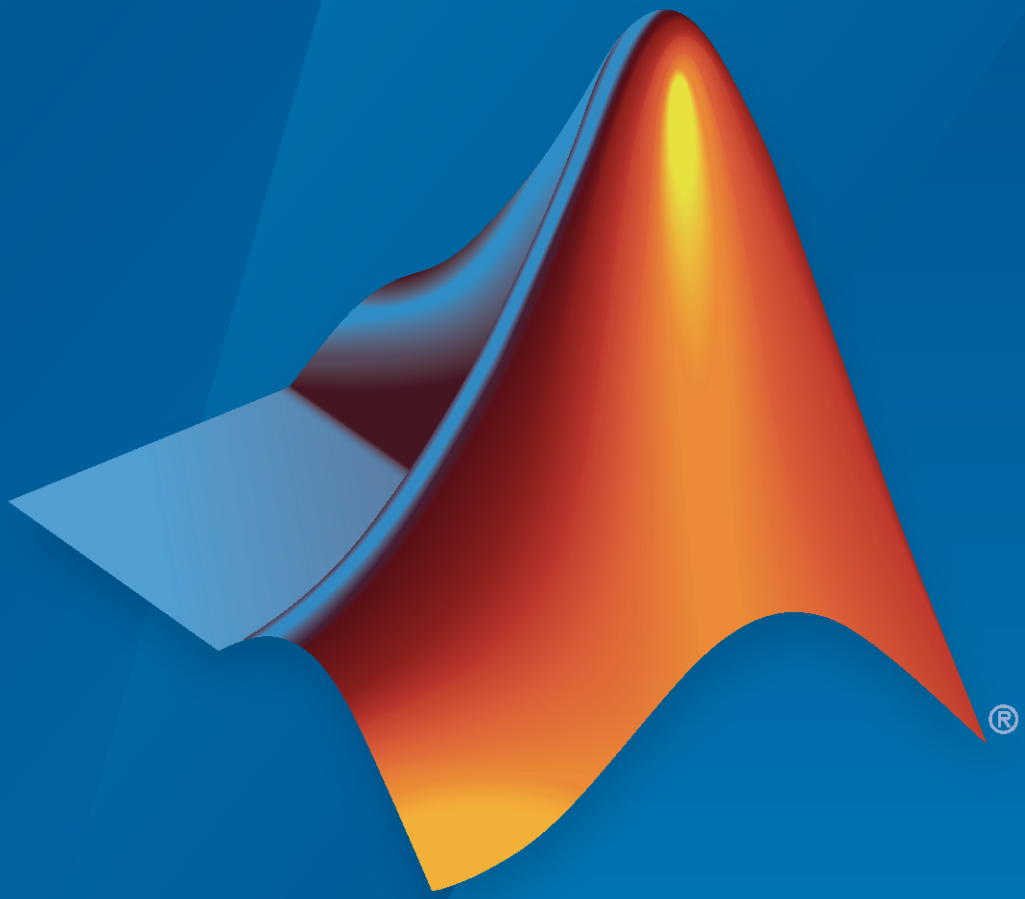
Simulink® Compiler™

Getting Started Guide



MATLAB® & SIMULINK®

R2022b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Simulink® Compiler™ Getting Started

Revision History

March 2020	Online only	New for Version 1.0 (Release 2020a)
September 2020	Online only	Revised for Version 1.1 (Release 2020b)
March 2021	Online only	Revised for Version 1.2 (Release 2021a)
September 2021	Online only	Revised for Version 1.3 (Release 2021b)
March 2022	Online only	Revised for Version 1.4 (Release 2022a)
September 2022	Online only	Revised for Version 1.5 (Release 2022b)

Contents

1 Simulink Compiler Getting Started

Simulink Compiler Getting Started

Simulink Compiler Product Description

Share simulations as standalone executables, web apps, and Functional Mockup Units (FMUs)

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 used 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.

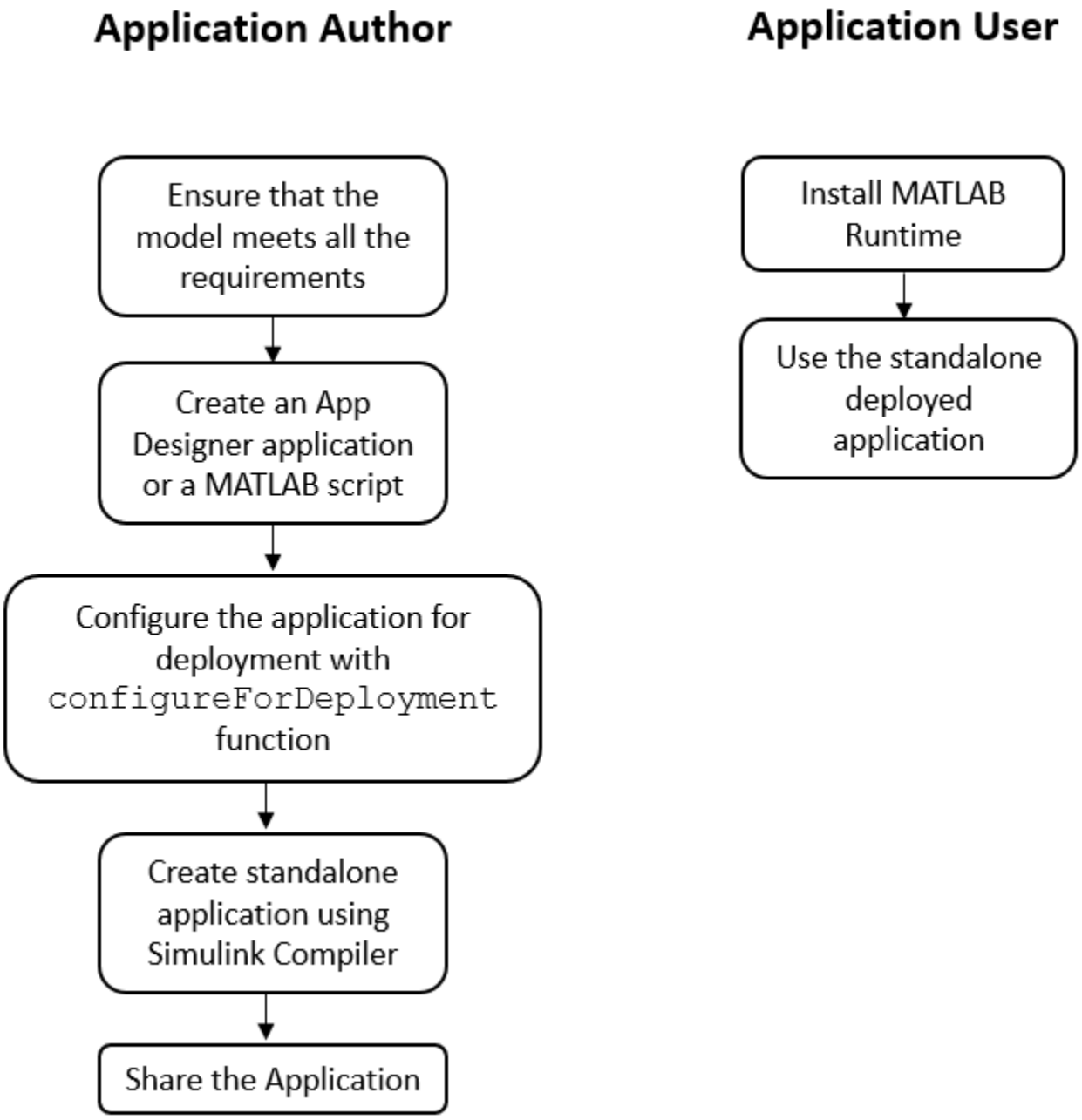
To provide browser-based access to your deployed simulation, you can create a web app and host it with MATLAB Web App Server™. Simulink simulations can be packaged into software components for integration with other programming languages (with MATLAB Compiler SDK™). Large-scale deployment to enterprise systems is supported through MATLAB Production Server™.

To generate C and C++ source code from Simulink, use Simulink Coder™.

Simulink Compiler Workflow Overview

Simulink Compiler lets you share simulations as standalone applications. Simulink Compiler extends the capabilities of MATLAB Compiler to allow Simulink `sim` command and associated Simulink functions in the deployed script or application. For more information about MATLAB Compiler, see MATLAB Compiler documentation.

The application users who use these deployed applications are not expected to interact with the Simulink model directly. The Simulink user provides the application user with a tool that allows them to explore task-specific scenarios without looking at the underlying model that represents the dynamic system. The application users can change model parameters and simulation inputs, and record and analyze simulation outputs.



To develop an application, the Simulink user:

1. Prepares the Simulink model to be compatible with Simulink Compiler, such as checking that the model simulates correctly in rapid accelerator mode. For limitations of rapid accelerator mode and Simulink Compiler, see “Rapid Accelerator Limitations”.

   Note For information on toolboxes supported by Simulink Compiler, see “Toolboxes Supported by Simulink Compiler” on page 1-7.

2. Creates an application that simulate the model using the `sim` command, in a script or an App designer app.
3. Configures the script or the app for deployment by using `simulink.compiler.configureForDeployment` function. The `simulink.compiler.configureForDeployment` function adapts the model to run in Rapid Accelerator mode.
4. Creates a standalone application using the `mcc` command or the `deploytool` app.
5. Shares the standalone application.

To use the application, the application user:

1. Installs MATLAB Runtime environment for the deployed application.
2. Uses the deployed application.

The following Simulink functions and classes are deployable:

Functions:

- `sim`
- `start_simulink`

Classes:

- `Simulink.SimulationInput` and its method `setVariable`
- `Simulink.SimulationOutput`
- `Simulink.SimulationData.Dataset`

Simulation Pacing in Rapid Accelerator Mode

Using simulation pacing in rapid accelerator mode allows you to use simulation pacing with the deployed standalone executables. You can slow down the simulation of your deployed executable to better visualize simulations and understand system behavior.

To use simulation pacing in rapid accelerator mode, or in the deployed executable, you can use the parameter `EnablePacing`. To use pacing for a model with the name `modelName`, use this:

```
% Enable pacing
set_param('modelName','EnablePacing','on')

%Set pacing rate
set_param('modelName','PacingRate',0.5)
```

Simulation pacing is turned off by default. The `EnablePacing` and the `PacingRate` parameters are saved with the model. Use of neither parameters dirties the model. `PacingRate` must be a non-zero positive integer.

If the model contains a Simulation Pacer block (Aerospace Blockset™), Simulink gives a warning during simulation in the Diagnostic Viewer or the MATLAB prompt. Simulation pacing takes place according to the `EnablePacing` and `PacingRate` parameters configuration. The simulation pace set on Simulation Pacer block is ignored.

Limitations

Simulation pacing is supported in command-line simulations and deployed mode only. In rapid accelerator mode, simulation pacing is not supported in the menu options in the Simulink Editor.

See Also

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

Related Examples

- “Create and Deploy a Script with Simulink Compiler” on page 1-9
- “Export Simulink Models to Functional Mock-up Units” on page 1-11
- “Toolboxes Supported by Simulink Compiler” on page 1-7

Toolboxes Supported by Simulink Compiler

Simulink Compiler supports the following toolboxes:

- Aerospace Blockset.
- Audio Toolbox™.
- Automated Driving Toolbox™.
- AUTOSAR Blockset.
- Communications Toolbox™.
- Computer Vision Toolbox™.
- Control System Toolbox™.
- Deep Learning Toolbox™: All blocks created from *gensim* that support code generation.
- DSP System Toolbox™.
- Fixed-Point Designer™: Fixed-point data point is supported.
- Fuzzy Logic Toolbox™.
- Model Predictive Control Toolbox™.
- Navigation Toolbox™.
- Phased Array System Toolbox™.
- Powertrain Blockset™.
- Robotics System Toolbox™.
- Simscape™.
- Simscape Driveline™.
- Simscape Electrical™.
- Simscape Fluids™.
- Simscape Multibody™.
- Simulink.
- Simulink Control Design™.
- Simulink Design Optimization™.
- Stateflow®.
- Symbolic Math Toolbox™.
- System Identification Toolbox™.
- Vehicle Dynamics Blockset™.
- Vehicle Network Toolbox™.
- Vision HDL Toolbox™.

Note Simulink Compiler supports all the blocks that support code generation in these toolboxes. You can use any blocks like the Current Measurement block in Simulink Compiler once you make sure to be able to run a model in Rapid Accelerator mode. Among these toolboxes, the prebuilt apps, UIs, and functions, and blocks that don’t support code generation are not supported by Simulink Compiler.

See Also

Related Examples

- “Create and Deploy a Script with Simulink Compiler” on page 1-9
- “Export Simulink Models to Functional Mock-up Units” on page 1-11

Create and Deploy a Script with Simulink Compiler

In this section...

In this example, you prepare a model to work with Simulink Compiler, develop and compile the script, and then deploy it as a standalone application.

Prepare the Model

Simulink Compiler uses rapid accelerator simulation targets to generate an executable to submit a Simulink model. Simulink Compiler only supports models which can run in rapid accelerator mode. To set the simulation mode of the model to rapid accelerator, use the model parameter `'SimulationMode'` with `SimulationInput` object. To enable simulation deployment of the model, your model must be supported by the Rapid Accelerator mode correctly.

Simulink Compiler only supports `sim` function syntax that takes `Simulink.SimulationInput` object and returns `Simulink.SimulationOutput` object.

If callbacks are present in the model, they are called during the build time of the application. However, once the application or the script is deployed, these callbacks are not invoked.

Write the Script to Deploy

After preparing the model, write the script that you would like to deploy. In this example, we use a model and change one of the tunable parameters in the script.

In the MATLAB Editor, create a function `deployedScript`. This example uses the model `sldemo_suspn_3dof`. In this function, create a `Simulink.SimulationInput` object for the model, `sldemo_suspn_3dof` and change the value of `Mb` with the `setVariable` method of the `Simulink.SimulationInput` object. To ensure that the model runs in rapid accelerator mode, set the `SimulationMode` to `Rapid` through the `setModelParameter` method of the `Simulink.SimulationInput` object or use the `simulink.compiler.configureForDeployment` function as shown below.

The variables modified in the simulations can be in the base workspace or in the top model workspace. If your model uses external input variables, that is, you can not use the method `in.setExternalInput` of the `Simulink.SimulationInput`. External input variables must be in the MATLAB workspace before packaging for deployment.

```
function deployedScript()
    in = Simulink.SimulationInput('sldemo_suspn_3dof');
    in = in.setVariable('Mb', 1000);
    in = simulink.compiler.configureForDeployment(in);
    out = sim(in);
end
```

Save the function as a `deployedScript.m`.

Compile Script for Deployment

Before compiling the script that you want to deploy, ensure that the files for the model and script, in this case `sldemo_suspn_3dof` and the `deployedScript.m`, are included on the MATLAB search path. To compile the script, use the `mcc` command with the script name. To learn more about the `mcc` command, see `mcc`.

```
mcc -m deployedScript.m
```

Troubleshooting Tips

Simulink Compiler automatically packages the dependencies in the model and the deployed scripts. If the command `mcc` cannot find a dependency, you might see errors.

- 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`.

Run the Deployed Script

Install MATLAB Runtime

To run the deployed executable, you need an appropriate runtime environment. To install the MATLAB Runtime, see https://www.mathworks.com/products/compiler/matlab-runtime.html.

Run the Deployed Application

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

Run the deployed application from the Windows command prompt. Running the deployed application from the command prompt enables the application to print diagnostic messages in the command prompt when it encounters errors. These messages can be a helpful tool in troubleshooting the problem.

See Also

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

Related Examples

- “Deploy an App Designer Simulation with Simulink Compiler”
- “Deploy Simulations with Tunable Parameters”

Export Simulink Models to Functional Mock-up Units

Export Models

Export Simulink models to functional mockup unit (FMU) that supports co-simulation in FMI version 2.0. To check that the exported block is still a valid Simulink model, you can also direct the software to import the FMU back to a Simulink model as part of the export process.

