

Simulink® Compiler™

User's Guide



MATLAB® & SIMULINK®

R2020a



1

Simulink Compiler

Deploy an App Designer Simulation with Simulink Compiler	1-2
Deploying A Simulation App with Simulink Compiler	1-2
Running the Deployed Application	1-5
Deploying A Simulation App with Simulink Compiler	1-7
Deploy Simulations with Tunable Parameters	1-10
Prepare a Script to Deploy Simulations with Parameter Tuning	1-10
Comparing Simulink Coder and Simulink Compiler	1-13
Differences	1-13
Common Questions about Simulink Compiler and Simulink Coder	1-13
Rapid Accelerator Limitations	1-15
Rapid Accelerator Mode	1-15
Rapid Accelerator Mode Limitations	1-15
Debug an Application for Deployment	1-17
Debug Application in Simulink	1-17
Debug Application	1-17
Export Simulink Model to Standalone FMU	1-18

Simulink Compiler

- “Deploy an App Designer Simulation with Simulink Compiler” on page 1-2
- “Deploying A Simulation App with Simulink Compiler” on page 1-7
- “Deploy Simulations with Tunable Parameters” on page 1-10
- “Comparing Simulink Coder and Simulink Compiler” on page 1-13
- “Rapid Accelerator Limitations” on page 1-15
- “Debug an Application for Deployment” on page 1-17
- “Export Simulink Model to Standalone FMU” on page 1-18

Deploy an App Designer Simulation with Simulink Compiler

In this section...

“Deploying A Simulation App with Simulink Compiler” on page 1-2

“Running the Deployed Application” on page 1-5
--

This example walks you through the workflow of creating a simulation app in App Designer and using Simulink Compiler to deploy it. The example explains the code that is used to build the app.

Deploying A Simulation App with Simulink Compiler

In this example, we use an app that is prepared in the App Designer and deploy it with Simulink Compiler.

Open and Explore Model

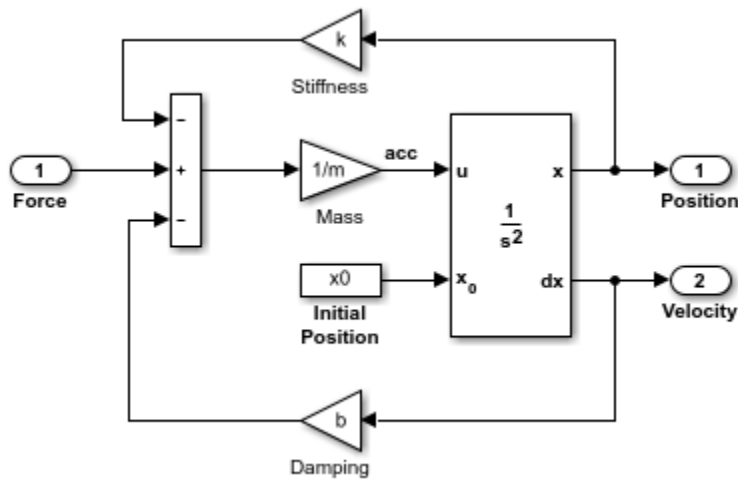
In this example, we use the model of a mass springer damper system. The mass-spring-damper model consists of discrete mass nodes distributed throughout an object and interconnected via a network of springs and dampers. This model is well-suited for modelling object with complex material properties such as non-linearity and elasticity. In this example we use the mass spring damper system. The system is parametrized by mass (m), spring stiffness (k), damping (b) and the initial position (x0). The input to the system is the applied force.

To explore this model with different values of the tunable parameters, create the following model workspace variables:

- Mass - m.
- Spring stiffness - k.
- Damping - b.
- Initial position - xo.

To create the model workspace variables, go to the **Modelling** tab and select **Model Workspace** in the **Data Repositories** in the **Design** section. Use the **Add MATLAB Variables** icon to add the above four variables. Add the appropriate initial values, for example, 3, 128, 2 and 0.5 respectively.

```
open_system('MassSpringDamperModel');
```



Create the App in App Designer

Use the MATLAB APP Designer to create an app to simulate the model with different parameter values and input signals. To learn more about how to create an app using the App Designer, see “Create and Run a Simple App Using App Designer” (MATLAB) Use the `MassSpringDamperApp.mlapp` file to use the app.

MassSpringDamperApp

App Details

The main part for the app is the simulate button callback function. It has the following salient parts: setup the `SimulationInput` object, configure it for deployment, simulate, and plot the simulation results.

The functionality of the application to change and experiment with the tunable parameters is defined in the callback function `SimulateButtonPushed`. This callback function enables you to change, experiment and analyze different simulations by modifying the values in the app designer.

