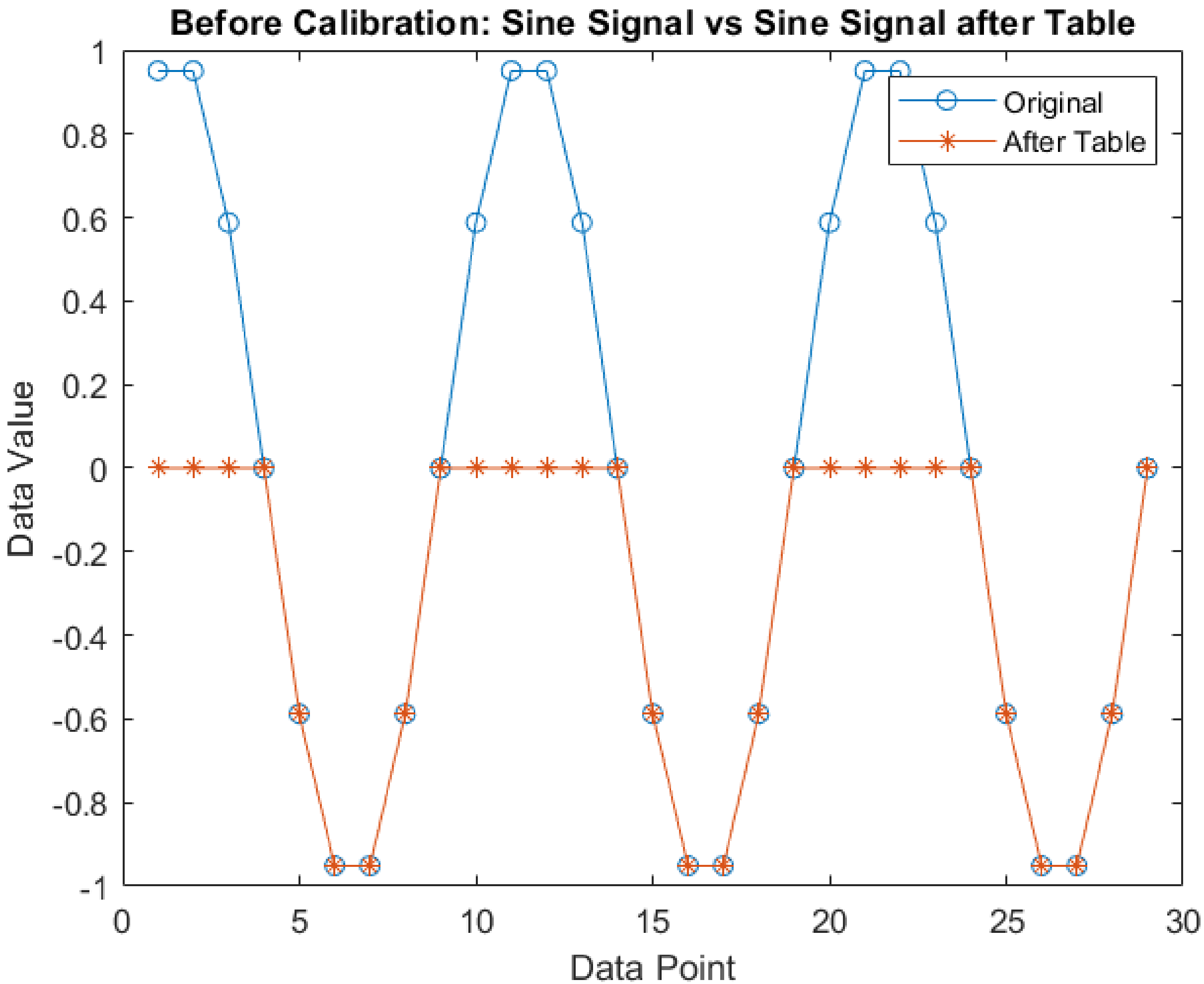
Plot the `SineAfterGain` measurement against the base `Sine` measurement. The value after `Gain` is boosted by a factor of 2, based on the original measurement, because the preloaded value of the characteristic `Gain` is 2, as shown previously.

```
plot(sine, "o-"); hold on;
plot(sineAfterGain, "*-"); hold off;
title("Before Calibration: Sine Signal vs Sine Signal after Gain");
legend("Original", "After Gain");
xlabel("Data Point");
ylabel("Data Value");
```



Plot the `SineAfterTable` measurement against the base `Sine` measurement. Any positive value of the original measurement is mapped to zero according to the preloaded 1-D look-up table, therefore the modified signal looks truncated and does not have any positive values.

```
plot(sine, "o-"); hold on;
plot(sineAfterTable, "*-"); hold off;
title("Before Calibration: Sine Signal vs Sine Signal after Table");
legend("Original", "After Table");
xlabel("Data Point");
ylabel("Data Value");
```



Calibrate the Gain and 1-D Look-up Table

Write a new value to the charateristic `Gain` using `writeCharacteristic`, and perform a read to verify the change using `readCharacteristic`.

```
writeCharacteristic(xcpCh, "Gain", 0.5);
newGain = readCharacteristic(xcpCh, "Gain")
```

```
newGain = 0.5000
```

Write new data points to the output of the 1-D look-up table using `writeCharacteristic`.

```
writeCharacteristic(xcpCh, "ydata", [0 0 0 0 0.2 0.5 1]);
```

Read the new 1-D look-up table data using `readAxis` and `readCharacteristic`, then plot the mapping. Now the table effectively maps any negative input value to zero output.

```
inputBreakpoints = readAxis(xcpCh, "xdata")
```

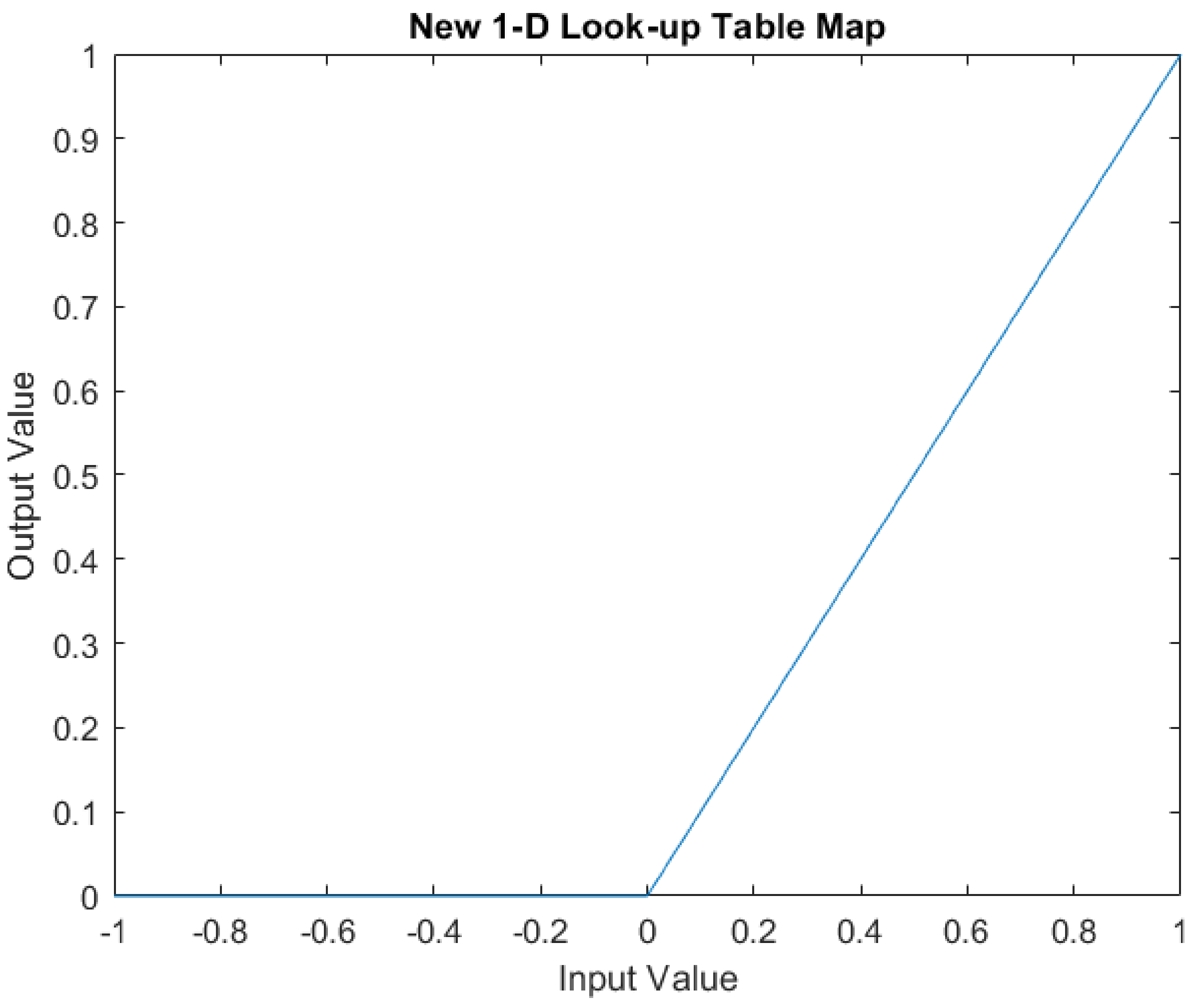
```
inputBreakpoints = 1×7
```

```
   -1.0000   -0.5000   -0.2000         0    0.2000    0.5000    1.0000
```

```
newOutputPoints = readCharacteristic(xcpCh, "ydata")
```

```
newOutputPoints = 1×7

         0         0         0         0    0.2000    0.5000    1.0000
```

```
plot(inputBreakpoints, newOutputPoints);
title("New 1-D Look-up Table Map");
xlabel("Input Value");
ylabel("Output Value");
```



Obtain Measurements after Calibration

Use the `startMeasurement` function and `stopMeasurement` function to run the DAQ list for a short period of time.

```
startMeasurement(xcpCh);
pause(3);
stopMeasurement(xcpCh);
```

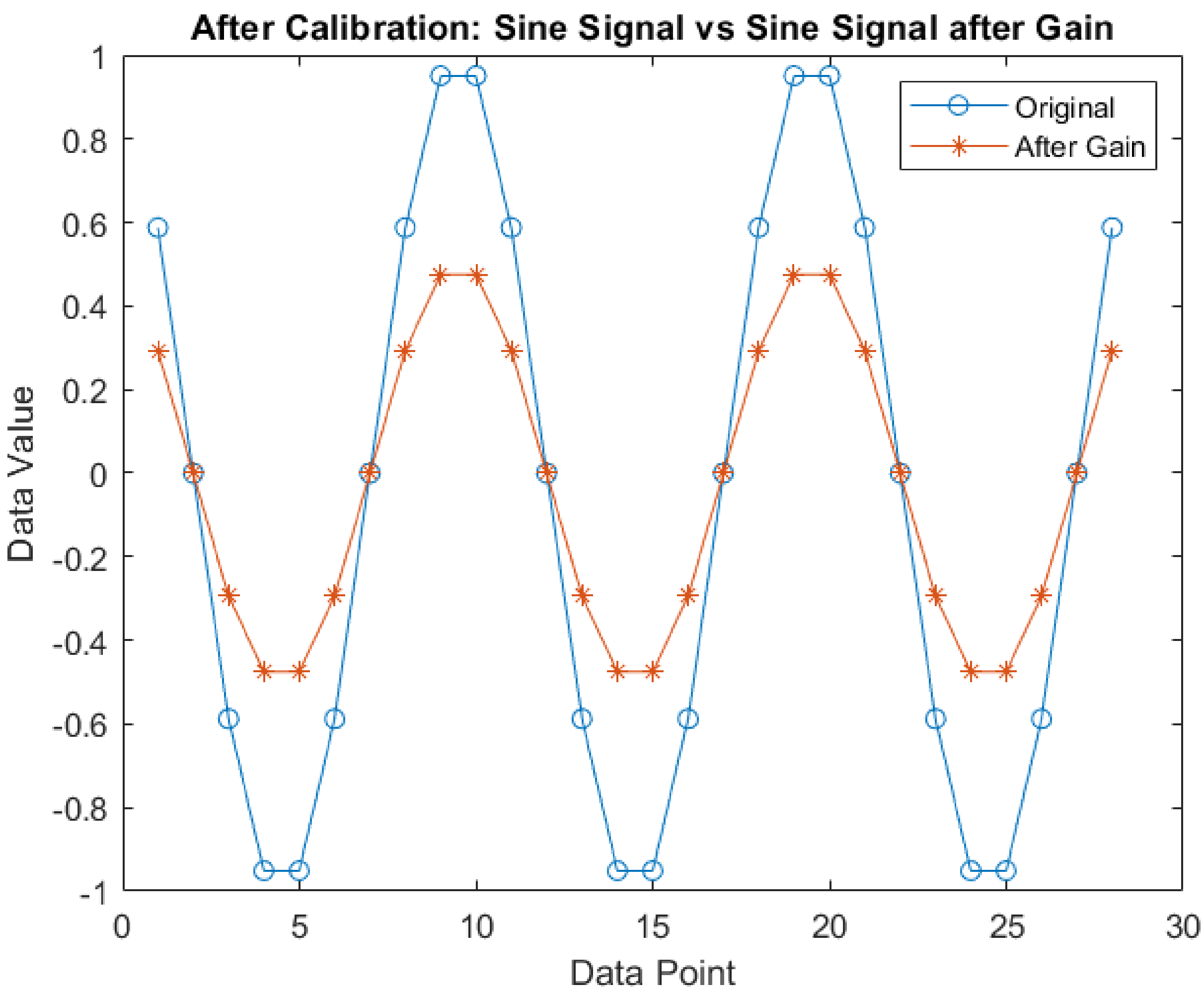
To retrieve the data acquired by the DAQ list for all the Sine-based measurements, use the `readDAQ` function.

```
sine = readDAQ(xcpCh, "Sine");
sineAfterGain = readDAQ(xcpCh, "SineAfterGain");
sineAfterTable = readDAQ(xcpCh, "SineAfterTable");
```