Requirements include:

- Simulink Compiler
- A writable folder into which to place the exported FMU.

Exported models can have:

- Input and output data types: `double`, `int32`, `boolean`, `string`
- Matrices
- Bus Signals
- Tunable parameters which can be model arguments, base workspace, or data dictionary variables.
- Unit and description.

When you export a model as a standalone FMU, certain metadata from Simulink is also exported with the FMU. The metadata includes:

- Model description
- Signal unit
- Parameter unit
- Signal description
- Parameter description

Standalone FMU

Simulink models can be exported to standalone co-Simulation FMU in version 2.0. The generated FMU package contains the following files:

- `modelDescription.xml`
- `model.png (optional)`
- `binaries\win64\modelname.dll`, or `binaries\linux64\modelname.so`, or `binaries\darwin64\modelname.dylib`

You might experience an expected time delay in the exported FMU for co-Simulation mode.

FMU Variables

FMU `modelDescription.xml` file contains interfacing variables converted from Simulink model:

- Variables with `causality='input'`: converted from root Inport block
- Variables with `causality='output'`: converted from root Outport block
- Variables with `causality='parameter'`: converted from referenced Runtime Tunable Parameters

- Independent variable `'time'`

To generate FMU input and output, define root Inport and Outport blocks in Simulink model. The name of the generated variable is converted from root Inport or Outport block name, by removing special and blank characters and avoiding duplicates. If input/output signal carries unit information, it is exported as `Unit` attribute of the FMU variable. If the input/output block has a non-empty description information under **Block Properties** > **General** , it is exported as `Description` attribute of the FMU variable.

The following input and output data types are supported:

- `double` (Real in FMI)
- `int32` (Integer in FMI)
- `boolean` (Boolean in FMI)
- `string` (String in FMI)

If model root Inport or Outport block is a non-virtual bus, individual bus elements will be expanded to variables using structured naming convention (`'.'`). If model root Inport or Outport block is array or matrix,, individual scalar elements will be expanded to variables using array naming convention (`'[]'`).

To export referenced variables as FMU parameter, you can:

- Define a variable.
- Define a Simulink Parameter object.

Ensure that the variable and the parameter object is directly references by tunable parameters of Simulink blocks. In FMU Export dialog, expand **Parameter Details...** to configure each parameter. You can:

- Unselect **Exported** option to hide a parameter
- Modify **Exported Name** so the parameter is displayed with a different name on FMU interface. Do not use special characters and duplicate names.
- Set `Unit` and `Description` of FMU parameter variable by clicking on parameter name, and directly modifying the parameter object

  If the FMU parameter is `Simulink.Parameter`, click the hyperlink to modify the `Unit` and `Description` of the variable.

  If FMU parameter is a regular MATLAB variable, clicking the hyperlink opens model explorer. You can convert MATLAB variable to a `Simulink.Parameter` so that it can carry `Unit` and `Description`.

  `Unit` and `Description` of FMU parameter variable cannot be updated directly in FMU Export dialog. You can configure `Unit` and `Description` through model explorer, double-clicking `Simulink.Parameter` in base workspace, etc.

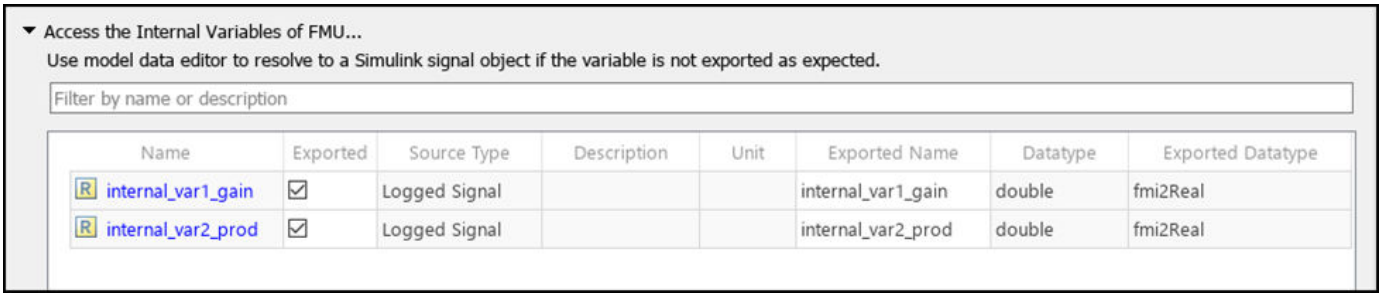
The following parameter data types are supported:

- `double` (Real in FMI)
- `int32` (Integer in FMI)
- `boolean` or logical (Boolean in FMI)

If referenced parameter is a `struct`, individual `struct` members will be expanded to variables using structured naming convention (`'.'`). If referenced parameter is array or matrix, individual scalar elements will be expanded to variables using array naming convention (`'[]'`).

When a Simulink model with model reference block is exported to FMU, you can also export base workspace variables, model arguments and instance parameters that are promoted from the sub model.

On the Simulink toolstrip, under **Save**, select **Export Model to Standalone FMU** to view options for exporting an FMU with internal variables.



FMU Solver

Fixed-step solvers are supported for standalone FMU export. It is recommended to set a fixed fundamental sample time (**Solver > Solver details > Fixed-step size**) before exporting the model. When simulating the standalone FMU in another environment, communication step-size must be an integral multiple of the fundamental sample time.

FMU Dynamic Library

A generated FMU contains a dynamic library build for the current platform. The default `fmi2TypesPlatform` value is used.

All required and optional `fmi2` functions defined by FMI standard can be invoked. However, the following functions have no operation and return `fmi2OK` immediately:

- Model-Exchange functions
- Functions accessing or serializing `FMUstate`
- Functions setting or getting input or output derivatives
- Functions querying `fmi2DoStep` status, or cancelling `fmi2DoStep`
- Function computing directional derivatives of variables

Save Source Code with FMU Export

You can export a Simulink model to FMU along with C source code. You can check **Save Source Code** in the **Export Model to FMU Co-Simulation** window or use the command `exportToFMU2CS('mdlName','SaveSourceCodeToFMU','on')` to export the model to FMU with C source code.

Note To export a Simulink model to FMU with C source code, install Simulink Coder

If the Simulink model contains model references with custom data types or fixed-point functions, exporting FMU with source code may cause an error due to duplicate header files in the `_sharedutils` folder. Follow instructions on Generate Shared Utility Code to set the **Code Generation > Interface > Shared Code Placement** parameter to '`Shared Location`' and regenerate the FMU.

You can export a Simulink model with a FMU Import blocks as nested standalone FMU. When exporting a nested FMU, Simulink packs all dependent inner FMUs into the resources/ folder of the nested FMU. When the nested FMU is instantiated in a simulation environment, all inner FMUs will share the same callback functions provided by the environment, for example, logger and memory allocation functions.

Specify Additional Files

While exporting a Simulink to a standalone FMU, you can specify additional files to be included in the generated FMU, such as resource, DLL etc. The target locations for these files can be:

- `<fmuroot>/binaries/<arch>/` – dependent DLLs
- `<fmuroot>/resources/` – data files, lookup tables, etc
- `<fmuroot>/documentation/` – user provide their own help content

For an example on specifying additional files while exporting a Simulink model, see “Export Simulink Model to Standalone FMU with User Specified Files and Archived Project with Harness Model” on page 1-37.

Export Protected Model

You can export a Simulink model that is protected. For an example on exporting protected models, see “Export Simulink Model with Protected Model and FMU Import Block to Standalone FMU” on page 1-24.

Limitations

You cannot generate FMU from a Simulink model, due to these limitations:

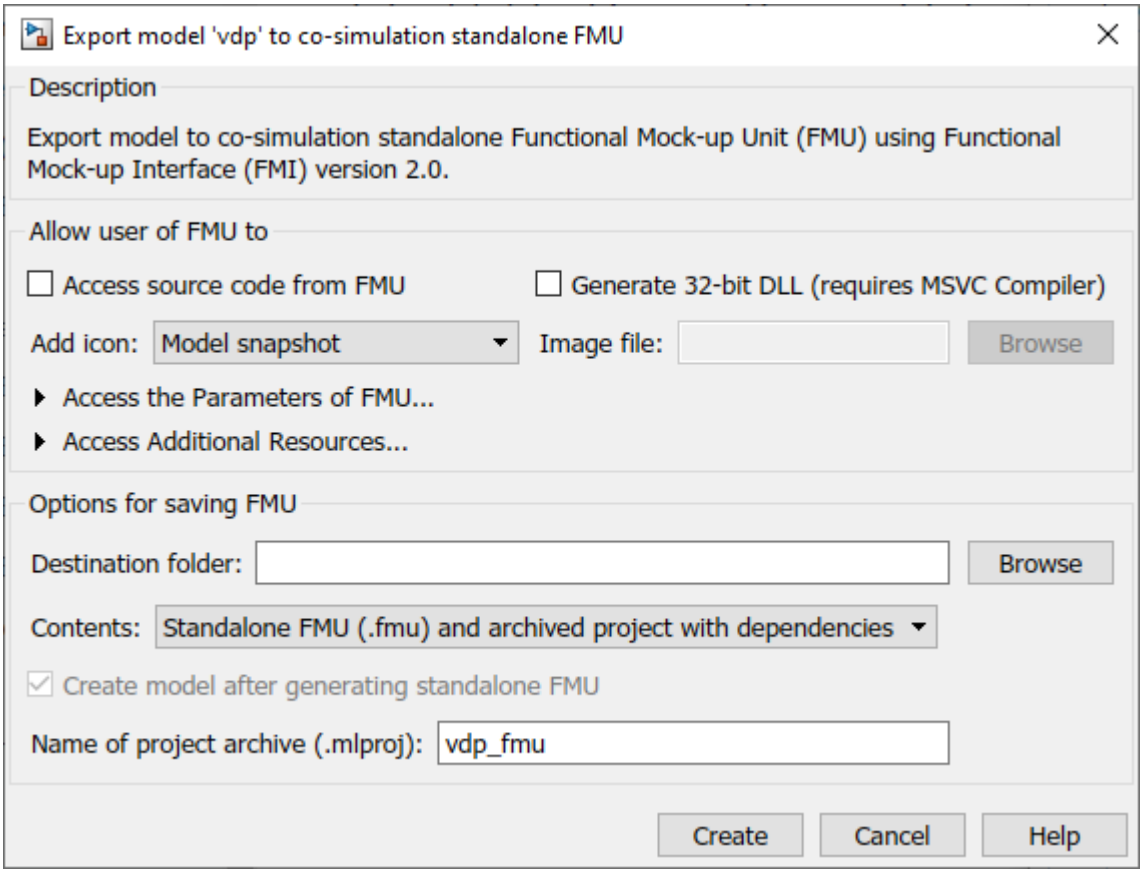
- Variable-step solvers are not supported.
- Non-zero simulation start time is not supported.

Export a Simulink Model

Use the Export Dialog Box

Export the `vdp` example using the Simulink toolstrip: **Simulation > Save > Export Model ToStandalone FMU**

1. Open the model `vdp`
2. In the Simulink Editor, navigate to **Simulation > Save > Standalone FMU**.
3. In Simulink Editor, select **Save > Export to > FMU Co-Simulation**.
4. In the export dialog box, specify the path to export the FMU.



5 Click **Create**

By default, Simulink creates the FMU and a harness model with its dependencies stored in a `MAT` file. It then packs them into archived project (`.mlproj`). You can change the behavior by setting **Contents** option to **Standalone FMU**.

Use the Programmatic Interface

- Export the `vdp` example to an FMU using the default `exportToFMU2CS` function. This command creates the FMU file `modelName.fmu`. By default, the command also creates a Simulink model `modelName_fmu.slx`, that contains an FMU Co-Simulation block with the original model. Create this model if you want to check the integrity of the exported FMU.

```
load_system('vdp')
set_param('vdp', 'SolverType', 'Fixed-step')
exportToFMU2CS('vdp')
```

- Export the `vdp` example to an FMU using the `exportToFMU2CS` function, but do not create a Simulink model. This command creates the FMU file `modelName.fmu`.

```
load_system('vdp')
set_param('vdp', 'SolverType', 'Fixed-step')
exportToFMU2CS('vdp','CreateModelAfterGeneratingFMU','off')
```

- Export the `vdp` example to an FMU using the `exportToFMU2CS` function. Create a model for the FMU and use an image of the original model as the block icon. This command creates the FMU file, `modelName.fmu` and a Simulink model with an FMU Co-Simulation block whose block icon is the original model.

```
exportToFMU2CS('vdp','AddIcon','snapshot')
```

Examples For Different Workflows

The examples below illustrate how to use FMU export for all different scenarios:

- “Export Simulink Model to Standalone FMU”
- “Export Standalone FMU with External C++ Code” on page 1-17
- “Export Simulink Model with Protected Model and FMU Import Block to Standalone FMU” on page 1-24
- “Export Simulink Model to Standalone FMU with User Specified Files and Archived Project with Harness Model” on page 1-37
- “Export Simulink Model to Standalone FMU with Source Code” on page 1-49

See Also

`exportToFMU2CS` | `configureForDeployment` | `Simulink.SimulationInput` | `mcc` | `deploytool` | `sim`

Related Examples

- “Deploy Simulations with Tunable Parameters”
- “Deploy an App Designer Simulation with Simulink Compiler”
- “Simulation Callbacks for Deployable Applications”

Export Standalone FMU with External C++ Code

This example shows how to import external C++ code into Simulink® model using S-Function Builder and export it to a standalone FMU. S-Function Builder block lets user import C/C++ code into Simulink Semantic by building an S-function wrapper for external code. This example demonstrates this process in a phased approach by implementing a C++ multiply class, integrating C++ code with S-Function Builder, and exporting the model as a standalone FMU.

