# SystemTest™
## User's Guide

# MATLAB®&SIMULINK®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

**Trademarks**

**Patents**

**Revision History**

| | | |
|---|---|---|
| May 2006 | Online only | New for Version 1.0 (Release 2006a+) |
| September 2006 | First printing | Revised for Version 1.0.1 (Release 2006b) |
| March 2007 | Online only | Revised for Version 1.1 (Release 2007a) |
| September 2007 | Second printing | Revised for Version 2.0 (Release 2007b) |
| March 2008 | Online only | Revised for Version 2.1 (Release 2008a) |
| October 2008 | Online only | Revised for Version 2.2 (Release 2008b) |
| March 2009 | Online only | Revised for Version 2.3 (Release 2009a) |
| September 2009 | Online only | Revised for Version 2.4 (Release 2009b) |
| March 2010 | Online only | Revised for Version 2.5 (Release 2010a) |
| September 2010 | Online only | Revised for Version 2.6 (Release 2010b) |
| April 2011 | Online only | Revised for Version 2.6.1 (Release 2011a) |
| September 2011 | Online only | Revised for Version 2.6.2 (Release 2011b) |
| March 2012 | Online only | Revised for Version 2.6.3 (Release 2012a) |
| September 2012 | Online only | Revised for Version 2.6.4 (Release 2012b) |
| March 2013 | Online only | Revised for Version 2.6.5 (Release 2013a) |
| September 2013 | Online only | Revised for Version 2.6.6 (Release 2013b) |
| March 2014 | Online only | Revised for Version 2.6.7 (Release 2014a) |
| October 2014 | Online only | Revised for Version 2.6.8 (Release 2014b) |
| March 2015 | Online only | Revised for Version 2.6.9 (Release 2015a) |

# Contents

# Working with Test Vectors

**2**

**3**

**4**

# Author Signals in the Test Case Editor

**5**

## Generate a Test Harness from a Model

**6**

## Use the Instrument Control Toolbox Elements

**7**

# Use Data Acquisition Toolbox Elements

**8**

# Use the Image Acquisition Toolbox Element

**9**

# 12

**Function Reference**

# A

**SystemTest Hot Keys**

# B

**Test Results Data**

# Getting Started

This section explains what the SystemTest™ software is and how to use it. It contains the following topics:

# SystemTest Product Description

### Manage tests and analyze results for system verification and validation

SystemTest software lets you develop and execute tests that exercise MATLAB® algorithms and Simulink® models. It includes predefined test elements that let you build and maintain standard test routines. You can map test variables into a result set for analysis.

Support for industry standards is available through DO Qualification Kit (for DO-178).

Learn more about verification, validation, and test in Model-Based Design.

## Key Features

- Develops, manages, and edits test structures using predefined test elements
- Stores tests independently of the model under test, for repeatable test execution
- Defines pass/fail criteria for tests using Boolean constraints and tolerance limits
- Generates random test vector values using probability distribution functions
- Runs iterations of Simulink models on multiple processors with Parallel Computing Toolbox™ (available separately)
- Generates reports of test execution and results
- Visualizes and analyzes multidimensional test result

# Overview of the SystemTest Software

| In this section... |
| --- |
| "Getting Familiar with the Desktop" on page 1-3 |
| "General Desktop Features" on page 1-5 |
| "Setting Preferences" on page 1-7 |
| "Viewing Test Results" on page 1-9 |

## Getting Familiar with the Desktop

The SystemTest software is an integrated development environment that lets you perform all of your testing activities from one centralized location. This section provides an overview of the SystemTest environment. For more information about how to use the SystemTest software to build tests and run them, see "Create and Run a Test" on page 1-11.

To get familiar with the SystemTest environment, open the SystemTest desktop by typing `systemtest` at the MATLAB command line.
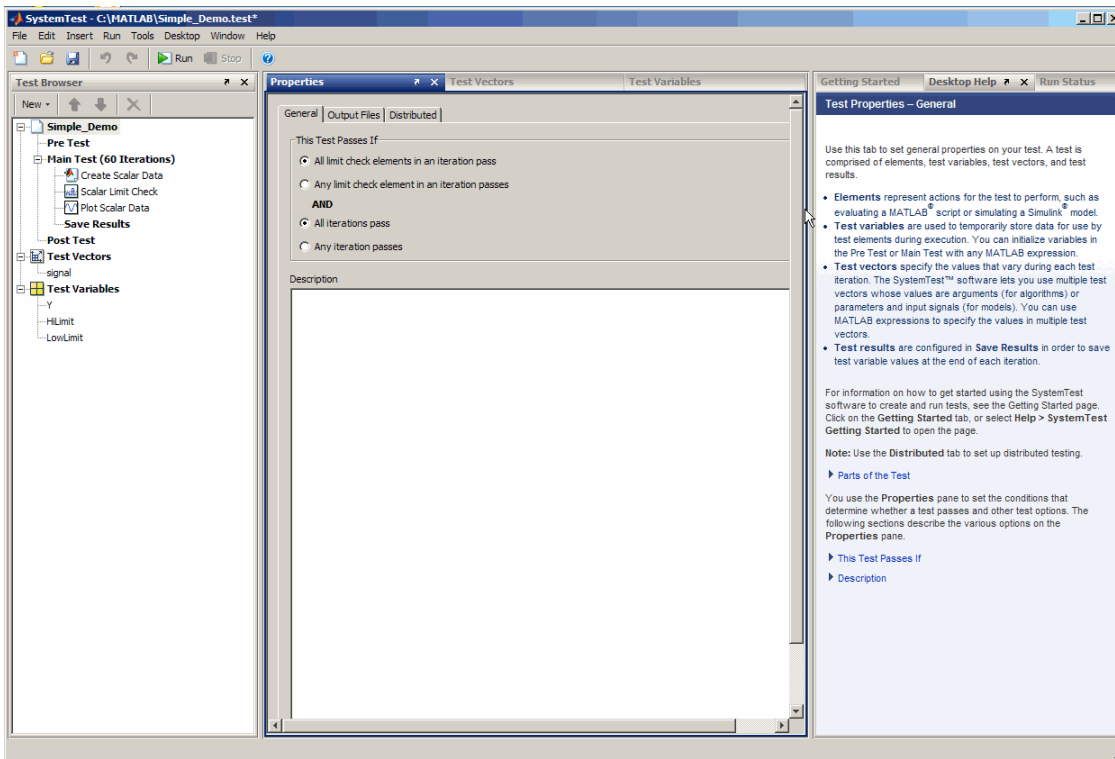
The desktop has a number of different panes that help you to build and run your test.

- **Test Browser** — Shows the overall structure of a test. A test is made up of Pre Test, Main Test, Save Results, and Post Test. Use the Test Browser to add elements to your test. These elements determine what actions your test performs.

- **Test Vectors** — Lets you define the parameters or test cases of your test. The test vectors you define determine the number of iterations performed by your test. Test vectors are automatically indexed during test execution.

- **Test Variables** — Lets you define variables used in the scope of your test. Variables can serve both input and output functions in your test. You can define variables that are declared in the Pre Test section of your test or in the Main Test section of your test.

- **Properties** — Shows the properties of the test or the element you are editing. The contents of this pane change when you select a section or element in your test.

- **Elements** — If open, this undocked **Elements** pane allows you to add elements to your test. If not open, you can add elements using the **New** button in the **Test Browser**.
- **Resources** — Lists the instrument or other external device resources associated with the current test. This is only used if you have a license for the Instrument Control Toolbox™ software.
- **Getting Started** — Shows information to help you start using the SystemTest software. If the Getting Started page is closed, select **Help > SystemTest Getting Started** to open it.
- **Help** — Shows help about the element or aspect of the test that is currently selected. For the full product Help, select **Help > SystemTest User's Guide**.
- **Run Status** — Shows a summary of the test's execution status.

## General Desktop Features

The SystemTest desktop has a variety of features to make navigation easier.

### Context Menus

Many areas of the user interface have context menus. For example, if you right-click in the **Test Vectors**, **Test Variables**, **Resources**, **Run Status**, **Getting Started**, or **Desktop Help** panes, you can access these context menus.

If you have the **Elements** pane open, you can add elements to your test using the context menus. If you right-click any element there, you can insert it directly into Pre Test, Main Test, or Post Test using the **Elements** pane context menus. If that section of the test already contains elements, the inserted element will be placed below the currently selected element in that section. You can change the order of elements in the test by using the arrow buttons in the **Test Browser**, or by dragging and dropping.

### Hot Keys

The SystemTest software offers various keyboard shortcuts, or hot keys, to access certain commands via the keyboard. For example, pressing **F5** is an alternative way to run a test, and pressing **Ctrl+N** creates a new untitled test.

### Undo/Redo Support

Undo and redo support is available through the **Edit** menu or on the SystemTest toolbar. This feature allows you to undo actions you have done throughout the desktop. The undo

queue is global to the entire desktop. For example, if you add a test vector and then perform an action in the Properties pane, those two actions will be the last two items in the queue. The undo order applies across all the panes in the desktop.

To use this feature, select the **Edit > Undo action** command, where *action* is the last action you performed. Use the **Undo** command repeatedly to undo multiple actions. The **Edit > Redo action** command will redo the last undo you performed.

Most actions in the desktop are undoable. Some actions pertaining to the elements that are part of the hardware toolboxes, Data Acquisition Toolbox™, Instrument Control Toolbox, and Image Acquisition Toolbox™, cannot be undone since they involve connections to hardware.

The following actions will clear the list of actions in the undo queue:

- Closing a test
- Opening a test
- Creating a new test
- Refreshing a Simulink model in the Simulink element

## Setting Preferences

You can set SystemTest preferences by selecting **File > Preferences** on the SystemTest desktop. This opens the MATLAB Preferences dialog box. Click **SystemTest** in the left tree if SystemTest Preferences are not showing in the right pane.
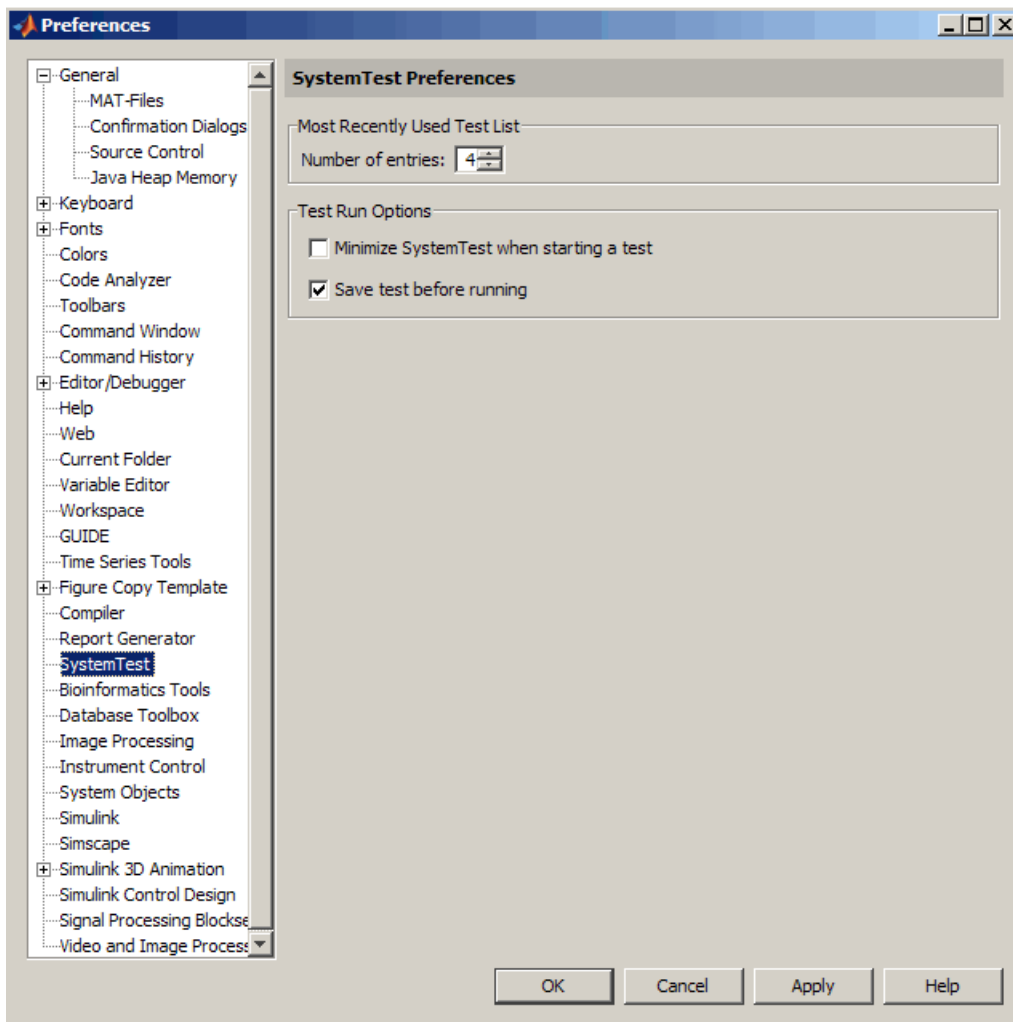
### Most Recently Used Test List

This option determines how many tests will appear on the SystemTest **File** menu's most recent files list. The default is 4 tests. If you change it to 0, no recent tests will appear on the list. The maximum number is 9.

### Test Run Options

Select **Minimize SystemTest when starting a test** if you want the SystemTest desktop to minimize when a test starts running. This check box is cleared by default.

Select **Save test before running** if you want the SystemTest software to save your test before it runs. If this option is selected and you run a test that is not yet saved, you will be prompted to name and save the test. This check box is selected by default.

**Note:** You can save a test any time, before or after running it, by selecting **File > Save**.

### Confirmation Dialog Boxes

You can also turn off confirmation dialog boxes used in the SystemTest software in a different area of the Preferences dialog box by selecting **General > Confirmation Dialogs**. Four SystemTest confirmation dialog boxes are listed there, as shown in the figure that follows.

- **Warn about using a Simulink model with an infinite simulation stop time** — Occurs if you attempt to run a test containing a Simulink element that uses a model with an infinite simulation stop time.
- **Warn about using a Simulink model with unnamed logged signals** — Occurs if you have a model that has logging enabled but has logged signals with no name, and you use that model in a Simulink element in the SystemTest software.
- **Warn each time a new signal is added in the test case editor** — Occurs if you add a signal in the Test Case Editor.
- **Warn each time a signal is deleted in the test case editor** — Occurs if you delete a signal in the Test Case Editor.
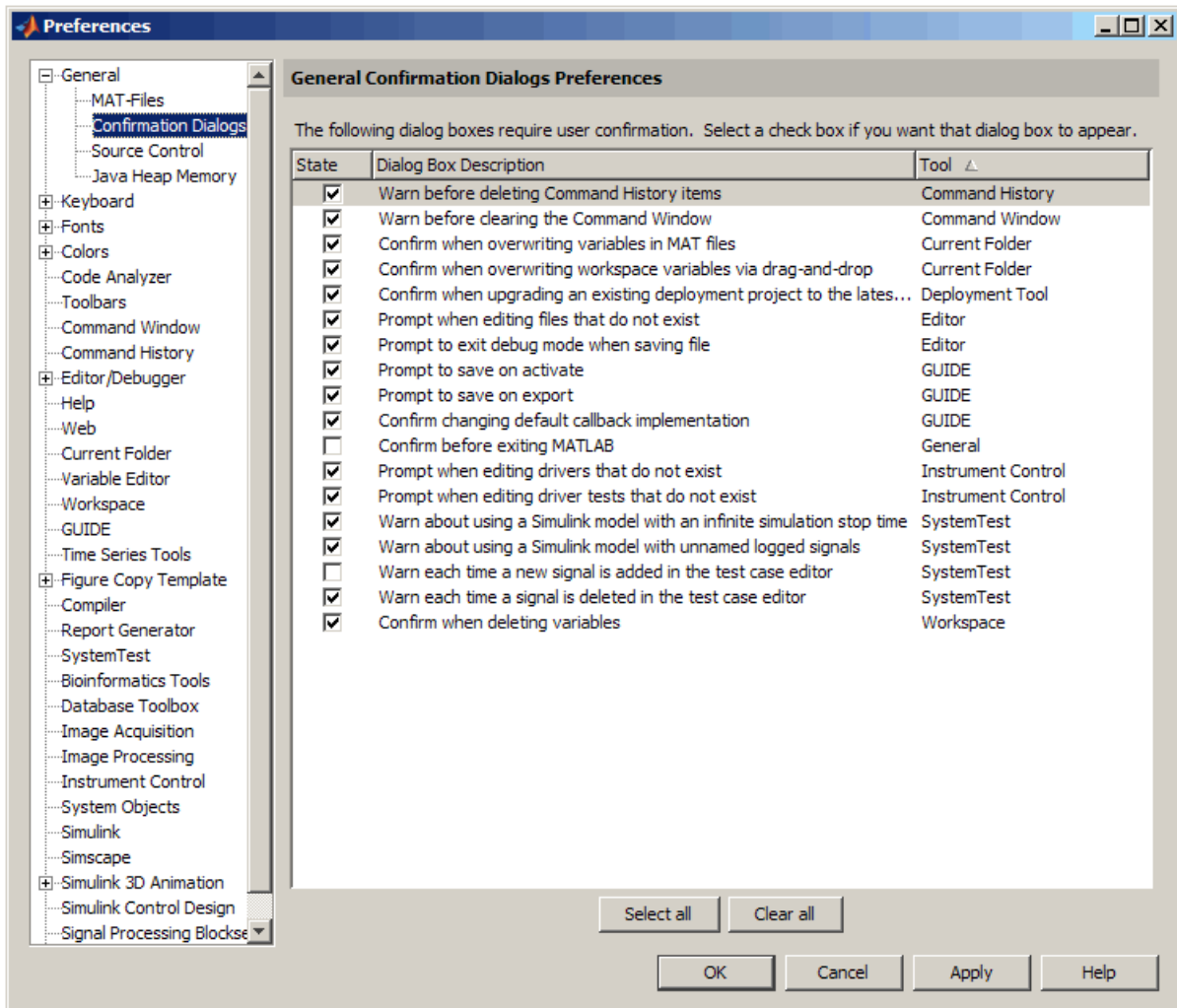
## Viewing Test Results

The SystemTest software allows you to view the results you have chosen to save for your test using a workspace variable called `stresults`. It provides access to the test results object, which is useful for comparing the results of separate test runs and for postprocessing test results.

# Run Tests from the MATLAB Command Line

You can run one or more SystemTest tests from the MATLAB command line, using the `strun` function. This is useful for running multiple test files as a batch or calling a test file as part of a MATLAB file.

---

**Note:** If you use this feature, it is a good idea to first run the test from the SystemTest desktop to verify that elements are not in an error state, and that the test will run successfully, before running it via the MATLAB command line using the `strun` function.

---

The function takes the name of your test file as a string. The test file must be on the MATLAB path, or you can specify the full path in the string.

For example, to run a test called `mytest` that is on the MATLAB path, use this syntax:

```
strun('mytest')
```

To run a test called `mytest` that is not on the MATLAB path, but is in a local directory called `c:\work`, use this syntax:

```
strun('c:\work\mytest.test')
```

To run multiple tests, use a cell array of strings, as follows:

```
strun({'mytest' 'mytest2'})
```

---

**Note:** MATLAB will remain busy while tests are executing via the `strun` command. Control is returned to the MATLAB command line once all tests execute.

---

If the SystemTest desktop is open when `strun` is called, `strun` leaves it open. Otherwise, `strun` closes the desktop after the test runs.

For more information, see strun.

# Create and Run a Test

This example illustrates the four primary stages of testing: planning, building, running the test, and viewing test results.

The example uses a simple MATLAB expression to emulate a scalar measurement during each iteration of the test. The example uses an arbitrary formula dependent on the test vector named `signal` to generate the Y data. The example tests each measurement to determine if it falls within certain specified limits. If a measurement exceeds these limits, that particular iteration of the test fails. By default, the test fails if any iteration fails, but you can configure other pass/fail criteria.

The following sections provide information about each stage, building the example test along the way.

**1** "Plan a Test" on page 1-12
**2** "Construct a Test" on page 1-13
**3** "Run a Test" on page 1-36
**4** "Analyze Test Results" on page 1-40

If you prefer, instead of working through the following sections to build the example, you can load it into the SystemTest software by running the Getting Started with SystemTest example by entering `systemtest Simple_Demo` at the MATLAB command prompt.

# Plan a Test

In this first stage, you must identify what it is you want to test. The SystemTest software lets you specify input data, such as measurements from a model or device, and compare this input data to some predefined limits. Based on this comparison, the SystemTest software can declare whether a test passes or fails.

Keep the following in mind as you plan tests:

- Identify your test data and test vectors.
- Specify test limits and determine if these limits can be expressed as scalar or matrix values. (The Limit Check element supports both scalar and matrix data.)
- Determine what operations your test must perform. Must certain operations happen before others?
- Determine pass/fail criteria for your test.
- Decide which test variables you want to save as test results.

After this planning, you can begin to construct your test, which is described in "Construct a Test" on page 1-13.

# Construct a Test

The SystemTest interface provides a graphical integrated environment that you can use to create and edit tests. Tests consist of elements, test vectors, and test variables. You can use each of these entities to create a variety of test scenarios ranging from a simple test that runs a series of elements once to a full parameter sweep that iterates over the values of test vectors that you define.

The following sections show how to construct a test:

| In this section... |
| --- |
| "Start the SystemTest Software" on page 1-13 |
| "Structure a Test" on page 1-13 |
| "How Test Vectors and Test Variables Relate to the MATLAB Workspace" on page 1-15 |
| "Create a Test Vector" on page 1-16 |
| "Define Test Variables" on page 1-19 |
| "Add Elements" on page 1-21 |
| "Define Pass/Fail Criteria" on page 1-29 |
| "Save Test Results" on page 1-30 |
| "Generate a Test Report" on page 1-33 |
| "Save a Test" on page 1-35 |

## Start the SystemTest Software

Start by opening the SystemTest desktop by typing `systemtest` at the MATLAB command line.

The SystemTest software displays the desktop on your screen. See "Overview of the SystemTest Software" on page 1-3 for an overview.

## Structure a Test

The SystemTest software divides tests into three *sections*.

- **Pre Test** — This section is used to execute test elements in order to perform any test set-up operations, such as initializing variables, loading data from a file, and initializing system resources. Using Pre Test variables, you can assign an initial value to a test variable that persists between Main Test section iterations (unless another element in Main Test modifies the value). Pre Test is not mandatory, but it can be used if your test requires set-up operations to be performed.

- **Main Test** — Main Test defines the test elements that need to be performed across the parameter space defined by your test vectors. In this section Main Test variables are initialized before each Main Test iteration, which lets you assign an initial value to a test variable each time the Main Test runs. This is useful if your test variable has a derived value such as being indexed by a test vector or is the result of a MATLAB expression.

  The number of iterations performed in the Main Test is indicated in the **Test Browser** in parentheses after **Main Test**. Iterations specifies the number of times the Main Test section will be run. This is determined from the test vectors you define. The SystemTest desktop also offers a **Save Results** area for you to specify which test variables you want to save as test results at the end of each Main Test iteration.

- **Post Test** — In this section you can perform any cleanup work necessary at the completion of the Main Test section, such as clearing workspace variables, closing a file, or freeing system resources.

The following figure illustrates the structure of a test.

**Test Structure**

**Pre Test**

pre test element

**Main Test** ← main test variables

main test element

ITERATIONS ← test vectors

save results

**Post Test**

post test element

end of test

## How Test Vectors and Test Variables Relate to the MATLAB Workspace

The SystemTest software has its own internal workspace that it uses to manage test variables and test vectors independently. However it does leverage the MATLAB workspace during test execution, and when using a MATLAB element.

During test execution, SystemTest test variables and test vectors are evaluated in the MATLAB base workspace. Then at the end of test execution, they are cleared out and the MATLAB base workspace is restored to what it was before the test execution.

When using a MATLAB element in the SystemTest software, you can reference a variable in the base workspace without having to create a test vector or test variable in the SystemTest software. However the SystemTest software will not be aware of this data, so you could not make use of it in any other element type or in saved results. You can only access it from a MATLAB element. If you need to use it in other elements, you can create test variables or test vectors in the SystemTest software.

## Create a Test Vector

Test vectors are composed of values derived from a MATLAB expression. You can use any MATLAB expression that evaluates to a 1-by-n matrix or cell array to define your test vector. Using test vectors, you can iterate through a range of values to see how a system performs. Test vectors constitute parameterized testing in the SystemTest software. They are the test cases for your test.

For tests with multiple test vectors, the product of the lengths of the test vectors defines the number of iterations the test performs. For example, if you define the test vector `[10 20 30]`, the test runs three times, using a value of 10 for the first run, 20 for the second, and 30 for the final run. If you add a second test vector with three other values, the total number of test runs would be nine. The SystemTest software iterates through each vector in combination with the other vector as though the test were a group of nested FOR loops —the outermost loop being the first test vector in your table and the innermost loop being the last test vector. The **Main Test** section in the **Test Browser** shows the total number of test iterations defined by your test vectors.

For the example, use the vector `[pi/15:pi/15:4*pi]` which defines 60 values for our test vector ranging from `pi/15` to `4*pi` in `pi/15` increments. To specify this test vector:

1   Click the **New Vector** button in the **Test Vectors** pane.

    The Insert Test Vector dialog box opens.
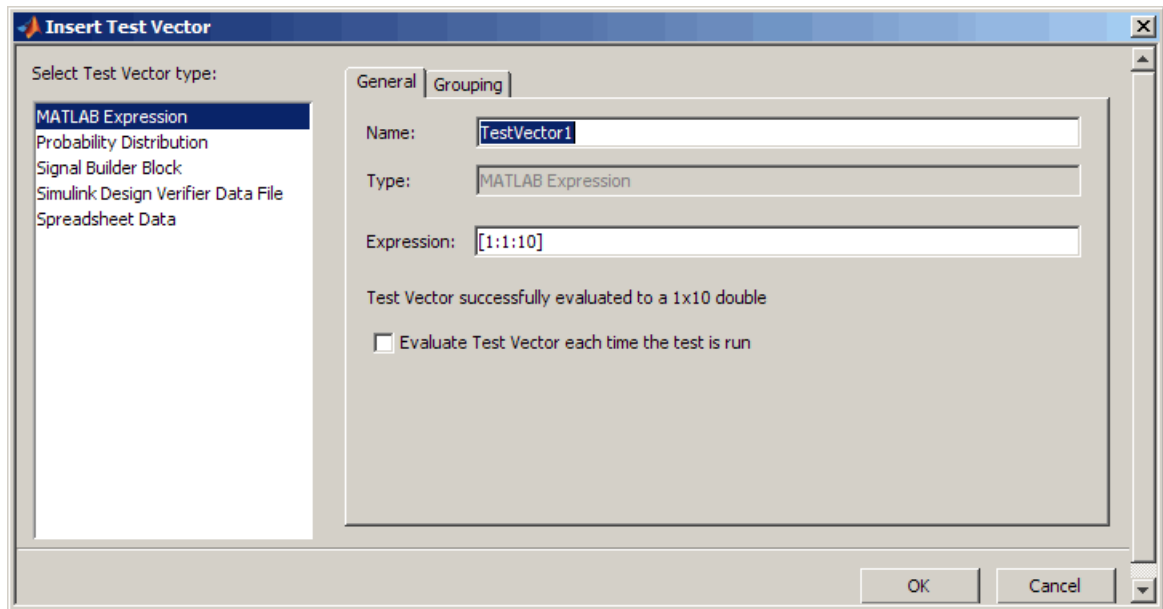
**2** Keep the default test vector type of **MATLAB Expression**. Assign a name to the test vector by clicking the **Name** field. For this example, name the test vector `signal`.

**3** Assign a value to the test vector by clicking the **Expression** field. Enter the test vector specified above for the pi values. Click **OK**.

| Properties | | Test Vectors | ↗ ✕ | Test Variables | |
|---|---|---|---|---|---|

New... ✕ Evaluate

| Name | Length | Group Name | Type |
|---|---|---|---|
| signal | 60 | | MATLAB Expression |

▲▼

General | Grouping

Name: signal

Type: MATLAB Expression

Expression: [pi/15:pi/15:4*pi]

Test Vector successfully evaluated to a 1x60 double

☐ Evaluate Test Vector each time the test is run
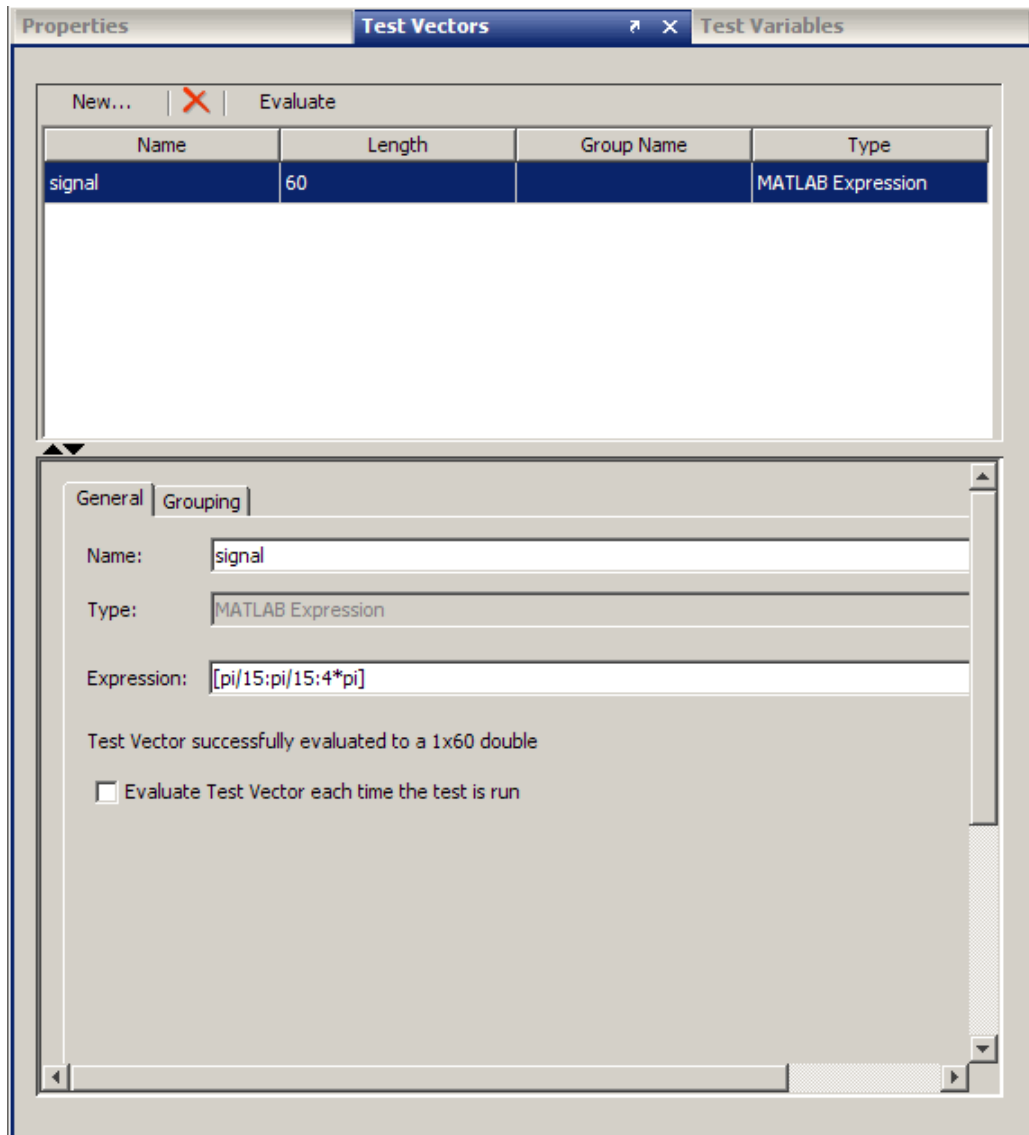
After you create the test vector, in the **Test Browser** pane, the **Main Test** section label updates to include the number of iterations defined by the test vector. It should say `Main Test (60 Iterations)`.

---

**Note:** Grouping test vectors determines how they will be iterated through when the test runs. For information on grouping vectors, see "Grouped Test Vectors".

---

**Note:** You can also use probability distributions when you create a test vector. For information, see "Randomized Test Vectors with Probability Distributions".

---

## Define Test Variables

The SystemTest software uses *test variables* to define temporary storage variables that a test acts on or generates. You assign test variables in the Pre Test or Main Test sections of your test.

You can define Pre Test variables or Main Test variables. Using Pre Test variables, you can assign an initial value to a test variable that persists between Main Test section iterations (unless another element in Main Test modifies the value). Pre Test is not mandatory, but it can be used if your test requires set-up operations to be performed.

Main Test defines the test elements that need to be performed across the parameter space defined by your test vectors. Main Test variables are initialized before each Main Test iteration, which allows you to assign an initial value to a test variable each time the Main Test runs. This is useful if your test variable has a derived value such as being indexed by a test vector or is the result of a MATLAB expression. You add elements in this section.

The example test requires three test variables:

- `Y` — Contains a value that will be calculated from the `signal` test vector at each iteration.
- `HiLimit` — Contains the upper limit for `Y` that you do not want the signal to exceed.
- `LowLimit` — Contains the lower limit for `Y` that you do not want the signal to go below.

To create these test variables:

1. Click the **Test Variables** tab in the middle pane of the SystemTest desktop.

2. Click the **New** button to create a Pre Test or Main Test variable. The Insert Test Variable dialog box opens. Leave the default value of `Main Test` in the **Assigned in** field, to create a new Main Test variable.

```
Insert Test Variable                              [X]

    Name:
    Var1

    Initial value (optional)


    Assigned in
    Main Test                                    [▼]

                              OK          Cancel
```
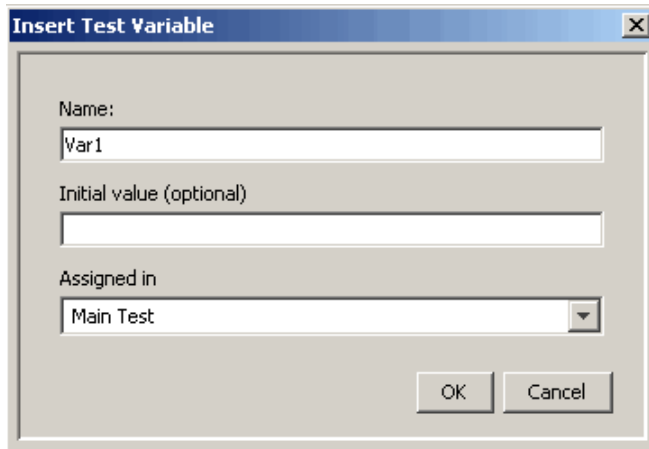
3. Assign a name to the test variable by clicking the **Name** field and entering the test variable name. For this example, enter `Y`.

4. Set the test variable's initial value by clicking the **Initial Value** field and entering a value. For the example test variable `Y`, enter `0`. Click **OK**.

---

**Note:** If you do not provide an initial value, it will default to empty, that is, `Var1 = [];` in MATLAB code.

---

**Note:** Test variables are re-initialized at the start of each test iteration. The **Initial value** field is blank by default when you create a test variable. If you leave it blank, it will initialize to `[]`. If you enter an initial value (which can be any valid MATLAB expression), that value gets assigned in every iteration.

---
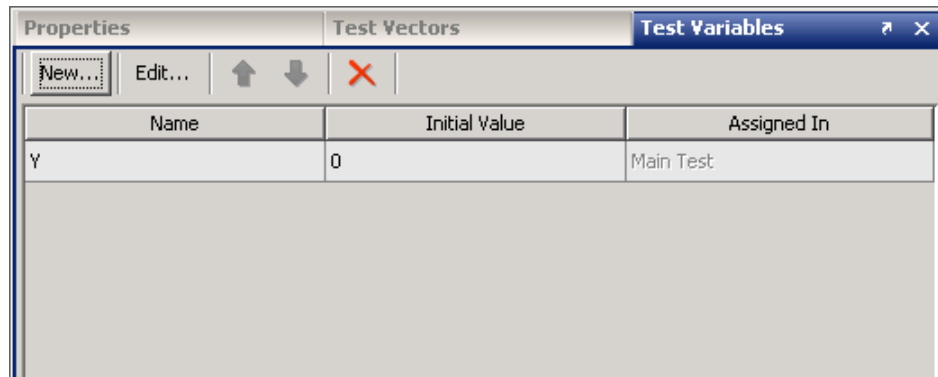
**5** Repeat steps 2 to 4 to create the remaining two test variables, using the settings listed in the following table:

| Variable Name | Initial Value | Assign in |
|---|---|---|
| HiLimit | 1 | **Main Test** |
| LowLimit | -1 | **Main Test** |

## Add Elements

Elements are the actions that a test performs. The SystemTest software includes the following set of elements, listed in alphabetical order.

- General Plot — Used to plot any type of data over multiple iterations.

- IF — Implements a logic control operator.

- Limit Check — Specifies the comparison to be performed of the value(s) under test and their expected value(s), or limit(s).

- MATLAB — Executes any MATLAB statements.

- Simulink — Runs a Simulink model. Note that you need to have a license for Simulink to use this element.

- Stop — Implements a logic control operator.

- Subsection — Creates a new section in a test that you can use to group elements within.

---

**Note:** Some MathWorks® products, such as the Image Acquisition Toolbox, the Data Acquisition Toolbox, and the Instrument Control Toolbox, provide their own elements that integrate those products' capabilities within the SystemTest software. If you have licenses for those products, those elements will also appear in the elements list.

---

You add elements to a section in your test; however, not all elements can be added to all sections. For example, you can use a MATLAB element anywhere within a test, but you can only use the Limit Check element in the Main Test section.

To illustrate using elements, let's continue with this example. This test uses three elements in the Main Test section.

| Element | Description |
|---|---|
| MATLAB | Use a MATLAB expression to assign data to Y that is dependent on the test vector `signal`. |
| Limit Check | Compare the value generated in the MATLAB element to the specified limit and see if the Y test variable exceeds the upper or lower limit you defined in your `HiLimit` and `LowLimit` test variables. |
| General Plot | Plot the current test variable values and see whether the test variable exceeds the upper and lower limits. |

To add these elements:

1   Select the section of the test in which you want to add the element. For this example, click **Main Test** in the **Test Browser**.
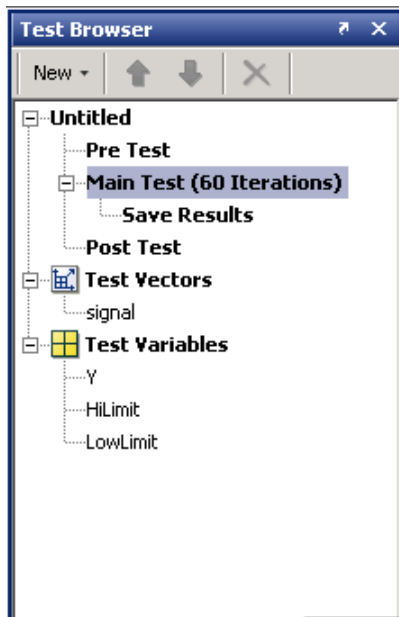
2   Specify the element you want to add to the test section. For this example, click the
    **New > Test Element** button and select **MATLAB**. A MATLAB element appears in
    the Main Test section of your test and the MATLAB element property page opens in
    the **Properties** pane of the SystemTest desktop.
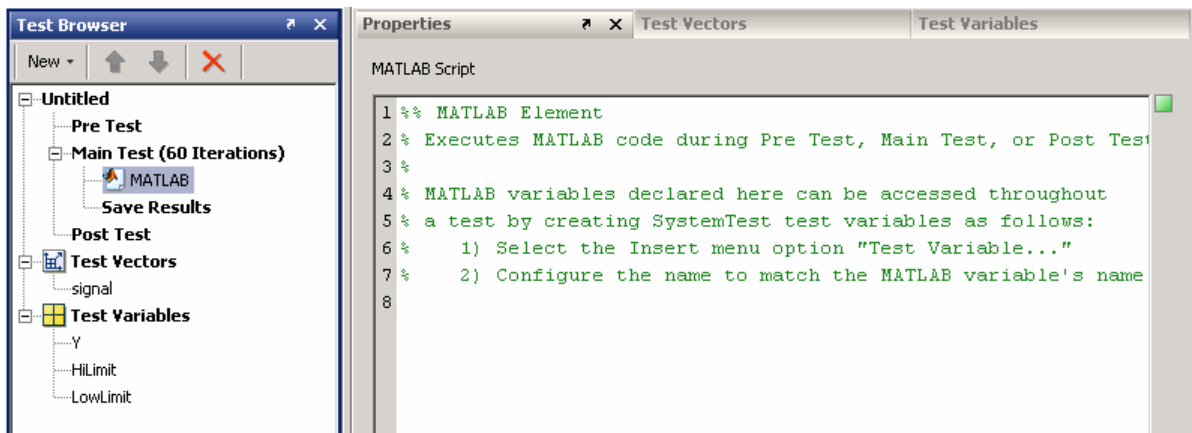
**3** In the **Properties** pane, type the following code in the MATLAB Script edit box. This MATLAB code calculates a value for Y that is dependent on the test vector `signal`.

```
Y = sin(signal)+ rand -.5;
```

| Properties | ↗ ✕ | Test Vectors | Test Variables |
|---|---|---|---|

MATLAB Script

```
1 %% MATLAB Element
2 % Executes MATLAB code during Pre Test, Main Test, or Post Test
3 %
4 % MATLAB variables declared here can be accessed throughout
5 % a test by creating SystemTest test variables as follows:
6 %    1) Select the Insert menu option "Test Variable..."
7 %    2) Configure the name to match the MATLAB variable's name
8
9 Y = sin(signal) + rand -.5;
```

During each iteration, the SystemTest software evaluates the MATLAB expression and assigns a value to Y.

**4** Add the Limit Check element to the **Main Test** section of the test. With the MATLAB element selected, click the **New > Test Element** button, and click **Limit Check**. A Limit Check element appears in the **Main Test** section of the test and the Limit Check properties page opens in the **Properties** pane. For this example, the Limit Check element must follow the MATLAB element in the test.

---

**Note:** You can reposition an element in a test by selecting the element and then clicking the up and down arrows in the **Test Browser** toolbar. You can also drag and drop elements within **Main Test**. You cannot move elements between test sections.

---
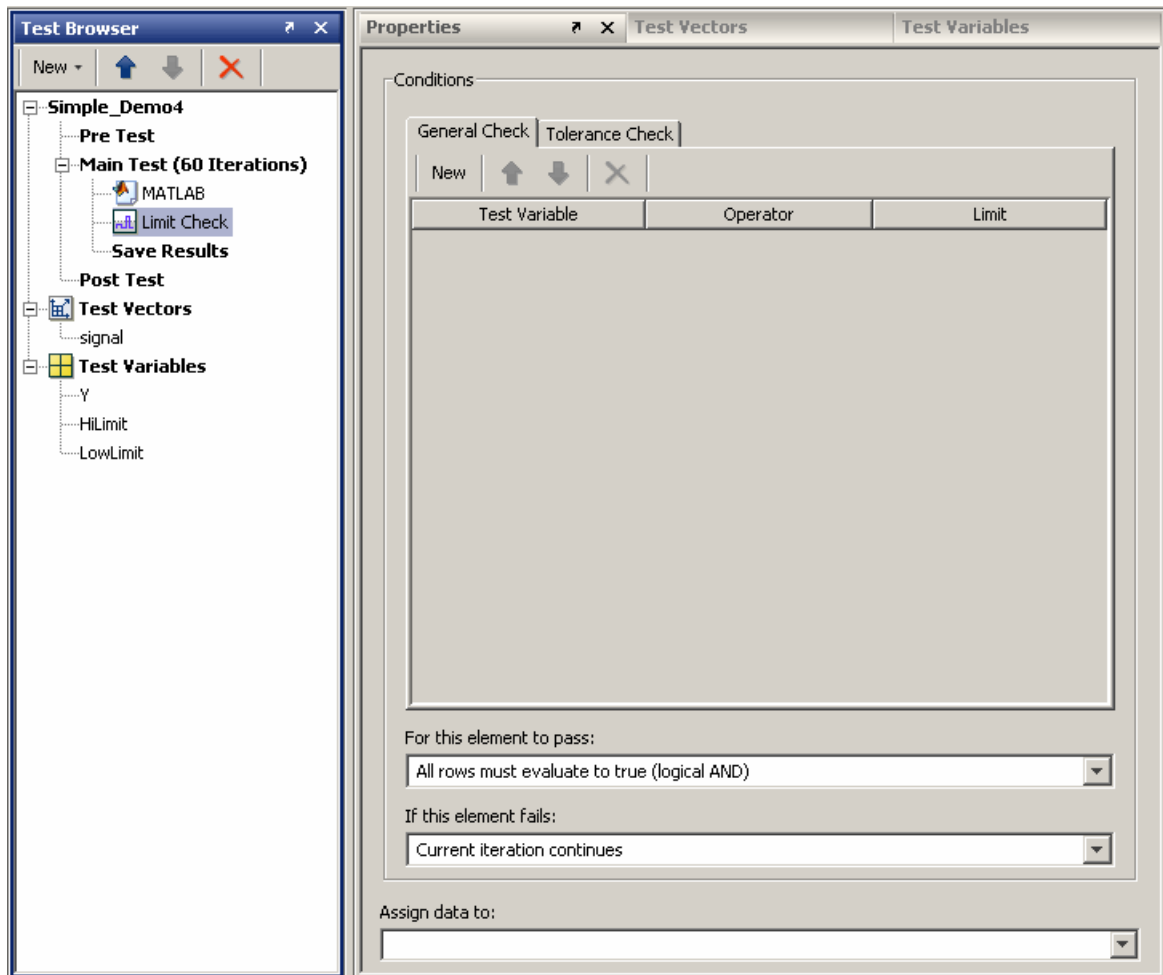
In the **General Check** tab, click the **New** button to add a limit check. Notice that the Limit Check element icon in the **Test Browser** shows a red x, which indicates that information is missing. The corresponding red outlining in the **Properties** pane highlights any fields that require configuration. A test cannot run unless everything is properly configured.

**5** Specify the limit comparison operations in the Limit Check element.

**a** In the **Test Variable** column, click the drop-down list and select a test variable you created in step 4. For this example, select Y.

**b** In the **Operator** column, click the drop-down list and select the comparison you want to perform. For this example, pick the less-than-or-equal-to operator, **<=**.

**c** In the **Limit** column, click the drop-down list and select the test variable you want to compare to. For this example, select HiLimit, which is the test variable you created earlier.

The following figure shows the configuration of this limit.
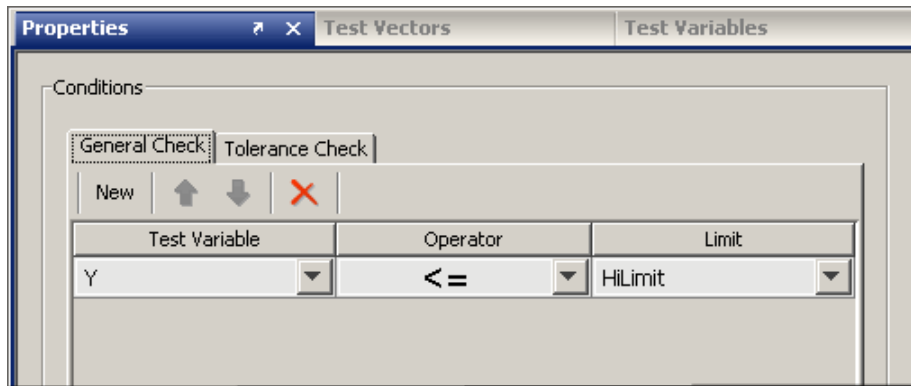


**6** To add another limit comparison operation, click the **New** button again in the **General Check Properties** pane. A new row appears below the last limit you specified. In this new row, set **Test Variable** to Y, set **Operator** to >=, and set **Limit** to LowLimit.

**7** To view the test variables as the test runs, plot the data. To add a Plot element to the test, click the **New > Test Element** button, and select **General Plot**. A General Plot element appears in the Main Test section, and the properties page for the element opens in the **Properties** pane.
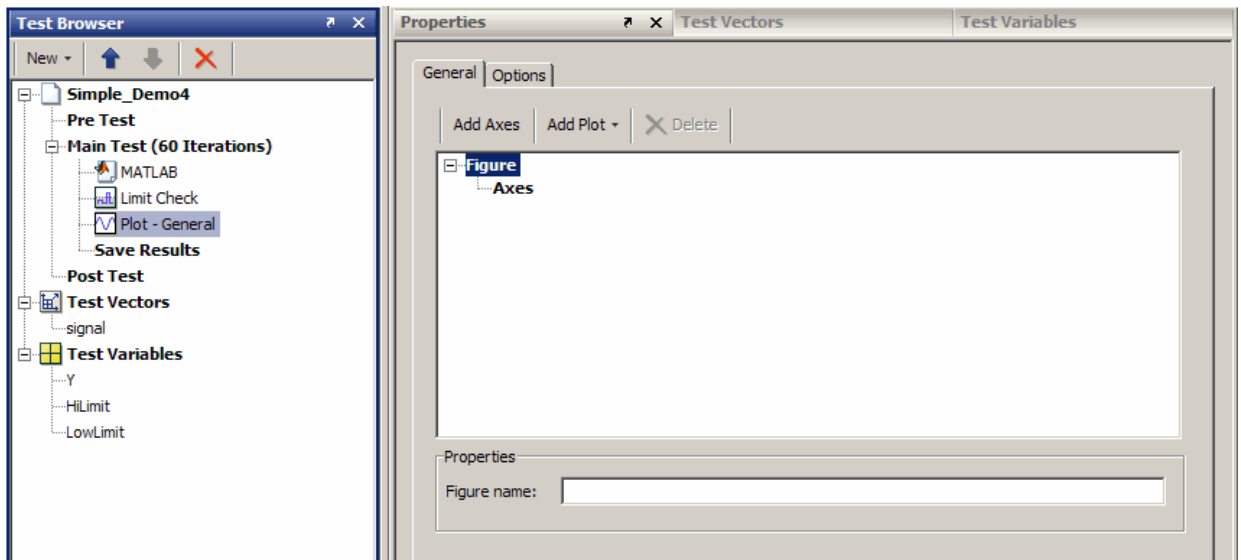
With each Main Test iteration of the test, the General Plot element updates a figure window with data you selected.

**8** Click the **Add Plot** button, and choose **plot** from the list to create a line plot.

**9** We will set up three axes. For the first axes, use the one automatically created. Configure it as follows:

Click the arrow in **Y Data Source** and select Y.

Keep the default **Line color** of blue, and keep the default **Line style** of solid.

Change **Line marker** to point (the first selection in the list that shows one dot).

On the **Options** tab, select **Keep any existing data on the figure**.

**10** Add the second and third axes by clicking the **Add Plot** button again twice and choosing **plot** from the list to create a line plot.

**11** Configure the second and third axes to match the following table, and using `<Iteration Number>` as the **X Data Source** for each one and selecting the **Keep any existing data on the figure** option for each one. The configured element looks like the figure following the table.

| Y Data Sorce | Line Color | Line Style | Line Marker |
|---|---|---|---|
| *Y* | **Blue** | **Solid** | **Point** |

| Y Data Sorce | Line Color | Line Style | Line Marker |
|---|---|---|---|
| *HiLimit* | **Red** | **Dashed** | **No Marker** |
| *LowLimit* | **Black** | **Dashed** | **No Marker** |
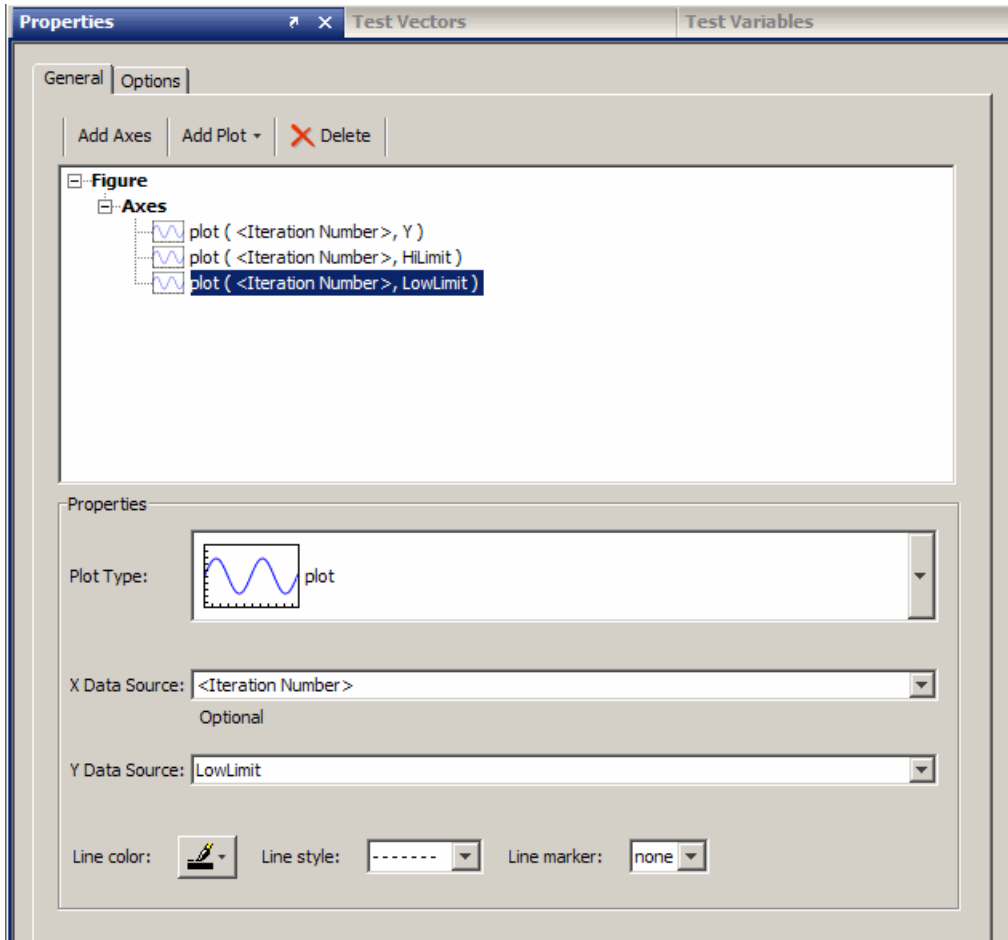


To see the resulting plot, see "Track Output" on page 1-36.

## Define Pass/Fail Criteria

You can define whether your test passes or fails by monitoring the outcome of any or all Limit Check elements during any or all Main Test iterations. Your test's threshold of success can range from the passing of any Limit Check in any single test iteration to the passing of all Limit Check elements in all test iterations. If your test contains no Limit Check elements, there is no notion of pass/fail and no pass/fail information is displayed. (Testing of this type is useful for experimenting with a system or to explore its behavior rather than validate its performance.)

You can set any of the following conditions to define when your test passes:

· All Limit Check elements pass in all test iterations.
· All Limit Check elements pass in any test iteration.
· Any Limit Check element passes in all test iterations.
· Any Limit Check element passes in any test iteration.

You can configure this behavior within the test's **Properties** pane. Click the test name in the **Test Browser** (named **Untitled** by default) to open the test's properties and look for the section labeled **This Test Passes If**.

Using the signal test example that you constructed in this section, set the test to pass if all Limit Check elements pass in all test iterations.
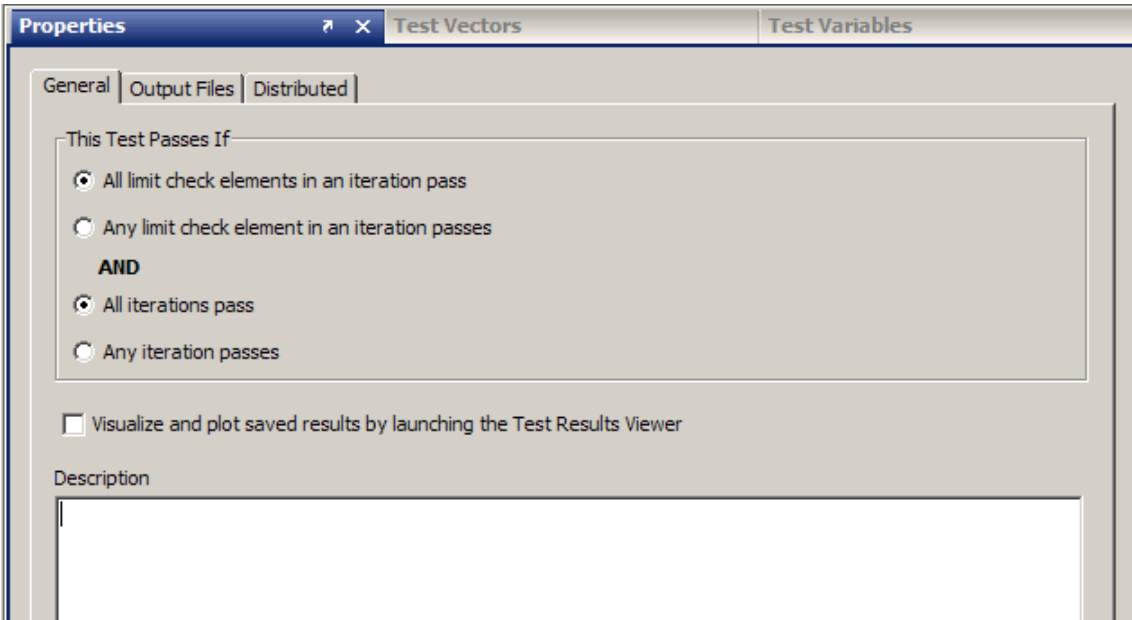
## Save Test Results

You can save the results from the iterations of your test in a MAT-file. You must explicitly specify which test variables to save as test results.

---

**Note:** Test variables that are not saved as a test result will be lost at the end of the test execution.

---

The SystemTest software lets you save results at the end of each iteration. Before running your test, select the **Save Results** section in your test and specify which test variables to save as test results. Click the **New Mapping** button and then select from the drop-down list the name of the test variable to map to a result. You can optionally specify a name for the results that you want to save. By default, the name of the saved result is the same as the test variable. The following figure shows the mapping of test variables to test results.
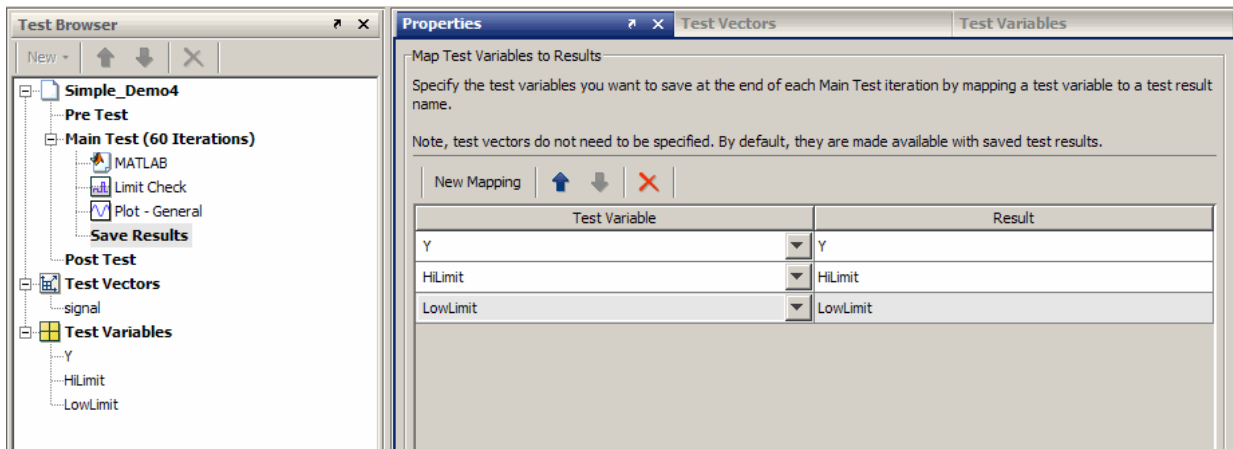
After specifying which test variables to save as test results, specify the name of the MAT-file to use. Using this MAT-file you can reload the test results into the base workspace. By default for a test that is untitled, the SystemTest software names the file `Untitled_results.mat` and puts the file in the same folder as the TEST-File. To change the name or location of the MAT-file, click the test name in the **Test Browser**, then click the **Output Files** tab in the **Properties** pane.

Use the **Select File Names** section to name the results file. Use the **Output Folder Numbering** section to select overwrite behavior. By default, each time you run the test you overwrite this file unless you select the **Create new folder for each run** option. If you select that option, a new folder is created for each run and a new results file is created and put inside the folder. In this case, the `Outputs` subfolder name is appended by successive numbers for each test run. For example, if the folder name is `MyResults`, the first test run will create `MyResults\Outputs` and the next test run results folder will be called `MyResults\Outputs(1)`, followed by `MyResults\Outputs(2)`, etc.
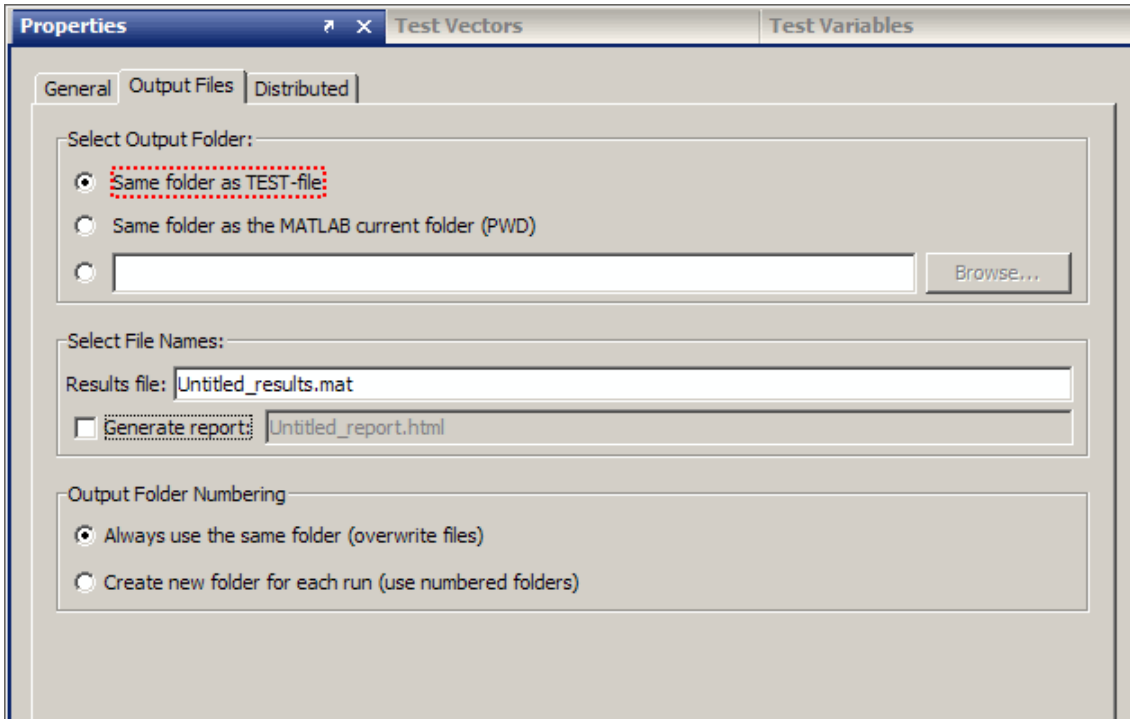
Use the **Select Output Folder** section to designate the location to save the results file, as follows:

- **Same folder as TEST-file** – This default setting saves any output files to the same location as the TEST-File. In this case your test and any output files it generates will be in the same location. You set this location when you select **File > Save** to save your test, or if prompted to save when you close the SystemTest software.

- **Same folder as the MATLAB current folder (PWD)** – Save any output files to the current working directory in MATLAB. You can see the current working directory when you open SystemTest from MATLAB.

- **Browse** – Select the third option and then click the **Browse** button to choose an absolute directory location for the output files. This location should be stable and not read-only.

Note that the location you select here is also where the Test Report will be saved if you generate one by selecting the **Generate report** check box.

---

**Note:** If a file or folder location is read-only, you will get an error when the test runs. For results files and Test Reports to be generated, the files and folder locations must be writable.

---

## Generate a Test Report

When you run your test, the test status appears in the **Run Status** pane. This display contains basic information about your test:

- Time elapsed since your test started running.
- Which section your test is in.
- How many test iterations have passed or failed as defined by any limit checks.
- Whether your test completed successfully.
- Any errors that cause your test to stop.

You can generate and save more detail about the running test by enabling the Test Report, which is a test execution log file in HTML format. This report is useful when you use limit checks in your test and you want to see specific test iterations that passed or failed. For example, instead of just finding that a test iteration failed, the report helps you determine how far a test variable varied from the upper or lower limit you defined in a Limit Check element. It also displays any plots that were generated. This report is also useful for documenting and sharing your test results.

To enable the Test Report:

1   Select the test name in the **Test Browser**, then click the **Output Files** tab on the **Properties** pane.

2   In the **Select File Names** section, select the **Generate report** check box.

3   Use the default name or type a new name in the edit field next to the check box.

4   Use the **Select Output Folder** section to designate the location to save the Test Report:

   • **Same folder as TEST-file** – This default setting saves any output files to the same location as the TEST-File. In this case your test and any output files it generates will be in the same location. You set this location when you select **File > Save** to save your test, or if prompted to save when you close the SystemTest software.

   • **Same folder as the MATLAB current folder (PWD)** – Save any output files to the current working directory in MATLAB. You can see the current working directory when you open SystemTest from MATLAB.

   • **Browse** – Select the third option and then click the **Browse** button to choose an absolute directory location for the output files. This location should be stable and not read-only.

The Test Report is stored in an `Outputs` subfolder in this folder, along with all dependent files, such as plot or Simulink model snapshots. The overwrite options you set for your test results MAT-file also apply to the file name and folder of your report file. To learn how to change these options, see "Save Test Results" on page 1-30 .

Note that the location you select here is also where the test results will be saved.

---

**Note:** If a file or folder location is read-only, you will get an error when the test runs. For results files and Test Reports to be generated, the files and folder locations must be writable.

---

The Test Report contains the following information about the test run, organized by iteration in the report:

   • The test description, if you entered one in the **Description** field of the **Properties** pane of the test.

- A test summary, including start and stop times, number of iterations completed, number of iterations that passed and failed, and final status of the test.
- Pass/fail results of Limit Check elements, by iteration.
- Values for any saved results you captured by setting up mappings in **Saved Results**, by iteration.
- Test vector values, by iteration.
- A snapshot of your model if you use a Simulink element in the test.
- A snapshot of your plot if you use a Vector Plot, Scalar Plot, or General Plot element in your test, by iteration.
- A summary of generated files, with links to them. These can include a Simulink model coverage report and test results.

---

**Note:** Because the Test Report generates while the test is running, this option results in the test taking longer to execute.

---

To see what information the report generates, see "View the Test Report" on page 1-40.

## Save a Test

You can save tests so that you can reuse them later. For example, to save the signal test:

**1** Select **File > Save As** to open the Save file as dialog box.

**2** Select a directory location and enter `mySavedTest` in the **File name** field.

**3** Click **Save**.

The SystemTest software saves the test as `mySavedTest.test` and renames your test as it appears in the **Test Browser**. This does not rename the test results MAT-file or the Test Report file. Their names are controlled separately from the name of the test, as explained in "Save Test Results" on page 1-30.

# Run a Test

After you build a test, you are ready to run it. At run time, the SystemTest software assigns values to test vectors and test variables in the order they appear in the **Test Vectors** and **Test Variables** panes. Each test section runs elements in the order that they appear in the **Test Browser**.

To execute your test, do one of the following:

- Click the **Run** button.
- Select **Run > Run**.
- Press the **F5** key.
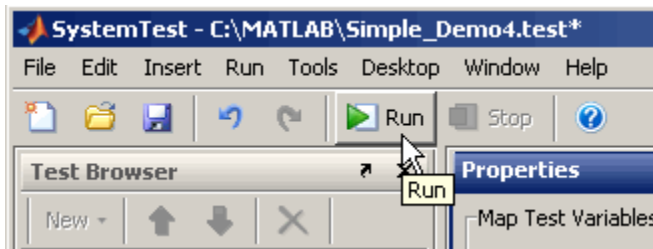


**Note:** While a test is running, you can stop its execution by pressing **Ctrl+C** or clicking the **Stop** button on the toolbar.

## Track Output

While the test runs, the **Run Status** pane shows summary test output, including start and stop times, number of iterations completed, number of iterations that passed and failed, and final status of the test. It will also display any error messages if the test has an error.

**Run Status** ↗ ✕   **Getting Started**      **Desktop Help**

## Generated Files

The following files were generated in C:\Temp\

| Open | Filename |
|------|----------|
| Test Results Viewer | Simple_Demo_results.mat |
| Test Report | Simple_Demo2_report\Simple_Demo2_report.html |

## Final Test Status

| Property | Value |
|----------|-------|
| Start Time | 05-Jul-2007 10:51:25 |
| Stop Time | 05-Jul-2007 10:51:43 |
| Iterations Completed | 60 |
| Iterations Passed | 48 |
| Iterations Failed | 12 |
| Final Status | Failed |

Test Status:   **Failed**

Time Elapsed:   00:00:17

100 %

If your test includes a Plot element, the SystemTest software creates the plot and updates the plot during each iteration. Since Limit Check elements evaluate whether an iteration passed or failed, they directly affect the data that appears in the Test Report and the **Run Status** pane.

In the example test, the plot includes the high and low limits defined in the Limit Check element, to show which test iterations exceed the limits.
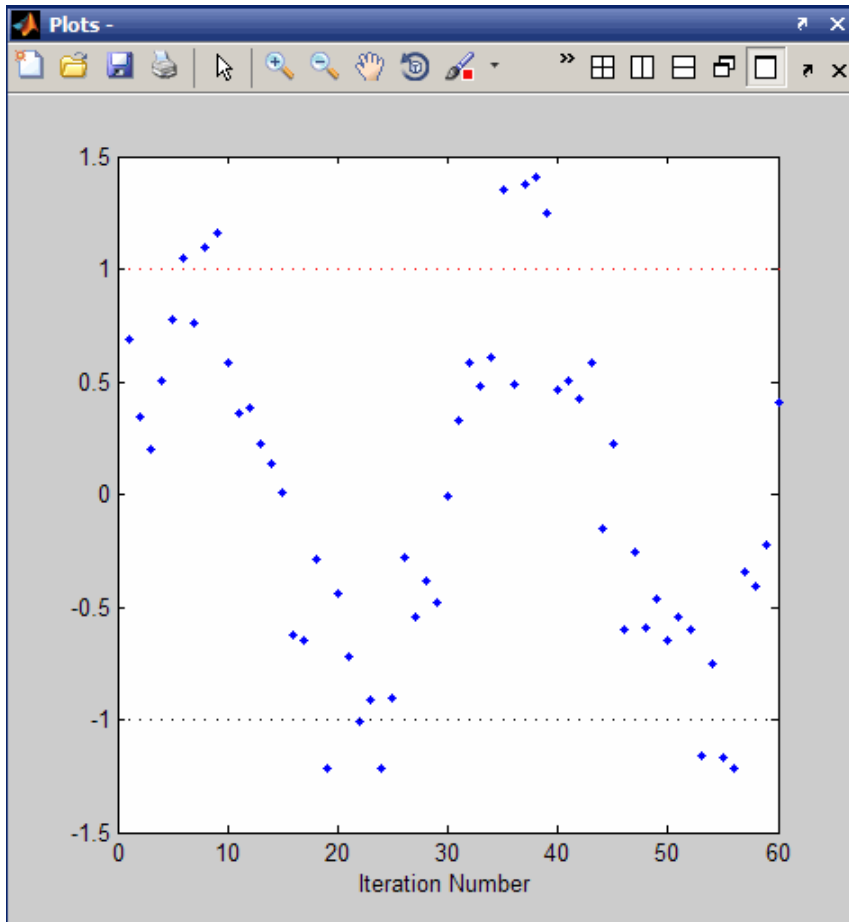


When the test is done running, the **Run Status** pane provides links to generated output. The **Generated Files** section contains a summary of generated files, with links to them,

such as the Test Report, saved test results, and the Simulink model coverage report, if your test uses the model coverage feature.

# Analyze Test Results

After the SystemTest software runs your test, you can explore the results that are generated. This section shows how to:

- View and interpret the Test Report.
- Inspect your test results.

## View the Test Report

When you enable the Test Report, the SystemTest software saves information about each test iteration in an HTML file. To enable the Test Report, check the **Generate report** option on the **Output Files** tab of the **Properties** pane before running your test. The report contains summary information about the test run, snapshots of any plots you used, snapshots of any models you used, pass/fail results of Limit Check elements, and other information. See Test Report for a full description of what the report contains.

After a test runs, you can see the contents of this file by clicking **Tools > Test Report** or using the **Test Report** link in the **Run Status** pane. The generated output resembles the following.
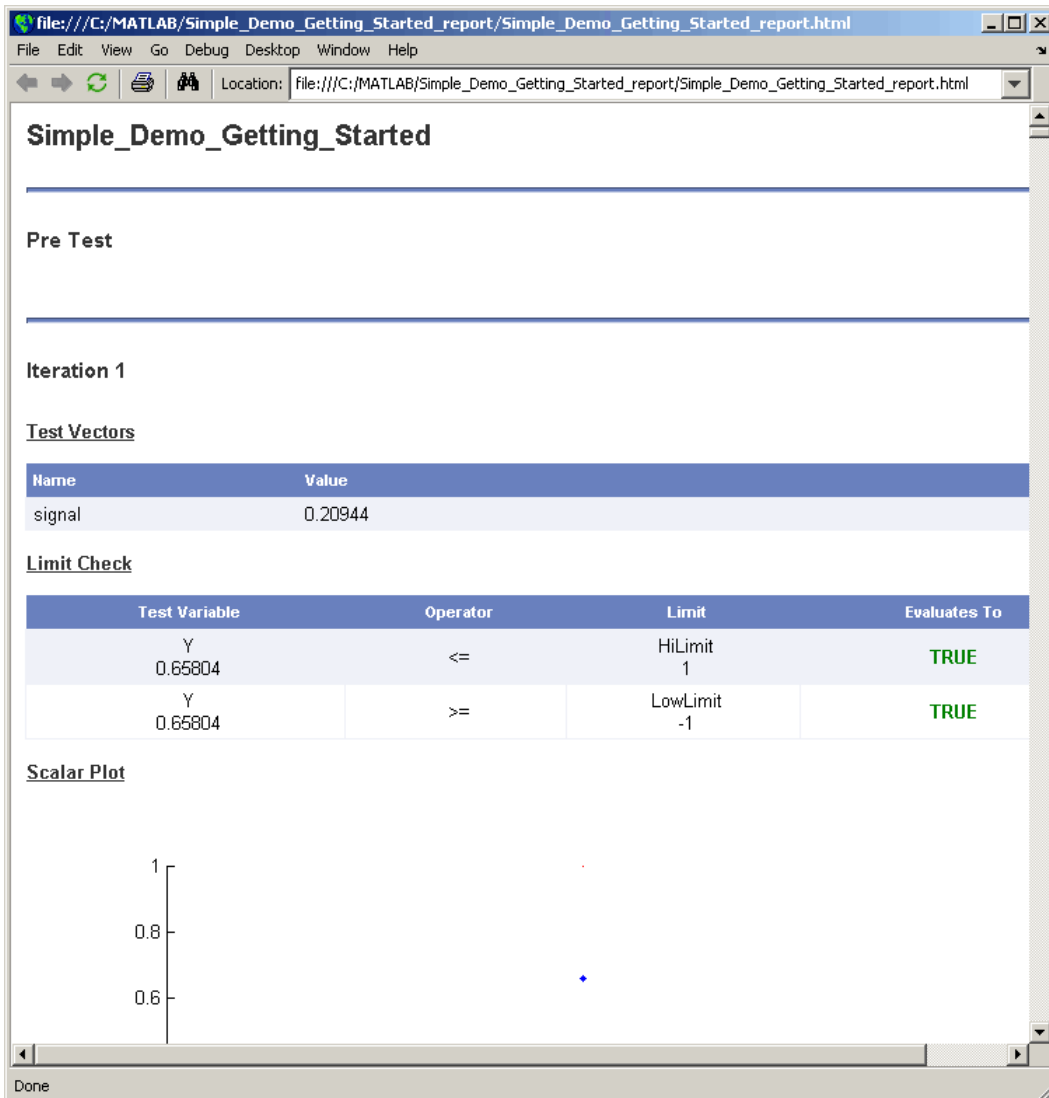
The Main Test section of the report shows each iteration. You see the value of the test vector `signal` and determine the values the Limit Check element used in evaluating whether the test passed. For the first several iterations, the value of Y did not exceed either the high or low limits so the iterations passed. You can also see this in the scalar plot drawn while the test ran. For other iterations that failed, you can scroll through the report to find the values of Y.