Inspect Measurements After Calibration

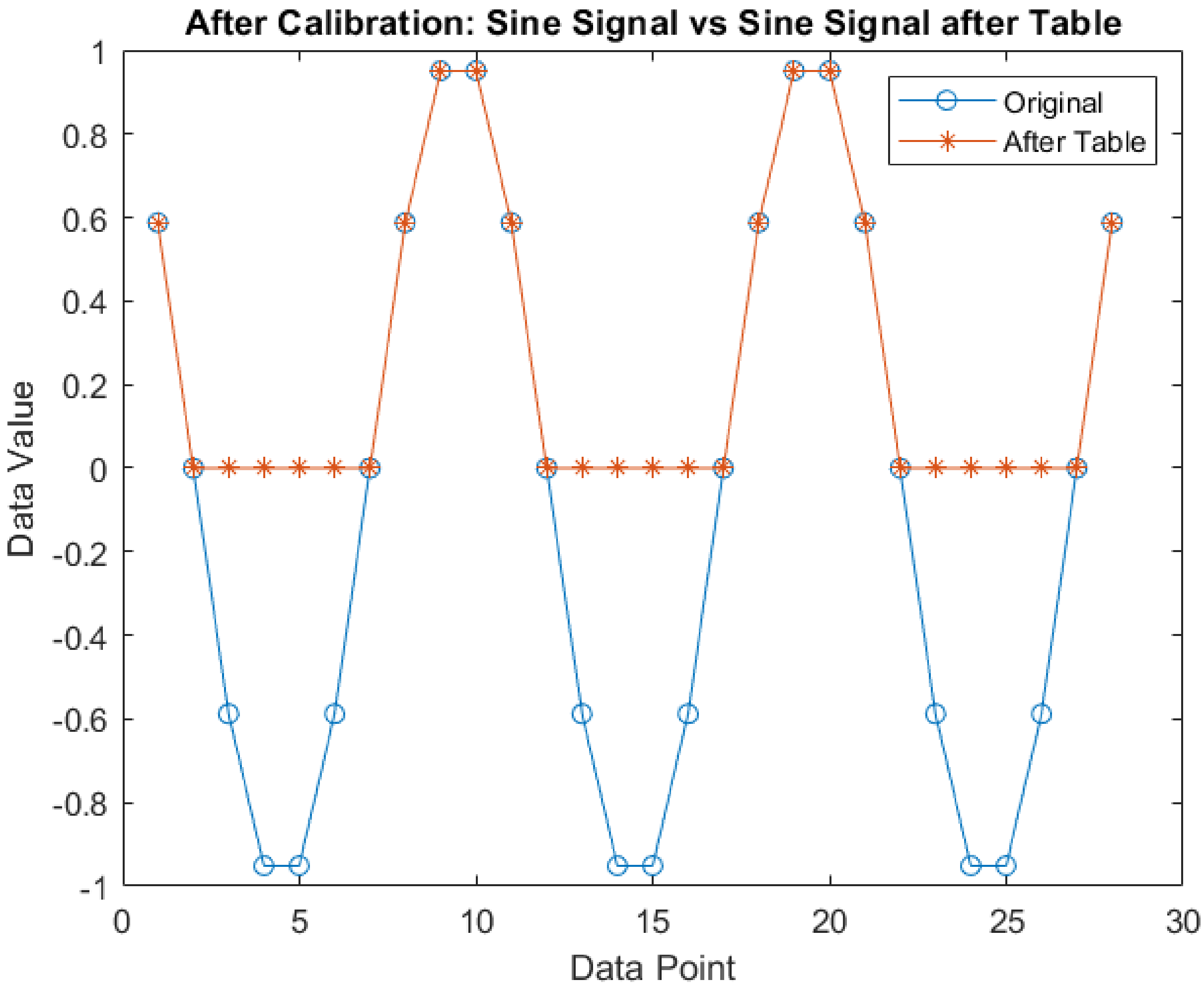
Plot the `SineAfterGain` measurement against the base `Sine` measurement. Now the value after `Gain` is decreased by a factor of 2, based on the original measurement, because the value of the characteristic `Gain` is set to 0.5 after calibration.

```
plot(sine, "o-"); hold on;
plot(sineAfterGain, "*-"); hold off;
title("After Calibration: Sine Signal vs Sine Signal after Gain");
legend("Original", "After Gain");
xlabel("Data Point");
ylabel("Data Value");
```



Plot the `SineAfterTable` measurement against the base `Sine` measurement. Any negative value of the original measurement is mapped to zero according to the new 1-D look-up table, therefore the modified signal looks truncated differently and does not have any negative values.

```
plot(sine, "o-"); hold on;
plot(sineAfterTable, "*-"); hold off;
title("After Calibration: Sine Signal vs Sine Signal after Table");
legend("Original", "After Table");
xlabel("Data Point");
ylabel("Data Value");
```



Disconnect from the Server

To deactivate communication with the server, use the `disconnect` function. The XCP server can be safely closed after disconnecting.

```
disconnect(xcpCh)
```

Clean Up

```
clear a2lInfo
```

Get Started with A2L-Files

This example shows how to access and view information stored in A2L-files.

XCP (Universal Measurement and Calibration Protocol) is a network protocol commonly used in the automotive industry for connecting calibration systems to electronic control units (ECUs). The calibration system is commonly referred to as the client and the ECU as the server. XCP enables read and write access to variables and memory contents at runtime.

Entire datasets can be acquired or stimulated synchronous to events triggered by timers or operating conditions. The XCP protocol specification is defined by ASAM (Association for Standardization of Automation and Measuring Systems), and allows for a variety of transport layers such as XCP over CAN or Ethernet.

An A2L-file is a structured ASCII text file that contains measurement, calibration, and event definitions used with XCP for acquiring and stimulating data. This example uses an A2L-file configured for XCP over Ethernet. An A2L-file follows the ASAM MCD-2 MC standard (ASAP2), which defines the description format of internal server variables used in measurement and calibration. The `.a2l` file extension is an abbreviation of "ASAM MCD-2 MC Language."

Open an A2L-File

An A2L-file contains measurement, calibration, and event definitions for one or more ECUs. If you intend to read data from or write data directly to memory of an XCP server, a necessary first step is to open the A2L-file representing that system. To access an A2L-file, create a file object in your MATLAB session using the `xcpA2L` function:

```
a2lfile = xcpA2L("XCPServerSineWaveGenerator.a2l")
```

```
a2lfile = 
  A2L with properties:

   File Details
                 FileName: 'XCPServerSineWaveGenerator.a2l'
                 FilePath: 'C:\examplefiles\XCPServerSineWaveGenerator.a2l'
               ServerName: 'ModuleName'
                 Warnings: [0×0 string]

   Parameter Details
                   Events: {'100 ms'}
                EventInfo: [1×1 xcp.a2l.Event]
             Measurements: {'Sine'  'SineAfterGain'  'SineAfterTable'  'XCPServer_DW.lastCos'  'X
          MeasurementInfo: [6×1 containers.Map]
          Characteristics: {'Gain'  'ydata'}
       CharacteristicInfo: [2×1 containers.Map]
                 AxisInfo: [1×1 containers.Map]
            RecordLayouts: [4×1 containers.Map]
             CompuMethods: [3×1 containers.Map]
                CompuTabs: [0×1 containers.Map]
               CompuVTabs: [0×1 containers.Map]

   XCP Protocol Details
        ProtocolLayerInfo: [1×1 xcp.a2l.ProtocolLayer]
                  DAQInfo: [1×1 xcp.a2l.DAQ]
    TransportLayerCANInfo: [0×0 xcp.a2l.XCPonCAN]
```

```
    TransportLayerUDPInfo: [0×0 xcp.a2l.XCPonIP]
    TransportLayerTCPInfo: [1×1 xcp.a2l.XCPonIP]
```

Access Measurement Information

A measurement describes the properties of a recordable, server-internal variable. This variable can be a scalar or an array. Bit masks and bit operations can be applied to the measurement. The address, byte order, computation method, upper and lower limits, and other properties are described. The standard also allows writing to measurement objects to stimulate the server during runtime.

View all available measurements via the `Measurements` property of the A2L-file object.

```
a2lfile.Measurements
```

```
ans = 1×6 cell
    {'Sine'}    {'SineAfterGain'}    {'SineAfterTable'}    {'XCPServer_DW.lastCos'}    {'XCPServe
```

Get information about the `Sine` measurement using the `getMeasurementInfo` function. This function returns information about the specified measurement from the specified A2L-file.

```
measInfo = getMeasurementInfo(a2lfile,"Sine")
```

```
measInfo = 
  Measurement with properties:
                   Name: 'Sine'
         LongIdentifier: 'Sine wave signal'
            LocDataType: FLOAT64_IEEE
             Conversion: [1×1 xcp.a2l.CompuMethod]
             Resolution: 0
               Accuracy: 0
             LowerLimit: -3
             UpperLimit: 3
              Dimension: 1
              ArraySize: []
                BitMask: []
           BitOperation: [1×0 xcp.a2l.BitOperation]
              ByteOrder: MSB_LAST
               Discrete: []
             ECUAddress: 1586712
    ECUAddressExtension: 0
                 Format: ''
                 Layout: ROW_DIR
               PhysUnit: ''
              ReadWrite: []
```

Using an `xcpChannel` you can read and write measurement data directly to memory of an XCP server with the `readMeasurement` and `writeMeasurement` functions, respectively. The `readMeasurement` function reads and scales a value for the specified `measurement` through the XCP channel object. This action performs a direct read from memory of the server. The `writeMeasurement` function scales and writes a value for the specified `measurement` through the XCP channel object. This action performs a direct write to memory of the server.

Access Characteristic Information

A characteristic describes the properties of a tunable parameter (Calibration). Possible types of tunable parameters include scalars, strings, and lookup tables. The address, record layout, computation method, upper and lower calibration limits are defined.

View all available characteristics by name via the `Characteristics` property of the A2L-file object.

```
a2lfile.Characteristics
```

```
ans = 1×2 cell
    {'Gain'}    {'ydata'}
```

Get information about the `Gain` characteristic using the `getCharacteristicInfo` function. This function returns information about the specified characteristic from the specified A2L-file.

```
charInfo = getCharacteristicInfo(a2lfile,"Gain")
```

```
charInfo = 
  Characteristic with properties:
                 Name: 'Gain'
       LongIdentifier: ''
   CharacteristicType: VALUE
           ECUAddress: 549960
              Deposit: [1×1 xcp.a2l.RecordLayout]
              MaxDiff: 0
           Conversion: [1×1 xcp.a2l.CompuMethod]
           LowerLimit: -5
           UpperLimit: 5
            Dimension: 1
       AxisConversion: {1×0 cell}
              BitMask: []
            ByteOrder: MSB_LAST
             Discrete: []
   ECUAddressExtension: 0
               Format: ''
               Number: []
             PhysUnit: ''
```

Using an `xcpChannel` you can read and write characteristic data directly to memory of an XCP server using the `readCharacteristic` and `writeCharacteristic` functions, respectively. The `readCharacteristic` function reads and scales a value for the specified `characteristic` through the XCP channel. This action performs a direct read from memory of the server. The `writeCharacteristic` function scales and writes a value for the specified `characteristic` through the XCP channel object. This action performs a direct write to memory of the server.

Access Event Information

Data can be acquired or stimulated synchronous to events triggered by timers or operating conditions.

View all available events via the `Events` property of the A2L-file object.

```
a2lfile.Events
```

```
ans = 1×1 cell array
    {'100 ms'}
```

Get information about the `100 ms` event using the `getEventInfo` function. This function returns information about the specified event from the specified A2L-file.

```
eventInfo = getEventInfo(a2lfile, "100 ms")
```

```
eventInfo = 
  Event with properties:
                         Name: '100 ms'
                    ShortName: '100 ms'
                ChannelNumber: 0
                    Direction: DAQ
                   MaxDAQList: 255
             ChannelTimeCycle: 1
              ChannelTimeUnit: 8
              ChannelPriority: 0
    ChannelTimeCycleInSeconds: 0.1000
```

Using an `xcpChannel` and specifying an event, you can acquire and stimulate measurements using the available XCP functions, such as `readDAQ and writeSTIM`. The use of events to acquire measurement data is further explored in the example "Read XCP Measurements with Dynamic DAQ Lists" on page 14-247.

View Protocol Layer Information

The protocol layer defines some of the core operation and organization of the messaging between the XCP server and client. This includes the sizing and structure of the bytes in XCP command and response messages.

Display protocol layer details via the `ProtocolLayerInfo` property of the A2L-file object.

```
a2lfile.ProtocolLayerInfo
```

```
ans = 
  ProtocolLayer with properties:
                    T1: 1000
                    T2: 200
                    T3: 0
                    T4: 0
                    T5: 0
                    T6: 0
                    T7: 0
                MaxCTO: 255
                MaxDTO: 65532
             ByteOrder: BYTE_ORDER_MSB_LAST
    AddressGranularity: ADDRESS_GRANULARITY_BYTE
```

View DAQ Information

XCP offers the synchronous data acquisition (DAQ) mode, as described in ASAM MDC-2 MC. DAQ is one of the main XCP services that a server can provide. XCP DAQ events can be defined by the client to trigger the sampling of measurement data. When the algorithm in the server reaches the location of such a sampling event, the server collects the values of the measurement parameters and sends them to the client. Display DAQ details via the `DAQInfo` property of the A2L-file object.

```
a2lfile.DAQInfo
```

```
ans = 
  DAQ with properties:
                    ConfigType: DYNAMIC
                        MaxDAQ: 65535
              MaxEventChannels: 128
                        MinDAQ: 0
              OptimizationType: OPTIMISATION_TYPE_DEFAULT
              AddressExtension: ADDRESS_EXTENSION_FREE
       IdentificationFieldType: IDENTIFICATION_FIELD_TYPE_ABSOLUTE
    GranularityODTEntrySizeDAQ: GRANULARITY_ODT_ENTRY_SIZE_DAQ_BYTE
            MaxODTEntrySizeDAQ: 255
            OverloadIndication: NO_OVERLOAD_INDICATION
       DAQAlternatingSupported: []
            PrescalerSupported: []
               ResumeSupported: []
                          STIM: [1×0 xcp.a2l.STIM]
                     Timestamp: [1×1 xcp.a2l.TimestampSupported]
                        Events: [1×1 xcp.a2l.Event Map]
```

View Transport Layer Information

The XCP packet is embedded in a frame of the transport layer, which is a packet of the chosen transport protocol. An A2L-file provides transport layer information for the supported protocols. If the transport layer information for a particular protocol is empty, the server does not support that transport. The XCP protocol specification allows for a variety of transport layers, such as CAN or Ethernet.