Implement Multiply Class with C++ Code

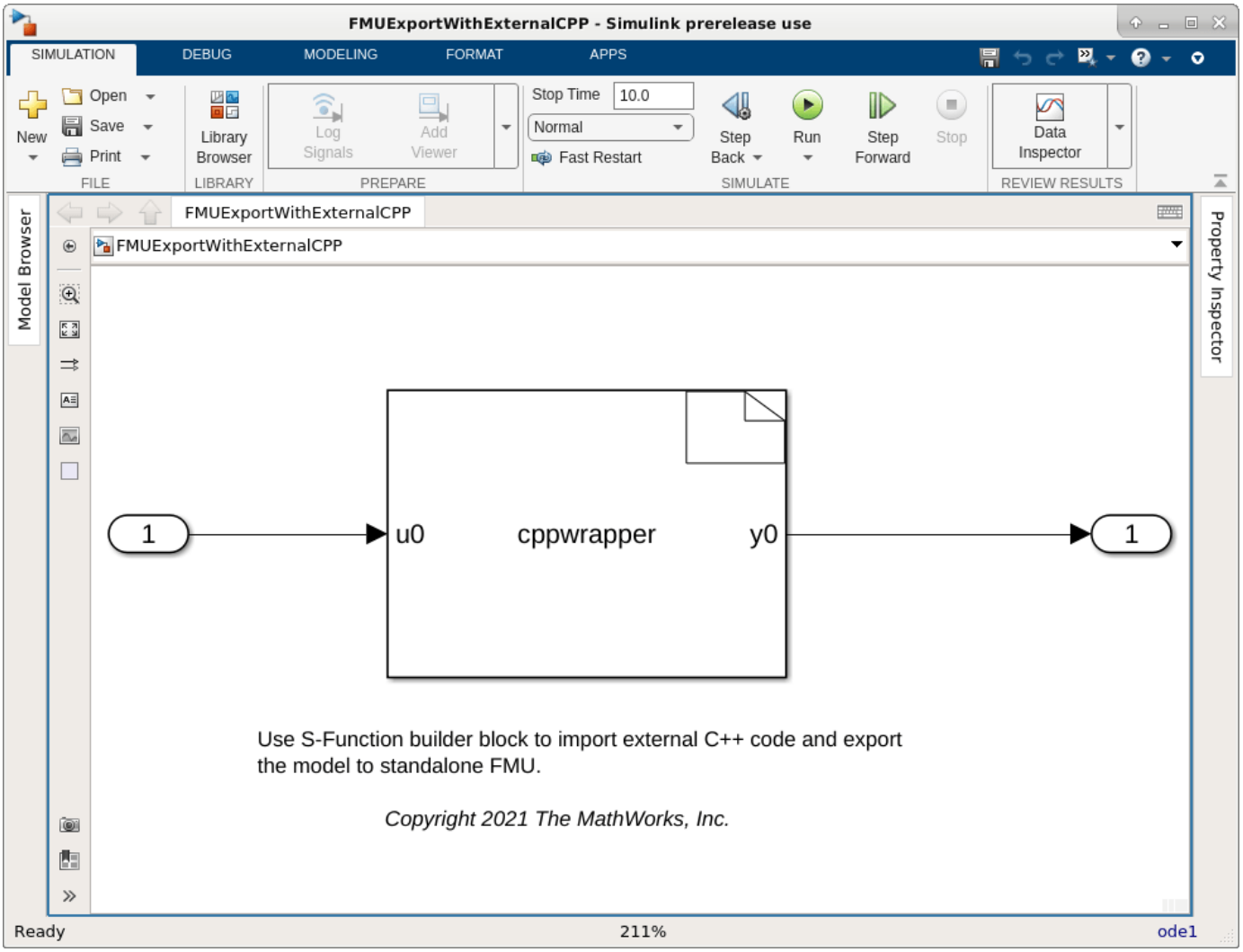
The following code implements a class *multiply* to be integrated into Simulink model. Class *multiply* takes a gain value in the constructor and multiplies it with an input value when the user calls member function *double multiply::apply(double)*. The following implementation can be found in *include/* and *src/* directory.

```
// multiply class header, the following code is defined in include/multiply.hpp

class EXPORT multiply {
public:
    multiply(double init);
    ~multiply() = default;

    double apply(int val);
private:
    double gain;
};

// multiply class source, the following code is defined in src/multiply.cpp

#include "multiply.hpp"

multiply::multiply(double init) {
    gain = init;
}

double multiply::apply(double val) {
    return val * gain;
}
```

Import External C++ Code with S-Function Builder

This section shows the process to integrate external C++ code into Simulink model using S-function Builder block:

- Open Simulink model with an S-Function Builder block.
- Instantiate class *multiply* in S-Function Output function.
- Add C++ code path to S-Function Builder block **Libraries** Table.




```
% Open example model with S-Function Builder block
open_system('FMUExportWithExternalCPP');

% Open S-Function Builder block dialog
open_system('FMUExportWithExternalCPP/S-Function Builder');
```

Use S-Function Builder block to include *multiply.hpp* and instantiate C++ class *multiply* in corresponding wrapper functions. The wrapper functions below instantiate and destroy an instance of class *multiply* in *void cppwrapper_Start_wrapper(const real_T*, const int_T, void**)* and *void cppwrapper_Outputs_wrapper(const real_T*, real_T*, const real_T*, const int_T, void**)*. S-Function Builder block uses a dialog parameter defined in **Parameter** Table to instantiate an instance of class *multiply* and creates a PWork to store the pointer.

```
// add the following code to include header
#include "multiply.hpp"

// add the following code to void cppwrapper_Start_wrapper(const real_T*, const int_T, void**)
// the code below takes parameter from S-Function Builder block dialog to instantiate class multi
    real_T val = p0[0];
    multiply* mulPtr = new multiply(val);
```

```
    pW[0] = mulPtr;

// add the following code to void cppwrapper_Outputs_wrapper(const real_T*, real_T*, const real_T
// gain input value
    multiply* mulPtr = static_cast<multiply*>(pW[0]);
    y0[0] = mulPtr->apply(u0[0]);

// add the following code to void cppwrapper_Terminate_wrapper(const real_T*, const int_T, void**
// instance of class multiply is destroy when simulation terminates
    multiply* mulPtr = static_cast<multiply*>(pW[0]);
    delete mulPtr;
```

A complete example code is shown below:

```
/* Includes_BEGIN */
#include <math.h>
#include "multiply.hpp"
/* Includes_END */

/* Externs_BEGIN */

/* Externs_END */

void cppwrapper_Start_wrapper(const real_T *p0, const int_T p_width0,
                              void **pW)
{
/* Start_BEGIN */
    real_T val = p0[0];
    multiply* mulPtr = new multiply(val);
    pW[0] = mulPtr;
/* Start_END */
}

void cppwrapper_Outputs_wrapper(const real_T *u0,
                                real_T *y0,
                                const real_T *p0, const int_T p_width0,
                                void **pW)
{
/* Output_BEGIN */
    multiply* mulPtr = static_cast<multiply*>(pW[0]);
    y0[0] = mulPtr->apply(u0[0]);
/* Output_END */
}

void cppwrapper_Terminate_wrapper(const real_T *p0, const int_T p_width0,
                                  void **pW)
{
/* Terminate_BEGIN */
    multiply* mulPtr = static_cast<multiply*>(pW[0]);
    delete mulPtr;
/* Terminate_END */
}
```

S-Function Builder block requires the *include* and *source* directory to build external C++ code. User can define C++ file path and entry in **Libraries** table of S-Function builder and specify a target language from the **Language** setting combobox. For S-Function Bluilder block reference, please see: Create an S-Function Builder Block and Specify Settings.

In this example, we add the following path to S-Function Builder **Libraries** Table. Click **Build** button on the S-Function Builder dialog to build the code.


```
% add c++ source and header to library table
handle = getSimulinkBlockHandle('FMUExportWithExternalCPP/S-Function Builder');
Simulink.SFunctionBuilder.add(handle,"LibraryItem","LibraryItemTag","INC_PATH","LibraryItemValue"
Simulink.SFunctionBuilder.add(handle,"LibraryItem","LibraryItemTag","SRC_PATH","LibraryItemValue"
Simulink.SFunctionBuilder.add(handle,"LibraryItem","LibraryItemTag","ENTRY","LibraryItemValue",'m

% build s-function
Simulink.SFunctionBuilder.build(handle);

Generating 'cppwrapper.cpp' ....Please wait
Compiling 'cppwrapper.cpp' ....Please wait
### 'cppwrapper.cpp' created successfully
### 'cppwrapper_wrapper.cpp' created successfully
### S-function 'cppwrapper.mexw64' created successfully
```

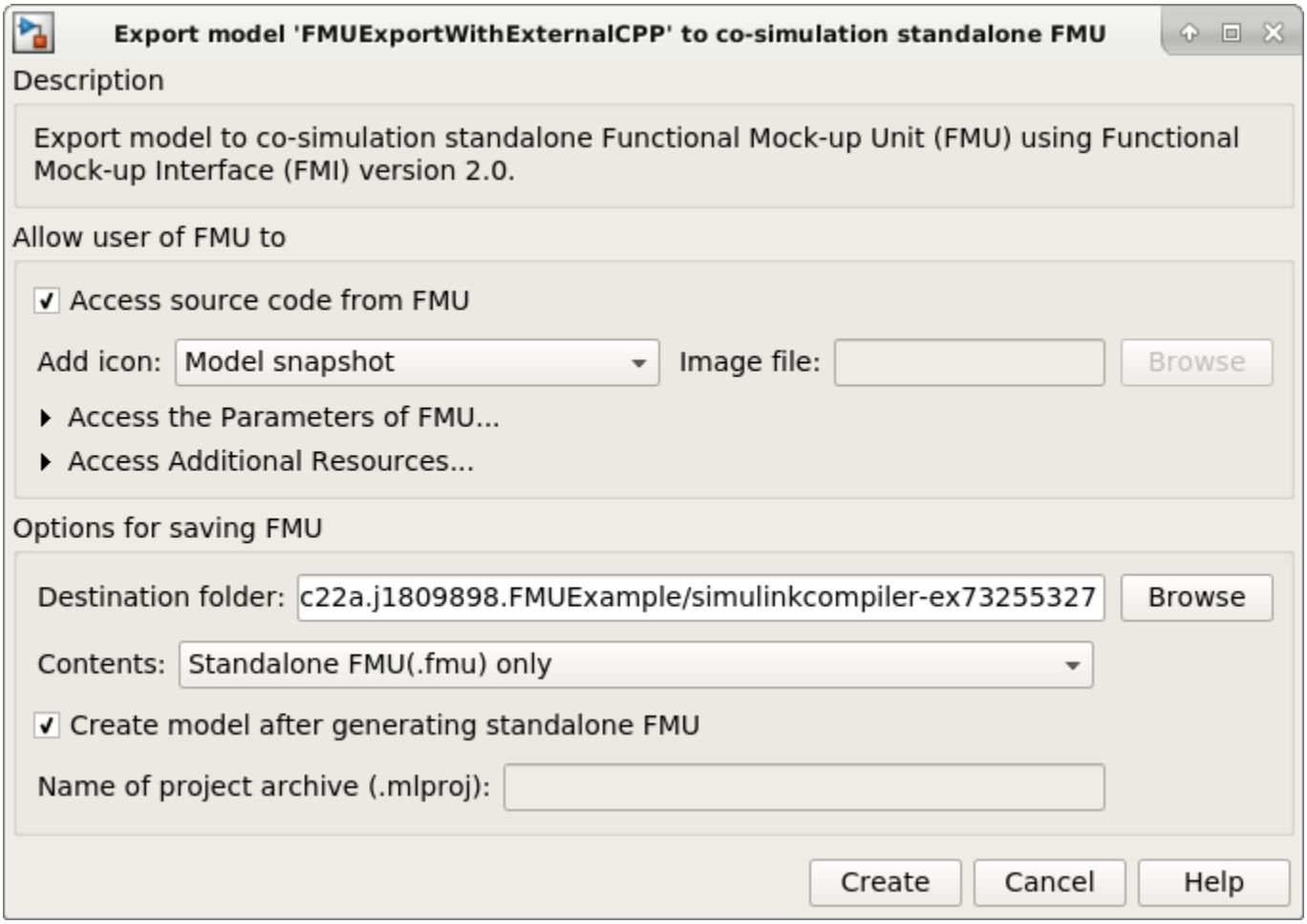
The Libraries Table also allows user to specify external shared/static library referenced by custom code. For more information on how to add external libraries, see Use the Libraries Table to Specify External Code and Paths.

Note: Shared libraries dependencies in standalone FMU may have symbol clashing, library loading order conflicts, and data racing issues which result in simulation error.

Export Simulink Model as Standalone FMU

To build and export your model to a standalone FMU, click drop-down button for **Save** from **Simulation** tab and select **Standalone FMU**.

The figure below shows Export Standalone FMU dialog, user can pack source code into FMU and generate model harness after export. Read more about the Standalone FMU export functionality: Export Simulink Model to Standalone FMU.




```
% Export model to Standalone Co-Simulation FMU 2.0
exportToFMU2CS('FMUExportWithExternalCPP', 'CreateModelAfterGeneratingFMU', 'on');

Setting System Target to FMU Co-Simulation for model 'FMUExportWithExternalCPP'.
Setting Hardware Implementation > Device Type to 'MATLAB Host' for model 'FMUExportWithExternalCP
### 'GenerateComments' is disabled for Co-Simulation Standalone FMU Export.

Build Summary

Top model targets built:

Model                     Action                        Rebuild Reason
==========================================================================================
FMUExportWithExternalCPP  Code generated and compiled.  Code generation information file does not

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 32.402s
### Model was successfully exported to co-simulation standalone FMU: 'C:\TEMP\Bdoc22b_2054784_600
```

A standalone FMU is generated in the **Destination folder** specified from the export dialog. User can also use **Access source code from FMU** option to pack source code into FMU package. Please note that FMU only accepts target landguage to be C in *R2022a*. Mixed compilng C++ in S-Function

Builder block and Simulink model results S-Function Builder block generate wrapper function with C calling convention ("extern C").

```
// the following code marks function name in C++ have C linkage
extern "C" {
    void cppwrapper_Start_wrapper(const real_T*, const int_T, void**);
    void cppwrapper_Outputs_wrapper(const real_T*, real_T*, const real_T*, const int_T, void**);
    void cppwrapper_Terminate_wrapper(const real_T*, const int_T, void**);
}
```

```
% Close all model
bdclose all;
```

Export Simulink Model with Protected Model and FMU Import Block to Standalone FMU

This example shows how to export Simulink® model with external references to Standalone FMU. In this example, the model `f14_flight_control` demonstrates exporting protected model and FMU Import block to a Standalone FMU.

Simulink Compiler™ license is required for standalone FMU Export and Simulink Coder™ license is required to create protected models.

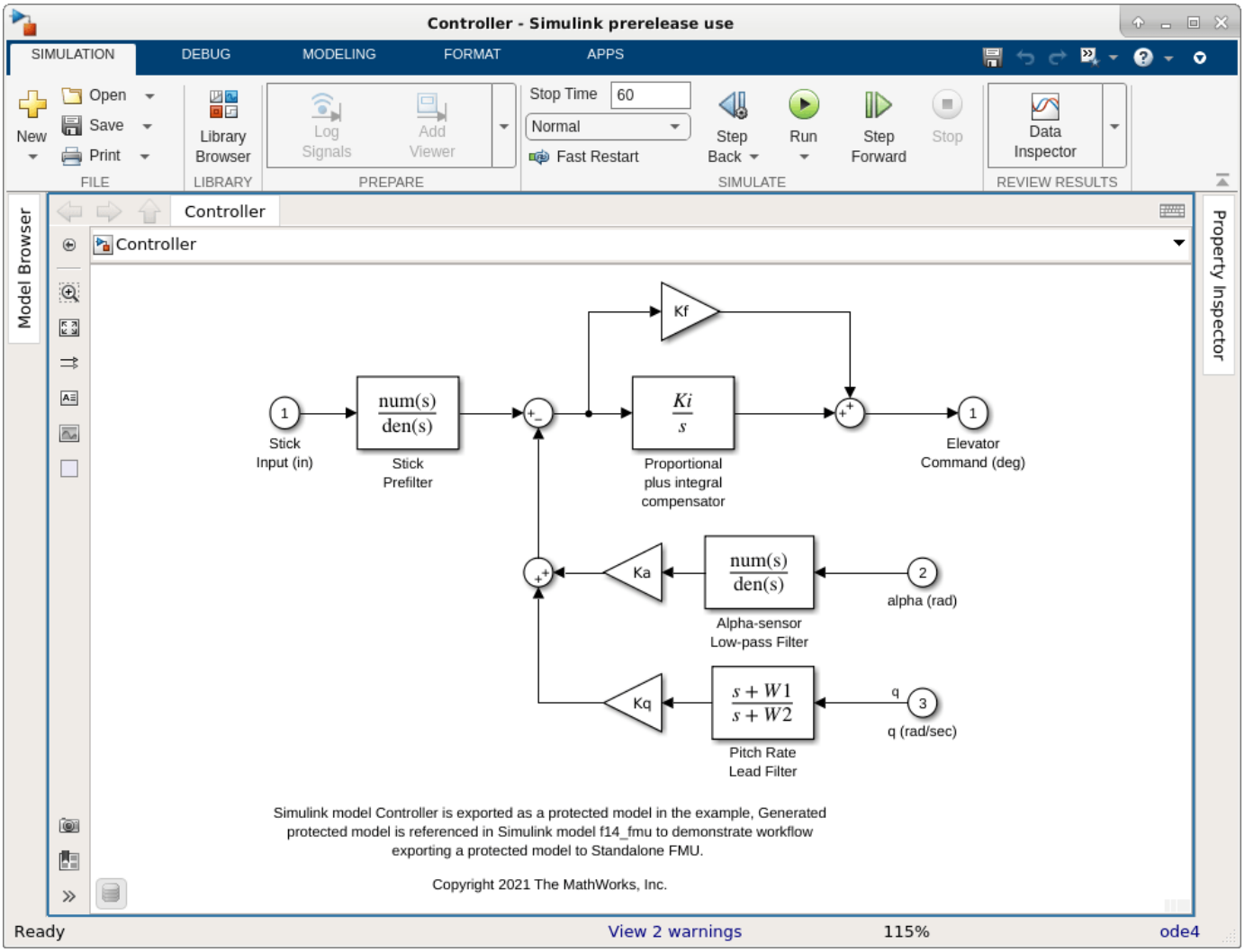
The example consists of three steps:

- Export Simulink Model to Protected Model with FMU Code Generation Capability on page 1-24
- Export Simulink Model to Standalone FMU with Tunable Parameters on page 1-29
- Export Simulink Model with External References to Standalone FMU on page 1-32

Export Simulink Model to Protected Model with FMU Code Generation Capability

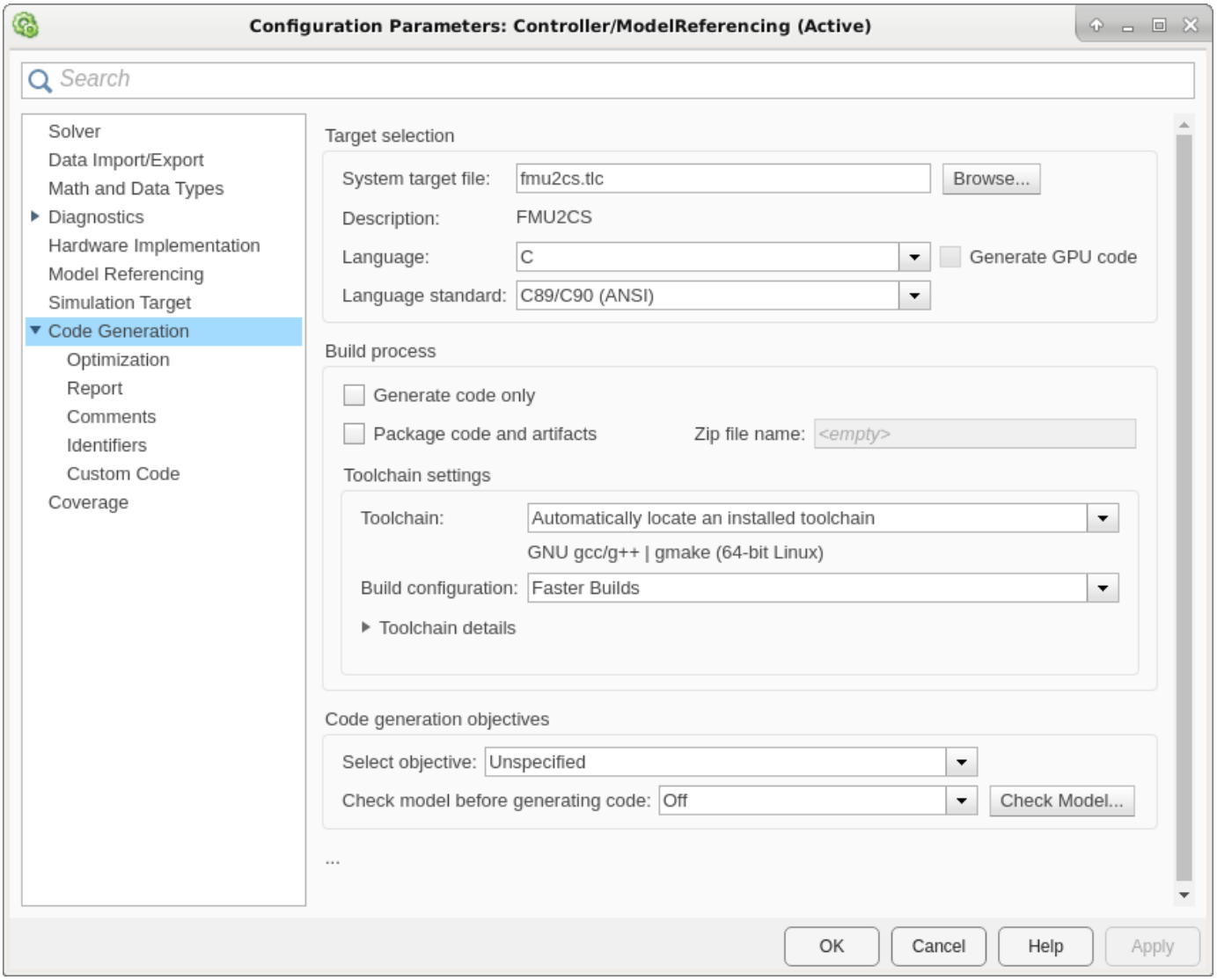
In this section, the following steps illustrate how to enable FMU Code Generation capability of a protected model for FMU export. FMU code generation artifacts must be packed into protected model before exporting to standalone FMU. Protected model author can use the following steps to create a protected model with FMU code generation artifacts. Alternatively, protected model author can add the code generation artifacts to an existing protected model with API *Simulink.ProtectedModel.addTarget*. If your protected model does not contain FMU code generation artifacts, please contact the protected model author. Use *Simulink.ProtectedModel.getSupportedTargets* to get a list of targets that protected model supports.

This example opens model `Controller`, sets system target file to `fmu2cs.tlc`, and exports model to protected model.



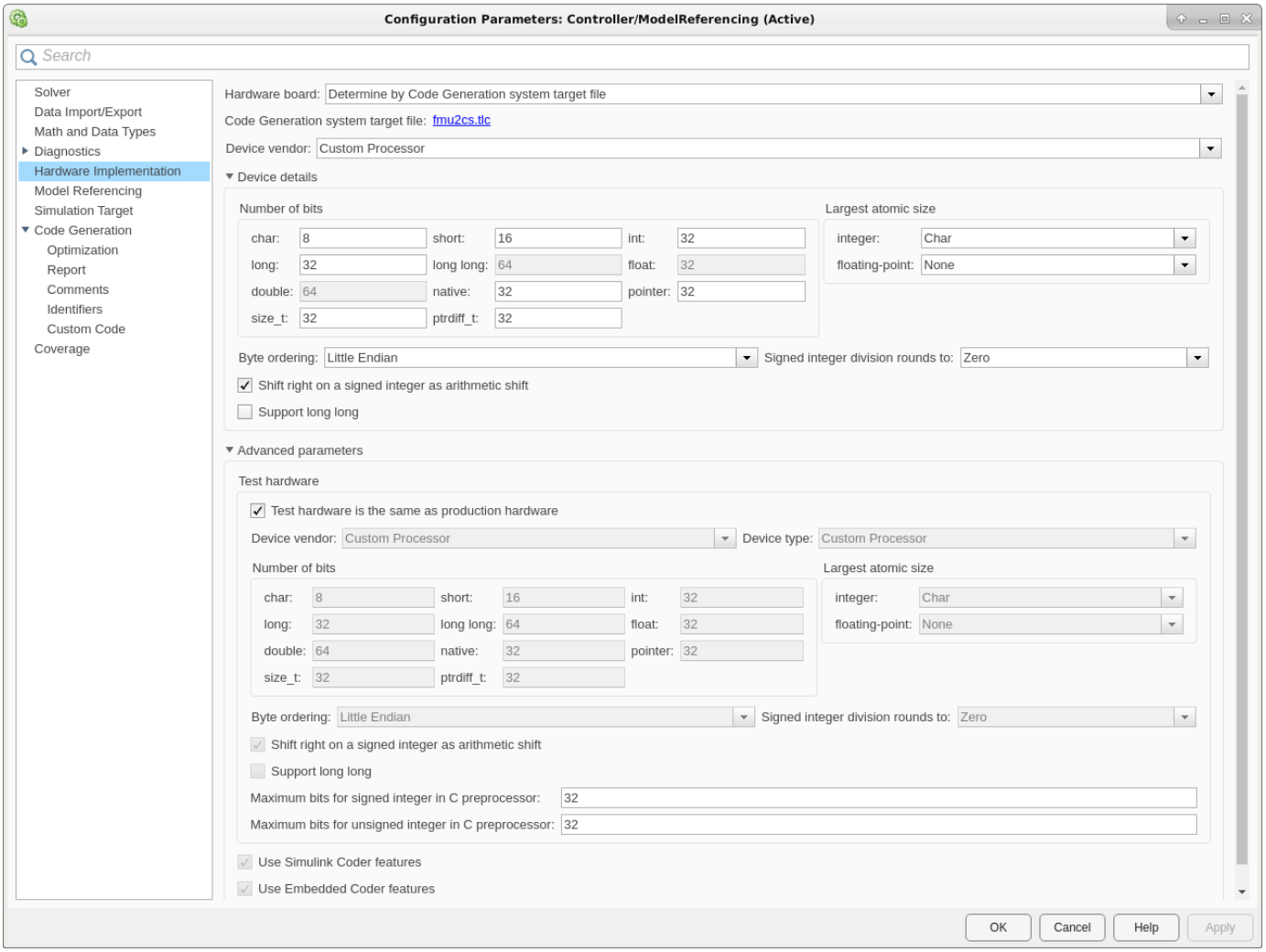

```
% Open example model Controller
model = "Controller";
open_system(model);
```

After the model is opened, go to the configuration dialog and update **System target file** to `fmu2cs.tlc` in **Configuration Parameter > Code Generation**.



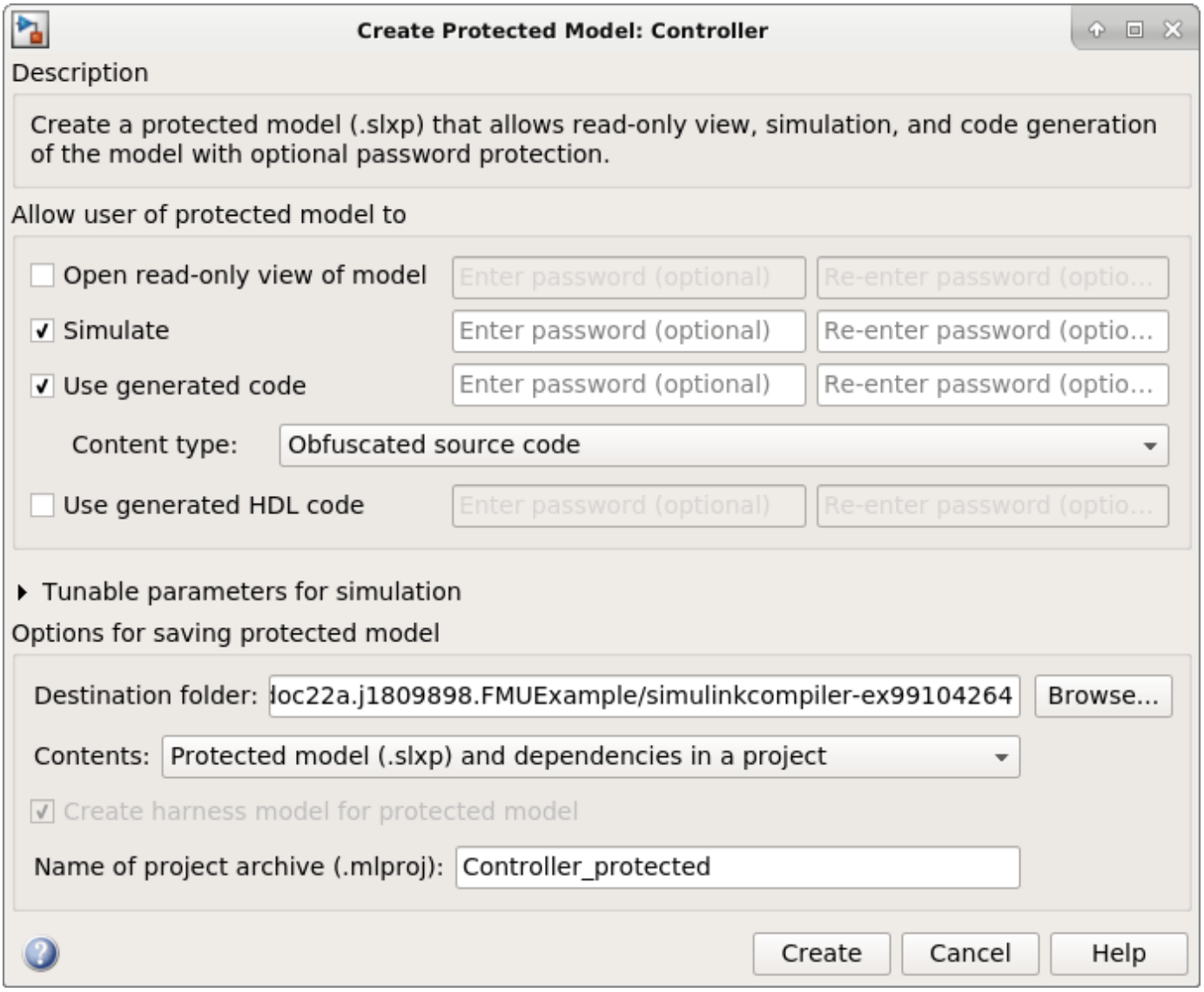

```
% alternative command-line option to set protected model code generation target to fmu2cs.tlc
% this allows the protected model to be exported as standalone FMU
set_param(model, 'SystemTargetFile', 'fmu2cs.tlc');
```

Inconsistent hardware implementation of hardware attributes can result in failure when exporting standalone FMU. To configure these parameters, user can open **Configuration Parameter > Hardware Implementation**. Model `Controller` uses the following setting for code generation. The same setting is used in model `f14_flight_control`.



Generate Protected Model from **Simulation** tab and select **Save > Protected Model**. A dialog box opens where user can select options for creating a protected model.

To enable code generation for standalone FMU export, check **Use Generated Code** and select **Obfuscated source code** in **Content type**.