## View Test Results

The SystemTest software allows you to view the results you have chosen to save for your test using a workspace variable called `stresults`. It provides access to the test results object, which is useful for comparing the results of separate test runs and for postprocessing test results.

To continue the example, after you run the test, return to MATLAB and type `stresults`. The following summary of the results is returned.

```
>> stresults

stresults =

    Test Results Object Summary for 'Simple_Demo':

            NumberOfIterations: 60
              TestVectorNames: signal
            SavedResultNames: HiLimit, LowLimit, Y
               ResultsDataSet: [60x4 dataset]

    There are no Test Vector Groups associated with this test result object.

    Artifacts associated with this test result object:
      TEST-File (Simple_Demo.test)
```

You can see that the test contains 60 iterations, that it contains a test vector `signal`, and the names of the three saved results are `HiLimit`, `LowLimit`, and `Y`.

To see a more complete list of properties, type:

`get(stresults)`

That displays the following.

```
>> get(stresults)
        ResultsDataSet: [60x4 dataset]
    DerivedResultNames: {}
    NumberOfIterations: 60
      SavedResultNames: {'HiLimit'  'LowLimit'  'Y'}
             StartTime: [2009 6 17 14 54 18.1980]
              StopTime: [2009 6 17 14 54 26.4170]
              TestFile: 'H:\Documents\Simple_Demo.test'
       TestVectorNames: {'signal'}
             Artifacts: {'TEST-File'  'H:\Documents\Simple_Demo.test'}
                   Tag: ''
              UserData: []
              Grouping: {''}
```

The `ResultsDataSet` property contains the test results data in the form of a `dataset` array. This is what you set up using the **Saved Results** node in the **Test Browser**. See "Save Test Results" on page 1-30 for more information on setting up saved results.

To access the `ResultsDataSet` property, type:

```
stresults.ResultsDataSet
```

This returns the test results data in the form of a dataset array.

In the example, a portion of the test results data looks like this:

```
>> stresults.ResultsDataSet

ans =

          signal        HiLimit       LowLimit      Y
    I1    [0.2094]      [1]           [-1]          [0.5226]
    I2    [0.4189]      [1]           [-1]          [0.8125]
    I3    [0.6283]      [1]           [-1]          [0.2148]
    I4    [0.8378]      [1]           [-1]          [1.1565]
    I5    [1.0472]      [1]           [-1]          [0.9984]
    I6    [1.2566]      [1]           [-1]          [0.5486]
    I7    [1.4661]      [1]           [-1]          [0.7730]
    I8    [1.6755]      [1]           [-1]          [1.0414]
    I9    [1.8850]      [1]           [-1]          [1.4086]
    I10   [2.0944]      [1]           [-1]          [1.3309]
```

In the `dataset` array, each row represents a test iteration, labeled using the convention of [`'I'` + `Iteration_Number`]. This example shows the first 10 iterations. Test vector values are listed first, in alphabetical order, followed by test results, listed in alphabetical order, as shown in the above figure. This is a simple way to view the results you set up in **Saved Results**. The test results for all iterations are displayed at the command line, even though only the first ten are only shown here.

You can now plot the results.

**2**

# Working with Test Vectors

# MATLAB Expression Test Vectors

Test vectors define the parameter space or set of test cases you want to run. Test vectors are composed of values that can be derived from a MATLAB expression. You can use any MATLAB expression that evaluates to a 1-by-N matrix or cell array to define your test vector. You must have at least one test vector defined to run a test.

The total number of Main Test iterations is determined by permuting all test vector values. For example, if one test vector is a 1-by-3 array and another is 1-by-2, it would result in a total of six iterations covering all the test vector value combinations.

To add a test vector:

1   Click the **New** button in the **Test Vectors** pane.

    In the Insert New Test Vector dialog box, keep the default test vector type of **MATLAB Expression**.
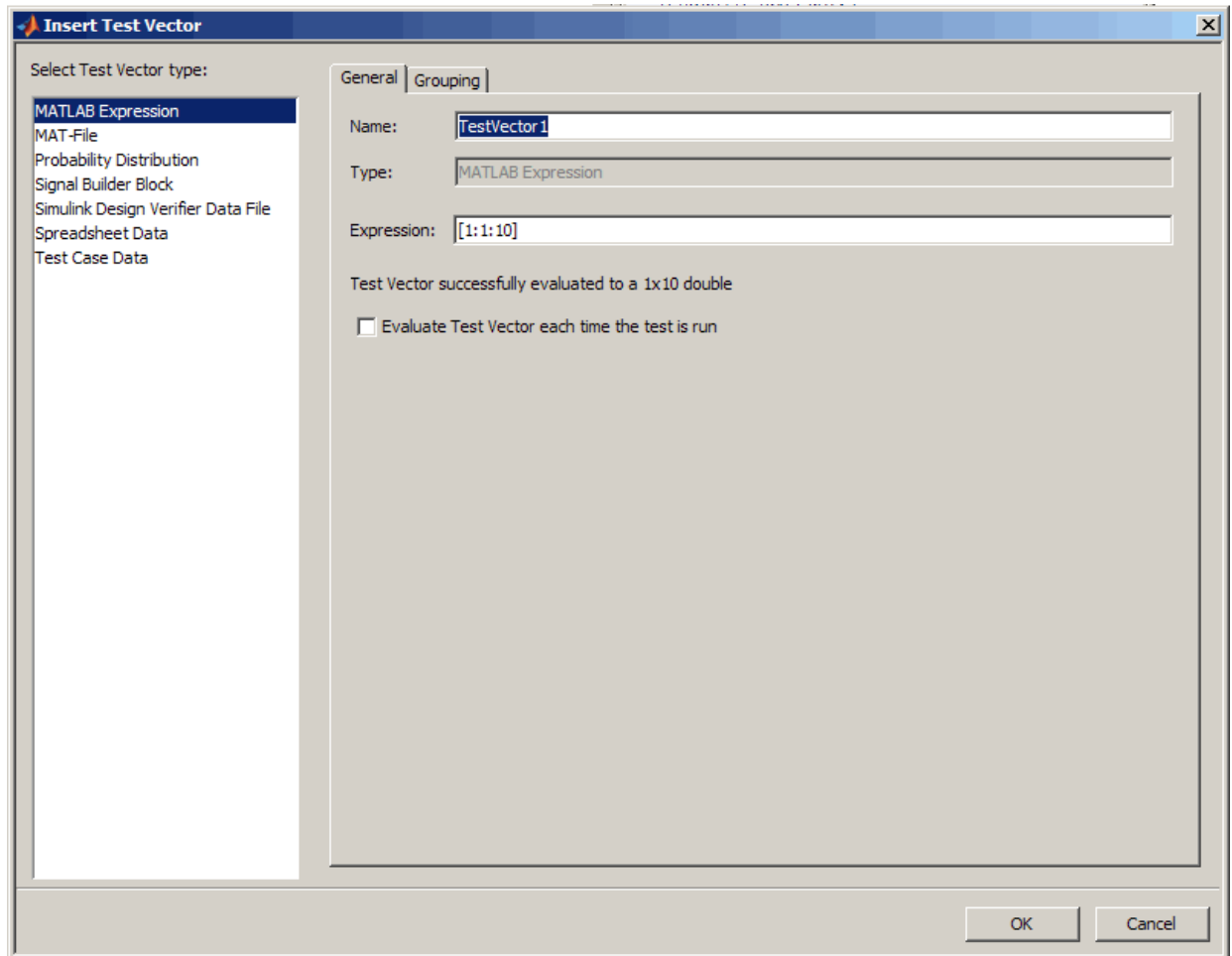
2  Assign a name to the vector in the **Name** field.

3  Enter the value by typing in values or a MATLAB expression in the **Expression** field.

The **Size** field fills in automatically based on what you entered if you press **Enter** or click outside of the **Size** field. For example, if you entered 1 : 1 : 10 in the **Expression** field, the **Size** would be a 1 x 10 double, which means 10 iterations.

**4** Select the **Evaluate Test Vector each time the test is run** option if you want to use new values every time the test is run. For example, if your expression included a `rand` function, a new set of random numbers would be calculated each time. Leave it unselected if you want to use the same values each time the test is run.

**5** Click **OK** in the Insert Test Vector dialog box.

The new vector appears in the **Test Vectors** pane.

---

**Note:** Grouping test vectors determines how they will be iterated through when the test runs. For information on grouping vectors, see "Grouped Test Vectors" on page 2-5.

---

For an example of creating test vectors in a test, see "Create a Test Vector" on page 1-16.

# Grouped Test Vectors

When you create a test vector, it is an ungrouped vector by default, except for Probability Distribution test vectors. You can also create grouped vectors, in order to affect the way iterations are run. By grouping test vectors, they will be indexed simultaneously with the other vectors in their group. Each set of grouped values are then permuted with all the ungrouped test vectors. This gives more control over the flow of tests and is useful for Design of Experiments (DOE) or Monte Carlo-based testing as well as defining signal groups, similar to those defined in the Simulink Signal Builder block.

For example, if you are testing a throttle body controller, you may want to sweep across a range of input level or gain values, while simultaneously selecting different throttle body types, each defined by their mass and damping characteristics.

An example of the vectors in this scenario could look like this:

```
gain = [1 10 100]
mass = [a b c d]
damping = [w x y z]
```

If the `gain` vector is ungrouped, and the `mass` and `damping` vectors are grouped, it will result in `mass` and `damping` being indexed simultaneously for each value of `gain`. The test runs would look like this:

```
Run 1: (1, a, w)
Run 2: (1, b, x)
Run 3: (1, c, y)
Run 4: (1, d, z)
Run 5: (10, a, w)
Run 6: (10, b, x)
Run 7: (10, c, y)
Run 8: (10, d, z)
Run 9: (100, a, w)
Run 10: (100, b, x)
Run 11: (100, c, y)
Run 12: (100, d, z)
```

**Note:** Grouped test vectors must be the same length.

You create a grouped test vector as you do any other vector, by clicking the **New** button in the **Test Vectors** pane. To group a vector, change the selection using the **Grouping**

tab in the Insert Test Vector dialog box. You can group any type of test vector, and you can create multiple test vector groups. You can also group or ungroup test vectors after you create them.

In general, it doesn't usually make sense to group Signal Builder Block test vectors or Simulink Design Verifier Data File test vectors. There are advantages to grouping MATLAB Expression, Probability Distribution, and Spreadsheet Data test vectors at times, depending on your test goals. One of the main advantages to grouping is for Monte Carlo-based testing, as described by the example above.

To group a test vector:

1 Create a test vector and configure it in the **General** tab of the Insert Test Vector dialog box.

2 Click the **Grouping** tab in the Insert Test Vector dialog box.

3 Select the **Assign test vector to a group** option.

   A group is created and given the default name of `Group1`, as shown here.
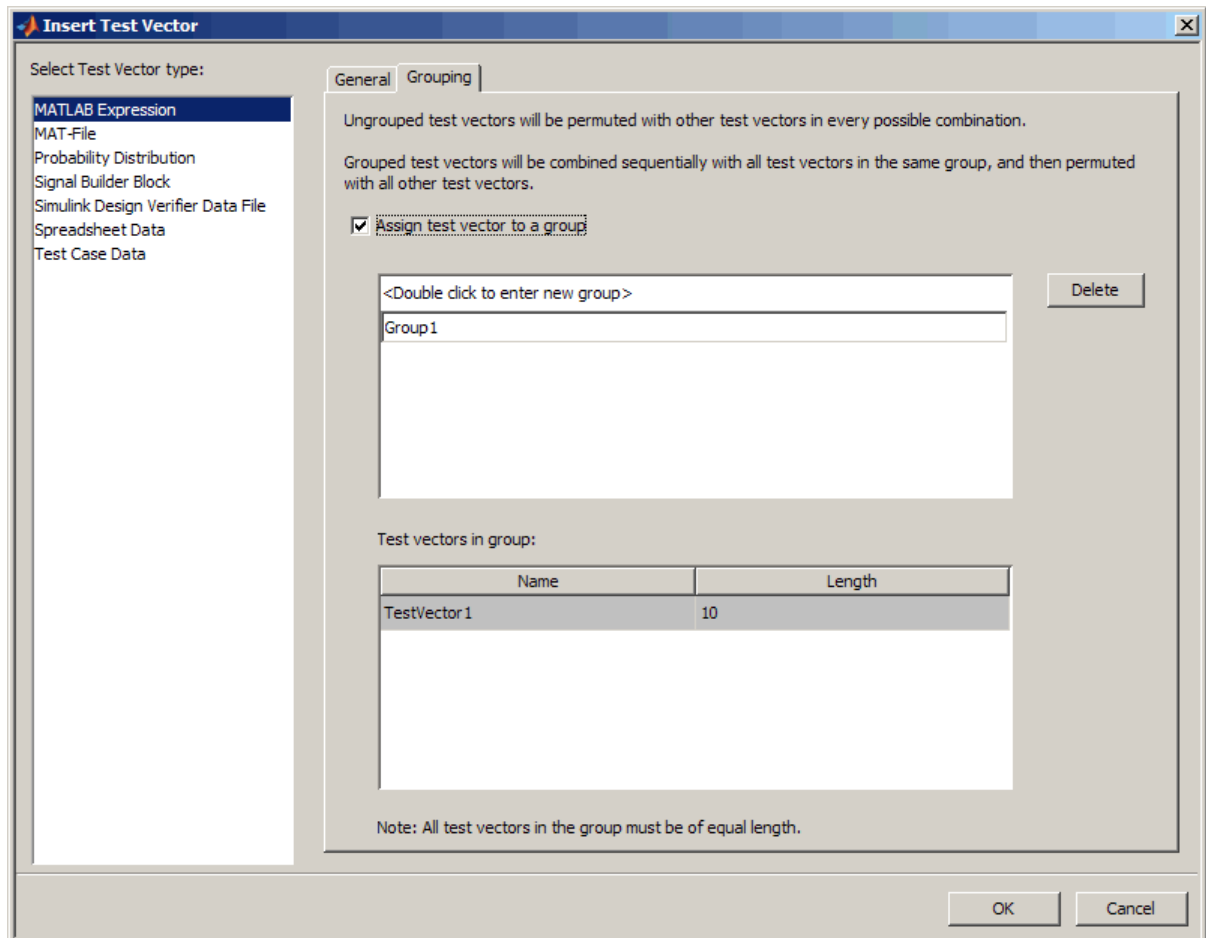
4   To change the name, type the new name over the default name and press **Enter**.

5   Click **OK** in the Insert Test Vector dialog box.

In the **Test Vectors** pane, the name of the group is displayed in the table.

6   Now if you create another test vector, you can add it to the same group as the first one. To do this, click the **New** button again.

7   Select the test vector type and configure it in the **General** tab.

8   Click the **Grouping** tab, and select the **Assign test vector to a group** option.

Note that test vectors in a group must all be the same length.
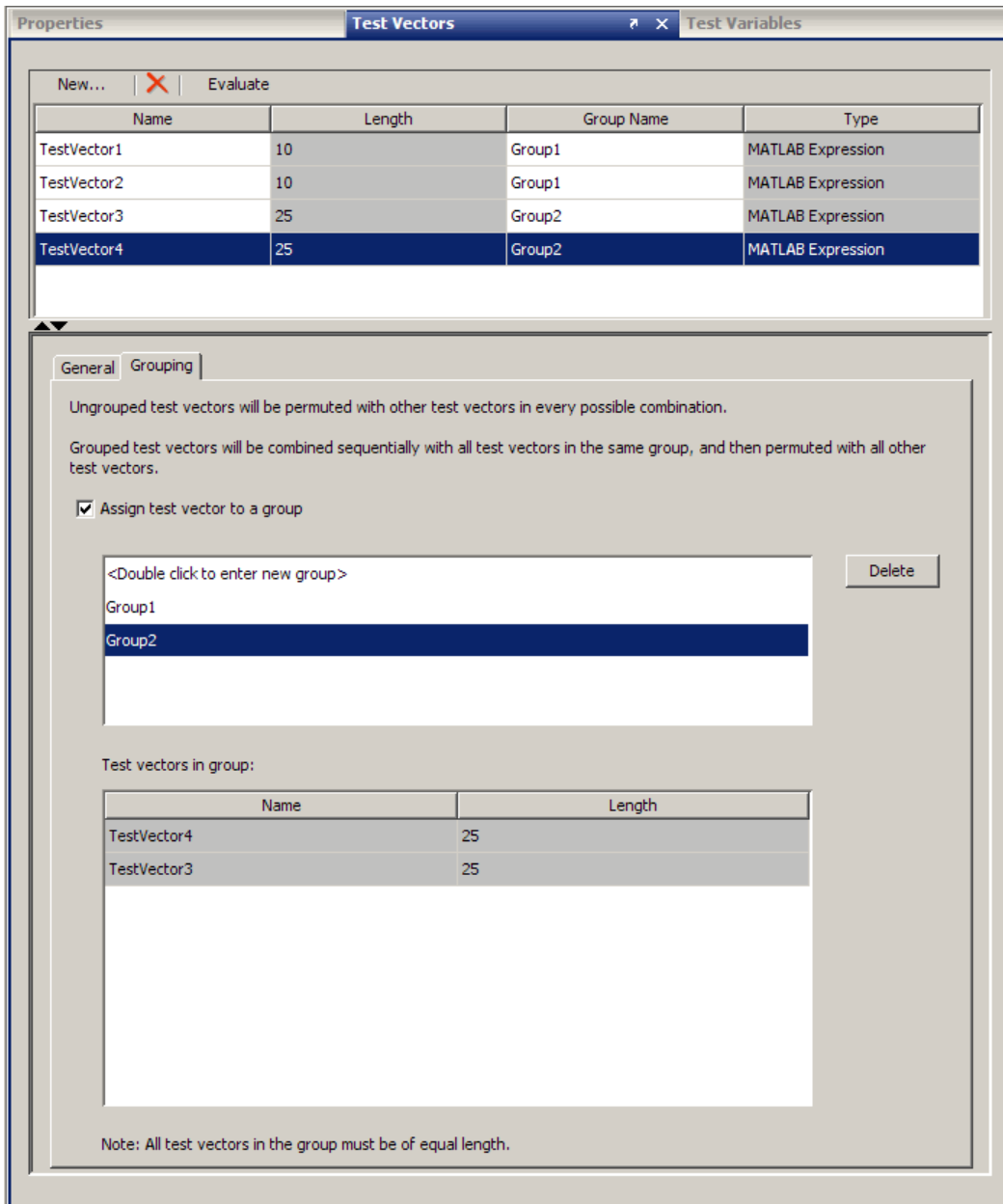
If you already have one test vector group, the new vector is placed in that group by default.
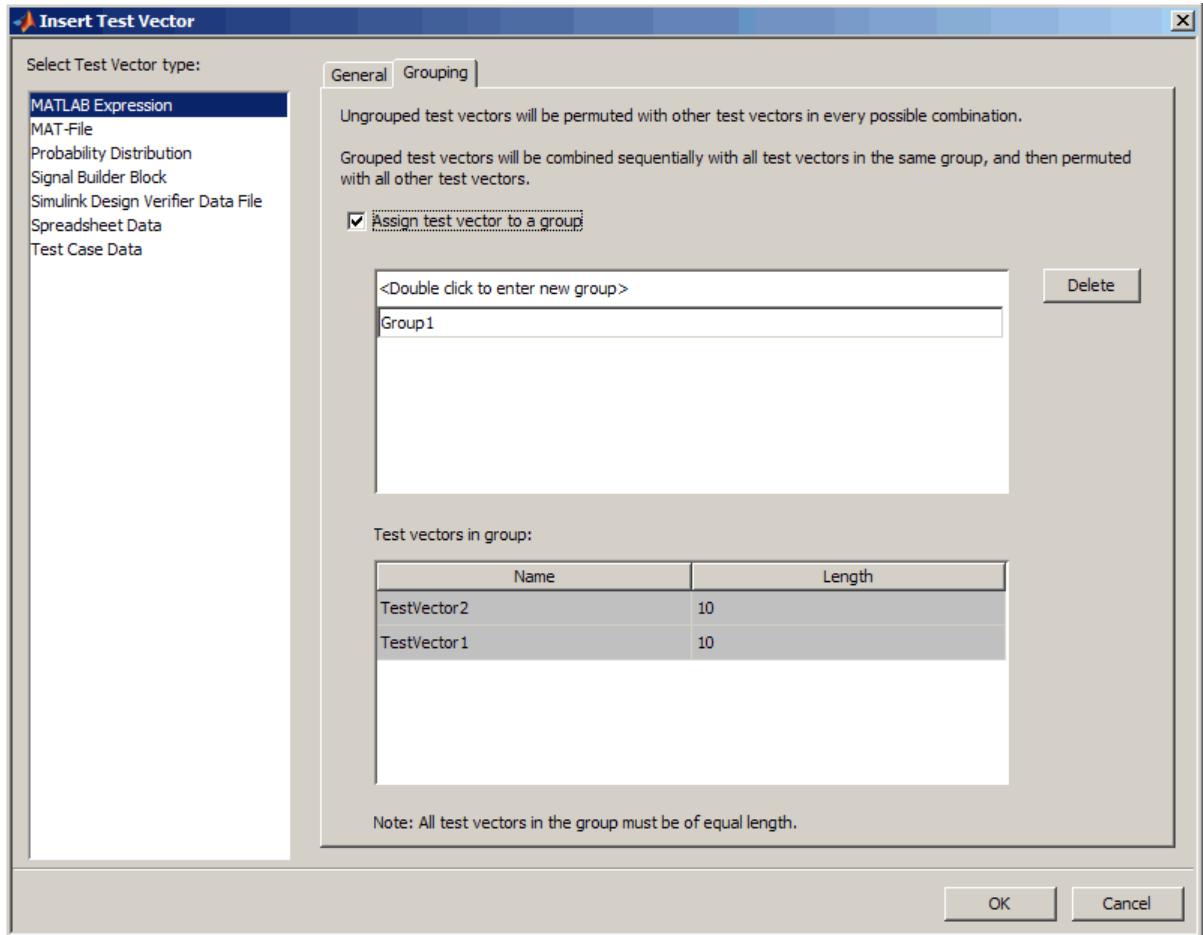


9   Click **OK** in the Insert Test Vector dialog box.

You can create multiple test vector groups. Once you have multiple groups, when you create new test vectors, you can select which group to put them in as you create them. The following figure shows `Group1` containing `TestVector1` and `TestVector2`, and `Group2` containing `TestVector3` and `TestVector4`.

| Properties | **Test Vectors** | ⤢ ✕ | Test Variables |
|---|---|---|---|

New... | ✕ | Evaluate

| Name | Length | Group Name | Type |
|---|---|---|---|
| TestVector1 | 10 | Group1 | MATLAB Expression |
| TestVector2 | 10 | Group1 | MATLAB Expression |
| TestVector3 | 25 | Group2 | MATLAB Expression |
| TestVector4 | 25 | Group2 | MATLAB Expression |

General | Grouping

Ungrouped test vectors will be permuted with other test vectors in every possible combination.

Grouped test vectors will be combined sequentially with all test vectors in the same group, and then permuted with all other test vectors.

☑ Assign test vector to a group

| <Double click to enter new group> |
|---|
| Group1 |
| Group2 |

Delete

Test vectors in group:

| Name | Length |
|---|---|
| TestVector4 | 25 |
| TestVector3 | 25 |

Note: All test vectors in the group must be of equal length.

You can also create groups after test vectors are already created by editing a test vector in the **Test Vectors** pane. Select a test vector in the table to edit its properties in the editor area below the table. There you can add it to a group using the **Grouping** tab. You can also add it to a group in the table by clicking in the **Group Name** column.

**Managing Test Vector Groups**

You can modify groups to ungroup a test vector, move a test vector to another group, rename a group, or delete a group.

- **Ungroup a test vector** — To remove a test vector from a group, select it in the test vectors table, then click the **Group Name** column. Use the down-arrow to select the first entry, which is a blank space. The **Group Name** column will then be empty for that test vector, indicating it is no longer in a group.

- **Move a test vector to another group** — To move a test vector from one group to another, select it in the test vectors table, then click the **Group Name** column. Use the down arrow to select the group to move it to. The **Group Name** column will then show the new group name.

- **Rename a group** — You can change the name of a test vector group either in the table or in the editor area. Renaming a group in the table results in the group name for a single test vector being changed. Renaming a group in the editor area results in the name being changed for all vectors in the group.
  To rename a group for a single test vector, select that vector in the table, then click in the **Group Name** column. Type a new name and press **Enter**.
  To rename a group for all test vectors in the group, select one of the test vectors in the table. Then in the **Grouping** tab in the editor area, select that group name in the upper section and type a new name. Press **Enter**. You then see all of the test vectors in that group change to the new name in the table.

- **Delete a group** — To delete a test vector group, select one of the test vectors in the table that is in that group. Then in the editor area, under the **Grouping** tab, that group name will be selected. Click the **Delete** button on the **Grouping** tab. The group is deleted and all test vectors belonging to that group become ungrouped.

# Test Vectors and the MATLAB Workspace

The SystemTest software has its own internal workspace that it uses to manage test variables and test vectors independently. However it does leverage the MATLAB workspace during test execution, and when using a MATLAB element.

During test execution, SystemTest test variables and test vectors are evaluated in the MATLAB base workspace. Then at the end of test execution, they are cleared out and the MATLAB base workspace is restored to what it was before the test execution.

When using a MATLAB element in the SystemTest software, you can reference a variable in the base workspace without having to create a test vector or test variable in the SystemTest software. However the SystemTest software will not be aware of this data, so you could not make use of it in any other element type or in saved results. You can only access it from a MATLAB element. If you need to use it in other elements, you can create test variables or test vectors in the SystemTest software.

# MAT-File Test Vectors

The MAT-File test vector offers an easy way for you to use data from a MAT-file in the SystemTest software.

To add a test vector:

**1** Click the **New** button in the **Test Vectors** pane.

In the Insert New Test Vector dialog box, select the test vector type of **MAT-File**.
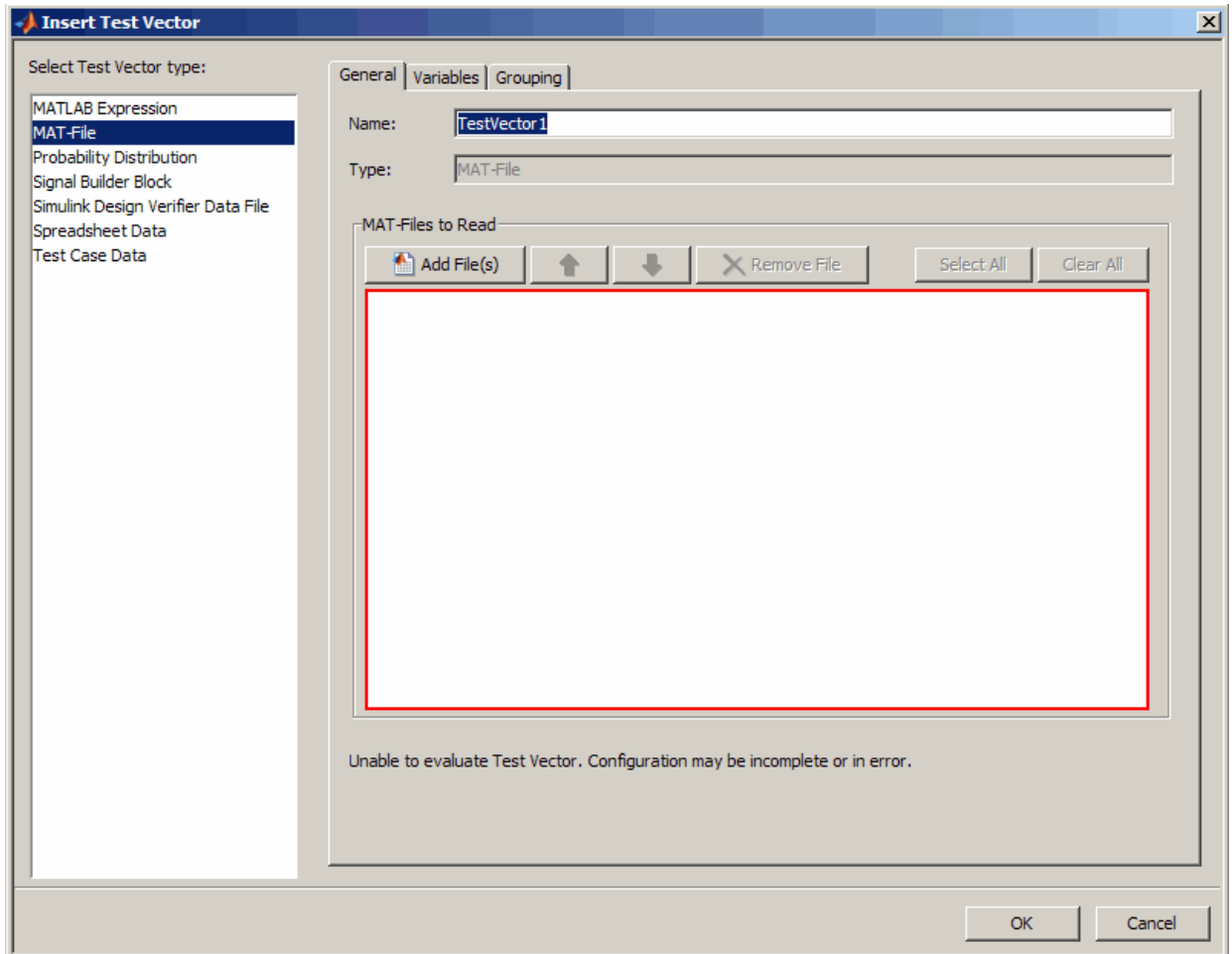
The red border indicating that the element is in an error state is normal, and will go away once you add file(s) in step 4.

**2** Assign a name to the vector in the **Name** field.

**3** Click the **Add File(s)** button.

**4** In the Select MAT-File dialog box, browse for your MAT-file(s). You can select multiple files at the same time. Only MAT-files can be added. Other file types

produce an error. After selecting the file(s), click the **Open** button to bring them into the test vector.

In the **MAT-Files to read** table on the **General** tab, MAT-files that are checked will be used in the test. Unchecking a file means it will not be included in the test.

5    Click the **Variables** tab. All the common variables contained in all the selected MAT-file(s) you added appear in the table.
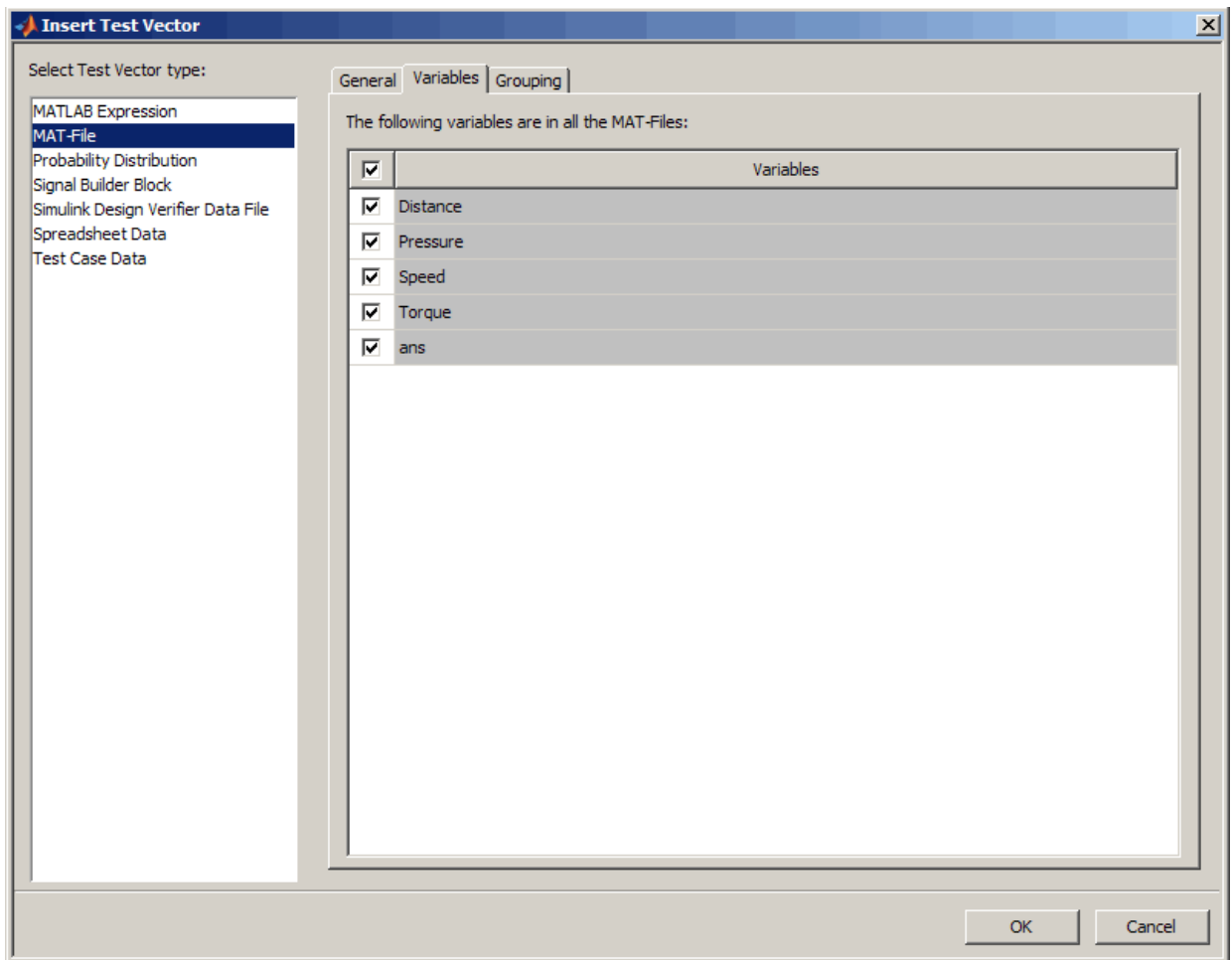
Note that the variables are sorted in alphabetical order. If you have multiple MAT-files, only variables that are common across all files appear in the table.

Variables that are checked will be used in the test. Unchecking a variable means it will not be included in the test. In the example above, all variables except for `ans` will be used in the test.

Checking or unchecking the checkbox in the table header will select or unselect all variables. It is a Select All/Unselect All toggle option.

**6** MAT-File test vectors are ungrouped by default. On the **Grouping** tab, you can select the **Assign test vector to a group** option if you want to group the test vector.

Grouping test vectors is useful for reducing the number of iterations to execute. It means that the SystemTest software will sequentially combine values for all grouped test vectors, instead of permuting their values. See "Grouped Test Vectors" on page 2-5 for more information on grouped test vectors.

**7** Click **OK** in the Insert Test Vector dialog box.

The new vector appears in the **Test Vectors** pane.

**Important Usage Notes**

- If you use multiple MAT-Files in a test vector, only commonly named variables included in all of the files will be read and used. For example, if you have variables in MAT-file A called `Speed`, `Distance`, and `Pressure`, and in MAT-file B you have variables `Speed`, `Pressure`, and `Torque`, only `Speed` and `Pressure` will be shown since they are included in both MAT-Files. `Distance` and `Torque` will not be used since they do not exist in both files.

- If the order of execution of the MAT-files is important, then use the up and down arrows to order the files accordingly in the test vector table. Each MAT-file is one iteration of the test vector, and they are executed in the order they appear in the table.

- The test vector is evaluated every time the test is run – that means the data is read from the MAT-File(s) every time the test is run.

- If a MAT-File test vector is mapped to the inport blocks in a Simulink element using the **All Inport blocks are mapped** option, the model is simulated using all the variables that are selected in the **Variables** table in the test vector. If it is mapped to the inport blocks using the **Individual Inport blocks are mapped** option, the model is simulated with individually selected variables from the MAT-file.

- Checking or unchecking the checkbox in the **Variables** table header will select or unselect all variables. It is a Select All/Unselect All toggle option. This option affects the variables selection behavior when you add or remove or select or unselect MAT-files in the MAT-file list on the **General** tab.

  For example, if the checkbox is selected (to Select All variables) and then a MAT-file is added/removed or selected/unselected, all common variables will be selected by default.

  If the checkbox is unselected (to Unselect All variables) and then a MAT-file is added/removed or selected/unselected, all common variables will be unselected by default.

# Randomized Test Vectors with Probability Distributions

| In this section... |
|---|
| |
| |
| |
| |
| |

## Probability Distributions in Test Vectors

The SystemTest software provides an easy way to generate randomized test vector values for your test. You can use probability distribution functions to set up test vectors, which is useful for performing Monte Carlo analyses.

If you have the Statistics and Machine Learning Toolbox™ software, the SystemTest software integrates with it to provide use of some of its probability distribution functions, such as exponential, gamma, lognormal, T (Student's t), and Weibull. If you do not have the Statistics and Machine Learning Toolbox software, you can use the MATLAB probability distribution functions normal (Gaussian) and uniform.

## Create a Test Vector with Probability Distributions

You can use a probability distribution when you create or edit a test vector. To use a probability distribution:

1   In the **Test Vectors** pane, click the **New** button.
2   In the Insert Test Vector dialog box, select **Probability Distribution** as the test vector type.
3   Enter a name for the new vector in the **Name** field.
4   Select a distribution function from the **Distribution** list.
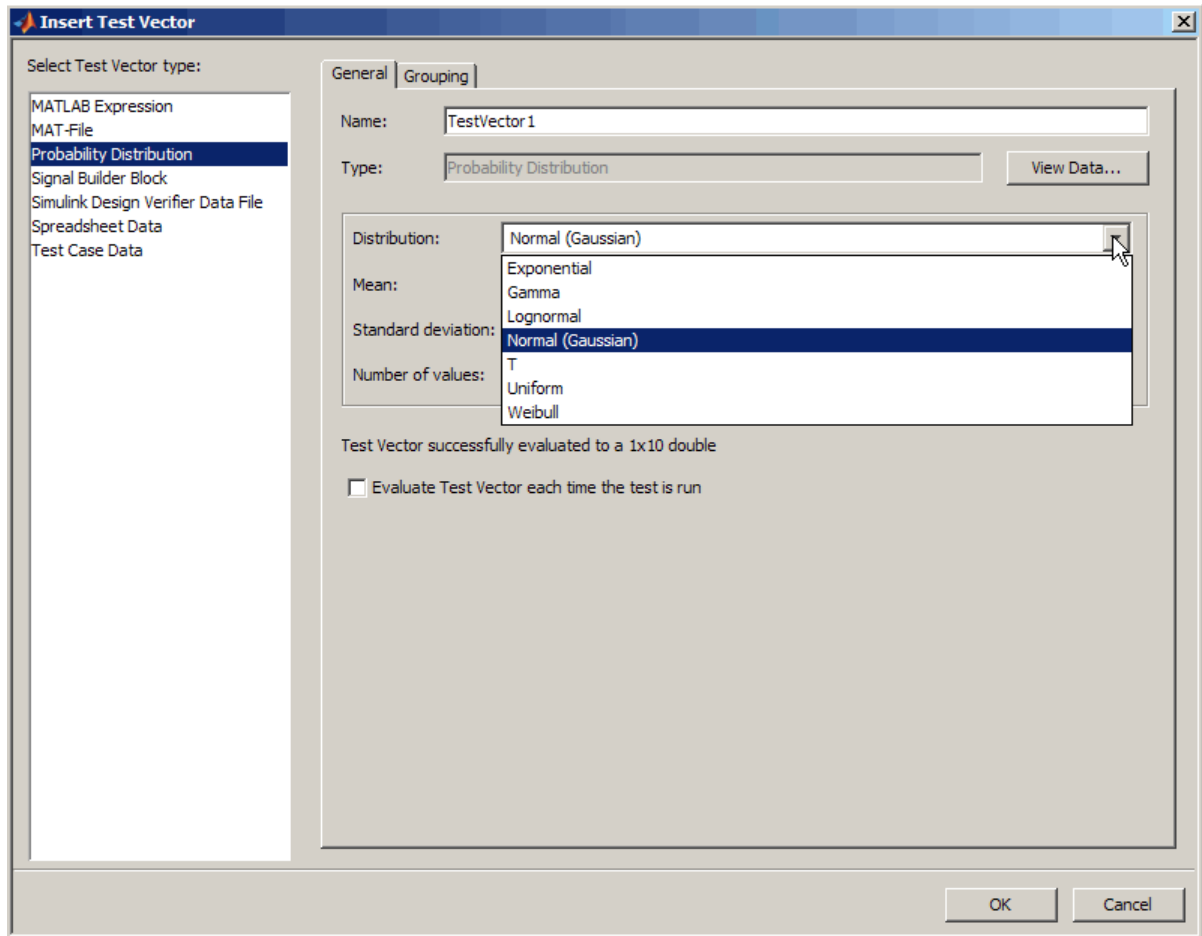
If you have the Statistics and Machine Learning Toolbox software, all of the functions shown in the figure appear in the list. If you do not have this toolbox, you can use normal (Gaussian) and uniform.

For information on the distribution functions, see "The Probability Distributions" on page 2-24.

**5** Once you select a distribution, the relevant options appear. Fill in the parameters for your distribution.

**6** After setting the relevant probability parameters, type in the **Number of values** you want to use. That is the number of values you would like to generate for the test vector.

The **Number of values** must be a positive integer. It must also be the same value for all of your probability distributions because the vector is grouped.

**7** If you want to see the data you have configured before running the test, click the **View Data** button. This displays a histogram visualization of the probability distribution data. If you are not satisfied with the data as it is configured, you can adjust one or more of the parameters and hit **Enter** to see the changes in the figure window.

For more information on viewing the data, see "View Data While Configuring the Test Vector" on page 2-22.

**8** Select the **Evaluate Test Vector each time the test is run** option if you want to use new values every time the test is run. For example, for the probability distribution, a new set of values for the parameters (such as Mean) would be calculated each time. Leave it unselected if you want to use the same values each time the test is run.

If you are doing Monte Carlo testing and you want repeatability of the data, do not use this option.

**9** On the **Grouping** tab, keep the default of **Grouped**, or select **Ungrouped**.

Randomized test vectors with probability distributions are grouped by default, as indicated by **Grouped** being selected.

Grouping test vectors is useful for reducing the number of iterations to execute. It means that the SystemTest software will sequentially combine values for all grouped test vectors, instead of permuting their values. In the case of randomized test vectors, grouping avoids introducing additional variation into your test. See Creating Grouped Test Vectors for more information on grouped test vectors.

**10** Click **OK** in the Insert Test Vector dialog box.

The new vector then appears in the **Test Vectors** pane.
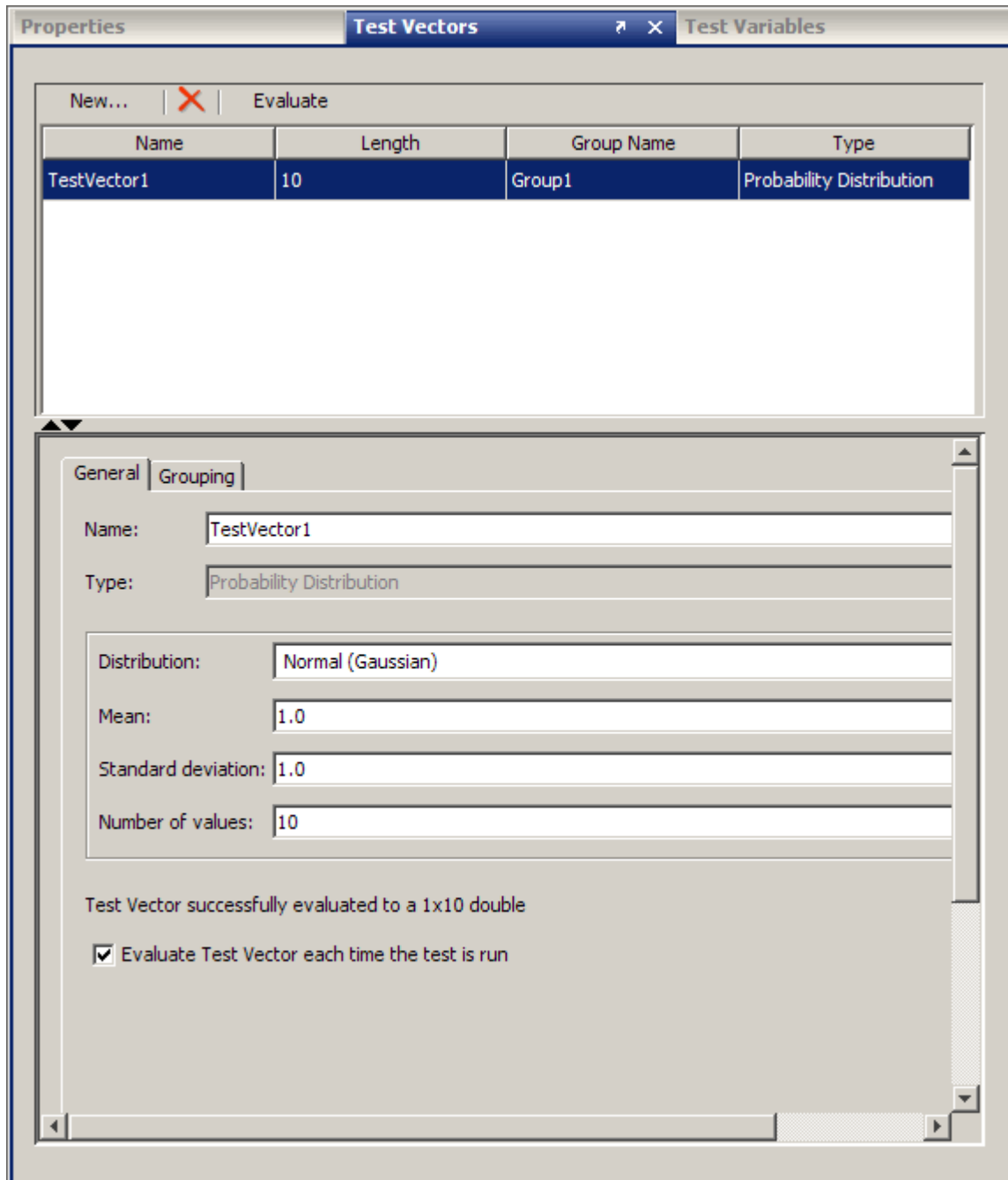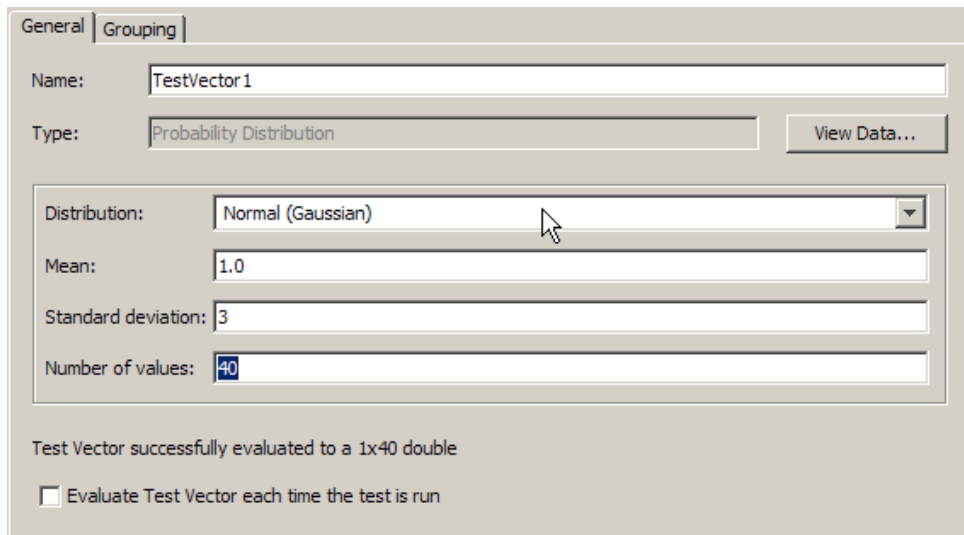
## View Data While Configuring the Test Vector

You can view your probability distribution data while configuring the test vector, without having to run the test. You can quickly inspect the test vector data for outliers, data range coverage, or correctness of the test function before running the test. This allows you to make necessary adjustments until you have data you are satisfied with, which saves time.

To view data while configuring a test vector:

**1** Create the test vector by clicking the **New** button in the **Test Vectors** pane.

**2** In the Insert Test Vector dialog box, select **Probability Distribution** as the test vector type.

**3** Select a distribution function from the **Distribution** list.

**4** Once you select a distribution, the relevant parameters appear. Fill in the parameters for your distribution.

In this example, Normal (Gaussian) is shown, with a mean of 1.0, standard deviation of 3, and 40 values.



**5** Click the **View Data** button on the **General** tab.

**6** The data viewer window displays the data you configured in a histogram visualization. The values are displayed on the *x-axis*, and in this case they range from approximately -6 to 9. The parameters are also displayed textually in the figure window in the upper right corner. For comparison purposes, a light orange line showing the "ideal" probability distribution is also displayed on top of your data.
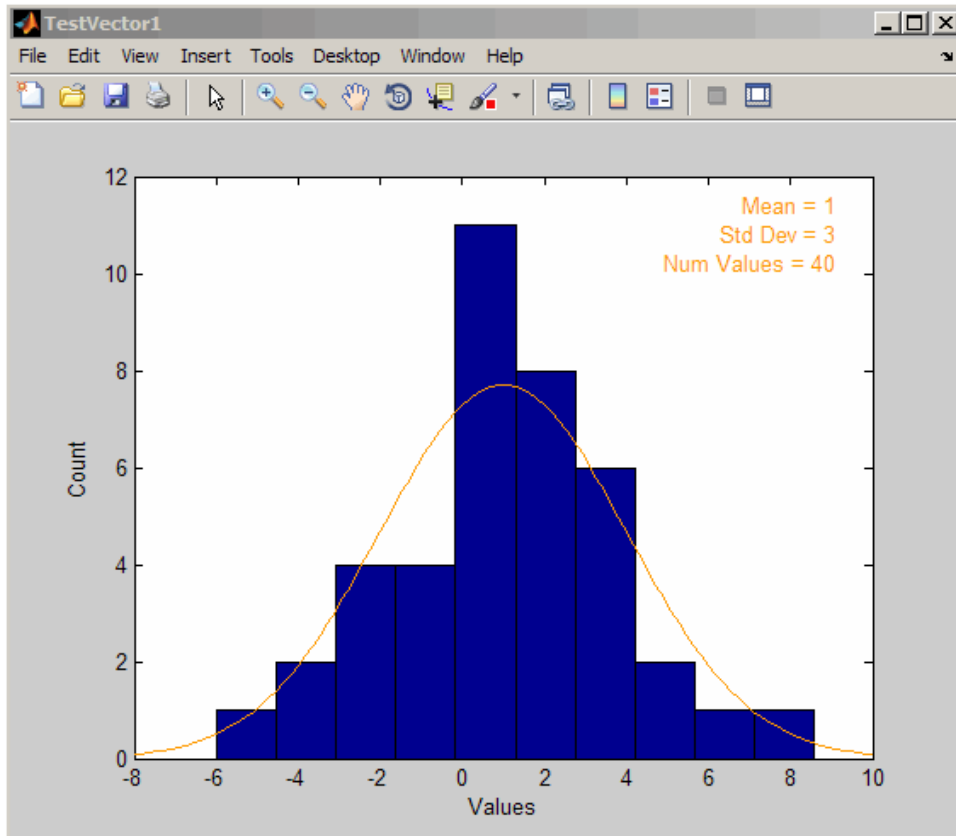


**7** When satisfied with the data that is shown, click **OK** to finish creating the test vector.

**8** If you are dissatisfied with the data, change one or more parameters and redisplay it. In this case, change the standard deviation from 3 to 2. To change a value, type a new value in the parameter you want to change and either press **Enter** or click outside of the field. The figure window automatically updates to display the new data.
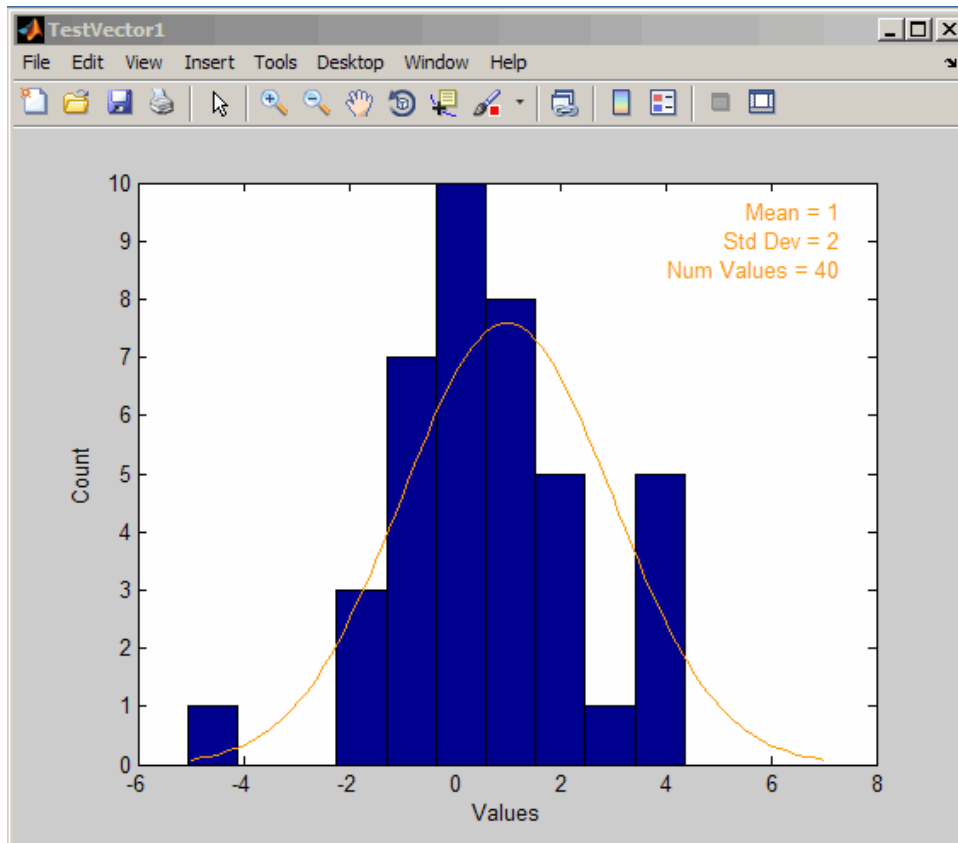
You can also view and modify the test vector data any time after creating a test vector. Access the data viewer by clicking the **Test Vectors** tab in the **Properties** pane, then selecting a test vector from the list. That test vector then becomes editable, and you can click the **View Data** button on the **General** tab.

## The Probability Distributions

If you have the Statistics and Machine Learning Toolbox software, the SystemTest software integrates with it to provide use of some of its probability distribution functions, such as exponential, gamma, lognormal, T (Student's t), and Weibull. If you do not have
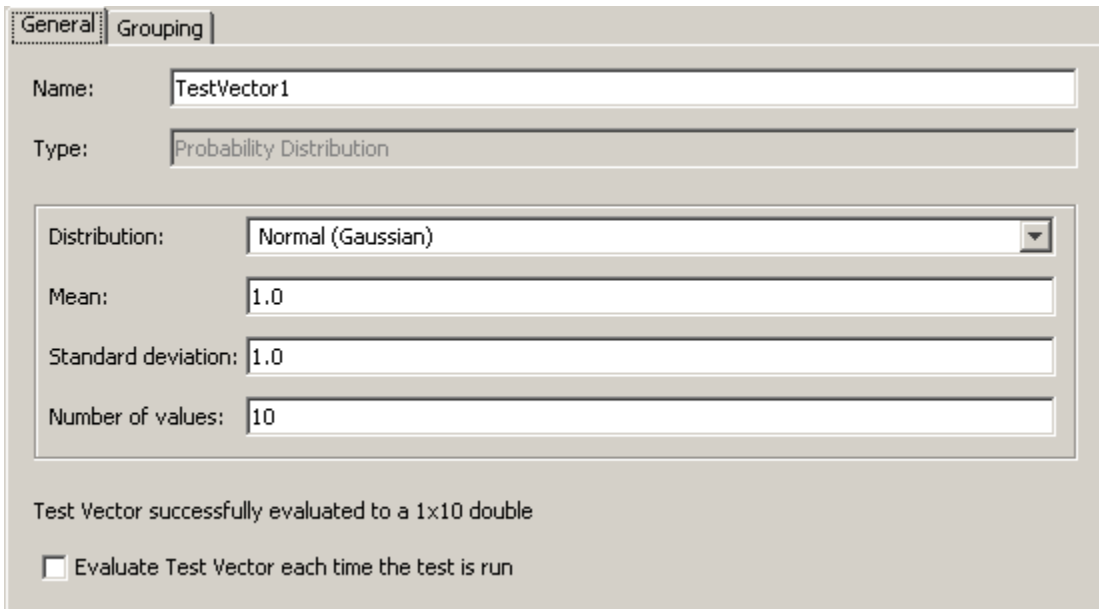
the Statistics and Machine Learning Toolbox software, you have access to the MATLAB probability distribution functions normal (Gaussian) and uniform.

The SystemTest software supports the distribution functions shown in the following sections. Select the **Probability Distribution** test vector type in the Insert Test Vector dialog box to access the functions.

The Insert Test Vectors dialog box shows fields specific to the distribution you pick in the list, as shown in the sections below. In each case, enter values for the function-specific parameters, and then enter the **Number of values** you want to generate for the test vector.

### Normal (Gaussian)

The normal distribution is a two-parameter family of curves. The first parameter is the mean. The second parameter is standard deviation. Normal is often used for data that is symmetrical about the mean.



Normal uses the function `randn` and takes parameters for **Mean** and **Standard deviation**. The SystemTest software uses the following calculation for normal:

```
mean + Std_Dev * randn(1, #values)
```

For more information, see `randn` in the MATLAB documentation.

### Uniform

The uniform distribution (also called rectangular) has a constant probability density function between its two parameters, the minimum and the maximum.

The uniform distribution is appropriate for representing the distribution of round-off errors in values tabulated to a particular number of decimal places.



Uniform uses the function `rand` and takes parameters for **Minimum value** and **Maximum value**. The SystemTest software uses the following calculation for uniform:

```
min + (max-min) * rand(1, #values)
```

For more information, see `rand` in the MATLAB documentation.

### Exponential

The exponential distribution is a special case of the gamma distribution. The exponential distribution is special because of its utility in modeling events that occur randomly over time.

Exponential is often used to model the time between independent events that happen at a constant average rate. For example, you could use it for the time it takes a radioactive particle decays, or the time between messages sent over a network.
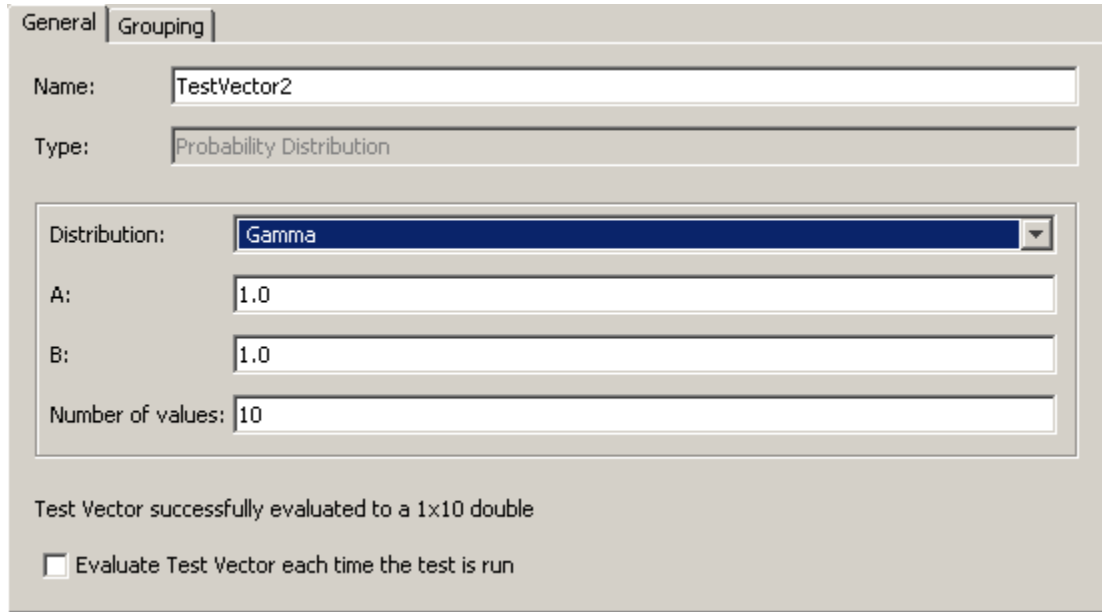


Exponential uses the function `exprnd` and takes one parameter for **Mean**.

For more information, see "Exponential Distribution" in the Statistics and Machine Learning Toolbox documentation.

### Gamma

The gamma distribution models sums of exponentially distributed random variables.



Gamma uses the function `gamrnd` and takes parameters for **A** and **B**.

For more information, see "Gamma Distribution" in the Statistics and Machine Learning Toolbox documentation.

### Lognormal

The normal and lognormal distributions are closely related. The lognormal distribution is applicable when the quantity of interest must be positive, since log(X) exists only when X is positive.

Lognormal can be used to model something that can be thought of as the multiplicative product of many small independent factors. A common example is the long-term return rate on a stock investment, because it can be considered as the product of daily return rates.



Lognormal uses the `lognrnd` function and takes parameters for **Mean** and **Standard deviation**.

For more information, see "Lognormal Distribution" in the Statistics and Machine Learning Toolbox documentation.

### T

The T (Student's t) distribution is a family of curves that depend on a single parameter v (the degrees of freedom). As v goes to infinity, the T distribution approaches the standard normal distribution.

T is often used to estimate properties when the sample size is small.



T uses the `trnd` function and takes one parameter for **Degrees of freedom**.

For more information, see "Student's t Distribution" in the Statistics and Machine Learning Toolbox documentation.

### Weibull

The Weibull distribution is an appropriate analytical tool for modeling the breaking strength of materials. Current usage also includes reliability and lifetime modeling. The Weibull distribution is more flexible than the exponential distribution for these purposes.



Weibull uses the function `wblrnd` and takes parameters for **A** and **B**.

For more information, see "Weibull Distribution" in the Statistics and Machine Learning Toolbox documentation.

## Create Test Vectors with Probability Distributions

Many models must take into account the effect of evaluating uncertainty in model parameters. In this example the tester needs to account for uncertainty in electric motor characteristics that come off the production line so the tester defines the model's parameters as distributions of values, rather than as single fixed values. The tester then performs a Monte Carlo simulation, running the model repeatedly with random combinations of parameter values to account for variability in manufacturing.

In this case, the tester defines the uncertain motor parameters as test vectors. The test varies parameters for armature resistance, armature inductance, and shaft inertia.

To create the first vector, for armature resistance:

1  In the **Test Vectors** pane, click the **New** button.

2  In the Insert Test Vector dialog box, select **Probability Distribution** as the test vector type.

3  Enter `ArmatureResistance` in the **Name** field.

4  In the Insert Test Vector dialog box, use the default distribution, normal (Gaussian).

   You do not need to have the Statistics and Machine Learning Toolbox software installed to use normal (Gaussian) since it is included with MATLAB.

5  In the **Mean** field, enter `1.71`.

6  In the **Standard deviation** field, enter `.056`.

7  In the **Number of values** field, enter `1000`.

General | Grouping |

Name:     ArmatureResistance

Type:     Probability Distribution     View Data...

Distribution:     Normal (Gaussian)

Mean:     1.71

Standard deviation: .056

Number of values:     1000

Test Vector successfully evaluated to a 1x1000 double

☐ Evaluate Test Vector each time the test is run

For this vector, the test is varying armature resistance up to a standard deviation of .056, around a mean of 1.71, and using 1000 values.

**8** Click the **View Data** button to see a visualization of the test vector data that you configured. This displays a histogram visualization of the probability distribution data that will be used when the test is run. If you are not satisfied with the data as it is configured, you can adjust one or more of the parameters and hit **Enter** to see the changes in the figure window. In this case, we keep the data, as shown here.
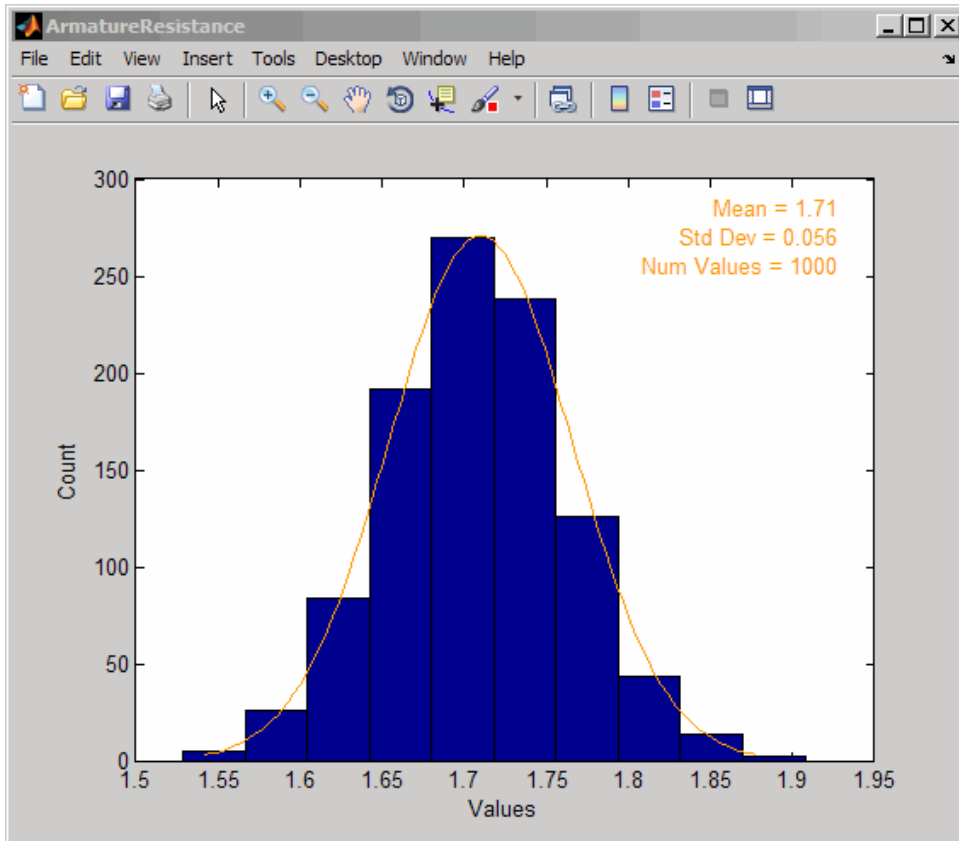
For more information on viewing the data, see "View Data While Configuring the Test Vector" on page 2-22.

**9** Click **OK** in the Insert Test Vector dialog box.

The new vector appears in the **Test Vectors** pane.

Properties | **Test Vectors** ↗ ✕ Test Variables

New... | ✕ | Evaluate

| Name | Length | Group Name | Type |
|------|--------|------------|------|
| ArmatureResistance | 1000 | Group1 | Probability Distribution |

▲▼

General | Grouping

Name: | ArmatureResistance

Type: | Probability Distribution

Distribution: | Normal (Gaussian)

Mean: | 1.71

Standard deviation: | .056

Number of values: | 1000

Test Vector successfully evaluated to a 1x1000 double

☐ Evaluate Test Vector each time the test is run
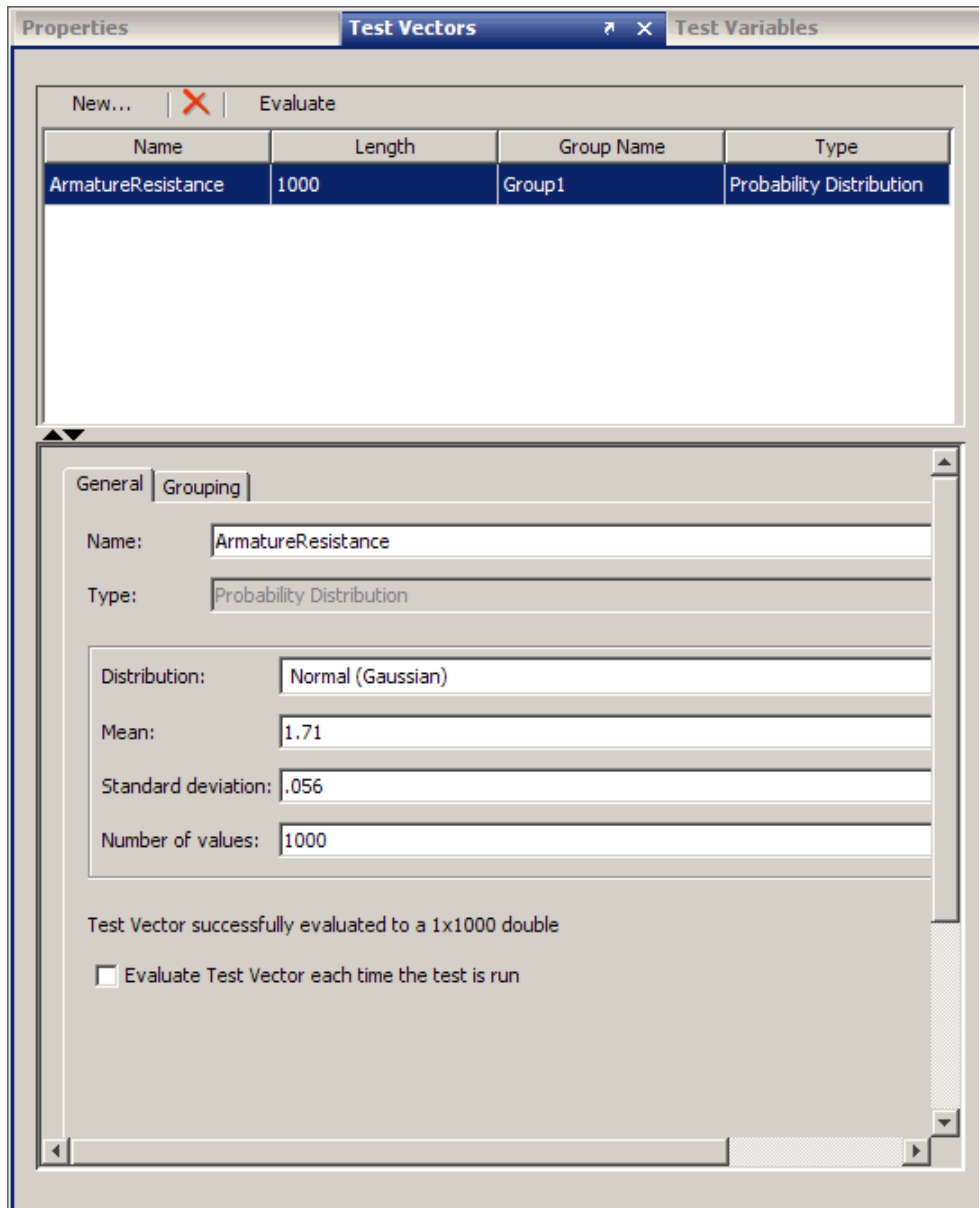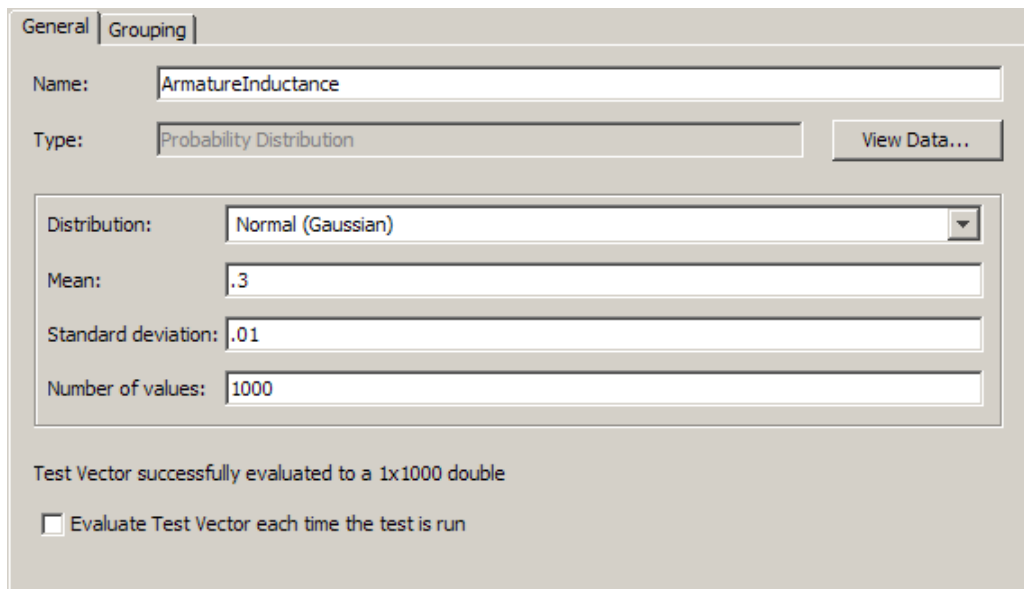
To create the second vector, for armature inductance:

**1**    In the **Test Vectors** pane, click the **New** button.

**2**    In the Insert Test Vector dialog box, select **Probability Distribution** as the test vector type.

**3**    Enter `ArmatureInductance` in the **Name** field.

**4**    Use the default distribution, normal (Gaussian).

**5**    In the **Mean** field, enter `.3`.

**6**    In the **Standard deviation** field, enter `.01`.

**7**    In the **Number of values** field, enter `1000`.

---

General | Grouping

Name:                ArmatureInductance

Type:                Probability Distribution                                     View Data...