This example uses an A2L-file configured for XCP over Ethernet, which requires an IP address and a port. These are specified in the A2L-file.

Display transport layer details via the `TransportLayerTCPInfo` property of the A2L-file object.

```
a2lfile.TransportLayerTCPInfo
```

```
ans = 
  XCPonIP with properties:
          CommonParameters: [1×1 xcp.a2l.CommonParameters]
    TransportLayerInstance: ''
                      Port: 17725
                   Address: 2.1307e+09
             AddressString: '127.0.0.1'
```

Close the A2L-File

Close access to the A2L-file by clearing its variable from the workspace.

```
clear a2lfile
```

Analyze Data Using MDF Datastore and Tall Arrays

This example shows how to work with a big data set using tall arrays and the MDF datastore feature. Tall arrays are commonly used to perform calculations on different types of data that do not fit in memory.

This example first operates on a small subset of data and then scales up to analyze the entire data set. Although the data set used here might not represent the actual size in real-world applications, the same analysis technique can scale up further to work on data sets so large that they cannot be read into memory.

To learn more about tall arrays, see the example "Analyze Big Data in MATLAB Using Tall Arrays".

Introduction to Tall Arrays

Tall arrays and tall tables are used to work with out-of-memory data that has any number of rows. Using tall arrays and tables, you can work with large data sets in a manner similar to in-memory MATLAB arrays.

The difference is that tall arrays typically remain unevaluated until the calculations are requested to be performed. This deferred evaluation enables MATLAB to combine the queued calculations where possible and take the minimum number of passes through the data.

Create an MDF Datastore

An MDF datastore can be used to read and process homogeneous data stored in multiple MDF-files as a single entity. If the data set is too large to fit in memory, a datastore also makes it possible to work with the data set in smaller blocks that individually fit in memory. This capability can be further extended by tall arrays which enable working with out-of-memory data backed up by a datastore using common functions.

Create an MDF datastore using the `mdfDatastore` function by selecting MDF-file `EngineData_MDF_TallArray.mf4` in the current workflow directory. This file contains time-stamped data logged from a Simulink model representing an engine plant and controller connected to a dynamometer.

```
mds = mdfDatastore("EngineData_MDF_TallArray.mf4")
```

```
mds = 
  MDFDatastore with properties:

  DataStore Details
                     Files: {
                            ' ...\Documents\MATLAB\Examples\vnt-ex08773747\EngineData_MDF_Ta
                            }
             ChannelGroups:
                              ChannelGroupNumber    AcquisitionName    Comment          ...
                              __________________    _______________    __________

                                      1               {1×1 cell}       {1×1 cell}


                  Channels:
                              ChannelGroupNumber      ChannelName       DisplayName
                              __________________    _________________   ___________
```

```
                       1             {'EngineSpeed'  }        ''
                       1             {'TorqueCommand'}        ''
                       1             {'EngineTorque' }        ''

             ... and 1 more rows

  Options
        SelectedChannelNames: {
                              'EngineSpeed';
                              'TorqueCommand';
                              'EngineTorque'
                               ... and 1 more
                              }
    SelectedChannelGroupNumber: 1
                    ReadSize: 'file'
                  Conversion: Numeric
```

It is possible to further configure the MDF datastore to control what and how data is read from the MDF-file. By default, the first channel group is selected and all channels from the group are read.

```
mds.SelectedChannelGroupNumber
```

```
ans = 1
```

```
mds.SelectedChannelNames
```

```
ans = 4×1 string
    "EngineSpeed"
    "TorqueCommand"
    "EngineTorque"
    "t"
```

Configure the MDF datastore to select only three variables of interest: `EngineSpeed`, `TorqueCommand`, and `EngineTorque`.

```
mds.SelectedChannelNames = ["EngineSpeed", "TorqueCommand", "EngineTorque"]
```

```
mds = 
  MDFDatastore with properties:

  DataStore Details
                       Files: {
                              ' ...\Documents\MATLAB\Examples\vnt-ex08773747\EngineData_MDF_Ta
                              }
               ChannelGroups: 
                                ChannelGroupNumber    AcquisitionName    Comment          ...
                                __________________    _______________    __________

                                        1               {1×1 cell}       {1×1 cell}

                    Channels: 
                                ChannelGroupNumber      ChannelName       DisplayName
                                __________________    _________________   ___________

                                        1             {'EngineSpeed'  }        ''
                                        1             {'TorqueCommand'}        ''
```

```
                                        1            {'EngineTorque' }        ''

                               ... and 1 more rows

  Options
        SelectedChannelNames: {
                               'EngineSpeed';
                               'TorqueCommand';
                               'EngineTorque'
                               }
    SelectedChannelGroupNumber: 1
                     ReadSize: 'file'
                   Conversion: Numeric
```

Preview the selected data using the `preview` function.

```
preview(mds)
```

```
ans=8×3 timetable
         Time          EngineSpeed    TorqueCommand    EngineTorque
    ______________    ___________    _____________    ____________

    0 sec                       0                0          47.153
    0 sec                2.37e-26                0          47.153
    1.47e-05 sec          0.11056           47.158          47.158
    8.85e-05 sec          0.66312           48.708          48.708
    0.00010107 sec        0.75762            49.77           49.77
    0.00010107 sec        0.75762            49.77           49.77
    0.0001405 sec           1.053           39.967          39.967
    0.00017993 sec         1.3482           23.143          23.143
```

Create Tall Array

Tall arrays are similar to in-memory MATLAB arrays, except that they can have any number of rows. Because the MDF datastore `mds` contains time-stamped tabular data, the `tall` function returns a tall timetable containing data from the datastore.

```
tt = tall(mds)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

tt =

  M×3 tall timetable

         Time          EngineSpeed    TorqueCommand    EngineTorque
    ______________    ___________    _____________    ____________

    0 sec                       0                0          47.153
    0 sec                2.37e-26                0          47.153
    1.47e-05 sec          0.11056           47.158          47.158
    8.85e-05 sec          0.66312           48.708          48.708
    0.00010107 sec        0.75762            49.77           49.77
    0.00010107 sec        0.75762            49.77           49.77
    0.0001405 sec           1.053           39.967          39.967
```

```
    0.00017993 sec       1.3482         23.143           23.143
          :                 :              :                :
          :                 :              :                :
```

The display includes the first several rows of data. The timetable size may display as `M×3` to indicate that the number of rows is not yet known to MATLAB.

Perform Calculations on Tall Array

You can work with tall arrays and tall tables similar to in-memory MATLAB arrays and tables. However, MATLAB does not perform most operations on tall arrays, and defers the actual computations until the output is requested.

It is common to work with unevaluated tall arrays and request output only when required. MATLAB does not know the content or size of an unevaluated tall array until you request that it be evaluated and displayed.

Calculate median, minimum, and maximum values of the `TorqueCommand` variable. Note that the results are not immediately evaluated.

```
medianTorqueCommand = median(tt.TorqueCommand)
```

```
medianTorqueCommand =

  tall double

    ?

Preview deferred. Learn more.
```

```
minTorqueCommand = min(tt.TorqueCommand)
```

```
minTorqueCommand =

  tall double

    ?

Preview deferred. Learn more.
```

```
maxTorqueCommand = max(tt.TorqueCommand)
```

```
maxTorqueCommand =

  tall double

    ?

Preview deferred. Learn more.
```

Gather Results into Workspace

The `gather` function forces evaluation of all queued operations and brings the resulting output back into memory.

Perform the queued operations, `median`, `min`, `max`, and evaluate the answers. If the calculation requires several passes through the data, MATLAB determines the minimum number of passes to save execution time and displays this information at the command line.

```
[medianTorqueCommand, minTorqueCommand, maxTorqueCommand] = gather(medianTorqueCommand, minTorque
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 4: Completed in 6.7 sec
- Pass 2 of 4: Completed in 0.73 sec
- Pass 3 of 4: Completed in 1.3 sec
- Pass 4 of 4: Completed in 0.62 sec
Evaluation completed in 12 sec

medianTorqueCommand = 116.2799

minTorqueCommand = 0

maxTorqueCommand = 232.9807
```

Select Subset of Tall Array

Use head to select a subset of 10,000 rows from the data for prototyping code before scaling to the full data set.

```
ttSubset = head(tt, 10000)
```

```
ttSubset =

  10,000×3 tall timetable

        Time         EngineSpeed    TorqueCommand    EngineTorque
    ______________    ___________    _____________    ____________

    0 sec                      0                0         47.153
    0 sec               2.37e-26                0         47.153
    1.47e-05 sec         0.11056           47.158         47.158
    8.85e-05 sec         0.66312           48.708         48.708
    0.00010107 sec       0.75762            49.77          49.77
    0.00010107 sec       0.75762            49.77          49.77
    0.0001405 sec          1.053           39.967         39.967
    0.00017993 sec        1.3482           23.143         23.143
          :                 :                :              :
          :                 :                :              :
```

Remove Duplicate Rows in Tall Array

Timetable rows are duplicates if they have the same row times and the same data values. Use the unique function to remove duplicate rows from the subset tall timetable.

```
ttSubset = unique(ttSubset)
```

```
ttSubset =

  9,968×3 tall timetable

        Time         EngineSpeed    TorqueCommand    EngineTorque
    ______________    ___________    _____________    ____________

    0 sec                      0                0          47.153
    0 sec               2.37e-26                0          47.153
    1.47e-05 sec         0.11056           47.158          47.158
    8.85e-05 sec         0.66312           48.708          48.708
    0.00010107 sec       0.75762            49.77           49.77
```

```
    0.0001405 sec         1.053          39.967            39.967
    0.00017993 sec       1.3482          23.143            23.143
    0.00037708 sec       2.8228          23.143         -0.021071
          :                 :               :                 :
          :                 :               :                 :
```

Calculate Engine Power

Calculate engine power in kilowatts (kW) with `EngineSpeed` and `EngineTorque` using the formula $P\,[\mathrm{kW}] = \frac{\pi \cdot N\,[\mathrm{rpm}] \cdot T\,[\mathrm{Nm}]}{30 \cdot 1000}$. Save the results to a new variable named `EnginePower` in the tall timetable.

```
ttSubset.EnginePower = (pi * ttSubset.EngineSpeed .* ttSubset.EngineTorque) / (30 * 1000)
```

```
ttSubset =

  9,968×4 tall timetable

         Time           EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    ______________    ___________    _____________    ____________    ___________

    0 sec                       0               0          47.153                0
    0 sec                2.37e-26               0          47.153       1.1703e-28
    1.47e-05 sec          0.11056          47.158          47.158       0.00054599
    8.85e-05 sec          0.66312          48.708          48.708        0.0033824
    0.00010107 sec        0.75762           49.77           49.77        0.0039487
    0.0001405 sec           1.053          39.967          39.967        0.0044072
    0.00017993 sec         1.3482          23.143          23.143        0.0032675
    0.00037708 sec         2.8228          23.143       -0.021071      -6.2287e-06
          :                  :               :               :                :
          :                  :               :               :                :
```

The `topkrows` function for tall arrays returns the top `k` rows in sorted order. Obtain the top 20 rows with maximum `EnginePower` values.

```
maxEnginePower = topkrows(ttSubset, 20, "EnginePower")
```

```
maxEnginePower =

  20×4 tall timetable

      Time       EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    _________    ___________    _____________    ____________    ___________

    15.17 sec        750           78.052           78.052         6.1302
    15.16 sec        750           77.841           77.841         6.1136
    15.15 sec        750           77.556           77.556         6.0912
    15.14 sec        750           77.326           77.326         6.0732
    15.18 sec        750           77.277           77.277         6.0693
    15.13 sec        750           77.157           77.157         6.0599
    15.12 sec        750           77.082           77.082          6.054
    15.11 sec        750           77.067           77.075         6.0534
        :             :               :                :               :
        :             :               :                :               :
```

Call the `gather` function to execute all queued operations and collect the results into memory.

```
[ttSubset, maxEnginePower] = gather(ttSubset, maxEnginePower)
```

```
ttSubset=9968×4 timetable
        Time          EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    ______________    ___________    _____________    ____________    ___________

    0 sec                       0                0          47.153              0
    0 sec                 2.37e-26               0          47.153     1.1703e-28
    1.47e-05 sec          0.11056           47.158          47.158     0.00054599
    8.85e-05 sec          0.66312           48.708          48.708      0.0033824
    0.00010107 sec        0.75762            49.77           49.77      0.0039487
    0.0001405 sec           1.053           39.967          39.967      0.0044072
    0.00017993 sec         1.3482           23.143          23.143      0.0032675
    0.00037708 sec         2.8228           23.143       -0.021071    -6.2287e-06
    0.00076951 sec         5.7492               15       -0.042938    -2.5851e-05
    0.0014014 sec          10.437               15       -0.078013    -8.5265e-05
    0.0023449 sec          17.382               15        -0.13009    -0.00023679
    0.0036773 sec          27.079               15        -0.20304    -0.00057575
    0.0054808 sec              40               15        -0.30067     -0.0012595
    0.0072843 sec          52.691               15        -0.39703     -0.0021907
    0.01 sec               71.373               15        -0.53973     -0.0040341
    0.013562 sec           95.119               15          51.176        0.50976
      ⋮
```