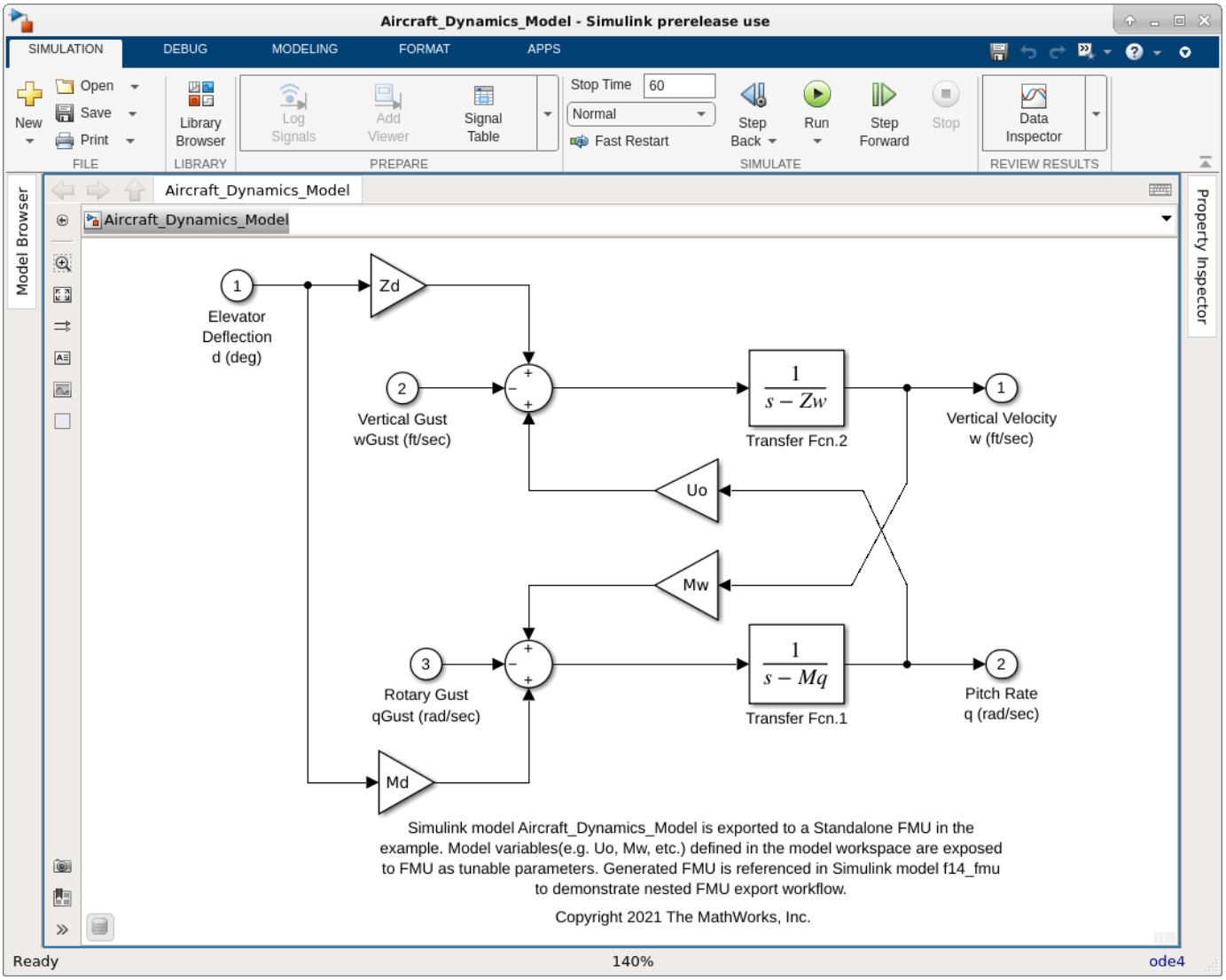

```
% generate protected model
Simulink.ModelReference.protect(model,'Mode','CodeGeneration','ObfuscateCode',true);

### Starting serial model reference simulation build.
### Successfully updated the model reference simulation target for: Controller

Build Summary

Simulation targets built:

Model       Action                          Rebuild Reason
========================================================================================
Controller  Code generated and compiled.  Controller_msp.mexw64 does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 1m 16.34s
### Starting serial model reference code generation build.
```

```
### Checking status of model reference code generation target for model 'Controller'.
### Model reference code generation target (Controller.c) for model Controller is out of date bec
### Setting Hardware Implementation > Device Type to 'MATLAB Host' for model 'Controller'.
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4
### Invoking Target Language Compiler on Controller.rtw
### Using System Target File: B:\matlab\toolbox\shared\simulink\fmuexport\fmu2cs.tlc
.............................................................### Saving binary information ca
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4155_824f_5f5833cbf284\slp
### Building 'Controller_rtwlib': nmake  -f Controller.mk all

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4155_824

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4155_824

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4155_824

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4155_824

Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
Controller.c
### Creating static library ".\Controller_rtwlib.lib" ...
    lib /nologo -out:.\Controller_rtwlib.lib @Controller.rsp
### Created: .\Controller_rtwlib.lib
### Successfully generated all binary outputs.

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc22b_2054784_6060\ibB18F8B\31\tp7e0cba53_aa9d_4155_824

Build Summary

Code generation targets built:

Model       Action                         Rebuild Reason
========================================================================
Controller  Code generated and compiled.  Controller.c does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 22.64s
```

```
% close model after protected model is generated
close_system(model);
```

Export Simulink Model to Standalone FMU with Tunable Parameters

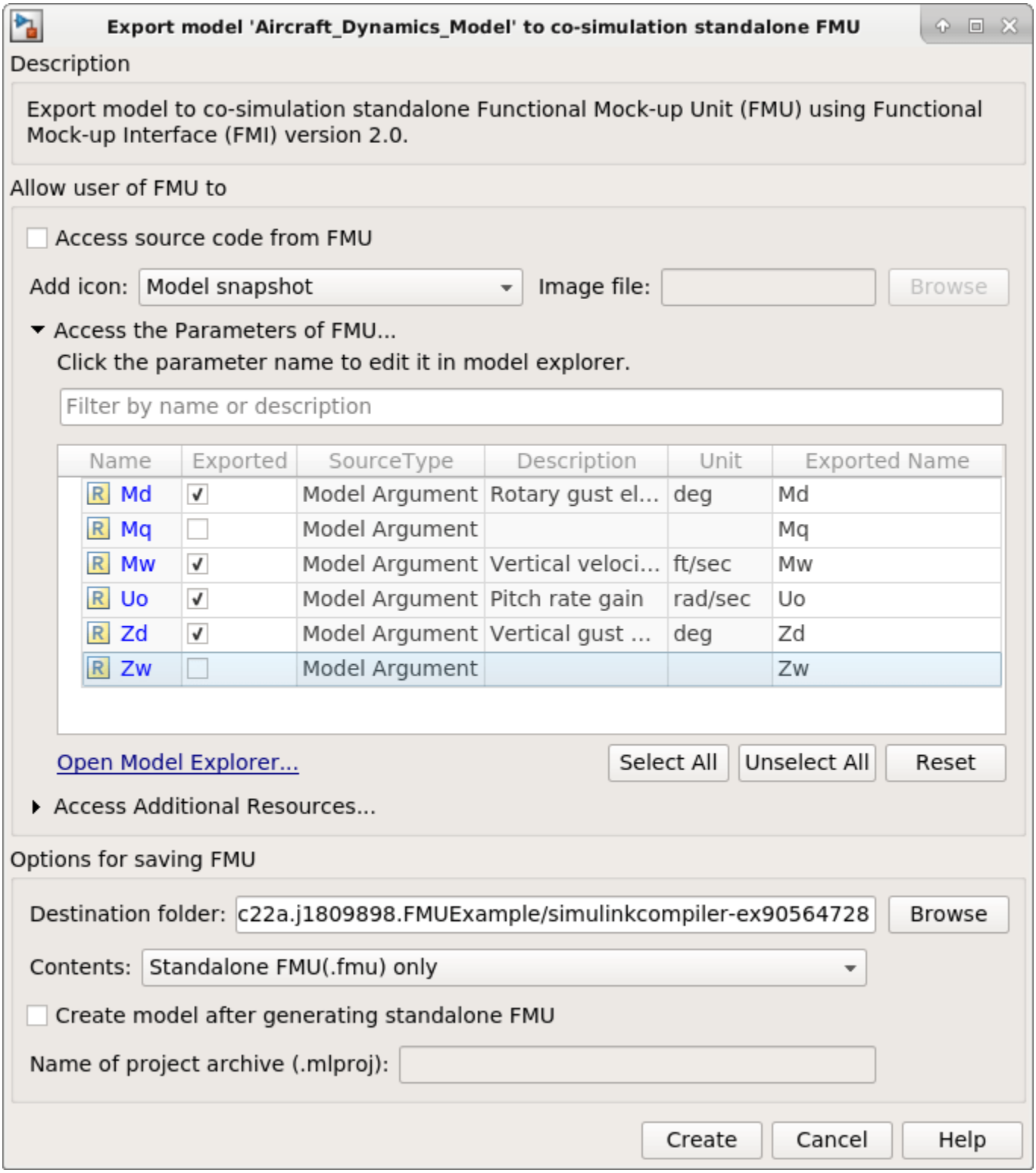
This section exports a reference model `Aircraft_dynamics_Model` as a standalone FMU that will be used in nested FMU export workflow.




```
% open model Aircraft_dynamics_Model
model = "Aircraft_Dynamics_Model";
open_system(model);
```

The figure below shows the Export Standalone FMU dialog. User can manually select model variables exposed to FMU interface by using **Parameter** Table under **Access the Parameters of FMU...** in FMU export dialog. In this example, we expose variables *Md*, *Mw*, *Uo*, and *Zd*. Other features like pack source code, configure model variables, add additional resources to FMU package are available via UI. Read more about the Standalone FMU export functionality: Export Simulink Model to Standalone FMU.

Note: You may see a warning message indicating variable names are not unique, too long, or contain invalid characters and they will be renamed. This is expected if you have variable names meet above conditions, and you will see the renamed variables when you import the FMU back to a FMU block.



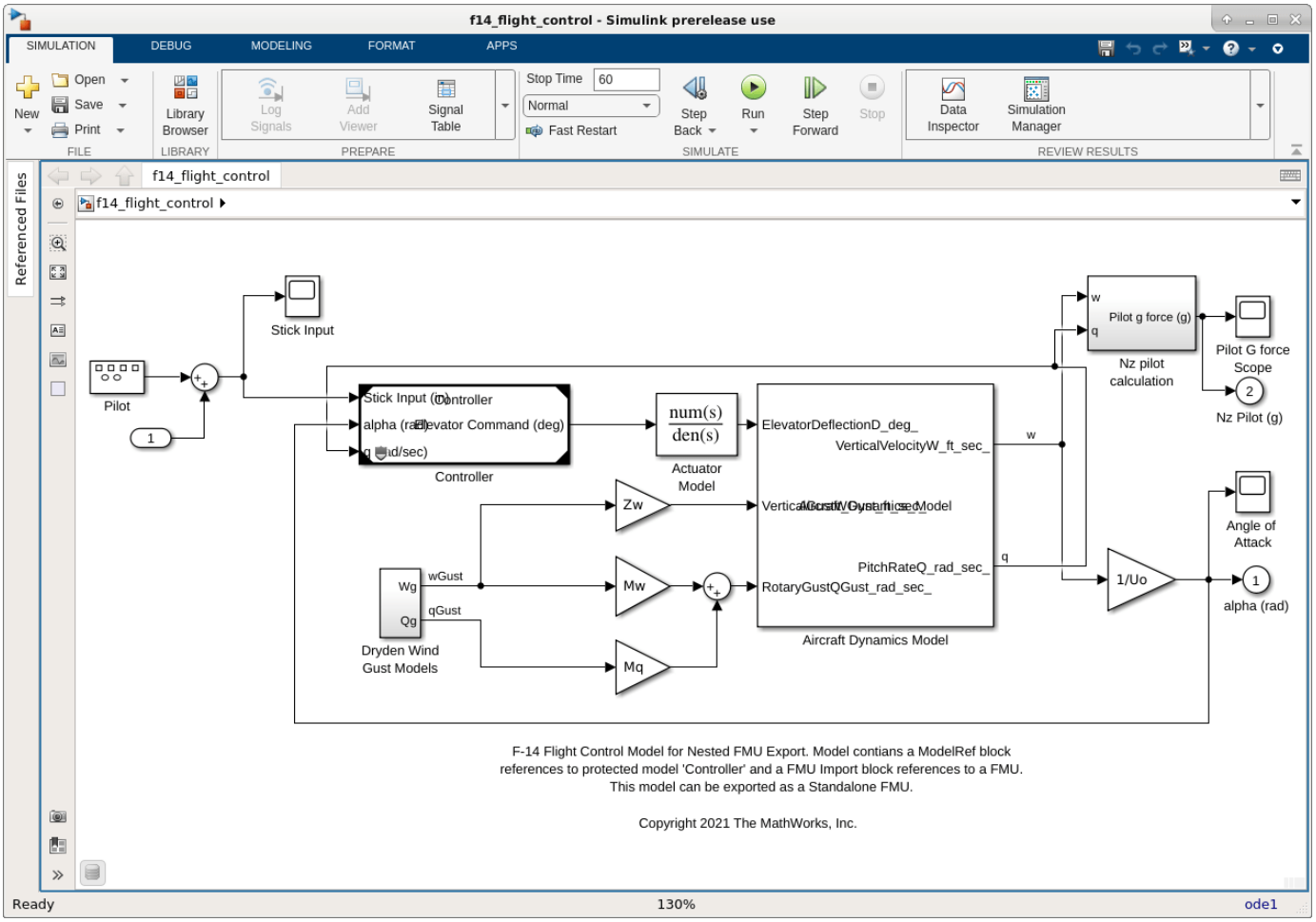

```
% export model Aircraft dynamics to standalone FMU
exportToFMU2CS(model,'CreateModelAfterGeneratingFMU', 'off');
```

```
Setting System Target to FMU Co-Simulation for model 'Aircraft_Dynamics_Model'.
Setting Hardware Implementation > Device Type to 'MATLAB Host' for model 'Aircraft_Dynamics_Mode
### 'GenerateComments' is disabled for Co-Simulation Standalone FMU Export.

Warning: The following variable names are not unique, too long, or contain invalid characters: 'E
Deflection
d (deg),Vertical Gust
wGust (ft/sec),Rotary Gust
qGust (rad/sec),Vertical Velocity
w (ft/sec),Pitch Rate
q (rad/sec)'. They will be renamed to: 'ElevatorDeflectionD_deg_,VerticalGustWGust_ft_sec_,Rotary

Build Summary

Top model targets built:

Model                     Action                         Rebuild Reason
=============================================================================================
Aircraft_Dynamics_Model  Code generated and compiled.  Code generation information file does not

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 30.99s
### Model was successfully exported to co-simulation standalone FMU: 'C:\TEMP\Bdoc22b_2054784_600
```

```
close_system(model);
```

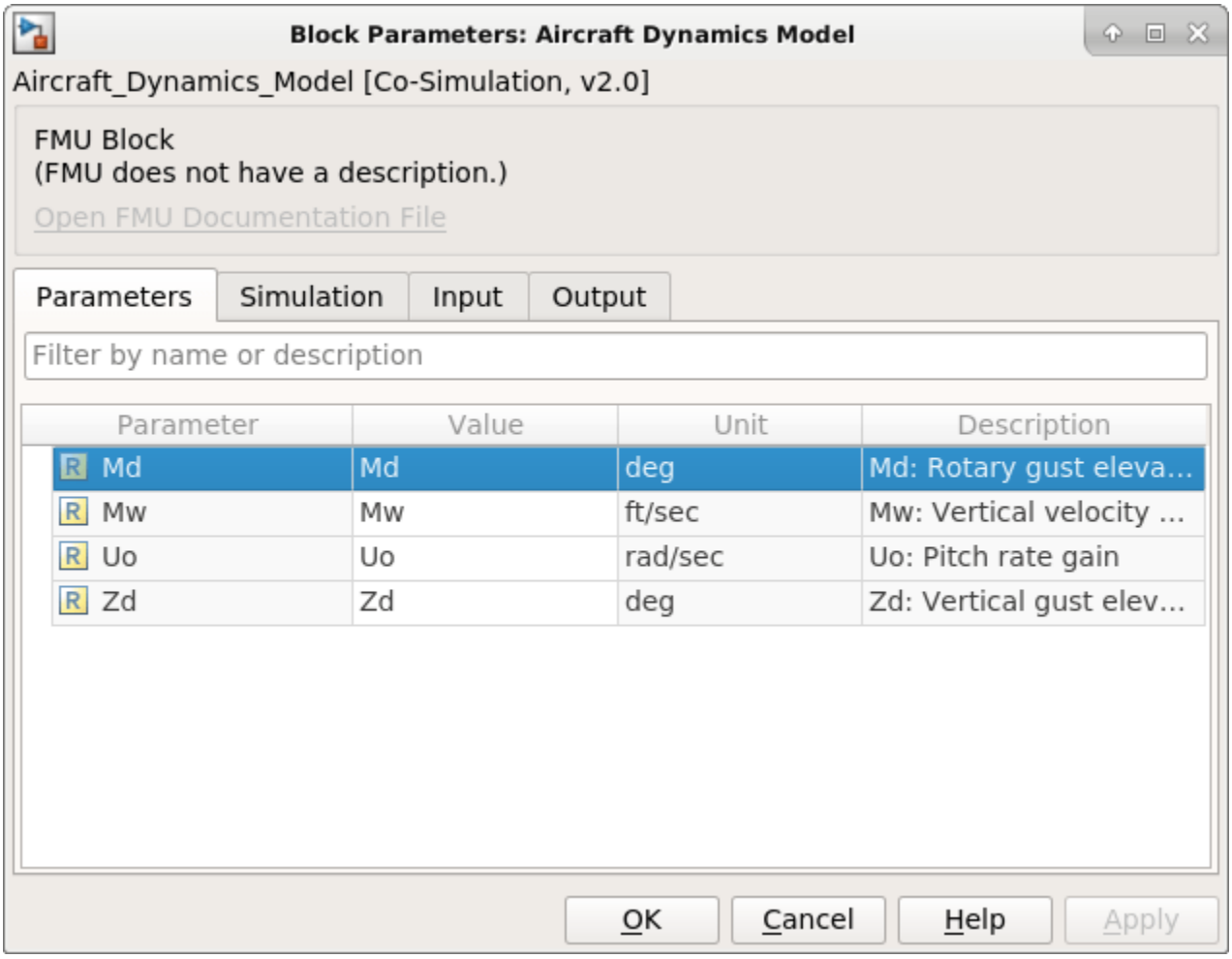
Export Simulink Model with External References to Standalone FMU

This section illustrates how to export Simulink model `f14_flight_control` with external references to a Standalone FMU with tunable parameters. Model `f14_flight_control` uses the protected model and FMU generated in the previous step, Model block `Controller` references the protected model `Controller.slxp`, and FMU Import block `Aircraft Dynamics Model` references `Aircraft_Dynamics_Model.fmu`.




```
% open top model that reference the protedted model and FMU
model = "f14_flight_control";
open_system(model);
```

Open FMU Import Block dialog and configure tunable parameter to be exported in the nested FMU. Parameters `Md, Mw, Uo, Zd` are `Simulink.Parameter` objects defined in model workspace.