Distribution:        Normal (Gaussian)

Mean:                .3

Standard deviation:  .01

Number of values:    1000

Test Vector successfully evaluated to a 1x1000 double

☐ Evaluate Test Vector each time the test is run

---

For this vector, the test is varying armature inductance up to a standard deviation of .01, around a mean of .3, and using 1000 values.
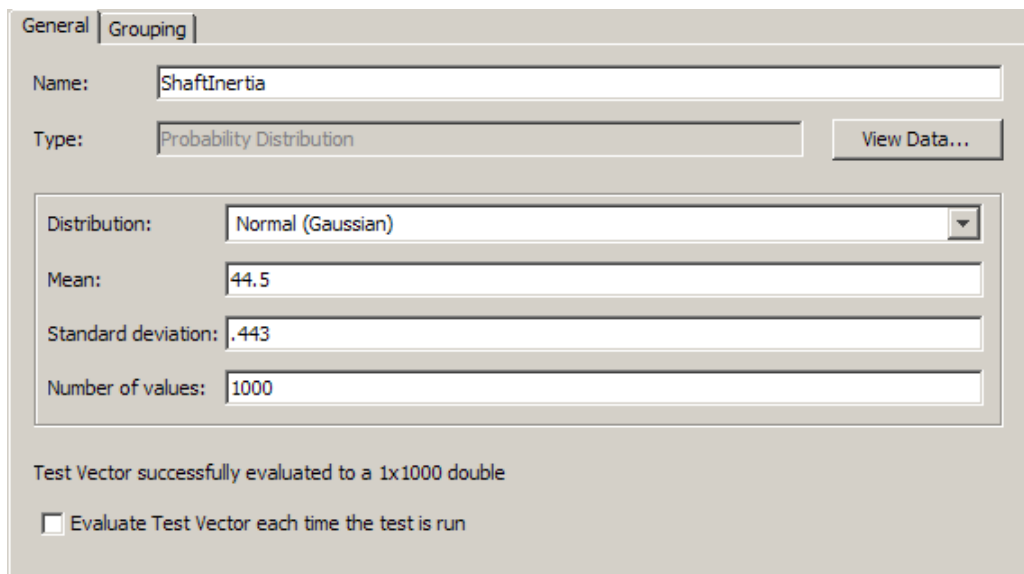
**8**    You can optionally click the **View Data** button to see a visualization of the test vector data that you configured.

**9**    Click **OK** in the Insert Test Vector dialog box.

The new vector appears in the **Test Vectors** pane.

| Properties | Test Vectors | ↗ × | Test Variables |
|---|---|---|---|

New... | ✕ | Evaluate

| Name | Length | Group Name | Type |
|---|---|---|---|
| ArmatureResistance | 1000 | Group1 | Probability Distribution |
| ArmatureInductance | 1000 | Group1 | Probability Distribution |

▲▼

**General** | Grouping

Name: ArmatureInductance

Type: Probability Distribution

Distribution: Normal (Gaussian)

Mean: .3

Standard deviation: .01

Number of values: 1000

Test Vector successfully evaluated to a 1x1000 double

☐ Evaluate Test Vector each time the test is run
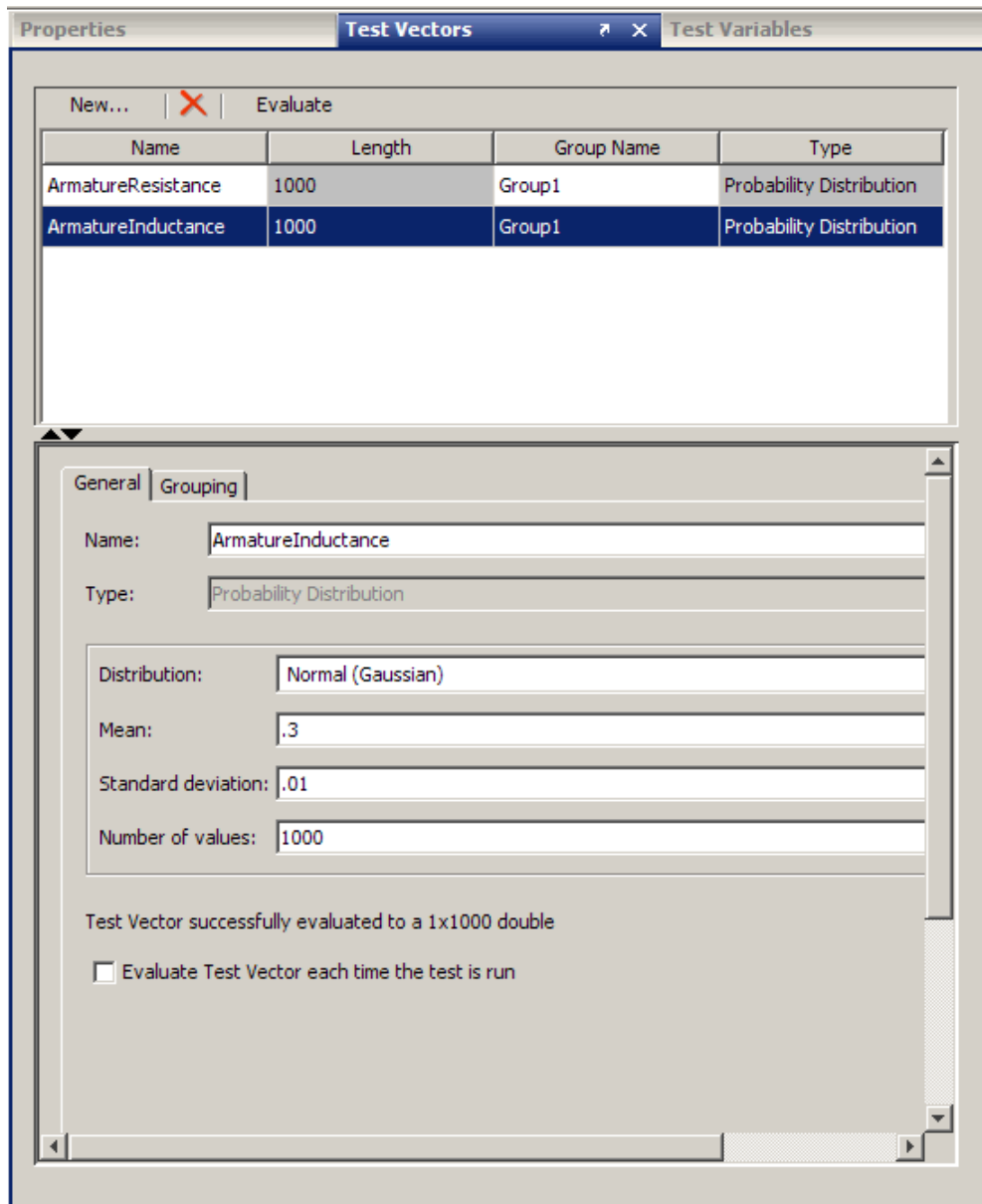
To create the third vector, for shaft inertia:

**1** In the **Test Vectors** pane, click the **New** button.

**2** In the Insert Test Vector dialog box, select **Probability Distribution** as the test vector type.

**3** Enter `ShaftInertia` in the **Name** field.

**4** Use the default distribution, normal (Gaussian).

**5** In the **Mean** field, enter `44.5`.

**6** In the **Standard deviation** field, enter `.443`.

**7** In the **Number of values** field, enter `1000`.

For this vector, the test is varying shaft inertia up to a standard deviation of .443, around a mean of 44.5, and using 1000 values.

**8** You can optionally click the **View Data** button to see a visualization of the test vector data that you configured.

**9** Click **OK** in the Insert Test Vector dialog box.

The new vector appears in the **Test Vectors** pane.

| | | | |
|---|---|---|---|
| **Properties** | | **Test Vectors** ↗ × | Test Variables |

New...  ✕  Evaluate

| Name | Length | Group Name | Type |
|---|---|---|---|
| ArmatureResistance | 1000 | Group1 | Probability Distribution |
| ArmatureInductance | 1000 | Group1 | Probability Distribution |
| ShaftInertia | 1000 | Group1 | Probability Distribution |

General | Grouping

Name:              ShaftInertia

Type:              Probability Distribution

Distribution:          Normal (Gaussian)

Mean:               44.5

Standard deviation:  .443

Number of values:    1000

Test Vector successfully evaluated to a 1x1000 double

☐ Evaluate Test Vector each time the test is run

# Spreadsheet Data Test Vectors

| **In this section...** |
| --- |
| "Introduction" on page 2-40 |
| "Create a Spreadsheet Data Test Vector" on page 2-40 |
| "Configure the Spreadsheet Data Test Vector" on page 2-44 |
| "Replace Strings" on page 2-47 |

## Introduction

The Spreadsheet Data test vector type can be used to read data from Microsoft® Excel® files or `.csv` files into the SystemTest software. This feature also supports file formats used by the MATLAB `xlsread` function.

You can read spreadsheet data from multiple sheets, and can read whole sheets or a subset of a sheet.

For a detailed example using the Spreadsheet Data test vector, see "Override Simulink Inport Blocks Using a Spreadsheet Data Test Vector" on page 4-27.

---

**Note:** For additional technical information and limitations of this feature, see the SystemTest Release Notes.

---

## Create a Spreadsheet Data Test Vector

To create a Spreadsheet Data test vector:

1 In the **Test Vectors** pane, click the **New** button.

2 In the Insert Test Vector dialog box, select **Spreadsheet Data** as the test vector type.
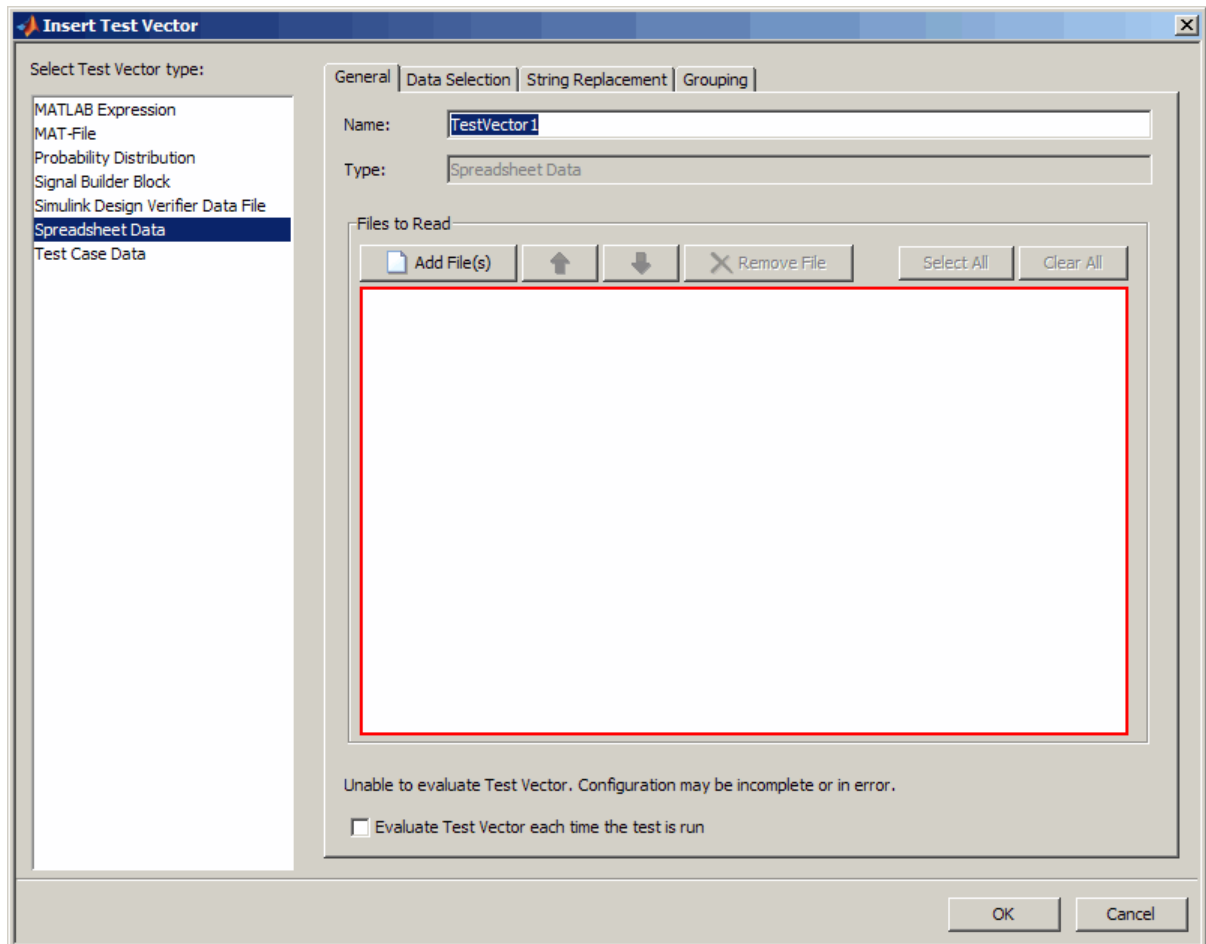
**3** On the **General** tab, click the **Add File** button.

Browse to your Microsoft Excel spreadsheet file or a `.csv` file and click **Open**.

**4** The first sheet of your file is selected by default. If the file has multiple sheets and you want to use them, select the other sheet(s). There is no limit to the number of sheets you can use.

**5** Select the **Evaluate Test Vector each time the test is run** option if you want to read the file every time the test is run. Leave it unselected if you want to use the same values each time the test is run.

In the case of a Spreadsheet Data test vector, using this option means that data would be read from the spreadsheet file every time the test is run. If you expect the data to change and want to have it read every time, select this option. If you know the data is static or you do not want it to be read each time, unselect the option.

Note that you can use the **Evaluate** button in the **Test Vectors** pane any time for an immediate evaluation.

**6** On the **Data Selection** tab, choose the range to use in the test vector. Enter this information in the **Data Range** section to select the range.

Specify whether your data is arranged by column or row using the **Data is arranged by** option.

Then select the specific range using the **Read data from** option. For example, if you have a file that has data in columns A, B, and C, and there is data in rows 3 through 13 and you want to read all the data, in the **Read data from column** option, fill in A to C. Then in the **starting at row** option, enter 3. The SystemTest software will read to the end of the data.

All data in the designated columns is read, from the start-at row through the end of data. Therefore you should only put data in the columns that you want to be read. Extraneous data should be removed if you do not want it to be read. Any blank cells within the read data range will be treated as NaN.

If the first row of your sheet is a header, you can select the **First row is a header** option to have the SystemTest software exclude it from the data.

**7** In the **For Each Selected Sheet** section, select the option to determine how the data is arranged when the vector is created. You can have each row (or column) of the spreadsheet be a separate test vector value, or you can have the entire sheet be one test vector value.
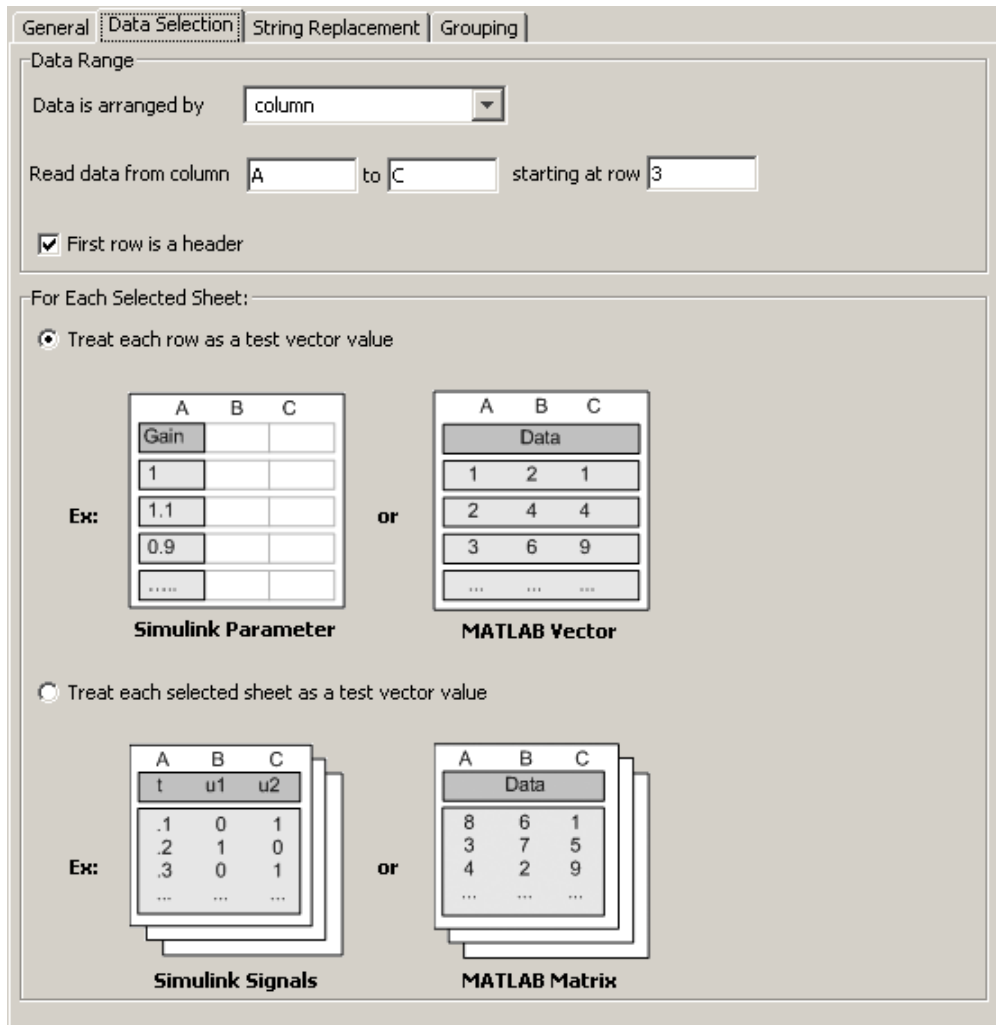
See "Configure the Spreadsheet Data Test Vector" on page 2-44 for more information about these two options.

**8** You can optionally replace strings in the file with values using the **String Replacement** tab. The table is automatically populated with any strings contained in your sheet(s). If you want to replace each occurrence of a particular string with a

value, type the value in the **Value** column of the table. Then when the test vector is evaluated, that string will be replaced with the value you indicated to populate the test vector.

See "Replace Strings" on page 2-47 for more information about this option.

**9** Click **OK** in the Insert Test Vector dialog box. The new vector then appears in the **Test Vectors** pane.

After creating a Spreadsheet Data test vector, you can edit it any time by selecting it in the table in the **Test Vectors** tab. If you make any changes to the configuration of the test vector in the SystemTest software, they will be applied immediately. If you make any changes to the underlying spreadsheet, you can have the data reread by clicking the **Evaluate** button above the test vectors table.

For a detailed example using the Spreadsheet Data test vector, see "Override Simulink Inport Blocks Using a Spreadsheet Data Test Vector" on page 4-27.

---

**Note:** If the data in your spreadsheet is numeric, it will be a double array in the test vector. If the data contains any strings, it will be a cell array. If the data contains header information and you specified the first row as a header, that will be excluded, and if the remaining data is numeric, it's treated as a double array.

---

## Configure the Spreadsheet Data Test Vector

As shown in step 7 in "Create a Spreadsheet Data Test Vector" on page 2-40, you can configure test vector values using the **Data Selection** tab when you create or edit a Spreadsheet Data test vector.

In the **For Each Selected Sheet** section, you select the option to determine how the vector is created. You can have each row (or column) of the spreadsheet be a separate test vector value, or you can have the entire sheet be one test vector value.

**Treat each row as a test vector value**

The **Treat each row as a test vector value** option means that each row or column (depending on what you selected in the **Data is arranged by** option) is one test vector value.



In the first case shown here, column A contains values for the parameter `Gain`. Suppose this column contains 10 values, in rows 2 through 11 (row 1 is a header). The resulting test vector would be a 1-by-10 array containing 10 values. The first value is `1`, the second value is `1.1`, etc. The ten populated rows result in a total of ten values, each row being one scalar value.

The same is true of the second example shown — that each row is a separate value, except that in this case each value is an array, instead of a single scalar. The first test vector value in this case is the array `[1 2 1]`. The second test vector value is `[2 4 4]`, etc. If this sheet also had ten rows, there would be ten separate values (each an array of 3 numbers) and the test vector length would be 10.

**Treat each selected sheet as a test vector value**

The **Treat each selected sheet as a test vector value** option means that each entire sheet is one test vector value.
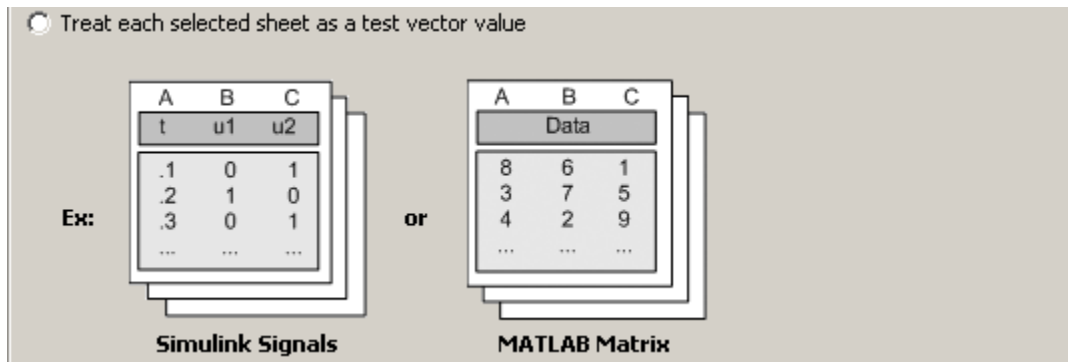


If the sheet contains multiple rows and columns, the resulting test vector value is a matrix. In the first example shown here, labeled Simulink Signals, this spreadsheet file contains 3 sheets. Suppose each sheet contained the three columns shown, t, u1, and u2, and had just the three rows of values shown. The resulting test vector would be of length 3 since each sheet is one test vector value and there are three sheets, and each of the three test vector values would be a 3-by-3 matrix.

Suppose the second example, labeled MATLAB Matrix, contained five sheets and each sheet had the three columns shown, each with ten rows of data. The resulting test vector would be of length 5 since each sheet is one test vector value and there are five sheets, and the five test vector values would each be a 10-by-3 matrix, since the sheets have ten rows of data and three columns.

Configuring each sheet to be one test vector value can be useful in a case where you have a test case in each sheet, and each test case is a matrix.

**Using Multiple Sheets**

If you configure a test vector to use multiple sheets in a file, and you use the **Treat each row as a test vector value** option, each sheet is read, turned into individual rows, and then appended together. For example, if your file has three sheets containing three, four, and five rows of data respectively, the resulting test vector is a set of row vectors as follows:

```
row 1 from sheet 1
row 2 from sheet 1
row 3 from sheet 1
row 1 from sheet 2
row 2 from sheet 2
row 3 from sheet 2
row 4 from sheet 2
row 1 from sheet 3
row 2 from sheet 3
row 3 from sheet 3
row 4 from sheet 3
row 5 from sheet 3
```

If you configure a test vector to use multiple sheets in a file, and you use the **Treat each selected sheet as a test vector value** option, the resulting test vector will have the same number of values as there are sheets in the file. The same file with three sheets would have three values:

```
sheet 1
sheet 2
sheet 3
```

## Replace Strings

As shown in step 8 in "Create a Spreadsheet Data Test Vector" on page 2-40, you can optionally replace strings in the data you read from your spreadsheet files with values using the **String Replacement** tab when you create or edit a Spreadsheet Data test vector. The table lists any strings contained in your sheet(s), excluding headers if you've specified they are present.

If you want to replace each occurrence of a particular string with a value, type the value in the **Value** column of the table. Then when the test is run, that string will be replaced with the value you indicated to create the test vector.

An example use case for this feature is that you could have a spreadsheet that contains values for switches, and the values are designated by the strings ON and OFF.

| | A | B | C |
|---|---|---|---|
| 1 | Switch A | Switch B | Switch C |
| 2 | OFF | ON | OFF |
| 3 | OFF | ON | OFF |
| 4 | ON | OFF | OFF |
| 5 | | | |

In this example, you might want to replace each instance of ON with a 1 and each instance of OFF with a 0. The **String Replacement** tab of the Insert Test Vector dialog box would look like the following:
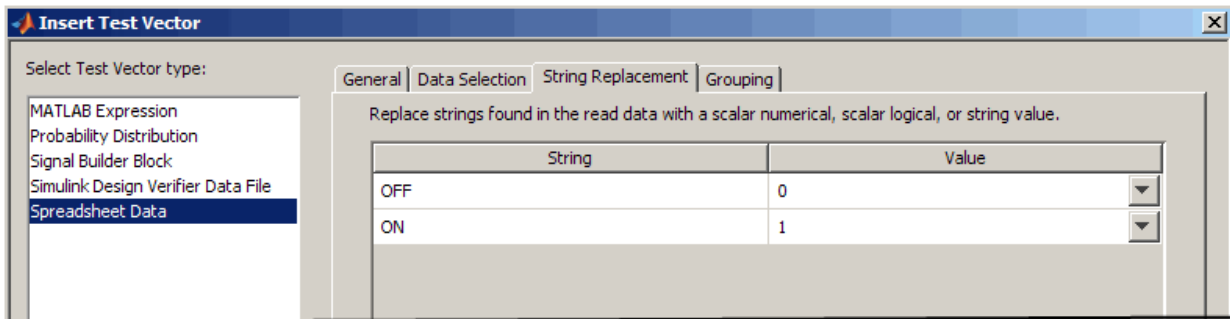


If you want to map the same strings to different values, you have to create separate test vectors and do each replacement mapping separately. For example, in the previous case, you might want the values for Switch A to map to 1 and 0 as shown, but for Switch B you might want to use 100 and 0. In this case, create a test vector that reads only column A and replace ON and OFF with 1 and 0, and then create a second test vector for column B that maps Switch B values to 100 and 0.

# Simulink Design Verifier Data File Test Vectors

| In this section... |
| --- |
| "Prerequisites" on page 2-49 |
| "Create SystemTest Harness from Simulink Design Verifier" on page 2-49 |
| "Create a Simulink Design Verifier Test Vector" on page 2-51 |
| "Important Usage Notes" on page 2-61 |

## Prerequisites

The Simulink Design Verifier Data File test vector can read test cases created by the Simulink Design Verifier™ software. In order to use this test vector, you need a Simulink Design Verifier data file with test cases.

To use this feature, you first run Simulink Design Verifier with the appropriate configuration. Then you can do one of two things:

- Generate a SystemTest harness for the model from Simulink. When it completes, a new test opens automatically in SystemTest and a Simulink Design Verifier Data File test vector is automatically created. This workflow is described in "Create SystemTest Harness from Simulink Design Verifier" on page 2-49.

- If you already have a data file from Simulink Design Verifier, you can create a test vector in SystemTest that uses the test cases in the data file, and configure overrides in a Simulink element. This workflow is described in "Create a Simulink Design Verifier Test Vector" on page 2-51.

## Create SystemTest Harness from Simulink Design Verifier

If you generate a SystemTest test harness from Simulink using Simulink Design Verifier, a new test opens automatically in SystemTest with a Simulink Design Verifier Data File test vector and a Simulink element automatically created for you. The following steps outline this workflow.

1   From your model, select **Analysis > Design Verifier > Options**.

2   In the Configuration Parameters dialog box, select **Design Verifier > Results**, and then enable the **Save test harness as SystemTest TEST-file** option.
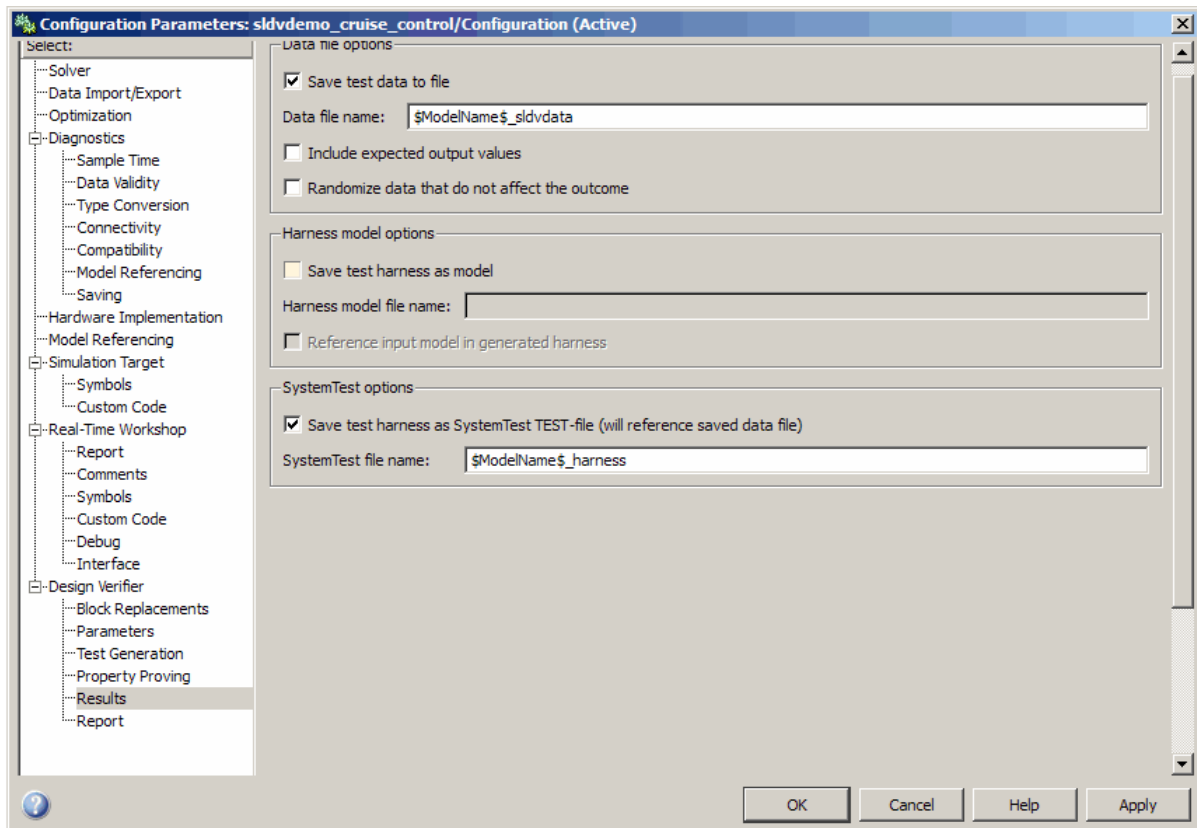
3  Click **OK**.

4  In Simulink, save the model.

5  From your model, select **Analysis > Design Verifier > Generate Tests** to run the model and generate the SystemTest test harness.

   After the model generates test cases, the SystemTest software opens automatically. A Simulink Design Verifier Data File test vector containing the generated test inputs is automatically created. A Simulink element is also created, configured with the model name, override mappings set, and model coverage enabled.

6  Optionally, in the SystemTest software, you can add other things to the test, such as a plot element. For an example of this, see "Create a Simulink Design Verifier Test Vector" on page 2-51.

**7** Run the test in the SystemTest software by clicking the **Run** button.

## Create a Simulink Design Verifier Test Vector

If you already have a data file from Simulink Design Verifier, you can create a test vector in the SystemTest software that uses the generated rest cases in the data file, and configure overrides in a Simulink element. The following steps outline this workflow.

**1** In the **Test Vectors** pane, click the **New** button.

**2** In the Insert Test Vector dialog box, select **Simulink Design Verifier Data File** as the test vector type.
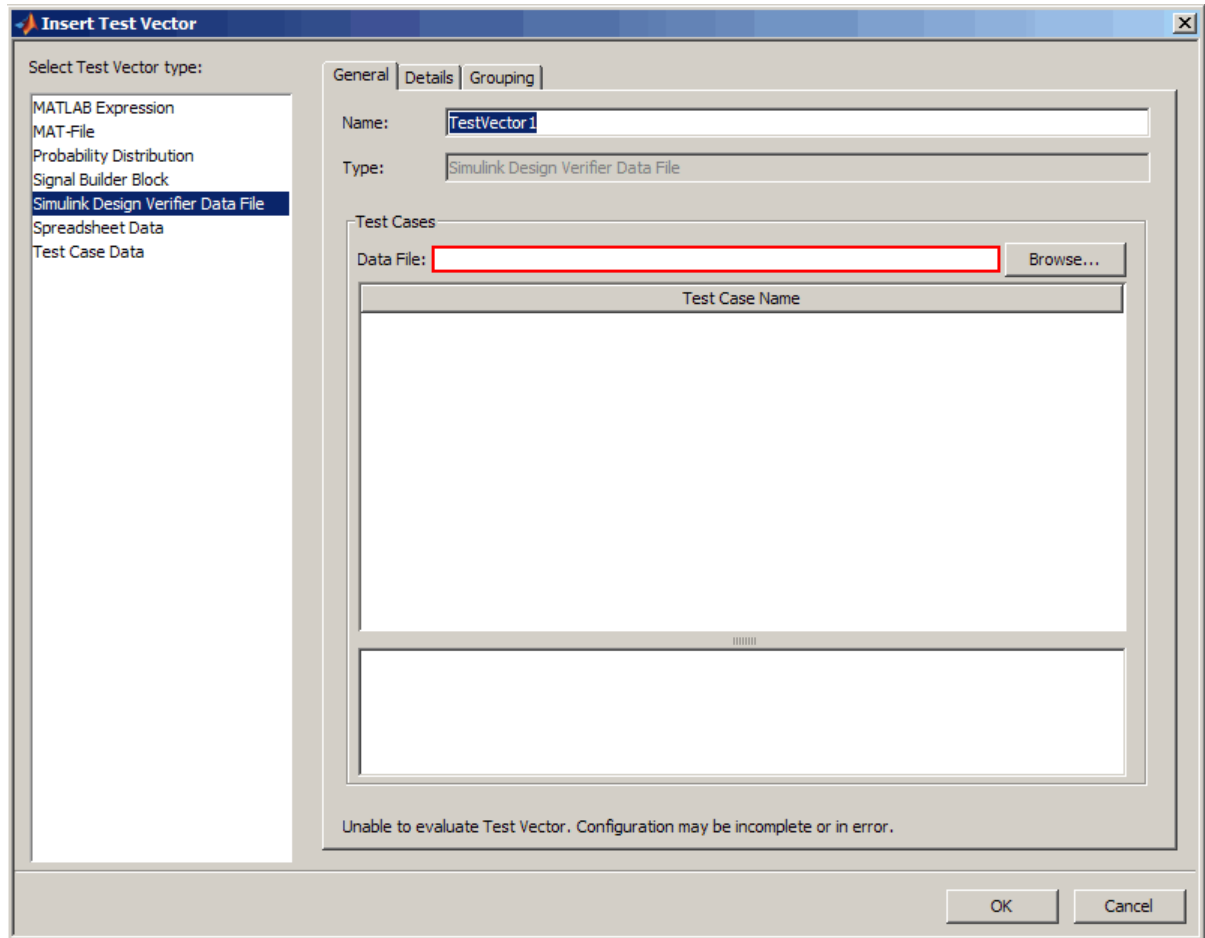
**3** Accept the default test vector name, or type a new one in the **Name** field.

**4** Type the name of the Simulink Design Verifier data file in the **Type** field, or use the **Browse** button to locate it. It will be a `.mat` file.

Note that you must use a valid MAT-file – a Simulink Design Verifier data file created in version R2008b or later. If you try to use a data file created in an earlier version of the software or a MAT-file that is not generated from Simulink Design Verifier, you will get an error.

**5** When the data file is read in, the test cases appear in the **Test Cases Name** table. Click any test case to see its test case description below the table.
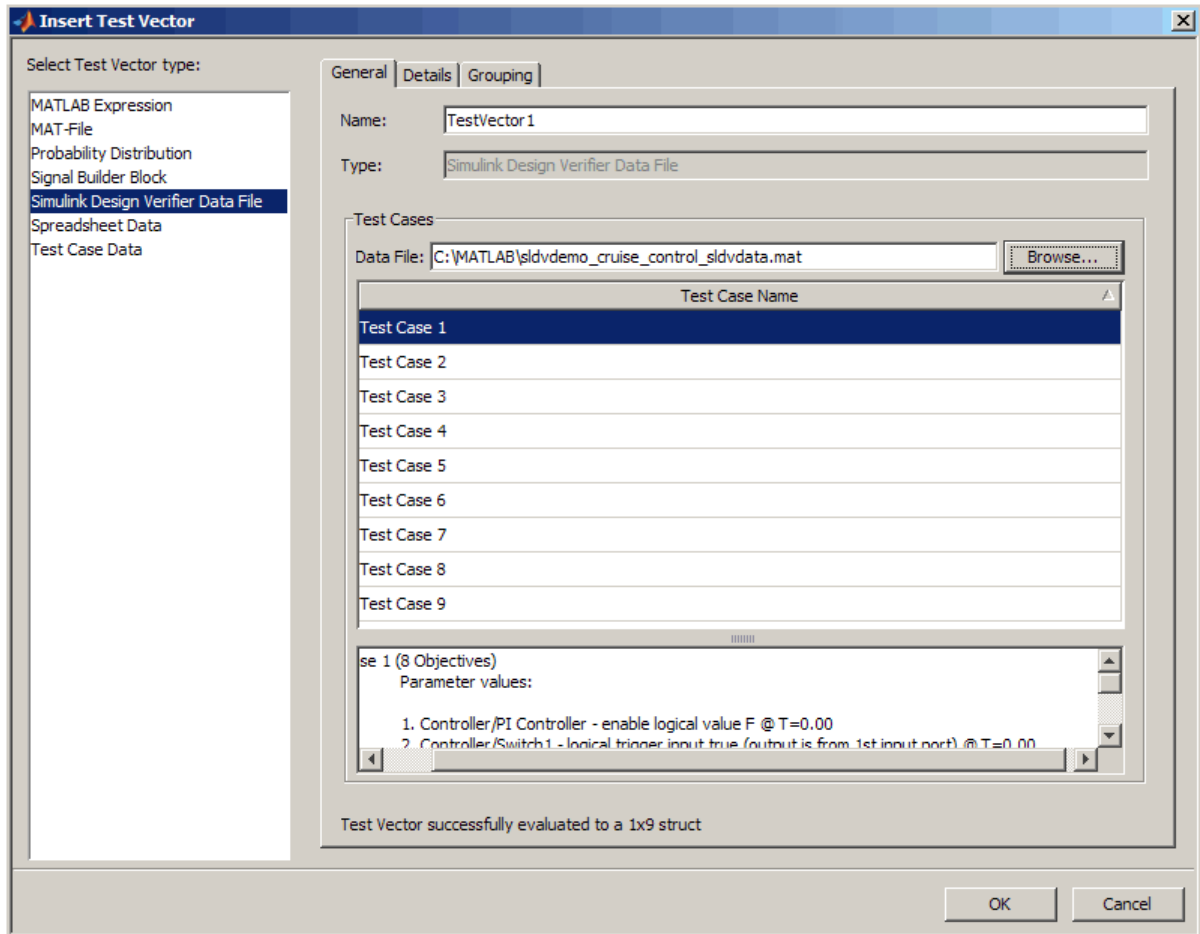


**6** To see information from the Simulink Design Verifier data file, click the **Details** tab. This provides analysis information on the data file, and the model Inport blocks associated with the test cases. If the test cases involve any model parameter configurations, they appear in the **Parameters** section. This section will list any parameters that are used as part of a test case. The information in this tab is not editable.
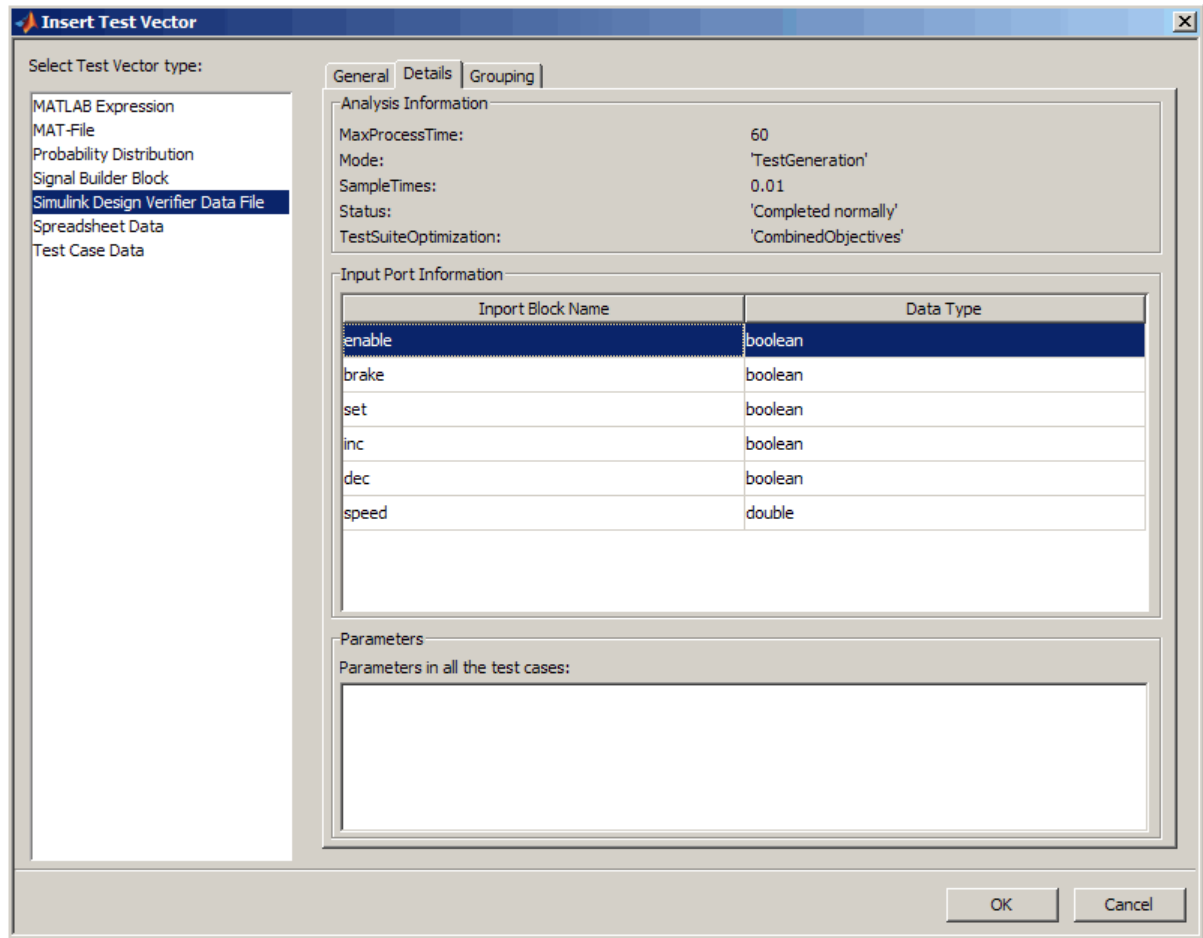
**7** Click the **OK** button to finish creating the new test vector. It then appears in the **Test Vectors** pane in the SystemTest desktop.

Now that the test vector is created, you can create mappings in a Simulink element.

**8** Create a Simulink element by clicking the **Main Test** node in the **Test Browser**, and clicking the **New** button. Select **Test Element > Simulink**.

**9** Type the name of the model, or use the **Browse** button to locate it. This should be the same model that was used to create the Simulink Design Verifier data file.

If you browsed for the file, when you click **OK**, the model opens.

**10** In the **Override Inport Block Signals with SystemTest Data** section of the Simulink element, select the **All Inport blocks are mapped using** option. You must select this option in order to correctly use the Simulink Design Verifier data file.

**11** From the drop-down list, select the test vector you created earlier in this workflow.
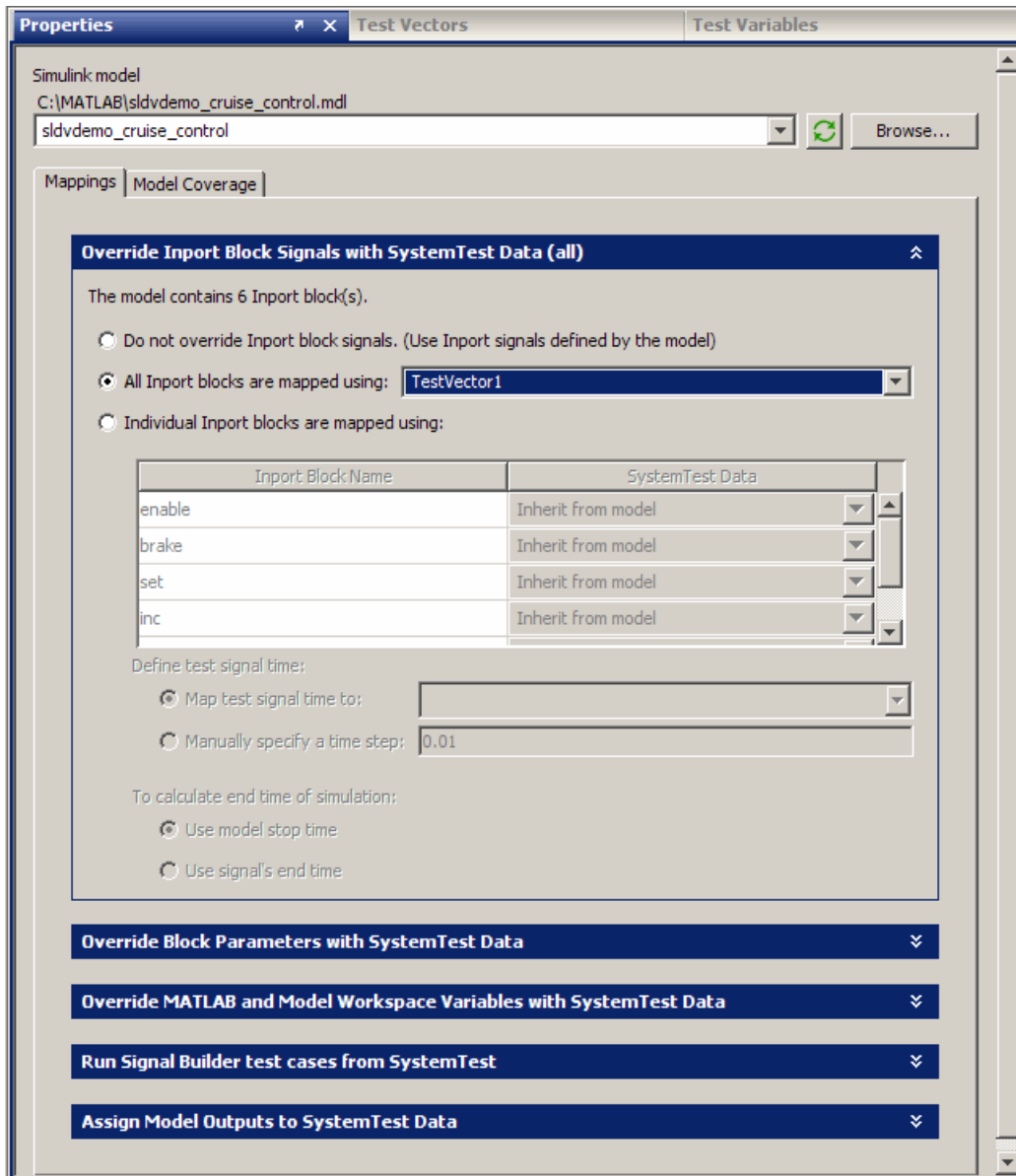
In the example shown here, the model name is `sldvdemo_cruise_control.slx` and the vector is `TestVector1`.

12 If you have the Simulink Verification and Validation™ software and you want to use the Model Coverage feature in the Simulink element, click the **Model Coverage** tab.

13 Select the **Enable Model Coverage** check box.

14 Select **Override model coverage metric settings**.

15 Select any metrics you want to cover in the **Coverage Metrics** section.
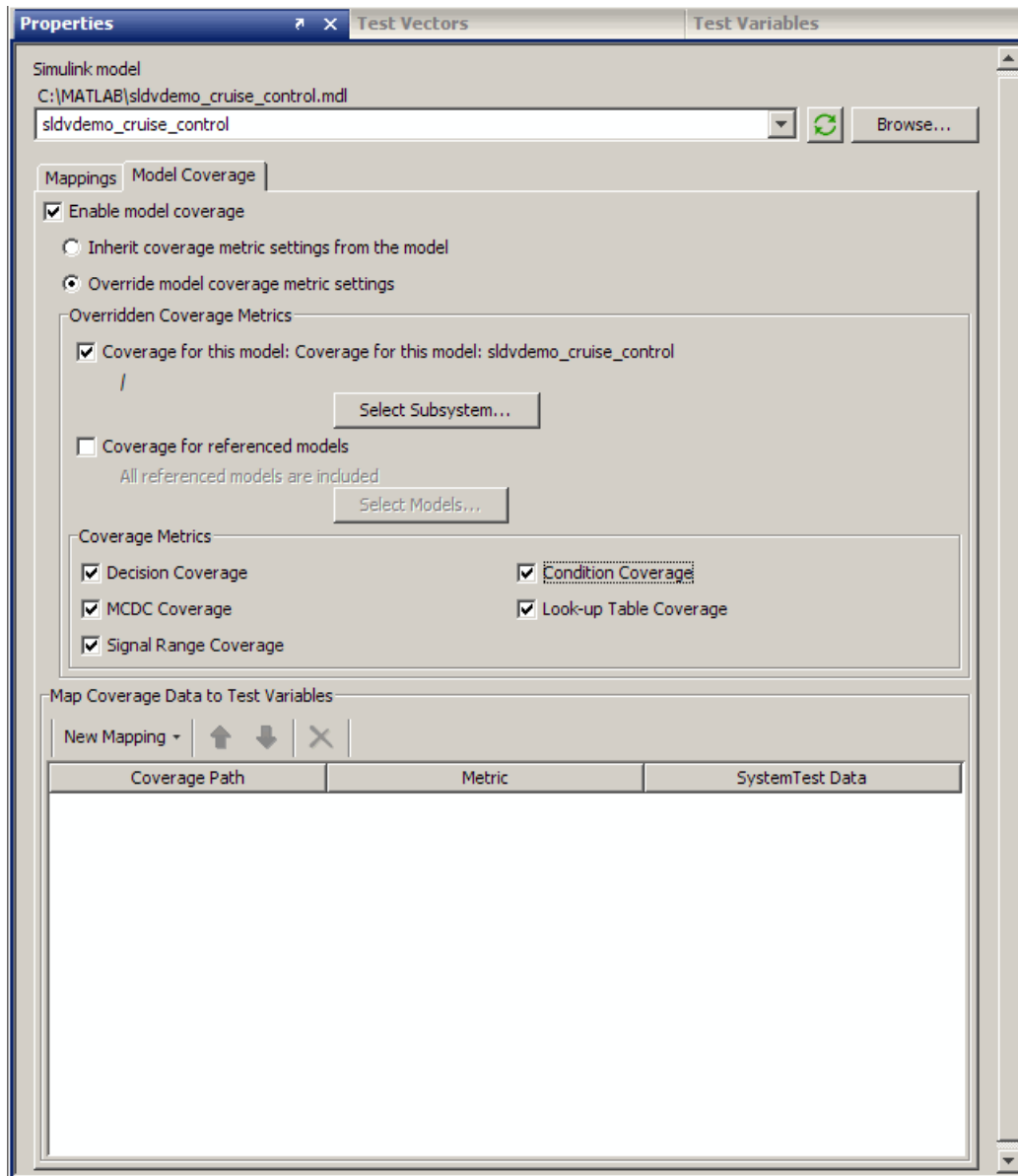
**16** Optionally, if you want to plot any of the signals, create a plot element.

Select the Simulink element you already created in the **Test Browser**, and select **New > Test Element > Plot – General**.

**17** In the Plot element, click the **Add Plot** button.

**18** Select **Simulink Data**.

**19** From the **Simulink Data** field, expand the test vector that you created to see the individual signals.
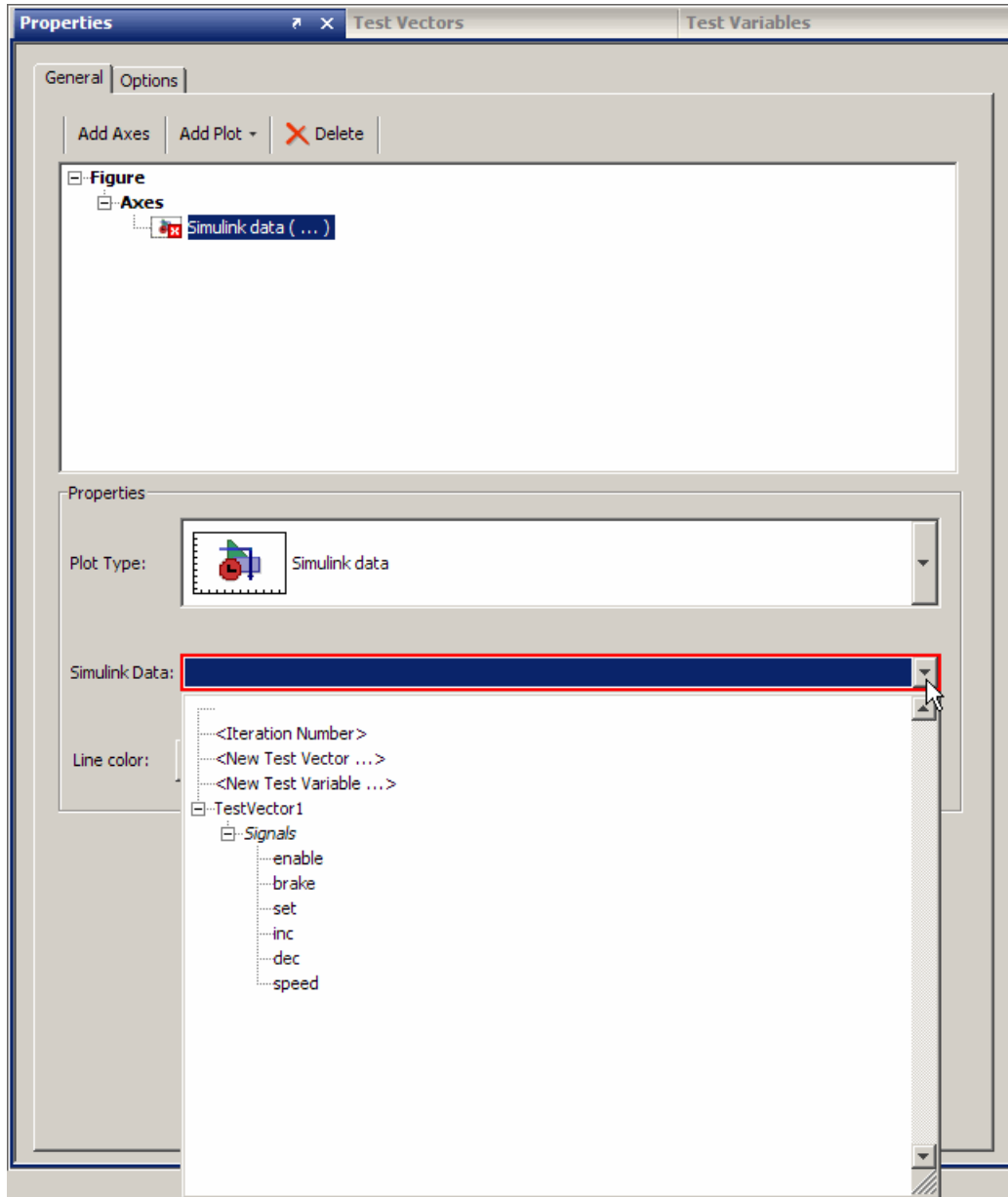
**20** Select one of the signals, for example, `speed`.

**21** Run the test by clicking the **Run** button on the SystemTest toolbar.

In this example, after the test runs, a model coverage report and a plot of the speed signal are generated.

## Important Usage Notes

The following notes pertain to the integration between the SystemTest software and Simulink Design Verifier using the Simulink Design Verifier Data File test vector:

- **Model Coverage Report** — The model coverage report generated by the model harness using Simulink Verification and Validation and that of the SystemTest harness generated by Simulink Design Verifier will be identical.

- **Data Format** — The format of the data from a Simulink Design Verifier Data File test vector, if seen in a MATLAB element or in saved test results for example, is a subset of the Simulink Design Verifier data format.

  It is a MATLAB structure with one field, `TestCases`. Then the `TestCases` field contains two fields, `dataValues` and `paramValues`. `TestCases` is a `1x1` structure. The following figure shows the data format for a Simulink Design Verifier Data File test vector called `TestVector1`:

```
K>> TestVector1

TestVector1 =

    TestCases: [1x1 struct]

K>> TestVector1.TestCases

ans =

     dataValues: {6x1 cell}
    paramValues: []
```

- **Data file Version** — To use the Simulink Design Verifier Data File test vector, you must use a Simulink Design Verifier data file created in version R2008b or later. If

you try to use a data file created in an earlier version of the software or a MAT-file that is not generated from Simulink Design Verifier, you will get an error.

- **Evaluating the Test Vector** — If you make changes in the underlying Simulink Design Verifier test cases, you can click the **Evaluate** button in the **Test Vectors** pane any time to see the changes reflected in the SystemTest user interface. However this is not necessary to pick up the changes for running the test. When you run a test containing a Simulink Design Verifier Data File test vector, the SystemTest software automatically queries the data file for the latest information in the test cases.

- **Changing the Underlying Model** — If you make changes in the underlying Simulink model, such as changes to Inport blocks, you should return to Simulink Design Verifier and regenerate the test cases and the test harness. Then return to SystemTest test harness to continue working with your test.

- **Model End Time** — In the use case where you automatically generate the SystemTest test harness from Simulink Design Verifier, the end time used will be that of the test cases per iteration. However, in the use case where you create the test vector in SystemTest using a Simulink Design Verifier data file that you already have, the underlying model's end time will be used per iteration.

- **Bus Support** — The Simulink Design Verifier Data File test vector supports the use of busses in Inport blocks. Bus support is only available in SystemTest through this feature.

# Create Signal Builder Block Test Vectors

If you have created a Simulink model test harness using a Signal Builder block, you can automate the running of all your test cases by integrating them into a SystemTest test. This also offers the ability to collect cumulative model coverage metrics for all your Signal Builder test cases.

The most common workflow for this feature is to create a Simulink element and then create the test vector from within the element, as follows:

1  In the SystemTest desktop, create a Simulink element by clicking the **Main Test** node in the **Test Browser**, and clicking the **New** button. Select **Test Element > Simulink**.

2  Type the name of the model, or use the **Browse** button to locate it. This should be the model that includes the Signal Builder block whose test cases you are interested in.

   When you click **OK**, the model opens.

   This example uses the model `systemtestsfcar`.

3  In the Simulink element, click the up arrows in the banner of the **Override Inport Block Signals with SystemTest Data** section to close it.

4  Click the down arrows in the banner of the **Run Signal Builder test cases from SystemTest** section to expand it.

5  Enable the Signal Builder test cases by selecting the **Use test vector** check box.

6  Click the down arrow and select **<New Signal Builder Block test vector...>**.
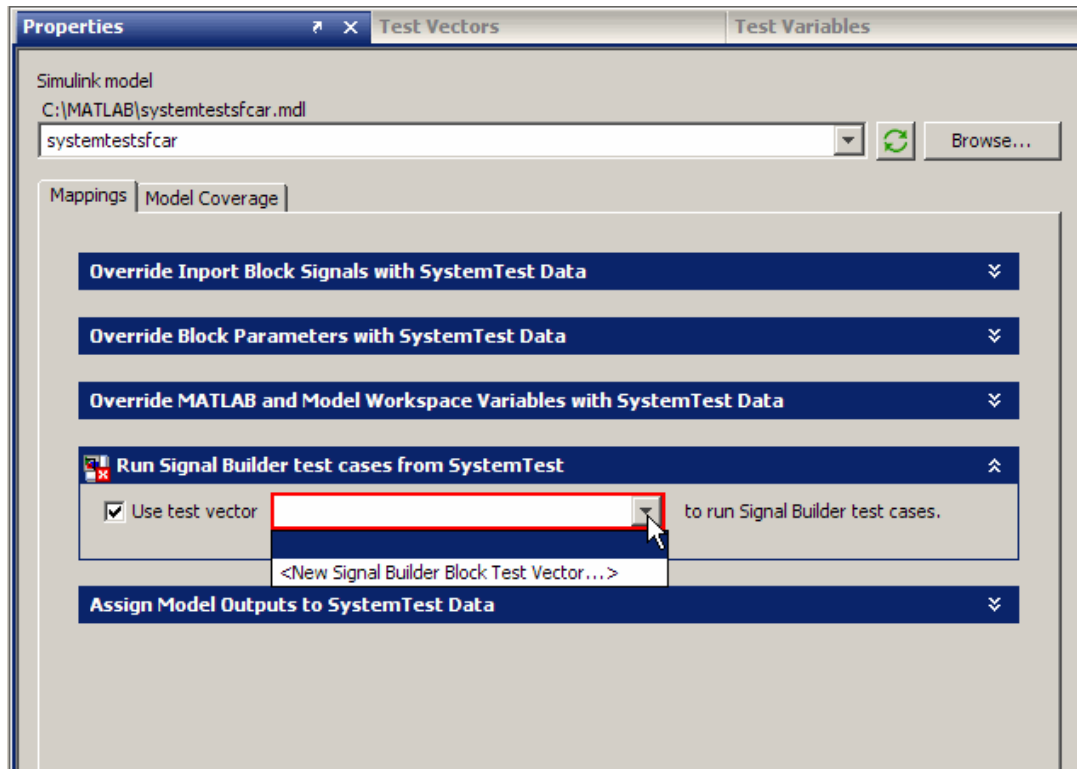
7    The Insert Test Vector dialog box opens and **Signal Builder Block** is the selected test vector type.

Keep the default test vector name or type a new one.

8    On the **General** tab, type the name of the model you used in the Simulink element, or click the **Browse** button to locate it.

> **Note:**  You cannot use a Signal Builder Block test vector with a Simulink element that uses a different model. You must refer to the same model in both the test vector and the Simulink element.

9    When the model is found, the Signal Builder test cases appear in the **Test Cases** section.
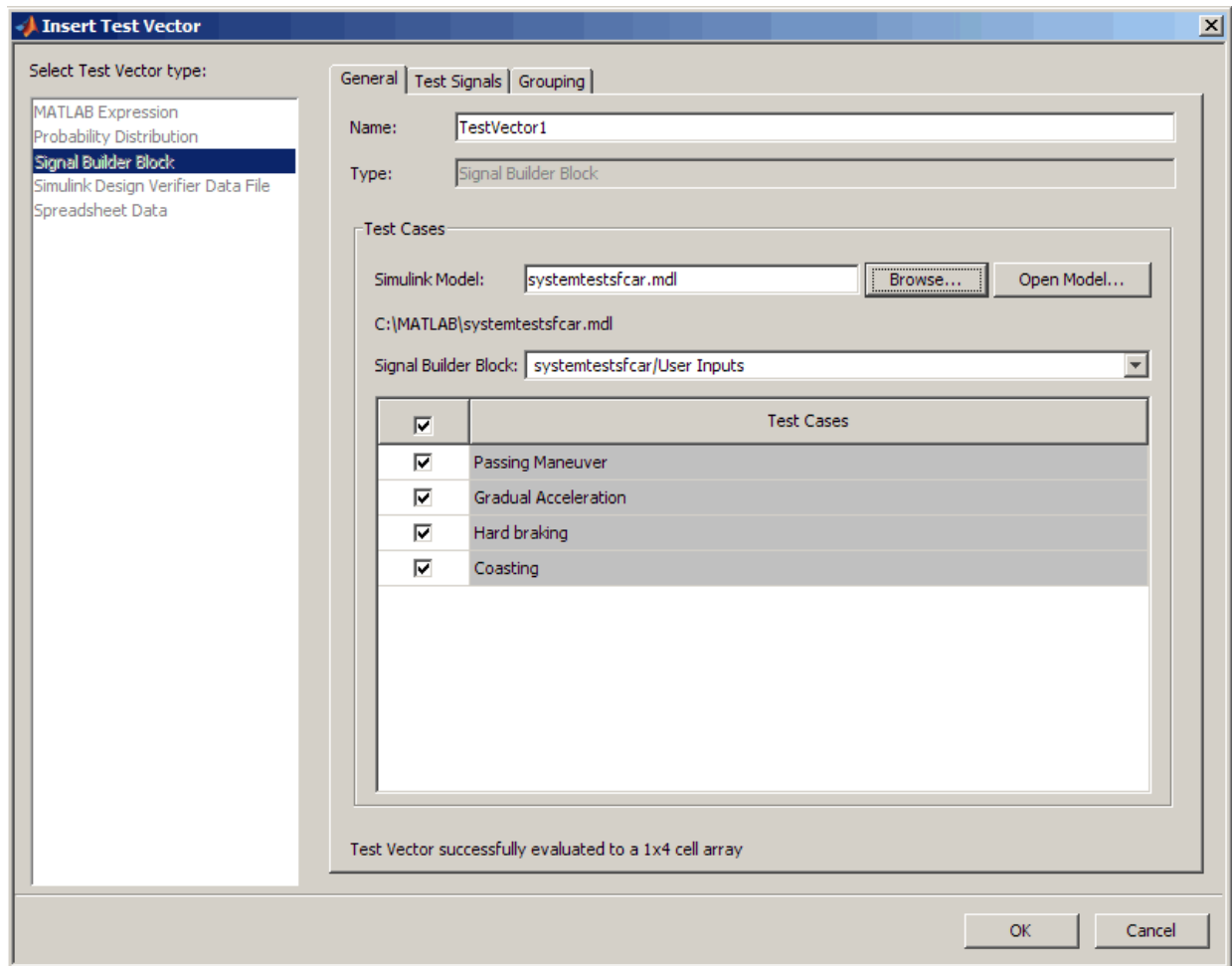
If there are any test cases you do not want to test, you can disable them using the check boxes. Test cases that are checked will be tested.

10 You can click the **Test Signals** tab to view the test signals associated with your Signal Builder block.

11 Click **OK** to finish creating the test vector.

12 To view or edit the test vector after it is created, click the **Test Vectors** tab in the SystemTest desktop.

**13** Optionally create other elements, test vectors, variables, or saved results, and run your test.

---

**Note:** If you make changes in the underlying Signal Builder block in your model, you can click the **Evaluate** button in the **Test Vectors** pane any time to see the changes reflected in the user interface. However this is not necessary to pick up the changes for running the test. When you run a test containing a Signal Builder Block test vector, the SystemTest software automatically queries the model for the latest information in the Signal Builder block.

---

---

**Note:** When you run the test, the Signal Builder test cases are run in the order in which they appear in the Signal Builder block in your model. This same order is reflected in the **Test Vectors** pane in the SystemTest software, unless you change the order in the table by sorting the columns.

---

---

**Note:** You may have tested a Signal Builder block in previous SystemTest versions by using the **Override Block Parameters with SystemTest Data** section of a Simulink element. In that scenario you would create a new mapping to the Signal Builder block.

However, using the **Run Signal Builder test cases from SystemTest** section in the Simulink element and creating the Signal Builder Block test vector is a better and easier solution. Because the Signal Builder test cases are in a test vector, you can do more with them, such as plotting. Also, the signals are stored in the SystemTest results set, rather than the index of the test case.

Note that if you have a Simulink element that contains the mappings from the former way of including a Signal Builder block, and then you use the new Signal Builder Block test vector and use the new section in the same Simulink element, the test will use the new information in the **Run Signal Builder test cases from SystemTest** section in the Simulink element.

---

# Create a Test Case Data Test Vector

You can create signals in the SystemTest software and use them to test a Simulink model. The Test Case Editor provides a graphical way of creating, editing, and visualizing signal data in SystemTest.

The Test Case Editor is accessed through the Test Case Data test vector in the SystemTest software. For more information, see "Author and Use Signals in Tests".

To create a Test Case Data test vector:

**1**    On the **Test Vectors** pane of SystemTest software, click the **New** button.

**2**    In the Insert New Test Vector dialog box, select **Test Case Data** as the test vector type.



**3**    Assign a name to the vector in the **Name** field.

**4**    Click **OK** in the Insert Test Vector dialog box.

**5**    On the **Test Vectors** pane, select the test vector you just created, and click the **Open Test Case Editor** button to create the test cases and signals, as described in "Author and Use Signals in Tests" on page 5-4.

6  Alternatively, you can click the **Open Test Case Editor** button after step 3, while creating the test vector. If you do that, click the **OK** in the Insert Test Vector dialog box once you return to the SystemTest desktop.

Whether you create the test cases and signals during creation of the test vector, or after you have created it, see "Create Test Cases, Signals, and Buses" on page 5-9 for information on creating and editing the test cases and signals.

# Access Test Case Data Using MATLAB Element

You can access the data from a Test Case Data test vector by using a MATLAB element in a test that has a Test Case Data test vector. You could use the data for a variety of reasons, such as writing it to a CSV file, calling a custom function, or creating a plot.

If you have a Test Case Data test vector called `TestVector1` with a signal called `Signal1`, and use a custom analysis function in your test, the following are examples of code you could use in a MATLAB element to access the signal's data for your function.

Get a cell array of signal names stored in the Test Case Data test vector.

```
signalNames = getValue(TestVector1);
```

Get the timeseries object.

```
Signal1Timeseries = getValue(TestVector1, 'Signal1');
```

Get the Time and Data out.

```
time = Signal1Timeseries.Time;
data = Signal1Timeseries.Data;
```

Call your custom function.

```
myCustomAnalysisFunction(time, data)
```

# Edit Test Vector from within an Element

If you want to edit a test vector while working within an element, you can open the editor by right-clicking on the name of the test vector in the table(s) on the **Properties** tab of some of the elements. This feature is included in the following elements:

- Limit Check – General Check
- Limit Check – Tolerance Check
- Simulink
- General Plot

**3**

# Constructing a Test

# Test Sections

## Overview

Each section of the test serves a different purpose and has different properties that can be set in the **Properties** pane. Click a part of the test or an element in the **Test Browser** to see the properties for that section or element.

The descriptions of the elements in this chapter include a list of which sections of the test you can use each element in. The following sections describe the sections of a test. They are followed by a description of how to use the basic elements.

## Pre Test

The Pre Test runs once prior to any number of iterations through Main Test. Pre Test can be used to perform general test setup such as:

- Opening a model.
- Initializing variables.
- Accessing system resources, such as opening a file.
- Initializing external test equipment.

In Pre Test, only test variables defined as a Pre Test variable may be modified or assigned to. Pre Test variables are initialized during Pre Test and persist throughout the Main Test and Post Test.

In Pre Test you can add the following element types: Simulink, MATLAB, Subsection, Stop, IF, Video Input, the three Instrument Control Toolbox elements, and the four Data Acquisition Toolbox elements.

With Pre Test you can initialize Pre Test variables and run elements that you only want to run once before any Main Test iterations. For example, you can:

- Add a Simulink element to run a model and assign baseline data to a Pre Test variable.
- Add a MATLAB element to load a MAT-file or perform some other test setup.
- Create conditions with the IF element and follow up with a Subsection element to define what to do when those conditions are met.

## Main Test

The Main Test is run one or more times based on the number of iterations. It is used to:

- Execute elements multiple times in order to perform batch testing or sweep through a parameter space.
- Perform batch testing or parameter sweeps that require multiple independent iterations using different test conditions for each iteration.

The number of iterations is defined by the number and length of test vectors you specify. The SystemTest software executes Main Test once for each permutation of values in the test vectors specified.

In Main Test you can add all of the element types.

## Post Test

The Post test runs once after all Main Test iterations have executed or when a run-time error occurs in Pre Test or Main Test. Post Test can be used to perform test cleanup, such as:

- Closing a model.
- Cleaning up your workspace.
- Releasing system resources, such as closing a file.
- Returning external test equipment to a safe state.

In Post Test you can add the following element types: MATLAB, Subsection, IF, Video Input, the three Instrument Control Toolbox elements, and the four Data Acquisition Toolbox elements.

# Basic Elements

## Elements

Elements are the actions that a test performs. The SystemTest software includes the following set of elements, listed in alphabetical order.

- General Plot — Used to plot any type of data over multiple iterations.
- IF — Implements a logic control operator.
- Limit Check — Specifies the comparison to be performed of the value(s) under test and their expected value(s), or limit(s).
- MATLAB — Executes any MATLAB statements.
- Simulink — Runs a Simulink model. Note that you need to have a license for Simulink to use this element.
- Stop — Implements a logic control operator.
- Subsection — Creates a new section in a test that you can use to group elements within.

**Note:** Some MathWorks products, such as Image Acquisition Toolbox, Data Acquisition Toolbox, and Instrument Control Toolbox, provide their own elements that integrate those products' capabilities within the SystemTest software. If you have licenses for those products, those elements will also appear in the elements list.

For more information about using the basic elements, see "Construct a Test".

You add elements to a section in your test; however, not all elements can be added to all sections. For example, you can use a MATLAB element anywhere within a test, but you can only use the Limit Check element in the Main Test section.

---

**Tip** You can rename any element or subsection by double-clicking its name in the **Test Browser**.

---

**Invalid Characters in Element Names**

The following characters are invalid to include within element names:

- '
- <
- >

You cannot use these three characters in element names. If you create a new test element with one or more of these characters in the element name, then the SystemTest software throws an error dialog and the element name is reset to the default value, which is the name of the element type.

If you try to load an existing test with an invalid element name (containing one or more of the three characters listed above), the SystemTest software displays an error dialog indicating that the element name is invalid. The test will load successfully, but the element with an invalid name is reset to use the default name for the element. If this occurs, simply rename the element to a name that does not contain any of the invalid characters.

## MATLAB Element

The MATLAB element lets you run MATLAB scripts from within a test. In addition to specifying any valid MATLAB script to execute, you can incorporate any test variable into your code, as well as access any variables residing in the MATLAB workspace.

### Allowed Test Sections

The MATLAB element can be used in the following test sections:

- Pre Test
- Main Test
- Post Test
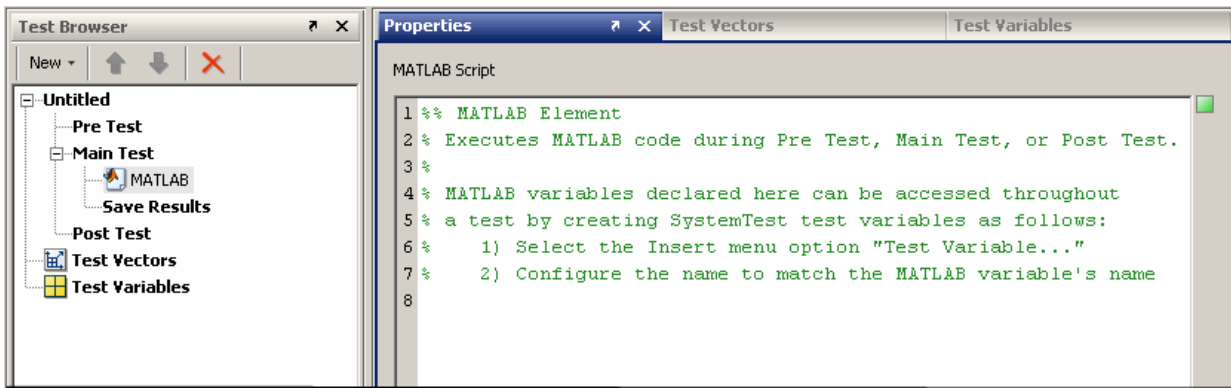
### Properties Pane

In the **MATLAB Script** edit field, enter any valid MATLAB script.

## Limit Check Element — General Check

The **General Check** tab of the Limit Check element determines whether test conditions are met by using scalar, vector, or matrix comparisons. It can be used to:

- Compare measured data to expected data.
- Stop an iteration or an entire test if a test constraint is violated.
- Assign a test variable the logical value derived from the comparison(s) for use by other elements.