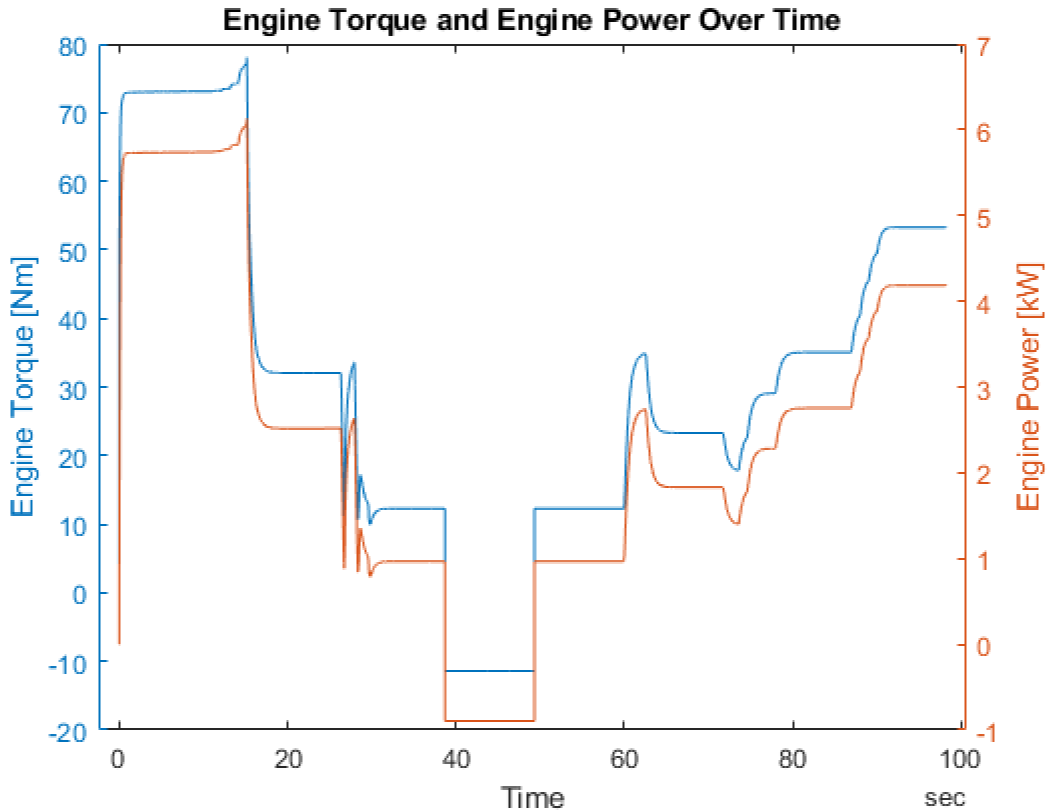
```
maxEnginePower=20×4 timetable
      Time       EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    _________    ___________    _____________    ____________    ___________

    15.17 sec        750           78.052           78.052         6.1302
    15.16 sec        750           77.841           77.841         6.1136
    15.15 sec        750           77.556           77.556         6.0912
    15.14 sec        750           77.326           77.326         6.0732
    15.18 sec        750           77.277           77.277         6.0693
    15.13 sec        750           77.157           77.157         6.0599
    15.12 sec        750           77.082           77.082          6.054
    15.11 sec        750           77.067           77.075         6.0534
    15.1 sec         750           77.067           77.067         6.0528
    15.09 sec        750           77.059           77.059         6.0522
    15.08 sec        750           77.051           77.051         6.0516
    15.07 sec        750           77.042           77.042         6.0509
    15.06 sec        750           77.034           77.034         6.0502
    15.05 sec        750           77.025           77.025         6.0495
    15.04 sec        750           77.016           77.016         6.0488
    15.03 sec        750           77.006           77.006         6.0481
      ⋮
```

Visualize Data in Tall Array

Visualize the `EngineTorque` and `EnginePower` signals over time in a plot with two y-axes.

```
figure
yyaxis left
plot(ttSubset.Time, ttSubset.EngineTorque)
title("Engine Torque and Engine Power Over Time")
xlabel("Time")
ylabel("Engine Torque [Nm]")

yyaxis right
```

```
plot(ttSubset.Time, ttSubset.EnginePower)
ylabel("Engine Power [kW]")
```



Scale to Entire Data Set

Instead of using the smaller data returned from `head`, scale up to apply the same steps on the entire data set by using the complete tall timetable.

```
tt = tall(mds)
```

```
tt =

  M×3 tall timetable

         Time           EngineSpeed    TorqueCommand    EngineTorque
    ______________    ___________    _____________    ____________

    0 sec                        0                0       47.153
    0 sec                 2.37e-26                0       47.153
    1.47e-05 sec           0.11056           47.158       47.158
    8.85e-05 sec           0.66312           48.708       48.708
    0.00010107 sec         0.75762            49.77        49.77
    0.00010107 sec         0.75762            49.77        49.77
    0.0001405 sec            1.053           39.967       39.967
    0.00017993 sec          1.3482           23.143       23.143
          :                  :                :             :
          :                  :                :             :
```

Firstly, remove duplicate rows from the tall timetable.

```
tt = unique(tt)
```

```
tt =

  M×3 tall timetable

    Time    EngineSpeed    TorqueCommand    EngineTorque
    ____    ___________    _____________    ____________

     ?           ?               ?               ?
     ?           ?               ?               ?
     ?           ?               ?               ?
     :           :               :               :
     :           :               :               :

Preview deferred. Learn more.
```

Secondly, calculate engine power and obtain the top 20 rows with maximum `EnginePower` values.

```
tt.EnginePower = (pi * tt.EngineSpeed .* tt.EngineTorque) / (30 * 1000)
```

```
tt =

  M×4 tall timetable

    Time    EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    ____    ___________    _____________    ____________    ___________

     ?           ?               ?               ?               ?
     ?           ?               ?               ?               ?
     ?           ?               ?               ?               ?
     :           :               :               :               :
     :           :               :               :               :

Preview deferred. Learn more.
```

```
maxEnginePower = topkrows(tt, 20, "EnginePower")
```

```
maxEnginePower =

  M×4 tall timetable

    Time    EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    ____    ___________    _____________    ____________    ___________

     ?           ?               ?               ?               ?
     ?           ?               ?               ?               ?
     ?           ?               ?               ?               ?
     :           :               :               :               :
     :           :               :               :               :

Preview deferred. Learn more.
```

```
[tt, maxEnginePower] = gather(tt, maxEnginePower)
```

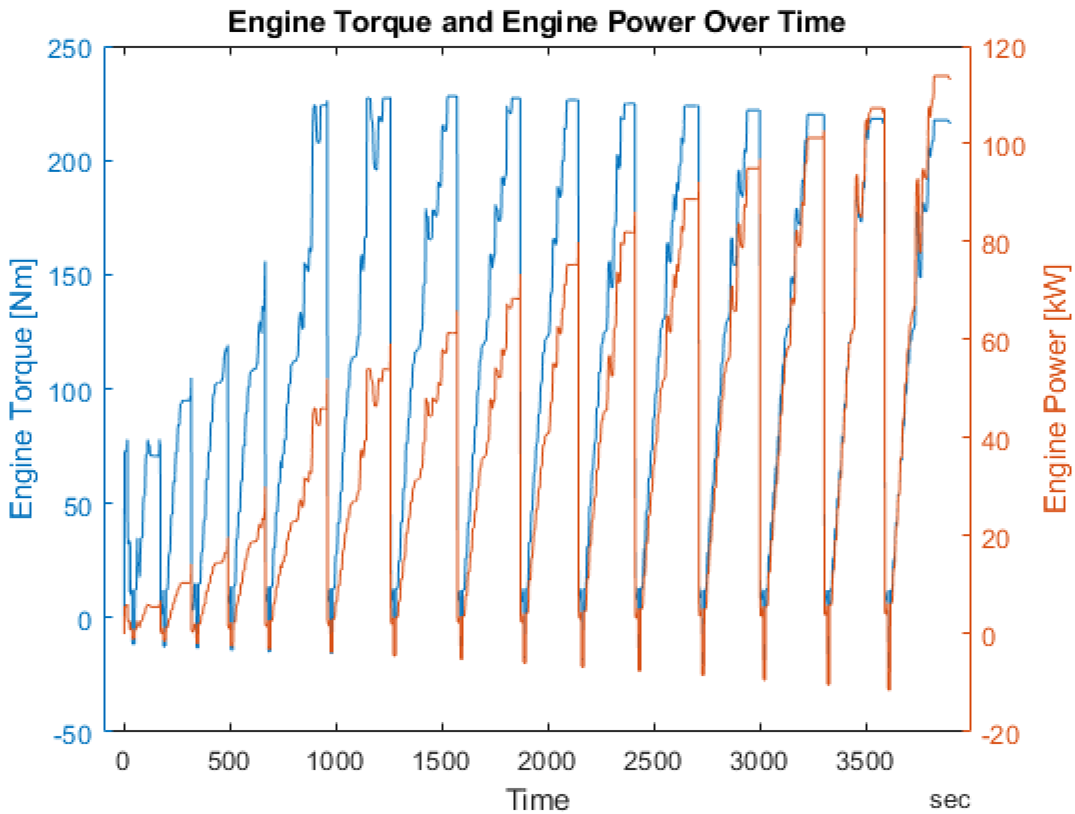
```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
```

```
Evaluation 0% complete

- Pass 1 of 1: Completed in 1.3 sec
Evaluation completed in 1.9 sec

tt=359326×4 timetable
         Time          EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    ______________     ___________    _____________    ____________    ___________

    0 sec                        0                0          47.153              0
    0 sec                 2.37e-26                0          47.153     1.1703e-28
    1.47e-05 sec           0.11056           47.158          47.158     0.00054599
    8.85e-05 sec           0.66312           48.708          48.708      0.0033824
    0.00010107 sec         0.75762            49.77           49.77      0.0039487
    0.0001405 sec            1.053           39.967          39.967      0.0044072
    0.00017993 sec          1.3482           23.143          23.143      0.0032675
    0.00037708 sec          2.8228           23.143       -0.021071    -6.2287e-06
    0.00076951 sec          5.7492               15       -0.042938    -2.5851e-05
    0.0014014 sec           10.437               15       -0.078013    -8.5265e-05
    0.0023449 sec           17.382               15        -0.13009    -0.00023679
    0.0036773 sec           27.079               15        -0.20304    -0.00057575
    0.0054808 sec               40               15        -0.30067     -0.0012595
    0.0072843 sec           52.691               15        -0.39703     -0.0021907
    0.01 sec                71.373               15        -0.53973     -0.0040341
    0.013562 sec            95.119               15          51.176        0.50976
      ⋮


maxEnginePower=20×4 timetable
       Time        EngineSpeed    TorqueCommand    EngineTorque    EnginePower
    __________     ___________    _____________    ____________    ___________

    3819.8 sec        5000           217.53           217.53           113.9
    3819.8 sec        5000           217.53           217.53           113.9
    3819.8 sec        5000           217.53           217.53           113.9
    3819.8 sec        5000           217.53           217.53           113.9
    3819.8 sec        5000           217.53           217.53           113.9
    3819.9 sec        5000           217.53           217.53           113.9
    3819.9 sec        5000           217.53           217.53           113.9
    3819.9 sec        5000           217.53           217.53           113.9
    3819.9 sec        5000           217.52           217.52          113.89
    3819.9 sec        5000           217.52           217.52          113.89
    3820 sec          5000           217.52           217.52          113.89
    3820.1 sec        5000           217.52           217.52          113.89
    3820.2 sec        5000           217.52           217.52          113.89
    3820.3 sec        5000           217.52           217.52          113.89
    3820.4 sec        5000           217.52           217.52          113.89
    3820.5 sec        5000           217.52           217.52          113.89
      ⋮
```

Lastly, visualize the `EngineTorque` and `EnginePower` signals over time in a plot with two y-axes.

```
figure
yyaxis left
plot(tt.Time, tt.EngineTorque)
title("Engine Torque and Engine Power Over Time")
xlabel("Time")
```

```
ylabel("Engine Torque [Nm]")

yyaxis right
plot(tt.Time, tt.EnginePower)
ylabel("Engine Power [kW]")
```



Close MDF-File

Close access to the MDF-file by clearing the MDF datastore variable from workspace.

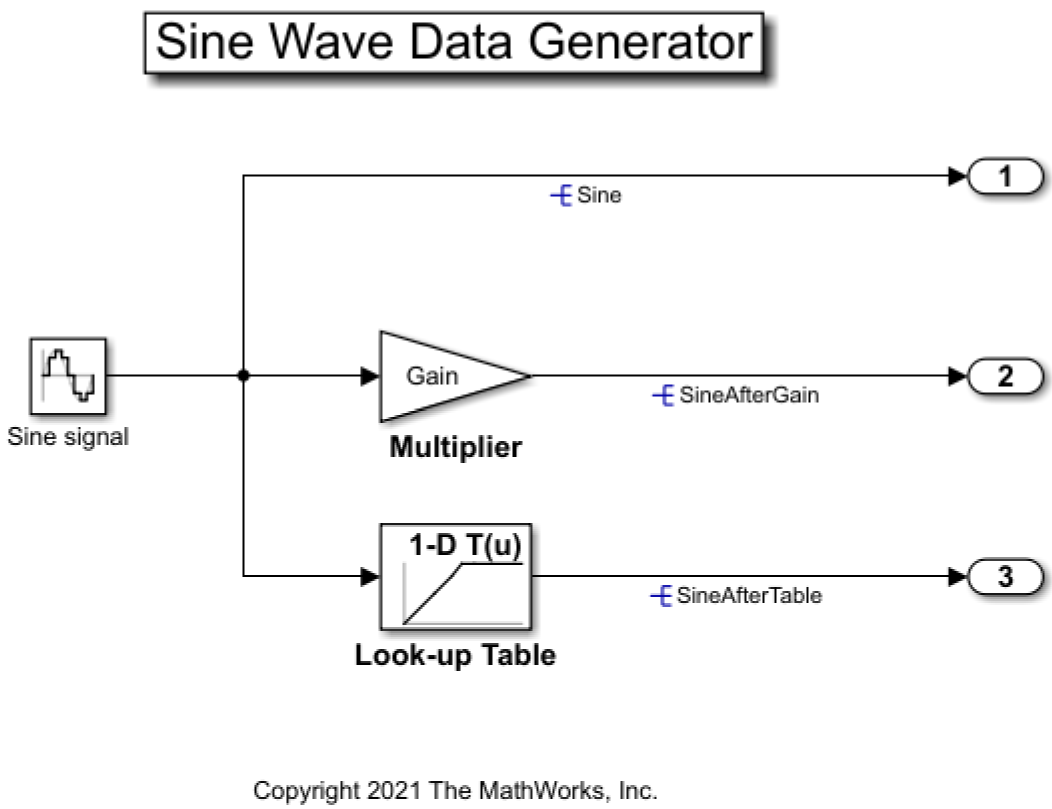
```
clear mds
```

Read XCP Measurements with Dynamic DAQ Lists

This example shows how to use the XCP protocol capability to connect and acquire data from a Simulink model deployed to a Windows executable. The example reads measurement parameters of the model using TCP and dynamic DAQ lists. XCP is a high-level protocol used for accessing and modifying internal parameters and variables of a model, algorithm, or ECU. For more information, refer to the ASAM standards.

Algorithm Overview

The algorithm used in this example is a Simulink model built and deployed as an XCP server. The model has already been compiled and is available to run in the file `XCPServerSineWaveGenerator.exe`. Additionally, the A2L-file `XCPServerSineWaveGenerator.a2l` is provided as an output of that build process. The model contains three measurements and two characteristics accessible via XCP. Because the model is already deployed, Simulink is not required to run this example. The following image illustrates the model.