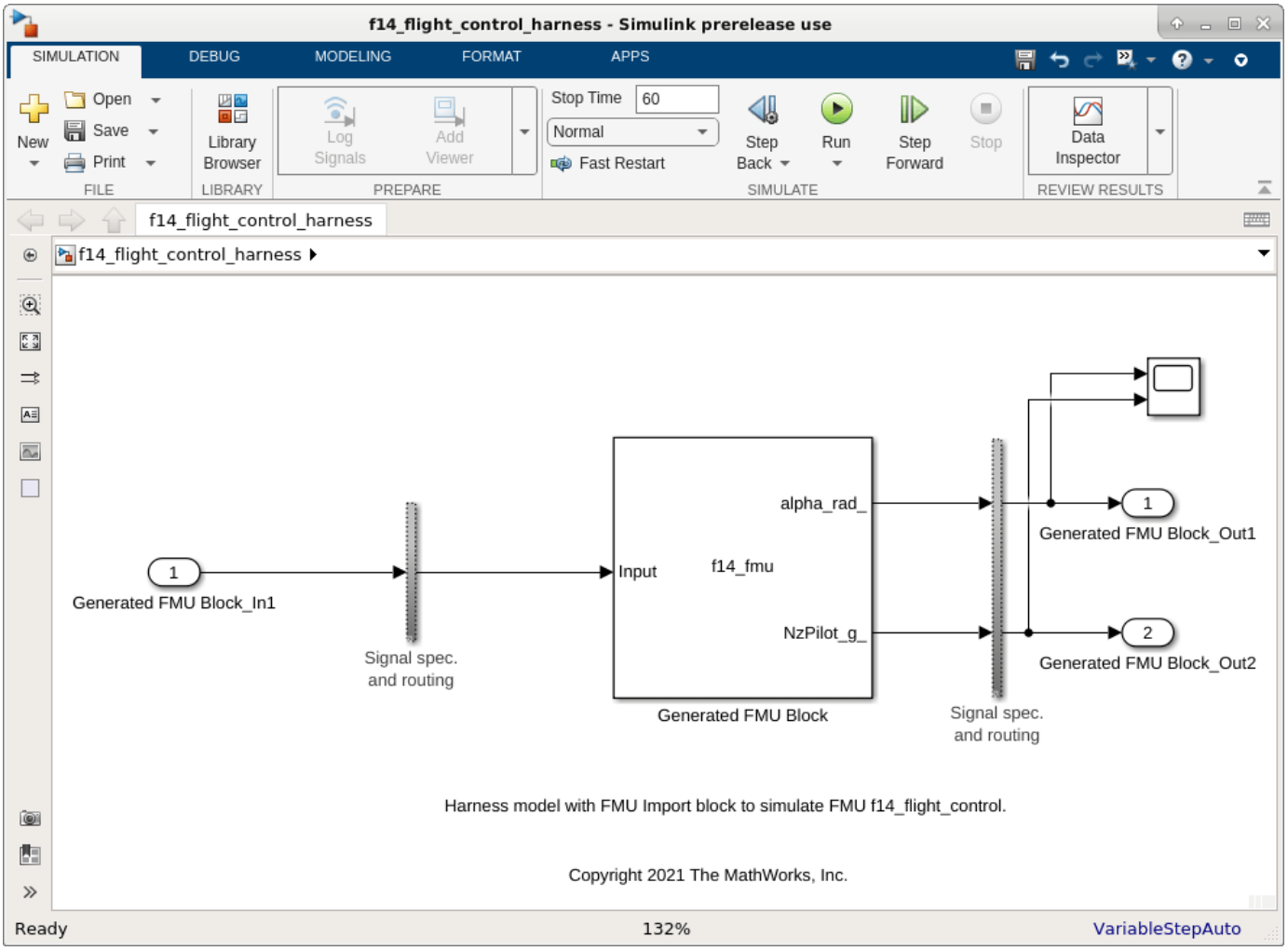


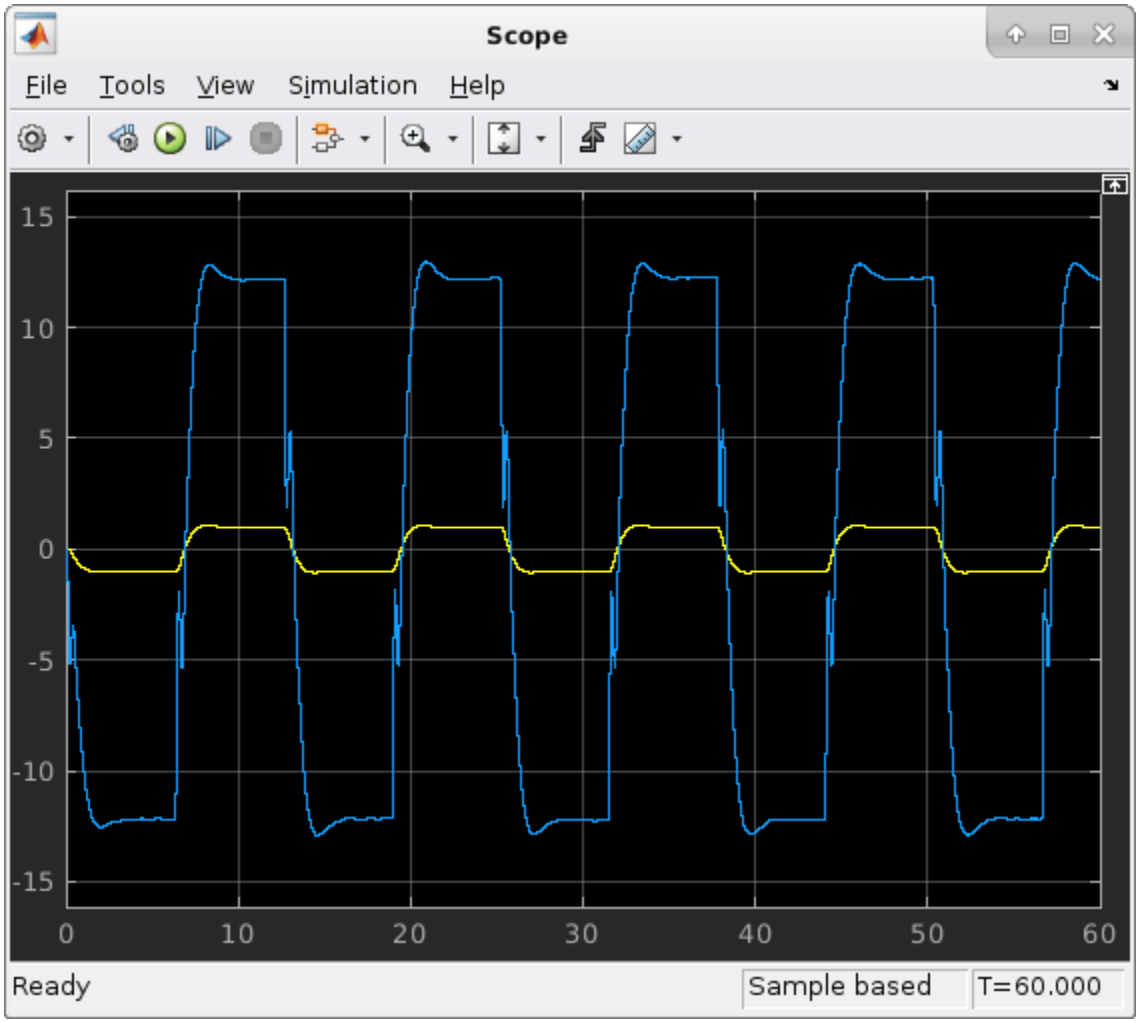

```
% Associate parameter in FMU export dialog with model variables
% This allows the variables to be export as tunable parameter in the generated nested FMU
set('f14_flight_control/Aircraft Dynamics Model','Md', 'Md');
set('f14_flight_control/Aircraft Dynamics Model','Mw', 'Mw');
set('f14_flight_control/Aircraft Dynamics Model','Uo', 'Uo');
set('f14_flight_control/Aircraft Dynamics Model','Zd', 'Zd');
```

To build and export model `f14_flight_control` to a standalone FMU, click drop-down button for **Save** from **Simulation** tab and select **Standalone FMU**. Follow the export procedure as described in previous section.

```
% Export model to Standalone Co-Simulation FMU 2.0
exportToFMU2CS(model,'CreateModelAfterGeneratingFMU', 'on');
```

A standalone FMU is generated in the **Destination** folder specified from the export dialog. A harness model is created and opened if the user selects **Create model after generating standalone FMU**.


f14_flight_control_harness - Simulink prerelease use
SIMULATION
DEBUG
MODELING
FORMAT
APPS
New
Open
Save
Print
FILE
Library Browser
LIBRARY
Log Signals
Add Viewer
PREPARE
Stop Time
60
Normal
Fast Restart
Step Back
Run
Step Forward
Stop
SIMULATE
Data Inspector
REVIEW RESULTS
f14_flight_control_harness
f14_flight_control_harness
1
Generated FMU Block_In1
Signal spec. and routing
Input
f14_fmu
alpha_rad_
NzPilot_g_
Generated FMU Block
Signal spec. and routing
1
Generated FMU Block_Out1
2
Generated FMU Block_Out2
Harness model with FMU Import block to simulate FMU f14_flight_control.
Copyright 2021 The MathWorks, Inc.
Ready
132%
VariableStepAuto




```
close_system(model);
```

Export Simulink Model to Standalone FMU with User Specified Files and Archived Project with Harness Model

This example shows how to export Simulink® component to standalone Co-Simulation FMU 2.0 with user specified documentation and harness model using Simulink Compiler™. This example demonstrate following features of Standalone FMU Export:

- User file packing.
- Archived project generation with harness model.

You may want too add files in the generated Standalone FMU for the following scenarios:

- FMU requires documentation and/or license information
- FMU needs additional files to determine its initial states
- FMU requires additional meta-data information for compliance.

Similarly, you may want to generated archived project with harness model for:

- Cross team collaboration for System Design in Simulink
- SIL and functional testing using Simulink Test™.

This example exports `fmuexport_fuelsys_controller` to standalone FMU with specified documentation. The documentation is in form of `index.html` with supporting image files inside `images` folders. This documentation outlines the behavior of component modeled using standalone FMU. The archived project generated can be shared for reuse in Simulink. For a detailed explanation of the model, see:

- “Modeling a Fault-Tolerant Fuel Control System”

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

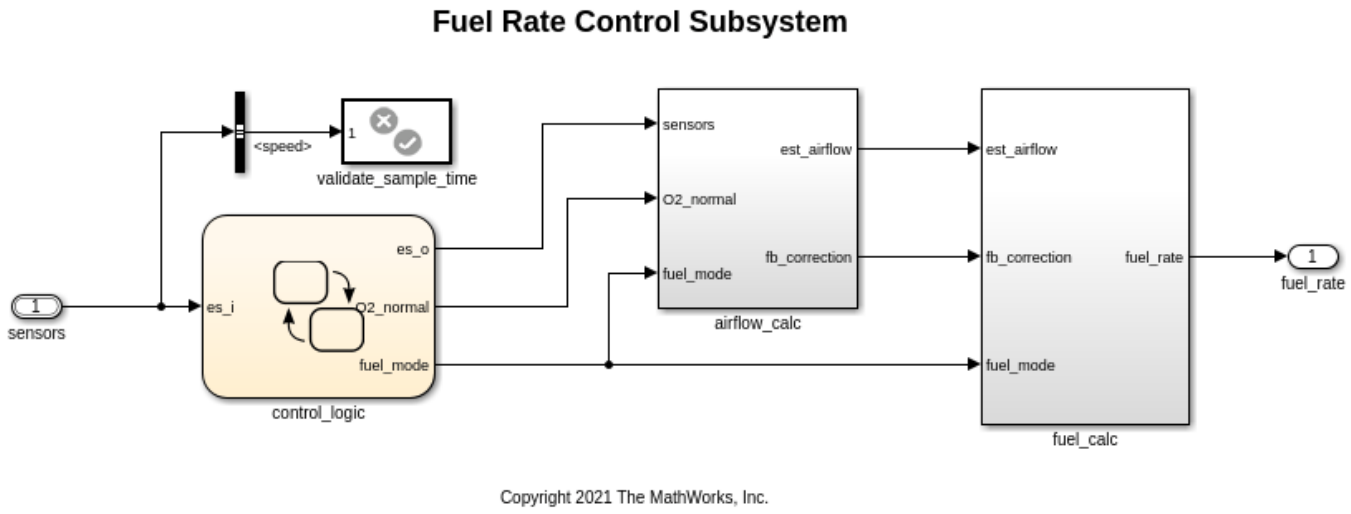
- Fuel Rate Control Component: `fmuexport_fuelsys_controller`,
- Engine Gas Dynamics Component: `fmuexport_fuelsys_plant`, and
- top-level model `fmuexport_fuelsys_top`.

For a list of tools that support FMI, see: https://fmi-standard.org/tools/.