SimulateButtonPushed Callback Function Code

This section explains the code written to create the app, `MassSpringDamperApp`. The callback function `SimulateButtonPushed` is called in the app designed in the App Designer. This callback function defines how the model is simulated. We use the `Simulink.SimulationInput` object to set the variables to the model and use these variables to change the values and analyze the model.

Create the `Simulink.SimulationInput` Object in the `SimulateButtonPushed` Function

In the `SimulateButtonPushed` function, create a `SimulationInput` object, `SimInp` for the model `MassSpringDamperModel`. Use the `setModelParameters` method on the `SimulationInput` object. In this example, we set the `StopTime` model parameter for the simulation.

```
function SimulateButtonPushed(app, event)
    try
        app.toggleUIC('off', 'Simulating ...')

        simInp = Simulink.SimulationInput('MassSpringDamperModel');

        stopTimeStr = num2str(app.StopTimeSpinner.Value);
        simInp = simInp.setModelParameter('StopTime', stopTimeStr);
```

Set the Values of the Tunable Parameters and the Input Signal

To set the input signal to the model, use the `ExternalInput` property of the `Simulink.SimulationInput` object, `simInp`. Use the `setVariables` method to set the values of the four tunable parameters. Create the force input signal and set it as the `ExternalInput` in the simulation input object.

```
simInp.ExternalInput = app.externalInput();

simInp = simInp.setVariable('k', app.StiffnessSpinner.Value);
simInp = simInp.setVariable('m', app.MassSpinner.Value);
simInp = simInp.setVariable('b', app.DampingSpinner.Value);
simInp = simInp.setVariable('x0', app.InitialPositionEditField.Value);
```

Configure for Deployment

Now that we have assigned all the values to the variables and set the input signal, the `Simulink.SimulationInput` object is required to be configured for deployment. Use the `simulink.compiler.configureForDeployment` function of Simulink Compiler. This function handles all the settings required for the script to be compatible for deployment by setting the simulation mode to rapid accelerator, and by setting the parameter `RapidAcceleratorUpToDateCheck` to off.

```
simInp = simulink.compiler.configureForDeployment(simInp);
```

Simulate and Plot the Results

Use the configured `Simulink.SimulationInput` object to run the simulation with the `sim` command. Plot the results from the simulation using the `Simulink.SimulationOutput` object, `simOut`.


```

simOut = sim(simInp);

t = simOut.y.time;
yp = simOut.y.signals(1).values;
yv = simOut.y.signals(2).values;
plot(app.PositionUIAxes, t, yp);
plot(app.VelocityUIAxes, t, yv);

catch ME
    errordlg(ME.message);
end
app.toggleUIC('on', 'Simulate');
end

```

Test Out the Application in App Designer

Before deploying the application, ensure that the app runs in the App Designer. Click the **Simulate** button on the app to verify that the application works by simulating the model for different values.

Compile Script for Deployment

To compile the app, use the `mcc` command, followed by the script name.

```
mcc -m MassSpringDamperApp.mlapp
```

Running the Deployed Application

Install MATLAB Runtime and Package the Deployable

To run the deployed executable, you need an appropriate runtime environment. For more information, see “MATLAB Runtime” (MATLAB Compiler).

Ensure that the path environment variable is free of other instances of MATLAB Runtime from previous installs. If there are any, remove them.

To install MATLAB Runtime, follow the instructions on “Install and Configure the MATLAB Runtime” (MATLAB Compiler).

Compile the deployable for the first time as follows:

- 1 Enter `deploytool` command in the MATLAB Command Window and select **Application Compiler**.
- 2 In the **Main File** section, add the file to be deployed, `MassSpringDamperApp.mlapp`
- 3 In the **Packaging Options** section on the toolstrip, select **Runtime included in package** and enter the `deployed_installer` in the text box.
- 4 Click **Package** in the **Package** section of the toolstrip.
- 5 Once the package is ready, use the `deployed_installer` in the `for_redistribution` folder to install the proper runtime environment for running the deployed application.

Run the Deployed Application

You can run the deployed script only on the platform that the deployed script was developed on.

It is recommended to run the deployed application from the Windows Command Prompt. Running the deployed application from the command prompt also enables the script to print errors when something is wrong in the deployed application. These errors can help troubleshoot the problem.

Note The `MassSpringDamperApp.mlapp` is not compatible to run on Web Apps.

See Also

`Simulink.SimulationInput | deploytool | mcc |
simulink.compiler.configureForDeployment`

More About

- “Create and Deploy a Script with Simulink Compiler”
- “Deploy Simulations with Tunable Parameters”

Deploying A Simulation App with Simulink Compiler

In this example, we use an app that is prepared in the App Designer and deploy it with Simulink Compiler.