You can do the following types of comparisons with the **General Check** tab of the Limit Check element:

- Scalar versus scalar
- Scalar versus vector
- Vector versus vector
- Matrix versus matrix

**Note:** Use the Tolerance Check tab of the Limit Check element to test absolute and relative tolerance.

**Allowed Test Sections**

The Limit Check element can be used in the following test section:

- Main Test
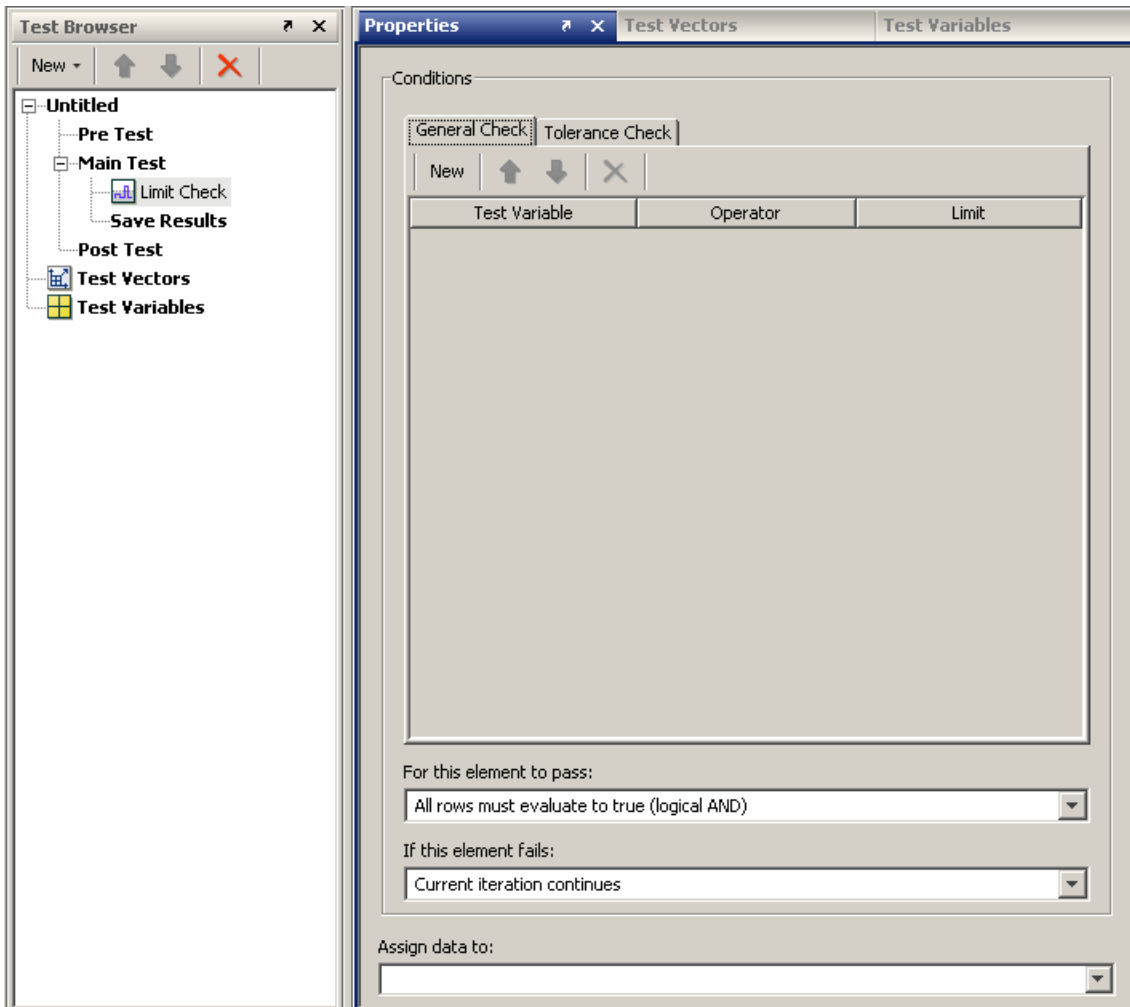


**How to Use**

1 Click the **New** button on the **General Check** tab to add a general limit check.

- Select an existing test variable or create a new one in the **Test Variable** column.

- Select an operator in the **Operator** column.

- Select an existing test variable or test vector or create a new one in the **Limit** column.

**2**  Set your test's passing conditions.

- The element can pass if all comparisons complete (a logical AND).

- The element can pass if one or more of the comparisons complete (a logical OR).

**3**  Set your fallback procedure if the element fails. You can:

- Allow the current iteration to continue executing.

- Stop the current iteration and proceed to the next iteration.

- Stop the test and proceed to Post Test.

**4**  Identify the SystemTest test variable you want to assign the logical value derived from the comparison(s) in the **Assign data to** field.

---

**Note:** Aside from setting limit checks on individual elements, you can set these properties for the entire test, reachable by clicking the test name in the **Test Browser**, to determine pass/fail criteria for the test as a whole.

---

### Properties Pane — General Check

You can set the following properties for the Limit Check element:

- **Test Variable** — Value to compare to limit using operator.

- **Operator** — Boolean operator used to compare test variable to limit.

- **Limit** — Value to compare to test variable using operator.

- **For this element to pass** — Choose between a logical AND (all comparisons must pass) or a logical OR (at least one comparison needs to pass) for the element to pass.

- **If this element fails** — Choose between continuing the test, stopping the current iteration, or stopping the entire test.

- **Assign data to** — Test variable assigned the logical value of this evaluation. The logical value will be 1 if the element passes or 0 if the element fails.

## Limit Check Element — Tolerance Check

The **Tolerance Check** tab of the Limit Check element verifies test conditions are met by using absolute and relative tolerance comparisons. It can be used to:

- Compare measured data to expected data.
- Stop an iteration or an entire test if a test constraint is violated.
- Assign a test variable the logical value derived from the comparison(s) for use by other elements.
- Define pass/fail criteria for an iteration.

You can do the following types of comparisons with the **Tolerance Check** tab of the Limit Check element:

- Absolute tolerance
- Relative tolerance

For a definition of these tolerance types, see the Properties Pane section.

**Note:** Use the General Check tab of the Limit Check element to test scalar, vector, and matrix comparisons.

### Allowed Test Sections

The Limit Check element can be used in the following test section:

- Main Test
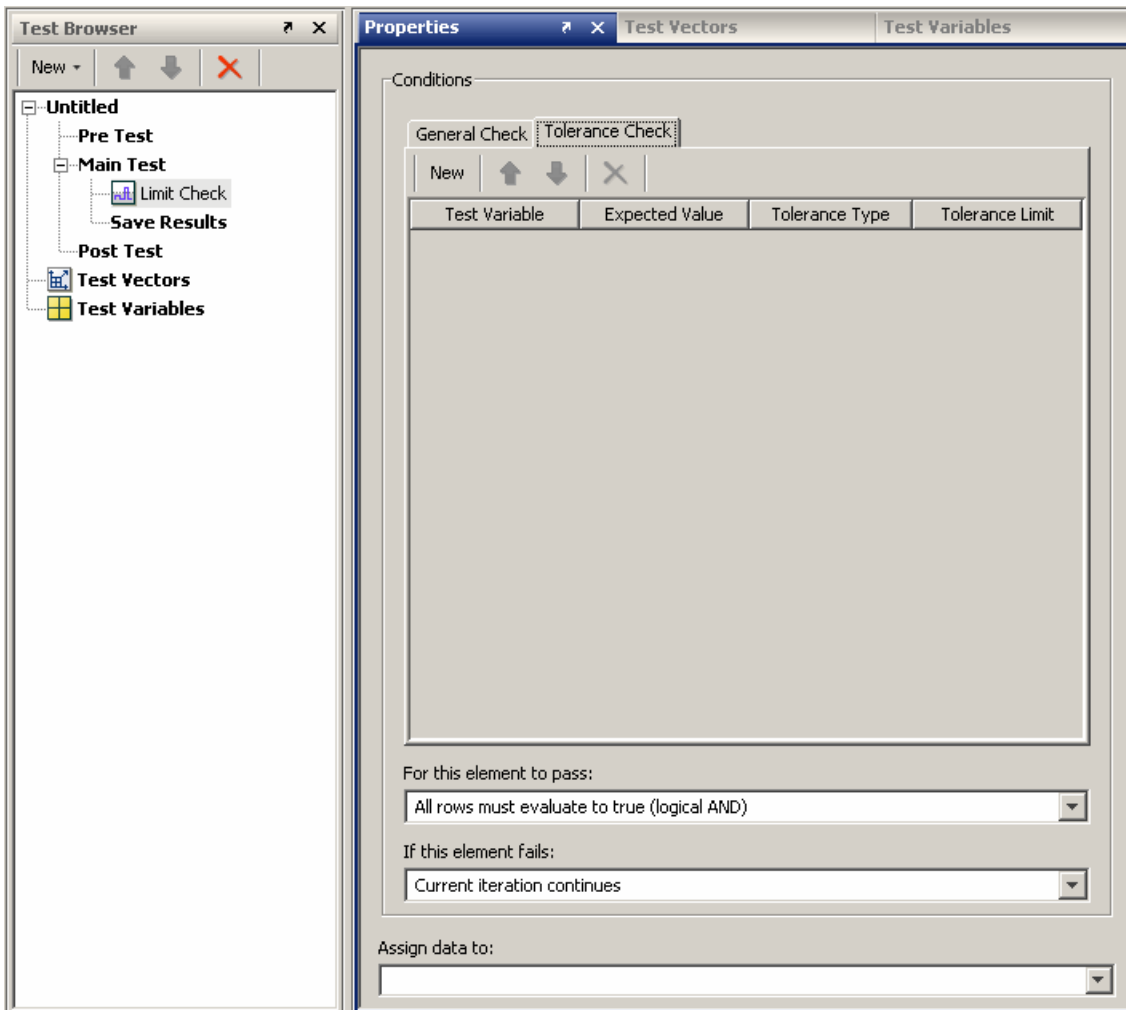
### How to Use

1 Click the **New** button on the **Tolerance Check** tab to add a tolerance limit check.

   • Select an existing test variable or create a new one in the **Test Variable** column.

   • Select an existing test variable or test vector or create a new one in the **Expected Value** column.

- Select `Absolute` or `Relative` in the **Tolerance Type** column.
- Select an existing test variable or test vector or create a new one in the **Tolerance Limit** column.

**2** Set your test's passing conditions.

- The element can pass if all comparisons complete (a logical AND).
- The element can pass if one or more of the comparisons complete (a logical OR).

**3** Set your fallback procedure if the element fails. You can:

- Allow the current iteration to continue executing.
- Stop the current iteration and proceed to the next iteration.
- Stop the test and proceed to Post Test.

**4** Identify the SystemTest test variable you want to assign the logical value derived from the comparison(s) in the **Assign data to** field.

---

**Note:** Aside from setting limit checks on individual elements, you can set these properties for the entire test, reachable by clicking the test name in the **Test Browser**, to determine pass/fail criteria for the test as a whole.

---

### Properties Pane — Tolerance Check

You can set the following properties for the Limit Check element.

- **Test Variable** — Variable to compare with expected value using a tolerance limit.
- **Expected Value** — Expected value to compare variable to using a tolerance limit.
- **Tolerance Type** — Tolerance type used to compare test variable to the expected value. Select `Absolute` or `Relative`. Absolute tolerance is calculated using this formula: `abs(test variable - expected value) <= tolerance limit`. Relative tolerance is calculated using this formula: `abs(test variable - expected value) <= tolerance limit.* abs(expected value)`.
- **Tolerance Limit** — Value used as the tolerance constraint to compare variable and expected value.
- **For this element to pass** — Choose between a logical AND (all comparisons must pass) or a logical OR (at least one comparison needs to pass) for the element to pass.
- **If this element fails** — Choose between continuing the test, stopping the current iteration, or stopping the entire test.

- **Assign data to:** — Test variable assigned the logical value of this evaluation. The logical value will be 1 if the element passes or 0 if the element fails.

## IF Element

The IF element provides logical control of a test by evaluating a condition.

The IF element allows sub-elements to run only when the IF element's condition evaluates to true. After adding an IF element, you should add one or more elements to perform a specific task.

### Allowed Test Sections

The IF element can be used in the following test sections:

- Pre Test
- Main Test
- Post Test
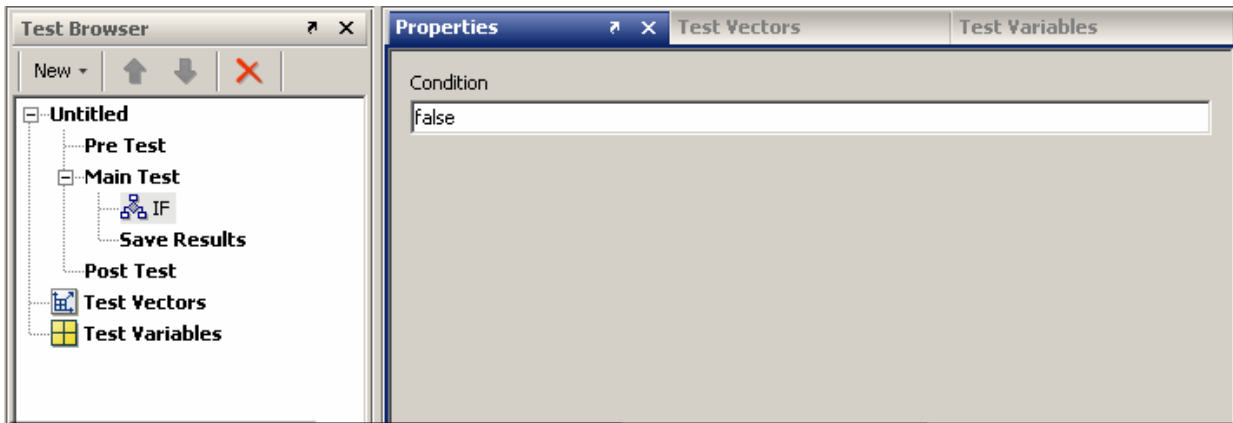


### Properties Pane

You can set the following property for the IF element.

- **Condition** — Enter a valid MATLAB expression that will evaluate to true or false.

## General Plot Element

The General Plot element is used to plot any type of data over multiple iterations.

Use this element during the Main Test to generate plots of any test vectors or test variables containing any type of data.

## Allowed Test Sections

The General Plot element can be used in the following test section:

- Main Test
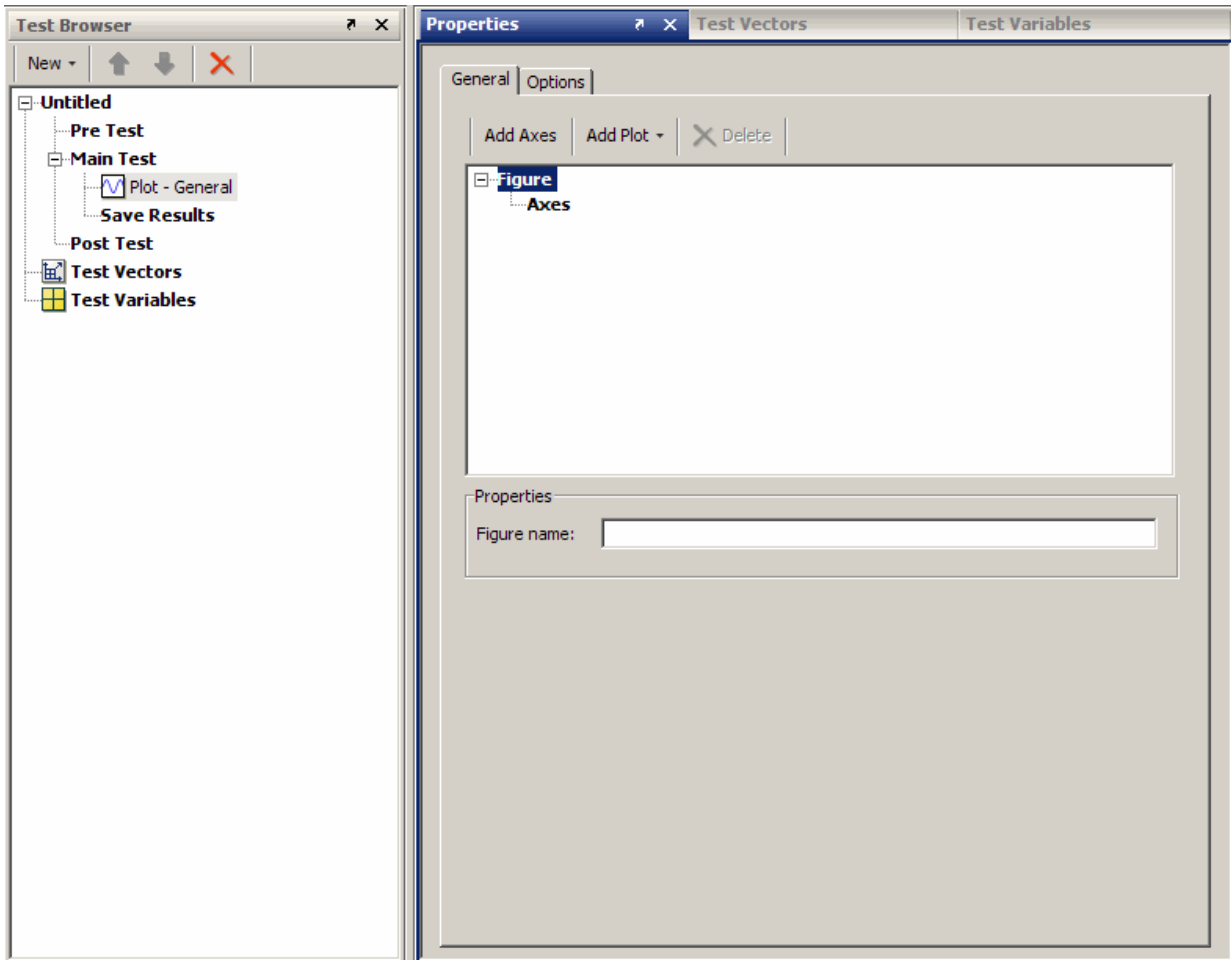
**General Tab**

To add a plot:

**1** Click the **Add Plot** button to create a plot.

**2** From the drop-down list, select one of the following:

- **plot** — A standard line plot that uses a 2-D line graph with linear axes.
- **Simulink data** — Lets you plot data produced from a Simulink model. The supported data types are such [time signal] array, a structure, a structure with time, or a time series. Note that the element creates a line for each signal in the Simulink data. If time is not present, the signals are plotted against their indices.

    You can also plot Simulink data provided by test vectors, such as the Signal Builder Block test vector, the Simulink Design Verifier Data File test vector, or the Spreadsheet Data test vector.

- **bar** — A standard bar plot that creates a bar graph.
- **scatter** — A standard scatter plot that creates a 2-D scatter graph displaying markers at x- and y-coordinates.
- **contour** — A standard line plot that creates a 3-D contour graph displaying isolines of a surface in a 3-D view.
- **imagesc** — An image plot with colormap scaling, which displays an image and scales it to use the full colormap.
- **surf** — A standard surface plot that creates a 3-D surface plot that displays a matrix as a surface.
- **mesh** — A standard surface plot that creates a 3-D mesh plot displaying a matrix as a wireframe surface.
- **More plots** — Opens the Choose Plot Type dialog box, which lets you choose any MATLAB plot. Select a plot type category in the **Categories** list to display the plot types from the **Plot Types** list. Select an individual plot type to read the **Description**.

**Add Axes Button**

You can have multiple axes in a plot figure. To add an axes, click the **Add Axes** button. Then click the **Add Plot** button to create the plot for that axes. Each axes is added as a subplot to the parent figure.

You can set properties for each axes individually by selecting the axes and then configuring properties in the **Properties** area. With the axes selected, you can configure the X and Y labels and add a title and legend. With the plot under the axes selected, you can configure the plot.

**Properties**

When the **Figure** node is selected or you have not yet added a plot, the **Figure name** field is displayed. Optionally use this text field to name the plot.

When you select a plot type and it is added to the tree, the **Properties** section displays the properties of that plot type. Fill in any parameters you want to set. For more information on the parameters, see the help in the Choose Plot Type dialog box when you select **More Plots**.

When you select an axes the axes properties are displayed. Use the **X label** and **Y label** fields to enter names for the X and Y axes. Use the **Title** field to enter a title for the plot. If you select the **Include legend** option, a legend is added to the plot. The legend is located in the least used space outside of the plot.

You can set other options for the General Plot element by clicking the **Options** tab.

**Plotting Simulink Data**

You can plot data produced from a Simulink model. The supported data types are such [time signal] array, a structure, a structure with time, or a time series. Note that the element creates a line for each signal in the Simulink data. If time is not present, the signals are plotted against their indices.

You can also plot Simulink data provided by test vectors, such as the Signal Builder Block test vector, the Simulink Design Verifier Data File test vector, or the Spreadsheet Data test vector.

The Simulink data types are plotted as follows:

• For an array, it is plotted against its indices.

- For a structure in the format generated by a Simulink Outport, its signal values are plotted against its indices.

- For a structure with time in the format generated by a Simulink Outport, its signal values are plotted against its time.

- For a structure with time in the format generated by the Signal Builder Block test vector, its signal values are plotted against its time.

- For a `Simulink.Timeseries` object, the plot is determined by the `plot()` function of the `Simulink.Timeseries` object.

### Options Tab

These options control the test behavior pertaining to plots.
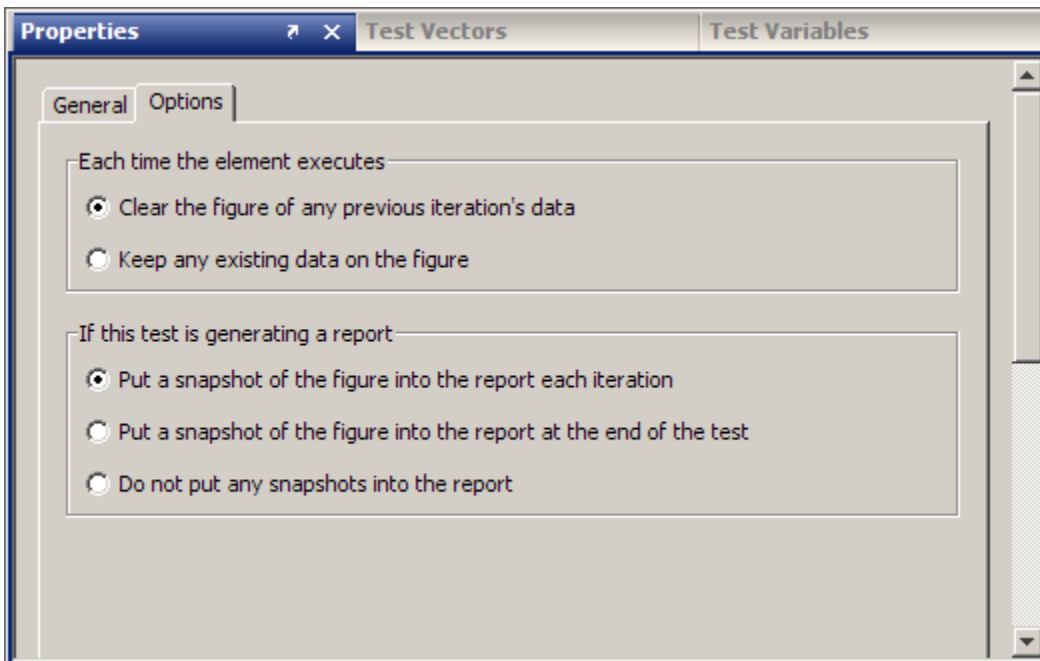


The **Each time the element executes** option determines run-time behavior of the element.

- **Clear the figure of any previous iteration's data** – Every time the element executes, the figure is cleared before plotting new data. This is the default.

- **Keep any existing data on the figure** – Previous plots are not removed from the figure. New data is added to the same figure.

The **If this test is generating a report** option determines what happens to the snapshots of the plots that are created when each iteration runs.

- **Put a snapshot of the figure into the report each iteration** – A snapshot of the plot is generated in each iteration and is displayed in its respective section of the report. This is the default.
- **Put a snapshot of the figure into the report at the end of the test** – Only one snapshot of the plot is taken, at the end of the completed test run. It is displayed in the report section for Post Test.
- **Do not put any snapshots into the report** – No snapshots of plots are added to the report.

## Stop Element

The Stop element stops an iteration or an entire test unconditionally.

You can use the Stop element with conditional logic elements, such as the IF element, to control the test's execution.

### Allowed Test Sections

The Stop element can be used in the following test sections:

- Pre Test
- Main Test
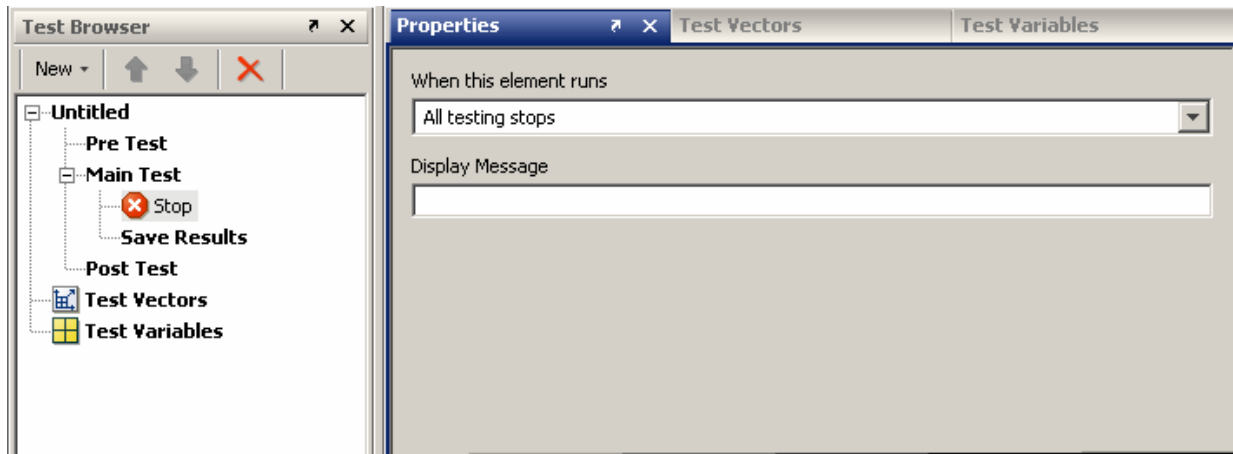
**Properties Pane**

You can set the following properties for the Stop element.

- **When this element runs** — Select a stop action for use in Main Test. The **Current iteration stops** option stops the current Main Test iteration. The **All testing stops** option stops all Main Test iterations and runs Post Test.

  Note that when a Stop element is used in Pre Test, **All testing stops** is the only option, since Pre Test does not have iterations.

- **Display Message** — Enter a message to display in the Test Report.

## Subsection Element

Use subsection elements to organize one or more elements to maintain readability of your test or to better manage complex test structures. Use a subsection to:

- Group elements under a single root element.
- Organize tests.
- Manage complex test structures.

**Allowed Test Sections**

The Subsection element can be used in the following test sections:

- Pre Test
- Main Test
- Post Test
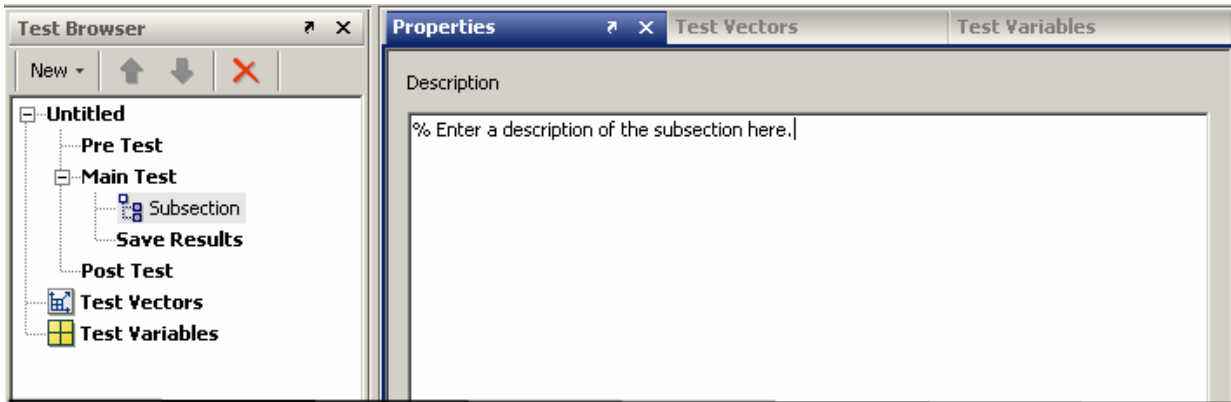


**Properties Pane**

You can set the following properties for the Subsection element.

- **Description** — Type in your description of the subsection.

# Deprecated Elements

| **In this section...** |
| --- |
| "Converting Elements" on page 3-22 |
| "Scalar Plot Conversion Details" on page 3-23 |
| "Vector Plot Conversion Details" on page 3-25 |

## Converting Elements

The Scalar Plot and Vector Plot elements are being replaced by the General Plot element that was introduced in R2008b. The General Plot element supports all MATLAB plot types as well as Simulink data.

**Note:** You can no longer create new Scalar Plot or Vector Plot elements. They no longer appear in the **Insert** menu or the **New Element** button in the SystemTest desktop.

Tests containing Scalar Plot or Vector Plot elements will not automatically load with those elements. You will be prompted to convert them to General Plot elements. For example, if you load a test containing a Scalar Plot element, the following dialog box opens:
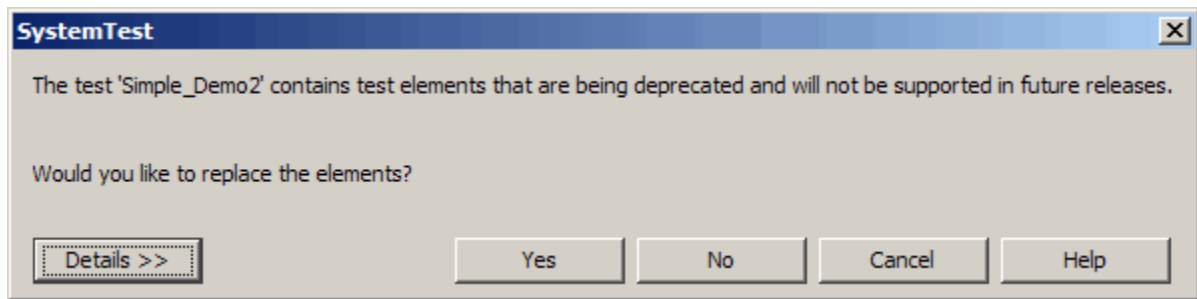


If you click the **Details** button, the dialog box shows the specific element(s) that will be converted.
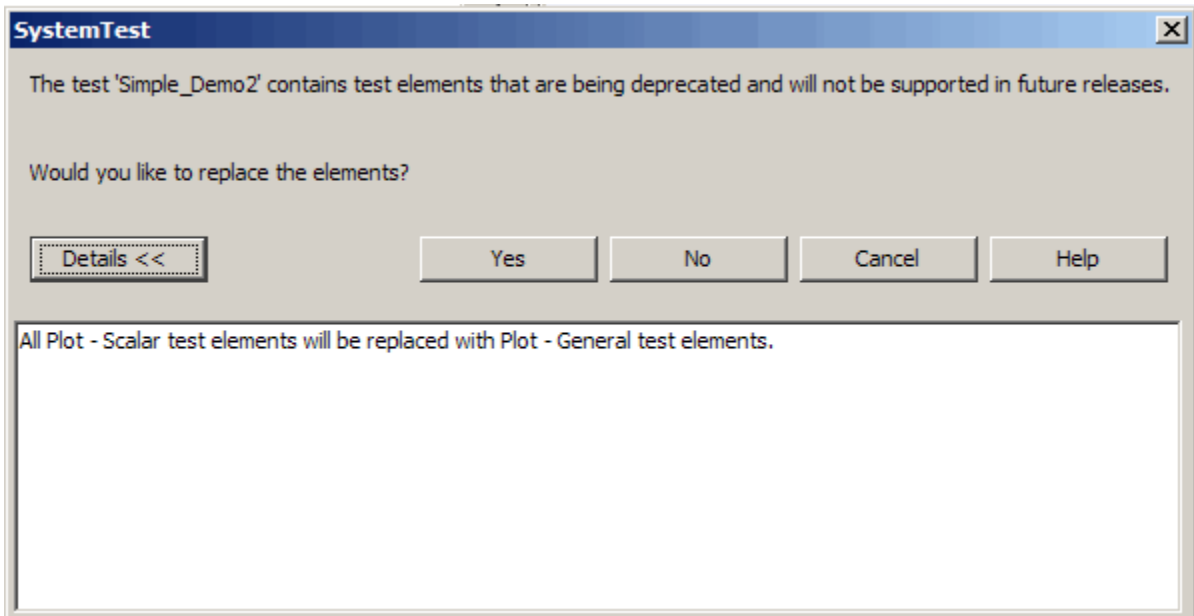
In this case, the test contained one or more Scalar Plot elements.

Choose the conversion option as follows:

• **Yes** — Your Scalar Plot and/or Vector Plot elements are converted to General Plot elements. The test is not saved until you explicitly do so.

See the next two sections for conversion details.

• **No** — Your Scalar Plot and/or Vector Plot elements are not converted to General Plot elements. The test loads with the old elements. You can save the test with the old elements. However, in a future release, you will not be able to load the test until you convert the elements.

• **Cancel** — The test is not loaded.

For information about the General Plot element, see "General Plot Element" on page 3-14.

## Scalar Plot Conversion Details

When you convert a Scalar Plot element to a General Plot element, these rules apply:

- Each Scalar Plot element maps one to one to a new General Plot element with the same name.
- Each General Plot element is created with default values.
- For each row in the **Lines to Plot** table in the Scalar Plot element, these actions occur in the same order as the rows appear in the table.

| Scalar Plot Element Component | What Is Loaded in the General Plot Element |
|---|---|
| **Plot Type** drop-down list | A "plot" plot type is added to the axes. |
| **X Axis** drop-down list | Mapped to **X Data Source** of the newly added plot type. |
| **Y Axis** drop-down list | Mapped to **Y Data Source** of the newly added plot type. |
| **Line Color** drop-down list | Mapped to **Line color** of the newly added plot type. |
| **Line Style** drop-down list | Mapped to **Line style** of the newly added plot type. |
| **Line Marker** drop-down list | Mapped to **Line marker** of the newly added plot type. |
| **Maximum Number of Points to Display at Once** option | This parameter is not converted and is ignored.<br>Note that every point in the plot is retained. There is no limit to the number of points. |

This figure shows an example of a Scalar Plot element (left) converted to a General Plot element (right).
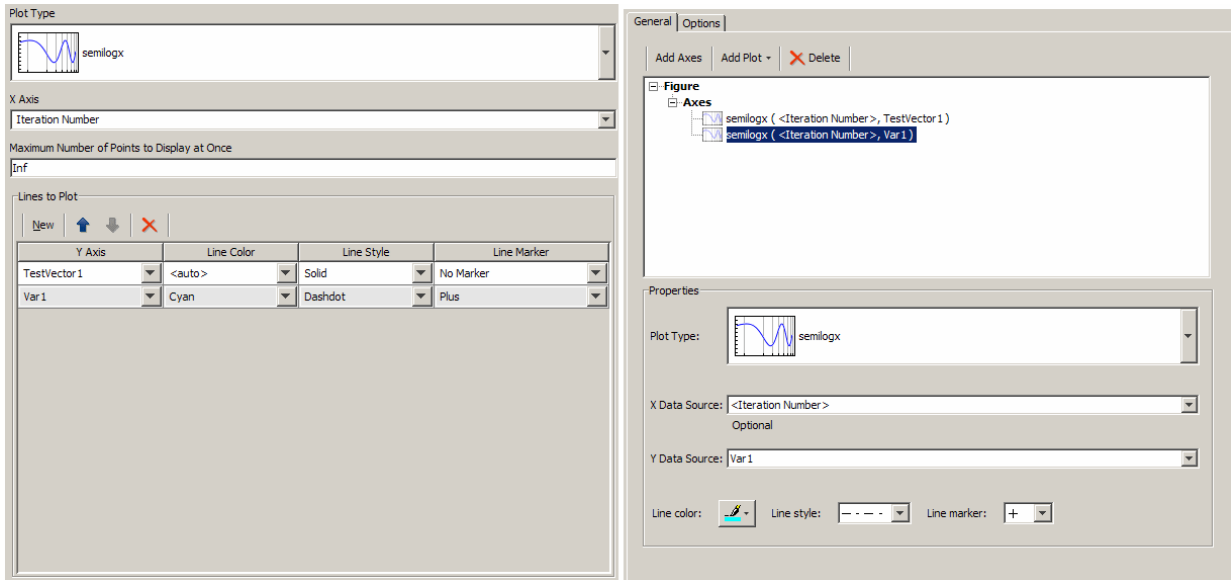


## Vector Plot Conversion Details

When you convert a Vector Plot element to a General Plot element, these rules apply:

- Each Vector Plot element maps one to one to a new General Plot element with the same name.
- Each General Plot element is created with default values.
- For each row in the **Lines to Plot** table in the Vector Plot element, these actions occur in the same order as the rows appear in the table.

| Vector Plot Element Component | What Is Loaded in the General Plot Element |
|---|---|
| **Plot Type** selector | A "plot" plot type is added to the axes. |
| **X Axis** drop-down list | Mapped to **X Data Source** of newly added plot type. If the selection was "none" it is mapped to "<Auto>". |
| **Y Axis** drop-down list | Mapped to **Y Data Source** of the newly added plot type. |
| **Line Color** drop-down list | Mapped to **Line color** of the newly added plot type. |
| **Line Style** drop-down list | Mapped to **Line style** of the newly added plot type. |
| **Line Marker** drop-down list | Mapped to **Line marker** of the newly added plot type. |
| **Subplot Dimensions** area | This is not converted and is ignored. |
| **Clear axes between iterations** option | If selected, on the **Options** tab, the **Clear the figure of any previous iteration's data** option becomes selected. If cleared, the **Keep any existing data on the figure** check box becomes selected. |

This figure shows an example of a Vector Plot element (left) converted to a General Plot element (right).

**4**

# Using the Simulink Element

# Simulink Element

The Simulink element allows you to override the inputs to a Simulink model with SystemTest test vectors and test variables. You can further map the model's outputs to SystemTest test variables for later processing by other test elements. This means that you can use the SystemTest software to define, generate or load input data, feed it into Simulink, run the model while iterating over the input data, and map the outputs back into the SystemTest software.

**Note:** In Simulink elements, you cannot have more than one model with the same name. Each model referenced within a test must have a unique name. If you ran a test containing two models with the same name, the SystemTest software would only use one of the models.

# Before You Begin

Load the Inverted Pendulum example from MATLAB and delete the Simulink element from the example. The following steps describe how to do this:

1 Start MATLAB.

2 Enter the following command at the MATLAB command line:

```
systemtest InvertedPendulum
```

The SystemTest desktop opens with the Inverted Pendulum example loaded.

3 Click the **Simulink** element in the **Test Browser**.
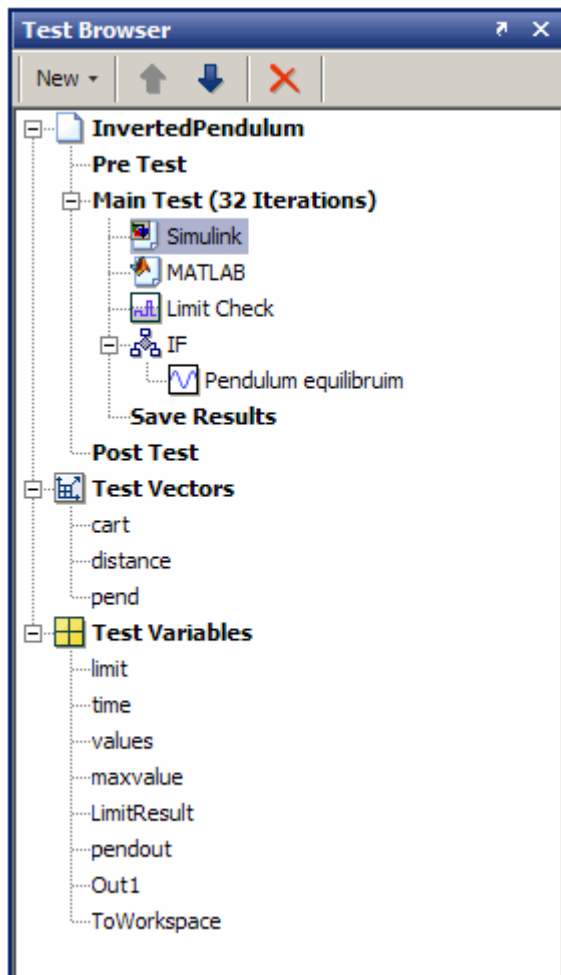
4   Click the **Delete element** button in the Test Browser button bar or press the
     **Delete** key.

# Configuration of a Simulink Element

| In this section... |
| --- |
| |
| |
| |
| |
| |
| |
| |

## Introduction

To help you learn how to use the Simulink element, this section walks you through the configuration of the Simulink element for the Inverted Pendulum test. The Inverted Pendulum example includes both a model of the pendulum and a model of a controller that keeps the inverted pendulum balanced. Moving the bottom of the pendulum disturbs the equilibrium, causing the pendulum to move and the controller to rebalance it. The Inverted Pendulum test varies the mass of the pendulum, the mass of the cart the pendulum is on, and the distance to the pendulum's center of mass, testing the robustness of the controller as it attempts to return the pendulum to equilibrium. Using the Simulink element in a test lets you vary the model inputs and assess the model outputs.

**Note:** The following sections assume you have loaded the Inverted Pendulum example and deleted the Simulink element, as explained in "Before You Begin" on page 4-3.

## Add a Simulink Element

To add a Simulink element to a test, click the **New > Test Element** button in the **Test Browser** and select the Simulink element. If you have a license for Simulink, the element list contains the Simulink element, as shown in the following figure.
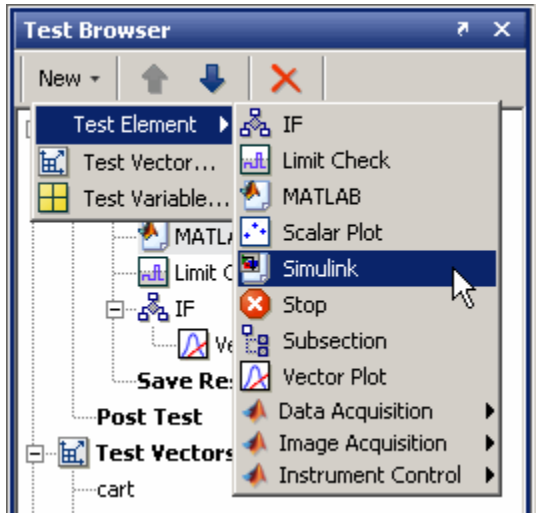
The SystemTest software adds the Simulink element to the test and opens the Simulink element **Properties** pane.
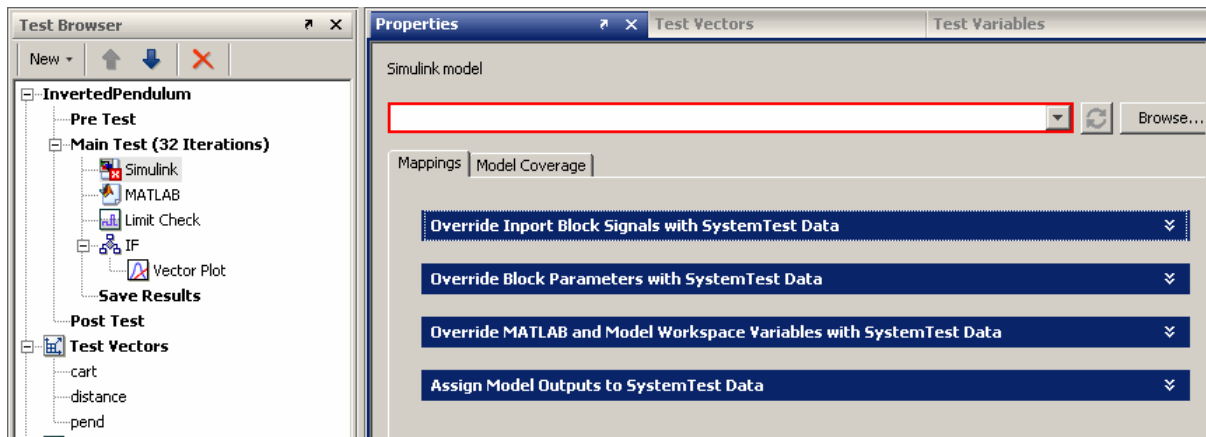


## Specify the Simulink Model

When you first add the element, the icon in the **Test Browser** has a red x, meaning that the element requires some information. The **Simulink model** field in the Simulink element **Properties** pane is outlined in red, indicating that it is a required field. You

must specify the model that the Simulink element will interact with. If the model is on the MATLAB path, you can type its name in the **Simulink model** field. If you are not sure of the name, or the model is not on the path, you can browse to its location using the browse button.

For the Inverted Pendulum example, type `systemtestpendulum` in the **Simulink model** field and press **Enter**. The SystemTest software opens the `systemtestpendulum` model in Simulink and opens the Pendulum Visualization window.

## Override Simulink Model Inputs

Using test vectors and test variables, you can override the following Simulink model inputs:

*   Block parameters — Described in "Override Simulink Block Parameters" on page 4-7
*   Model and base workspace variables — Described in "Override to Workspace Variables" on page 4-9
*   Inport signals — Described in "Override Simulink Model Inport Signals" on page 4-11

### Override Simulink Block Parameters

You can override Simulink block parameters with SystemTest test vectors or test variables. When you run the test, Simulink runs the model using data provided by the SystemTest software. Overriding does not change your Simulink model file; it only overrides in the test. The procedure for creating block parameter overrides requires that you select your block in the Simulink model, but everything else you need to do happens within the Simulink element **Properties** pane.

**4-7**

To override a Simulink block parameter:

1  In the **Mappings** tab of the **Properties** pane for the Simulink element in the
   SystemTest software, expand the **Override Block Parameters with SystemTest
   Data** section and click the **New Mapping** button, and select **Select Block to Add**.
   This opens the model in Simulink, if it is not already open.

2  In the Simulink model window, click the block containing the parameter you want to
   override. For this example, click the Pendulum block in the **systemtestpendulum**
   model window.

3  In the SystemTest software, return to the Simulink element **Properties** pane and,
   in the **Override Block Parameters** section, you'll see that the **Pendulum** was
   added. If you click the **New Mapping** button again, you'll see that the SystemTest
   software also adds an entry to this menu for the block.
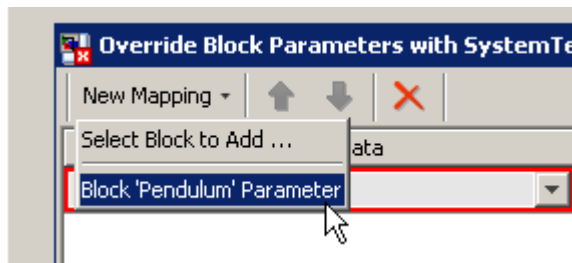


In the override table, the **Simulink Data** field shows that this entry is linked to the
Pendulum block but the question mark (?) indicates that no parameter for the block
has been mapped.
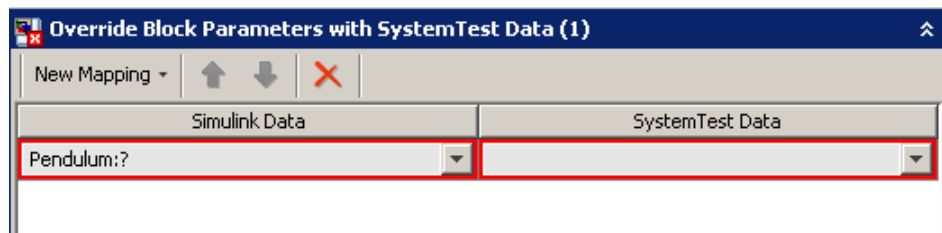


4  Select the parameter from the block that you want to map. Click the **Simulink Data**
   field for the block and select a parameter from the list. For the Inverted Pendulum
   example, select Pendulum:Mass of cart (kg).

5  Specify the SystemTest test vector or test variable you want to map to this block
   parameter. Click the **SystemTest Data** field for the block parameter. This shows

you all defined SystemTest test vectors and test variables available for mapping. For this example, select **cart**.
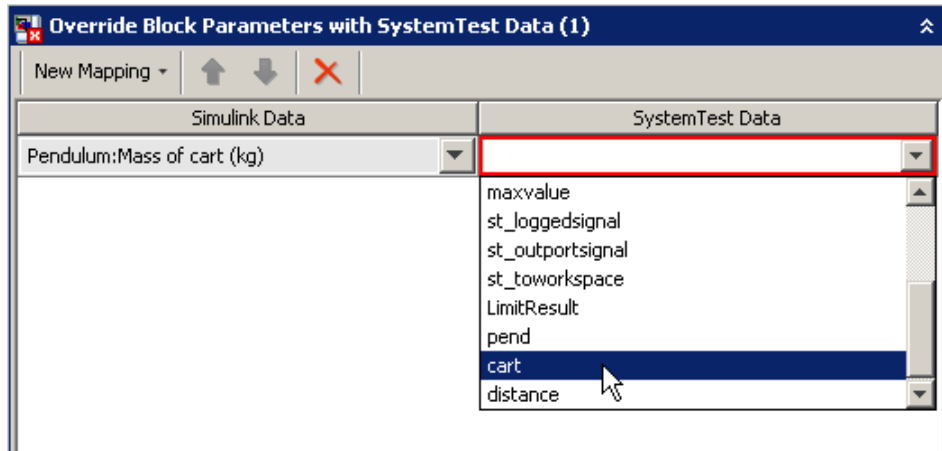


### Override to Workspace Variables

You can use a SystemTest test vector or test variable to override either a MATLAB base workspace variable or a Simulink model workspace variable. This lets you define test values and conditions in the SystemTest software and have a Simulink model act on them.

This section describes how you can use the values in the `pend` and `distance` test vectors to override the model workspace variables `masspend` and `penddistance` in the Inverted Pendulum example.

To override workspace variables:

**1**   Expand the **Override MATLAB and Model Workspace Variables with
SystemTest Data** area of the Simulink element **Properties** pane, and click the
**New Mapping** button.

**2**   Select the workspace variable you want to override. Click the **Simulink Data**
field of this row to see all available base workspace variables and Simulink model
workspace variables. For the Inverted Pendulum example, select **masspend**.
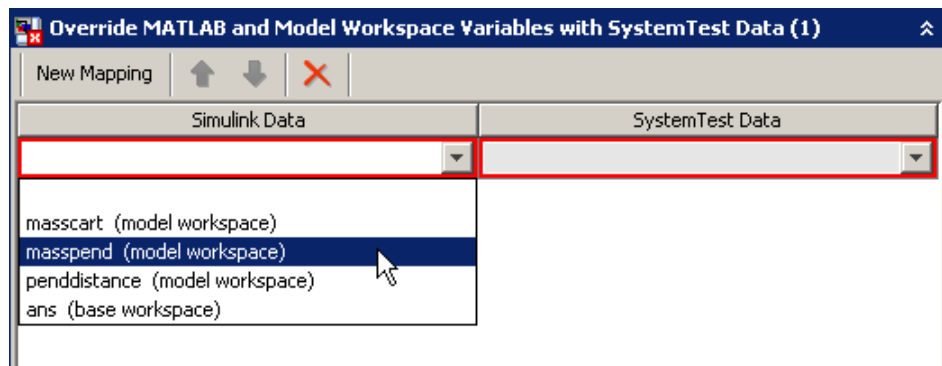


**3**   Specify which SystemTest test vector or test variable you want to map to the
Simulink workspace variable. Click the **SystemTest Data** field of this row to see all
available test vectors and test variables. For this example, select **pend**.
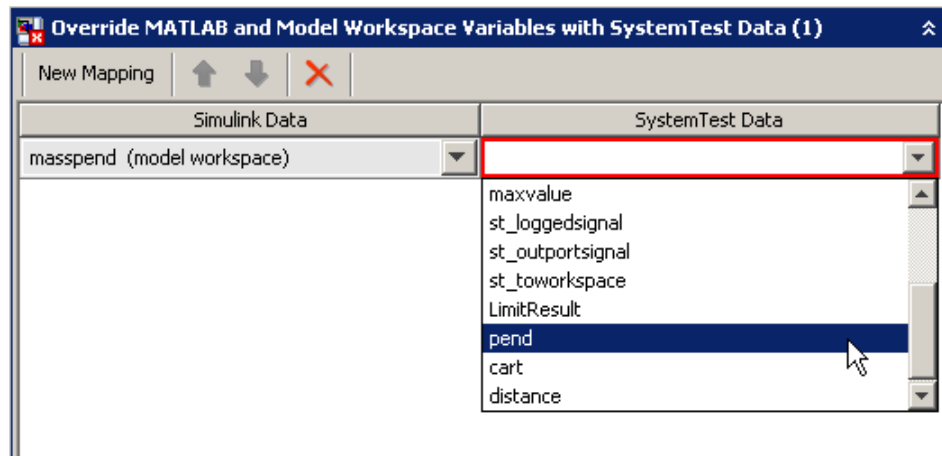
**4**   Repeat steps 1 to 3 to override the Simulink model workspace variable `penddistance` with the SystemTest test vector `distance`.

| Override MATLAB and Model Workspace Variables with SystemTest Data (2) | | ≫ |
|---|---|---|
| New Mapping   ⬆ ⬇ ✕ | | |

| Simulink Data | | SystemTest Data | |
|---|---|---|---|
| masspend  (model workspace) | ▼ | pend | ▼ |
| penddistance  (model workspace) | ▼ | distance | ▼ |
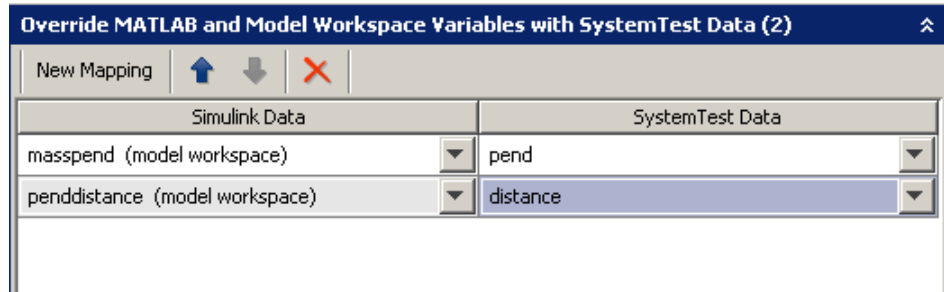
### Override Simulink Model Inport Signals

As with block parameters and workspace variables, you can use the SystemTest software to override a model's inport signals. This lets you externally manipulate the input signal of a Simulink model.

The Inverted Pendulum example does not override any inport signals.

For information on how to override inport signals and an example, see "Override Inport Block Signals" on page 4-21.

**Optimizing Test Vectors to Work with Inport Signals**
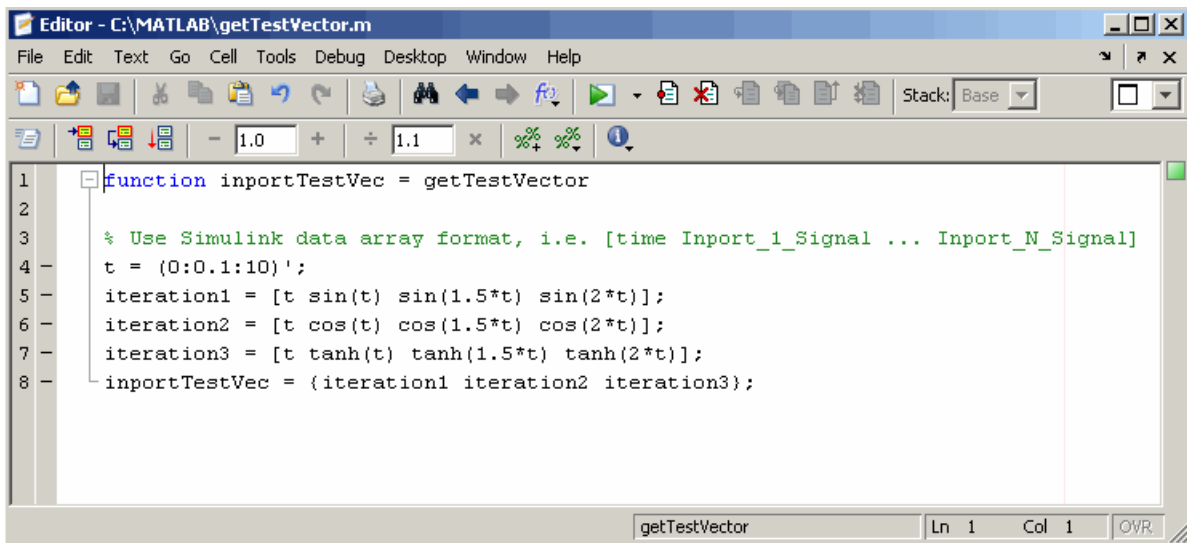
Simulink allows you to import input signal and initial state data from the MATLAB workspace and export output signal and state data to the MATLAB workspace during simulation. In the SystemTest software, you can specify the contents of a test vector so that it is used as a Simulink inport. To do that, use the vector as the mapping in your Simulink element, by selecting it in the **SystemTest Data** row as described above.

The Simulink documentation contains guidance on importing data to Simulink inport signals. You can create the same type of data in your SystemTest test vectors that you then map to inport signals.

**Example for Overriding Inport Signals Using Data Arrays**

One of the data formats described in "Data Format for Exported Simulation Data" is the use of data arrays for specifying input data to an Inport block. This example uses the systemtestinputdemo model to illustrate how the SystemTest software can be used to override the three Inport blocks in the model with test signals.

The first step involves constructing a test vector that specifies the different signal test cases. This can be done by creating a MATLAB function that simply returns a test vector containing the different test cases you would like to use for each test iteration. A sample MATLAB function, called GETTESTVECTOR, that does this is provided below.

Once this function is saved as GETTESTVECTOR, you can create a SystemTest test vector whose expression is set to GETTESTVECTOR. This will create a 1-by-3 test vector cell array within the SystemTest software, where each entry in the cell array represents the time and signal data for the three Inport blocks.

## Map Simulink Model Outputs to Test Variables

Using test variables you can assign the output from the following types of Simulink model data:

- Logged signals — Described in "Map Simulink Logged Signals to Test Variables" on page 4-13
- Outport signals — Described in "Map Simulink Outport Signals to Test Variables" on page 4-16
- To Workspace blocks — Described in "Map Simulink To Workspace Blocks to Test Variables" on page 4-17

After you map model outputs to test variables, you can incorporate the model data into the SystemTest software. This section shows you how to map this data for the Inverted Pendulum example.

---

**Note:** The output from Simulink models can only be mapped to SystemTest test variables. You cannot map this output to SystemTest test vectors.

---

**Note:** If you are mapping logged signals, outport signals, or To Workspace blocks to test variables, as shown in the following procedures, then you can optionally use the Mappings Assistant if you want the variables to have the same names as the inputs. This is useful if your model contains many signals or blocks and you want to name the outputs the same way. You no longer have to create test variables with matching names manually. See "Model Output Mappings Assistant" on page 4-19 for more information.

---

### Map Simulink Logged Signals to Test Variables

Logged signals are a way to obtain outputs from a model without adding more outports. Using logged signals, you can identify a particular signal and map the output to a SystemTest test variable.

To map logged signals to a SystemTest test variable:

1  Expand the **Assign Model Outputs to SystemTest Data** section of the Simulink element **Properties** pane, and click the **New Mapping** button. From the list, select **Logged Signal**. The SystemTest software adds a row for a new mapping of this type.
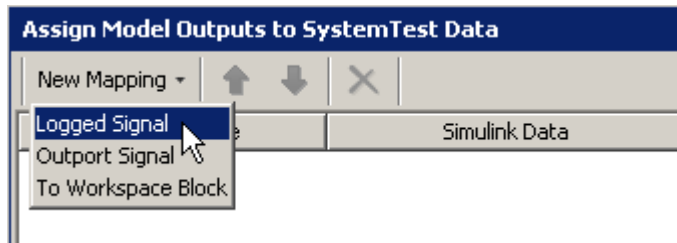


2  Specify the signal you want to capture. Click the **Simulink Data** field to see all the signals in the model. For the Inverted Pendulum example, select **pendout**.

> **Note:** If you added logged signals to your model and they do not appear in this list, click the refresh button, on the **Properties** pane next to the model name, to update the list.
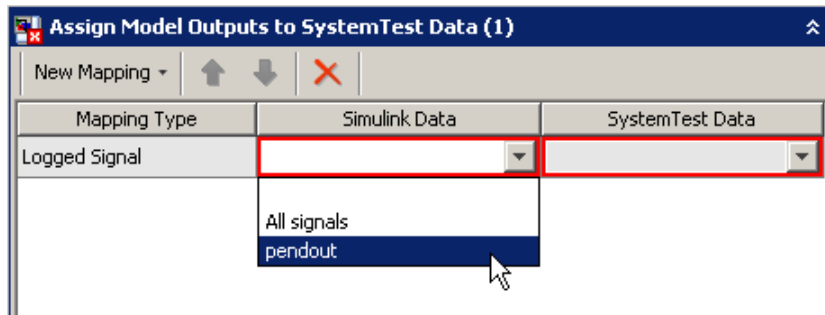


3  Specify the SystemTest test variable to which you want to map the output. Click the **SystemTest Data** field and select a test variable. For the Inverted Pendulum example, select `st_loggedsignal`.
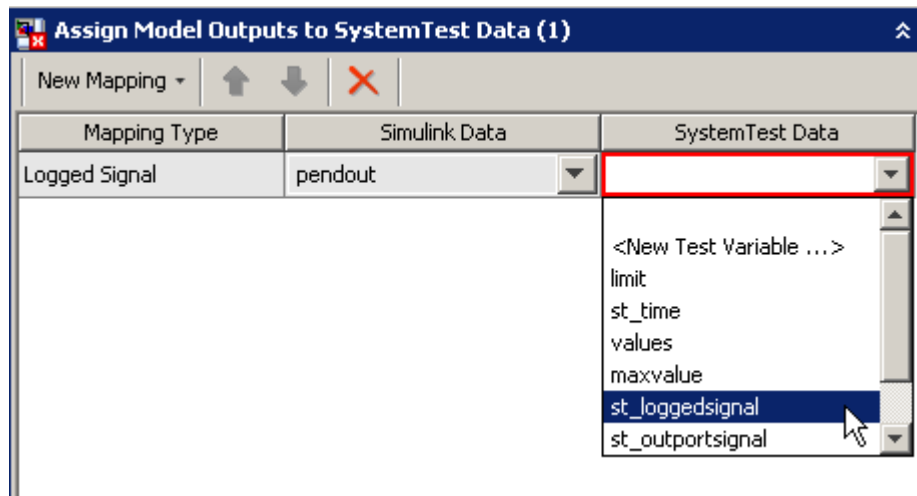
The SystemTest software creates the mapping to the test variable.
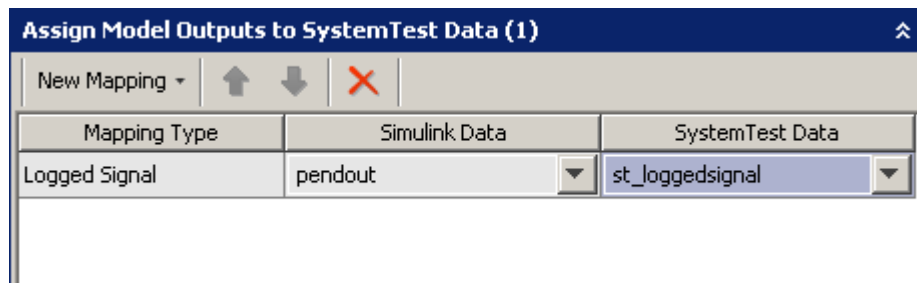


**Note:** If you are mapping logged signals to test variables, as shown in the above procedure, then you can optionally use the Mappings Assistant if you want the variables to have the same names as the inputs. This is useful if your model contains many signals or blocks and you want to name the outputs the same way. You no longer have to create test variables with matching names manually. See "Model Output Mappings Assistant" on page 4-19 for more information.

### Map Simulink Outport Signals to Test Variables

The SystemTest software lets you map all outport signals to a test variable for further processing in the SystemTest software.

To map Simulink outport signals to a test variable:

1 In the **Assign Model Outputs to SystemTest Data** section of the Simulink element **Properties** pane, click the **New Mapping** button. From the list, select **Outport Signal**. The SystemTest software adds a row for a new mapping of this type.

2 Specify the outport signal you want to capture. Click the **Simulink Data** field and select a signal. For this example, select **Out1**.



3 Specify the SystemTest test variable to which you want to map the outport signals. Click the **SystemTest Data** field and select a test variable from the list. For this example, select **st_outportsignal**.

The SystemTest software creates the mapping to the test variable.

| Assign Model Outputs to SystemTest Data (2) | | | ⌃ |
|---|---|---|---|
| New Mapping ▾   ⬆ ⬇   ✕ | | | |
| Mapping Type | Simulink Data | SystemTest Data | |
| Logged Signal | pendout ▾ | st_loggedsignal ▾ | |
| Outport Signal | Out1 ▾ | st_outportsignal ▾ | |
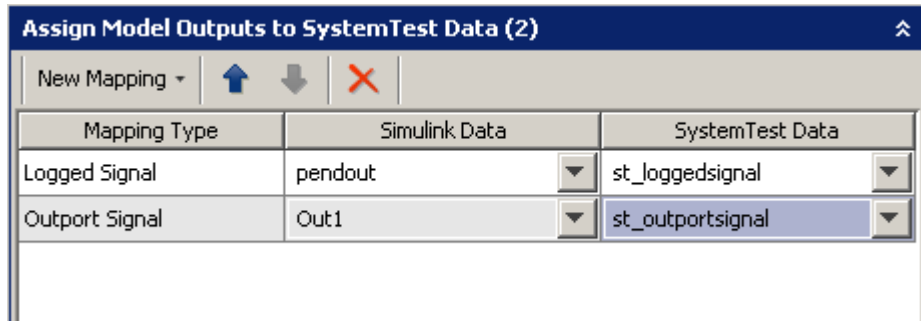
**Note:** If you are mapping outport signals to test variables, as shown in the above procedure, then you can optionally use the Mappings Assistant if you want the variables to have the same names as the inputs. This is useful if your model contains many signals or blocks and you want to name the outputs the same way. You no longer have to create test variables with matching names manually. See "Model Output Mappings Assistant" on page 4-19 for more information.

### Map Simulink To Workspace Blocks to Test Variables

When Simulink runs a model with To Workspace blocks, these blocks save model information in the MATLAB workspace as variables. Using the SystemTest software, this data can be mapped to SystemTest test variables.

This section shows how you create To Workspace block mappings in the SystemTest software using the Inverted Pendulum example.

To map the To Workspace block:

1   In the **Assign Model Outputs to SystemTest Data** section of the Simulink element **Properties** pane, click the **New Mapping** button. From the list, select **To Workspace Block**. The SystemTest software adds a row for a new mapping of this type.

2   Specify the To Workspace block in the model that you want to capture. Click the **Simulink Data** field and select the block from the list. For this example, select **To Workspace**.
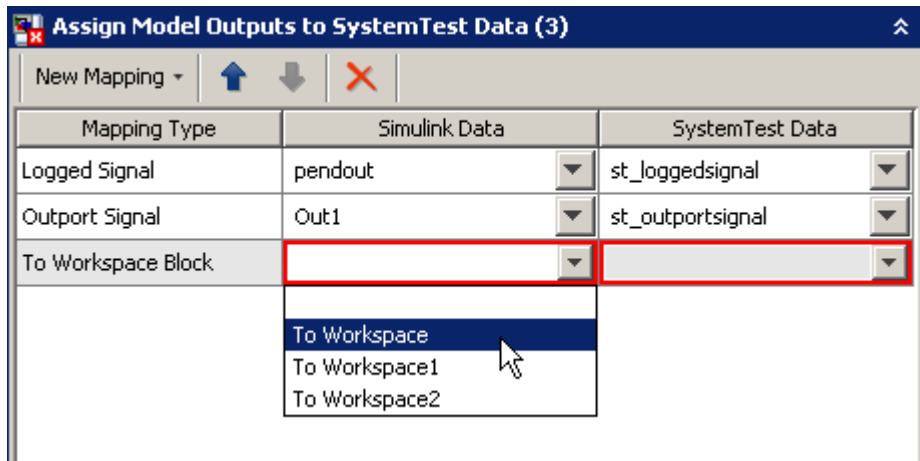
**3** Specify the SystemTest test variable to which you want to map the To Workspace block. Click the **SystemTest Data** field and select a test variable from the list. For this example, select **New Test Variable** to create a test variable.
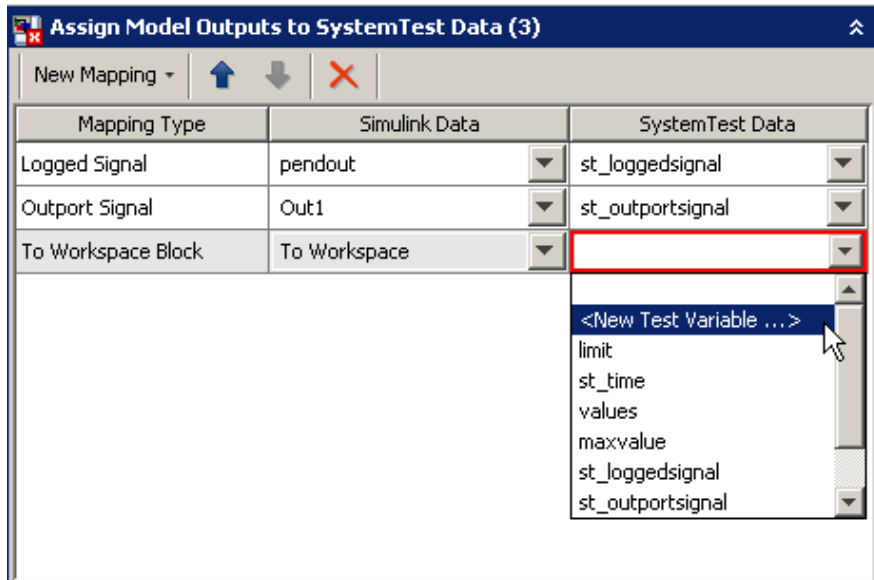
The SystemTest software opens the Edit Variable dialog box. Assign a name to the test variable and optionally an initial value, and then click **OK**. Name the test variable `ToWSResult`.
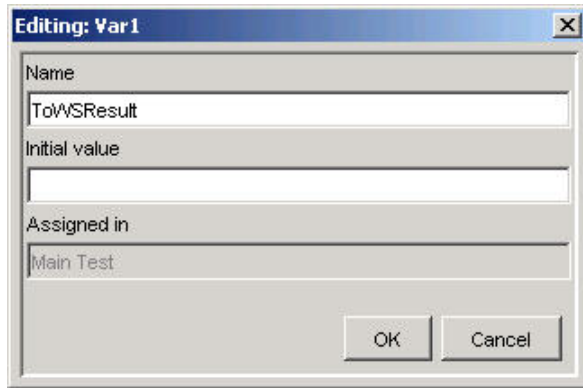
**Note:** If you are mapping To Workspace blocks to test variables, as shown in the above procedure, then you can optionally use the Mappings Assistant if you want the variables to have the same names as the inputs. This is useful if your model contains many signals or blocks and you want to name the outputs the same way. You no longer have to create test variables with matching names manually. See "Model Output Mappings Assistant" on page 4-19 for more information.

## Model Output Mappings Assistant

If you are mapping logged signals, outport signals, or To Workspace blocks to test variables, for example in the procedures in the above section "Map Simulink Model Outputs to Test Variables" on page 4-13, then you can optionally use the Mappings Assistant if you want the variables to have the same names as the inputs. This is useful if your model contains many signals or blocks and you want to name the outputs the same way. You no longer have to create test variables with matching names manually. Using the Mappings Assistant is the preferred method of setting up mappings since it is easier.

1   In the **Assign Model Outputs to SystemTest Data** section of the Simulink element, click the **Mappings Assistant** button above the table.

**2** In the Model Output Mappings Assistant dialog box, choose your mapping(s) in the **Create mappings for each** section.

- If your model contains any logged signals, the **Logged Signals** option is available. Select the option to map the logged signal(s) to test variable(s). If the model contains no logged signals, this option is disabled.

- If your model contains any outport signals, the **Outport Signals** option is available. Select the option to map the outport signal(s) to test variable(s). If the model contains no outport signals, this option is disabled.

- If your model contains any To Workspace blocks, the **To Workspace Blocks** option is available. Select the option to map the block(s) to test variable(s). If the model contains no To Workspace blocks, this option is disabled.

**3** Click **OK** to create the mappings.

The **Simulink Data** column then displays the names of the logged signals, outports, or To Workspace blocks that the model contained. The **SystemTest Data** column displays the test variables created with the same name.

For example, if the model contains two outports called Out1 and Out2, the **Simulink Data** column displays Out1 and Out2, and the **SystemTest Data** column displays Out1 and Out2 to represent the test variables that were created.

## Edit a Test Vector or Test Variable from within the Element

If you want to edit a test vector or test variable while working in the Simulink element, you can open the appropriate editor by right-clicking on the name of the test vector or test variable in any of the tables on the **Mappings** tab.

# Override Inport Block Signals

## Introduction

The examples in "Configuration of a Simulink Element" on page 4-5 described how to override block parameters and workspace variables. Similarly, you can override signals to root-level Inport blocks in Simulink with SystemTest data.

Because the Simulink element uses the Inport block names, not the port numbers, your test works even if you reorder the Inport blocks in the model.

Some users store signal values in a Microsoft Excel spreadsheet or `.csv` file. You can create a test vector that reads values from a spreadsheet and use that as your Inport block signal mapping. The "Override Simulink Inport Blocks Using a Spreadsheet Data Test Vector" on page 4-27 section shows such a scenario.

You can also store signal values in a MAT-file and then create a test vector that reads the values from the MAT-file. The "Map Logged Signals from a Model to Inport Blocks" on page 4-34 section shows this scenario.

## Override Inport Block Signals in a Simulink Element

To override Inport block signals:

**1**   If you have a model that contains Inport blocks and you have created a Simulink element that uses that model, click the **Mappings** tab inside the Simulink element.

**2**   Expand the **Override Inport Block Signals with SystemTest Data** section by clicking the expander arrow on the right side of the section title.

**3**   Designate your mappings.

The user interface indicates how many Inport blocks your model contains. For example, the model used in the Simulink Input example contains three Inport blocks, as shown here. You can open this example by typing the following in the MATLAB command line:

```
systemtest SimulinkInputDemo1
```

The first option, **Do not override Inport block signals**, is selected by default. That means the test will run the model without modifying any Inport block settings. Any data the Inport blocks are configured to use will be used during execution. If you want to override the model, use one of the other two options.

The **All Inport blocks are mapped using** option allows you to map data to all Inport blocks at once. Use the drop-down list to choose an existing test vector or test variable, or to create a new one. This supports any data format the Simulink model supports. For example, it could be a test vector that is an array of time and three signal values, such as [`time, U1, U2, U3`].