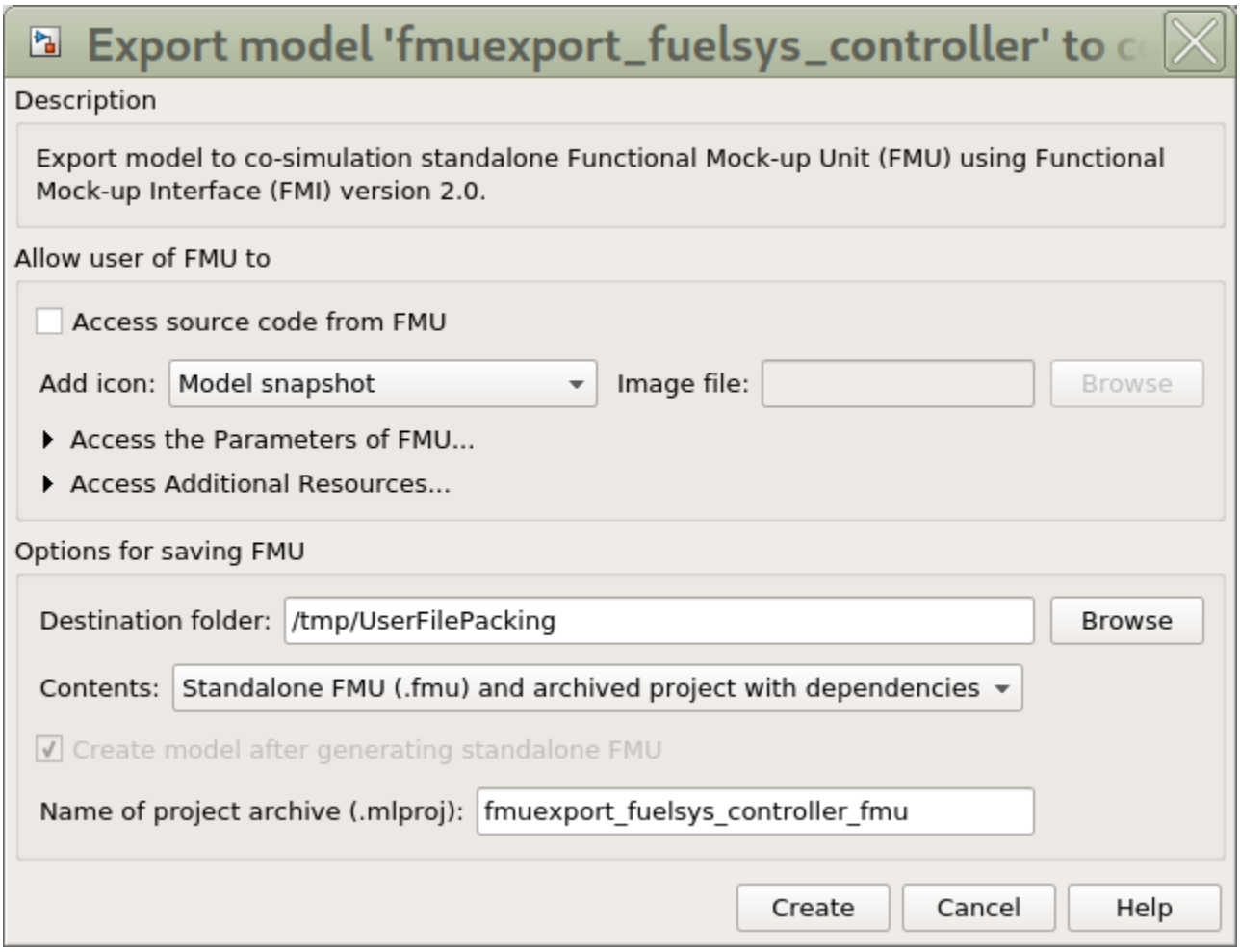
Export Fuel Rate Control Component to FMU with User Specified Documentation

Open the `fmuexport_fuelsys_controller` example model.

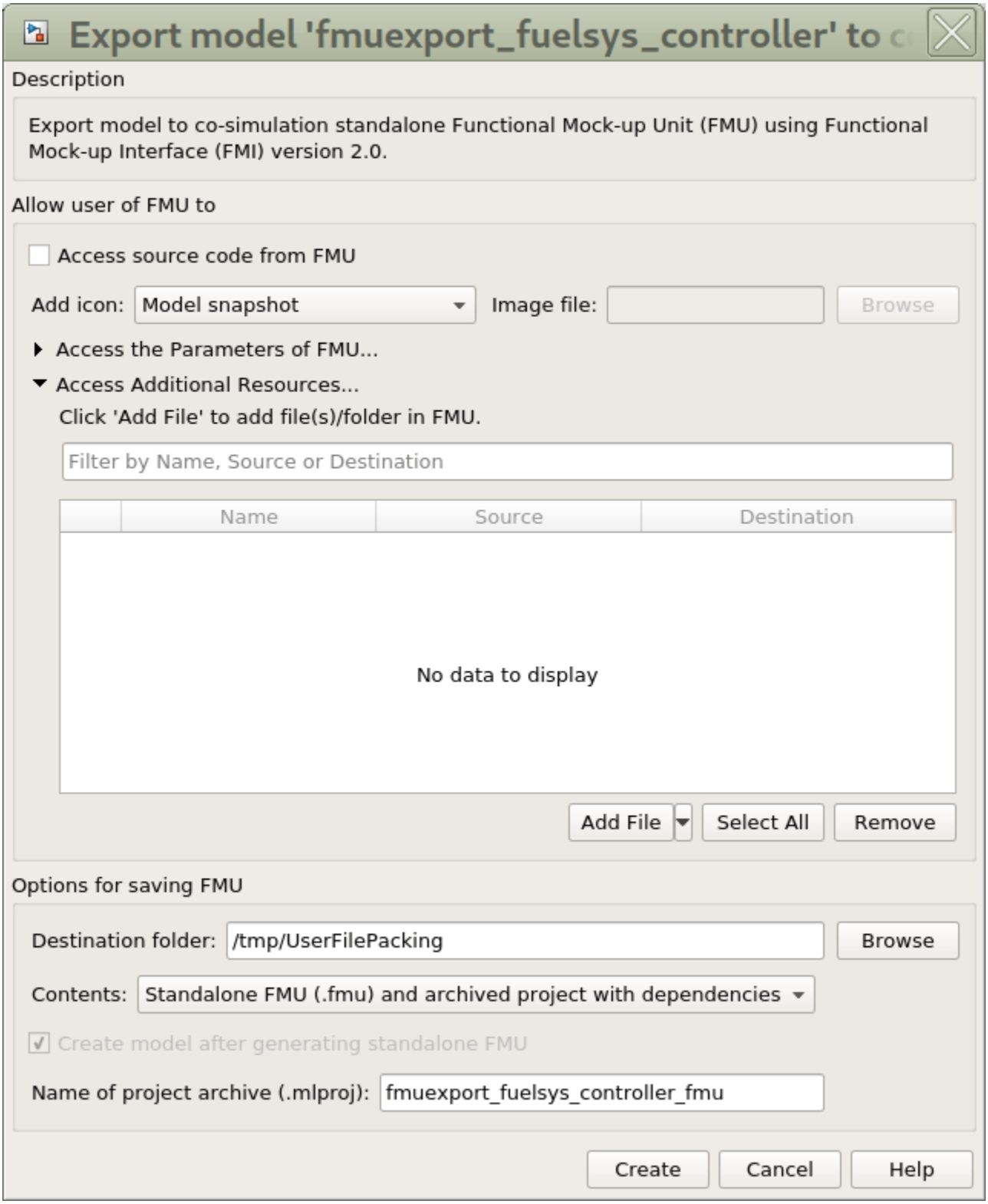
```
open_system('fmuexport_fuelsys_controller');
```

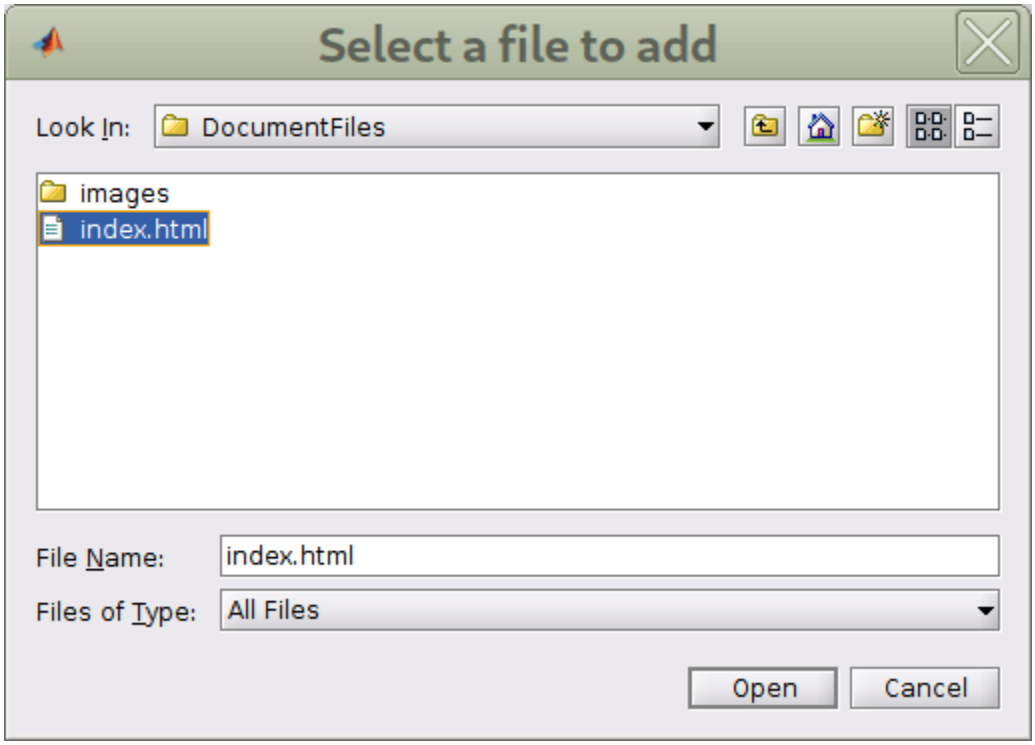


From **Simulation** tab, click drop-down button for **Save**. In **Export Model To** section, click **Standalone FMU...**. In FMU Export dialog, click **Access Additional Resource** to expand the section for adding user specified files

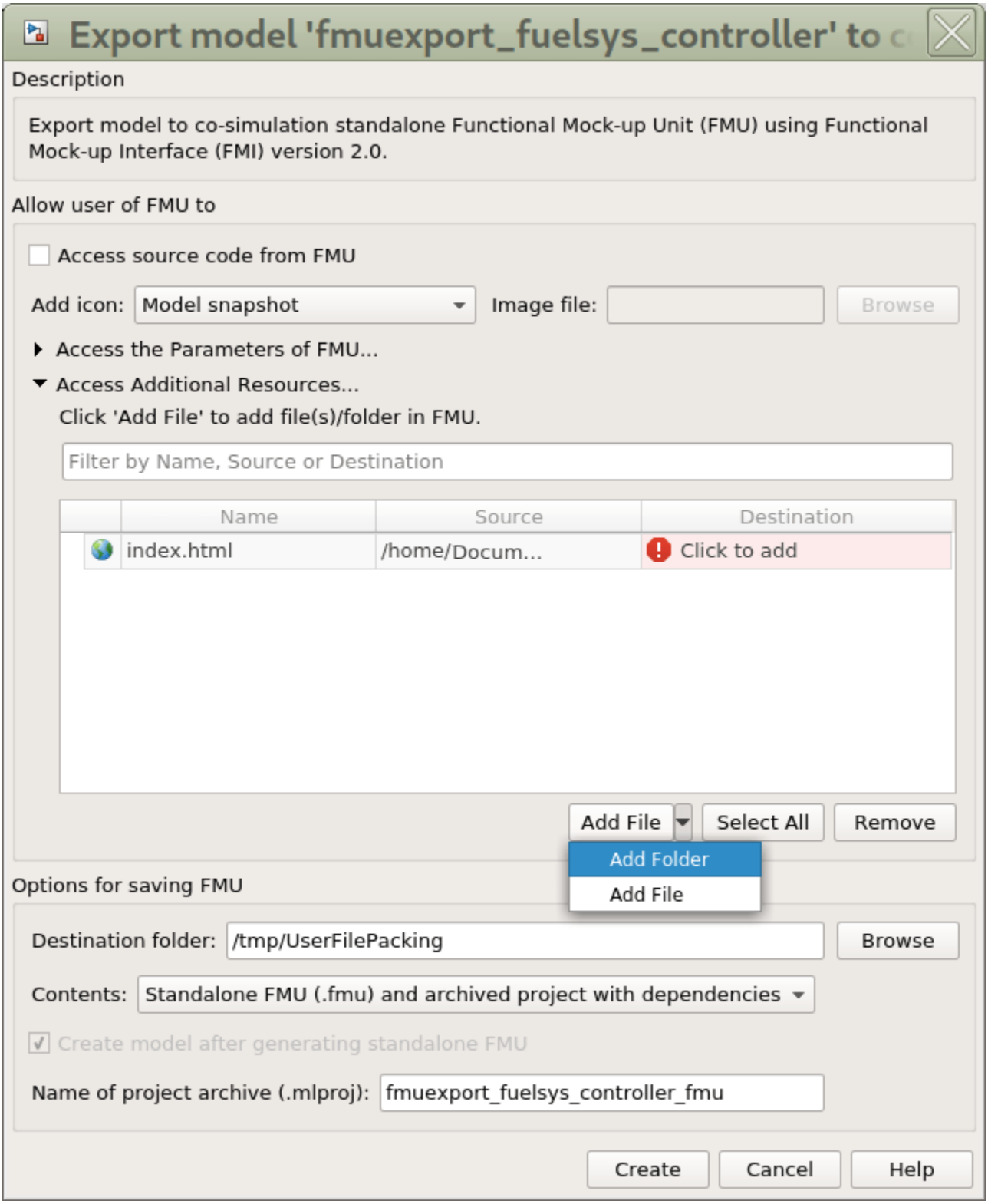


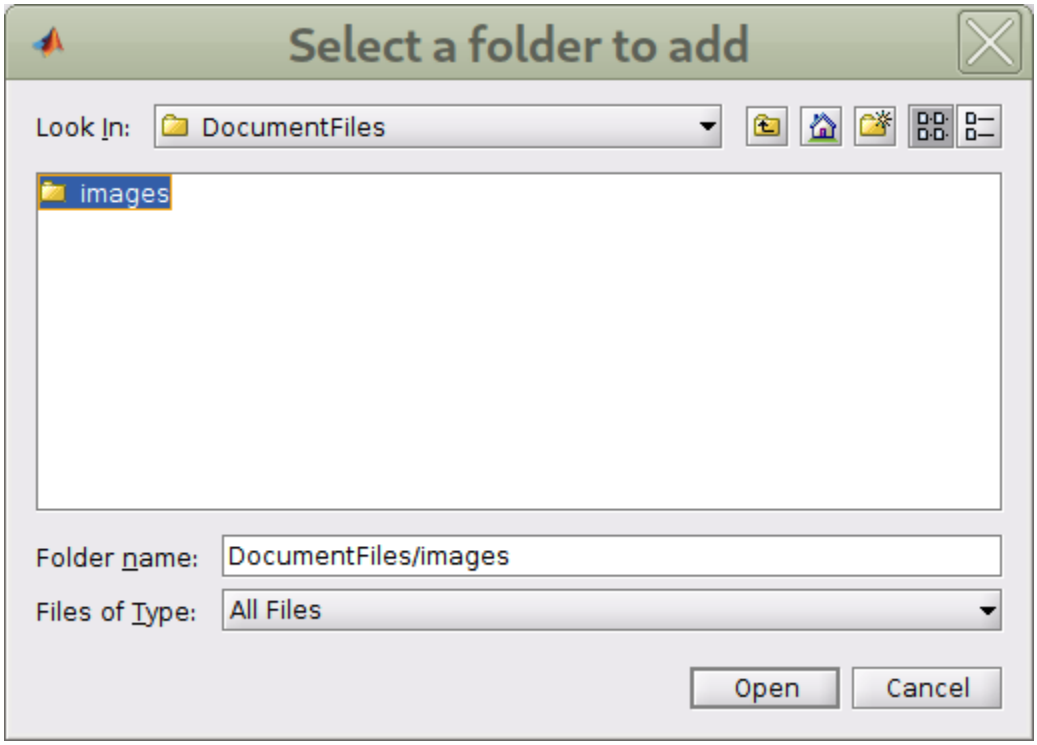
Click **Add File** to add `index.html` in `DocumentFiles` folder