Open and Explore Model

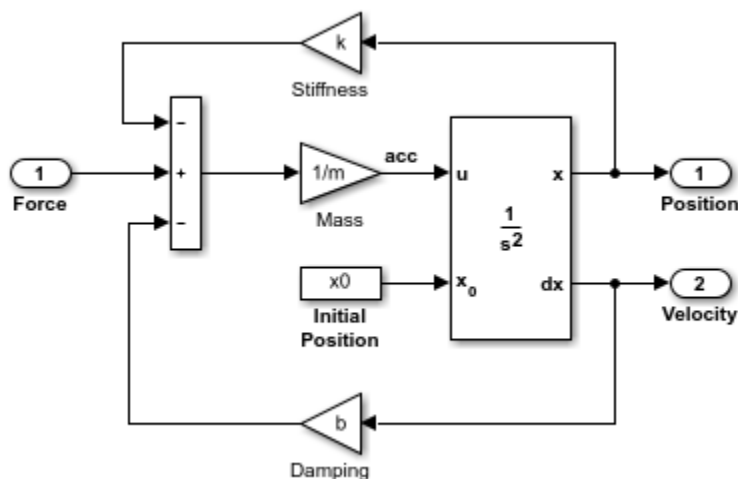
In this example, we use the model of a mass spring damper system. The mass-spring-damper model consists of discrete mass nodes distributed throughout an object and interconnected via a network of springs and dampers. This model is well-suited for modelling object with complex material properties such as non-linearity and elasticity. In this example we use the mass spring damper system. The system is parametrized by mass (m), spring stiffness (k), damping (b) and the initial position (x_0). The input to the system is the applied force.

To explore this model with different values of the tunable parameters, create the following model workspace variables:

- Mass - m .
- Spring stiffness - k .
- Damping - b .
- Initial position - x_0 .

To create the model workspace variables, go to the **Modelling** tab and select **Model Workspace** in the **Data Repositories** in the **Design** section. Use the **Add MATLAB Variables** icon to add the above four variables. Add the appropriate initial values, for example, 3, 128, 2 and 0.5 respectively.

```
open_system('MassSpringDamperModel');
```



Create the App in App Designer

Use the MATLAB APP Designer to create an app to simulate the model with different parameter values and input signals. To learn more about how to create an app using the App Designer, see “Create and Run a Simple App Using App Designer” (MATLAB) Use the `MassSpringDamperApp.mlapp` file to use the app.

MassSpringDamperApp

App Details

The main part for the app is the simulate button callback function. It has the following salient parts: setup the `SimulationInput` object, configure it for deployment, simulate, and plot the simulation results.

The functionality of the application to change and experiment with the tunable parameters is defined in the callback function `SimulateButtonPushed`. This callback function enables you to change, experiment and analyze different simulations by modifying the values in the app designer.

SimulateButtonPushed Callback Function Code

This section explains the code written to create the app, `MassSpringDamperApp`. The callback function `SimulateButtonPushed` is called in the app designed in the App Designer. This callback function defines how the model is simulated. We use the `Simulink.SimulationInput` object to set the variables to the model and use these variables to change the values and analyze the model.

Create the `Simulink.SimulationInput` Object in the `SimulateButtonPushed` Function

In the `SimulateButtonPushed` function, create a `SimulationInput` object, `simInp` for the model `MassSpringDamperModel`. Use the `setModelParameters` method on the `SimulationInput` object. In this example, we set the `StopTime` model parameter for the simulation.

```
function SimulateButtonPushed(app, event)
    try
        app.toggleUIC('off', 'Simulating ...')

        simInp = Simulink.SimulationInput('MassSpringDamperModel');

        stopTimeStr = num2str(app.StopTimeSpinner.Value);
        simInp = simInp.setModelParameter('StopTime', stopTimeStr);
```

Set the Values of the Tunable Parameters and the Input Signal

To set the input signal to the model, use the `ExternalInput` property of the `Simulink.SimulationInput` object, `simInp`. Use the `setVariables` method to set the values of the four tunable parameters. Create the force input signal and set it as the `ExternalInput` in the simulation input object.

```
simInp.ExternalInput = app.externalInput();

simInp = simInp.setVariable('k', app.StiffnessSpinner.Value);
simInp = simInp.setVariable('m', app.MassSpinner.Value);
simInp = simInp.setVariable('b', app.DampingSpinner.Value);
simInp = simInp.setVariable('x0', app.InitialPositionEditField.Value);
```