For details about how to build a Simulink model, including an XCP server and generating an A2L-file, see “Export ASAP2 File for Data Measurement and Calibration” (Simulink Coder).

Run the XCP Server Model

To communicate with the XCP server, the deployed model must be run. By using the `system` function, you can execute the `XCPServerSineWaveGenerator.exe` from inside MATLAB. The function requires constructing an argument list pointing to the executable. A separate command window opens and shows running outputs from the server.

```
sysCommand = ['"', fullfile(pwd, 'XCPServerSineWaveGenerator.exe'),'"', ' &'];
system(sysCommand);
```

Open the A2L-File

An A2L-file is required to establish a connection to the XCP server. The A2L-file describes all of the functionality and capability that the XCP server provides, as well as the details of how to connect to the server. Use the `xcpA2L` function to open the A2L-file that describes the server model.

```
a2lInfo = xcpA2L("XCPServerSineWaveGenerator.a2l")
```

```
a2lInfo = 
  A2L with properties:

   File Details
                 FileName: 'XCPServerSineWaveGenerator.a2l'
                 FilePath: 'C:\Users\kuanliu\Documents\MATLAB\Examples\vnt-ex16421241\XCPServerS
               ServerName: 'ModuleName'
                 Warnings: [0×0 string]

   Parameter Details
                   Events: {'100 ms'}
                EventInfo: [1×1 xcp.a2l.Event]
             Measurements: {'Sine'  'SineAfterGain'  'SineAfterTable'  'XCPServer_DW.lastCos'  'X
          MeasurementInfo: [6×1 containers.Map]
          Characteristics: {'Gain'  'ydata'}
       CharacteristicInfo: [2×1 containers.Map]
                 AxisInfo: [1×1 containers.Map]
            RecordLayouts: [4×1 containers.Map]
             CompuMethods: [3×1 containers.Map]
                CompuTabs: [0×1 containers.Map]
               CompuVTabs: [0×1 containers.Map]

   XCP Protocol Details
        ProtocolLayerInfo: [1×1 xcp.a2l.ProtocolLayer]
                  DAQInfo: [1×1 xcp.a2l.DAQ]
    TransportLayerCANInfo: [0×0 xcp.a2l.XCPonCAN]
    TransportLayerUDPInfo: [0×0 xcp.a2l.XCPonIP]
    TransportLayerTCPInfo: [1×1 xcp.a2l.XCPonIP]
```

TCP is the transport protocol used to communicate with the XCP server. Details for the TCP connection, such as the IP address and port number, are contained in the `TransportLayerTCPInfo` property.

```
a2lInfo.TransportLayerTCPInfo
```

```
ans = 
  XCPonIP with properties:
          CommonParameters: [1×1 xcp.a2l.CommonParameters]
    TransportLayerInstance: ''
```

```
                  Port: 17725
               Address: 2.1307e+09
         AddressString: '127.0.0.1'
```

Create an XCP Channel

To create an active XCP connection to the server, use the `xcpChannel` function. The function requires a reference to the server A2L-file and the type of transport protocol to use for messaging with the server.

```
xcpCh = xcpChannel(a2lInfo, "TCP")
```

```
xcpCh = 
  Channel with properties:

              ServerName: 'ModuleName'
             A2LFileName: 'XCPServerSineWaveGenerator.a2l'
          TransportLayer: 'TCP'
    TransportLayerDevice: [1×1 struct]
              SeedKeyDLL: []
```

Connect to the Server

To make communication with the server active, use the `connect` function.

```
connect(xcpCh)
```

Create and View a Measurement List

A measurement in XCP represents a variable in the memory of the model. Measurements available from the server are defined in the A2L-file. One way to read measurement data is using dynamic DAQ lists. Use the `createMeasurementList` function to create a dynamic DAQ list with a specified event used to trigger the data acquisition and measurements that comprise the list.

```
createMeasurementList(xcpCh, "DAQ", "100 ms", ["Sine", "SineAfterGain", "SineAfterTable"])
```

View configured dynamic DAQ lists using the `viewMeasurementLists` function.

```
viewMeasurementLists(xcpCh)
```

```
DAQ List #1 using the "100 ms" event @ 0.100000 seconds and the following measurements:
   Sine
   SineAfterGain
   SineAfterTable
```

Acquire Data form XCP server

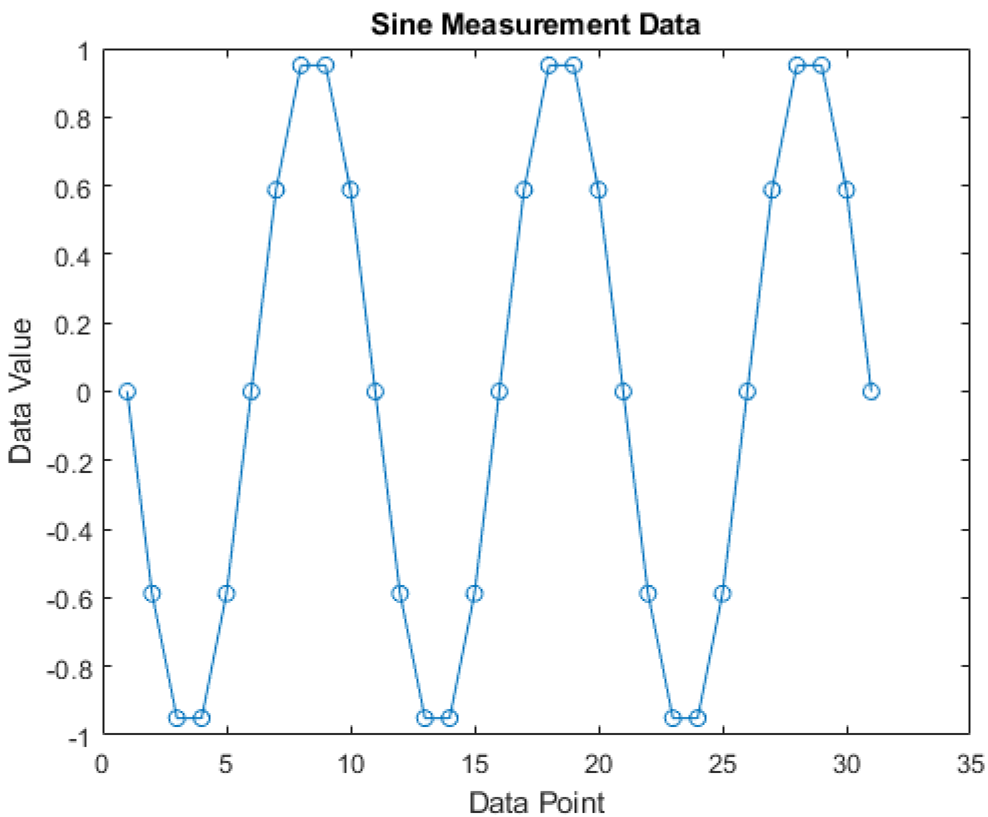
Start the configured dynamic DAQ list using the `startMeasurement` function. It begins the transmission of DAQ data from the server and stores the DAQ data in the XCP channel. After running for a few seconds, stop measurements using the `stopMeasurement` function.

```
startMeasurement(xcpCh)
pause(3);
stopMeasurement(xcpCh)
```

Retrieve the Sine Measurement Data

To retrieve the acquired data from the XCP channel for the `Sine` measurement, use the `readDAQ` function. The function requires a reference to the XCP channel and the specified measurement to read. `readDAQ` returns all available samples held by the XCP channel. Measurement data returned by `readDAQ` is fully scaled using the compute methods defined for the measurement in the A2L-file.

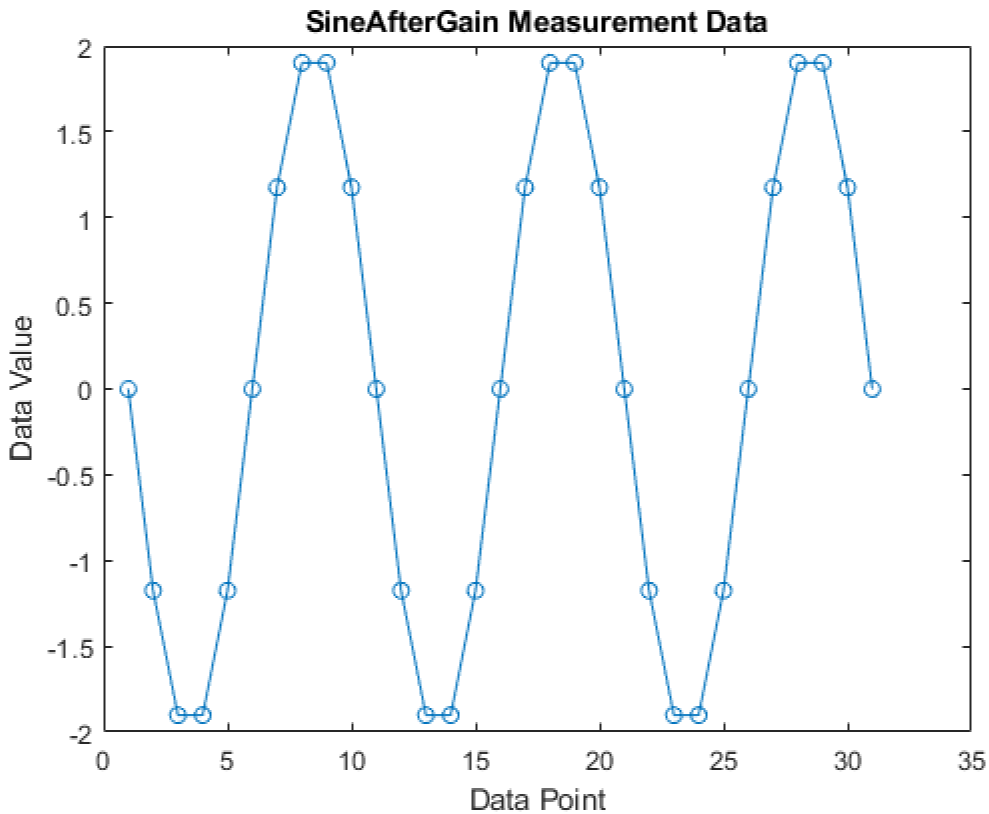
```
dataSine = readDAQ(xcpCh, "Sine");
plot(dataSine, "o-")
title("Sine Measurement Data")
xlabel("Data Point")
ylabel("Data Value")
```



Retrieve the SineAfterGain Measurement Data

To retrieve the acquired data from the XCP channel for the `SineAfterGain` measurement, use the `readDAQ` function.

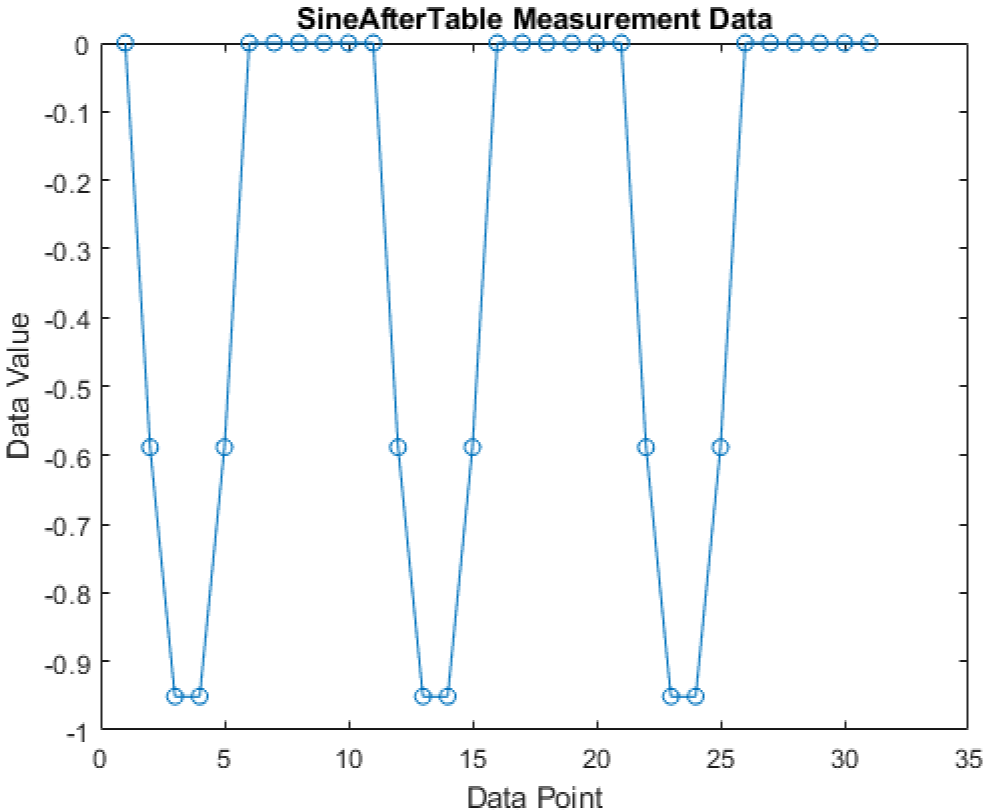
```
dataSineAfterGain = readDAQ(xcpCh, "SineAfterGain");
plot(dataSineAfterGain, "o-")
title("SineAfterGain Measurement Data")
xlabel("Data Point")
ylabel("Data Value")
```



Retrieve the SineAfterTable Measurement Data

To retrieve the acquired data from the XCP channel for the `SineAfterTable` measurement, use the `readDAQ` function.

```
dataSineAfterTable = readDAQ(xcpCh, "SineAfterTable");
plot(dataSineAfterTable, "o-")
title("SineAfterTable Measurement Data")
xlabel("Data Point")
ylabel("Data Value")
```



Disconnect from the Server

To make communication with the server inactive, use the `disconnect` function. The XCP server can be safely closed after disconnecting.

```
disconnect(xcpCh)
```