If you want to map individual Inport blocks, select the **Individual Inport blocks are mapped using** option.
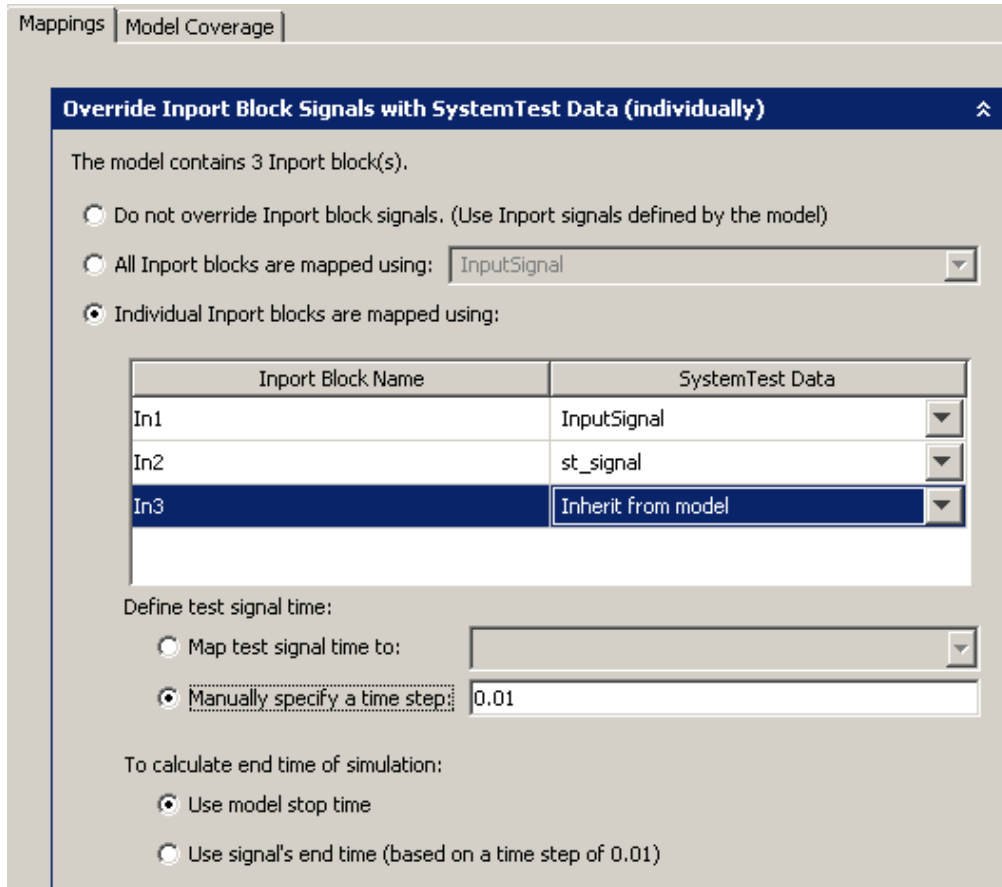


When you select this option, the mapping table becomes editable. In the case shown here, `In1` and `In2` are being overridden with SystemTest data, and `In3` is using the value in the model.

The table displays all Inport blocks contained in the model. By default, the **SystemTest Data** column is assigned as **Inherit from model**. This is especially convenient if you have a large number of Inport block signals and only want

to override a small number of them in your test. You would just change the **SystemTest Data** column value for the ones you want to override.

You can update the list of Inport blocks that are displayed in the table by clicking the **Open and update model state** button in the Simulink element. The Inports listed in the table are sortable.

---

**Note:** If you open a TEST-File and do not see the Inport blocks from your model reflected in the Simulink element, click the **Open and update model state** button:



to populate the Inport table.

---

**Note:** If you have variables in a Spreadsheet Data test vector or a MAT-File test vector, you can optionally use the Mappings Assistant. Click the **Mappings Assistant** button above the table. For more information on using the Mappings Assistant, see "Inport Block Mappings Assistant" on page 4-26.

---

**4** If you are using the map all option or individual mappings, you need to define the test signal time and the end time. If you are using the inherit from model option, skip this step since the time options do not apply to inherited mappings.

In the **Define test signal time** option, you can specify the simulation time signal to provide to the model. To specify the time signal using a test vector or test variable, select **Map test signal time to**. To specify a time signal based on a desired simulation time step, select **Manually specify a time step** and then enter a valid time step, which must be a positive number.

In the **To calculate end time of simulation** option, either use the model's stop time, or use the signal's end time based on the time step you specified. The **Use model stop time** option stops the simulation of the model at the end time configured in the model. The **Use signal's end time** option stops the simulation of the model at the end of the test signal, temporarily overriding the end time of the model with the test signal end time.

> **Note:** The **Define test signal time** option and the **To calculate end time of simulation** option are disabled if all individual Inport mappings are set to inherit from the model.

## Inport Block Mappings Assistant

If you are overriding Inport block signals, as shown in "Override Inport Block Signals in a Simulink Element" on page 4-22, then you can optionally use the Mappings Assistant when you use the **Individual Inport blocks are mapped using** option.

1 When overriding Inport block signals, select the **Individual Inport blocks are mapped using** option.

2 Click the **Mappings Assistant** button above the table. This button is only available when you are configuring individual mappings.

3 In the Inport Block Mappings Assistant dialog box, choose your mapping in the **Override each Inport block using** section.

   • If you are using a Spreadsheet Data test vector, select the **A spreadsheet test vectors headers** option. Then in the drop-down list, select an existing Spreadsheet Data test vector, or create a new one.

   • If you are using a MAT-File test vector, select the **A MAT-File test vectors selected variables** option. Then in the drop-down list, select an existing MAT-File test vector, or create a new one.

   • If you have a test variable whose name matches the Inport block, select the **An existing test variable with a matching name** option.

4 The **If an Inport block name cannot be matched** section determines what happens in the case of one or more variables in the selected test vector not matching an Inport block name. Select the option you want the Simulink element to perform. **Inherit from Model** is the default.

5 The **Summary** section displays information on how many root-level Inport blocks are found in the model, how many are mapped to the test vector you selected if you are mapping from a test vector, and how may will use the option you chose in step 4 in the case of non-matches.

   When you are finished configuring the mappings and viewing the summary, click **OK** to create the mappings.

The mappings are then displayed in the table.

## Override Simulink Inport Blocks Using a Spreadsheet Data Test Vector

In this example, a Simulink element is being used to test a model of a fuel rate controller. To see the test and the model, open the example by typing the following at the MATLAB command line:

```
systemtest('demosystest_fuelctrlsldv.test');
```

The model has four Inport blocks that represent throttle angle, engine speed, exhaust gas, and manifold pressure.
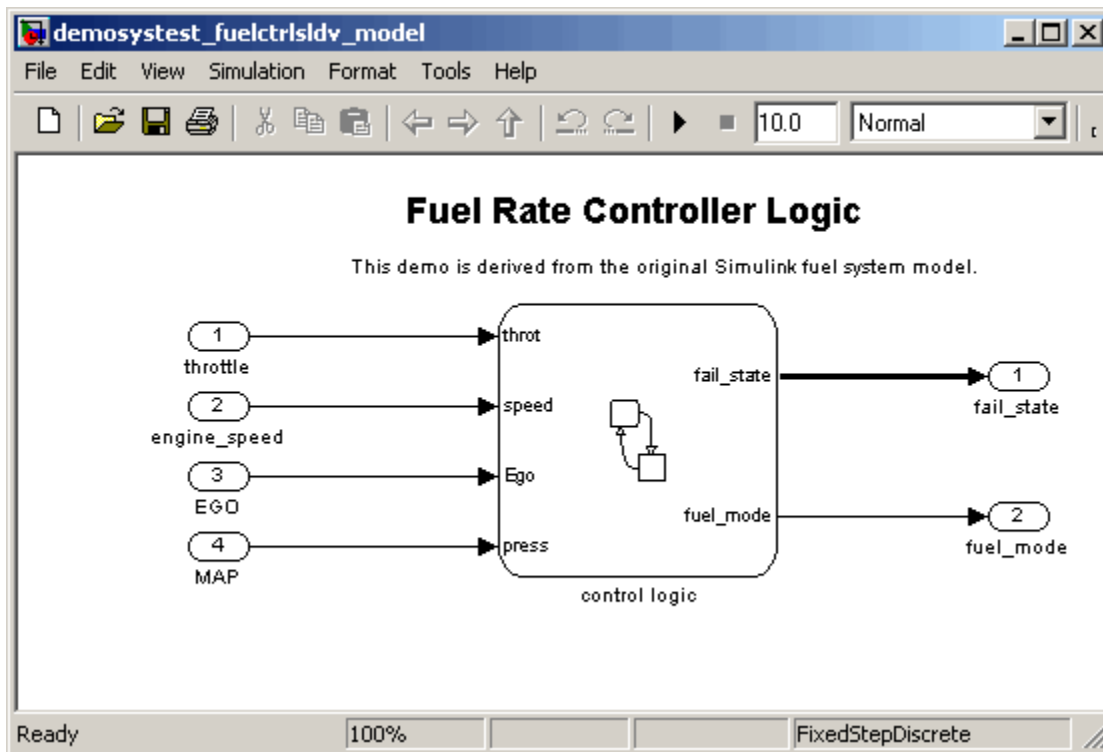


The tester has values for these four blocks in a Microsoft Excel spreadsheet. It contains 37 sets of generated values for the blocks. Each set of values is on a different sheet within

the spreadsheet, representing a testing scenario for the model. One of the sheets is shown here.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | Time | throttle | engine spe | EGO | MAP |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0.01 | 4 | 0 | 1.4 | 0.002 |
| 4 | 0.02 | 4 | 629 | 1.4 | 0.002 |
| 5 | 0.03 | 91 | 0 | 0 | 1 |
| 6 | 0.04 | 3 | 0 | 0 | 2 |
| 7 | 0.05 | 4 | 1 | 0 | 0.002 |
| 8 | 0.06 | 4 | 1 | 1.4 | 1 |
| 9 | 0.07 | 90 | 0 | 1.4 | 0.002 |
| 10 | 0.08 | 90 | 0 | 1.4 | 0.002 |

Column A represents the simulation time signal. Columns B through E represent test data for the four Inports in the model. Each of the 37 sheets is set up the same way but contains different values.

To set up the test vector that reads the data from the spreadsheet:

1 Create the test vector by clicking the **New** button in the **Test Vectors** pane.

2 In the Insert Test Vector dialog box, select **Spreadsheet Data** as the vector type.

3 On the **General** tab, name the test vector `InputSignal`.

4 Click the **Add File** button and browse to the Microsoft Excel spreadsheet.
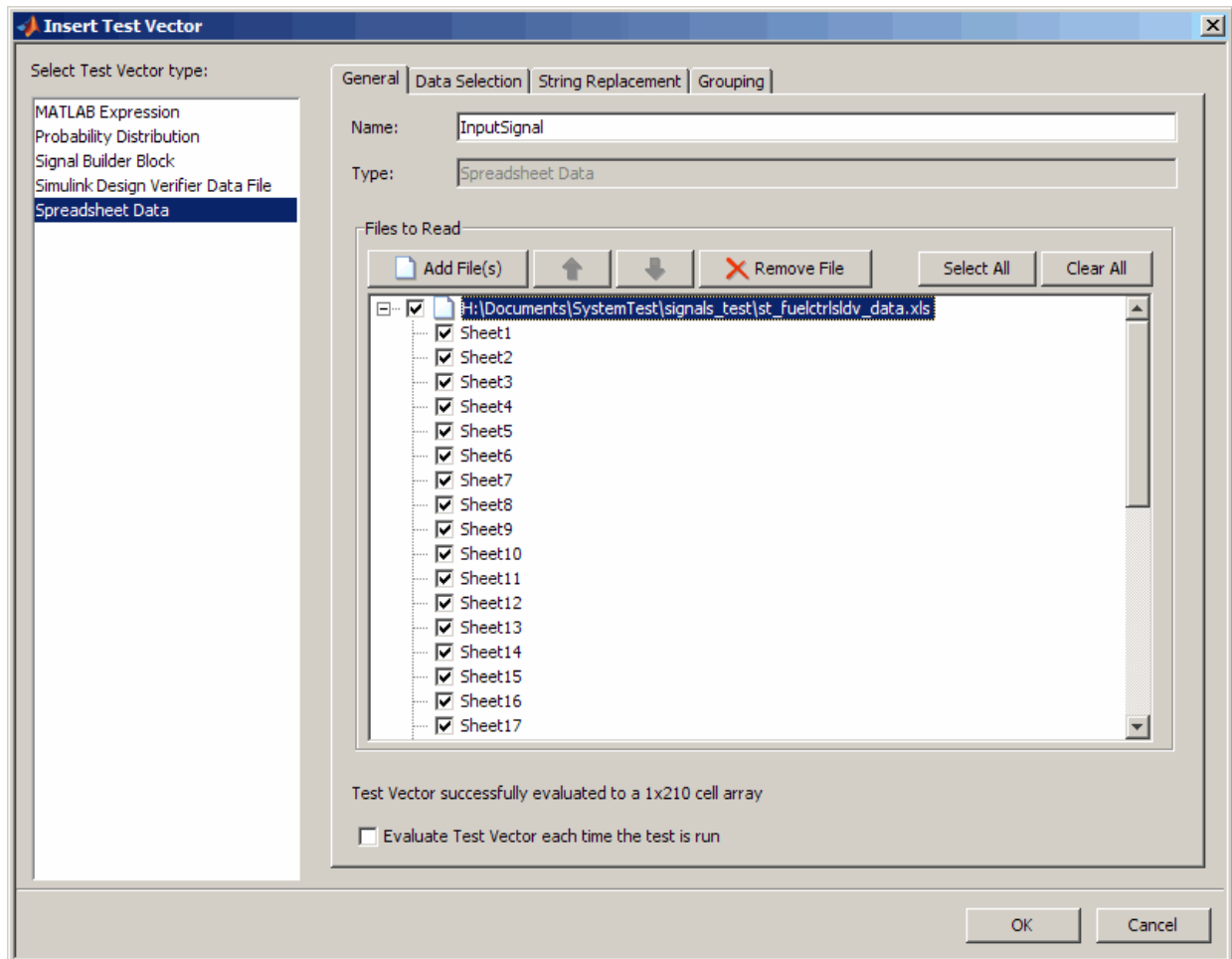
5   Click the **Select All** button to select all sheets in the spreadsheet file.

6   On the **Data Selection** tab, keep the default of **column** in the **Data is arranged by** option.

7   In the **Read data from column** option, enter A to E, starting at row 1.

8   Select the **First row is a header** option, since you can see in the above figure of the spreadsheet that row 1 of the file contains text labels.

9   Select the **Treat each selected sheet as a test vector value** option.

The configured test vector appears as follows.
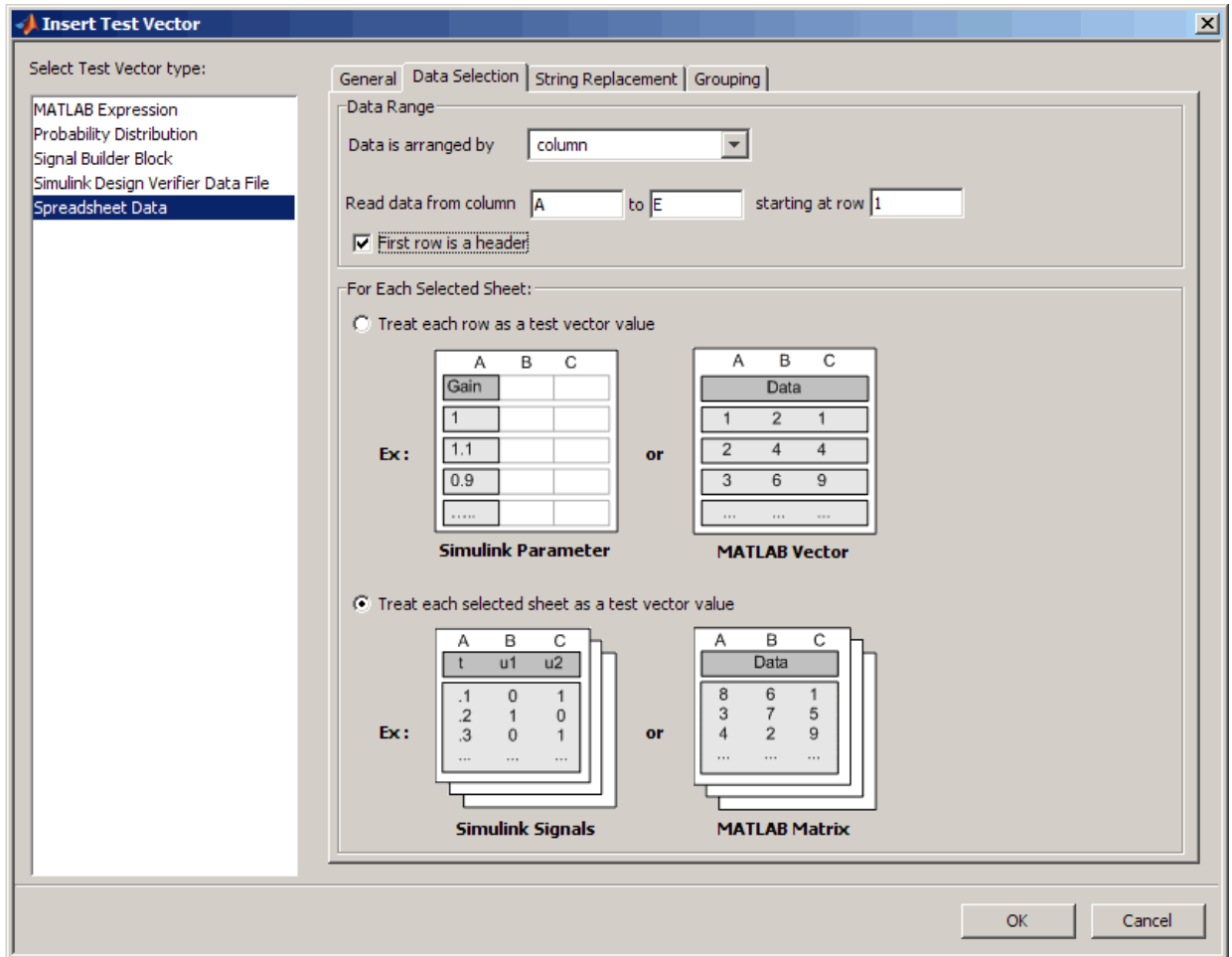


**10** Click **OK** in the Insert Test Vector dialog box.

The new vector appears in the table in the **Test Vectors** pane. You can see that the length is 37 because there are 37 sheets in the spreadsheet file and each sheet is being treated as one value in the vector.
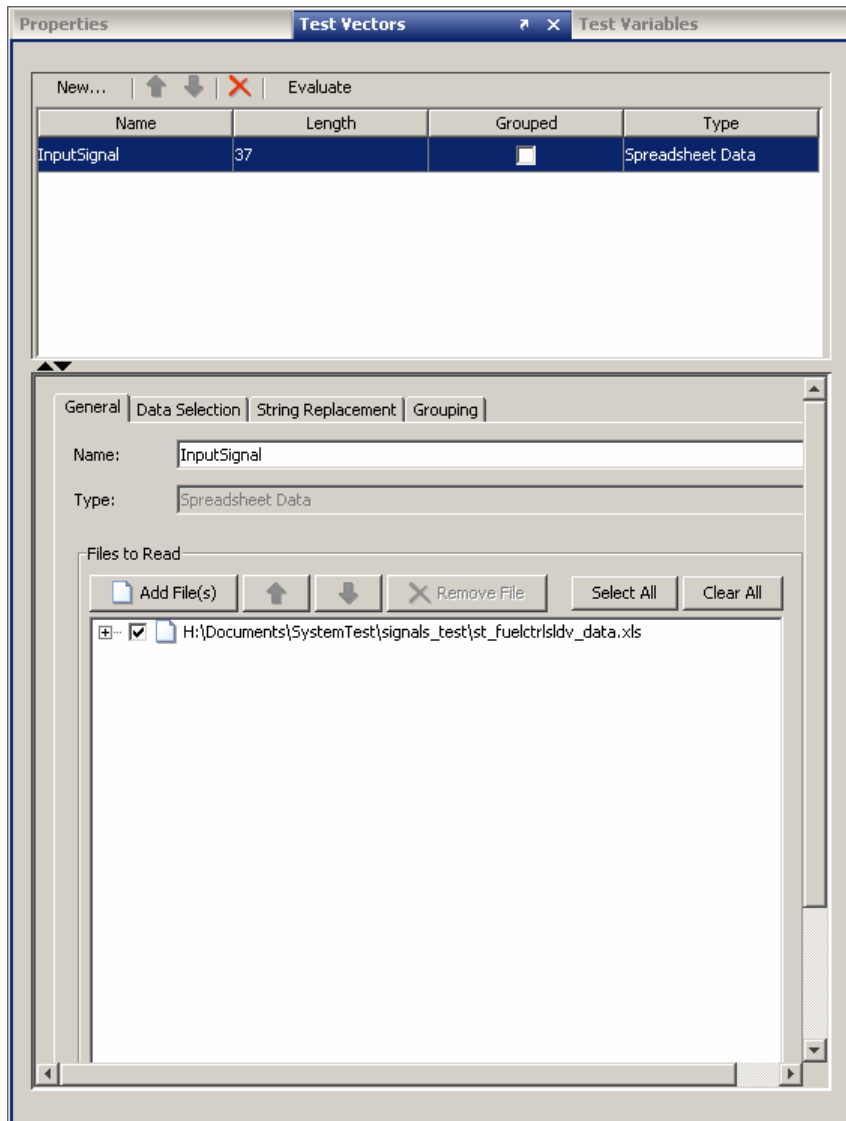
Now that the test vector is set up, you can set up the Simulink element to override the Inport blocks using the test vector values from the underlying spreadsheet file.

1 Create a Simulink element by clicking **New > Test Element > Simulink** button in the **Test Browser**.

2 Click the browse button to locate the Fuel Rate Controller model.

3 On the **Mappings** tab, expand the **Override Inport Block Signals with SystemTest Data** section if it is not open.

4 Select the **Individual Inport blocks are mapped using** option. The four Inport blocks appear in the table.

5 For each Inport block, use the drop-down list in the **SystemTest Data** column to override the Inport block with the appropriate data in the test vector that was created earlier.

   For example, for throttle, click the drop-down list, expand the `InputSignal` test vector entry, and select `throttle`. Do the same for the other three signals.

   The entries under the `InputSignal` test vector represent the underlying columns in the spreadsheet. Since the Spreadsheet Data test vector called `InputSignal` was created using the columns and the headers, the columns appear named with their headers in the list for easy identification, for example, `InputSignal(throttle)`.

6 In the **Define test signal time** option, select **Map test signal time to** and choose `InputSignal(Time)`.

   `Time` is the first column in the spreadsheet and contains the simulation time signal for the model. The test will use these time step values when the Simulink element is executed.

7 Select the **Use signal's end time** option, so that the end times provided in the spreadsheet are used.

   The configured Simulink element appears as follows.
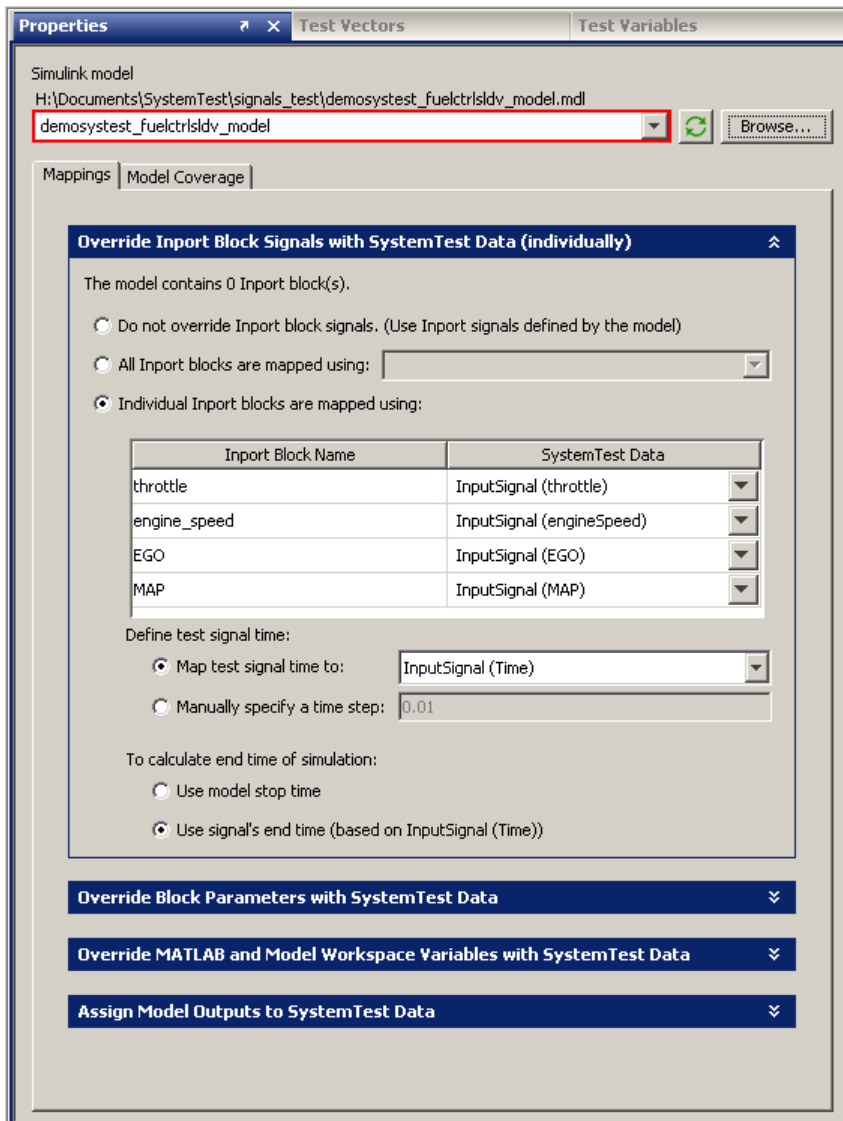
When the test is executed, the Simulink element will test the model using the Inport
block signals mapped from the spreadsheet.

## Map Logged Signals from a Model to Inport Blocks

You can map logged signals from a Simulink model, including bus signals, to Inport blocks of a model using the Simulink element and a test vector or test variable that contains the logged signal data.

A common usage scenario is to log the signals while running a model and store them in a MAT-file. Then you can acquire them from the Mat-file using a MAT-File test vector and map that data to Inport blocks in the Simulink element. The following high-level steps outline this usage scenario.

1   Do one of the following:

    Run your model that contains signals. The signals are logged as variables in the MATLAB workspace. Save the variables as a MAT-file.

    OR

    Alternatively, use MAT-file(s) that have already been created and saved.

2   In the SystemTest software, create a MAT-File test vector using the MAT-file that your signals are saved in. See "MAT-File Test Vectors" on page 2-13 for more information on MAT-File test vectors.

3   Add a Simulink element to your test, and select the model that you want to test.

4   In the **Override Inport Block Signals with SystemTest Data** section, select the **All Inport blocks are mapped** option.

5   From the drop-down list, select the MAT-File test vector you created, and drill down into the variable that represents the signal you want to use. This is how you map the logged signals to the Inport blocks.

    Note that the logged signal(s) and the Inport block(s) must have exactly the *same* name(s) for the Simulink element to simulate the model successfully. See the usage notes below.

6   Run the test.

**Important Usage Notes**

– The logged signal(s) and Inport block(s) must have identical names that match exactly. To use this feature, each Inport block name must match a corresponding signal name within the logged data. If it does not, you can rename the Inport block or the signal in the logged data to avoid an error. You can have other signals in the logged data, but each Inport block must exactly match a signal name.

– The following data types are supported:

- Simulink.Timeseries.
- Simulink.TsArrays.
- Simulink.SubsysDataLogs.

## Edit a Test Vector or Test Variable from within the Element

If you want to edit a test vector or test variable while working in the Simulink element, you can open the appropriate editor by right-clicking on the name of the test vector or test variable in any of the tables on the **Mappings** tab.

# Simulink Model Coverage

The model coverage feature provided by the Simulink Verification and Validation software allows you to control the generation of coverage metrics for a Simulink model from within your SystemTest test. Model coverage metrics allow you to validate your model by identifying unexecuted subsystems, unselected switch positions, or untaken conditional transition paths. You can generate a cumulative coverage report, specify individual coverage options, or inherit a model's coverage settings.

---

**Note:** To use the model coverage feature, you need a license for Simulink Verification and Validation.

---

The following basic steps describe the typical work flow to use this feature:

1   Use an existing Simulink element or add one by clicking the **New > Test Element** button and selecting **Simulink**.

2   On the **Properties** pane, browse for your Simulink model using the browse button next to the **Simulink model** field.

   To see an example, you can run the Signal Builder example by typing `systemtest demosystest_sigbldr` in MATLAB.

3   Configure the Simulink element as described in this chapter, using the **Mappings** tab of the **Properties** pane to define model overrides and map Simulink data to test variables.

4   On the **Model Coverage** tab, which appears if you have a license for the Simulink Verification and Validation software, select the **Enable model coverage** check box. The following figure shows the Signal Builder example.
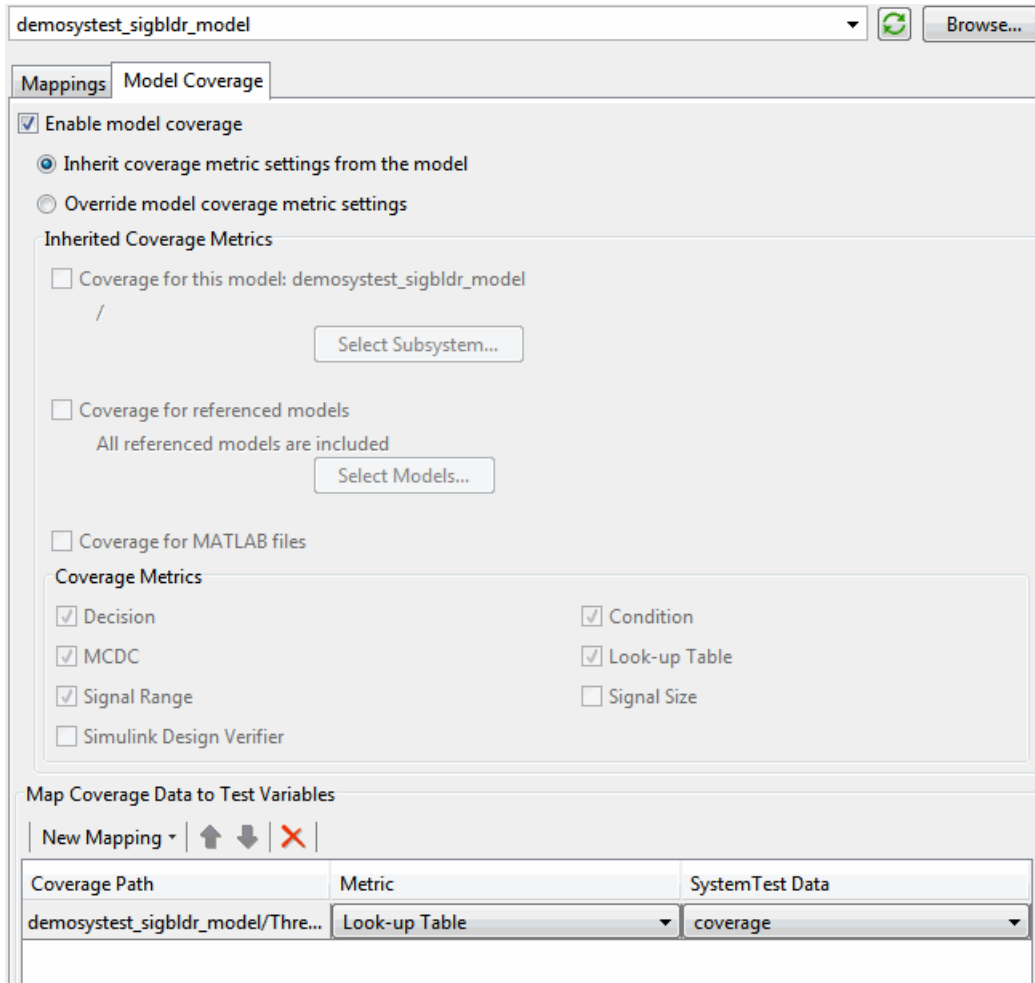
**5** If you want to use the model coverage settings you already have on the Simulink model, select the **Inherit coverage metric settings from the model** option. Then go to step 12.

When you use this option, if the settings on the model change, the inherited settings will also change.

**6** If you want to override the existing settings, select the **Override model coverage metric settings** option.

These settings are independent of the model.

**7** Select **Coverage for this model: <modelname>**.
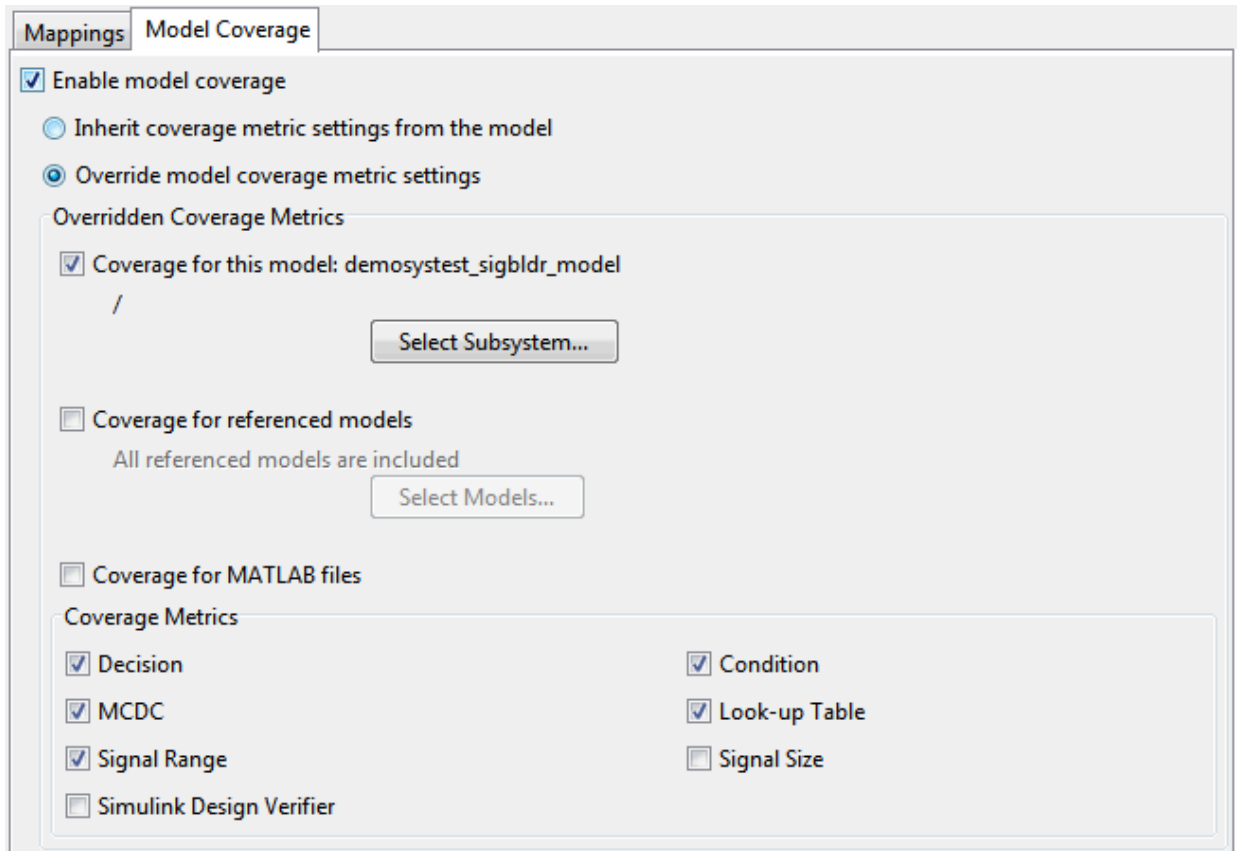


**8** Click the **Select Subsystem** button in the **Overridden Coverage Metrics** section to specify the root model of your coverage data. Make your selection in the System Selector dialog box and click **OK**.
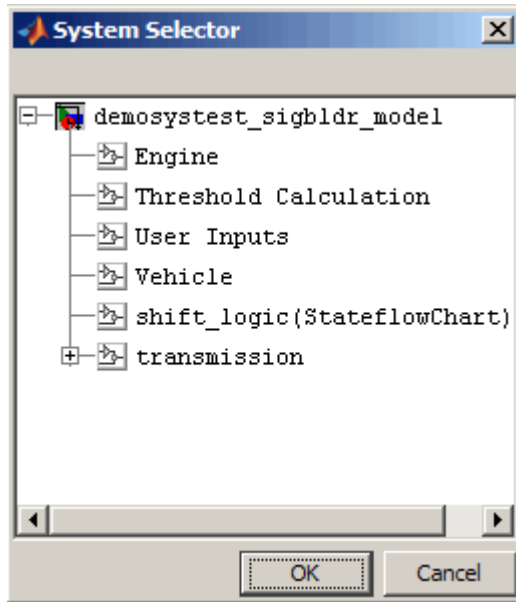
**9**   If you have one or more referenced models and you want to gather coverage for them, select the **Coverage for referenced models** option.

Then click the **Select Models** button to select the referenced model(s) for coverage. Make your selection in the Select Models for Coverage Analysis dialog box and click **OK**.

Note that you can record coverage only for referenced models that operate in Normal mode. You cannot enable coverage for referenced models operating in Accelerated mode.

**10**  If you have MATLAB functions in your model that you want to test, select the **Coverage for MATLAB files** option. This enables coverage for MATLAB functions called from your model.

**11**  In the **Coverage Metrics** area, select the metrics you require. The selected metrics will be generated and shown in the coverage report.
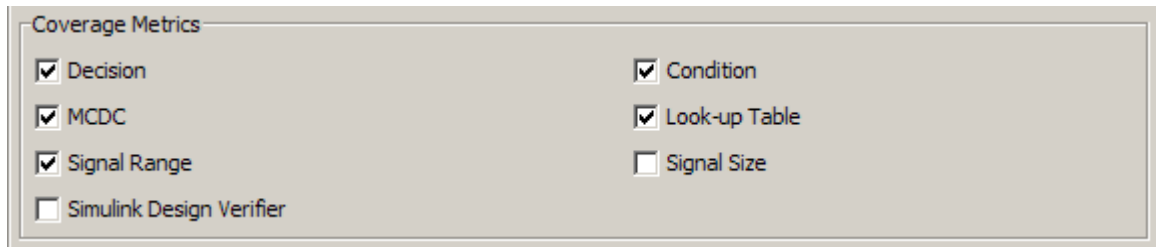
Summary of coverage metrics:

**Decision** — analyzes elements that represent decision points in a model, such as a Switch block or Stateflow states.

**Condition** — analyzes blocks that output the logical combination of their inputs, such as the Logical Operator block, and Stateflow transitions.

**MCDC** — modified condition/decision coverage analysis extends the decision and condition coverage capabilities. It analyzes blocks that output the logical combination of their inputs and Stateflow transitions to determine the extent to which the test case tests the independence of logical block inputs and transition conditions.

**Lookup Table** — examines blocks, such as the n-D Lookup Table block, that output information from inputs in a table of inputs and outputs, interpolating between or extrapolating from table entries. Lookup table coverage records the frequency that table lookups use each interpolation interval.

**Signal Range** — records the minimum and maximum signal values at each block in the model, as measured during simulation. Only blocks with output signals receive signal range coverage.

**Signal Size** — records the minimum, maximum, and allocated size for all variable-size signals in a model.

**Simulink Design Verifier** — collects model coverage data for some Simulink Design Verifier blocks.

12 Use the **Map Coverage Data to SystemTest Variables** field to map coverage metrics to test variables. Click **New Mapping** and select **Full Coverage Instrumentation Path** if you want coverage data below the root you specified

under **Coverage for this model**, or select **Select Path to Map** if you want to pick an alternate coverage path, which must be within the coverage instrumentation path. If you select the latter, your Simulink model will open and you can select a block to specify an alternate root for your coverage path.

**13** Select the **Metric** you want to map to a test variable, and specify the test variable to use under the **SystemTest Data** column.
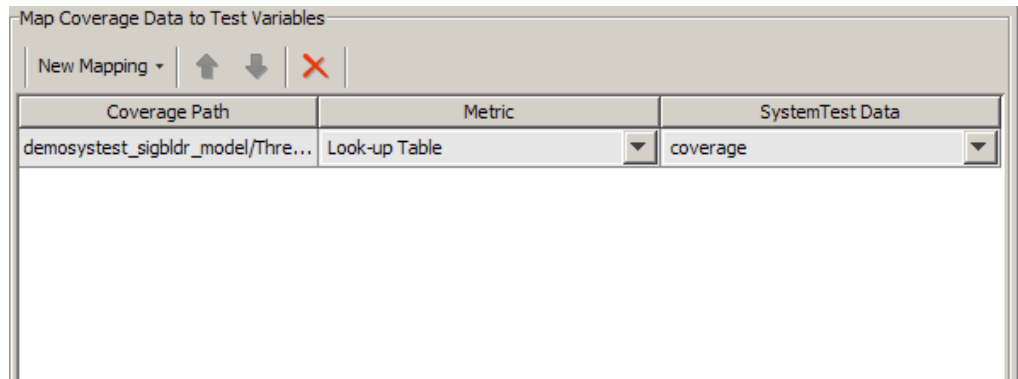


Note that if you select **<New Test Variable>** in the **SystemTest Data** column, the default name will match the input. For example, if the model name is systemtestpendulum from the **Coverage Path** column, and the metric you select in the **Metric** column is MCDC Coverage then the default name for the new variable would be systemtestpendulum_mcdcinfo. The logical default name reflects the name of the model and the metric. You can use the default name or change it.

**14** Run your test.

**15** View the coverage report by clicking the link in the **Run Status** pane.
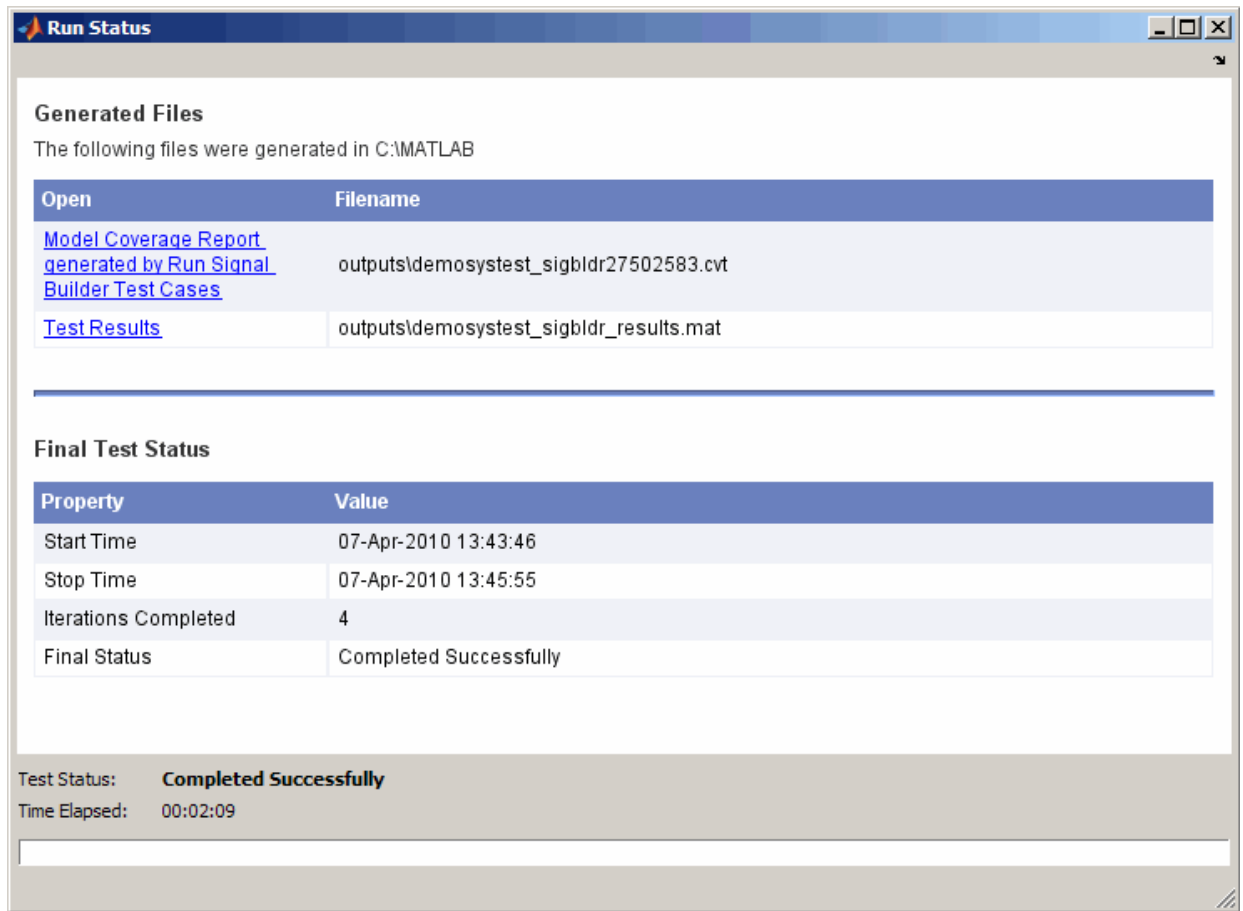
For more information on the model coverage feature, including details about the coverage metrics, see "Model Coverage Analysis".
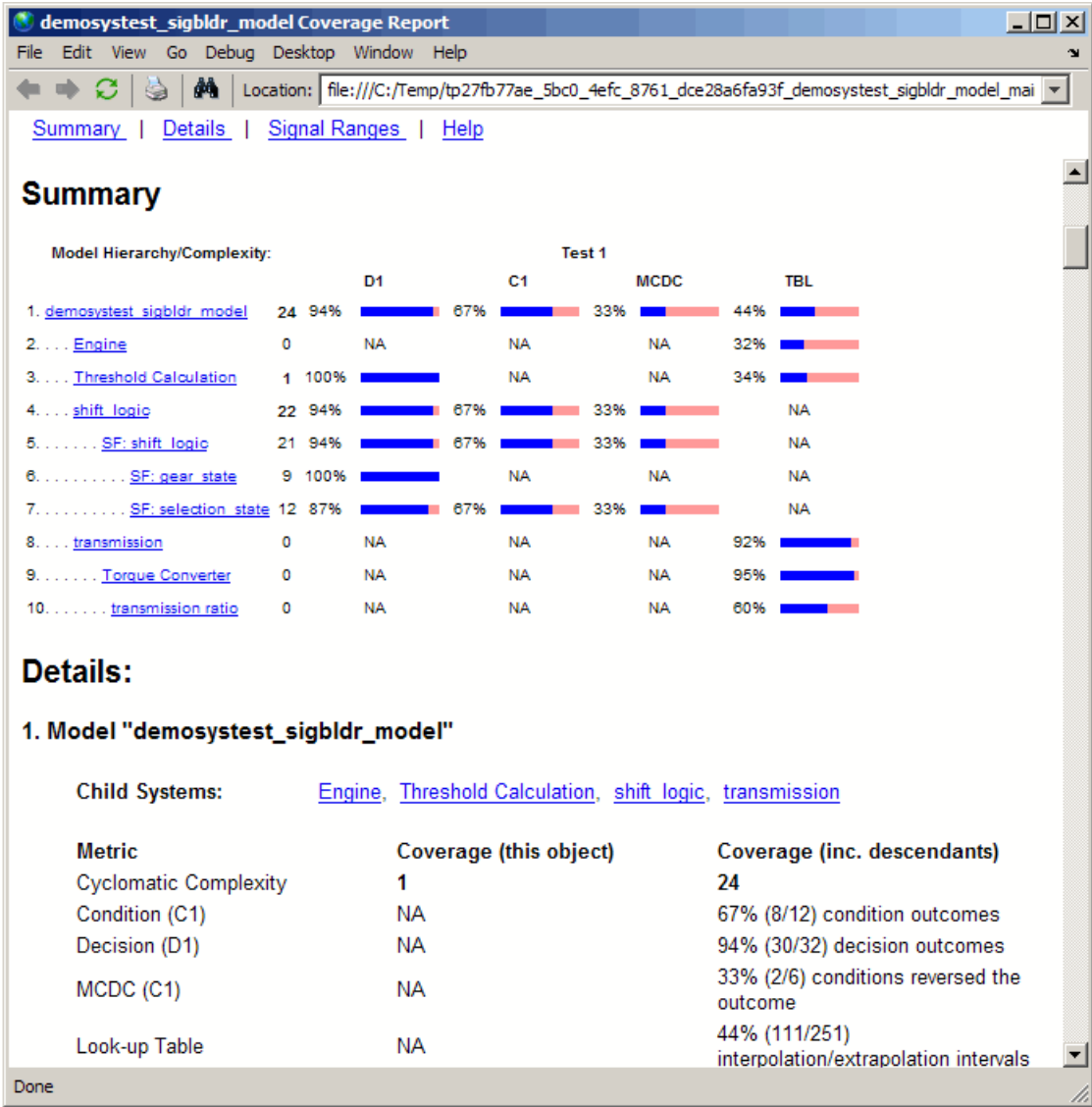
# Use Simulink Design Verifier Test Cases

The Simulink Design Verifier Data File test vector can read test cases created by Simulink Design Verifier. To use this test vector, you must have a Simulink Design Verifier data file with test cases.

You first create a Simulink Design Verifier test harness and set the generate SystemTest harness option in the Configuration Parameters in Simulink. Then you can do one of two things:

· Generate a SystemTest harness for the model from Simulink. When it completes, a new test opens automatically in SystemTest and a Simulink Design Verifier Data File test vector is automatically created for you. A Simulink element is also automatically created, with links to the model, override mappings set, and model coverage enabled if your model uses that feature. For more information, see "Create SystemTest Harness from Simulink Design Verifier".

· If you already have a data file from Simulink Design Verifier, you can create a test vector in SystemTest that uses the data, and create a Simulink element and configure overrides in it. For more information, see "Create a Simulink Design Verifier Test Vector".

# Use Signal Builder Block Test Cases

If you use a Signal Builder block in a Simulink model, you can use the test cases in a SystemTest test.

The most common workflow for this feature is to create a Simulink element using the model containing the Signal Builder block, and create a Signal Builder Block test vector from within the element. For an example of this procedure, see "Create Signal Builder Block Test Vectors".

# Test Cases and Signals in Simulink Element

You can create signals in the SystemTest software and use them to test a Simulink model. The Test Case Editor provides a graphical way of creating, editing, and visualizing signal data in SystemTest. You can then map signals in the Simulink element using a Test Case Data test vector.

Here is an example of one possible high-level workflow of using test cases and signals in your test via the Simulink element. You will create a Test Case Data test vector, set up signals in the Test Case Editor, and then map Inport blocks to the signals in a Simulink element.

1  In the **Test Vectors** pane, click the **New** button.

2  Select **Test Case Data** as the test vector type. Click **OK** to create it.

   For more information, see "Create a Test Case Data Test Vector"

3  In the **Test Vectors** pane, select the Test Case Data test vector you just created.

4  Click the **Open Test Case Editor** button to open the tool.

5  In the Test Case Editor, add one or more test cases using the **Add Test Case** button.

   For more information, see "Create Test Cases".

6  Select a test case and add one or more signals to it using the **Add Signal** button. If you plan to map these signals to Inport blocks in your model, you could create the signals with the same names as the blocks. They are not required to be the same name, but making them the same name allows the Mapping Assistant to work (in a later step).

   See "Add Signals to Test Cases" on page 5-17 for more information on this step.

7  For each signal, append the desired segment or segments to create the signal to use, and configure its attributes as needed.

   See "Add Signals to Test Cases" on page 5-17 for more information on this step.

8  Once you have created and edited the test case or cases and signals that you need, close the Test Case Editor by clicking the **OK** button at the bottom of the window. When you close the tool, the SystemTest software saves the data in the Test Case Data test vector.

9  Return to the SystemTest desktop. You can now use the Test Case Data test vector and the signals it contains in your test, via the Simulink element, the Limit Check element, and the General Plot element.

10 Use an existing Simulink element or add one by clicking the **New > Test Element** button and selecting **Simulink**.

11 On the **Properties** pane, browse for your Simulink model using the **Browse** button next to the **Simulink model** field.

Suppose you are using the following model, which has four Inport blocks that represent throttle angle, engine speed, exhaust gas, and manifold pressure of an automobile fuel controller.
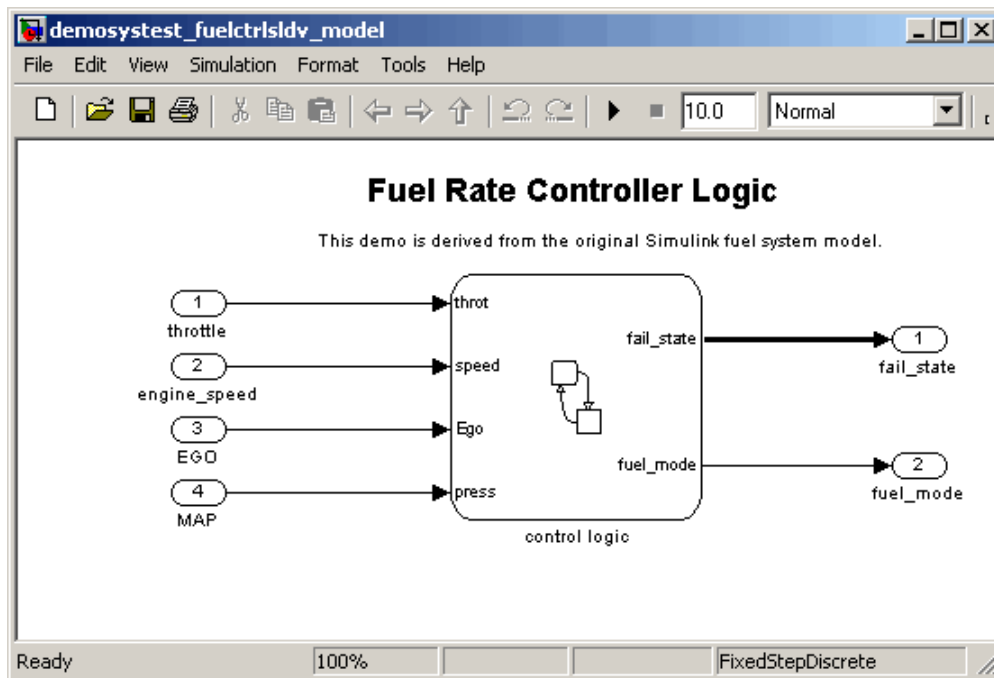


12 In this case, you want to map signals you created in the Test Case Editor to these four Inports in your model. Use the **Override Inport Block Signals with SystemTest Data** section of the Simulink element.

You must use the **Individual Inport blocks are mapped using** option. Note that you cannot use the **All Inport blocks are mapped using** option. You can map the individual Inports by selecting the signal under the expanded test vector that matches the Inport block in each row of the table, as shown here.
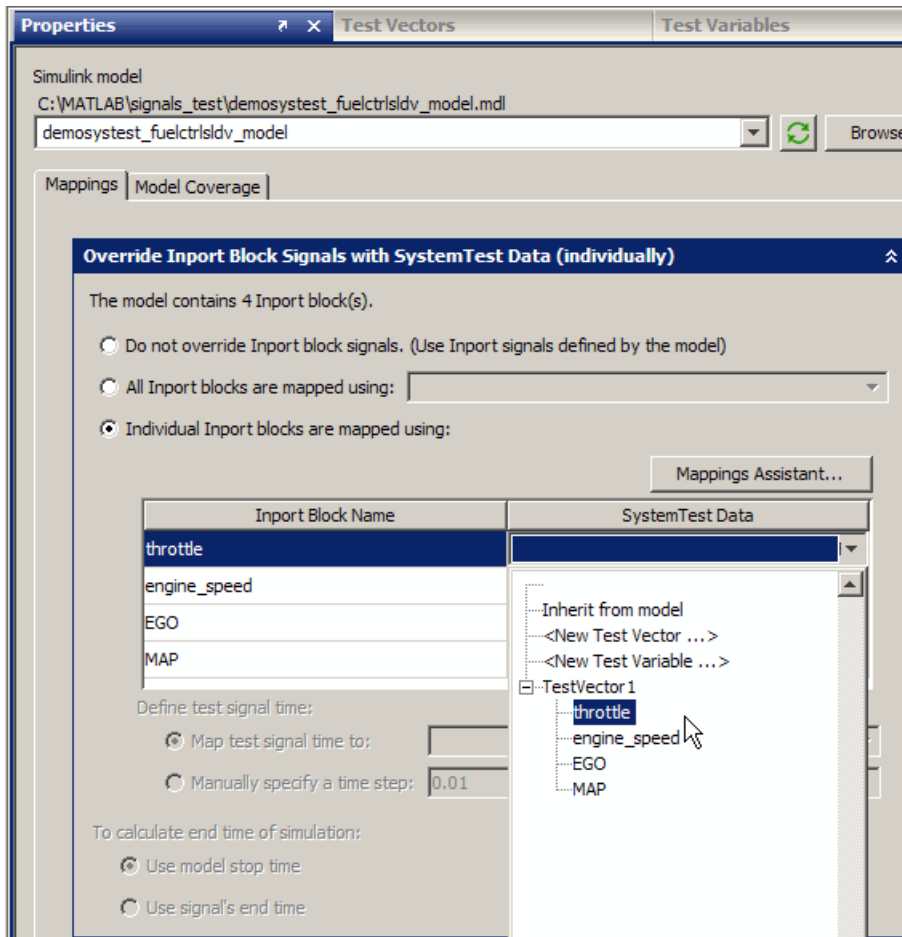
You could also use the Mappings Assistant (click the **Mappings Assistant** button to open it) and select them all at once by selecting the Test Case Data test vector in the **A test case data test vector's signals** override option, as shown next.
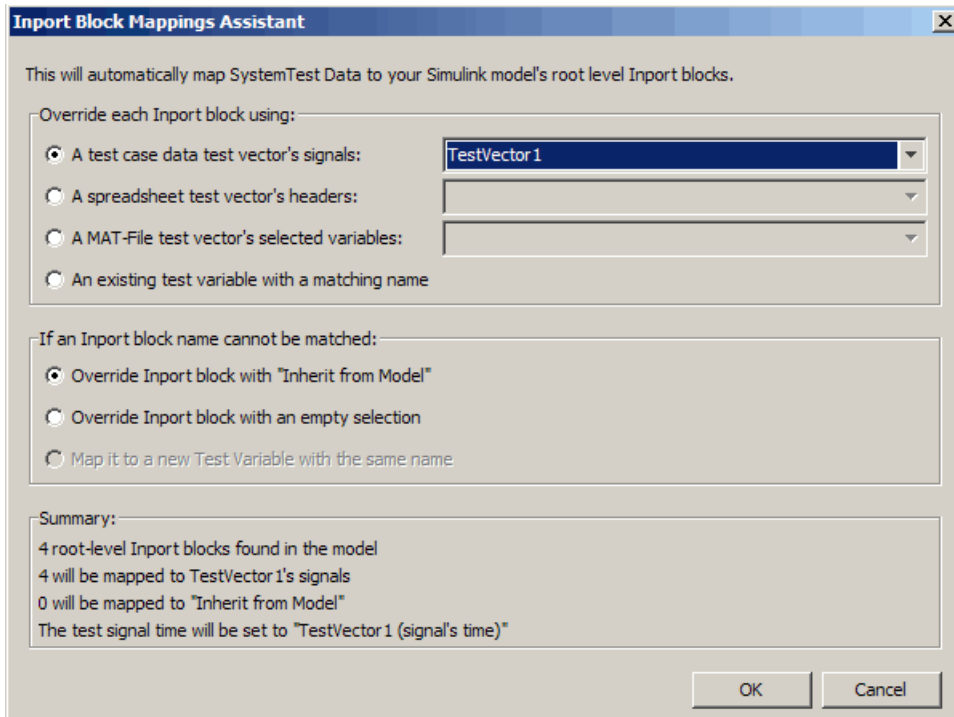
When you click **OK** in the Mappings Assistant, the signals are entered into the **SystemTest Data** column in the table, and the test uses the signals' time by default.
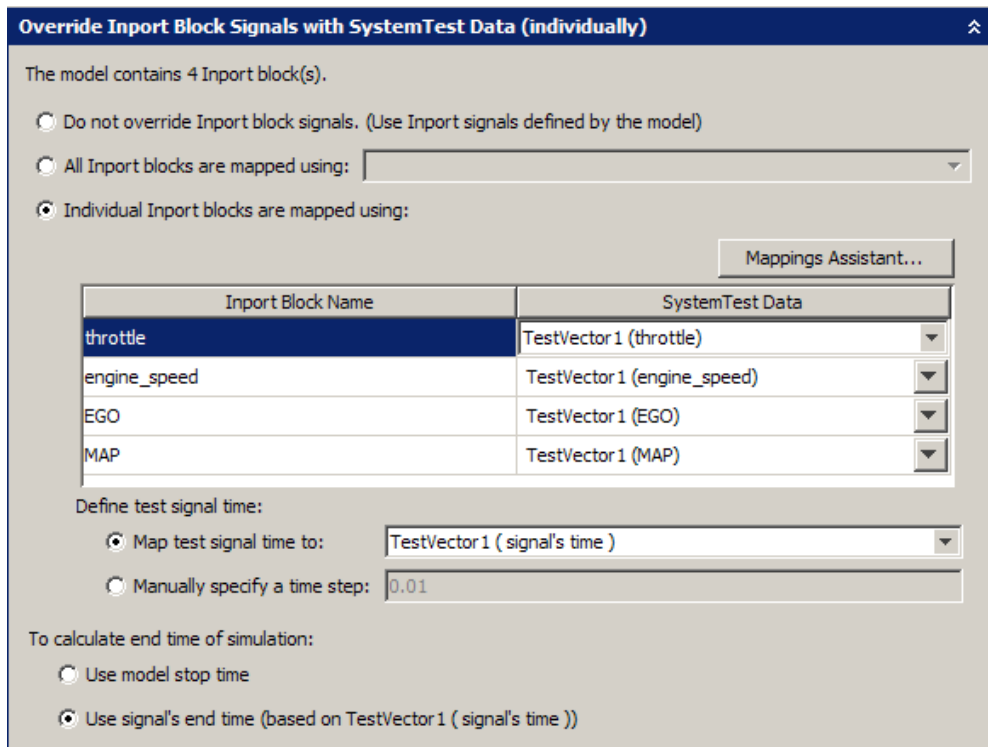
13    Set up any other elements for the test.

14    Run the test.

---

**Note:** When using a Test Case Data test vector to simulate your model as described in this example, the **Interpolate data** option in the Simulink Block Parameters of your model's Inport blocks will be turned on. This allows data coming from signals containing segment types like Ramp and Custom to accurately reflect their value when sampled.

After the test runs, the **Interpolate data** option will be restored in your model.

---

**5**

# Author Signals in the Test Case Editor

# Signal Authoring for Test Data

You can create signals in the SystemTest software and use them to test a Simulink model. The Test Case Editor provides a graphical way of creating, editing, and visualizing signal data in SystemTest.

Use this tool to create signals based on commonly used patterns and to specify values for attributes of those signal segments. You can easily create the following types of signals:

- Constant
- Step
- Ramp
- Pulse
- Square
- Sine
- Custom

The tool also allows you to view and manage buses and signals, and to organize them into test cases. You can manage a large number of test cases and signals.

The Test Case Editor is accessed through the Test Case Data test vector in the SystemTest software.

## Definitions

The following definitions apply to creating and editing signals in the Test Case Editor.

- **Segment** — A common signal pattern providing property configurations specific to its type. This individual portion of a signal is used as a building block for constructing more complex signals.

  See "Add Signals to Test Cases" on page 5-17.

- **Signal** — An array of time-based data used for testing a Simulink model. Created from one or more segments.

  See "Add Signals to Test Cases" on page 5-17 and "Signal Types" on page 5-29.

- **Test Case** — Created in the Test Case Editor, a collection of one or more signals that can be treated as a set of inputs to a Simulink model.

See "Create Test Cases" on page 5-12.

- **Test Case Data test vector** — Type of test vector in the SystemTest software used to manage a 1xN array of test cases. Also the way to open the Test Case Editor from the SystemTest software.

  See "Create a Test Case Data Test Vector" on page 5-6.

- **Edit view** — View in the Test Case Editor that shows plots of the selected signals in a test case. Where you build and edit signals.

  See "Edit View" on page 5-9.

- **Test Case view** — View in the Test Case Editor that shows a graphical representation of each signal. Allows an easy way to find and manage signals when you have many signals in a test case.

  See "Test Case View" on page 5-11.

- **Test Case Options** — Accessed by right-clicking a test case, options that govern the length of test cases and place to add a description.

  See "Test Case Options" on page 5-16.

- **Signal Properties** — Properties, such as extrapolation policy and data type, set on an entire signal in the **Signal Properties** area of the Edit view, and place to name a signal.

  See "Add Signals to Test Cases" on page 5-17.

- **Segment Properties** — Properties set on an individual signal segment in the **Segment Properties** area of the Edit view. Place to define duration and values of the segment.

  See "Add Signals to Test Cases" on page 5-17.

# Author and Use Signals in Tests

This section describes the high-level workflow of authoring and using signals in tests.

Note that you would often create a Simulink element first and can create the test vector from the element, but in this workflow example you start by creating a new test vector.

1   On the **Test Vectors** pane of SystemTest software, click the **New** button.

2   Select **Test Case Data** as the test vector type. Click **OK** to create it.

    See "Create a Test Case Data Test Vector" on page 5-6 for more information on this step.

3   On the **Test Vectors** pane, select the Test Case Data test vector you just created.

4   Click the **Open Test Case Editor** button to open the tool.

5   In the Test Case Editor, add one or more test cases using the **Add Test Case** button.

    See "Create Test Cases" on page 5-12 for more information on this step.

6   Select a test case and add one or more signals to it using the **Add Signal** button.

    See "Add Signals to Test Cases" on page 5-17 for more information on this step.

7   For each signal, append the desired segment(s) to create the signal you'd like to use.

    See "Add Signals to Test Cases" on page 5-17 for more information on this step.

8   For each segment, configure its attributes as needed.

    See "Add Signals to Test Cases" on page 5-17 for more information on this step.

9   Once you have created and edited the test cases and signals that you need, close the Test Case Editor by clicking the x button in the banner or the **OK** button at the bottom of the window. When you close the tool, the SystemTest software saves the data in the Test Case Data test vector.

10  Return to the SystemTest desktop. You can now use Test Case Data test vector and the signals it contains in your test, via the Simulink element, the Limit Check element, and the General Plot element.

**Note:** For an example of using signals created in the Test Case Editor in a Simulink element, see "Test Cases and Signals in Simulink Element" on page 4-46.

**Note:** You can access the signal data from a Test Case Data test vector by using a MATLAB element in your test. For an example of this, see "Access Test Case Data Using MATLAB Element" on page 2-69.

# Create a Test Case Data Test Vector

As described in "Author and Use Signals in Tests" on page 5-4, you create a Test Case Data test vector from the SystemTest software, and then add signals to it using the Test Case Editor. This example starts from the **Test Vectors** pane.

To create the Test Case Data test vector:

**1** On the **Test Vectors** pane of SystemTest software, click the **New** button.

**2** In the Insert New Test Vector dialog box, select **Test Case Data** as the test vector type.



**3** Assign a name to the vector in the **Name** field.

**4** Click **OK** in the Insert Test Vector dialog box.

**5** On the **Test Vectors** pane, select the test vector you just created, and click the **Open Test Case Editor** button to create the test cases and signals, as described in "Author and Use Signals in Tests" on page 5-4.

**6**  Alternatively, you can click the **Open Test Case Editor** button after step 3, while creating the test vector. If you do that, click **OK** in the Insert Test Vector dialog box once you return to the SystemTest desktop.

Whether you create the test cases and signals during creation of the test vector, or after you have created it, see "Create Test Cases, Signals, and Buses" on page 5-9 for information on creating and editing the test cases and signals.

**Note:** For an example of using signals created in the Test Case Editor in a Simulink element, see "Test Cases and Signals in Simulink Element" on page 4-46.

**Note:** You can access the signal data from a Test Case Data test vector by using a MATLAB element in your test. For an example of this, see "Access Test Case Data Using MATLAB Element" on page 2-69.

# Create Test Cases, Signals, and Buses

| In this section... |
| --- |
| |
| |
| |
| |
| |

## Navigate in the Test Case Editor

The Test Case Editor has two views you can work in. By default it opens in the Edit view. You can switch between the two views using the **Test Case View** and **Edit View** buttons on the top right of the tool. In the following diagram, the **Edit View** button is selected.
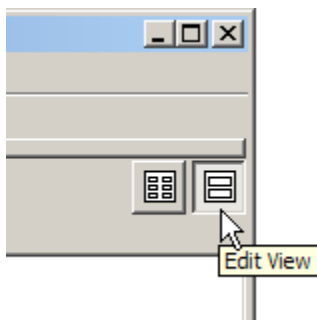
You edit and work with signals in the Edit view. The Test Case view offers easy navigation when you have many signals in a test case.

### Edit View

The Edit view shows plots of the selected signals in a test case. Select a test case in the **Test Case** list to see its signals in the Edit view. Only one test case at a time can be selected.
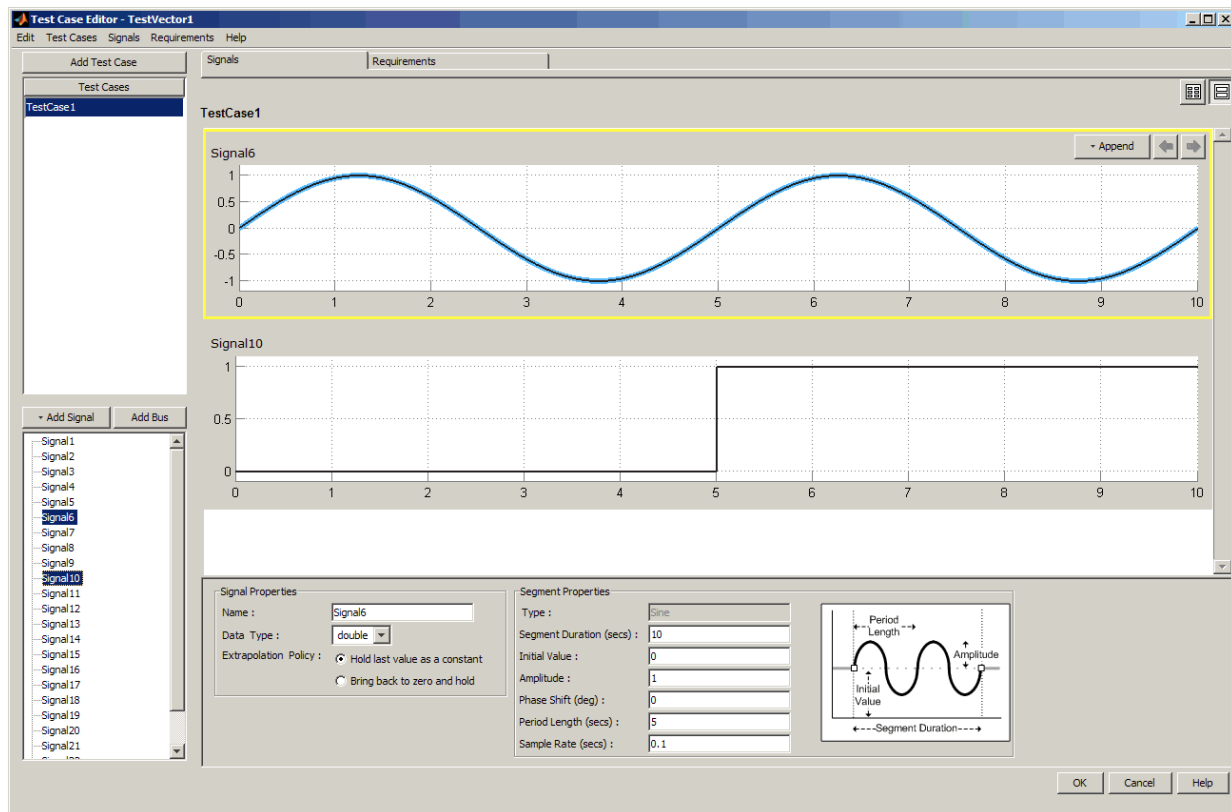
In the lower-left pane is a signal list that lists every signal in the selected test case. You can click signals in the signal list to select or clear them. You can select multiple signals by using the **Ctrl** key as you click on them.

Only signals that are selected in the signal list are shown in the signal display area. If all signals are selected, all the signals' plots will be displayed in the signal display area. A scroll bar appears if the signal plots take up more vertical room than is available in the signal display area.

In the signal display area, only one signal at a time can be selected. The plot of the currently selected signal is outlined in yellow. Properties shown in the **Signal Properties** section are the values for the currently selected signal. For information on setting signal properties, see step 4 in "Add Signals to Test Cases" on page 5-17.

Properties shown in the **Segment Properties** section are the values for the currently selected segment inside the selected signal. The diagram to the right of the properties shows graphical definitions of the values of the signal type. For information on setting segment properties, see steps 5 through 8 in "Add Signals to Test Cases" on page 5-17.

### Test Case View

The Test Case view shows an icon for each signal in the selected test case. The shape of the signal and its name are shown. The signals are displayed from left to right across each row and then continue in the next row down, in the order of creation. A scroll bar appears if the icons use more vertical room than is available in the view area.
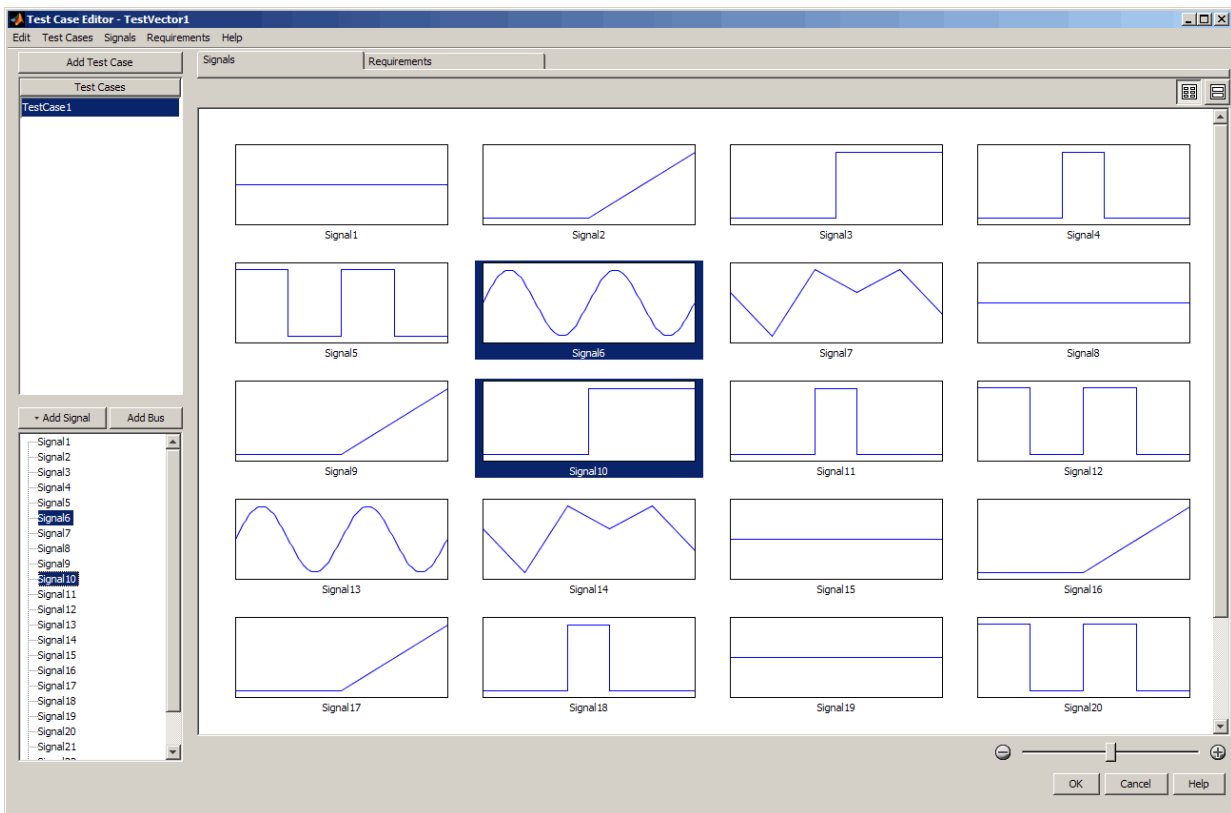


In the Test Case view, multiple signals can be selected. The signal icons that are selected are shown with a dark blue background. Unselected signals have a white background.