Configure for Deployment

Now that we have assigned all the values to the variables and set the input signal, the `Simulink.SimulationInput` object is required to be configured for deployment. Use the `simulink.compiler.configureForDeployment` function of Simulink Compiler. This function handles all the settings required for the script to be compatible for deployment by setting the simulation mode to rapid accelerator, and by setting the parameter `RapidAcceleratorUpToDateCheck` to off.

```
simInp = simulink.compiler.configureForDeployment(simInp);
```

Simulate and Plot the Results

Use the configured `Simulink.SimulationInput` object to run the simulation with the `sim` command. Plot the results from the simulation using the `Simulink.SimulationOutput` object, `simOut`.

```
simOut = sim(simInp);

t = simOut.y.time;
yp = simOut.y.signals(1).values;
yv = simOut.y.signals(2).values;
plot(app.PositionUIAxes, t, yp);
plot(app.VelocityUIAxes, t, yv);

catch ME
    errordlg(ME.message);
end
app.toggleUIC('on', 'Simulate');
end
```

Test Out the Application in App Designer

Before deploying the application, ensure that the app runs in the App Designer. Click the **Simulate** button on the app to verify that the application works by simulating the model for different values.

Compile Script for Deployment

To compile the app, use the `mcc` command, followed by the script name.

```
mcc -m MassSpringDamperApp.mlapp
```

Deploy Simulations with Tunable Parameters

With Simulink Compiler, you can deploy simulations that use tunable parameters.

As you construct a model, you can experiment with block parameters, such as the coefficients of a Transfer Fcn block, to help you decide which blocks to use. You can simulate the model with different parameter values, and capture and observe the simulation output.

You can change the values of most numeric block parameters during a simulation. This technique allows you to quickly test parameter values while you develop an algorithm. You can:

- Tune and optimize control parameters.
- Calibrate model parameters.
- Test control robustness under different conditions.

The following example shows how to set a tunable parameter in a model, write a standalone application that can be used to tune the parameters, and analyze the simulations. For more information on tunable parameters, see “Tune and Experiment with Block Parameter Values” (Simulink).

Prepare a Script to Deploy Simulations with Parameter Tuning

In this example, create a MATLAB® function to simulate the model `sldemo_suspn_3dof_deploy` with the values of `Simulink.SimulationInput`. Save the script as `deployParameterTuning.m` on the MATLAB path.

Prepare a Function to Deploy

Create a function called `deployParameterTuning` containing the code shown below. This code creates a `Simulink.SimulationInput` object for the model `sldemo_suspn_3dof_deploy`. `mb` is the value that we pass through the `setVariable` method for the tunable parameter, `Mb`. To configure this script to be deployed, use the function `simulink.compiler.configureForDeployment`. `simulink.compiler.configureForDeployment` configures the `Simulink.SimulationInput` object for by deployment by setting its simulation mode to Rapid Accelerator and by restricting inputs that require rebuilding the deployed app.

```
function deployParameterTuning(oName, mb)

    if ischar(mb) || isstring(mb)
        mb = str2double(mb);
    end

    if isnan(mb) || ~isa(mb, 'double') || ~isscalar(mb)
        disp('The value of mb given to deployedScriptSaveResults must be a double scalar or a string')
    end

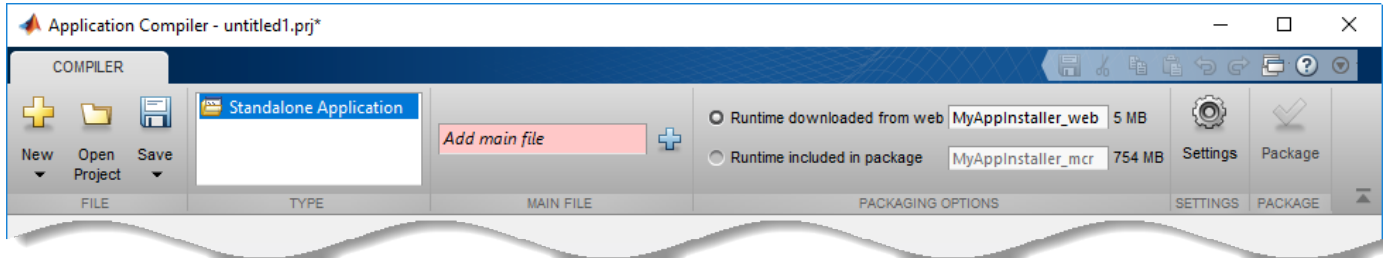
    in = Simulink.SimulationInput('sldemo_suspn_3dof_deploy');
    in = in.setVariable('Mb', mb);
    in = simulink.compiler.configureForDeployment(in);
    out = sim(in);

    save(oName, 'out');


end
```

Deploy the Prepared Function

- 1 On the **Apps** tab, in the **Apps** section, click the arrow. In **Application Deployment**, click **Application Compiler**.