Clean Up

```
clear a2lInfo
```

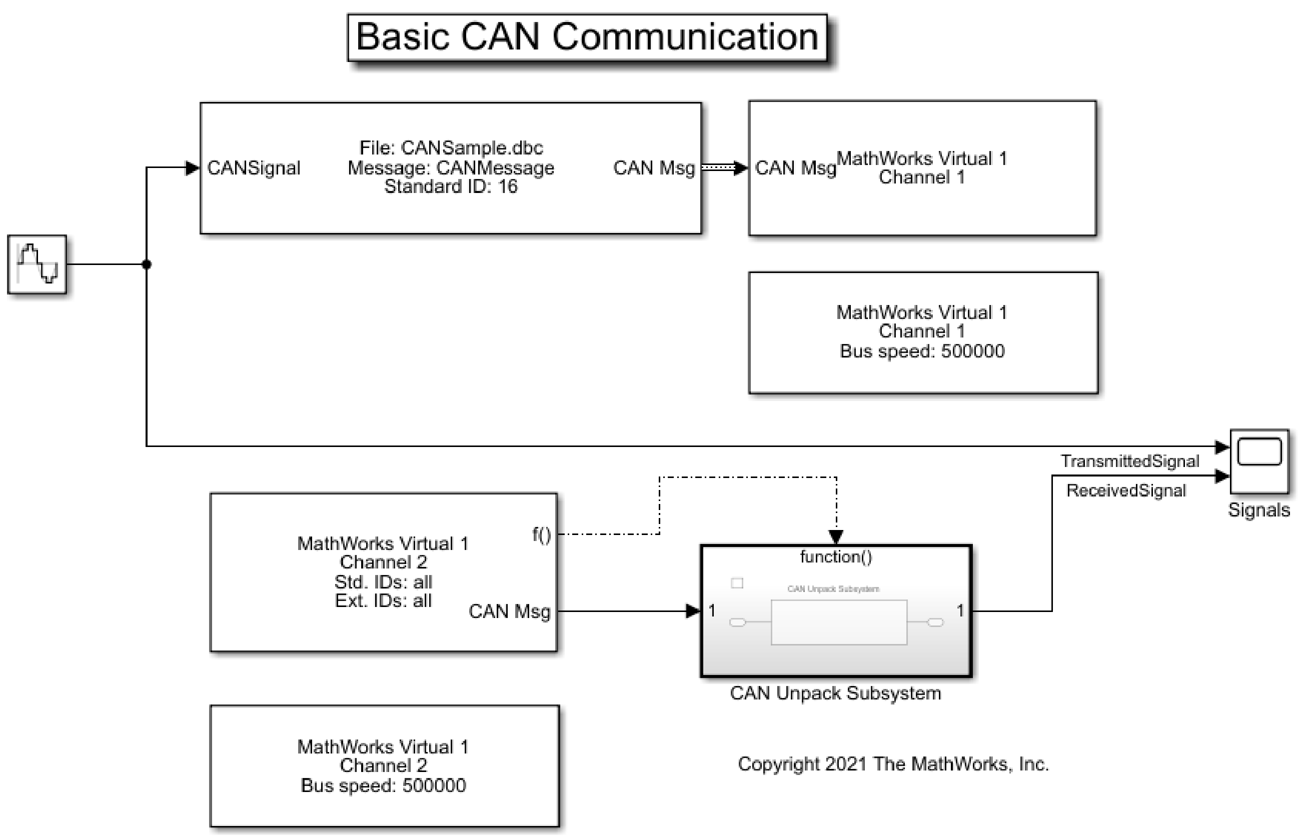
Get Started with CAN Communication in Simulink

This example shows how to use MathWorks virtual CAN channels to set up transmission and reception of CAN messages in Simulink. The virtual channels are connected in a loopback configuration.

Vehicle Network Toolbox provides Simulink blocks for transmitting and receiving live messages via Simulink models over networks using the Controller Area Network (CAN) format. This example uses the CAN Configuration, CAN Pack, CAN Transmit, CAN Receive, and CAN Unpack blocks to perform data transfer over a CAN bus.

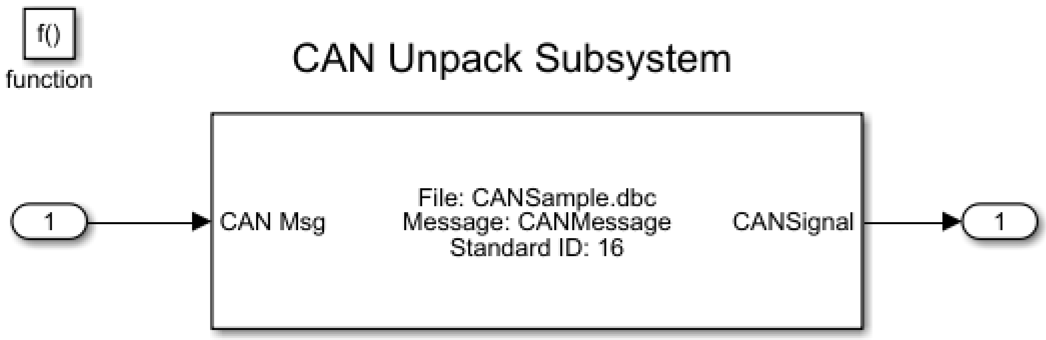
Transmit and Receive CAN Messages

Create a model to transmit and receive a CAN message carrying a sine wave data signal. The model transmits a single message per timestep. A DBC-file defines the message and signal used in the model.



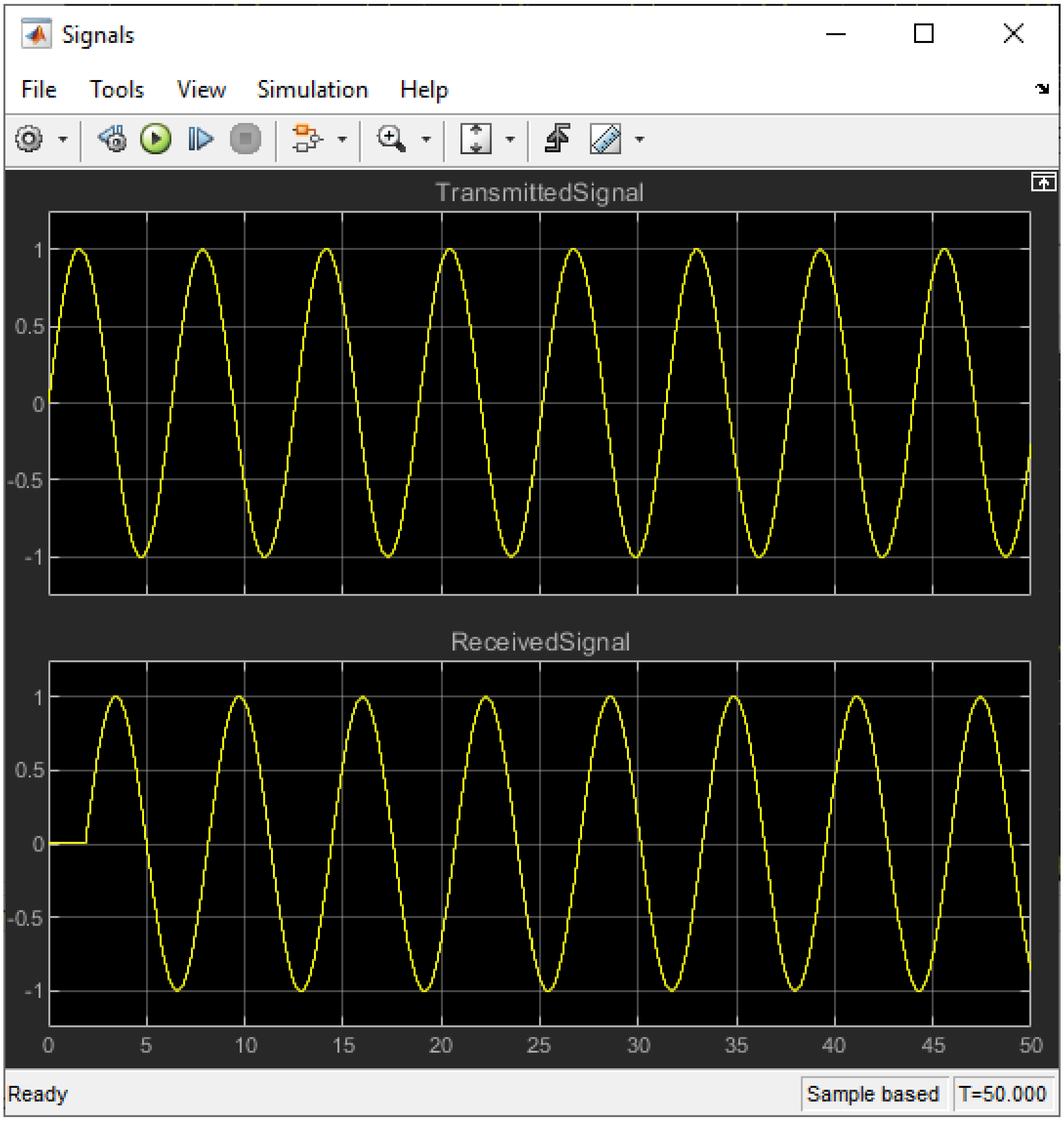
Process CAN Messages

The CAN Receive block generates a function-call trigger if it receives a new message at any particular timestep. This indicates to other blocks in the model that a message is available for decoding activities. Signal decoding and processing is performed inside the Function-Call Subsystem (Simulink).



Visualize Signal Data

Plot the sine wave values before and after transmission. The X-axis corresponds to the simulation timestep and the Y-axis corresponds to the value of the signal. The phase shift between the two plots represents the propagation delay as the signal travels across the network.



Extend the Example

This example uses MathWorks virtual CAN channels. You can connect your models to other supported hardware. You can also modify the model to transmit at periodic rates.

Work with Unfinalized and Unsorted MDF-Files

This example shows how to work with unfinalized and unsorted MDF-files. The unfinalized MDF-file used in this example, `MDFUnfinalized.MF4`, was recorded by a CANedge2 CAN bus data logger from CSS Electronics.

Introduction to Unfinalized and Unsorted MDF-Files

Sometimes an MDF-file creator tool can experience a premature termination caused by an unexpected power-down or an application error. In such cases, the MDF-file might be left in an **unfinalized** state, possibly violating certain format rules of the ASAM MDF standard or causing data loss during read operations.

In general, a data group can be either sorted or unsorted. Some recording tools write unsorted MDF-files without sorting them after the recording completes. A sorted data group cannot contain more than one channel group, while an unsorted data group may contain several channel groups. If all data groups in an MDF-file are sorted, the MDF-file is **sorted**; if at least one data group is unsorted, the entire MDF-file is **unsorted**.

An unfinalized MDF-file can be either sorted or unsorted. Conversely, an unsorted MDF-file can be either finalized or unfinalized.

Use Unfinalized MDF-Files in MATLAB

Because unfinalized files can contain format issues and lead to unreliable read operations, an error is thrown when attempting to open an unfinalized MDF-file using the `mdf` function.

```
try
    m = mdf("MDFUnfinalized.MF4")
catch ME
    disp(ME.message)
end
```

```
Cannot perform operation on unfinalized file. Use mdfFinalize to create a finalized file.
```

You can finalize an unfinalized MDF-file using the function `mdfFinalize`. If the MDF-file is both unfinalized and unsorted, `mdfFinalize` also attempts to sort the file as part of the finalization process.

Use Finalized but Unsorted MDF-Files in MATLAB

If an MDF-file is finalized but unsorted, you can open the file using the `mdf` function, but an error might occur if you subsequently try to read data from the unsorted file using the `read` function.

You can sort a finalized but unsorted MDF-file using function `mdfSort`. If the unsorted MDF-file is also unfinalized, using `mdfSort` on the file causes an error. Instead, use `mdfFinalize` to finalize and sort the file at the same time.

This example continues to demonstrate the use of `mdfFinalize` with unfinalized MDF-files. However, you can follow a similar workflow to use the `mdfSort` function on finalized but unsorted MDF-files.

Finalize an MDF-File In-Place

The `mdfFinalize` function allows you to finalize an unfinalized MDF-file in place by overwriting the source file with a finalized copy.

For demonstration purposes, make a copy of the original file using `copyfile`, and use the extra copy `MDFFinalizedInPlace.MF4` in the subsequent finalization operation.

```
copyfile("MDFUnfinalized.MF4", "MDFFinalizedInPlace.MF4")
```

Use `mdfFinalize` with only the source file name `MDFFinalizedInPlace.MF4` specified to create a finalized copy that overwrites itself. The function returns the full path of the finalized file.

```
finalizedPath1 = mdfFinalize("MDFFinalizedInPlace.MF4")
```

```
finalizedPath1 = 
'C:\Users\michellw\Documents\MATLAB\Examples\vnt-ex16754708\MDFFinalizedInPlace.MF4'
```

`MDFFinalizedInPlace.MF4` is now finalized and can be opened using the `mdf` function. You can specify the full path returned by `mdfFinalize`. Alternatively, specify the file name if it is located on MATLAB path.

```
m1 = mdf(finalizedPath1)
```

```
m1 = 
  MDF with properties:

   File Details
                 Name: 'MDFFinalizedInPlace.MF4'
                 Path: 'C:\Users\michellw\Documents\MATLAB\Examples\vnt-ex16754708\MDFFinalizedI
               Author: ''
           Department: ''
              Project: ''
              Subject: ''
              Comment: ''
              Version: '4.11'
             DataSize: 2596814
     InitialTimestamp: 2021-04-12 10:06:43.000000000

   Creator Details
    ProgramIdentifier: 'CE      '
              Creator: [1×1 struct]

   File Contents
           Attachment: [0×1 struct]
         ChannelNames: {8×1 cell}
         ChannelGroup: [1×8 struct]

   Options
           Conversion: Numeric
```

Inspect the `Sorted` field of each channel group struct. Note that all channel groups are sorted now.

```
[m1.ChannelGroup.Sorted]
```

```
ans = 1×8 logical array

   1   1   1   1   1   1   1   1
```

When the MDF-file is finalized and sorted, you can proceed to use all MDF functionaly, such as extracting data using the `read` function.