This view allows an easy way to find and manage signals when you have many signals in a test case. It is easier to locate a signal by looking at the graphics in this view. It is also easy to add and delete signals in this view. As in the Edit view, click the **Add Signal** button to add a signal in the Test Case view. If you want to edit the signal, you must return to the Edit view, by double-clicking the signal or by clicking the **Edit View** button. You can delete a signal in this view by selecting its icon and pressing the **Delete** key.

If you have a large number of signals, use the slider at the bottom of the Test Case view to adjust the size of the graphics, which in turn determines how many are shown at once. Sliding the bar to the left shrinks the size of the graphics and displays more of them. Sliding the bar to the right increases the size of the graphics and shows less of them.

When you click the **Edit View** button to return to the Edit view, the first signal that is selected in the Test Case view will be selected in the Edit view and its plot will be displayed in the edit area.

## Create Test Cases

When the Test Case Editor is opened from the Test Case Data test vector in the SystemTest desktop, it opens in the Edit view and contains one test case called `TestCase1` by default.

There are two workflows you could follow. You can add multiple test cases, and then go into each test case and add the signals, or you could add one test case and then add the signals to it, and then add another test case and its signals if you have multiple test cases. Both of these workflows are described below.

**To add test cases one at a time with their signals:**

1   Rename the default test case, by double-clicking `TestCase1` in the **Test Cases** list.

2   Type a new name for the first test case and press **Enter** to change the name.

3   With the renamed test case selected, edit it to add signals, as described in "Add Signals to Test Cases" on page 5-17.

4   Once the first test case is configured, if you need multiple test cases, click the **Add Test Case** button to add a second test case.

5   When creating a new test case, you have the option of populating it with signals from another test case, or with constant values of 0. In this case, your first test case

contains signals with values that you set, so you may want to use the second option, **The same signal values specified in the following test case**, and then select the first test case in the drop-down list. You can then edit the new test case to vary it from the first. This option is useful to duplicate values from another test case. If you do not want to start out with the same signals, select the other option, **A constant value of 0**.
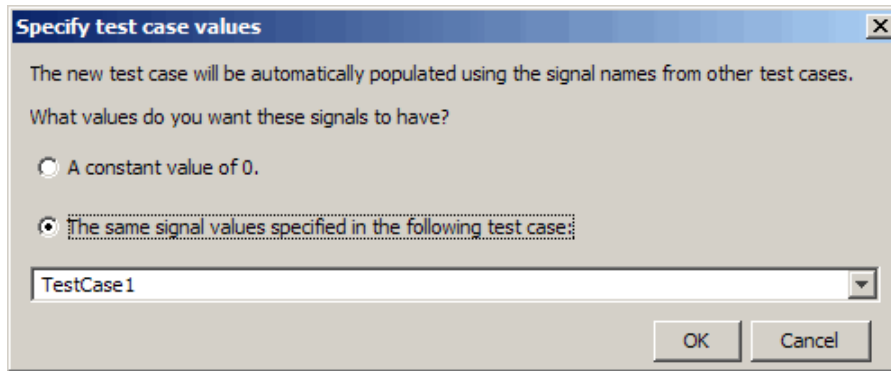


Click **OK**.

**6**  Add signals to the second test case and/or modify signals that it already contains.

**7**  Repeat these steps as necessary to create more test cases.

**To add multiple test cases and then signals:**

**1**  Rename the default test case, by double-clicking `TestCase1` in the **Test Cases** list.
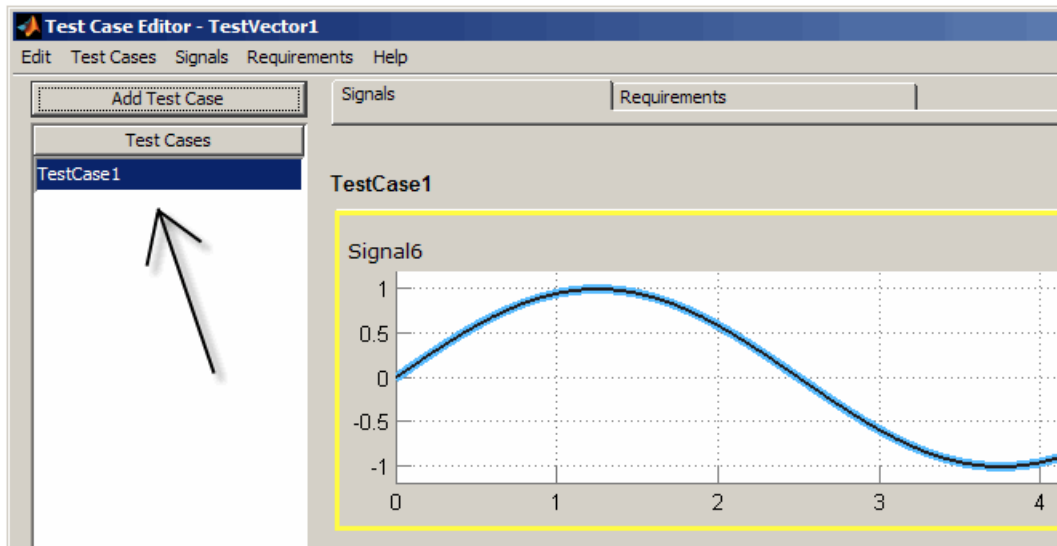
2   Type a new name for the first test case and press **Enter** to change the name.

3   Click the **Add Test Case** button to add a second test case.

---

**Note:** All test cases must have the same number of signals as well as the same set of names for their signals. For example if `TestCase1` has signals named `SignalA` and `SignalB`, when you create a new test case `TestCase2`, it will be populated with two signals named `SignalA` and `SignalB`. The two sets of signals may have different parameters and values set though.

---

4   When creating a new test case, you have the option of populating it with signals from another test case, or with constant values of 0. In this case the first test case's signal has not been modified yet, so keep the default option **A constant value of 0**, and click **OK**.
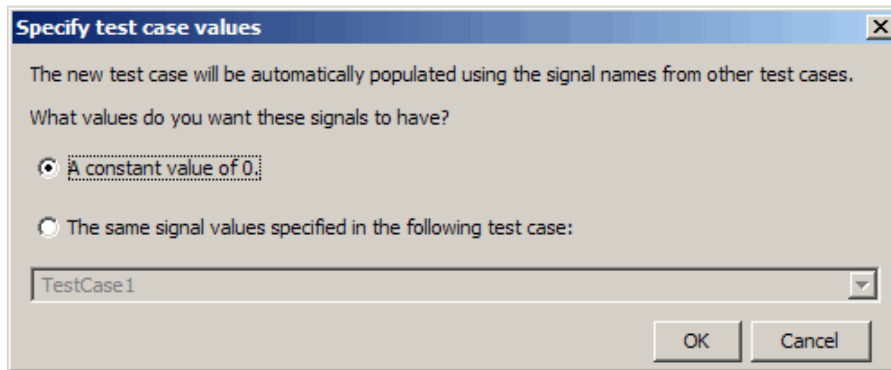
**5** Rename the second test case by double-clicking its default name, typing a new name, and pressing **Enter**.

**6** Repeat these steps to add as many test cases as you need.

**7** When the test cases are present, add signals to each one by selecting it in the **Test Cases** list and then editing it as described in "Add Signals to Test Cases" on page 5-17.

To delete a test case, select it in the **Test Cases** list and press the **Delete** key, or right-click and select **Delete** from the context menu.

**Test Case Options**

You can set options for the test case by right-clicking on a test case in the **Test Cases** list, and selecting **Options** from the context menu. This opens the Test Case Options dialog box.
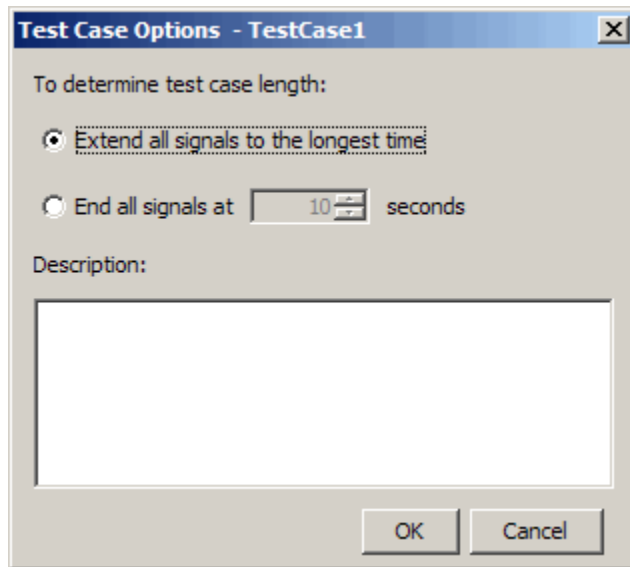


**To determine test case length:**

In order to run a test using the test cases you created, all signals within a test case must be the same length. In case some signals are shorter than others, use the following options to determine how to define the end time for all signals in the test case.

- **Extend all signals to the longest time** means that all the signals will be extended to the length of the longest signal. For example, if you have four signals of length 3, 5, 7, and 7, the signals of length 3 and 5 will both be extended to 7 seconds.

  Note that when a signal is extended, it is extended in the way that you select in the **Extrapolation Policy** property in the **Signal Properties** of a given signal. See step 4 in "Add Signals to Test Cases" on page 5-17 for more information on setting that property.

- If you select **End all signals at n seconds**, use the arrows to choose the length that you want all signals to use. Every signal in the test case then ends at that time in seconds.

**Description**

You can optionally add a description for the test case here. Enter your text in the text field and it is saved when you click **OK**.

## Add Signals to Test Cases

You build test cases in the Test Case Editor by adding signals to them. The test cases are then used in the Test Case Data test vector, through the Simulink element in the SystemTest software. The tool supports the following signal types: constant, ramp, step, pulse, square, sine, and custom.

For information on opening the Test Case Editor from the SystemTest software, see "Author and Use Signals in Tests" on page 5-4. For information on creating test cases, see "Create Test Cases" on page 5-12.

To create signals:

**1**    In the **Test Case** list, select the test case you want to populate.

By default, the first time you open the tool from a new Test Case Data test vector, one test case is created.

**2**    The test case that is created by default contains one signal when it is created. You can modify that signal and use it as your first signal, or delete it.

**3**    For additional signals, click the **Add Signal** button that appears above the signal list in the lower-left pane.

Select the type of signal to add from the drop-down button. For information about the parameters and constraints of each signal type, see "Signal Types" on page 5-29.
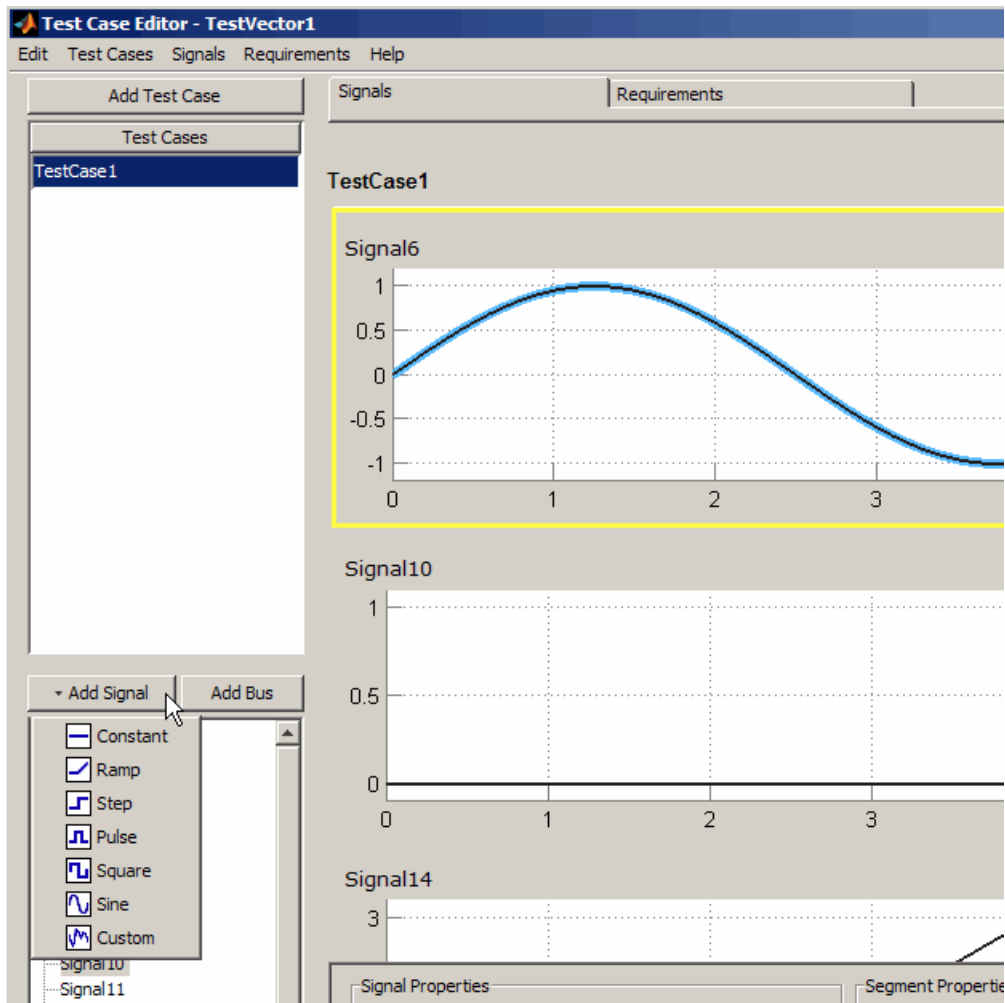
If you have only one test case, the new signal is added underneath whatever signal is currently selected in that test case.

If there are multiple test cases, you are prompted with the Add New Signal dialog box. Since all test cases need to contain the same signals, the new signal will also be added to all other test cases. Select the value to propagate to the other test case(s), then click **OK**.

**4** Set the signal properties for the signal. These are set in the **Signal Properties** section underneath the signal plots.
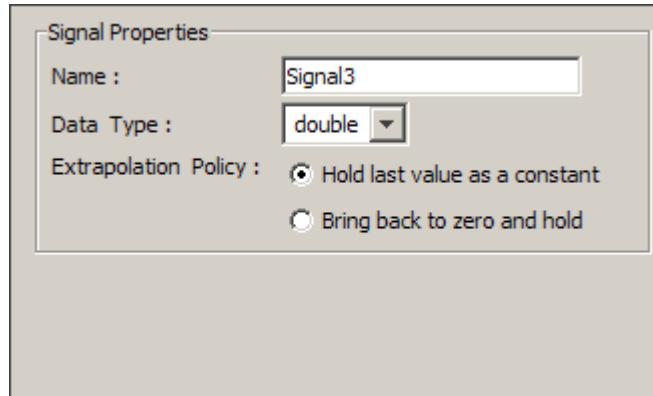


You can edit signal properties for the selected signal, as follows:

**Name** — New signals are given a default name. Type a new name in the edit field.

**Data Type** — New signals are data type `double` by default. Accept the default or select a different data type from the list of standard MATLAB data types.

Changing the data type changes the value of the signal. Note that if you change the data type, the change will be applied at run time of the test, but will not be visually reflected in the Test Case Editor – the Editor's user interface will display the signal as if it were still data type `double` in the signal plot in the **Edit** view, as well as the plot icon in the **Test Case** view. When you run the test, the values that are used will reflect the data type that is set.

**Extrapolation Policy** — All the signals in a test case must have the same length. If this signal is shorter than the longest signal, this option determines what value is used to lengthen it. **Hold last value as a constant** means that the end value of the signal's last segment will be held as a constant for the rest of the time. Selecting **Bring back to zero and hold** drops the signal to 0 after the last segment ends.

Property edits are committed when you press **Enter** or click outside of the edit field.

**5** When you add a signal, it will contain one segment and use default values for that signal type. Set values for the selected segment in the **Segment Properties** section. Each type of signal has a different set of properties to set. For example, a `constant`

**5-19**

has only **Segment Duration** and **Value** properties, and a `pulse` has properties for **Segment Duration**, **Initial Value**, **Offset**, **Final Value**, and **Pulse Width**.
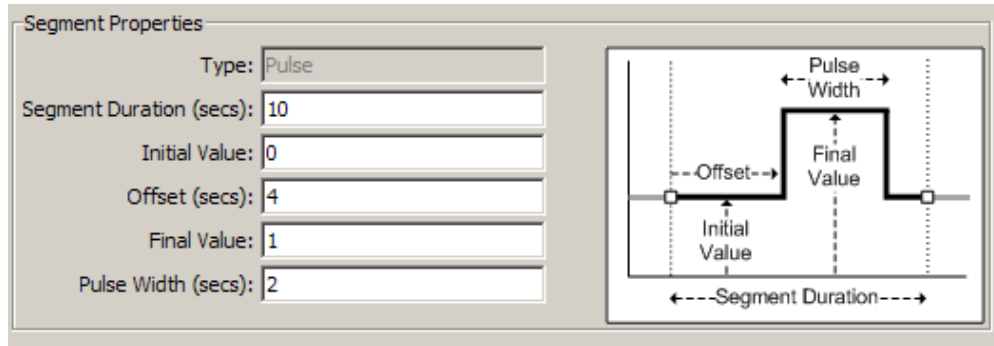


For information about the parameters and constraints of each signal type, see "Signal Types" on page 5-29.

As you set values in the **Segment Properties** section, they are committed and immediately reflected in the plot of the signal when you click outside of a field or press **Enter**.

6  Signals contain one segment by default but you can append multiple segments to define your signal. You can create a signal by concatenating together any of the supported signal types as segments. The contiguous segments make up the entire signal's value.

To append a segment to the selected signal, click the **Append** button in the signal plot. Select a signal type for the new segment from the drop-down list.

The **Append** button and the arrow buttons only appear in the plot of the currently selected signal. A signal can be selected by clicking it when in the Edit view.
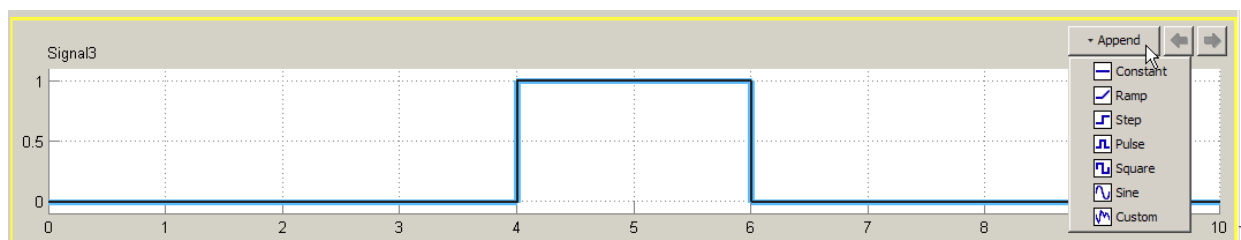
The new segment is appended to the end of the currently selected segment, and uses default values for that signal type.

**7**   Edit the **Segment Properties** for the new segment. Each segment has its own properties and any edits made will be applied to the currently selected segment.

See "Signal Concatenation" on page 5-21 for information on the rules that govern concatenation.

**8**   While editing a signal with multiple segments:

- The currently selected segment is highlighted with a blue line in the plot. Click a segment to select it.

- You can use the left and right arrow buttons in the plot to move the currently selected segment.

- Increasing or decreasing a segment's length results in shifting the other segments of the signal as well. (However, it does not change their lengths.)

- You can delete the currently selected segment by pressing the **Delete** key.

- Note that if you set the **Extend all signals to the longest time** option in the Test Case Options, when you make the current signal longer by adding a segment, any other signals in the test case will be extended to the same length, using the **Extrapolation Policy** you selected in the **Signal Properties**. In the other signal(s), the additional length is shown as a dotted line in its plot.

**9**   Repeat these steps to add as many segments to a signal as necessary.

---

**Note:** For an example of using signals created in the Test Case Editor in a Simulink element, see "Test Cases and Signals in Simulink Element" on page 4-46.

---

**Signal Concatenation**

You can create a signal by concatenating any of the supported signal types as segments. The contiguous segments make up the entire signal's value. The following rules apply:

- The first segment's start time is 0.

- Any other segments' start time is the same as the end time of the previous segment.

- The length of a segment is its duration.

- The end time of a segment is its start time + duration.

- The length of a signal is the sum of the duration of all of the segments.

When the duration of a segment changes, it has no effect on the duration of any other segments. The length of the other segments remains the same. The length of the entire signal changes however, because one of its segments became shorter or longer.

If a segment is added or deleted from a signal, this has no effect on neighboring segments' parameters. Individual segments remain the same length but the signal length changes as a result.

## Buses in the Test Case Editor

The Test Case Editor supports the use of buses in your model and you can use it to create buses and signals within buses.

The hierarchy of buses and signals appears in the signal list in the lower-left pane of the Test Case Editor, as shown here in the Edit view.

In this example you can see that `In1` is a bus containing two signals, `pressure` and `temperature`. The signal `pressure` is selected in the signal tree and consequently is displayed and selected in the editing pane, when in the Edit view.

Nested bus hierarchies are supported. In the example you can see that bus `In5` contains another bus `a1`, which contains two buses `left` and `right`. The `left` and `right` buses each contain two signals, `pressure` and `temperature`.

You can select multiple signals in the signal list by pressing the **Ctrl** key as you select signals. The following example shows a test case with three signals selected. You can see that the three signals are selected in the signal list, two of them within bus `In1` and one root-level individual signal `In6`. Those three signals are also shown in the signal editing area. Notice that the signal that is selected in the editing pane, `pressure`, is labeled `In1.pressure` in the signal diagram, to denote it is a signal within bus `In1`.

---

**Note:** If you have a model that contains buses, you can automatically generate a test harness from the model. It will create and configure all of the appropriate parts of the test, including elements, test vectors, and mappings, and create a test case containing all of the Inport blocks as buses and/or signals. For more information, see "Test Harness Generation".

---

You can perform some actions by right-clicking buses and signals in the signal tree. The context menu includes the following commands:

- Cut
- Copy
- Paste
- Add Signal
- Add Bus
- Rename
- Delete
- Copy Signal Values to the Same Signal in All Test Cases

In the Test Case view, each level of the bus hierarchy is displayed via icons, as shown here.
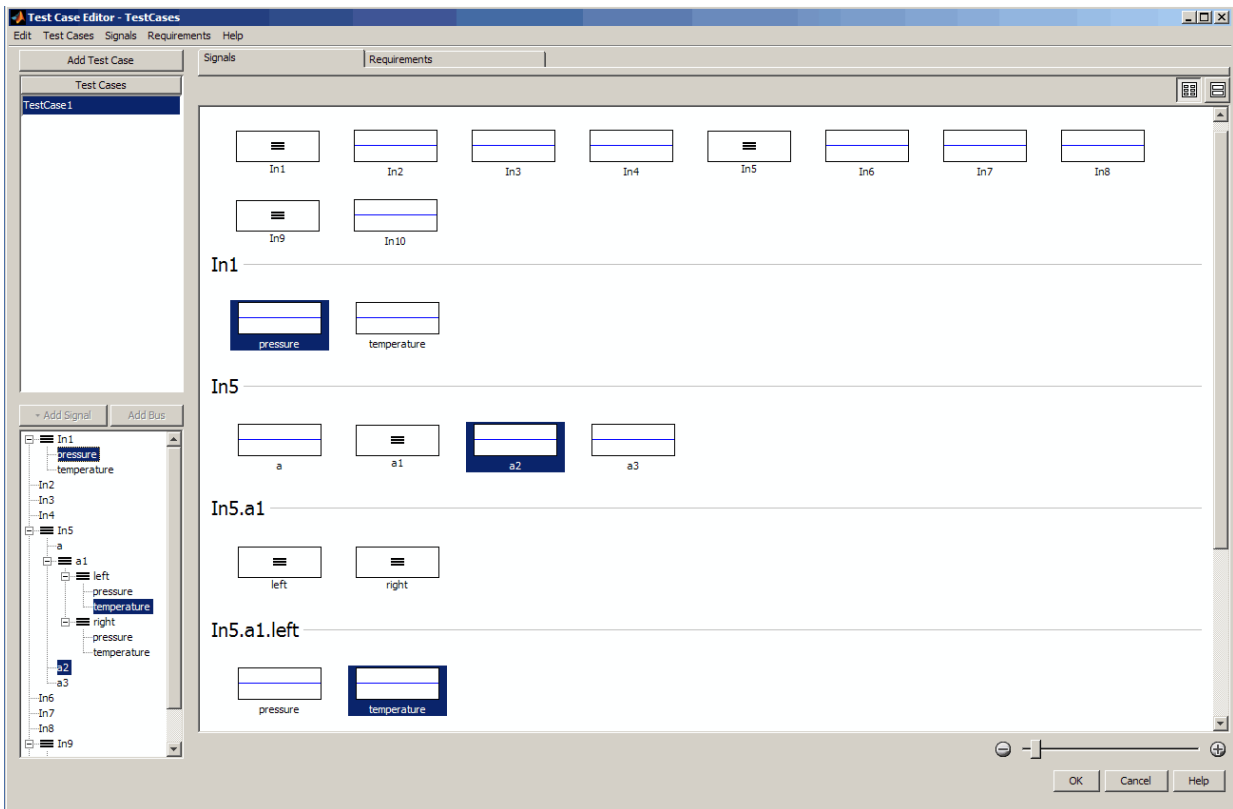
The section at the top of the Test Case view shows all of the signals and buses that are included in the test case, in the order they appear in the signal tree. After that, each bus is shown as a group with the signals it contains. The groups appear in the order they appear in the signal tree. Groups for nested buses are also shown.

Notice in the example that bus `In1` is a root-level bus containing two signals and no nested buses. But bus `In5` is a root-level bus that contains three individual signals, `a`, `a2`, and `a3`, as well as a nested hierarchy of buses `a1`, `left`, and `right`. Each subgroup is shown in its own section in the Test Case view.

Notice that one of the signals that is selected, `temperature`, is in a group that is labeled `In5.a1.left` in the signal diagram. This indicates it is a signal within the bus `left`, which is within bus `a1`, which is within bus `In5`.

To edit a signal from the Test Case view, double-click the signal's icon in the icon area or select the signal icon and right-click **Edit Signal**. The Edit view appears and that signal will be selected for editing.

### Add Buses to a Test Case

You can add a bus to a test case, and then add one or more signals to the bus.

1  In the Edit view, click the **Add Bus** button that appears above the signal list in the lower-left pane.

   The bus is added below any existing buses or signals in the list, at the highest level in the hierarchy (the root level). It is called Bus1 by default. You can double-click the bus in the signal list to rename it.

2  The bus is empty until you add signals to it. With the name of the bus still selected in the signal list, click the **Add Signal** button.

   Select the type of signal to add from the drop-down button.

3  Add as many signals to the bus as you need, as described in "Add Signals to Test Cases" on page 5-17.

4  To add another bus, click the **Add Bus** button again. A second bus, called Bus2 by default, is added under the first one, at the same level in the hierarchy if a top-level individual signal was selected. If a bus or a signal within a bus is selected when you click **Add Bus**, the new bus is added under the first bus in a nested hierarchy.

   For example, in the following diagram, In10 is selected when **Add Bus** is clicked. As a result, Bus1 is added at the end of the bus and signal list. With In10 still selected, if the button is clicked again, Bus2 is added under Bus1 at the same root level.

   With Bus1 selected, clicking **Add Signal** results in Signal1 being added to that bus. With Bus2 selected, **Add Signal** results in a Signal1 being added to that bus. Then with Bus1 selected, if you click **Add Bus** again, the new bus is added under the signal(s) of the first bus as a nested bus, as shown here.
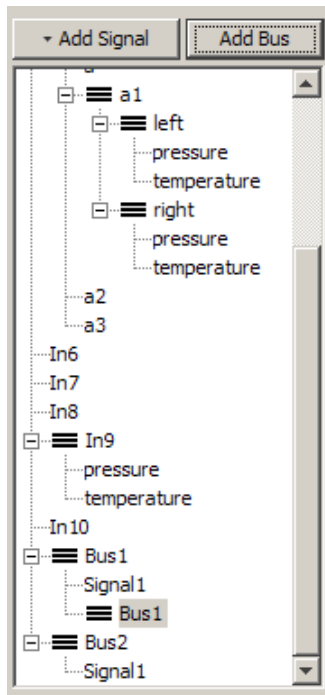
## Signal Types

Each signal is defined by several parameters. Using these parameters, the signal generates time-based data. The following rules apply to all the signals:

- All signal parameters are readable, writable scalar doubles, unless otherwise noted.
- All time-related parameters are defined in seconds.
- All parameters have constraints that must always be true, and are enforced when the value is set. For example, `Duration` must always be positive.

The following tables describe the built-in signal types the tool uses.

**Constant**
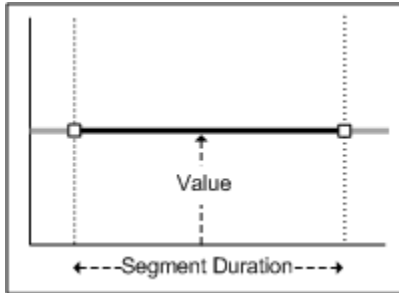
| Parameter | Definition | Default Value | Constraints |
|---|---|---|---|
| **Segment Duration** | The length of the signal in seconds. | 10 | > 0 |
| **Value** | The constant value of the signal the entire **Duration**. | 1 | none |

Configuration constraint: none.

**Step**
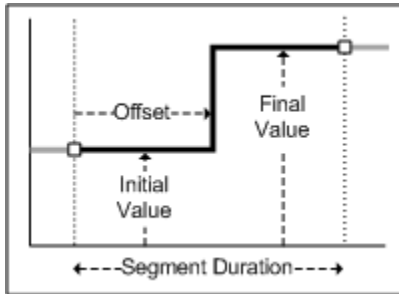


| Parameter | Definition | Default Value | Constraints |
|---|---|---|---|
| **Segment Duration** | The length of the signal in seconds. | 10 | > 0 |
| **Initial Value** | The value of the signal before **Offset**. | 0 | none |
| **Final Value** | The value of the signal after **Offset**. | 1 | none |
| **Offset** | The time in seconds when the signal switches from **Initial Value** to **Final Value**. | 5 | > 0 |

Configuration constraint: Duration > Offset.

**Ramp**
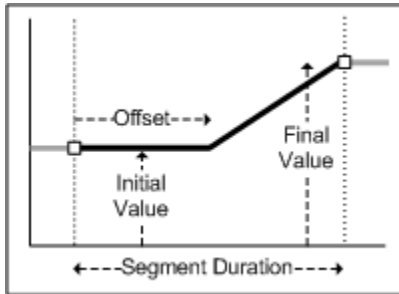


| Parameter | Definition | Default Value | Constraints |
|---|---|---|---|
| **Segment Duration** | The length of the signal in seconds. | 10 | > 0 |
| **Initial Value** | The starting value of the signal. | 0 | none |
| **Final Value** | The ending value of the signal. | 1 | none |
| **Offset** | The time in seconds when the signal begins to transition from **Initial Value** to **Final Value**. | 5 | => 0 |
| **Slope** | The rate of change of the signal over time. | .2 | read-only |

Configuration constraint: Initial Value ≠ Final Value.

**Pulse**



| Parameter | Definition | Default Value | Constraints |
|-----------|------------|---------------|-------------|
| **Segment Duration** | The length of the signal in seconds. | 10 | > 0 |
| **Initial Value** | The value of the signal before **Offset** and after **Offset + Pulse Width**. | 0 | none |
| **Final Value** | The value of the pulse after **Offset** and before **Offset + Pulse Width**. | 1 | none |
| **Offset** | The time in seconds when the signal transitions from **Initial Value** to **Final Value**. | 4 | > 0 |
| **Pulse Width** | The amount of time in seconds after **Offset** when the signal has the value **Final Value**. The signal returns to **Initial Value** afterward for the rest of the signal. | 2 | > 0 |

Configuration constraint: Duration > Offset + Pulse Width.

Configuration constraint: Initial Value ≠ Final Value.

**Square**
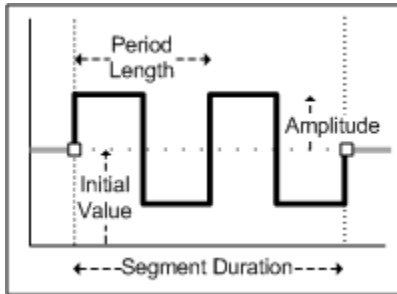


| Parameter | Definition | Default Value | Constraints |
|---|---|---|---|
| **Segment Duration** | The length of the signal in seconds. | 10 | > 0 |
| **Initial Value** | The amount the square wave is offset vertically. | 0 | none |
| **Amplitude** | The value of the signal while in the high state. | 1 | > 0 |
| **Duty Cycle** | The percentage of time the square wave has **Amplitude** as opposed to –**Amplitude**. | .5 | 0 < dc <1 |
| **Phase Shift** | The value in degrees the signal is horizontally shifted into its period. | 0 | none |
| **Period Length** | The length in time for a full repetition of the wave. | 5 | > 0 |

Configuration constraint: none.

**Sine**
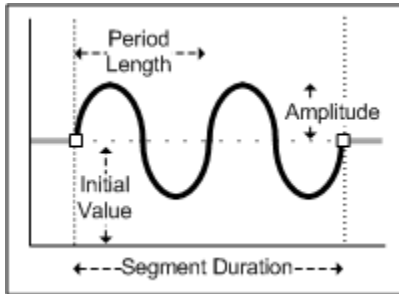


| Parameter | Definition | Default Value | Constraints |
|---|---|---|---|
| **Segment Duration** | The length of the signal in seconds. | 10 | > 0 |
| **Initial Value** | The amount the wave is offset vertically. | 0 | none |
| **Amplitude** | The amplitude of the sine wave. | 1 | > 0 |
| **Phase Shift** | The horizontal shift of the period in degrees. | 0 | none |
| **Period Length** | The length in seconds of a period. | 5 | > 0 |
| **Sample Rate** | The amount in seconds between each sampled point. | .1 | > 0 |

Configuration constraint: none.

**Custom**
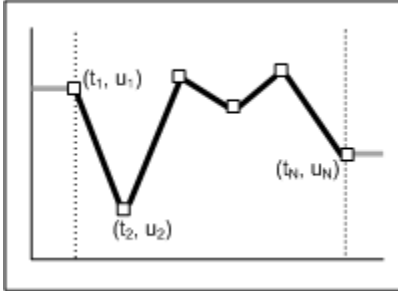


| Parameter | Definition | Default Value | Constraints |
|-----------|-----------|---------------|-------------|
| **Time** | User-defined time vector. | [0 2 4 6 8 10] | 1xN increasing double |
| **Data** | User-defined value vector. | [2 0 3 2 3 1] | 1xN double |

Configuration constraint: Time and Values must have the same length and dimension.

When specified, Time or Values may be 1xN or Nx1. If specified as Nx1, it will automatically be converted to a 1xN.

# Link to Requirements in IBM Rational DOORS

| In this section... |
|---|
| "Introduction and Setup" on page 5-37 |
| "Add Requirements" on page 5-37 |
| "Requirements Tab" on page 5-40 |
| "Test Case Report" on page 5-43 |
| "Create Requirements Programmatically" on page 5-45 |

## Introduction and Setup

You can link test cases that you created in the Test Case Editor to requirements that are in IBM Rational DOORS. This is done through the **Requirements** tab in the Test Case Editor. The integration allows you to easily link any DOORS requirements to any test cases by selecting them using their object headings.

**Note:** You need a license for Simulink Verification and Validation to use this feature.

**Note:** Before using this feature, you must run a setup program one time on the machine you will be using. In a command window, type

```
rmi setup
```

and press **Enter**. This function is part of Simulink Verification and Validation setup and enables the use of the IBM Rational DOORS integration to link requirements to a test case in SystemTest. For more information, see the `rmi` function in the Simulink Verification and Validation documentation.

## Add Requirements

Requirements are linked to the currently selected test case in the Test Case Editor.

To add requirements to a test case:

1   Select a test case from the **Test Cases** list in the Test Case Editor.

2   Click the **Requirements** tab.

3   Click the **Add link to new DOORS module** button.
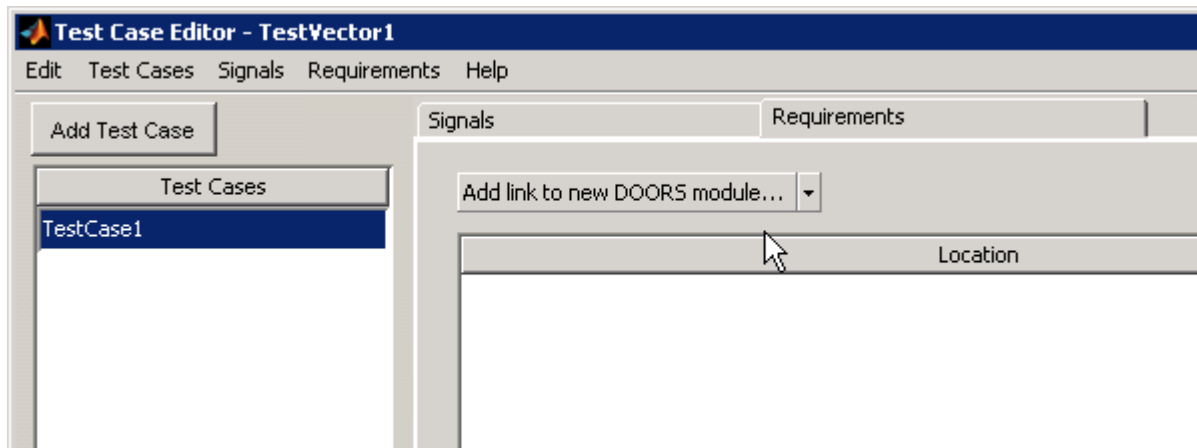


The DOORS module selection dialog box opens. Note that DOORS must be open for this integration to work. If DOORS is not open, an error occurs.

4   In the Browse DOORS dialog box, browse to the module you want to link to.
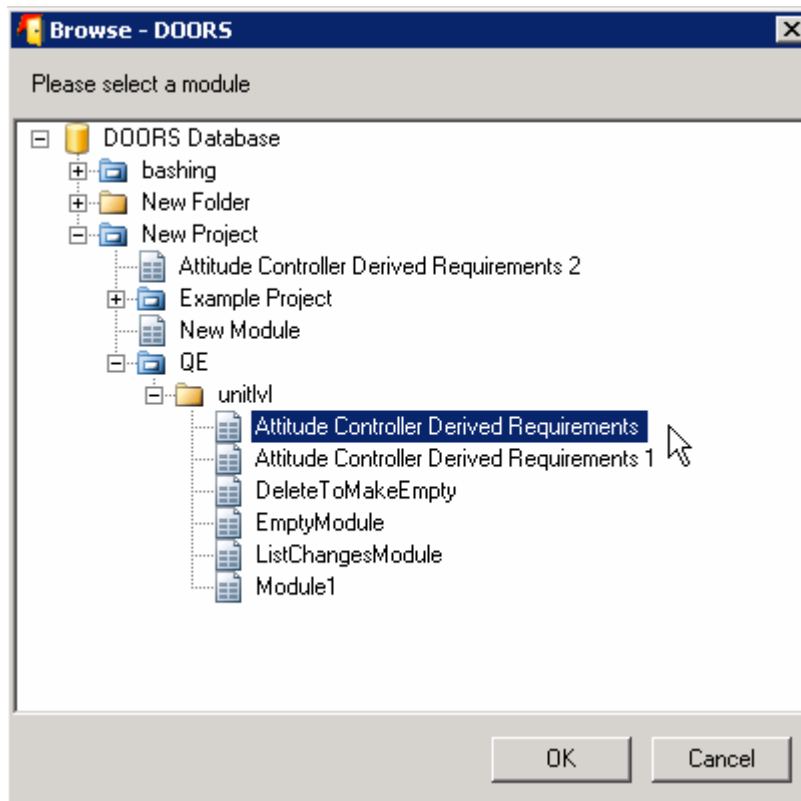
**5**   Click **OK** to add the module.

The requirement appears in the table in the main area of the **Requirements** tab.

**6**   Select the object heading you want to link to.
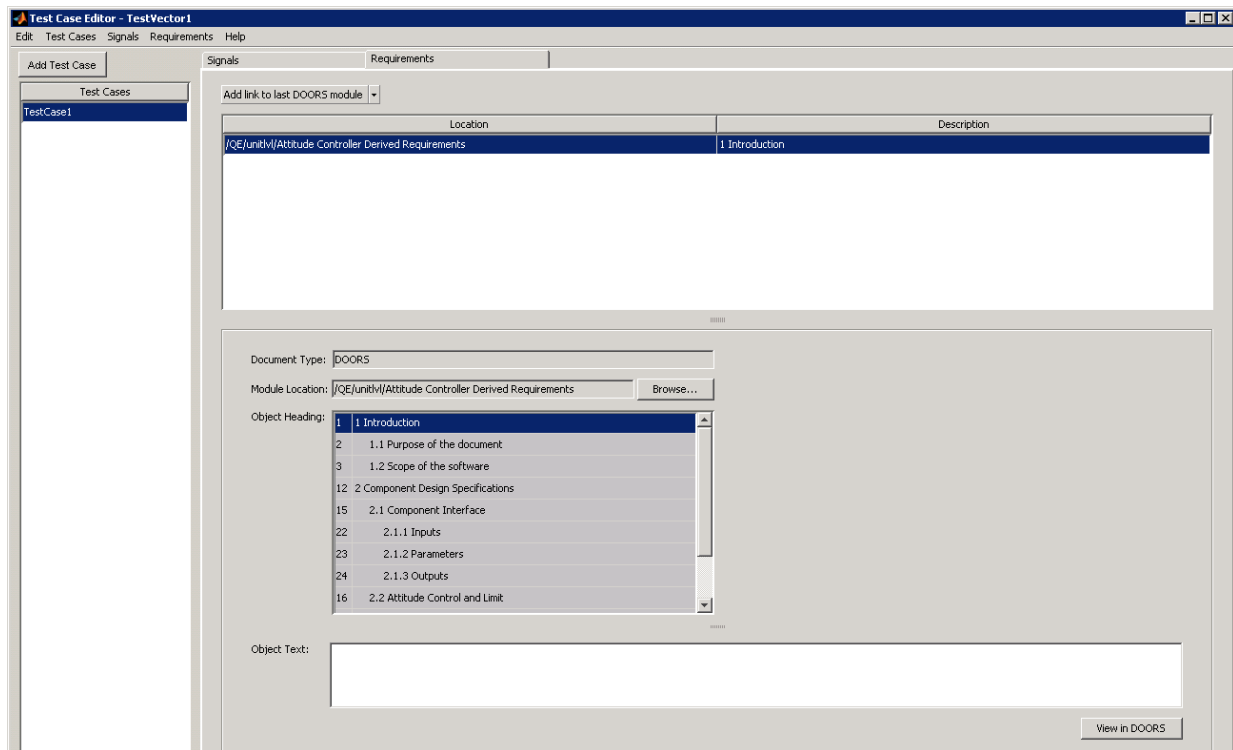
After you add a requirement, the **Add** button becomes **Add link to last DOORS module** and the Browse DOORS dialog opens to the module from which you have already selected.

## Requirements Tab

The table displays requirements and contains the following columns:

- **Location** — Displays the module location.
- **Description** — Displays the DOORS title number and heading.
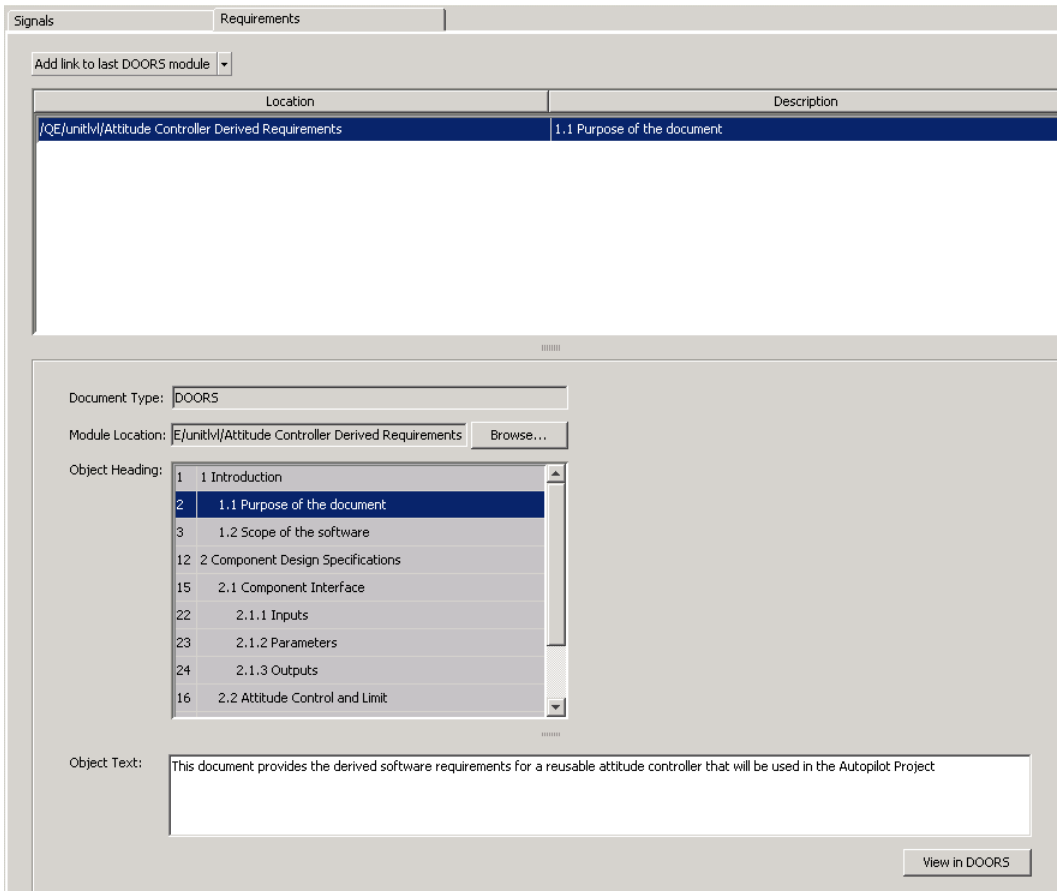
In the details section under the table, details of the selected requirement are displayed, as follows:

- **Document Type** — Indicates the source document of the requirements, in this case, IBM Rational DOORS.

- **Module Location** — Shows location of the DOORS document.

- **Object Heading** — Displays the list of available objects in the module location. It shows the DOORS object ID number, title number, and heading. Selecting an object updates the description in the **Object Text** area.

- **Object Text** — Displays DOORS object text of the currently selected object heading. It is empty if DOORS is not open or available.

  Note in the previous illustration that object 2: "1.1 Purpose of the Document" is selected, and its **Object Text** is displayed, "This document provides.....".

- **View in DOORS** button — Navigates to the object in DOORS. If DOORS is not open or available, it produces an error.

  In the example shown above, where the requirement 2 (number 1.1, Purpose of the Document) was selected, when the **View in DOORS** button is clicked, the following graphic shows how it opens in DOORS with that requirement selected.
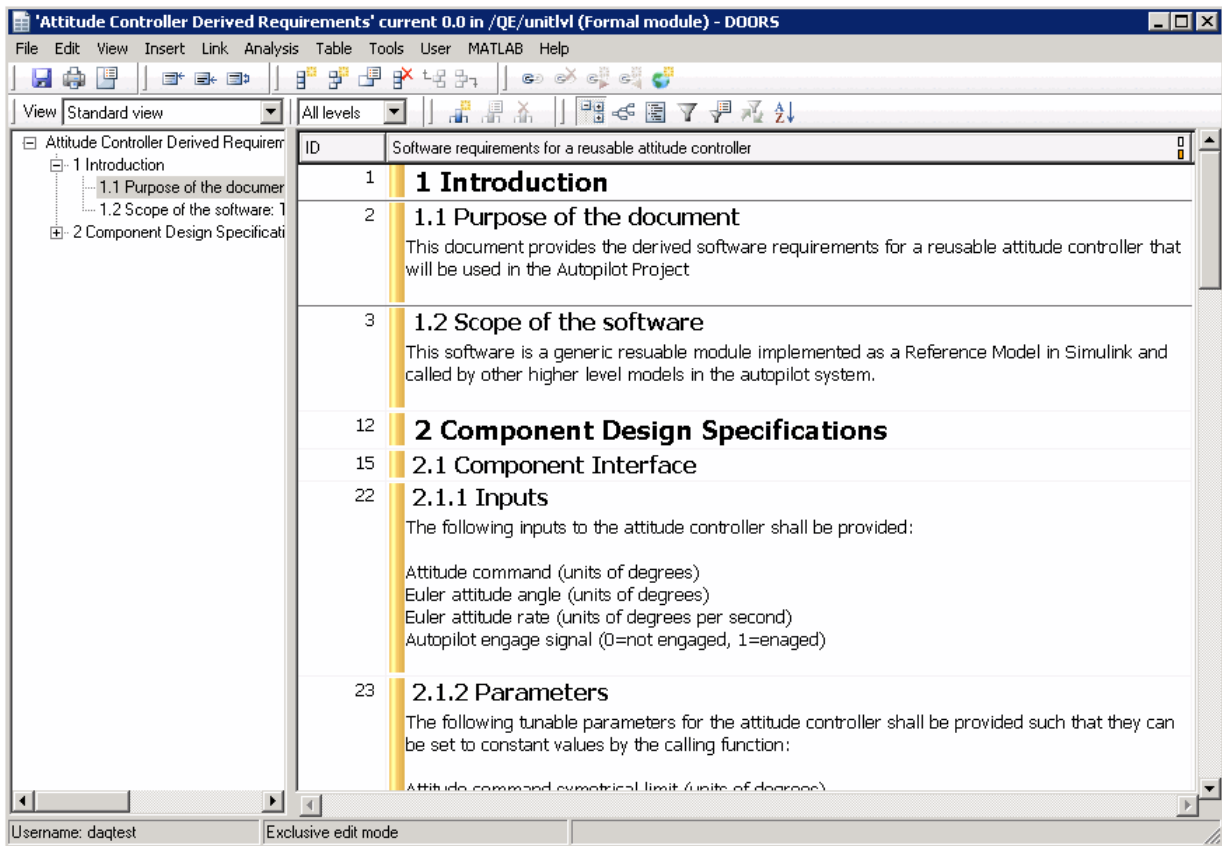
## Test Case Report

The Test Case Editor has a separate report that links from the SystemTest Test Report. If you link requirements to a test case, additional sections are added to the Test Case report.

The **Requirements** section is created if at least one requirement is attached to a Test Case Data test vector. If you have a Simulink Verification and Validation license, the Object Text will be available in addition to the other information.

A Test Case Editor report is generated when a SystemTest Test Report is generated and you run a test that uses a Test Case Data test vector. To enable the Test Report, in the SystemTest desktop, click the test name in the **Test Browser**, and then click the **Output Files** tab on the **Properties** pane. In the **Select File Names** section, select the **Generate report** option.

If the report is enabled and you run a test containing a Test Case Data test vector with requirements, you can open the report at the end of the run by clicking the **Test Report** link in **Run Status** pane of the SystemTest desktop.

Using the example from the previous section, the following window shows the report that is created.
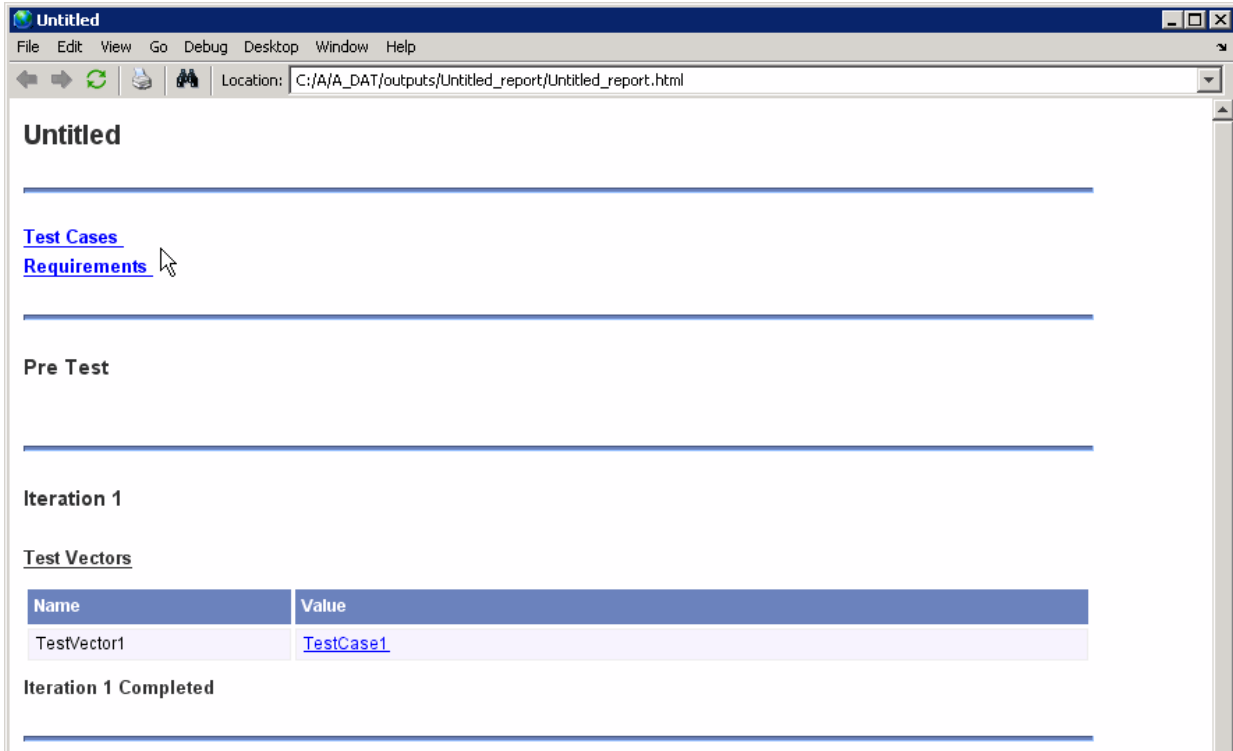
At the top of the report you can see the link to the Test Case report. When you click that link, the Test Case Report opens.



Notice the link to the requirement in that report. If you click that link, the details about that requirement appear.



## Create Requirements Programmatically

In addition to creating requirements in the Test Case Editor as described in the previous sections, you can create requirement links programmatically for sue with IBM Rational DOORS.

To create a requirement link, use the `systest.requirements.createlink` function, as follows. Note that DOORS must be running.

Create a requirement link object to a DOORS object "1" in the module "/example/MyModule".

```
 reqLinkObj = systest.requirements.createLink('DOORS', '/example/MyModule' ,'DOORS Object','1')
```

Create a requirement link object from a requirement link structure attached to a Signal Builder block in a model.

```
blockPath = 'mymodel/SignalBuilderBlock/';
reqStruct = rmi('get',blockPath,1);
reqLinkObj = systest.requirements.createLink(reqStruct);
```

For more information, see the reference page for systest.requirements.createlink.

You can determine the supported requirements information using the getInfo function.

info = systest.requirements.getInfo(*format,modulelocation*) returns information describing the supported link values in the modulelocation for a given format. format and modulelocation must be specified as a string. format is not case sensitive but modulelocation is case sensitive. info is returned as a 1x1 structure containing the following fields:

- ModuleID – A string containing the module ID.
- ModuleLocation – A string containing the module location.
- AvailableObjectIds – A 1xN cell array of strings containing the object IDs for the specified ModuleLocation.
- ObjectID – The ID string for the DOORS object.
- ObjectHeading – The heading string of the DOORS object.

To get the module location of the DOORS object:

doorsObject.ModuleLocation

You can use the function getObjectText() to get the Object Text of a DOORS object when DOORS is open:

txt = getObjectText(doorsObject);

You can use the getStatus() function to determine the requirement link's status.

[validflag msg] = getStatus(obj) gets the status of a DOORS Requirement link object obj. validflag is true if you are also able to navigate to the DOORS object.

If `validflag` is false, the function returns a message `msg` describing why the link is invalid.

You can view the requirement link in DOORS using the `view()` function. To open the link in the module:

```
view(obj)
```

This function opens the module in DOORS if the link is valid. It throws an error if DOORS is not available or open. It also errors if Simulink Verification and Validation is not installed. It errors if `getStatus(obj)` is false.

### Examples

A design engineer at an automobile company uses DOORS to capture her requirements. The requirements are in a module inside a project. The engineer has created a test case for three of the requirement objects in the module. She wants to link these requirements to the test case.

```
objectIDList = {'OO1234','OO1235','OO1236'};

module = '/ProjectCar/EngineModel';

doorsReqObjs = systest.requirements.createLink('DOORS',module,'DOORS Object',objectIDList);

testCases.Properties.Requirements = doorReqObjs;
```

While working in the test case, the engineer wants to look at the requirements to make sure the test case has the correct values.

```
view(testCase.Properties.Requirements(3))
```

# Test Cases and Signals in SystemTest Elements

| In this section... |
| --- |
| |
| |
| |
| |

## Introduction

You can use the test cases and signals you create in the Test Case Editor within your test by using some of the test elements within the SystemTest software.

You can select a Test Case Data test vector or individual signals from the test vector within the following elements:

- Simulink element
- MATLAB element
- General Plot element

The following sections discuss using test cases and signals in these elements.

## Simulink Element

You can create signals in the Test Case Editor and use them to test a Simulink model. You do this by mapping the signals in the Simulink element using a Test Case Data test vector.

One possible high-level workflow of using test cases and signals in your test via the Simulink element is:

- Create a Test Case Data test vector.
- Open the Test Case Editor from the test vector.
- Create a test case and signals in the Test Case Editor.
- Return to the SystemTest desktop and create a Simulink element.

- In the Simulink element, map Inport blocks in your model to the signals you created in the Test Case Editor by selecting the Test Case Data test vector or individual signals in the Simulink element.

**Note:** For an example of using signals created in the Test Case Editor in a Simulink element, see "Test Cases and Signals in Simulink Element" on page 4-46. It includes the workflow outlined here and gives details on the steps in the Simulink element.

## MATLAB Element

You can access the data from a Test Case Data test vector by using a MATLAB element in a test that has a Test Case Data test vector. You could use the data for a variety of reasons, such as writing it to a CSV file, calling a custom function, or creating a plot.

To see example code you could use in a MATLAB element, see "Access Test Case Data Using MATLAB Element" on page 2-69.

## General Plot Element

You can plot data from a Test Case Data test vector or any individual signals from a Test Case Data test vector in a General Plot element. Test Case Data test vectors and signals are supported in two plot types – **plot** and **Simulink data**. Any other plot type results in an error at run time.

**Note:** You can only plot an individual signal in the General Plot element. If your test case contains a bus, you cannot select the bus in the plot. You can select an individual signal within the bus.

The following sections describe the behavior of using a Test Case Data test vector or an individual signal for these two plot types.

**plot Plot Type**

- Test Vector — If you use a Test Case Data test vector as the **Y Data Source** and **X Data Source** is left as `<Auto>`, then all signals within the test vector are plotted on the same axes versus their times. In the example shown here, the test vector `TestVector1` is selected, so all four of its signals will be plotted.
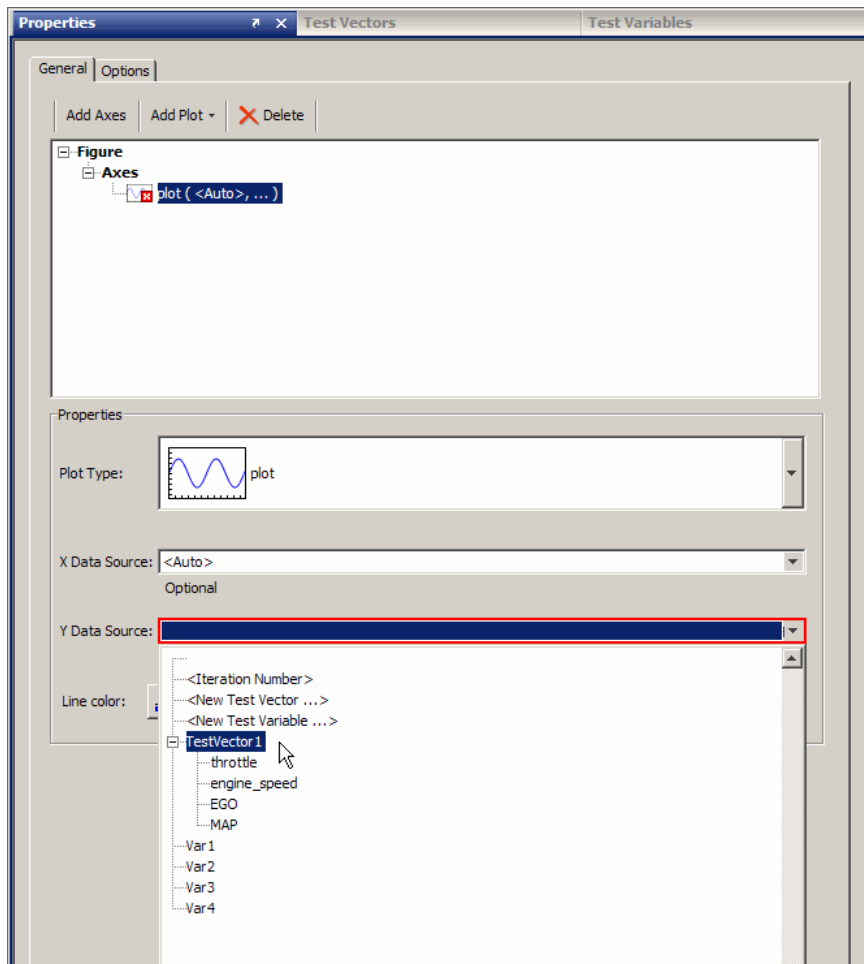


If your test vector includes signals that are scaled very differently, see the note about scaling at the end of this section.

- Individual Signal — If you specify an individual signal as the **Y Data Source** and **X Data Source** is left as `<Auto>`, then that signal is plotted versus its time. In the example shown here, the signal `engine_speed` is selected, so that signal will be plotted.
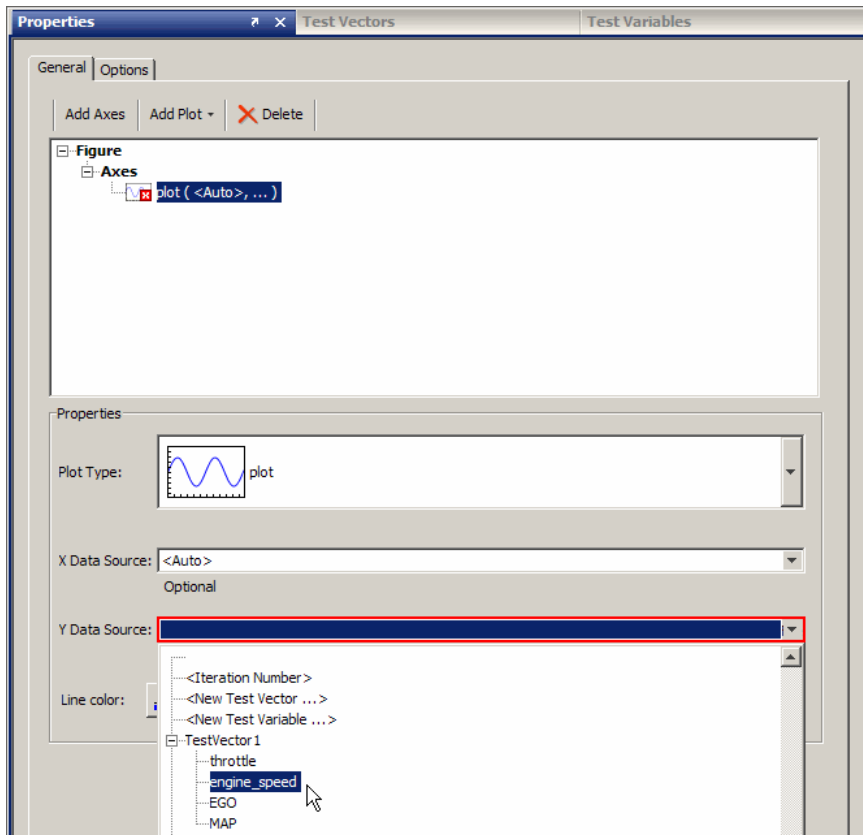
**Simulink data Plot Type**

- Test Vector — If you select a Test Case Data test vector in the **Simulink Data** field, then all signals within the test vector are plotted on the same axes versus their times. In the example shown here, the test vector `TestVector1` is selected, so all four of its signals will be plotted.
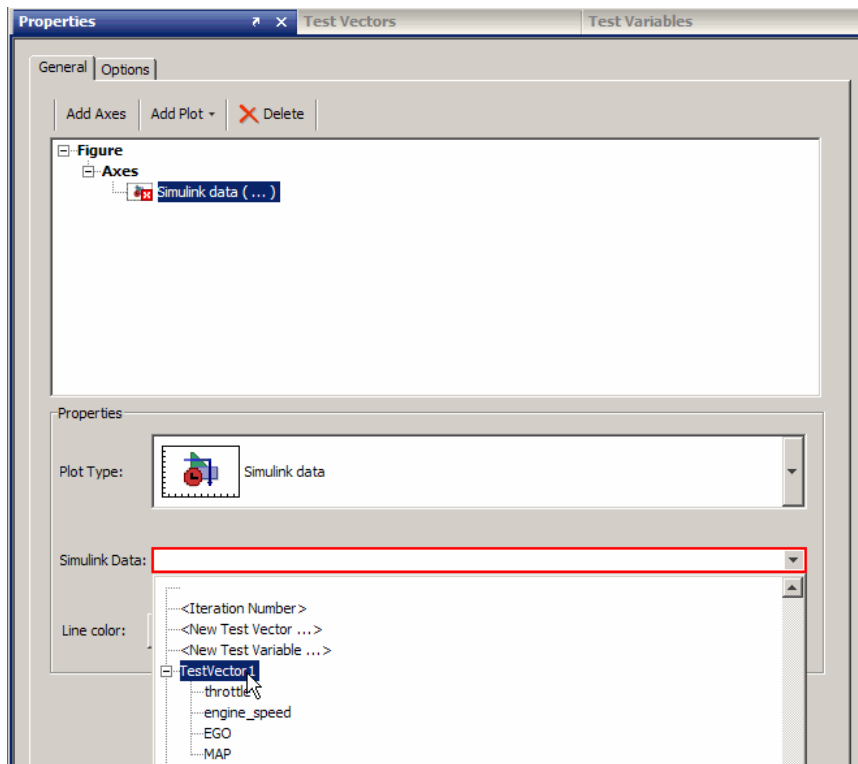


If your test vector includes signals that are scaled very differently, see the note about scaling at the end of this section.

- Individual Signal — If you specify an individual signal in the **Simulink Data** field, then that signal is plotted versus its time. In the example shown here, the signal `engine_speed` is selected, so only that signal will be plotted.
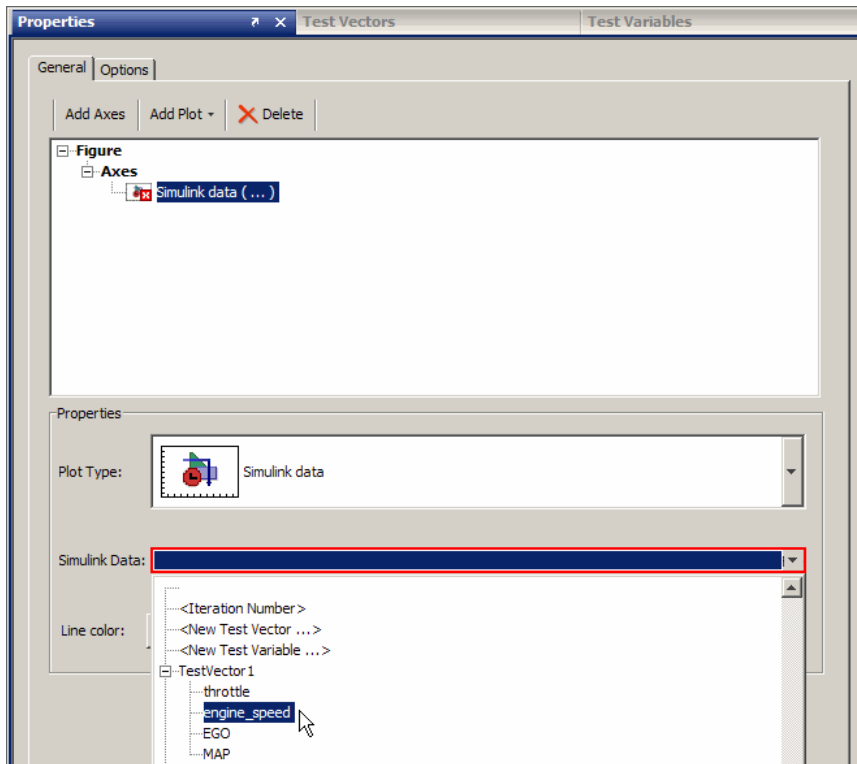
**Note:** If you plot a Test Case Data text vector, either using **plot** or **Simulink data** plot type, and the signals within the test vector are scaled very differently, you may prefer to plot the signals on different axes.

If you want each signal to appear with its own scale, add an axes for each signal and then add the plot to each axes. For example, if you have `TestVector1` and it has three signals, `Signal1`, `Signal2`, and `Signal3`, you could plot it as shown here.
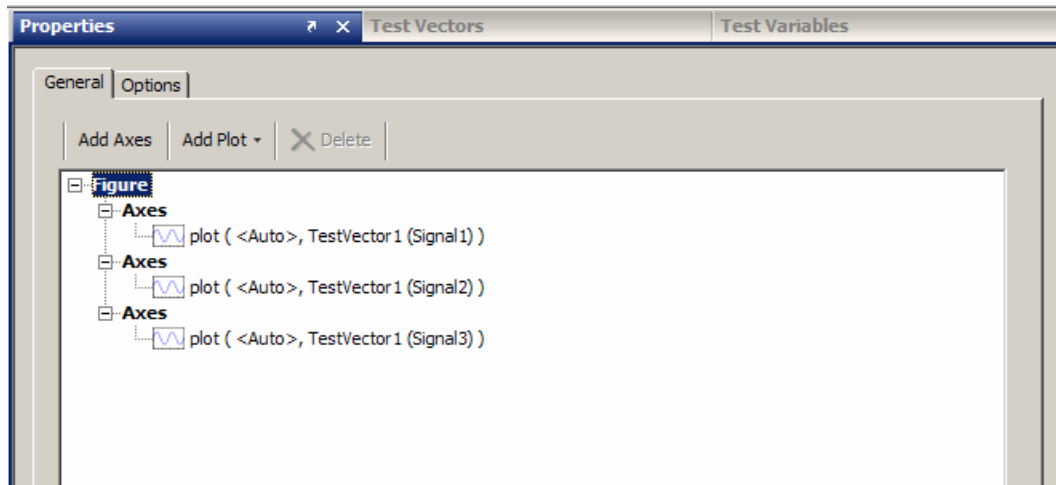
# Programmatic Test Case and Signal Authoring

| In this section... |
| --- |
| |
| |
| |
| |
| |

## Overview

The functionality of the Test Case Editor, as described in this chapter, is also available via a command-line interface. The functions allow you to do the following tasks programmatically:

- create test cases, signals, and segments
- add test cases, signals, and segments to a SystemTest test
- configure and edit test cases, signals, and segments in a test
- extract test cases, signals, and segments from a test

For information on usage and syntax of the functions, see these reference pages.

The objects:

- `systest.TestCase`
- `systest.signals.Signal`
- `systest.signals.segments`

The saving and loading functions:

- `stLoadTestCases`
- `stSaveTestCases`

The signal manipulation functions:

- `isSignal`
- `setSignal`

- `getSignal`
- `removeSignal`
- `renameSignal`
- `setDataType`
- `horzcat`
- `getInfo`

## Load and Save Test Cases

If you have a test with test cases you can load and save them programmatically. The test cases can be created using the Test Case Editor, as described in this chapter, or they can be created with the automatic test harness generation feature. A test can be generated automatically from Simulink (see "Generate a Test Harness").

In this example workflow, we will generate a test automatically, then load the test cases, modify them, then save the test cases.

1  Create a SystemTest test called `myTest.test` from the command line based on the model `myModel`.

```
systest.createHarness('C:\Work\myTest.test','myModel');
```

The test is created and put into your `C:\Work` folder.

2  As part of the test generation, one or more test cases are created with signal names corresponding to the Inport blocks in your model. You can now access the test case(s) programmatically.

Load the test cases contained in the test `myTest.test`.

```
testCases = stLoadTestCases ('myTest.test');
```

3  You can now modify the test cases, using the functions listed in "Overview" on page 5-55. For example, you could add and/or modify signals contained in the test case(s).

4  After you have finished working with the test case(s), you can save them back to the test.

```
stSaveTestCases('myTest.test', testCases);
```

5  Load the test in the SystemTest desktop.

```
systemtest('myTest.test')
```

## Edit Test Cases

You can add test cases, signals, and segments to a test and configure and edit existing test cases, signals, and segments in a test.

`signal = systest.signals.Signal(`*`segment_type`*`)` creates a signal with a segment of type *segment_type*.

For a list of supported segment types and their properties, see the reference page for the `systest.signals.Signal` class.

The following is an example workflow of editing existing test cases. In this case we load the test case, add a signal to it, then save the test case.

**1** Load the test cases contained in the test `myTest.test`.

```
testCases = stLoadTestCases ('myTest.test');
```

**2** Create a `ramp` segment with an `Offset` of `2` and a `FinalValue` of `6`.

```
segment = systest.signals.segments.Ramp('Offset', 2, 'FinalValue', 6)
```

**3** Update the first test case's signal `MySignal` to use the new segment.

```
testCases(1).MySignal.Segments = segment;
```

**4** After you have worked with the test case(s), you can save them back to the test.

```
stSaveTestCases('myTest.test', testCases);
```

## Create Signals

You can create signals with default properties or create them and specify properties. You can create the following signal/segment types:

- `Constant` – A segment with a constant value.
- `Custom` – A segment with user-specified time and data vectors.

  Properties include:

- `Pulse` – A segment with a pulse value.

- `Ramp` – A segment with a linearly changing value.
- `Sine` – A periodic sine wave.
- `Square` – A periodic series of pulses.
- `Step` – A segment that transitions from a one value to another.

For a list of the properties for each segment type, see the reference page for the `systest.signals.segments` package.