Alternately, you can open the **Application Compiler** app by entering `applicationCompiler` at the MATLAB prompt.

- 2 In the **Compiler** project window, specify the main file of the MATLAB application that you want to deploy.
 - a In the **Main File** section, click .
 - b In the **Add Files** window, browse to path where you have saved the prepared function, and select `deployParameterTuning.m`. Click **Open**.

The function `deployParameterTuning.m` is added to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an installer that includes the MATLAB Runtime installer.
- 4 Customize the packaged application and its appearance:
 - **Application information** — This section lists editable information about the deployed application. You can also customize the standalone applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” (MATLAB Compiler).
 - **Command line input type options** — This section lists selection of input data types for the standalone application. For more information, see “Determine Data Type of Command-Line Input (For Packaging Standalone Applications Only)” (MATLAB Compiler).
 - **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” (MATLAB Compiler) .
 - **Files required for your application to run** —Files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” (MATLAB Compiler).
 - **Files installed for your end user** — This section lists the files that are installed with your application. These files include:
 - A generated `readme.txt` file
 - The generated executable for the target platform

See “Specify Files to Install with Application” (MATLAB Compiler).

- **Additional runtime settings** —This section lists platform-specific options for controlling the generated executable. See “Additional Runtime Settings” (MATLAB Compiler).
- 5 To generate the packaged application, click **Package**. In the Save Project dialog box, specify the location to save the project.
 - 6 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the packaging process is complete, examine the generated output.

- `PackagingLog.txt` — Log file generated by MATLAB Compiler.
- Three folders are generated in the target folder location: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`. For more information about the files generated in these folders, see .

See Also

`Simulink.SimulationInput` | `configureForDeployment` | `deploytool` | `mcc` | `sim`

More About

- “Simulink Compiler Workflow Overview”
- “Tune and Experiment with Block Parameter Values” (Simulink)
- “Code Regeneration in Accelerated Models” (Simulink)

Comparing Simulink Coder and Simulink Compiler

Simulink Compiler enables you to share Simulink simulations as standalone executables. You can build the executables by packaging the compiled Simulink model and the MATLAB code to set up, run, and analyze a simulation. Standalone executables can be complete simulation apps that use MATLAB graphics and UIs designed with MATLAB App Designer. To cosimulate with an external simulation environment, you can generate standalone Functional Mockup Unit (FMU) binaries that adhere to the Functional Mockup Interface (FMI) standard.

Simulink Coder generates and executes C and C++ code from Simulink models, Stateflow® charts, and MATLAB functions. The generated source code can be used for real-time and non-real-time applications, rapid prototyping, and hardware-in-the-loop testing. You can tune and monitor the generated code using Simulink or run and interact with the code outside MATLAB and Simulink.

Differences

The following table states the major comparisons between Simulink Compiler and Simulink Coder. Use this table to understand the differences between the applications and usage of the two products.

Outputs and Support	Simulink Compiler	Simulink Coder
Main Use Case	Deploy simulations as standalone executables on desktop or production servers	Generate portable C/C++ code for Simulink model that can be deployed on embedded platforms or desktop
Output	Executable or software component or shared library	Portable and readable C/C++ source code
Simulink Block Support	All the blocks supported in Rapid Accelerator mode in Simulink	A subset of Simulink blocks
Supported Blocksets	All the blocksets supported by Rapid Accelerator mode in Simulink	A subset of Simulink blocks
Production	MATLAB Production Server	Embedded Coder
Simulink Graphics Support	Supports MATLAB Graphics.	None
Library Dependencies	MATLAB Runtime	None

Common Questions about Simulink Compiler and Simulink Coder

The following table answers some of the common questions about using Simulink Compiler and Simulink Coder, such as the memory required, performance, and other questions about support.

Common Questions	Simulink Compiler	Simulink Coder
What files are produced?	Shared executables or libraries, along with the required MATLAB Runtime components.	Source code (*.c & *.h) that can be compiled into shared libraries and executables

Common Questions	Simulink Compiler	Simulink Coder
Which platforms can these files be deployed to?	All platforms supported by MATLAB (Windows, Mac, and Linux)	Any platform that supports ANSI/ISO C/C++ code
Does it generate readable code?	No, only non-readable shared libraries	Yes, readable source code
Is it faster than Simulink?	Runs at the same speed as Rapid Accelerator mode in Simulink.	Has the potential to be faster, depending on the algorithm. The generated code is not faster for optimized MATLAB functions (such as FFT, or Image Processing, and Computer Vision functions) Use GPU CoderGPU Coder™ to generate CUDA source code that potentially runs faster on NVIDIA GPUs.
Does it take advantage of hardware accelerators?	Supports the same hardware as MATLAB, including GPUs and AVX. Multicore and clusters are supported via Parallel Computing Toolbox	C code running on local multicore machines can be supported using the OpenMP API. Use GPU Coder to generate CUDA source code that runs on NVIDIA GPUs. Use HDL CoderHDL Coder™ to generate Verilog or VHDL that runs on FPGAs.
What is the difference in memory use on a desktop?	Highly dependent on the executables. MATLAB Runtime itself uses more memory than the Simulink Coder.	Highly dependent on the MATLAB code.
What file I/O formats does each software support?	Supports all formats supported in MATLAB	Limited file support: text files, audio, and video formats. Does not support image formats.