Finalize an MDF-File Out-of-Place

The `mdfFinalize` function also allows you to finalize an unfinalized MDF-file out-of-place by creating a separate finalized copy. Call the function specifying both the source file name and a destination file name.

```
finalizedPath2 = mdfFinalize("MDFUnfinalized.MF4", "MDFFinalizedOutOfPlace.MF4")
```

```
finalizedPath2 = 
'C:\Users\michellw\Documents\MATLAB\Examples\vnt-ex16754708\MDFFinalizedOutOfPlace.MF4'
```

`MDFFinalizedOutOfPlace.MF4` is a newly created finalized copy and can be opened using the `mdf` function.

```
m2 = mdf(finalizedPath2)
```

```
m2 = 
  MDF with properties:

   File Details
                Name: 'MDFFinalizedOutOfPlace.MF4'
                Path: 'C:\Users\michellw\Documents\MATLAB\Examples\vnt-ex16754708\MDFFinalizedOu
              Author: ''
          Department: ''
             Project: ''
             Subject: ''
             Comment: ''
             Version: '4.11'
            DataSize: 2596814
    InitialTimestamp: 2021-04-12 10:06:43.000000000

   Creator Details
    ProgramIdentifier: 'CE      '
             Creator: [1×1 struct]

   File Contents
          Attachment: [0×1 struct]
        ChannelNames: {8×1 cell}
        ChannelGroup: [1×8 struct]

   Options
          Conversion: Numeric
```

Inspect the `Sorted` field of each channel group struct. Note that all channel groups are sorted now.

```
[m2.ChannelGroup.Sorted]
```

```
ans = 1×8 logical array

   1   1   1   1   1   1   1   1
```

When the MDF-file is finalized and sorted, you can proceed to use all MDF functionality, such as extracting data using the `read` function.

Close and Delete Created MDF-Files

Close access to the finalized MDF-files created in this example by clearing their variables from the workspace.

```
clear m1 m2
```

Delete the MDF-files created in this example to clean up the working directory.

```
delete MDFFinalizedInPlace.MF4 MDFFinalizedOutOfPlace.MF4
```

Conclusion

Similar to `mdfFinalize`, the `mdfSort` function supports sorting operations both in-place and out-of-place. You can apply the same workflow to sort unsorted MDF-files.

To summarize:

- If an MDF-file is finalized and sorted, it can be opened using `mdf` and data can be read using `read`.
- If an MDF-file is finalized and unsorted, it can be opened using `mdf` but data cannot be read using `read`. Use `mdfSort` to sort the file.
- If an MDF-file is unfinalized and sorted, it cannot be opened using `mdf`. Use `mdfFinalize` to finalize the file.
- If an MDF-file is unfinalized and unsorted, it cannot be opened using `mdf`. Use `mdfFinalize` to finalize and sort the file.

CAN Message Reception Behavior in Simulink

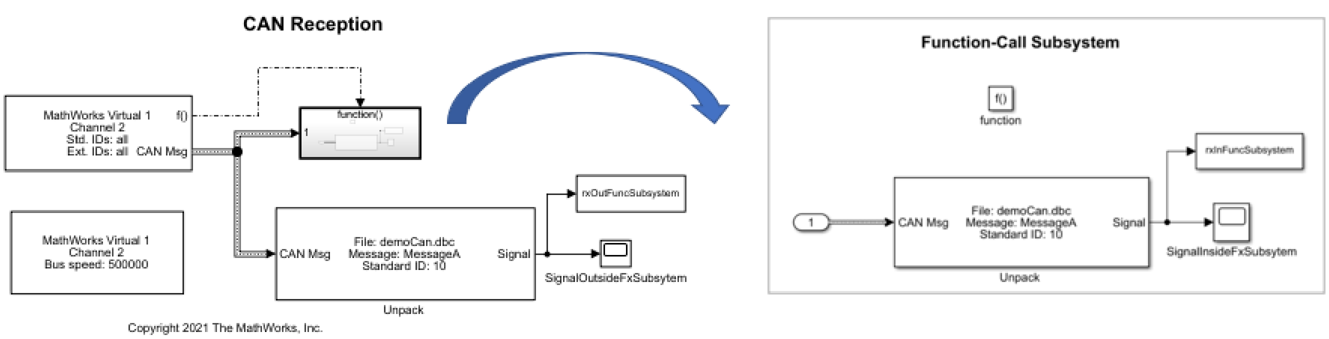
This example shows how to observe the message processing behaviors of the CAN Receive and CAN Unpack blocks in multiple modeling scenarios. This example demonstrates two cases, with and without using the function trigger `f()` port of the CAN Receive block. The outputs of the model indicate the number of CAN messages unpacked for downstream processing in each case. The example uses MathWorks virtual CAN channels to send CAN messages from MATLAB to the Simulink model. These modeling practices and behavior also apply to the CAN FD protocol using the Vehicle Network Toolbox CAN FD blocks.

Explore the Example Model

The example model contains a CAN Receive block configured for Mathworks virtual channels sampling every 500 ms. The received CAN messages are unpacked in two ways:

- A CAN Unpack block inside a function-call subsystem, triggered by the function trigger `f()` port of the CAN Receive block.
- A CAN Unpack block connected directly to the `CAN Msg` output port.

Scopes are placed to view the received signals in both cases. Also, the signal values from the output port of the CAN Unpack blocks are exported to the MATLAB workspace, and used to plot the results.




```
open CanReceiveModel
```

Prepare the CAN Messages for Transmission

To demonstrate the operation of the model, CAN messages are sent from MATLAB. The messages are loaded from the provided MAT-file. A `canChannel` is created to transmit the data later in this example. The messages to send are timed periodically at 100 ms, and the contained signal data is incrementing linearly.

```
load canMessages.mat
txCh = canChannel("MathWorks","Virtual 1", 1);
```

Execute the Model and Replay CAN Messages from MATLAB

Assign a finite simulation time and run the model.

```
simTime = 10;
set_param("CanReceiveModel","StopTime",num2str(simTime))
set_param("CanReceiveModel","SimulationCommand","start")
```

Pause script execution until the model is recognized as fully started.

```
while strcmp(get_param("CanReceiveModel","SimulationStatus"),"stopped")
end
```

Start the CAN channel and execute the `replay` of the loaded CAN messages.

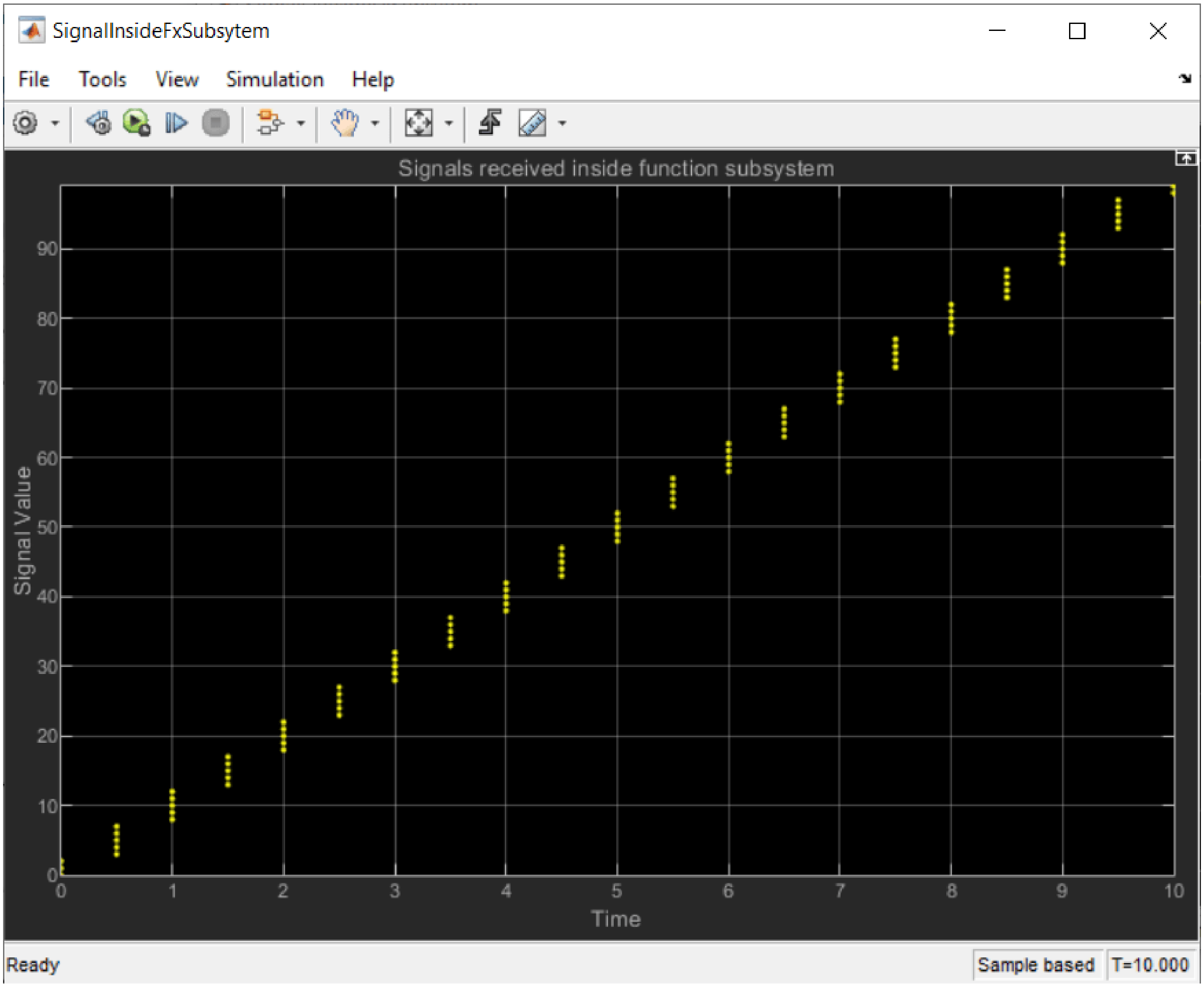
```
start(txCh);
replay(txCh, canMessages);
```

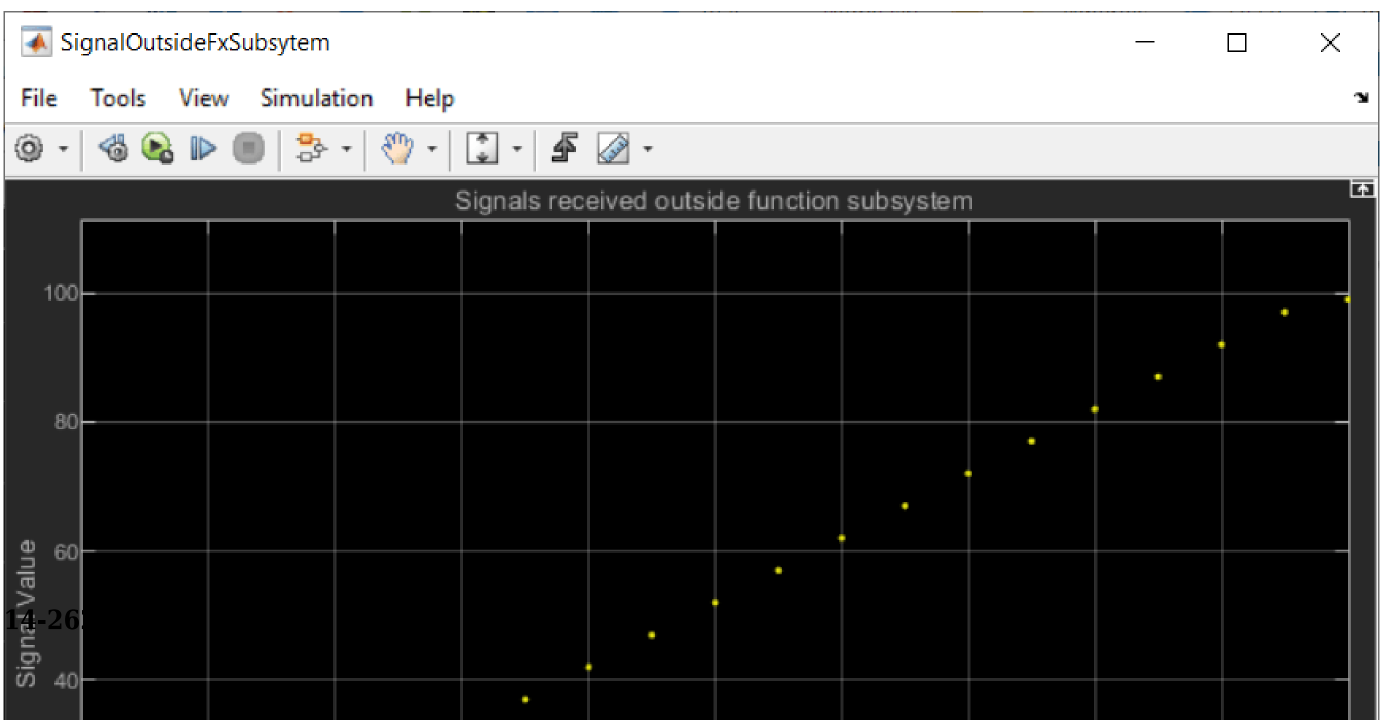
As the replay happens, the CAN Receive block in the model is receiving and processing the messages. You can view the signal values received in real time in the scopes placed inside and outside of the function-call subsystem. Wait until the model finishes simulation to continue.