If you add a segment and do not specify any properties, it is created with default properties. Default properties of the signals are defined in "Signal Types" on page 5-29. To add properties, follow the syntax shown in the examples below.

**Creating Signals with Default Values**

Create a signal with one segment using default values, in this case a `constant`.

```
systest.signals.Signal('Constant')
```

Create a Signal with two segments using default values, in this case a `constant` segment followed by a `step`.

```
systest.signals.Signal('Constant', 'Step')
```

**Creating Signals with Properties**

Create a segment with one property, in this case a `Constant` segment with a `Value` of `5`.

```
 segment = systest.signals.segments.Constant('Value', 5)
```

Create a segment with multiple properties, in this case a `ramp` segment with an `Offset` of `2` and a `FinalValue` of `10`.

```
 segment = systest.signals.segments.Ramp('Offset', 2, 'FinalValue', 10)
```

**Appending Segments to Signals**

If you create segments, you need to append them to signals. The following is an example of creating a signal, creating segments, then adding the segments to the signal.

Create a signal with one segment.

```
 signal = systest.signals.Signal('Step');
```

Create two stand-alone segments.

```
ramp = systest.signals.segments.Ramp();
pulse = systest.signals.segments.Pulse();
```

Add the two segments to the end of the signal's `Segments` property, which is an array of segment objects.

```
signal.Segments = [signal.Segments ramp pulse]
```

## Import from External Source to Test Case

You can use the programmatic interface to import test cases from external sources, such as an Excel file, a Simulink Signal Builder harness, or a Simulink Design Verifier data file.

The following is an example of the basic high-level workflow of importing data from an external source. To see the details of this example, see the example "Importing Test Cases from Excel into a Test Harness" by opening the SystemTest Help, then Examples > Programmatic Interface > Importing Test Cases from Excel into a Harness.

1   Start with a model. In the case of this example, the model contains an Inport block with a bus that contains four signals.
2   The example imports a test case from a single worksheet in an Excel file using the function `xlsread`. It also assigns the data in the columns to signals in the test case.
3   Create a test case using the `systest.TestCase` function.
4   Create signals from the data in the spreadsheet columns using the `systest.signals.Signal` function.
5   Create a SystemTest test file using the `systest.createHarness` function.
6   Then append the test case to the newly created test using the `stLoadTestCases` and `stSaveTestCases` functions.

To see the specific commands for these steps, see the above referenced example in the SystemTest Help.

**6**

# Generate a Test Harness from a Model

# Test Harness Generation

You can automatically generate a SystemTest test harness from a model in Simulink. It will create and configure all of the appropriate parts of the test, including elements, test vectors, and mappings.

The following steps are automatically performed from your model:

- Creates a SystemTest test.
- Creates a Simulink element.
- Creates a Test Case Data test vector.
- Automatically maps each Inport block to the corresponding signal of the test vector in the Simulink element.
- Creates signals with names that match the root-level Inport block names.
- Sets up the data type of each signal based on the Inport blocks' data type.
- If the signal has buses, sets up the signals' data type for the Inport block using buses as data, and matches the hierarchy of the bus in the test case.
- Automatically sets up the model name and location in the Simulink element for top-level models with root-level inports and outports.

# Model Requirements for Test Harness Generation

The automatic test generation requires that the model contain root-level Inports. If it does not, you will get an error message and a test harness will not be created. The following conditions apply to the Inports:

- Model must contain root-level Inports.
- The model's root-level Inports must have scalar dimensions. If any contain non-scalar dimensions, you will get an error.
- The model's root-level Inports must use a supported datatype. If any use an unsupported datatype, you will get an error. Supported datatypes include `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `logical`.

The test file name and folder location must be writable in order for the test to be created (step 2 in the next section). If it is not, you will get an error. Choose another name or location that is writable.

The model must be able to be compiled. If it fails to compile (using the **Update Diagram** button in Simulink), you will get an error.

# Generate a Test Harness

You can create a SystemTest test harness based on a Simulink model by using the command line. The resulting test that is automatically generated is configured for you and the appropriate components of the test are created, including the Simulink element and a Test Case Data test vector.

The automatic test generation requires that the model contain root-level Inports. If it does not, you will get an error message and a test harness will not be created. There are also other conditions that apply to the root-level Inports and the model. For a list of conditions to use this feature, see "Model Requirements for Test Harness Generation" on page 6-3. The test is set up with a Test Case Data test vector and a Simulink element using the information from the Simulink model. The model must be on the MATLAB path. The testFileName must be a writable file location.

This example uses the following simple model, which contains two Inports, one of which is a bus.
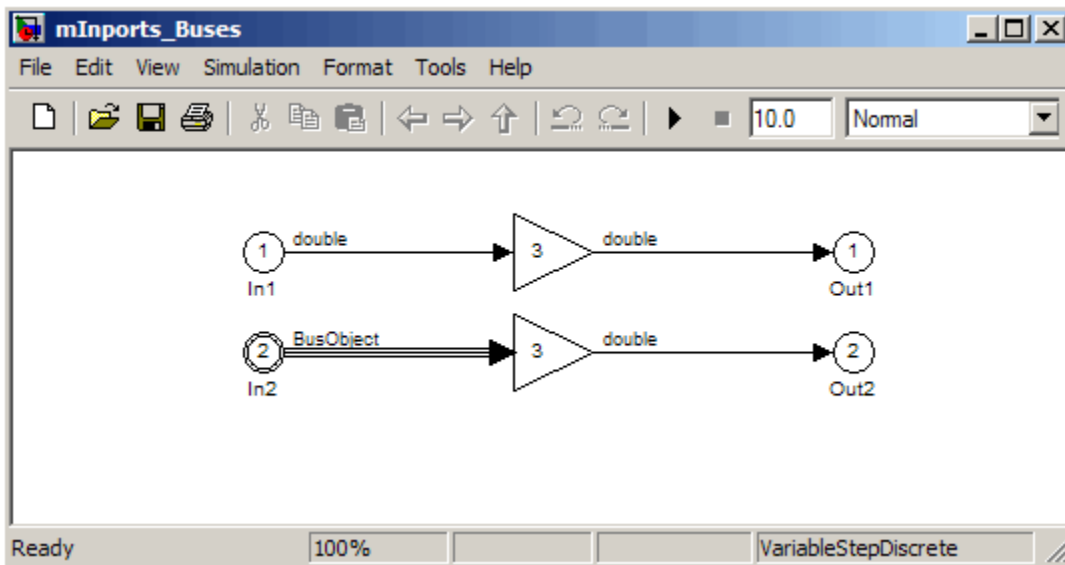


To create a SystemTest test harness:

**1** In the MATLAB command window, enter:

```
systest.createHarness('H:\Documents\MATLAB\mInports_Buses_harness.test','mInports_B
```

**2**  Open SystemTest and the new test from the command window using:

```
systemtest('H:\Documents\MATLAB\mInports_Buses_harness.test')
```

You can see the Simulink element and the test vector that were automatically created in the **Test Browser**.
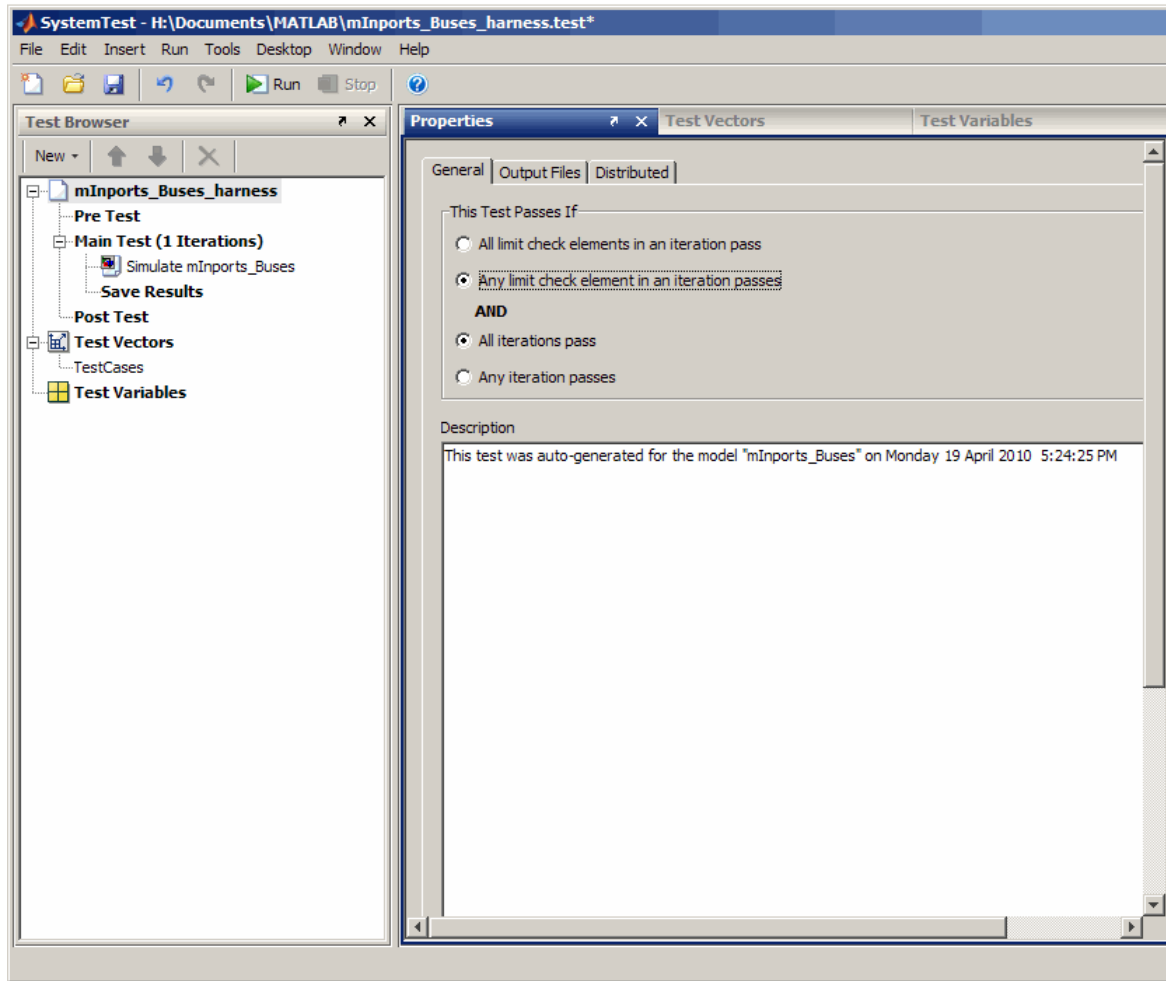
Note that text is added to the Test **Description** on the **General** tab indicating that this test was auto-generated from your model.

Other test properties are set to their defaults – **Select Output Folder** is set to **Same folder as TEST-file**, the results file is named "*<test file name>*_results.mat", the **Generate report** option is selected, and the **Output Folder Numbering** option is set to **Always use the same folder (overwrite files)**. You can see these options on the **Output Files** tab of the **Properties** pane.

**3** Select the Simulink element in the **Test Browser**. By default it is named "Simulate *<model name>*".

If you want to change the name of the element, double-click it in the **Test Browser** and type a new name.

The generated test automatically maps Inport blocks from the model to signals in the test vector that is created, and uses the **Individual Inports blocks are mapped** option. Notice in the example model shown in the beginning of this section that there are two Inport blocks, In1 and In2. Those two Inports are mapped in the Simulink element, as shown here.
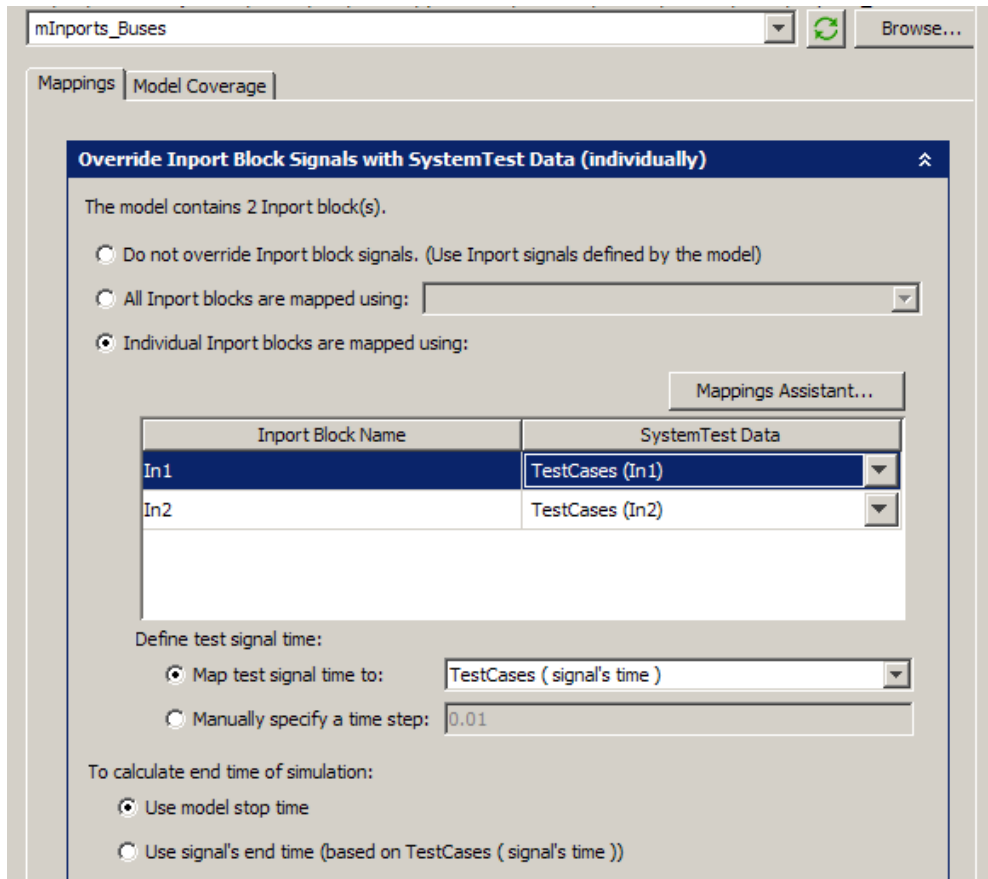
TestCases (In1) is the signal called In1 in the Test Case Data test vector called TestCases. In1 is a regular signal and In2 is a bus signal. You can see that it is a bus if you expand the signal in the **SystemTest Data** list.
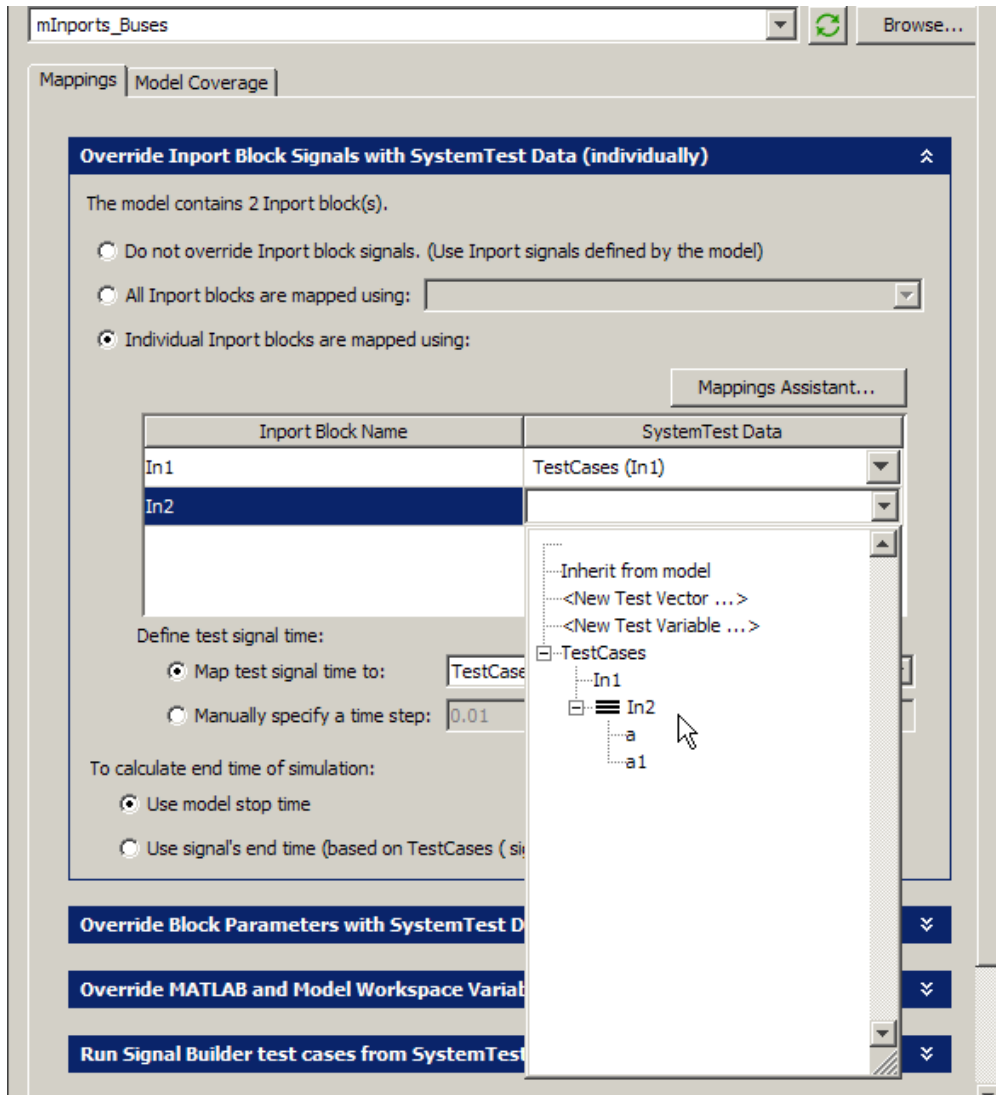
The Simulink element is configured to map the signal time to the test vector signal's time and to use the model stop time.

**4** Select the test vector in the **Test Browser**. The test vector is called `TestCases` by default. If you want to change the name, type a new name in the **Name** field on the **General** tab.

**5** Click the **Open Test Case Editor** button on the **Test Vectors** pane to see the test case and signals that were created from the model. By default, the test case is called `TestCase1` and the signals are named the same as the Inport blocks in the model, as shown here. You can rename the test case by double-clicking it in the **Test Cases** list.
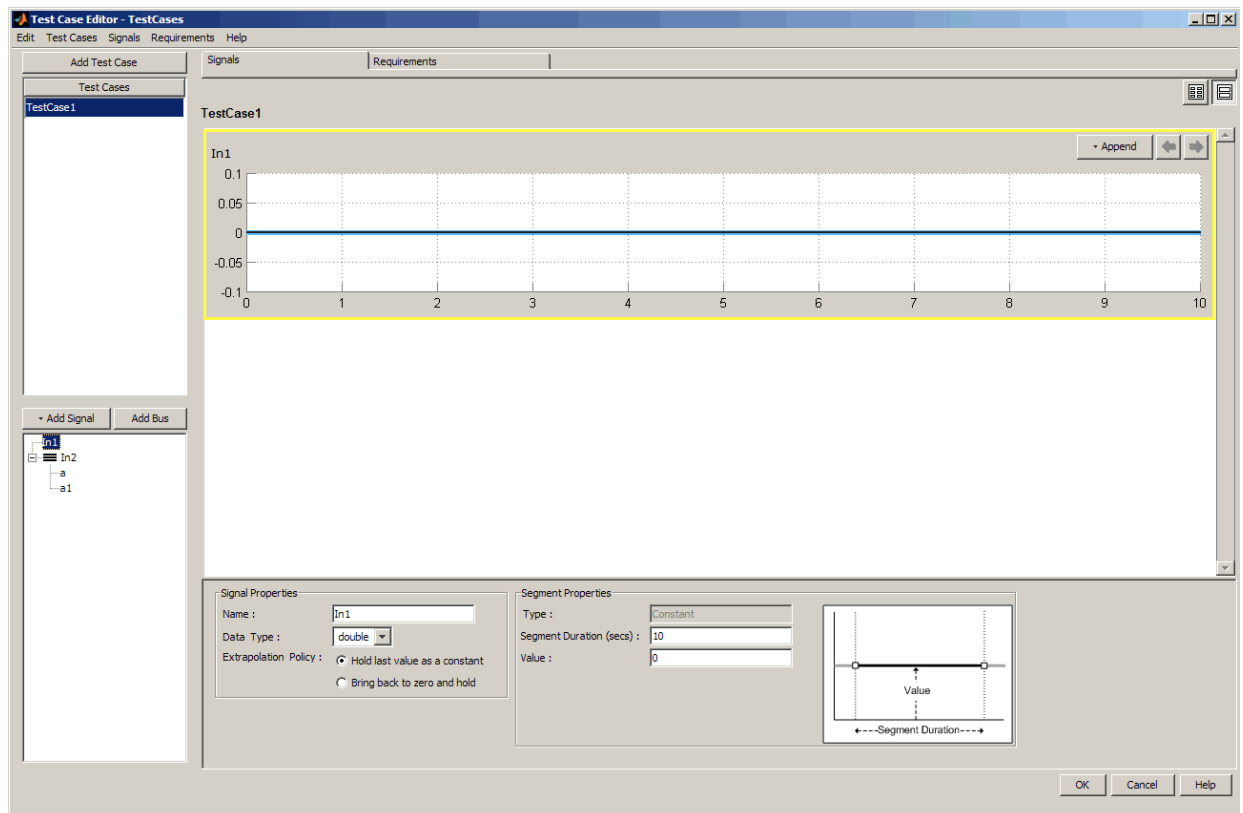


The example model has two Inport blocks, which appear in the signal list in the Test Case Editor. `In1` is selected here and you can see that the signal was created using the same **Data Type** for that signal as the Inport block had in the model, `double` in this case.

You can see in the signal list that the Inport block In2 is a bus and its two signals are shown in the signal tree.

Signals that are created are of **Type** Constant, have a default **Value** of 0, and a default **Duration** of 10 seconds. You can change any of these parameters by editing them in the **Signal Properties** or **Segment Properties**.

When you are done working in the Test Case Editor, click the **OK** button. Any additions or changes you made will be saved to the test vector.

6   Once you have created the test from your model as described here, you can make additions or modifications to any part of it. You can add test cases or signals in the Test Case Editor. You can add other elements to the test, such as the General Plot element to plot your data.

7   Run the test.

# Generate Test Harness at the Command Line

You can create a SystemTest test harness based on a Simulink model by using the command line.

The resulting test that is automatically generated is configured for you and the appropriate components of the test are created, including the Simulink element and a Test Case Data test vector.

The automatic test generation requires that the model contain root-level Inports. If it does not, you will get an error message and a test harness will not be created. There are also other conditions that apply to the root-level Inports and the model. For a list of conditions to use this feature, see "Model Requirements for Test Harness Generation" on page 6-3.

You use the `systest.createHarness` function to create the test. `systest.createHarness(testFileName,modelName)` creates a SystemTest test harness named *<testFileName>* for the model *<modelName>*. The test is set up with a Test Case Data test vector and a Simulink element using the information from the Simulink model. The model must be on the MATLAB path. The testFileName must be a writable file location.

The following example creates a test harness from a model:

```
>>modelName = 'C:\mymodel.slx';
>>testFileName = 'C:\my_new_harness.test';

>>systest.createHarness(testFileName,modelName)
```

# Use the Instrument Control Toolbox Elements

The Instrument Control Toolbox software provides several elements to use in the SystemTest software.

# Instrument Control Toolbox Elements

| **In this section...** |
| --- |
| "Overview" on page 7-2 |
| "Access Resources" on page 7-2 |

## Overview

The Instrument Control Toolbox elements provide a way to bring data from instruments into a SystemTest test, or to transmit data from your instrument. You can use these elements along with the other elements in the SystemTest software to create tests for Simulink models and other applications.

---

**Note** To use the Instrument Control Toolbox elements, you need a license for the Instrument Control Toolbox software. These three elements will not appear in the SystemTest software without this license.

---

The Instrument Control Toolbox software provides three of elements that you can use in the SystemTest software:

- To Instrument — For sending commands or data to your instrument
- From Instrument — For reading data from your instrument
- Query Instrument — For querying your instrument status or properties

You can configure these elements to communicate with your instruments by using SystemTest resources supported by the Instrument Control Toolbox software.

## Access Resources

If your MATLAB installation includes elements that require resources, the SystemTest desktop includes a **Resources** pane that lets you access the resources available through these toolboxes. For example, if your MATLAB installation includes the Instrument Control Toolbox software, you can see the **Resources** pane, if you open it from the **Desktop** menu. Select **Desktop > Resources** to open the pane. It will tab with the **Test Vectors** and **Test Variables** on the lower-left corner of the desktop. Resources are

toolbox-specific. For example, an Instrument resource might be configured to connect to a device over your computer's serial port.

# Measure Generator's Frequency Using Instrument Control Toolbox

| In this section... |
|---|
| |
| |
| |
| |
| |
| |

## Introduction

To illustrate how to use some of the Instrument Control Toolbox elements in the SystemTest software, this section provides a step-by-step example.

In this example a SystemTest element configures a signal generator to produce signals of various frequencies, which are measured by an oscilloscope configured by other SystemTest elements.

The signal generator is a Hewlett-Packard 33120A at GPIB address 5, and the oscilloscope is a Tektronix TDS 210 at GPIB address 4. For this example, the generator output is fed directly to the scope input. The generator will be programmed to generate signals of 1500, 5000, and 7500 Hz, while the oscilloscope will measure each signal's frequency.

The following sections explain the steps in this example.

## Setting Up the Signal Generator

The first element in the test programs the generator to output signals of various frequencies. To test at three frequencies, the test be comprised of three test cases, i.e., three iterations. This is a one-way communication to the generator, so you use a To Instrument element.

1. Open the SystemTest software from MATLAB by typing `systemtest` at the MATLAB command line.

2. No setup is required in the Pre Test, so the elements of this test are all in the main test, so click **Main Test** in the **Test Browser**.

3. Add an element by clicking **New Test Element > Instrument Control > To Instrument**.
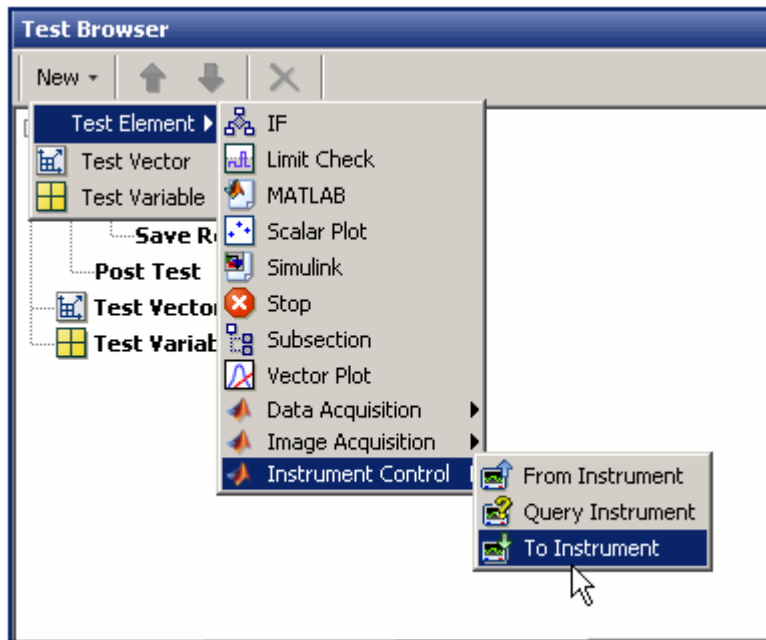


The element appears in the browser as `To Instrument`.

4. Double-click `To Instrument`, rename it `Set Generator`, and press **Enter**.

5. From the **Properties** pane's **Select an instrument resource** list, select `New Instrument Resource`. The instrument resource is the communication channel

between MATLAB and your instrument, in this case the generator at GPIB address 5.

**6** In the Edit: Instrument1 dialog box, enter `Generator` in the **Name** field.

**7** Click **Create** to create an instrument resource.

**8** In the New Object Creation dialog box, select GPIB in the **Instrument object type** list. Select the appropriate **Vendor** (in this example, `ni` for National Instruments), **Board index** (`0`), and instrument **Primary address** (in this example, `5`).
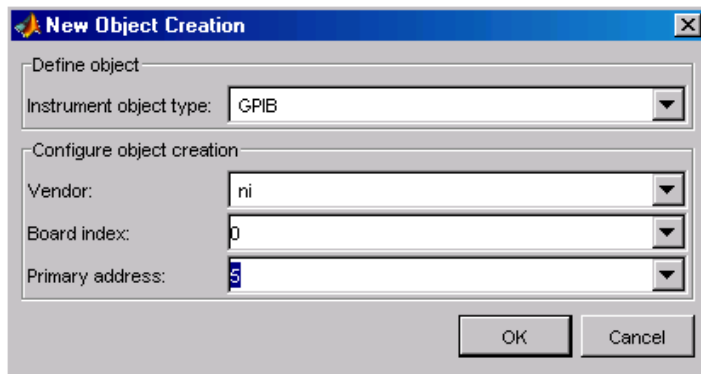
**9** Click **OK** to return to the Edit: Instrument1 dialog box, where the instrument object is now filled in and selected for this resource (`GPIB0-5`).
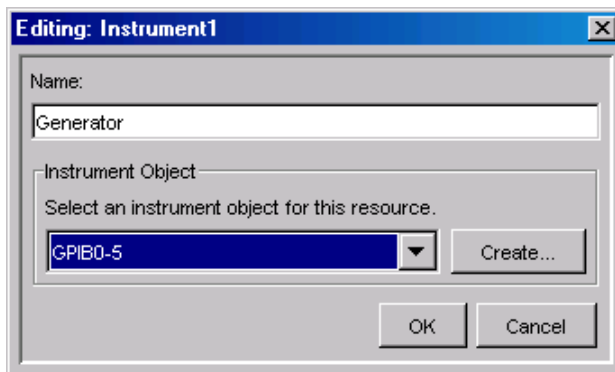
**10** Click **OK** to apply this resource and return to the **Properties** pane in the SystemTest desktop.

**11** In the **Command text** field, enter `frequency` followed by a space to separate the text from the variable that will follow. This is the command to set the frequency of

the 33120A generator, as described in the instrument's reference manual proved by the vendor.

**12** Click **Data source** and select `New Test Vector`. The name of the vector you create for setting the generated frequencies is called `genfreq`. In the Insert Test Vector dialog box, enter that text in the **Name** field, and set the **Expression** field to `[1500 5000 7500]`, including the brackets.
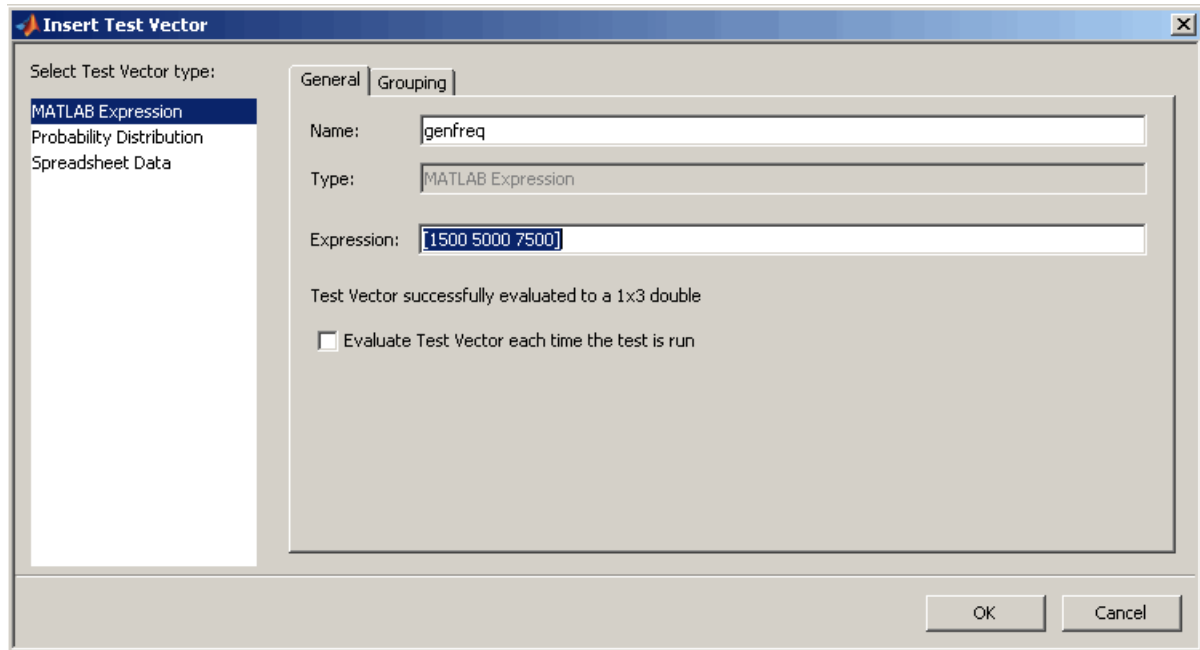


**13** Click **OK** to return to the SystemTest desktop.

Notice that the **Main Test** node in the tree now says `(3 Iterations)`. Because you entered three values in the test vector, the test is comprised of three iterations, one for each frequency value in the test vector.

**14** Keep the **Send variable data as** setting as `String`. The generator is expecting string values for its commands.

**15** Set a pause value of `2` seconds. This allows the generator to settle before you measure its output.

The element should now resemble the following figure:
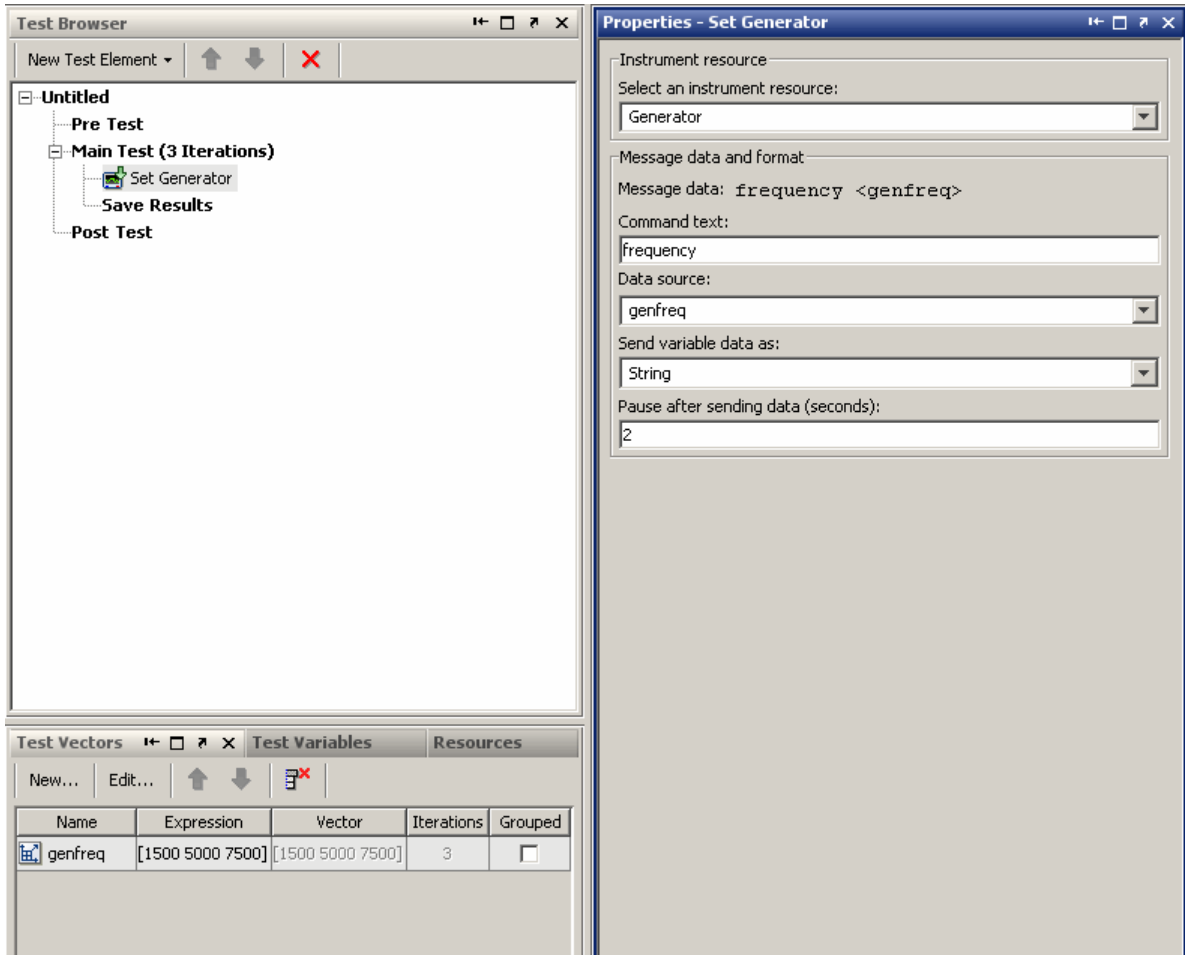
## Setting Up the Oscilloscope

You use a To Instrument element, which provides a one-way communication to the oscilloscope, to program the scope to measure frequency.

1   Add an element by clicking **New Test Element > Instrument Control > To Instrument**.

2   Double-click **To Instrument** in the tree, rename it Set Scope, and press Enter.

**3** As before, create a new instrument resource, but this time call it `Scope`. Create a new instrument object for it using Board index 0, and GPIB primary address 4.

**4** For the command text, enter `measurement:immed:type frequency`. This puts the scope in the frequency measurement mode, as described in the instrument's reference manual provided by the vendor.

There is no test variable or pause required for this element, so the element looks like the following figure:
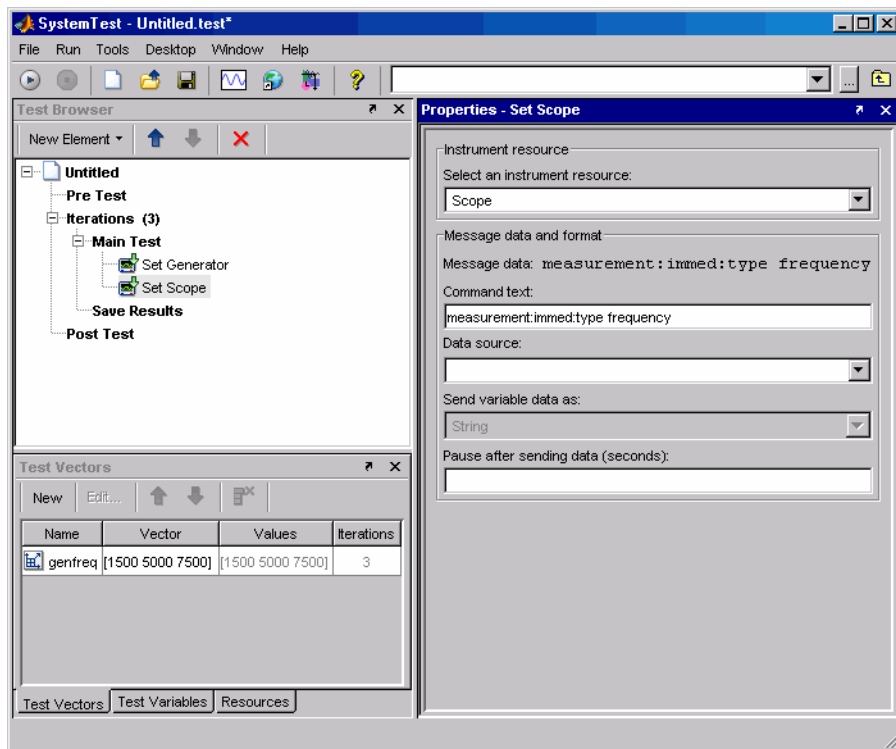


To see the resources you created for communications with your two instruments, click the **Resources** tab at the bottom of the SystemTest window. You can see the `Generator` and `Scope` resources, along with their GPIB settings.
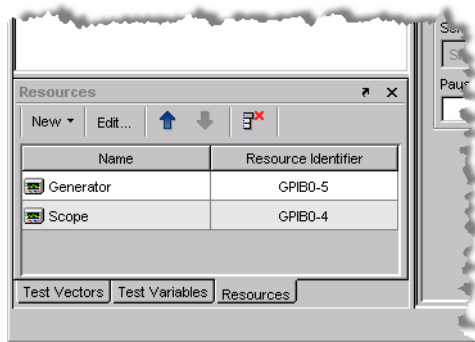
## Taking the Measurement

With the generator and scope set up, you can take the measurement with the scope using a Query Instrument element, which sends the command to the scope for taking the measurement.

1   Add an element by clicking **New Test Element** > **Instrument Control** > **Query Instrument**.

2   Double-click **Query Instrument** in the tree, rename it `Measure with Scope`, and press Enter.

3   Use the existing instrument resource called `Scope`, by selecting it in the **Instrument resource** list.

4   Enter the command to query for a measurement by typing `measurement:immed:value?` in the **Instrument query command** field.

5   Select **Store complete response**, and select the **Empty input buffer after read** check box.

6   From the **Interpret data as** list, select `String` (this scope returns ASCII strings), and select the **Convert string value to a numeric result** check box.

7   From the **Assign data to** list, select `New Test Variable`. For the oscilloscope's frequency measurement, name the test variable `scopefreq`. It needs no initial value.

The element now looks like the following figure:
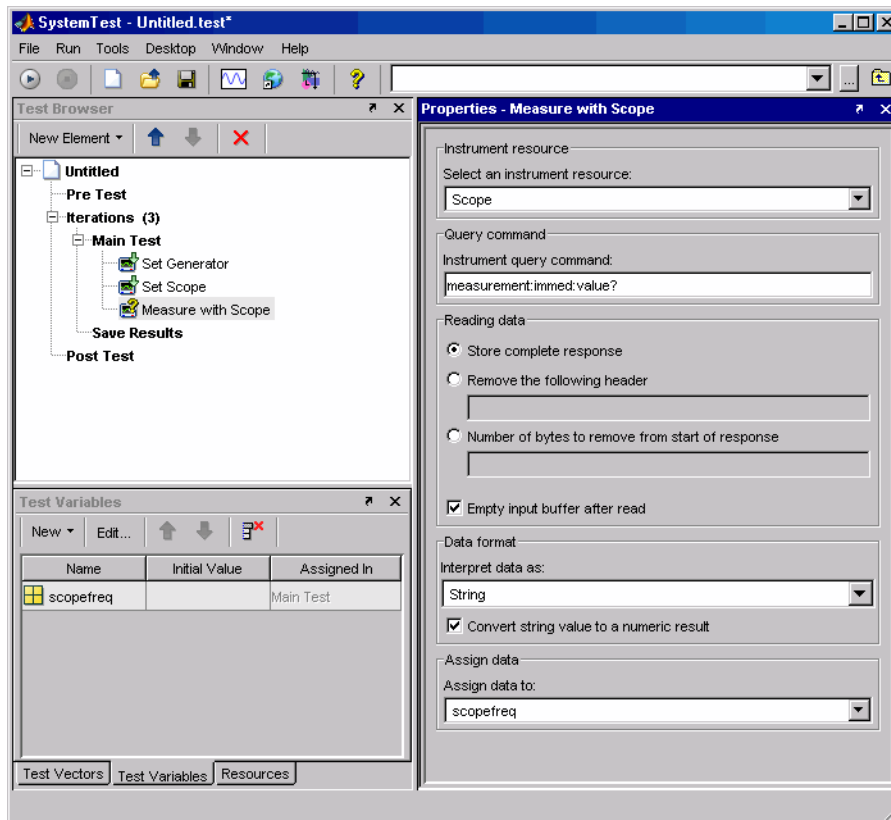
## Saving Test Results

To view the results of your test, you must first specify the test variables you want to save as test results. This is done in the **Save Results Properties** pane.

1. Click **Save Results** in the test browser tree.

2. In the **Properties** pane, click **New Mapping**.

3. From the **Test Variable** list, select scopefreq. This test variable contains the frequency measurements provided by the oscilloscope during each Main Test iteration, as shown in the following figure:
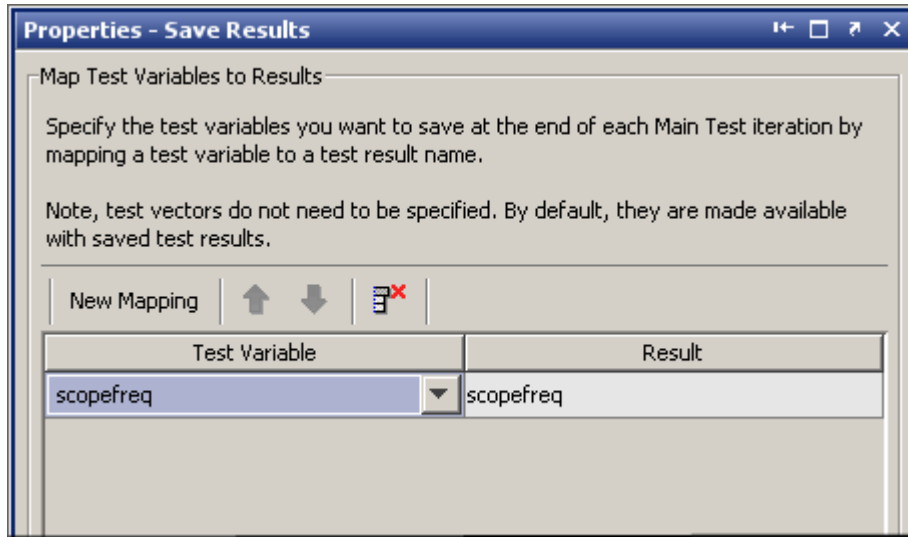
## Running the Test and Viewing Test Results

Now that the test elements are all created, you can run the test.

1   Run your test.

2   When the test is complete, click on the link in the **Run Status** pane to display your test results.

3   To see the measurement results, at the MATLAB prompt type

```
format short g
scopefreq
scopefreq =
        1501.5
          5000
          7500
```

This verifies that the signal generator is producing the expected signal frequencies.

# Use Data Acquisition Toolbox Elements

The Data Acquisition Toolbox software provides several elements to use in the SystemTest software.

- "Data Acquisition Toolbox Test Elements" on page 8-2
- "Test Voltage Regulator Using Data Acquisition Toolbox" on page 8-3

# Data Acquisition Toolbox Test Elements

The Data Acquisition Toolbox elements provide a way to bring analog and digital data from a data acquisition device into a SystemTest test, or to send analog or digital data from your device. You can use these elements along with the other elements in the SystemTest software to create tests for Simulink models and other applications.

**Note** To use the Data Acquisition Toolbox elements, you need a license for the Data Acquisition Toolbox software. These four elements will not appear in the SystemTest software without this license.

The Data Acquisition Toolbox software provides four elements that you can use in the SystemTest software:

- Analog Input — For reading analog data from your data acquisition device
- Analog Output — For sending analog data to your data acquisition device
- Digital Input — For reading digital data from your data acquisition device
- Digital Output — For sending digital data to your data acquisition device

You can configure each test element to communicate with your data acquisition devices for sending or receiving digital or analog data.

# Test Voltage Regulator Using Data Acquisition Toolbox

## Introduction

To illustrate how to use some of the Data Acquisition Toolbox test elements in the SystemTest software, this section provides a step-by-step example. The example shows how to use the elements that send data to a device under test (DUT) and receive data from a device under test, using both analog channels and digital lines.

This example samples the response of a 5-V voltage regulator that is stimulated with three different voltages of 4, 5, and 7.5 volts. The regulator has an enable function controlled by a digital signal. In this example, you collect 22,000 samples per second of the DUT response for 2 seconds.

All data going to and from the DUT is handled by a National Instruments® PCI-6035E data acquisition card. The example uses this card's analog output for the DUT stimulus, analog input for capturing the DUT response, and digital output for controlling the DUT's enable line. The test configuration is shown in the following figure:
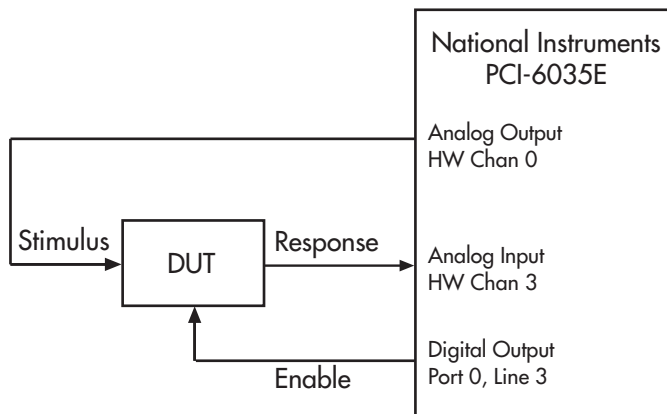
The following sections contain the steps in this example.

## Sending Analog Stimulus Data to the DUT

Stimulus data is sent to the DUT from an analog output channel of your data acquisition card.

1   Open the SystemTest software in MATLAB by typing `systemtest` at the MATLAB command line.

2   This example does not use the Pre Test section, so select the **Main Test** section in the **Test Browser** pane.

3   Add an Analog Output element by selecting **New Test Element** > **Data Acquisition** > **Analog Output**.
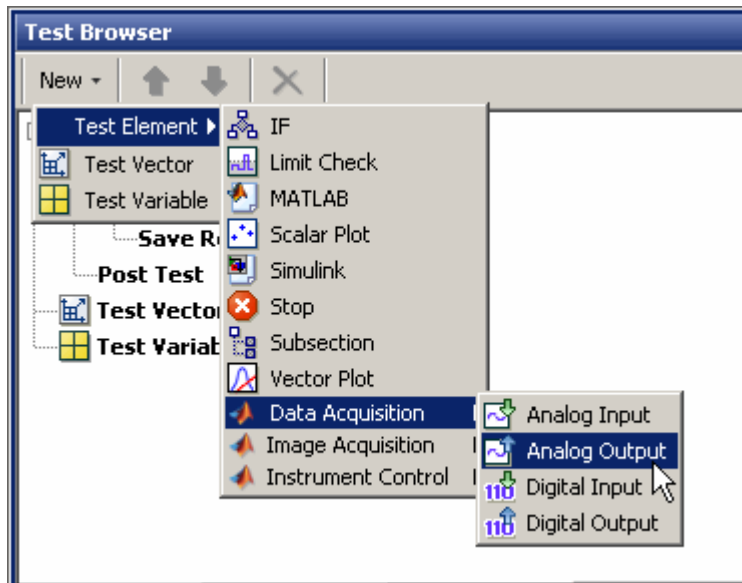
The new element appears in the browser tree, and its properties appear in the **Properties** pane. The SystemTest software scans your computer for installed data acquisition adaptors and devices. This can take several seconds.

4   Double-click the new `Analog Output` node in the browser tree, and enter a new name for this element, such as `Stimulate DUT`.

5   Since we have three test cases, we need to create a test vector containing the three voltage settings to test against. Click the **Test Vectors** tab. The voltage values for the stimulus to the DUT are held in a test vector. Click **New Vector** to create a new test vector.

6   In the Insert Test Vector dialog box, click the name `TestVector1` and enter a new name for your vector, such as `DUTstimulus`.

7   Click the default `1 : 1 : 10` entry in the **Expression** field, and replace it with the values for your test: `[4, 5, 7.5]` (be sure to include the brackets) and click **OK**. Notice that because there are three values in your vector, the browser tree now indicates that the Main Test will run three iterations. Each iteration will use one of the three values in the vector for the DUT stimulus voltage.

8   In the **Properties** pane, select the adaptor and device to use for the test. This example uses the `nidaq` adaptor, and the device is a `PCI-6035E`.

9   The example hardware configuration uses the card's analog output hardware channel 0 to provide the DUT's stimulus. So select the check box for this channel. The element will generate signals of 4, 5, and 7.5 volts, so keep the default output range of `[-10.0 10.0]`.

10   From the **Data source** list, select the `DUTstimulus` test vector.

11   Enter a value of 1 for **Output rate**. You are using a single static value rather than a sampled waveform, so this is not critical.

12   Enter a value of 1 for **Number of times to output data**. The card will hold its last programmed value, so you need to send it only once.

## Enabling the DUT with Digital Data

To send a digital enable signal to the DUT, use a digital output element.

1   Select **New Test Element > Data Acquisition > Digital Output**.

2   Double-click the new Digital Output element in the browser tree, and type a new name for this element, such as `Enable DUT`.

3   Click the **Test Variables** tab.

4   Click the **New** button to create a new variable. You will create two variables: one for enabling and one for disabling the DUT.

5   Click the name `Var1`, and replace it with the text `DUTenable`.

6   Click its empty **Initial Value** entry, and enter `1`.

7   Repeat steps 4 to 6 to create a second test variable, but name it `DUTdisable` with an initial value of `0`.

8   In the **Properties** pane for the `Enable DUT` element, select the adaptor and device for sending this data. Again, you are using the `nidaq` adaptor, and the device is a `PCI-6035E`.

9   The hardware configuration uses the card's digital output port 0, line 3 for the enable signal, so select the check box for this line.

10   From the **Data source** list, select the variable `DUTenable`.

The **Properties** pane now looks like the following figure:
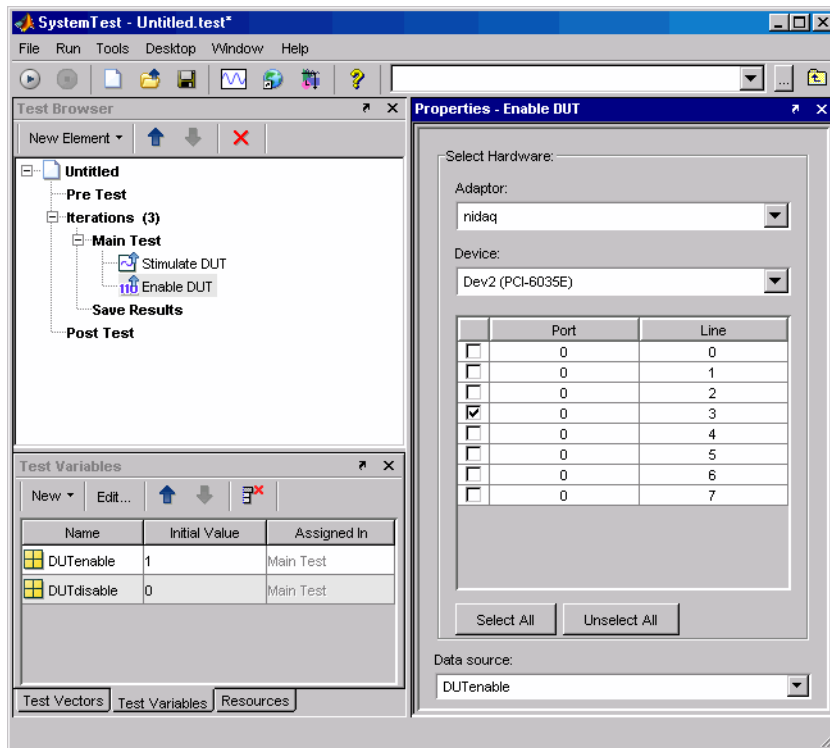
## Receiving Analog Response Data from the DUT

The next element in the test samples the output from the DUT and assigns the acquired data to a test variable.

1. Select **New Test Element > Data Acquisition > Analog Input**.

2. Double-click the new **Analog Input** element in the browser tree, and enter a new name for this element, such as DUT Response.

3. In the **Properties** pane, select the adaptor and device to use for the test. This example uses the `nidaq` adaptor, and the device is a `PCI-6035E`.

4. The hardware configuration uses the card's analog input hardware channel 3 to read the DUT's response, so select the check box for this channel. The expected signal will be about 5 volts, so keep the default output range of `[-10.0 10.0]`.

5   Set a sample rate of `22000`. Because of hardware limitations, the actual sample rate may not exactly match the value you specify.

6   In the **Acquire** field, specify to acquire data for `2 seconds`. Set `seconds` in the unit list to the right of the value field.

7   In the **Assign data to** field, select `New Test Variable` from the list. This is where you specify what test variable to assign the acquired data to. The Edit dialog box appears.
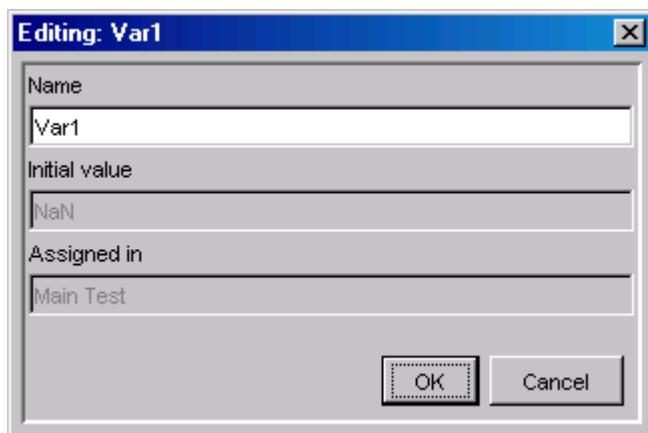


8   Enter a name for the test variable, such as `DUTresponse`, then click **OK** to create the test variable.

The **Properties** pane now looks like the following figure:
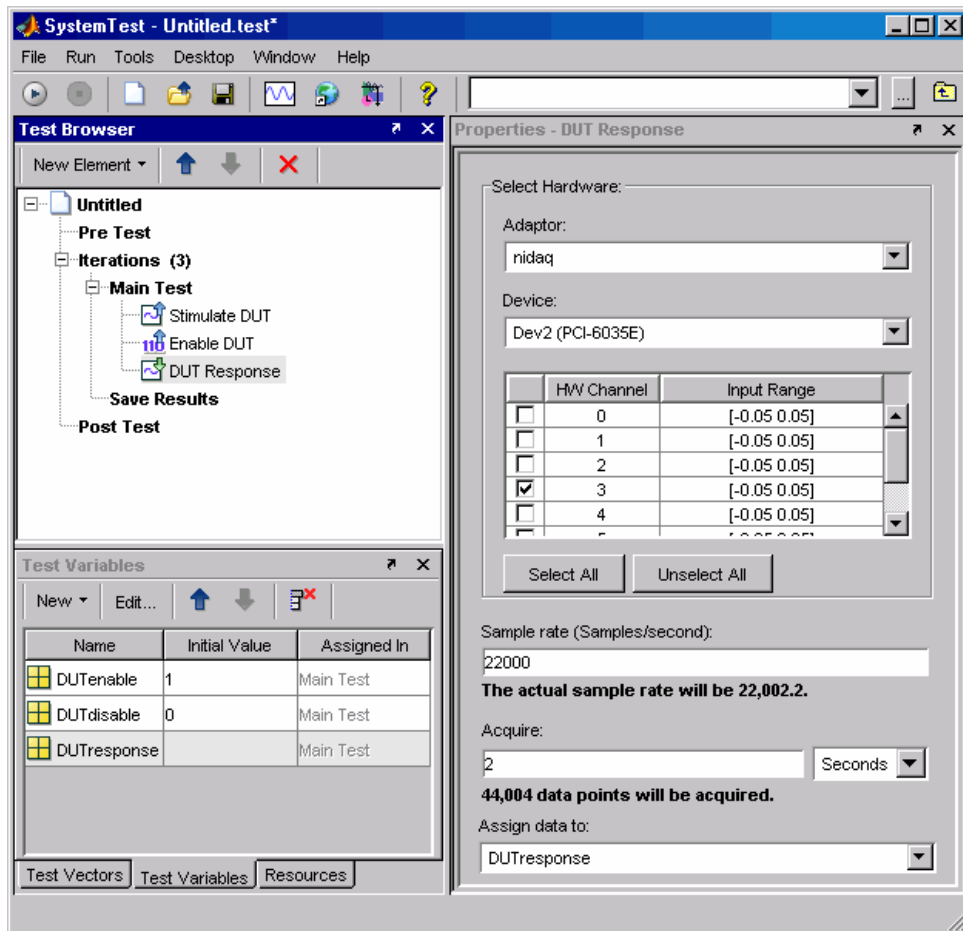
## Disabling the DUT with Digital Data

The next step is to disable the DUT with a digital output element that turns off the DUT's enable line. This element is similar to the Enable DUT element, except it sends a different value to the DUT.

1 Select **New Test Element > Data Acquisition > Digital Output**.

2   Double-click the new **Digital Output** element in the browser tree, and enter a new name for this element, such as `Disable DUT`.

You already created the test variable `DUTdisable`, which you will use in this element.

3   In the **Properties** pane for the `Disable DUT` element, select the adaptor and device for sending this data. Again, you are using the `nidaq` adaptor, and the device is a `PCI-6035E`.

4   The hardware configuration uses the card's digital output port 0, line 3 for the enable signal, so select the check box for this line.

5   From the **Data source** list, select the variable `DUTdisable`.

## Performing Data Analysis

At this stage, you might perform any analysis or visualization routines on the data generated by the DUT. You can do this in a MATLAB element.

1   Select **New Test Element > MATLAB**.

2   Double-click the new **MATLAB** element in the browser tree, and enter a new name for this element, such as `Process Data`.

3   In the **MATLAB Script** edit field of the **Properties** pane, enter any MATLAB code that you need for analyzing your test variables. You might be interested in measuring ripple, noise, regulation, or many other characteristics. You can access the DUT response by referring to the test variable `DUTresponse`. The stimulus data is available in the test variable `DUTstimulus`.

The following figure shows a MATLAB element with only some comments added in the **Properties** pane.
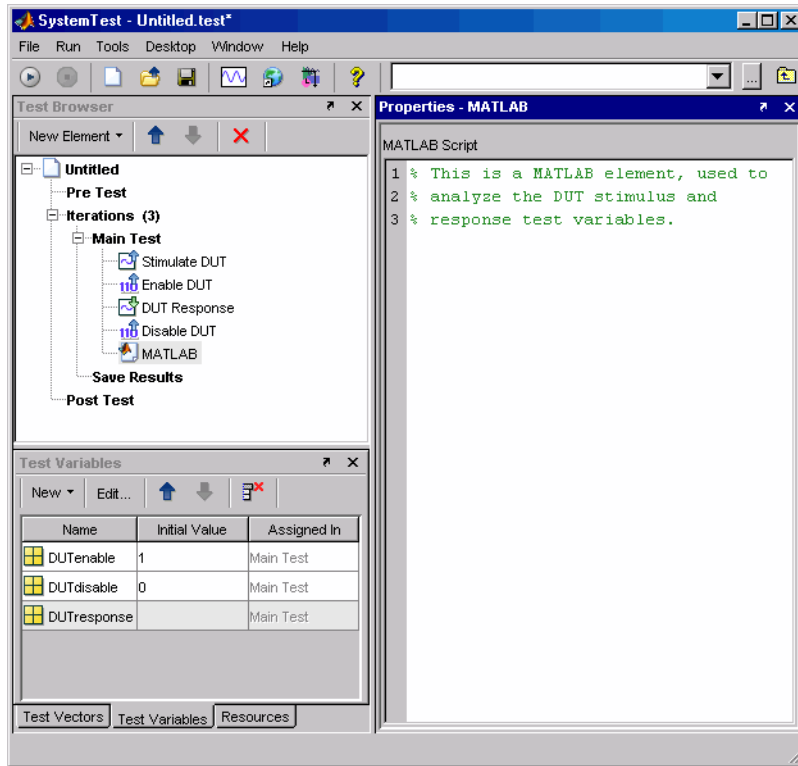
## Defining Post Test Elements

In this example, it is recommended to include an element in the Post Test section to disable the DUT.

1 Click the **Post Test** section in the browser tree.

2 Create a digital output element set up like the element you made in "Disabling the DUT with Digital Data" on page 8-9.

With the extra `Disable DUT` element, the test now looks like the following figure:
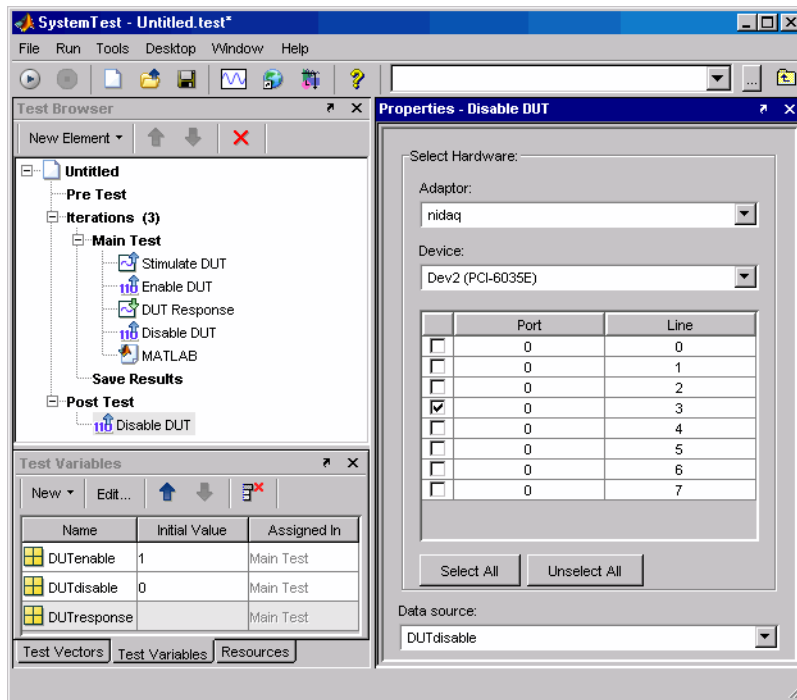
The Post Test section of the test could also perform any analysis that requires completion of all the iterations of the Main Test.

## Saving and Viewing Test Results

Before running a test, you must specify which test variables you want to save as a test result. In the **Save Results Properties** pane, you select the test variable that you want to save and map it to a test result name.

The SystemTest software allows you to view the results you have chosen to save for your test using a workspace variable called `stresults`. It provides access to the test results object, which is useful for comparing the results of separate test runs and for postprocessing test results. For more information, see "View Test Results at the Command Line".

# Use the Image Acquisition Toolbox Element

The Image Acquisition Toolbox software includes a SystemTest element that you can use to bring live video data into a SystemTest test.

# Image Acquisition Toolbox Element

The Image Acquisition Toolbox element, called Video Input, provides a way to acquire live video data in a SystemTest test. You can use this element along with the other elements in the SystemTest software to create tests for Simulink models and other applications.

To learn how to use the Image Acquisition Toolbox element in the SystemTest software, see "Acquire Test Data Using Image Acquisition Toolbox Element" on page 9-3.

---

**Note**  To use the Image Acquisition Toolbox element, you need a license for the Image Acquisition Toolbox software. The Video Input element will not appear in the SystemTest software if you do not.

---

# Acquire Test Data Using Image Acquisition Toolbox Element

| In this section... |
|---|
| "Add Video Input Element to a Test" on page 9-3 |
| "Save and View Test Results" on page 9-8 |
| "Run the Test" on page 9-9 |

## Add Video Input Element to a Test

This example illustrates how to use the Video Input element in the SystemTest software. The example uses the Video Input element to acquire a single frame of video for each iteration of the test and uses the MATLAB element to display the acquired image.

The first step is to add the element, as shown in this section. The two following sections contain the remaining steps.

To create a test using the Video Input element:

1 Open the SystemTest software by typing `systemtest` at the MATLAB command line.

2 In the SystemTest desktop, start to create your test by selecting **Main Test** and adding the Video Input element. In the **Test Browser**, click **New Test Element > Image Acquisition > Video Input**.
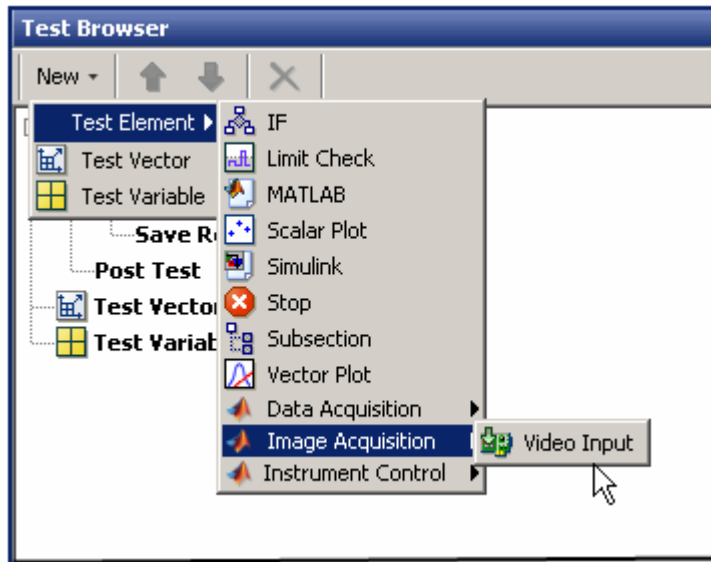
The SystemTest software adds the Video Input element to the Main Test section of the test and displays the **Properties** pane for the Video Input element. (You can also add elements to the Pre Test or Post Test sections of a test but this example does not require it.)

In the following figure, note the red x in the Video Input element icon in the Test Browser. This red x indicates that the element is in an error state. The SystemTest software outlines the required fields in red in the **Properties** pane.
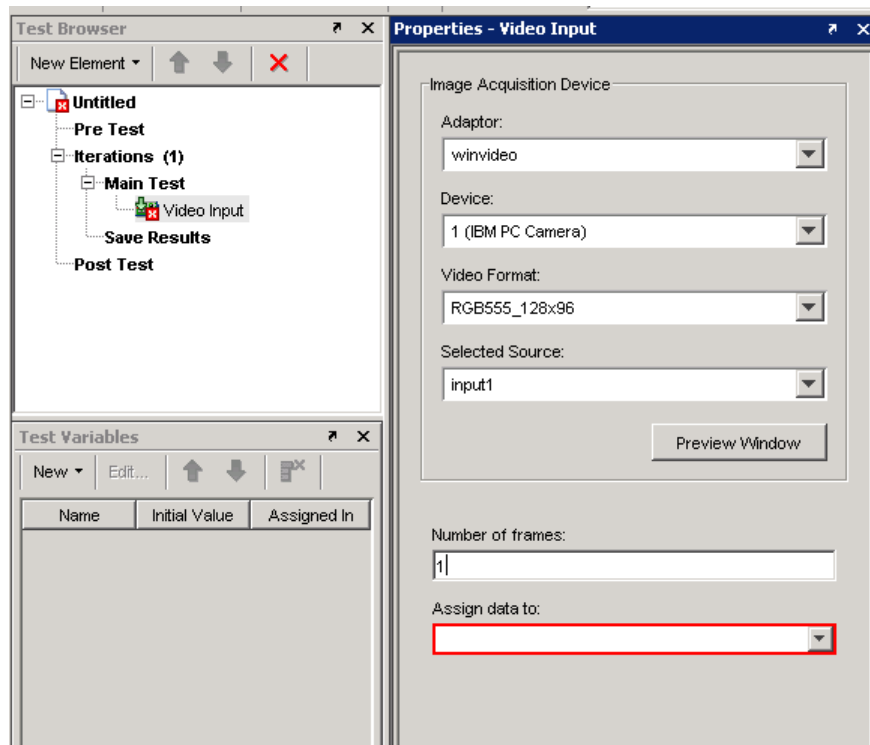
**3**  Specify the device you want to use to acquire image data in the **Properties** pane for the Video Input element. You must specify the name of the adaptor you want to use in the **Adaptor** field, which is a required field. (The SystemTest software uses red outlining to indicate required fields that are not filled in yet.) The SystemTest software can detect any image acquisition devices supported by the Image Acquisition Toolbox software that are connected to your system and fills in this field with a default value based on the alphabetical list of devices, if one is available. For our example, in the figure, the SystemTest software sets the **Adaptor** field to winvideo. If your system has other adaptors that can connect to devices, select the adaptor that you want to use from the **Adaptor** list.

After the **Adaptor** field is set, the SystemTest software fills in the **Device**, **Video Format**, and **Selected Source** fields with default values. The SystemTest software populates the drop-down lists associated with each field with all available options for the field. Adaptors can support multiple devices and devices can support multiple

formats. The SystemTest software preselects the default values for these fields but lists all available options in the drop-down lists associated with these fields. The following figure shows the list for the **Video Format** field:
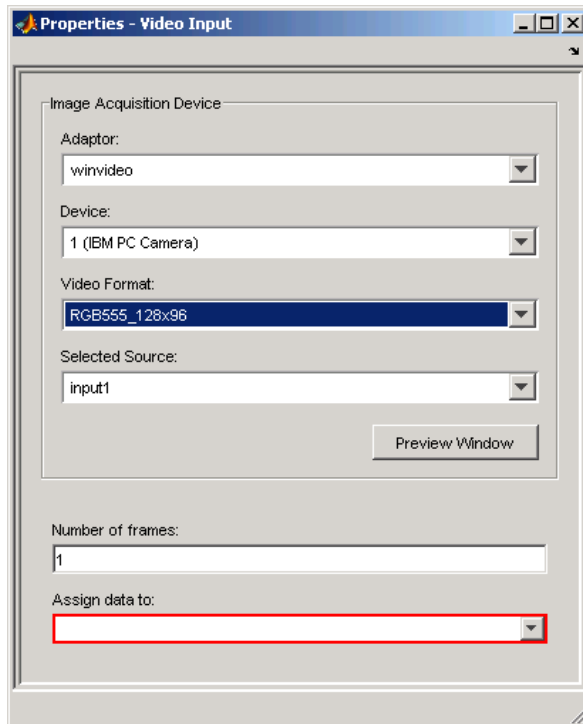


4  Specify the number of frames you want to acquire at each iteration of the test in the **Number of frames** field, which is a required field. For this example, we only need to acquire one frame for each iteration, so set this field to 1.

5  Specify the name of the SystemTest test variable that the acquired video data will be assigned to at each iteration. This is a required field. You can select a test variable from the list or create a new test variable by selecting **New Test Variable**.
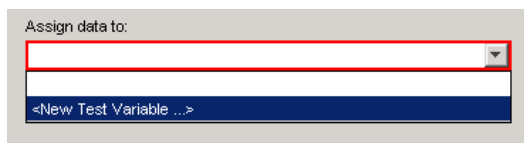
If you select **New Test Variable**, the SystemTest software opens the Edit dialog box. Assign a name to the test variable, or accept the default name, and click **OK**. You do not need to assign the test variable an initial value.
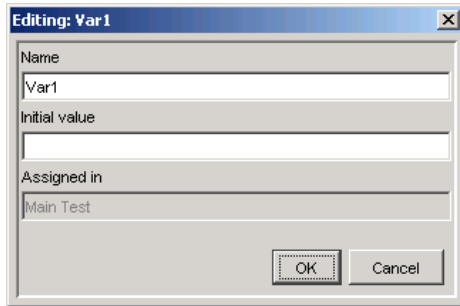


The SystemTest software adds the new test variable to the list in the **Test Variables** pane.

**6** Optionally, verify the Video Input element settings by clicking the **Preview Window** button. The SystemTest software opens a Video Preview window and displays a live video stream from your camera. You can use this to verify that your hardware is configured correctly. You should close the preview window before running the test.