See Also

More About

- “Simulink Compiler Workflow Overview”

Rapid Accelerator Limitations

Rapid Accelerator Mode

The rapid accelerator mode creates a *Rapid Accelerator standalone executable* from your model. This executable includes the solver and model methods, but it resides outside of MATLAB and Simulink. It uses external mode to communicate with Simulink.

MATLAB and Simulink run in one process, and if a second processing core is available, the standalone executable runs there.

Rapid Accelerator Mode Limitations

- Rapid Accelerator mode does not support:
 - Algebraic loops.
 - Targets written in C++.
 - Interpreted MATLAB Function blocks.
 - Noninlined MATLAB language or Fortran S-functions. You must write S-functions in C or inline them using the Target Language Compiler (TLC) or you can also use the MEX file. For more information, see .
 - Debugger or Profiler.
 - Run time objects for Simulink.RunTimeBlock and Simulink.BlockCompOutputPortData blocks.
- Model parameters must be one of these data types:
 - `boolean`
 - `uint8` or `int8`
 - `uint16` or `int16`
 - `uint32` or `int32`
 - `single` or `double`
 - Fixed-point
 - Enumerated
- You cannot pause a simulation in Rapid Accelerator mode.
- If a Rapid Accelerator build includes referenced models (by using Model blocks), set up these models to use fixed-step solvers to generate code for them. The top model, however, can use a variable-step solver as long as the blocks in the referenced models are discrete.
- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change is changing the number of delays in a DSP simulation. In these cases, you must regenerate the code for the model. See for more information.
- For root inports, Rapid Accelerator mode supports only base as the Srcworkspace.
- For root inports, when you specify the minimum and maximum values that the block should output, Rapid Accelerator mode does not recognize these limits during simulation.
- In Rapid Accelerator mode, To File or To Workspace blocks inside function-call subsystems do not generate any logging files if the function-call port is connected to Ground or unconnected.

See Also

More About

- [“Simulink Compiler Workflow Overview”](#)
- [“Rapid Accelerator Mode” \(Simulink\)](#)
- [“Select Blocks for Rapid Accelerator Mode” \(Simulink\)](#)
- [“Parameter Tuning in Rapid Accelerator Mode” \(Simulink\)](#)

Debug an Application for Deployment

This topic provides high-level tips on debugging the standalone applications. The following lists highlight the solutions and tips for most frequently encountered errors.

Debug Application in Simulink

Use the following tips while preparing the standalone application to be deployed:

- To ensure that the model runs successfully in rapid accelerator mode correctly, run the rapid accelerator target in a writable directory.
- While writing the script, ensure that the `sim` command uses the `Simulink.SimulationInput` object as the input.
- If you see the error "Unable to resolve the name `Simulink.SimulationInput`", check that the model is on the path.
- If the dependent files are located in another directory, attach them by using the flag `-a`. For example, `mcc -m scriptName.m -a myDataFile.dat`.
- If you are using a function as string, either:

- Add a function pragma `##function`.

```
set(gca, 'ButtonDownFcn', 'foo'); % function foo is a string here.
```

```
##function foo
```

```
set(gca, 'ButtonDownFcn', 'foo'); % function foo is a string here.
```

- Write it as an anonymous function

```
set(gca, 'ButtonDownFcn', @foo);
```

Debug Application

- Callback functions in the model might include functions that are not deployable. Ensure that the functions in the callbacks of the model are deployable.
- Callback functions are not invoked at runtime. Ensure that the deployed simulation application does not use callback functions that are required to be invoked at runtime.

See Also

More About

- "Simulink Compiler Workflow Overview"

Export Simulink Model to Standalone FMU

This example shows how to export Simulink component to standalone Co-Simulation FMU 2.0 with Simulink Compiler®. For a detailed explanation of the model, see `sldemo_fuelsys` and `fxpdemo_fuelsys`.