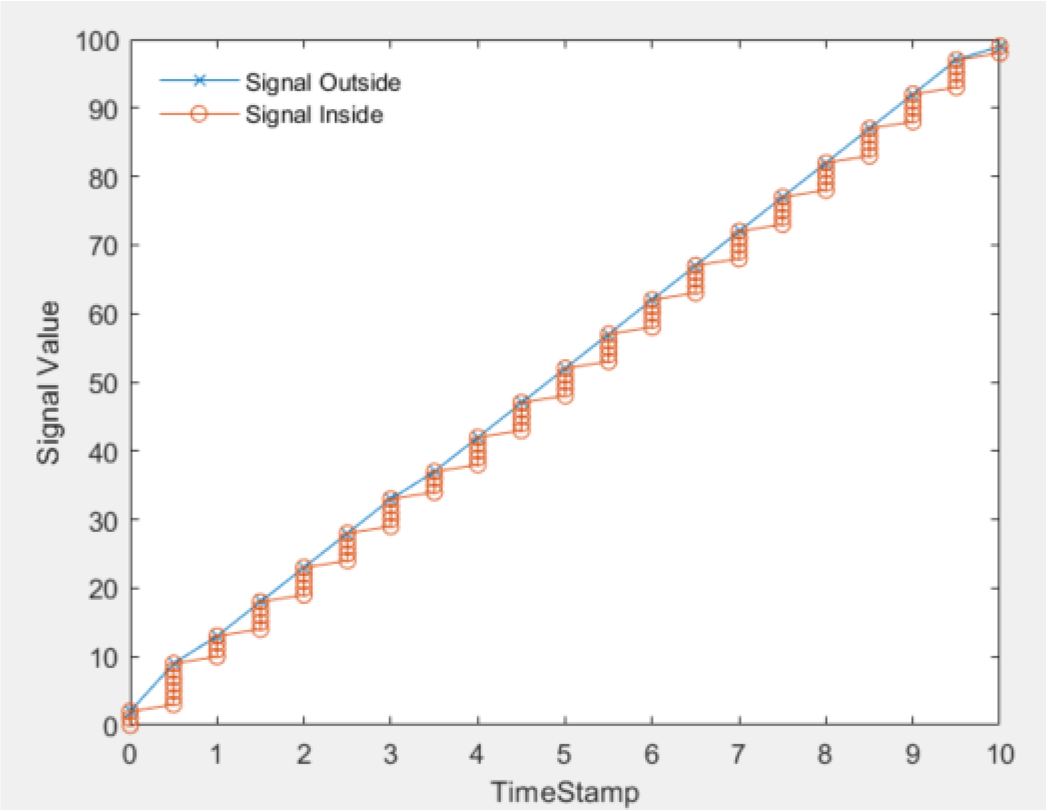
Explore Received Data Handling Results

The scopes provided in the model show the signal values from the messages received inside and outside the function-call subsystem as they are unpacked. The following views show the scopes after the model finishes simulation. Note these differences:

- Inside function-call subsystem: 4-6 messages with increasing signal values are received at every sample time. As such, all CAN messages from the replay were individually received, triggered to the subsystem, and processed by the CAN Unpack block inside the subsystem per sample time.
- Outside function-call subsystem: One message with a jump in signal value is received at every sample time. As such, only the latest CAN message from the replay per sample time was provided to and processed by the CAN Unpack block. The other intermediate messages are not processed.


SignalInsideFxSubsytem
File Tools View Simulation Help
Signals received inside function subsystem
Signal Value
Time
0 1 2 3 4 5 6 7 8 9 10
0 10 20 30 40 50 60 70 80 90
Ready
Sample based
T=10.000


SignalOutsideFxSubsytem
File Tools View Simulation Help
Signals received outside function subsystem
Signal Value
40 60 80 100



Using the exported model signal values from the output port of the CAN Unpack blocks, a plot compares both cases. The function used to plot the results, is included with this example and configured to execute in the Stop Function callback of the model, so that it is executed when the model stops simulation.

The CAN message transmission occurred periodically every 100 ms, while the CAN Receive block sampled at 500 ms. So, in every sample there are 4-6 CAN messages. The following conclusions can be drawn from these waveforms:

Case 1: Unpacking the CAN messages using function trigger (inside function-call subsystem) unpacks all the messages received in each sample.

- Multiple signal values are observed at each sample time.
- The linearly increasing value of the signals indicates that all the messages in every sample time are unpacked.
- No data is suppressed this way, as the function-call subsystem is triggered for each message received and unpacking is done inside it.

Case 2: Unpacking the CAN messages without using function trigger (outside function-call subsystem) unpacks only the latest message in each sample.

- Only one signal value is observed at each sample time.
- Therefore only one CAN message is unpacked at each sample time.
- Only the latest message in the sample is unpacked at each sample time.
- All other messages, except the latest one, are suppressed in each sample.

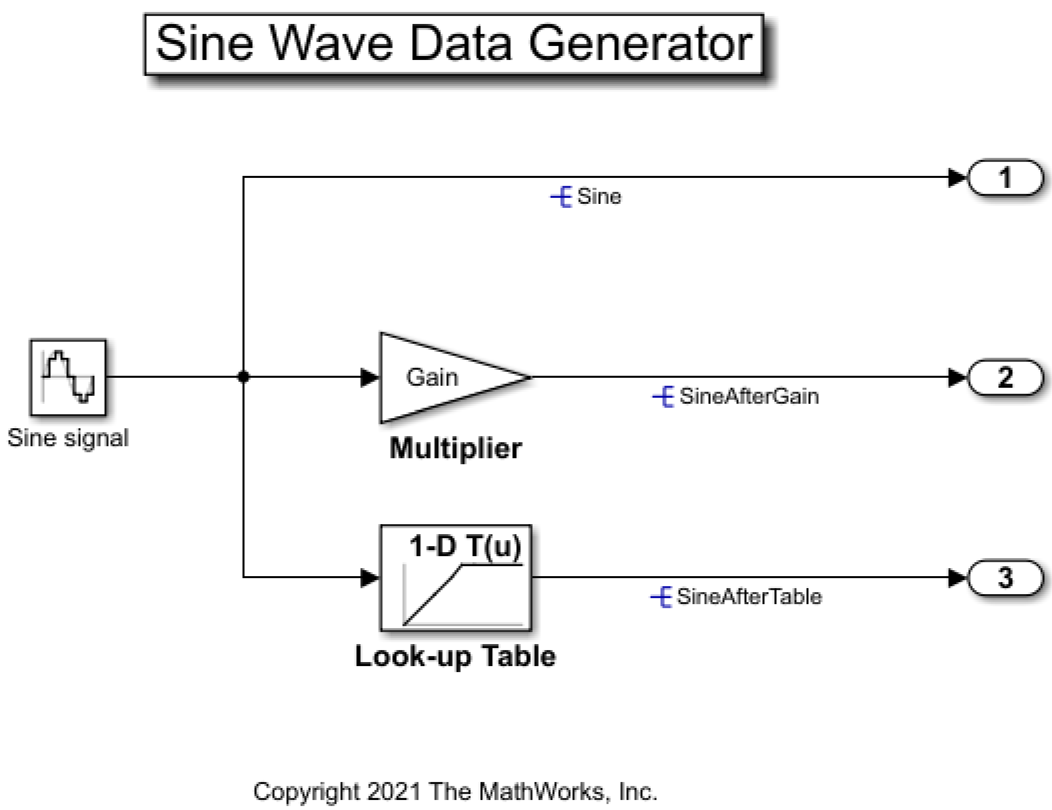
In summary, the function trigger port of the CAN Receive block is used to unpack all the messages received every sample time. If not used, then only the latest message is unpacked in each sample time. Choose the model behavior based on the requirement of your system and data processing needs.

Read XCP Measurements with Direct Acquisition

This example shows how to use the XCP protocol capability to connect and acquire data from a Simulink model deployed to a Windows executable. The example reads measurement parameters of the model using TCP and direct memory access. XCP is a high-level protocol used for accessing and modifying internal parameters and variables of a model, algorithm, or ECU. For more information, refer to the ASAM standards.

Algorithm Overview

The algorithm used in this example is a Simulink model built and deployed as an XCP server. The model has already been compiled and is available to run in the file `XCPServerSineWaveGenerator.exe`. Additionally, an A2L-file is provided in `XCPServerSineWaveGenerator.a2l` as an output of that build process. The model contains three measurements and two characteristics accessible via XCP. Because the model is already deployed, Simulink is not required to run this example. The following image illustrates the model.



For details about how to build a Simulink model, including an XCP server and generating an A2L-file, see “Export ASAP2 File for Data Measurement and Calibration” (Simulink Coder).

Run the XCP Server Model

To communicate with the XCP server, the deployed model must be run. By using the `system` function, you can execute the `XCPServerSineWaveGenerator.exe` from inside MATLAB. The function requires constructing an argument list pointing to the executable. A separate command window opens and shows running outputs from the server.

```
sysCommand = ['"', fullfile(pwd, 'XCPServerSineWaveGenerator.exe'),'"', ' &'];
system(sysCommand);
```

Open the A2L-File

An A2L-file is required to establish a connection to the XCP server. The A2L-file describes all of the functionality and capability that the XCP server provides, as well as the details of how to connect to the server. Use the `xcpA2L` function to open the A2L-file that describes the server model.

```
a2lInfo = xcpA2L("XCPServerSineWaveGenerator.a2l")
```

```
a2lInfo = 
  A2L with properties:

   File Details
                 FileName: 'XCPServerSineWaveGenerator.a2l'
                 FilePath: 'C:\Users\kuanliu\OneDrive - MathWorks\Documents\MATLAB\Examples\vnt-e
               ServerName: 'ModuleName'
                 Warnings: [0×0 string]

   Parameter Details
                   Events: {'100 ms'}
                EventInfo: [1×1 xcp.a2l.Event]
             Measurements: {'Sine'  'SineAfterGain'  'SineAfterTable'  'XCPServer_DW.lastCos'  'X
          MeasurementInfo: [6×1 containers.Map]
          Characteristics: {'Gain'  'ydata'}
       CharacteristicInfo: [2×1 containers.Map]
                 AxisInfo: [1×1 containers.Map]
            RecordLayouts: [4×1 containers.Map]
             CompuMethods: [3×1 containers.Map]
                CompuTabs: [0×1 containers.Map]
               CompuVTabs: [0×1 containers.Map]

   XCP Protocol Details
        ProtocolLayerInfo: [1×1 xcp.a2l.ProtocolLayer]
                  DAQInfo: [1×1 xcp.a2l.DAQ]
    TransportLayerCANInfo: [0×0 xcp.a2l.XCPonCAN]
    TransportLayerUDPInfo: [0×0 xcp.a2l.XCPonIP]
    TransportLayerTCPInfo: [1×1 xcp.a2l.XCPonIP]
```

TCP is the transport protocol used to communicate with the XCP server. Details for the TCP connection, such as the IP address and port number, are contained in the `TransportLayerTCPInfo` property.

```
a2lInfo.TransportLayerTCPInfo
```

```
ans = 
  XCPonIP with properties:
          CommonParameters: [1×1 xcp.a2l.CommonParameters]
    TransportLayerInstance: ''
                      Port: 17725
                   Address: 2.1307e+09
             AddressString: '127.0.0.1'
```

Create an XCP Channel

To create an active XCP connection to the server, use the `xcpChannel` function. The function requires a reference to the server A2L-file and the type of transport protocol to use for messaging with the server.

```
xcpCh = xcpChannel(a2lInfo, "TCP")
```

```
xcpCh = 
  Channel with properties:

              ServerName: 'ModuleName'
             A2LFileName: 'XCPServerSineWaveGenerator.a2l'
          TransportLayer: 'TCP'
    TransportLayerDevice: [1×1 struct]
              SeedKeyDLL: []
```

Connect to the Server

To activate communication with the server, use the `connect` function.

```
connect(xcpCh)
```

Directly Acquire Measurement Data

A measurement in XCP represents a variable in the memory of the model. Measurements available from the server are defined in the A2L-file. One way to read measurement data is using direct memory access. The `readMeasurement` function acquires the current value for a given measurement from the server. It is a single read at this moment without buffering.

```
readMeasurement(xcpCh, "Sine")
```

```
ans = -0.9511
```

```
readMeasurement(xcpCh, "SineAfterGain")
```

```
ans = -1.1756
```

```
readMeasurement(xcpCh, "SineAfterTable")
```

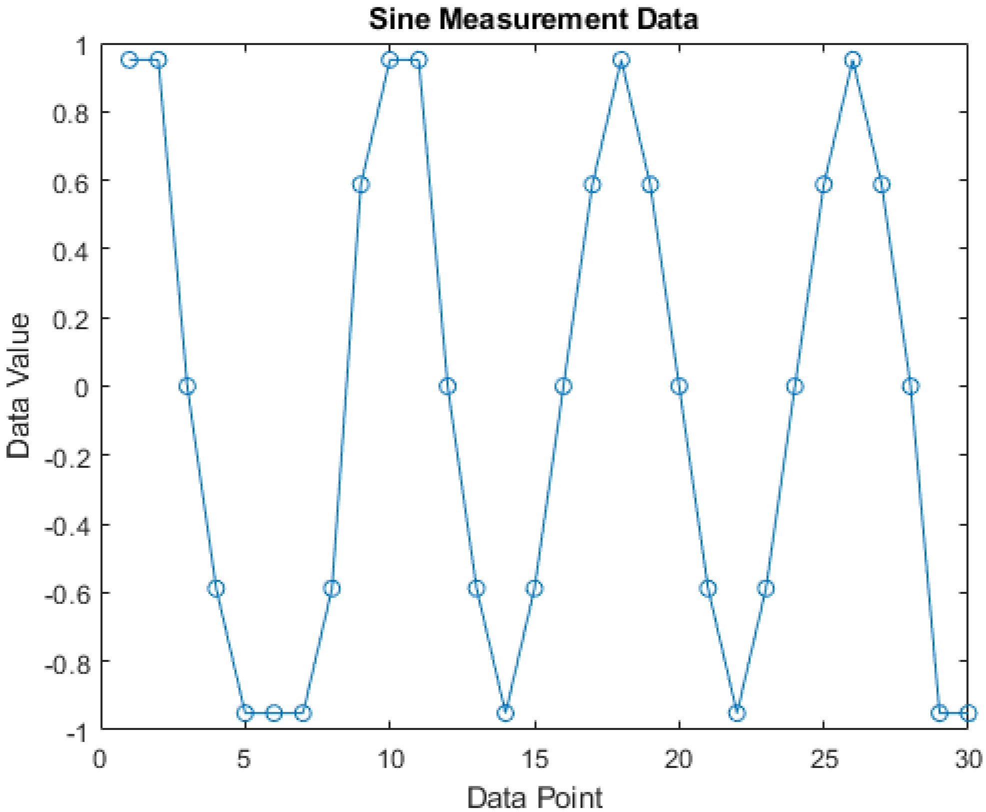
```
ans = 0
```

Continuously Acquire Measurement Data

It might be necessary to read a measurement continuously at some regular interval, such as for visualizing a value in a custom UI or using the value as input to some processing code. In such cases, `readMeasurement` is callable at any type of interval driven by a timer or loop. Below, `readMeasurement` is called in a fixed loop with no delay to accumulate and plot the values read. The value of the measurement is continuously changing in the memory of model, so not every data change is reflected in the plot as the values are relative to the rate of the read call itself. Reading measurements this way is best suited for asynchronous or low frequency purposes.

```
allSamples = zeros(30,1);
for ii = 1:30
    allSamples(ii) = readMeasurement(xcpCh, "Sine");
end
plot(allSamples, "o-")
```

```
title("Sine Measurement Data")
xlabel("Data Point")
ylabel("Data Value")
```



Disconnect from the Server

To deactivate communication with the server, use the `disconnect` function. The XCP server can be safely closed after disconnecting.

```
disconnect(xcpCh)
```

Clean Up

```
clear a2lInfo
```