**7** To complete this example test, add a MATLAB element to the Main Test section. In this MATLAB element, call the MATLAB `image` function to display the image frame acquired at each iteration.
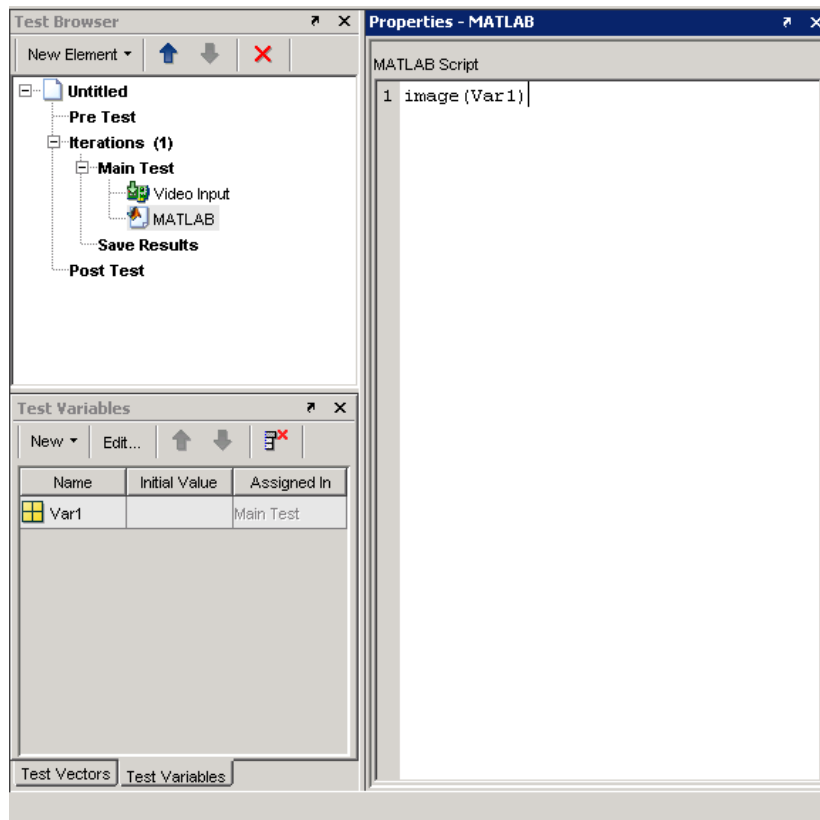
This completes this example test illustrating how to incorporate image data into the SystemTest software. In a real testing application, you can define test vectors that determine the number of iterations of your test that the SystemTest software performs. You can also compare test variables against defined limits in the Limit Check element and specify pass/fail criteria.

## Save and View Test Results

Before running a test, you must specify which test variables you want to save as a test result. In the **Save Results Properties** pane, you select the test variable that you want to save and map it to a test result name.

The SystemTest software allows you to view the results you have chosen to save for your test using a workspace variable called `stresults`. It provides access to the test results object, which is useful for comparing the results of separate test runs and for postprocessing test results.

For more information, see "View Test Results at the Command Line".
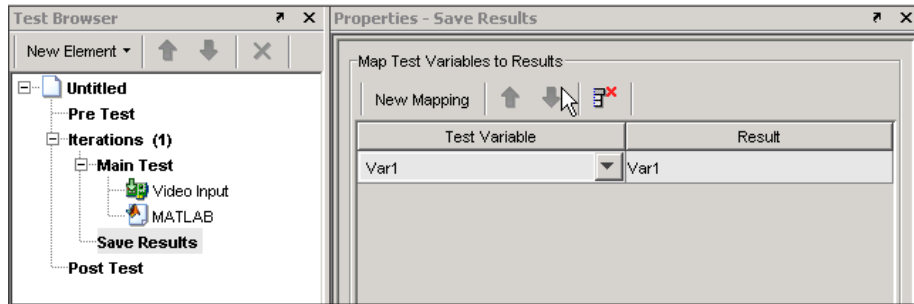


## Run the Test

To run the test, do one of the following:

- Click the **Run** button.
- Select **Run > Run**.
- Press the **F5** key.

While the test executes, the SystemTest software reports on the progress of the test in the **Run Status** pane.

# Distributing Tests Using Parallel Computing Toolbox Integration

# SystemTest Software and Parallel Computing Toolbox Integration

You can distribute SystemTest tests across multiple computers or processors. You can set up a test and then distribute Main Test iterations as tasks, which are performed concurrently by different workers. This can help speed up the total time the test takes to execute.

---

**Note:** To distribute tests in the SystemTest software, you need a license for the Parallel Computing Toolbox software.

---

You set up a distributed test as you would set up any test, using the SystemTest desktop. Then you use the **Distributed** tab on the **Test Properties** pane to set up the test distribution.

To access the distributed testing functionality in the SystemTest software, do one of the following:

- Select your test name in the **Test Browser**. This is the top node in the **Test Browser**, that lists the name you give the test when you save it, or "Untitled," if you have not saved it yet. Then click on the **Distributed** tab in the **Test Properties** pane.
- Select **Tools > Distributed Testing** on the SystemTest menu. This opens the **Distributed** tab.

Note that if you do not have the Parallel Computing Toolbox software installed, the tab displays a message indicating you cannot use the distributed testing functionality.

---

**Note:** To see a diagram that shows how distributed testing with the SystemTest software works and illustrates the relationship between the SystemTest software, the scheduler, and the workers, see "Run a Distributed Test" on page 10-13.

---

# Enable Distributed Testing

You must select the **Enable Distributed Testing** check box to distribute a test. Once enabled, the rest of the fields on the **Distributed** tab are activated.

The check box is not enabled by default on new tests. However, once you have set up a distributed test, if you save and close a test with the check box enabled, it will reload in the enabled state.

The **Main Test** node on the **Test Browser** indicates if your test is set up to be distributed. For example, if you have a distributed test containing 60 iterations, the node displays **Main Test (60 Iterations)** – *Distributed*, as shown in the following figure.

**Test Browser**                                    ⤴ ✕

New ▾    ⬆    ⬇    ✕

- □ **Simple_Demo**
  - **Pre Test**
  - □ **Main Test (60 Iterations)** - *Distributed*
    - 🔺 Create Scalar Data
    - 📊 Scalar Limit Check
    - ⚬ Plot Scalar Data
  - **Save Results**
  - **Post Test**
- □ 📊 **Test Vectors**
  - signal
- □ ⊞ **Test Variables**
  - Y
  - HiLimit
  - LowLimit

**Properties**        ⤴ ✕    **Test Vectors**          **Test Variables**

General | Distributed |

☑ Enable Distributed Testing

┌─ User Configuration ──────────────────────────┐
│  ⦿ Default (local)                              │
│  ○ [ local                               ▼ ]   │
└────────────────────────────────────────────────┘

File Dependencies | Path Dependencies |

Files or folders to be copied to the worker machines

┌──────────────────────────────────────────┐  [ Delete ]
│ <Double-click to enter or browse to create File Dependencies> │
│                                            │
│                                            │
│                                            │
│                                            │
│                                            │
│                                            │
└──────────────────────────────────────────┘

Italicized entries are dependencies from the User Configuration and cannot be edited

┌─ Distribution of main test runs across tasks ──────────┐
│ Calculated using 60 iterations                          │
│  ⦿ Default (2 iterations per task, 30 tasks)            │
│  ○ [ 1        ]  iterations per task, 60 tasks          │
└─────────────────────────────────────────────────────────┘

# Select a User Configuration

You or an administrator must set up a user configuration in the Parallel Computing Toolbox software before distributing tests in the SystemTest software. The user configuration determines certain administrative options, such as what scheduler is used. You can use the MathWorks job manager that comes with MATLAB Distributed Computing Server™, and the local scheduler that comes with the Parallel Computing Toolbox software. You can also use a third-party scheduler, such as Windows CCS, Platform Computing LSF, mpiexec, or a generic scheduler.

In the **User Configuration** field on the **Distributed** tab, select the user configuration that will be used when you distribute tests.

- The **Default** option indicates the configuration that is designated as the default in the Parallel Computing Toolbox software. The name of the configuration appears in parentheses.
- If you have any other configurations defined, they will appear in the drop-down list under **Default**. Either use the default, or click the second radio button and choose a user configuration from the list.

In the following example, this user has several different schedulers and has a separate user configuration for each scheduler. In this example, the user configurations are named for the schedulers they use.
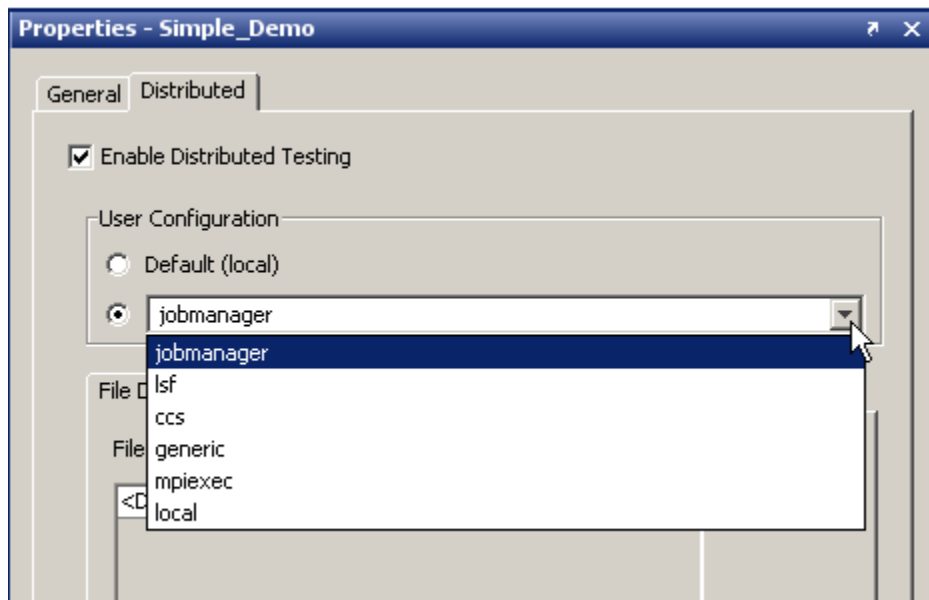


User configurations contain other information in addition to scheduler selection, and are used to define other distributed computing parameters. See Programming with User Configurations in the Parallel Computing Toolbox documentation for details on setting up the user configuration.

If you load a test containing a user configuration that no longer exists, this option will be in an error state. You can correct the error by selecting a valid user configuration.

# Set Up File Dependencies

Use the **File Dependencies** table to indicate files or folders of files to be copied to the worker machines. If the worker machines need to access files that your test is dependent on, you can add the names of the files or directories of files as dependencies in the SystemTest software and they will be copied to each worker.

**Note:** There is overhead in copying files for each task. If there are files that can be accessed from a shared location by the worker machines, use **Path Dependencies** instead. For example, if you use a Simulink element that references a large model available from a shared network folder, you should set a path dependency to the directory containing your model.

File dependencies can be defined in the **File Dependencies** table, as described below, or can be defined in the user configuration that is set up in the Parallel Computing Toolbox software. If there are any file dependencies specified by the currently selected user configuration, they will also be listed in this table, but will appear in italics and are not editable here. File names you enter through the SystemTest software appear in regular text and are editable here.

To set up a file dependency:

1   Click the **File Dependencies** tab within the **Distributed** tab.
2   Double-click the entry row in the table (the top row).

    The row becomes a text field.
3   Do either:

    · Type the full path and file or folder name in the field, and then press **Enter**.
    · Click the browse button in the entry row, browse to the file or folder, then click **Open** in the browse dialog box.

The dependency you entered then appears as a new row in the list.

The example below shows file dependencies for a MATLAB code file and a small model to be copied to the worker machines.
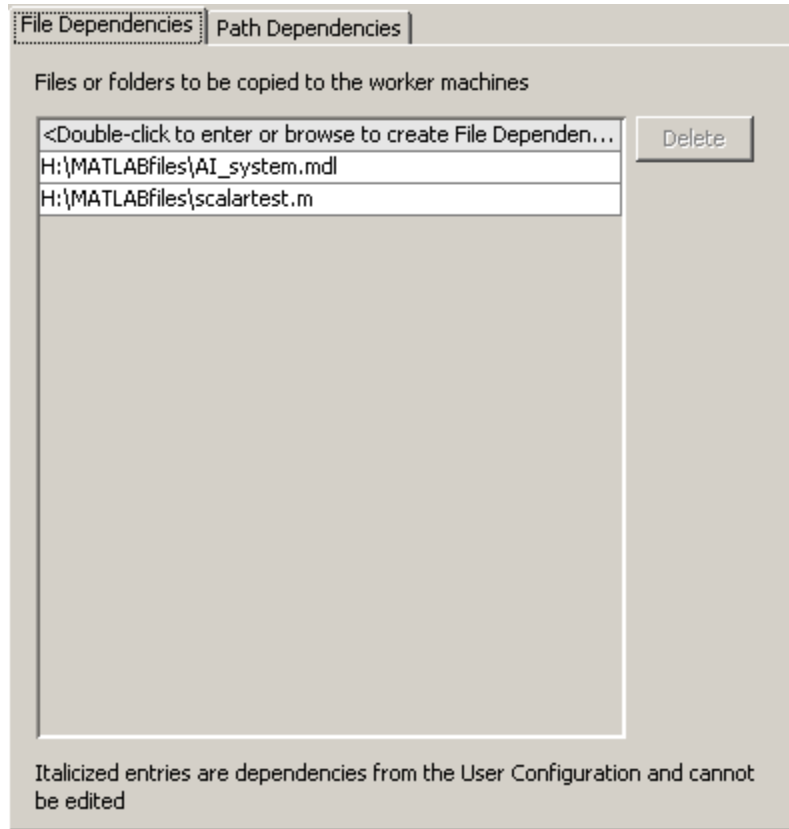


If you want to delete a file dependency, select it and click the **Delete** button. You can delete only dependencies added in the SystemTest software. You cannot delete any that are specified by the user configuration.

# Set Up Path Dependencies

Use the **Path Dependencies** table to indicate directories to be added to the workers' MATLAB path. If the worker machines need to access certain files during the test, you can add the directories here. These directories are added to the workers' MATLAB path such that the necessary files can be located. For example, if you use a Simulink element that references a large model available from a shared network folder, you should set a path dependency to the directory containing your model.

**Note:** If there are files that cannot be accessed from a shared location, use **File Dependencies** instead.

You can enter path dependencies in the **Path Dependencies** table, as described below, or in the user configuration that is set up in the Parallel Computing Toolbox software. If there are any path dependencies specified by the currently selected user configuration, they will also be listed on this tab, but will appear in italics and are not editable in the SystemTest software. Paths you enter through the SystemTest software appear in regular text and are editable here.

To set a path dependency:

1 Click the **Path Dependencies** tab within the **Distributed** tab.
2 Double-click the entry row in the table (the top row).

   The row becomes a text field.
3 Do either:

   • Type the path in the field, and then press **Enter**.
   • Click the browse button in the entry row, browse to the directory, then click **Open** in the browse dialog box.

The dependency you entered then appears as a new row in the list.

In the following example, because the model is very large the user set up a path dependency for the directory containing the model that the test uses.
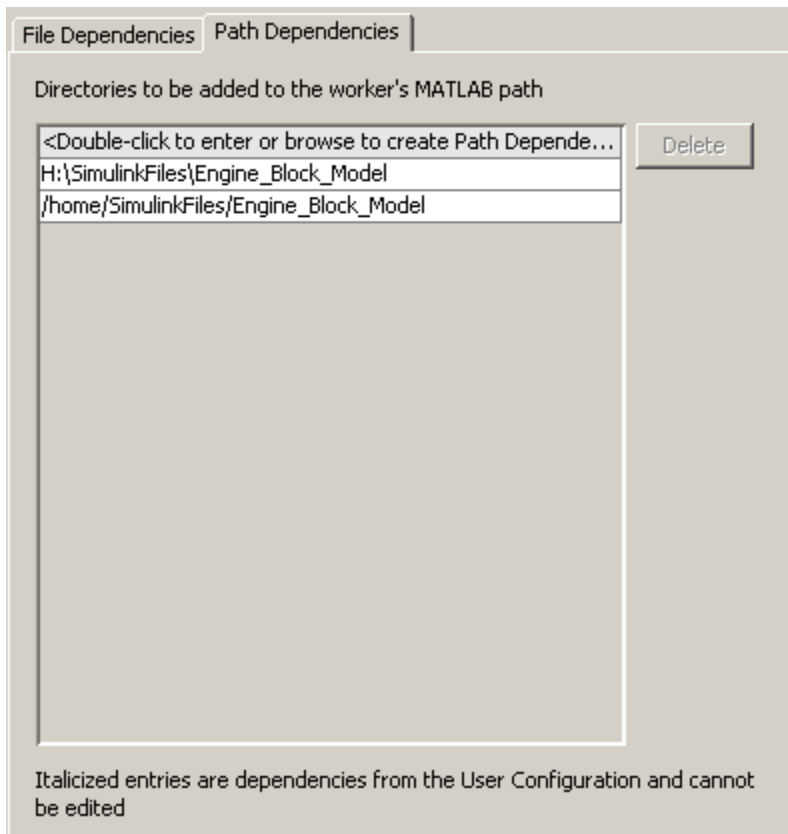


Notice in this example that the path is listed twice, once in Windows® format and once in UNIX® or Linux® format. If you have a heterogeneous cluster that contains both Windows and UNIX or Linux worker machines, you need to add the path twice so that all workers can use it.

---

**Note:** Path dependencies must be listed in the format supported by the type of worker machines the cluster contains, as shown in the previous figure (which shows both styles). Also, for Windows machines that cannot be directly accessed by all the workers, you need to specify the path as a UNC path.

---
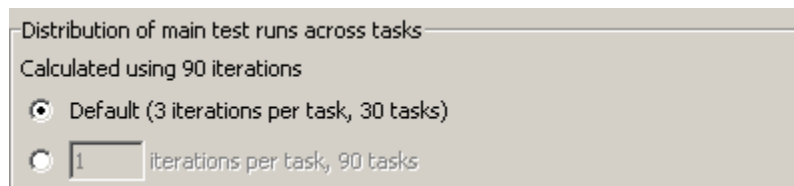
If you want to delete a path dependency, select it and click the **Delete** button. You can delete only dependencies added in the SystemTest software. You cannot delete any that were specified by the user configuration.

# Distribute Iterations Across Tasks

The **Distribution of main test runs across tasks** option on the **Distributed** tab determines the distribution of Main Test iterations into tasks that the workers perform. The calculation is based on the total number of iterations your test contains.

By default, the **Default** option is selected and the text in parentheses shows the number of iterations per task and number of tasks. The default is calculated by dividing the number of iterations in your test by 32 (an approximation based on a setup of 8 workers, with a target of 4 tasks per worker), and using the closest number to that. For example, if your test has 90 iterations, the default will be 3 iterations per task and 30 tasks, as shown below.
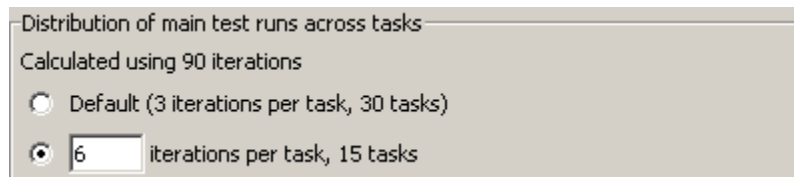


If you run your test and this does not seem efficient, you can change the number of iterations per task and number of tasks. To change it:

1   Select the second option. The number field becomes editable.

2   Enter the number of iterations per task you want to use.

3   Press **Enter**.

The number of tasks is then calculated (total number of iterations divided by number of iterations per task) and shown in parentheses. For example, if you had the same test with 90 iterations, but changed iterations per task to 6, you get 15 tasks.

# Run a Distributed Test

You run a distributed test as you would run any other test, by clicking the **Run** button in the SystemTest toolbar.

When you run a distributed test:

- Pre Test executes once, on the client machine (the machine from which you run the test).
- Main Test iterations execute on the cluster of worker machines defined by the user configuration.
- The SystemTest software waits for the distributed test to complete.
- If there are errors when the distributed test iterations run, only the first error from the tasks will be reported to the **Run Status** pane in the SystemTest software once all tasks have completed.
- At the end of each Main Test iteration, test results are saved and returned to the client machine once all Main Test iterations have finished executing.

**Note:** Because Main Test iterations run across a number of tasks, there is no guarantee as to the order the tasks (Main Test iterations) will execute. Tests should not be written with the assumption that iterations will execute in a fixed order.

Also, because Pre Test runs on the client machine, and tasks run independent of each other, Main Test iterations should not rely on data persisting across multiple iterations.

- Post Test executes once on the client machine, after Main Test executes or has errored while running.
- Test execution reports are generated at the end of the test, if enabled by the test.
- Generated plots are not shown on the client machine while the test runs, but are captured and displayed in the Test Report. Note that plots generated on worker machines will only reflect information generated as part of the task. Plotting multiple data points or lines on a single plot will only reflect the data pertaining to iterations executed as part of a single task.

Note that MATLAB and the SystemTest software remain in a busy state until the distributed test is done running or is stopped.

**Caution** It is recommended that you do not run a test containing hardware-related elements in distributed mode. That includes the Image Acquisition Toolbox element, the Data Acquisition Toolbox elements, and the Instrument Control Toolbox elements. These elements will likely error out because the connected hardware will not be available on the workers.

The following diagram illustrates the relationship between the SystemTest software, the scheduler, and the workers. Task 1 to Task X and Iteration 1 to Iteration N are determined by what is shown in the **Distribution of main test runs across tasks** section in the **Distributed** tab. For example, if the **Distribution of main test runs across tasks** for a test with 90 iterations is set to **Default (3 iterations per task, 30 tasks)**, that means your test will execute 3 iterations for each of 30 tasks. In this case, Task 1 might run iterations 1, 2, and 3, and Task 2 might run iterations 4, 5, and 6, etc.
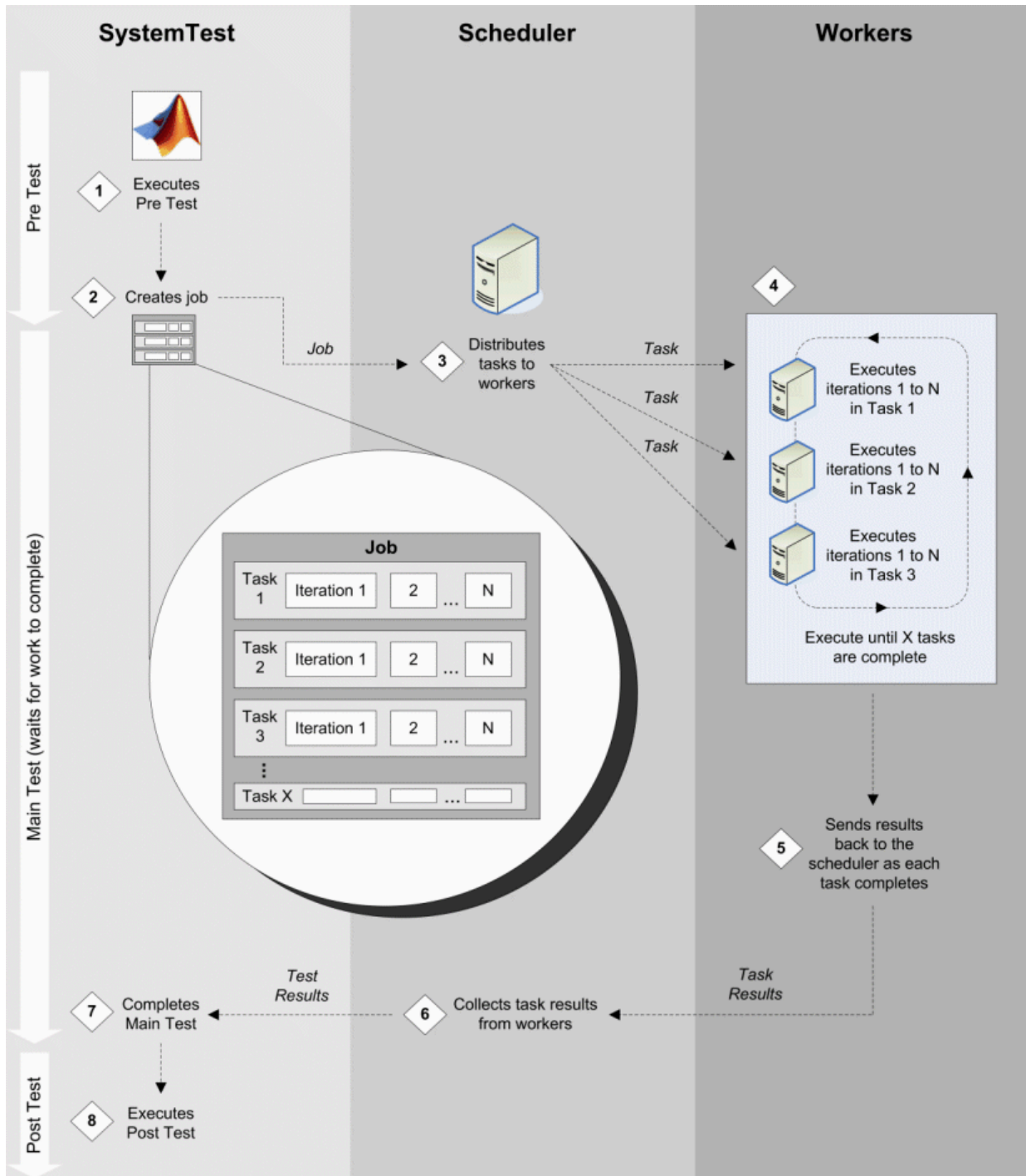
# Distribute a Test

The following general example shows how you can distribute any test you have created.

You create and set up a distributed test as you would set up any test, using the SystemTest desktop. If you determine that the test takes a long time to execute, you may benefit from distributing it. You then use the **Distributed** tab on **Test Properties** to set up the test distribution.

To distribute a test:

1   Select your test name in the **Test Browser**. Then click the **Distributed** tab in the **Test Properties** pane.

2   Select the **Enable Distributed Testing** check box to enable distributed testing and activate the other options on the tab.

3   The SystemTest software uses the user configurations set up in the Parallel Computing Toolbox software. User configurations identify various settings, such as which scheduler to use.

   In the **User Configuration** section, keep the default user configuration, or select the second radio button and choose a different configuration from the drop-down list.

   For more details, see "Select a User Configuration" on page 10-5.

4   If your test is dependent on files, such as models, MATLAB code files, or MAT-files, in order to execute, you need to specify the dependent files so that the worker machines can access the files while the test is running.

   If there are files that need to be copied onto the worker machines, use the **File Dependencies** tab. If there are files available on a shared network location that need to be accessed by the worker machines, use the **Path Dependencies** tab instead. For example, if you use a Simulink element that references a large model available from a shared network folder, you should set a path dependency to the directory containing your model.

   Enter the necessary file or path dependencies into the respective tabs by double-clicking the top row in the tables. For more details, see "Set Up File Dependencies" on page 10-7 and "Set Up Path Dependencies" on page 10-9.

5   The SystemTest software will calculate number of iterations per task for you, or you can specify that, in the **Distribution of main test runs across tasks** section.

Use **Default**, or change it by selecting the second option, which enables the number field. Enter the number of iterations per task you want to use and press **Enter** or click outside the field.

For details on how these values are calculated, see "Distribute Iterations Across Tasks" on page 10-12.

6   Run the distributed test as you would run any other test, by clicking the **Run** button in the SystemTest toolbar.

For information on what happens when you execute a distributed test, see "Run a Distributed Test" on page 10-13.

# Access Test Results from MATLAB Command Line

# View Test Results at the Command Line

| In this section... |
| --- |
| "Introduction" on page 11-2 |
| "Accessing the Results Summary" on page 11-2 |
| "Accessing the dataset Array" on page 11-5 |

## Introduction

After you run a test, the SystemTest software will automatically populate the MATLAB workspace with a variable called `stresults`. This variable provides access to the test results object, which is useful for comparing the results of separate test runs and for postprocessing test results.

## Accessing the Results Summary

You access the results using the `stresults` variable. To see an example, use the Fault Tolerant Fuel Control System model.

1  To open the model in the SystemTest software, type the following at the MATLAB command line:

   `systemtest demosystest_fuelctrl`

2  Run the test by clicking the **Run** button on the SystemTest toolbar.

3  To view the results after the test runs, return to MATLAB and type:

   `stresults`

The test results object looks like the following for the Fault Tolerant Fuel Control System example:

```
stresults =

    Test Results Object Summary for 'demosystest_fuelctrl':

            NumberOfIterations: 96
               TestVectorNames: EGOSensor, EngineSpeed, MAPSensor, SpeedSensor,
                                ThrottleSensor
             SavedResultNames: AvgAirFuel, AvgFuelRate, NSensorsActive,
                                SimAFRatio, SimFuelRate
                 ResultsDataSet: [96x10 dataset]

    Artifacts associated with this test result object:
       TEST-File (demosystest_fuelctrl.test)
       Test Report (demosystest_fuelctrl_report.html)

>>
```

The summary shows the number of iterations that ran, the names of the test vectors included in the test, the saved results you specified in **Save Results**, the dataset array, and generated artifacts.

`NumberOfIterations` reflects how many iterations actually executed when the test ran. This will match what is reflected in the SystemTest software in the **Main Test** node of the **Test Browser** if all iterations ran. If any iterations stopped or errored out, this will show only the number that did execute.

`TestVectorNames` is a 1-by-N string cell array containing the test vector names. The values are an alphabetical list of test vector names.

`SavedResultNames` is a 1-by-N string cell array containing the test result names. The values are an alphabetical list of test result names.

`ResultsDataSet` is the `dataset` array storing the test vector and test result values for each iteration. See "Accessing the dataset Array" on page 11-5 for information on accessing the test results data.

`Artifacts` provides links to SystemTest-generated documents, such as the test report. You can open the report by clicking the link. If your test includes a model coverage report, that would also be included here.

Note that in the example shown here, the Test Report was enabled before the test was run, so the link is displayed in the results. By default the report is not enabled. To see the link to the report in this example (or any test you run), enable the report before running the test. To enable the report, click on the test name in the **Test Browser**, then select the **Generate report** option on the **Output Files** tab of the **Properties** pane.

### Accessing Properties of the Test Results Object

You can see a complete list of test results object properties before looking at the actual test results data. At the command line, type:

```
get(stresults)
```

In the example using the Fault Tolerant Fuel Control System model, you see the following properties:

```
>> get(stresults)
         ResultsDataSet: [96x10 dataset]
     DerivedResultNames: {}
      NumberOfIterations: 96
        SavedResultNames: {'AvgAirFuel'  'AvgFuelRate'  'NSensorsActive'  'SimAFRatio'  'SimFuelRate'}
               StartTime: [2010 1 5 18 7 26.7400]
                StopTime: [2010 1 5 19 1 4.2210]
                TestFile: 'C:\MATLAB\demosystest_fuelctrl.test'
         TestVectorNames: {'EGOSensor'  'EngineSpeed'  'MAPSensor'  'SpeedSensor'  'ThrottleSensor'}
               Artifacts: {'TEST-File'   'C:\MATLAB\demosystest_fuelctrl.test'}
                     Tag: ''
                UserData: []
                Grouping: {''  ''  ''  ''  ''}

>>
```

In addition to information that is also included in the summary, this includes derived results names, start time, stop time, tags, user data, and grouping information.

`DerivedResultNames` contains values if you created any derived results using the Test Results Viewer. In the previous example there are no derived results, so the value is {}. If there were derived results, this property would contain an alphabetical list of their names.

`StartTime` provides the time the test was started in the form of a MATLAB clock vector.

`StopTime` provides the time the test was stopped in the form of a MATLAB clock vector.

`TestFile` stores the full path and name of the test that generated the test results. If the test has been saved, the value will contain the full path and name of the test. If the test has not yet been saved, the value will show only the test name.

`Tag` displays any string you specified using the `set` function. It is a descriptive string used for labeling purposes. By default, this property is empty.

`UserData` is a property for storing user data. It is used to store any arbitrary MATLAB data you would like to associate with the test results object. By default, this property is empty.

`Grouping` displays information about grouping of the test vectors. If you assign any test vectors to groups (using the **Grouping** tab on the **Test Vectors** pane), then the groups are listed here.

## Accessing the dataset Array

The `ResultsDataSet` property contains the test results data in the form of a `dataset` array. This is what you set up using the **Saved Results** node in the **Test Browser**. See "Save Test Results" on page 1-30 for more information on setting up saved results.

To access the test results data:

1   After running a test, use the `stresults` variable to view the test results object summary, as described in the previous section.

2   To access the `ResultsDataSet` property, type:

```
stresults.ResultsDataSet
```

or

```
get(stresults, 'ResultsDataSet')
```

This returns the test results data in the form of a dataset array.

In the Fault Tolerant Fuel Control System example, a portion of the test results data looks like this:

```
>> stresults.ResultsDataSet

ans =

        EGOSensor    EngineSpeed    MAPSensor    SpeedSensor    ThrottleSensor
   I1    [1]          [300]          [1]          [1]            [1]
   I2    [0]          [300]          [1]          [1]            [1]
   I3    [1]          [400]          [1]          [1]            [1]
   I4    [0]          [400]          [1]          [1]            [1]
   I5    [1]          [500]          [1]          [1]            [1]
   I6    [0]          [500]          [1]          [1]            [1]
   I7    [1]          [600]          [1]          [1]            [1]
   I8    [0]          [600]          [1]          [1]            [1]
   I9    [1]          [700]          [1]          [1]            [1]
   I10   [0]          [700]          [1]          [1]            [1]
```

In the `dataset` array, each row represents a test iteration, labeled using the convention of `['I' + Iteration_Number]`. The previous example shows the first 10 iterations. Test vector values are listed first, in alphabetical order, as shown, followed by test results, listed in alphabetical order, as shown in the following figure.

```
   I95    [1]          [800]          [0]          [0]            [0]
   I96    [0]          [800]          [0]          [0]            [0]


        AvgAirFuel    AvgFuelRate    NSensorsActive    SimAFRatio
   I1    [14.4466]     [1.3302]       [4]               [4098x1 Simulink.Timeseries]
   I2    [11.8858]     [1.6251]       [3]               [4098x1 Simulink.Timeseries]
   I3    [14.4283]     [1.5517]       [4]               [4084x1 Simulink.Timeseries]
   I4    [11.7511]     [1.9158]       [3]               [4084x1 Simulink.Timeseries]
   I5    [14.4281]     [1.6298]       [4]               [4067x1 Simulink.Timeseries]
   I6    [11.6776]     [2.0261]       [3]               [4067x1 Simulink.Timeseries]
   I7    [14.4196]     [1.6302]       [4]               [4005x1 Simulink.Timeseries]
   I8    [11.6281]     [2.0346]       [3]               [4005x1 Simulink.Timeseries]
   I9    []            [0.0020]       [4]               [4009x1 Simulink.Timeseries]
   I10   []            [0.0020]       [3]               [4009x1 Simulink.Timeseries]
```

Notice that this example shows the test vectors list for the last two iterations (I95 and I96), and the beginning of the display of the test result values. There are five results,

shown in alphabetical order. The display wraps in MATLAB, so the fifth result is shown after all the iterations for the first four.

In this example, the value for `AvgAirFuel` is `14.4466` for the first iteration, `11.8858` for the second iteration, etc.

# Manage Test Results

| In this section... |
| --- |
| "Introduction" on page 11-8 |
| "Manage Test Results in Native Format" on page 11-8 |
| "Manage Test Results as a Dataset Array" on page 11-9 |
| "Plot Results Data" on page 11-10 |

## Introduction

After accessing test results data in the form of a `dataset` array, you can work with the data in MATLAB. This feature is useful for comparing the test results data of separate test runs and for postprocessing of test results data.

One advantage to accessing test results data at the command line is that all of the MATLAB plotting tools are available to use on the test results data. You can plot the data using any of the plot types MATLAB offers.

Another major use of the `datatset` array is to quickly see the results when you use a Limit Check element in your test. You can see whether each iteration passed or failed, and what the value was.

## Manage Test Results in Native Format

You can use indexing to extract data of the dataset in its native format. You can index by string or value.

For example, you can assign a variable to represent the dataset, then access one column of the set using that variable. In the case of the Fault Tolerant Fuel Control System example, it could look like the following.

1   Create a variable to refer to the test results dataset array:

```
SetA = stresults.ResultsDataSet;
```

In this example the test results data is assigned to the variable `SetA`.

2   Specify the desired columns of data by referencing the name of the test result.

```
SetA.AvgFuelRate
```

This indexed into the column called `AvgFuelRate`.

---

**Note:** When extracting data in its native format, the test results are always returned as a cell array.

---

MATLAB displays the contents of that column of data, as shown in this example:

```
>> SetA = stresults.ResultsDataSet;
>> SetA.AvgFuelRate

ans =

    [1.3302]
    [1.6251]
    [1.5517]
    [1.9158]
    [1.6298]
    [2.0261]
    [1.6302]
    [2.0346]
    [0.0020]
    [0.0020]
```

The first 10 iterations are shown in the example.

## Manage Test Results as a Dataset Array

You can also choose to manage the test results as a dataset array, refining the data as finely as needed. Suppose you just want to get the average fuel rate for iterations 4 through 8. Use standard MATLAB indexing, as shown in the next example:

```
>> SetA(4:8, 'AvgFuelRate')

ans =

            AvgFuelRate
    I4      [1.9158]
    I5      [1.6298]
    I6      [2.0261]
    I7      [1.6302]
    I8      [2.0346]
```

The value returned represents the average fuel rate for iterations 4 through 8, in the form of a dataset array.

## Plot Results Data

To demonstrate plotting results:

**1** Open the example in the SystemTest software by typing the following at the MATLAB command line:

```
systemtest simple_demo
```

**2** Run the test by clicking the **Run** button on the SystemTest toolbar.

**3** View the results summary using `stresults` at the command line.

```
>> systemtest simple_demo
>> stresults

stresults =

    Test Results Object Summary for 'Simple_Demo':

            NumberOfIterations: 60
              TestVectorNames: signal
             SavedResultNames: HiLimit, LowLimit, Y
               ResultsDataSet: [60x4 dataset]

    Artifacts associated with this test result object:
       TEST-File (Simple_Demo.test)
       Test Report (Simple_Demo_report.html)
```

You can see that this test has one test vector for a signal, called `signal`, and three saved results. The result for Y is the signal's value for a given test run.

Note that in the example shown here, the Test Report was enabled before the test was run, so the link to the report is displayed in the results. By default the report is not enabled. To see the link to the report in this example (or any test you run), enable the report before running the test. To enable the report, click on the test name in the **Test Browser**, then select the **Generate report** option on the **Output Files** tab of the **Properties** pane.

**4** Look at the test results dataset by typing the following:

```
stresults.ResultsDataSet
```

The first 10 iterations are shown here:

```
>> stresults.ResultsDataSet

ans =

          signal            HiLimit     LowLimit      Y
    I1     [ 0.2094]         [1]         [-1]       [ 0.5226]
    I2     [ 0.4189]         [1]         [-1]       [ 0.8125]
    I3     [ 0.6283]         [1]         [-1]       [ 0.2148]
    I4     [ 0.8378]         [1]         [-1]       [ 1.1565]
    I5     [ 1.0472]         [1]         [-1]       [ 0.9984]
    I6     [ 1.2566]         [1]         [-1]       [ 0.5486]
    I7     [ 1.4661]         [1]         [-1]       [ 0.7730]
    I8     [ 1.6755]         [1]         [-1]       [ 1.0414]
    I9     [ 1.8850]         [1]         [-1]       [ 1.4086]
    I10    [ 2.0944]         [1]         [-1]       [ 1.3309]
```

You can see the test vector `signal` followed by the three results, including the one of interest in this example, `Y`.

5   Create a variable called `SetB` for the results dataset for ease of use in working with the data.

```
SetB = stresults.ResultsDataSet;
```

6   Create variables for the signal (the test vector) and the `Y` test result.

```
signalA = SetB.signal;
VarA = SetB.Y;
```

7   Plot the signal. Because `Y` represents the current value of the signal for each iteration of the test, plotting the signal against `Y` shows the values of the signal throughout the test.

```
plot([signalA{:}], [VarA{:}])
```

The `plot` command produces a line plot, as shown here. You can use any type of plot that MATLAB offers.
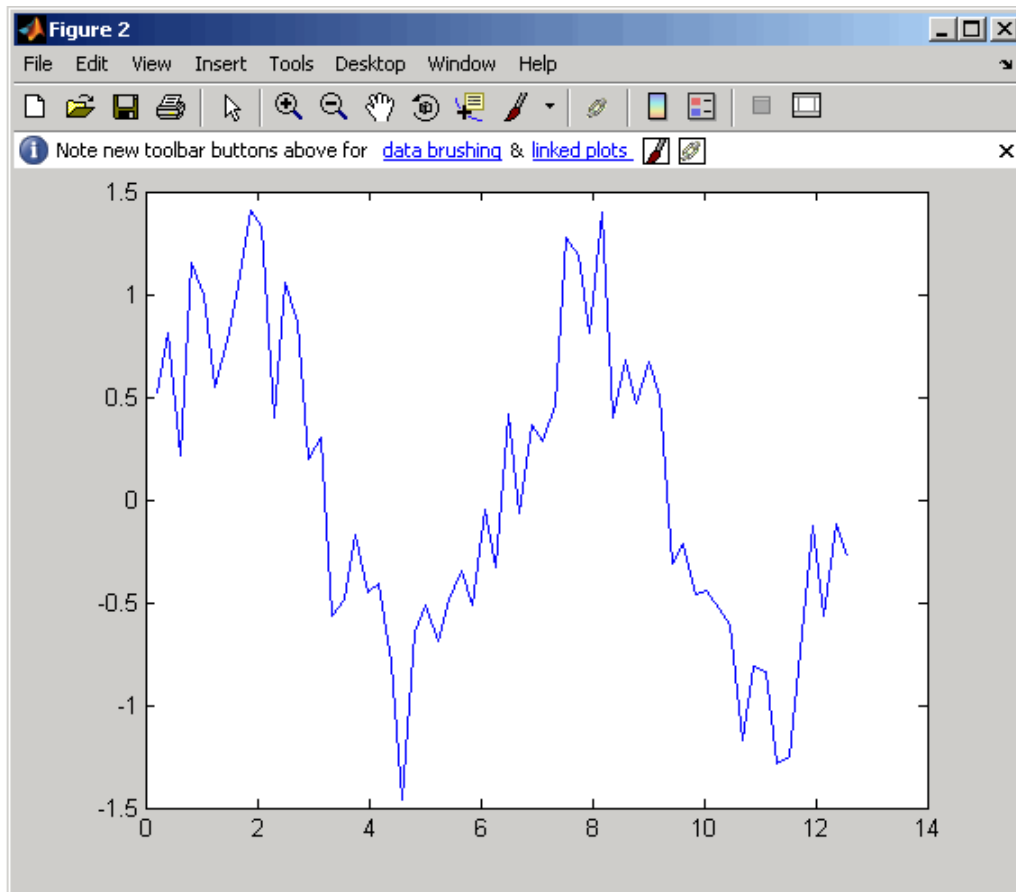
To use another plot type, such as a scatter plot, replace the `plot` command.

```
scatter([signalA{:}], [VarA{:}])
```
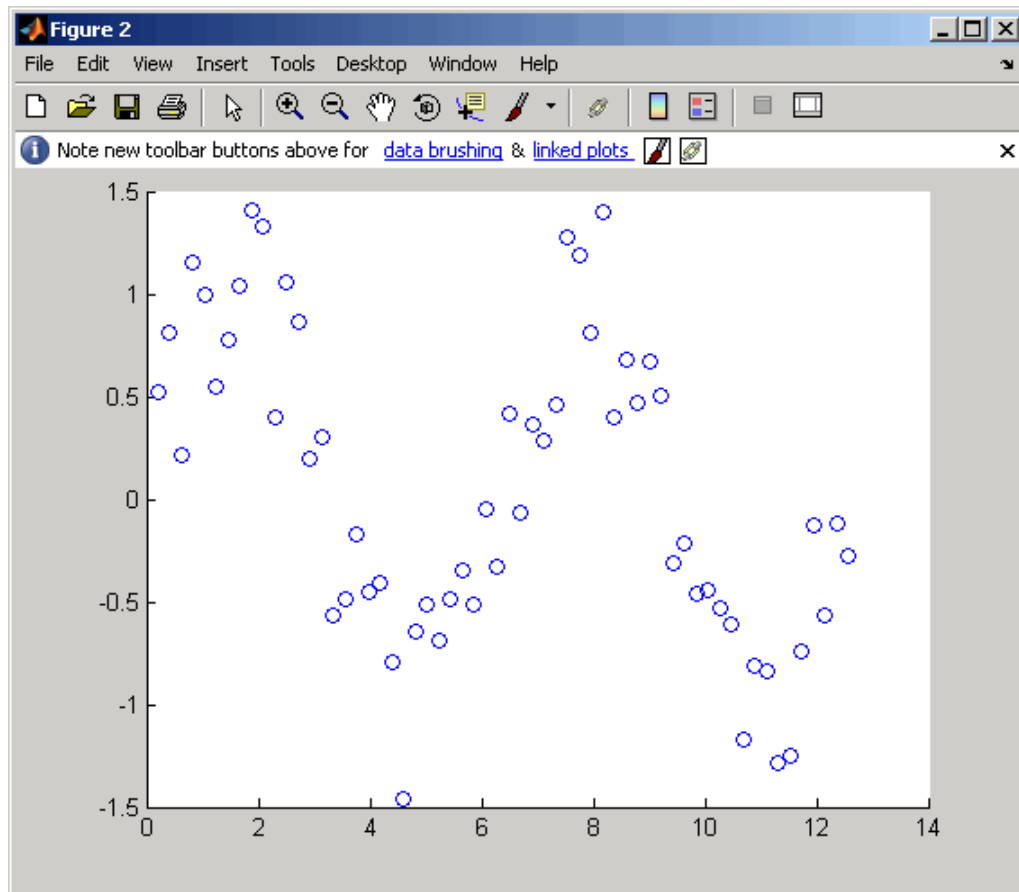
# Access Test Results While Test Is Running

While a test is executing in the SystemTest software, you can access test results using the `systest.testresults.getCurrent` method.

The `getCurrent` function is intended to be used in a MATLAB element within the Pre Test, Main Test, or Post Test sections of a TEST-File, in order to access test information or test results during test execution.

This is a function of the `systest.testresults` class, which is the class definition for a test results object, allowing you to access test results from MATLAB.

The following example used in a MATLAB element will allow you to access the test results object while the test is executing. You can query the `ResultsDataSet` property to access the underlying test data that is currently available.

```
obj = systest.testresults.getCurrent;
currentResults = obj.ResultsDataSet;
```

# Function Reference

# addArtifact

Add artifact to test results object

## Syntax

```
addArtifact(obj, name, filepath)
```

## Description

addArtifact(obj, name, filepath) adds an artifact to the test results object obj using the string name, representing a user-customizable display name, and the string filepath, representing the full file path to the artifact.

This function is a convenience for adding additional artifacts to the Artifacts property of the test results object obj.

Artifacts can be any document or report associated with a test results object. By associating artifacts with a test results object, hyperlinks are automatically provided to access the artifacts when the test results object is displayed at the MATLAB command line.

This is function of the systest.testresults class, which is the class definition for a test results object, allowing you to access test results from MATLAB.

## More About

- "View Test Results at the Command Line"

# getCurrent

Access test results object from SystemTest TEST-File

## Syntax

```
obj = systest.testresults.getCurrent
```

## Description

`obj = systest.testresults.getCurrent` returns `obj`, the test results object associated with the currently running SystemTest test file.

If no TEST-File is currently executing, `obj` is returned as `[]`.

The `getCurrent` function is intended to be used in a MATLAB element within the Pre Test, Main Test, or Post Test sections of a TEST-File, in order to access test information or test results during test execution.

This is a function of the `systest.testresults` class, which is the class definition for a test results object, allowing you to access test results from MATLAB.

## Examples

The following code example used in a MATLAB element will allow you to access the test results object while the test is executing. The `ResultsDataSet` property can be queried in order to access the underlying test data that is currently available.

```
obj = systest.testresults.getCurrent;
currentResults = obj.ResultsDataSet;
```

## More About

• "View Test Results at the Command Line"

# getInfo

List of available Segment type classpaths

## Syntax

```
INFO = systest.signals.segments.getInfo()
```

## Description

`INFO = systest.signals.segments.getInfo()` returns a cell array of strings representing the fully qualified classpaths of each available segment type.

From the list of classpaths, more information can be learned by calling

```
getDisplayName()
getParameterInfo()
```

## Examples

Get type / parameter info about the first Segment.

```
classes = systest.signals.segments.getInfo
eval(sprintf('%s.getDisplayName()', classes{1}))
eval(sprintf('%s.getParameterInfo()', classes{1}))
```

## More About

·    "Programmatic Test Case and Signal Authoring"

# getSignal

Signal mapped to signal name

## Syntax

```
MAPPEDSIGNAL = getSignal(OBJ, SIGNALNAME)
```

## Description

MAPPEDSIGNAL = getSignal(*OBJ, SIGNALNAME*) Gets the signal object currently mapped to the SIGNALNAME in this TestCase OBJ.

If SIGNALNAME is not an existing name an error will be thrown.

## Examples

Get a signal using getSignal().

```
 testCase = systest.TestCase('Test Case 1');
 testCase.In1 = systest.signals.Signal('Step');
 getSignal(testCase, 'In1')
```

Get a signal using getSignal() to get all signals.

```
testCase1 = systest.TestCase('Test Case 1');
testCase1.In1 = systest.signals.Signal('Step');

testCase2 = systest.TestCase('Test Case 2');
testCase2.In1 = systest.signals.Signal('Ramp');

testCases = [testCase1 testCase2];

getSignal(testCases, 'In1')
```

Get a signal by referencing its name space using '.' syntax.

```
 testCase = systest.TestCase('Test Case 1');
```

```
testCase.In1 = systest.signals.Signal('Step');
testCase.In1
```

## More About

- "Programmatic Test Case and Signal Authoring"

# horzcat

Horizontally concatenate one to many TestCase Objects

## Syntax

```
TESTCASES = horzcat(VARARGIN)
```

## Description

`TESTCASES = horzcat(VARARGIN)` horizontally concatenates one to many scalar or arrays of systest.TestCase objects for the overloaded function `systest.TesCase/horzcat`.

All TestCase objects must have unique names.

When creating an array of TestCases, the SignalNames property will be updated to ensure all TestCases have the same SignalNames. If a TestCase does not have a SignalName that another TestCase does, then it will be updated to map to the same Signal as the other TestCase.

## Examples

Create a 1 x 3 list of TestCase objects.

```
tc1 = systest.TestCase('Test Case 1', 'In1');
tc2 = systest.TestCase('Test Case 2', 'In2');
tc3 = systest.TestCase('Test Case 3', 'In3');

testCases = [tc1 tc2 tc3]

testCases(2)
```

## More About

·   "Programmatic Test Case and Signal Authoring"

# isSignal

Check if signal name is mapped

## Syntax

```
RESULT = isSignal(OBJ, SIGNALNAME)
```

## Description

`RESULT = isSignal(OBJ, SIGNALNAME)` returns true if SIGNALNAME is an existing mapping in the given TestCase OBJ.

## Examples

Create a test case; create two signals within the test case – `In1`, which is a step, and `In2`, which is a ramp. Verify that `In1` is mapped.

```
testCase = systest.TestCase('Test Case 1');
testCase.In1 = systest.signals.Signal('Step');
testCase.In2 = systest.signals.Signal('Ramp');

result = isSignal(testCase, 'In1')
```

## More About

- "Programmatic Test Case and Signal Authoring"

# removeSignal

Remove mapped signal

## Syntax

```
OBJ = removeSignal(OBJ, SIGNALNAME)
```

## Description

`OBJ = removeSignal(OBJ, SIGNALNAME)` removes the SIGNALNAME from the mappings in OBJ and returns the updated TestCase OBJ. OBJ may be a scalar TestCase object or an array.

If SIGNALNAME is not an existing name an error will be thrown.

## Examples

Create a test case; create signals `In1` and `In2`; remove signal `In1`.

```
testCase = systest.TestCase('Test Case 1');
testCase.In1 = systest.signals.Signal('Step');
testCase.In2 = systest.signals.Signal('Ramp');

testCase = removeSignal(testCase, 'In1');
```

## More About

· "Programmatic Test Case and Signal Authoring"

# renameSignal

Rename mapped signal to new name

## Syntax

```
OBJ = renameSignal(OBJ, oldSignalName, newSignalName)
```

## Description

OBJ = renameSignal(*OBJ, oldSignalName, newSignalName*) changes the mapping of OLDSIGNALNAME to NEWSIGNALNAME and returns the updated TestCase OBJ.

## Examples

Create a test case; create signal In1; rename signal In1 to signal In2.

```
testCase = systest.TestCase('Test Case 1');
testCase.In1 = systest.signals.Signal('Step');

testCase = renameSignal(testCase, 'In1', 'In2')
```

## More About

- "Programmatic Test Case and Signal Authoring"

# setDataType

Update data type for signal

## Syntax

```
OBJ = setDataType(OBJ, SIGNALNAME, NEWDATATYPE)
```

## Description

`OBJ = setDataType(OBJ, SIGNALNAME, NEWDATATYPE)` sets the data type of SIGNALNAME in all TestCase OBJs to NEWDATATYPE.

## Examples

Change `In1` to be `single` data type.

```
testCase = systest.TestCase('Test Case 1', 'In1');
testCase = setDataType(testCase, 'In1', 'single');
```

## More About

• "Programmatic Test Case and Signal Authoring"

# setSignal

Assign signal to signal name

## Syntax

```
OBJ = setSignal(OBJ, SIGNALNAME, SIGNALOBJ)
OBJ.SIGNALNAME = SIGNALOBJ
```

## Description

`OBJ = setSignal(`*OBJ, SIGNALNAME, SIGNALOBJ*`)` maps the SIGNALOBJ to SIGNALNAME and returns the object TestCase OBJ.

`OBJ.SIGNALNAME = SIGNALOBJ` is an alternative syntax.

## Examples

Assign a signal using setSignal().

```
testCase = systest.TestCase('Test Case 1')
signal = systest.signals.Signal('Constant')
testCase = setSignal(testCase, 'In1', signal)
```

Assign a signal using "dot" field assignment.

```
testCase = systest.TestCase('Test Case 1')
signal = systest.signals.Signal('Constant')
testCase.In1 = signal
```

## More About

- "Programmatic Test Case and Signal Authoring"

# stLoadTestCases

Load `systest.TestCase` objects from SystemTest TEST-file

## Syntax

```
testcases = stLoadTestCases(testFile)
```

## Description

`testcases = stLoadTestCases(testFile)` returns the `systest.TestCase` object saved in *testfile*. It returns the list of test cases in the test file.*testfile* must be a SystemTest TEST-file (.test) available on the MATLAB path or specified with a full path.

The function will return empty if the test does not contain a Test Case Data test vector containing at least one test case. The function will error if called when the `testfile` is open in the SystemTest desktop.

## Examples

Name your test file and model; create a test to that name using that model; load the test cases.

```
testFile = 'f14.test';
modelName = 'f14';

systest.createHarness(testFile, modelName);
testCases = stLoadTestCases(testFile)
```

## More About

*   "Programmatic Test Case and Signal Authoring"

# strun

Run series of SystemTest test files

# Syntax

```
strun(testfile)
```

# Description

strun(*testfile*) runs the SystemTest test file specified by the string *testfile*. You can specify *testfile* as the name of a test file, or as the full path to a test file. If a test file name is specified without a full path, the test file must reside on the MATLAB path.

*testfile* may also be specified as a 1-by-N or N-by-1 cell array of test files, each of which is run serially.

Running tests that you set up in the SystemTest software from the MATLAB command line using strun is useful for running multiple test files as a batch or calling a test file as part of a MATLAB file.

**Note:** If the SystemTest desktop is open when strun is called, strun leaves it open. Otherwise, strun closes the desktop after the test runs.

strun will run in a synchronous manner, that is, the MATLAB command line will be blocked until strun finishes executing. strun will finish executing when either of the following conditions is met:

- All test files have finished executing.
- A **Ctrl+C** is issued.

When a test is run, it is executed using the settings specified in the test file.

If only one test file is specified, and the test encounters an execution error, strun will error. If multiple test files have been specified, a warning will be issued for any test execution errors, and the remaining test files will be run.

Note that it is recommend that you run the test from the SystemTest desktop to verify that elements are not in an error state, and the test will run successfully, before running it via the MATLAB command line using this function.

Note that MATLAB will remain busy while tests are executing via the `strun` command. Control is returned to the MATLAB command line once all tests execute.

## Examples

Run a test called `mytest` that is on the MATLAB path.

```
strun('mytest')
```

Run a test called `mytest` that is not on the MATLAB path, but is in a local directory called `c:\work`.

```
strun('c:\work\mytest.test')
```

Run two tests, called `mytest` and `mytest2`, that are both on the MATLAB path.

```
strun({'mytest' 'mytest2'})
```

Run three tests, two of which are on the MATLAB path, and one of which is not.

```
strun({'mytest' 'c:\work\mytest2.test' 'mytest3'})
```

## More About

• "Run Tests from the MATLAB Command Line"

# stSaveTestCases

Save `systest.TestCase` objects from SystemTest TEST-file

## Syntax

```
testcases = stSaveTestCases(testFile, testCases)
```

## Description

`testcases = stSaveTestCases(`*`testFile`*`, `*`testCases`*`)` saves *testcases* to *testfile*. *testfile* must be a SystemTest TEST-file (.test) available on the MATLAB path or specified with a full path. *testcases* must be a 1xN `systest.TestCase` object.

If there is already a Test Case Data test vector present in *testfile*, the test vector will have its test cases overridden by *testcases* and *testfile* will be updated. If there is no Test Case Data test vector in *testfile*, then one with the name 'TestCases' will be created.

This will error if called when the `testfile` is open in the SystemTest desktop.

## Examples

Name your test file and model; create a test to that name using that model; edit the test cases; save the test cases back to the test.

```
testFile = 'f14.test';
modelName = 'f14';

systest.createHarness(testFile, modelName);

testCases = systest.TestCase('My Test Case', 'u');
stSaveTestCases(testFile, testCases);
```

## More About

• "Programmatic Test Case and Signal Authoring"

# systemtest

Open SystemTest desktop

## Syntax

```
systemtest
systemtest(testfile)
```

## Description

`systemtest` opens the SystemTest desktop with a new untitled test.

`systemtest(testfile)` opens *testfile* in the SystemTest desktop, where *testfile* is a SystemTest test file (`.test`) available on the MATLAB path or specified with a full path.

## Examples

Open a test called `mytest` that is on the MATLAB path.

```
systemtest('mytest')
```

Open a test called `mytest` that is not on the MATLAB path, but is in a local directory called `c:\work`.

```
systemtest('c:\work\mytest.test')
```

# systest.createHarness

Create SystemTest test harness from model

## Syntax

```
systest.createHarness(testFileName,modelName)
```

## Description

systest.createHarness(*testFileName,modelName*) creates a SystemTest test harness named <*testFileName*> for the model <*modelName*>. The test is set up with a Test Case Data test vector and a Simulink element using the information from the Simulink model. The model must be on the MATLAB path. The testFileName must be a writable file location.

## Examples

The following example creates a test harness from a model:

```
>>modelName = 'C:\mymodel.slx';
>>testFileName = 'C:\my_new_harness.test';

>>systest.createHarness(testFileName,modelName)
```

## More About

- "Generate a Test Harness"

# systest.requirements.createlink

Create requirements link object

## Syntax

```
reqlinkobj=
systest.requirements.createlink(format,location,linktype,linkvalue)
reqlinkobj= systest.requirements.createlink(reqlinkstruct)
```

## Description

`reqlinkobj= systest.requirements.createlink(format,location,linktype,linkvalue)` ) creates a requirement link object to the `linkvalue` in the `location` for a given `format`.

`format,location,linktype,linkvalue` must be specified as a string. `format` and `linktype` are not case sensitive, but `location` and `linkvalue` are case sensitive.

If `linkvalue` is a 1xN cell array of strings, then 1xN array of requirement link objects `reqlinkobj` will be created.

`reqlinkobj= systest.requirements.createlink(reqlinkstruct)` ) creates a requirement link object from the requirement link MATLAB structure returned from the Simulink Verification and Validation toolbox. If `reqlinkstruct` is a 1xN struct, then a 1xN array of requirement link objects `reqlinkobj` will be returned.

`reqlinkstruct` - Requirement links are represented in MATLAB in a structure array with the following format:

- `reqlinkstruct.description` – Requirement description.
- `reqlinkstruct.doc` – Document name.
- `reqlinkstruct.id` – Location within the above document.
- `reqlinkstruct.keywords` – User keywords.
- `reqlinkstruct.linked` – Indicates if the link should be reported.
- `reqlinkstruct.reqsys` – Link type registration name.

---

**Note:** `createlink` throws an error if DOORS is not installed or open. `createlink` throws an error if `reqlinkstruct` is not a DOORS link.

---

## Examples

Create a requirement link object to a DOORS object "1" in the module "/example/ MyModule". Note: DOORS must be running.

```
 reqLinkObj = systest.requirements.createLink
    ('DOORS', '/example/MyModule' ,'DOORS Object','1')
```

Create a requirement link object from a requirement link structure attached to a Signal Builder block in a model. Note: DOORS must be running.

```
blockPath = 'mymodel/SignalBuilderBlock/';
reqStruct = rmi('get',blockPath,1);
reqLinkObj = systest.requirements.createLink(reqStruct);
```

## More About

- `systest.requirements.getInfo`
- "Create Requirements Programmatically" on page 5-45

# systest.requirements.getInfo

Information on supported requirements linking objects

## Description

`formats = systest.requirements.getInfo` returns a 1xN cell array of strings describing the supported formats for which requirement links can be created.

`linktypes = systest.requirements.getInfo(`*`format`*`)` returns information describing the supported `linktypes` for a given `format`. `format` must be specified as a string. `format` is not case sensitive. `linktypes` is returned as a 1x1 structure containing the following fields:

- `SupportedLinkType` – A 1xN cell array of strings of `linktypes` specific to the specified `format`.
- `AvailableModuleLocations` – A 1xN cell array of strings containing the module locations for the specified `format`.

`info = systest.requirements.getInfo(`*`format,modulelocation`*`)` returns information describing the supported link values in the `modulelocation` for a given `format`. `format` and `modulelocation` must be specified as a string. `format` is not case sensitive but `modulelocation` is case sensitive. `info` is returned as a 1x1 structure containing the following fields:

- `ModuleID` – A string containing the module ID.
- `ModuleLocation` – A string containing the module location for the `ModuleLocation`.
- `AvailableObjectIds` – A 1xN cell array of strings containing the object IDs for the specified `ModuleLocation`.

## Examples

Find out what requirements in DOORS you can link to in your test case.

```
listOfLinkSuppor = systest.requirements.getInfo('DOORS');
```

## More About

- `systest.requirements.createlink`

# systest.signals.segments

Contain supported segment types for creating signals

## Syntax

```
segment = systest.signals.segments.(segment_type)
```

## Description

`segment = systest.signals.segments.(segment_type)` creates a signal with a segment of type *segment_type*.

The following segment types can be used on the `systest.signals.segments` function:

- `Constant` – A segment with a constant value.

  Properties include:

  `Duration` – The length in seconds of the segment.

  `Value` – The constant value of the segment.
- `Custom` – A segment with user-specified time and data vectors.

  Properties include:

  `Data` – User-specified data vector for each time point.

  `Time` – User-specified time vector for each point.

  `Duration` – The length in seconds of the segment.
- `Pulse` – A segment with a Pulse value.

  Properties include:

  `Duration` – The length in seconds of the segment.

  `InitialValue` – The value of the segment before the pulse.

`Offset` – The length in seconds before the pulse begins.

`FinalValue` – The value of the segment during the pulse.

`PulseWidth` – The length in seconds of the pulse.

- `Ramp` – A segment with a linearly changing value.

   Properties include:

   `Duration` – The length in seconds of the segment.

   `FinalValue` – The value the segment finishes at.

   `InitialValue` – The value the segment starts at.

   `Offset` – The time in seconds before the ramp starts changing.

   `Slope` – The rate of change between InitialValue and FinalValue.

- `Sine` – A periodic sine wave.

   Properties include:

   `Amplitude` – The amplitude of the sine wave.

   `Duration` – The length in seconds of the segment.

   `InitialValue` – The vertical offset of the sine wave.

   `PeriodLength` – The length of time in seconds for a full period.

   `PhaseShift` – The amount in degrees started into the first period.

   `SampleRate` – The amount in seconds between each sampled point.

- `Square` – A periodic series of pulses.

   Properties include:

   `Amplitude` – The amplitude of the square wave.

   `Duration` – The length in seconds of the segment.

DutyCycle – The percentage of time the wave has positive amplitude.

InitialValue – The vertical offset of the square wave.

PeriodLength – The length of time in seconds for a full period.

PhaseShift – The amount in degrees started into the first period.

- Step – A segment that transitions from a one value to another.

  Properties include:

  Duration – The length in seconds of the segment.

  FinalValue – The value of the segment after Offset.

  InitialValue – The value the segment before Offset.

  Offset – The time in seconds before the step occurs.

## Examples

Create a segment of type Constant, with no properties set.

```
segment = systest.signals.segments.Constant
```

Create a segment of type Constant with a value of 5.

```
segment = systest.signals.segments.Constant('Value', 5)
```

Create a segment of type Custom with a time of 0:99 and data of rand(1,100).

```
segment = systest.signals.segments.Custom('Time', 0:99, 'Data', rand(1,100))
```

Create a segment of type Pulse with an offset of 3 and a FinalValue of 2.

```
segment = systest.signals.segments.Pulse('Offset', 3, 'FinalValue', 2)
```

Create a segment of type Ramp with an offset of 2 and a FinalValue of 12.

```
segment = systest.signals.segments.Ramp('Offset', 2, 'FinalValue', 12)
```

Create a segment of type Sine with an amplitude of 5.

```
segment = systest.signals.segments.Sine('Amplitude', 5)
```

Create a segment of type Square with an `amplitude` of `5`.

```
segment = systest.signals.segments.Square('Amplitude', 5)
```

Create a segment of type Step with an `InitialValue` of `4`.

```
segment = systest.signals.segments.Step('InitialValue', 4)
```

## More About

- "Programmatic Test Case and Signal Authoring"

# systest.signals.Signal

Collection of segments used to generate time-based data

## Syntax

`systest.signals.Signal('`*`segment_type`*`')`

## Description

`systest.signals.Signal('`*`segment_type`*`')` creates a signal with a segment of type *segment_type*.

The following properties can be used on the `systest.signals.Signal` function:

- `DataType` – The class of time-based data that will be generated.
- `Duration` – The ending point of the Time vector.
- `ExtrapolationMode` – Used to determine Data values after the endpoint of the last Segmentl.
- `Segments` – An array of Segment objects inside this Signal.
- `Time` – The Time vector of the Signal.
- `Data` – The Data vector of the Signal.

The following static functions can be used with the `systest.signals.Signal` function:

- `getAvailableExtrapolationModes` – Returns a list of valid values for `ExtrapolationMode` property.
- `getAvailableDataTypes` – Returns a list of valid values for `DataType` property.

## Examples

Create a signal with a Step segment followed by a Pulse.

`systest.signals.Signal('Step', 'Pulse')`

Create a signal with custom user data.

```
time = [0:.1:10]';
data = rand(length(time), 1);
systest.signals.Signal('Custom', {'Time', time, 'Data', data'}))
```

## More About

· "Programmatic Test Case and Signal Authoring"

# systest.TestCase

Collection of signals for creating time-based data

## Syntax

```
testCase = systest.TestCase(test_case_name)
```

## Description

`testCase = systest.TestCase(test_case_name)` is a collection of signals in the test case *test_case_name*.

`TestCase` objects allow you to map signal names to `systest.signals.Signal` objects.

`Properties` is a structure holding all properties of the `TestCase`. To access or modify a property of a `TestCase` object, use the `Properties` property.

```
testCase.Properties.Name = 'New Name;
testCase.Properties.Description = 'My Description;
```

The following functions can be used on the `TestCase` object:

- `TestCase` – Creates a `TestCase` object.
- `isSignal` – Checks if a signal name is mapped.
- `setSignal` – Maps a signal name to a signal.
- `getSignal` – Gets the signal mapped to a signal name.
- `removeSignal` – Removes a mapped signal.
- `renameSignal` – Renames a mapped signal to a new name.
- `setDataType` – Updates data type for a signal.
- `horzcat` – Combines TestCases into an array.

## Examples

The following example creates a test case where the ramp linearly increases.

Create a test case by specifying the name.

```
testCase = systest.TestCase('Test Case 1')
```

Map signal names directly to signal objects.

```
testCase.In1 = systest.signals.Signal('Constant');
testCase.In2 = systest.signals.Signal('Step');
testCase.In3 = systest.signals.Signal('Ramp');
```

## More About

- "Programmatic Test Case and Signal Authoring"

# SystemTest Hot Keys

The following keyboard shortcuts are available in the SystemTest software.

| Key | Description |
| --- | --- |
| **Alt+N** | Activates the **New** button to create a new test vector or test variable. |
| **F1** | Opens Help. |
| **F5** | Runs a test. |
| **Ctrl+C** | While a test is running, stops the test. |
| **Ctrl+C** | When a test is not running, copies selection in some parts of the user interface. |
| **Ctrl+N** | Adds a new untitled test. |
| **Ctrl+O** | Opens a test. |
| **Ctrl+Q** | Closes the SystemTest software. |
| **Ctrl+S** | Saves a test. |
| **Ctrl+V** | Pastes the copied selection. |
| **Ctrl+W** | Closes a test. |
| **Ctrl+X** | Cuts a selection in some parts of the user interface. |
| **Ctrl+Y** | Performs redo of last undo action. |
| **Ctrl+Z** | Performs undo of last action. |
| **Ctrl+0** | Gives focus to the **Test Browser**. |
| **Ctrl+1** | Gives focus to the **Properties** pane. |
| **Ctrl+2** | Gives focus to the **Test Vectors** pane. |
| **Ctrl+3** | Gives focus to the **Test Variables** pane. |
| **Ctrl+4** | Gives focus to the **Resources** pane. |
| **Ctrl+5** | Gives focus to the **Run Status** pane. |

| Key | Description |
|---|---|
| **Ctrl+6** | Gives focus to the **Desktop Help** pane. |
| **Ctrl+7** | Gives focus to the **Elements** pane. |
| **Ctrl+8** | Gives focus to the **Getting Started** pane. |
| **Ctrl+Shift+0** | Gives focus to the **Plots** pane. |
| **Ctrl+Shift+U** | Undocks the currently selected pane. |
| **Ctrl+Shift+D** | Docks the currently selected pane. |

# Test Results Data

# Dataset Arrays

| In this section... |
| --- |
| "Overview" on page B-2 |
| "Test Results Data" on page B-2 |
| "Looking at Data" on page B-3 |

## Overview

When you run a test, you can view your test results data as a dataset array in MATLAB. This appendix contains general information on the dataset array that is the format used for test results that can be accessed in MATLAB. For more information, see "View Test Results at the Command Line".

Dataset arrays are used to collect heterogeneous data and metadata including into a single container variable. Dataset arrays can be viewed as tables of values, with rows representing different observations and columns representing different measured variables. Dataset arrays can accommodate variables of different types, sizes, units, etc.

---

**Note:** In the SystemTest software, each observation (i.e., row) is used to represent a test iteration, while each measured variable (i.e., column) represents a test vector or test result value.

---

Dataset arrays combine the organizational advantages of basic MATLAB data types while addressing their shortcomings with respect to storing complex heterogeneous data.

Dataset arrays have a family of functions for assembling, accessing, manipulating, and processing the collected data. Basic array operations parallel those for numerical, cell, and structure arrays.

## Test Results Data

MATLAB data containers (variables) are suitable for completely homogeneous data (numeric, character, and logical arrays) and for completely heterogeneous data (cell and structure arrays). Test results data, however, are often a mixture of homogeneous

variables of heterogeneous types and sizes. Dataset arrays are suitable containers for this kind of data.

Dataset arrays can be viewed as tables of values, with rows representing different test iterations or cases and columns representing different test vector and test result values. Basic methods for creating and manipulating dataset arrays parallel the syntax of corresponding methods for numerical arrays. Because of the potentially heterogeneous nature of the data, dataset arrays have indexing methods with syntax that parallels corresponding methods for cell and structure arrays.

## Looking at Data

Dataset arrays in MATLAB are variables created with the `dataset` function, and then manipulated with associated functions. In the case of the SystemTest software, when a test is run, a dataset array is created and stored as part of a test results object. The test results object is assigned to a variable named `stresults` in the MATLAB workspace when the test stops running. For information on using `stresults`, see "View Test Results at the Command Line".

The following table lists the accessible properties of dataset arrays. Properties can be configured using the `set` function, or accessed using the `get` function.

| Dataset Property | Value |
| --- | --- |
| Description | A string describing the data set. The default is an empty string. |
| Units | A cell array of strings giving the units of the variables in the data set. The number of strings must equal the number of variables. Strings may be empty. The default is an empty cell array. |
| DimNames | A cell array of two strings giving the names of the rows and columns, respectively, of the data set. The default is {'Observations' 'Variables'}. |
| UserData | Any variable containing additional information to be associated with the data set. The default is an empty array. |
| ObsNames | A cell array of nonempty, distinct strings giving the names of the observations in the data set. The number of strings must equal the number of observations. The default is an empty cell array. |
| VarNames | A cell array of nonempty, distinct strings giving the names of the variables in the data set. The number of strings must equal the |

| Dataset Property | Value |
|---|---|
| | number of variables. The default is the cell array of string names for the variables used to create the data set. |

Functions associated with dataset arrays are used to display, summarize, convert, concatenate, and access the collected data. Examples include `disp`, `summary`, `double`, `horzcat`, and `get`, respectively. Many of these functions are invoked using operations analogous to those for numerical arrays, and do not need to be called directly. (For example, `horzcat` is invoked by `[ ]`.) Other functions access the collected data and must be called directly (for example, `replacedata`).

Dataset arrays are implemented as MATLAB *objects*; the associated functions are their *methods*. It isn't necessary to understand objects and methods to make use of dataset arrays—in fact, dataset arrays are designed to behave as much as possible like other, familiar MATLAB arrays.

# Dataset Array Operations

This table lists available methods for dataset arrays. Many of the methods are invoked by familiar MATLAB operators and do not need to be called directly. For full descriptions of individual methods, type

```
help dataset/methodname
```

| Dataset Method | Description |
| --- | --- |
| cat | Concatenate dataset arrays. The `horzcat` and `vertcat` methods implement special cases. |
| dataset | Create dataset array. |
| datasetfun | Apply function to each variable of dataset array. |
| disp | Display dataset array, without printing data set name. |
| display | Display dataset array, printing data set name. This method is invoked when the name of a dataset array is entered at the command prompt. |
| double | Convert dataset variables to `double` array. |
| end | Last index in indexing expression for dataset array. |
| get | Get dataset array property. |
| horzcat | Horizontal concatenation for dataset arrays (add variables). This method is invoked by square brackets. |
| isempty | True for empty dataset array. |
| join | Merge observations from two dataset arrays. |
| length | Length of dataset array. |
| ndims | Number of dimensions of dataset array. |
| numel | Number of elements in dataset array. |
| replacedata | Convert array to dataset variables. |
| set | Set dataset array property value. |
| single | Convert dataset variables to `single` array. |
| size | Size of dataset array. |
| sortrows | Sort rows of dataset array. |

| Dataset Method | Description |
|---|---|
| `subsasgn` | Subscripted assignment for dataset array. This method is invoked by the parenthesis, dot, and curly brace indexing. |
| `subsref` | Subscripted reference for dataset array. This method is invoked by the parenthesis, dot, and curly brace indexing. |
| `summary` | Print summary statistics for dataset array. |
| `unique` | Unique observations in dataset. |
| `vertcat` | Vertical concatenation for dataset arrays (add observations). This method is invoked by square brackets. |