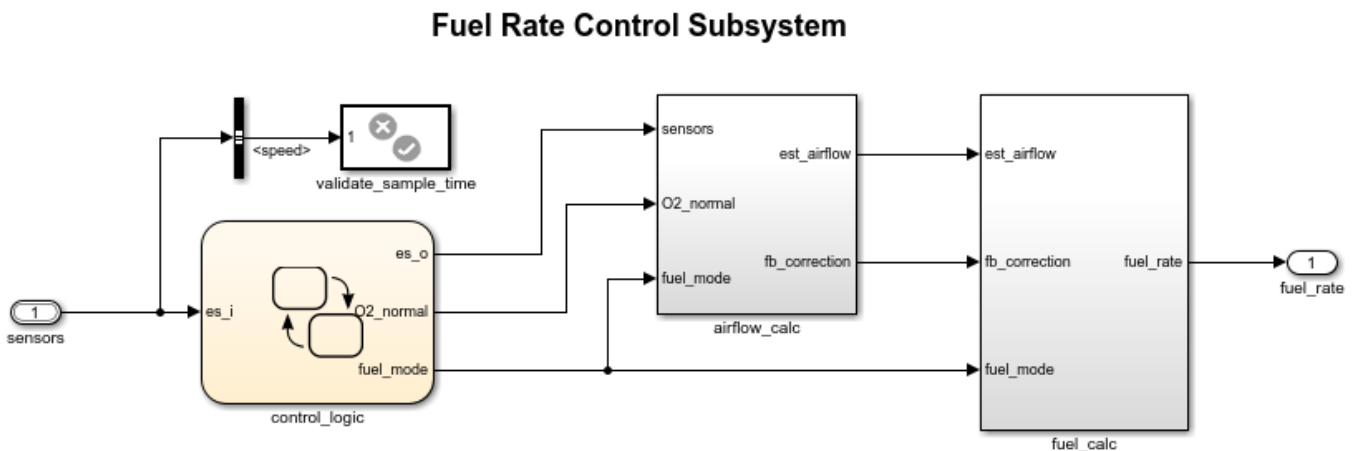
In this example, the air-fuel ratio control system is composed of three Simulink models:

- Fuel Rate Control Component: `fmudemo_export_fuelsys_controller`,
- Engine Gas Dynamics Component: `fmudemo_export_fuelsys_plant`, and
- top-level model `fmudemo_export_fuelsys_top`.

Once the controller and plant components are export to FMU format, they can be integrated using the top-level model. The generated FMUs can also be imported into other simulation tools that support FMI. For a list of Tools that support FMI, see: <https://fmi-standard.org/tools/>.

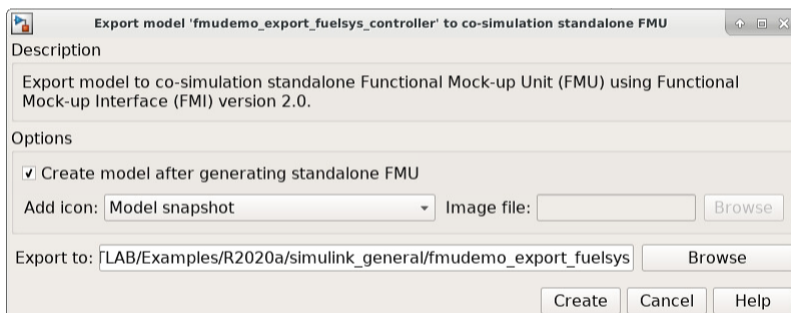
Export Fuel Rate Control Component to FMU

Open the `fmudemo_export_fuelsys_controller` example model.



Copyright 1990-2020 The MathWorks, Inc.

From **Simulation** tab, click drop-down button for **Save**. In **Export Model To** section, click **Standalone FMU...** In FMU Export dialog, configure wrapper model and icon settings, and specify save location for generated FMU.

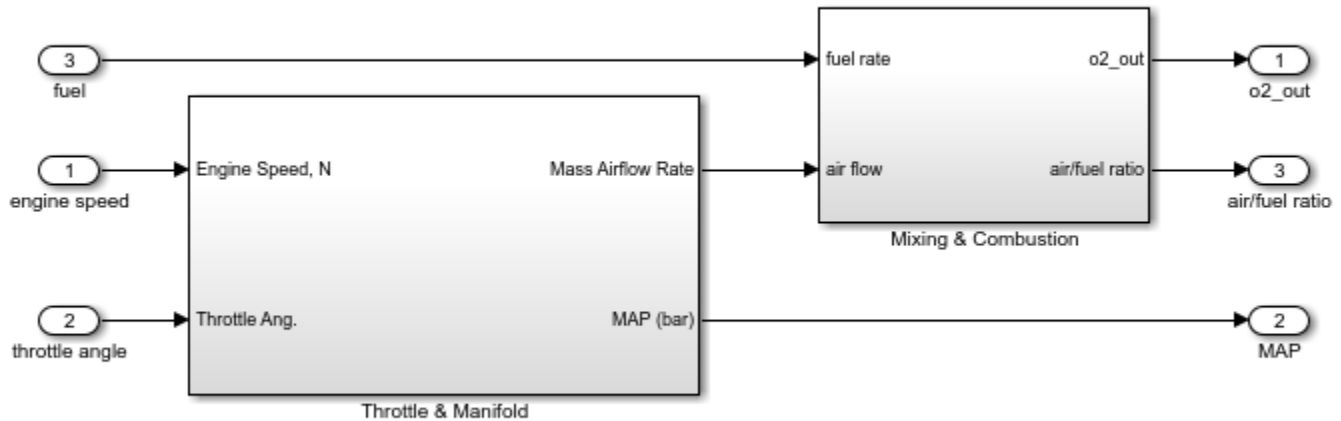


Click **Create** to export to FMU. The `fmudemo_export_fuelsys_controller.fmu` file can be found at specified save location.

Export Engine Gas Dynamics Component to FMU

Open the `fmudemo_export_fuelsys_plant` example model.

Engine Gas Dynamics



Copyright 1990-2020 The MathWorks, Inc.

FMU can also be exported using command-line. In MATLAB command line window, use `exportToFMU2CS` command:

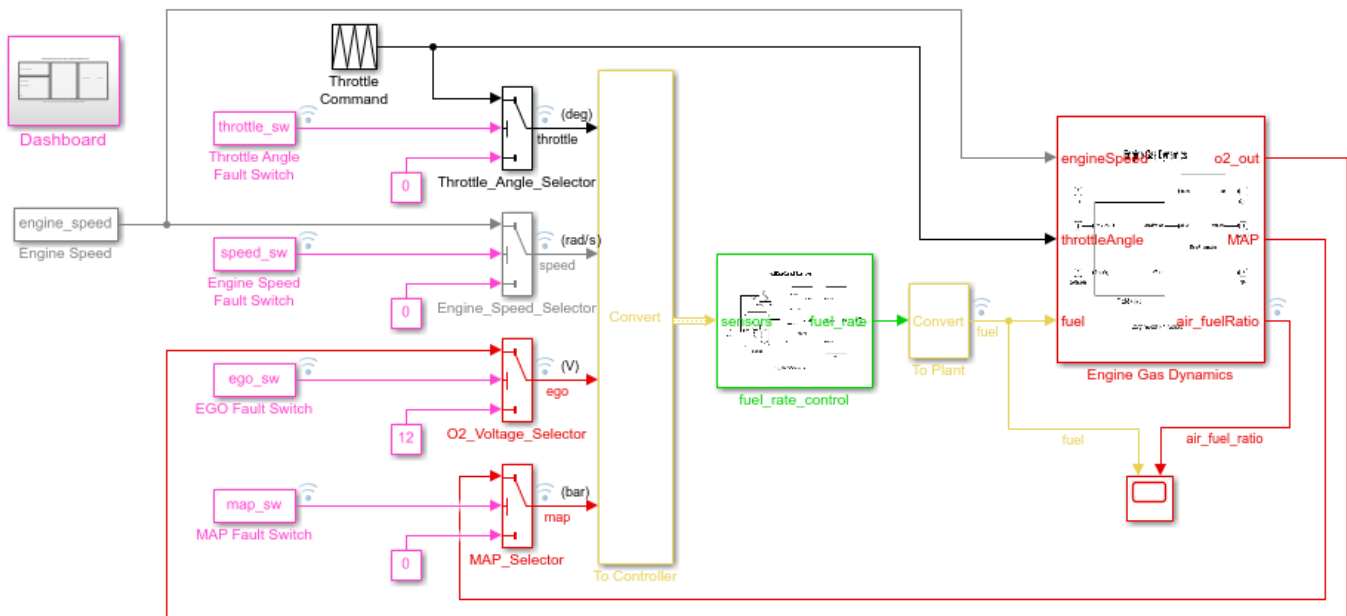
```
% Export model to Standalone Co-Simulation FMU 2.0
exportToFMU2CS('fmudemo_export_fuelsys_plant', 'CreateModelAfterGeneratingFMU', 'off', 'AddIcon')
```

You can use optional arguments **CreateModelAfterGeneratingFMU**, **AddIcon**, **SaveDirectory** to configure FMU export settings. For more information, call `help ExportToFMU2CS`.

Integrate FMU components in Simulink

Once both FMUs are successfully exported, you may use the top model `fmudemo_export_fuelsys_top` to fully integrate the system for testing.

Fault-Tolerant Fuel Control System



[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

Copyright 1990-2020 The MathWorks, Inc.