Export model 'fmuexport_fuelsys_controller' to c
Description
Export model to co-simulation standalone Functional Mock-up Unit (FMU) using Functional Mock-up Interface (FMI) version 2.0.
Allow user of FMU to
Access source code from FMU
Add icon:
Model snapshot
Image file:
Browse
Access the Parameters of FMU...
Access Additional Resources...
Click 'Add File' to add file(s)/folder in FMU.
Filter by Name, Source or Destination
Name
Source
Destination
No data to display
Add File
Select All
Remove
Options for saving FMU
Destination folder:
/tmp/UserFilePacking
Browse
Contents:
Standalone FMU (.fmu) and archived project with dependencies
Create model after generating standalone FMU
Name of project archive (.mlproj):
fmuexport_fuelsys_controller_fmu
Create
Cancel
Help

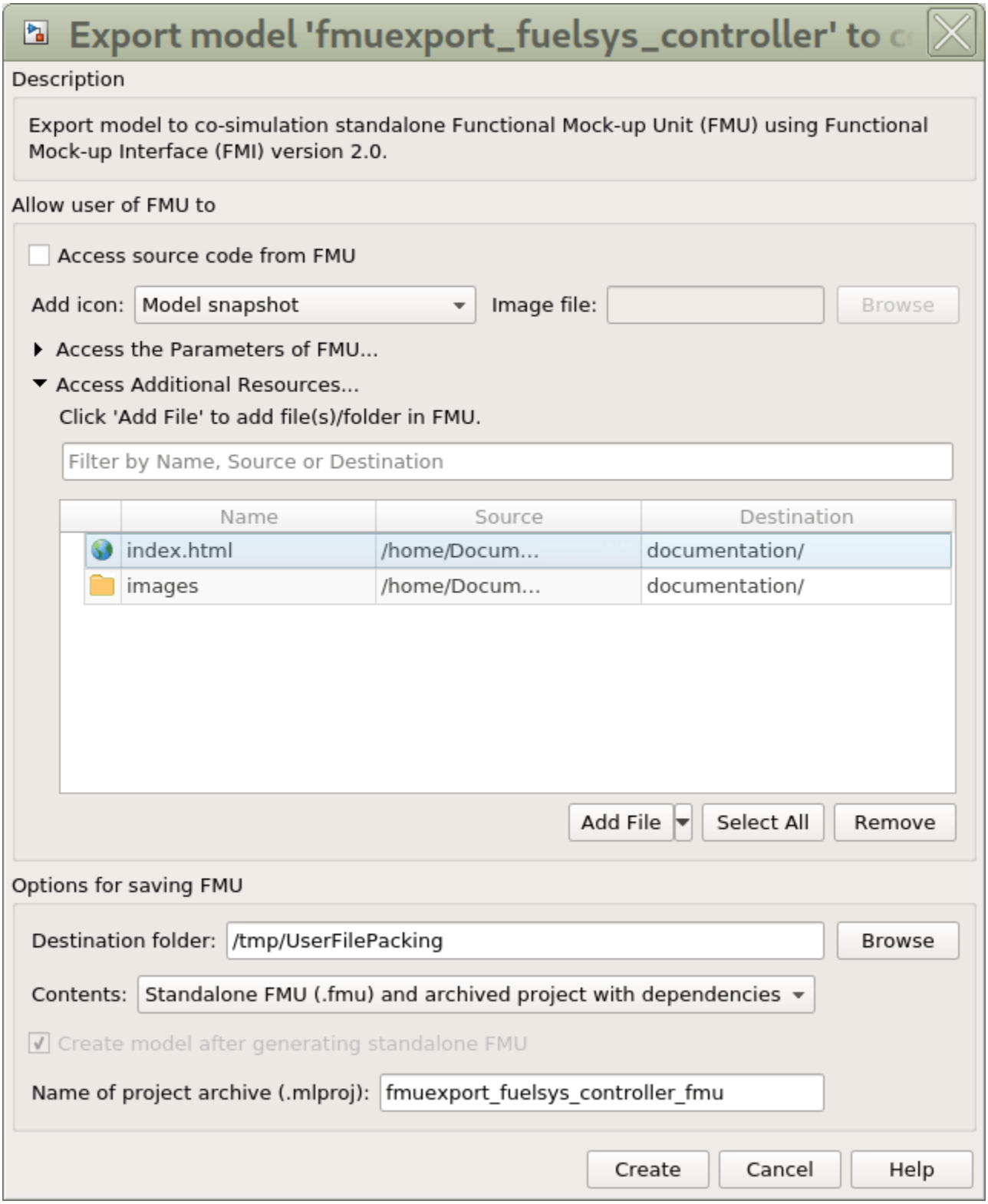


Click the dropdown associated with **Add File** and select **Add Folder** to add `images` folder inside `DocumentFiles`


Export model 'fmuexport_fuelsys_controller' to c
Description
Export model to co-simulation standalone Functional Mock-up Unit (FMU) using Functional Mock-up Interface (FMI) version 2.0.
Allow user of FMU to
Access source code from FMU
Add icon:
Model snapshot
Image file:
Browse
Access the Parameters of FMU...
Access Additional Resources...
Click 'Add File' to add file(s)/folder in FMU.
Filter by Name, Source or Destination
Name
Source
Destination
index.html
/home/Docum...
Click to add
Add File
Select All
Remove
Add Folder
Add File
Options for saving FMU
Destination folder:
/tmp/UserFilePacking
Browse
Contents:
Standalone FMU (.fmu) and archived project with dependencies
Create model after generating standalone FMU
Name of project archive (.mlproj):
fmuexport_fuelsys_controller_fmu
Create
Cancel
Help

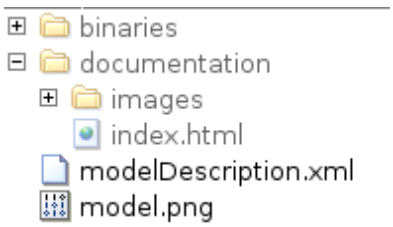


Click **Select All** button to select the entries in the spreadsheet. Click one of the rows under **Destination** column and update the destination to `documentation`



Click **Create** button to export standalone FMU for Fuel Rate Control Component

After FMU is generated, unzip the generated FMU and notice the presence of `index.html` and `images` folder under `documentation`




```
restoreval = get_param(0,'AutoSaveOptions');
newval = restoreval;
% Disable auto save
newval.SaveOnModelUpdate = false;
set_param(0,'AutoSaveOptions',newval);
restoreOC = onCleanup(@()set_param(0,'AutoSaveOptions',restoreval));
% Export Simulink model to Standalone Co-Simulation FMU 2.0
evalc('exportToFMU2CS(''fmuexport_fuelsys_controller'', ''Package'', {''documentation'',{fullfile
restoreOC.delete;
close_system('fmuexport_fuelsys_controller', 0);
```

You can also export FMU and archived project using command-line. At the MATLAB® command line, use `exportToFMU2CS` command.

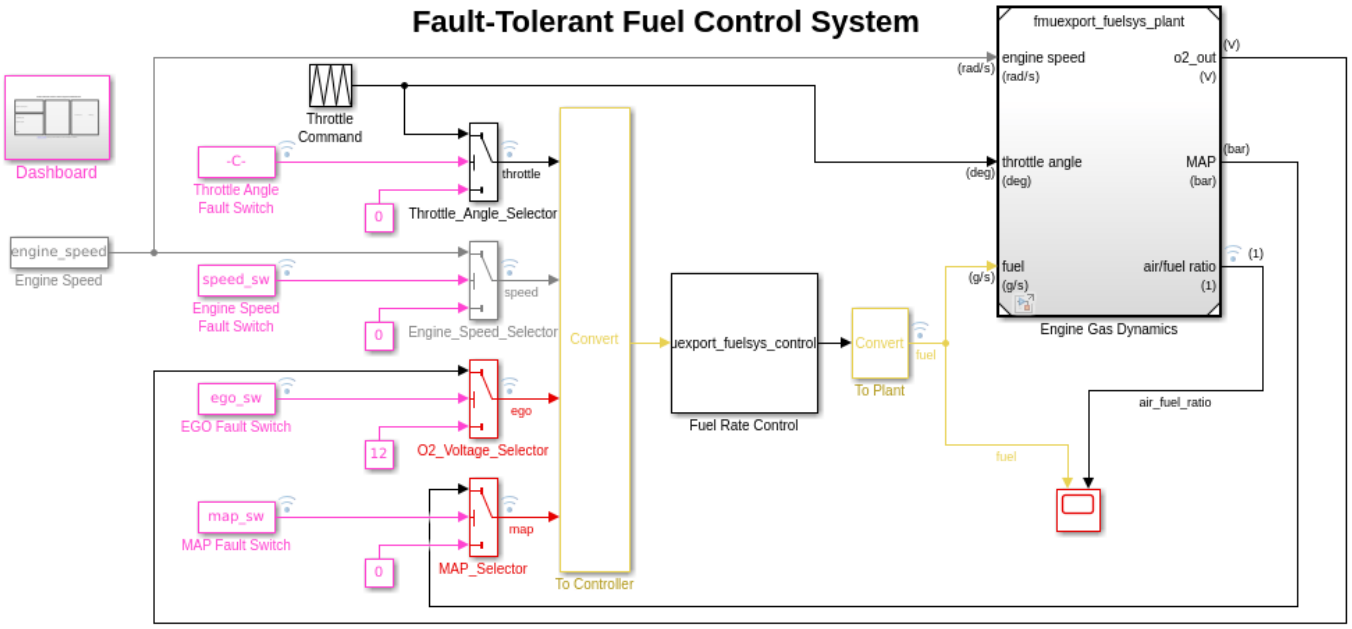
```
% Export model to Standalone Co-Simulation FMU 2.0 with user specified
% file
exportToFMU2CS('fmuexport_fuelsys_controller', 'Package', {'documentation',{fullfile(pwd,'Docume
```

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

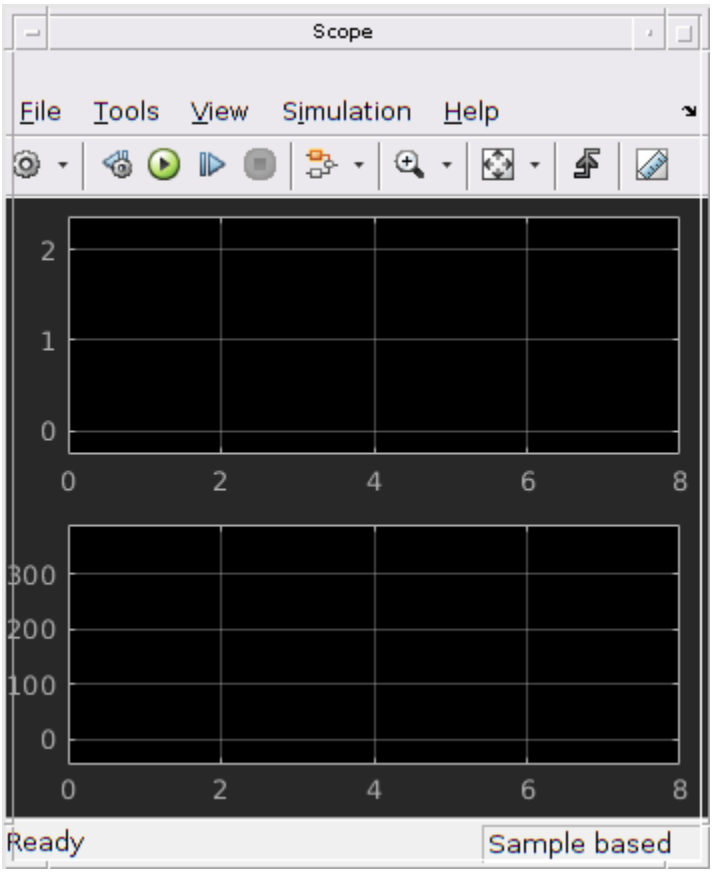
Integrate Standalone FMU with Simulink Model

Once the FMU is successfully exported, you may use the top model `fmuexport_fuelsys_top` to fully integrate the system for testing.

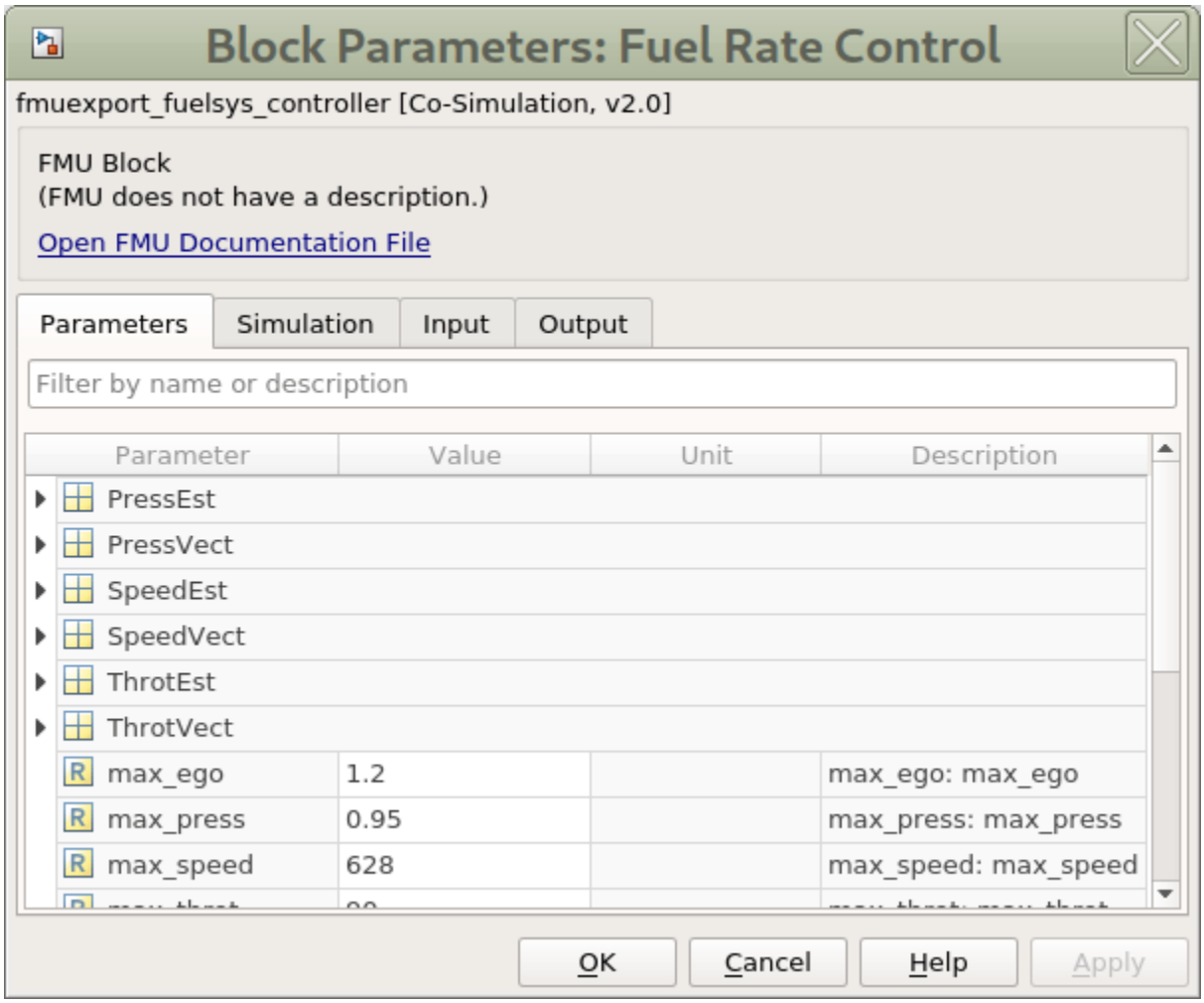
```
clear Atm PumpCon RampRateKiZ SpeedVect max_press min_press p0 st_range ...
    PressEst RampRateKiX SonicFlow ThrotEst hys max_speed min_speed ...
    restoreOC zero_thresh PressVect RampRateKiY SpeedEst ThrotVect ...
    max_ego max_throt min_throt s;
open_system('fmuexport_fuelsys_top');
```



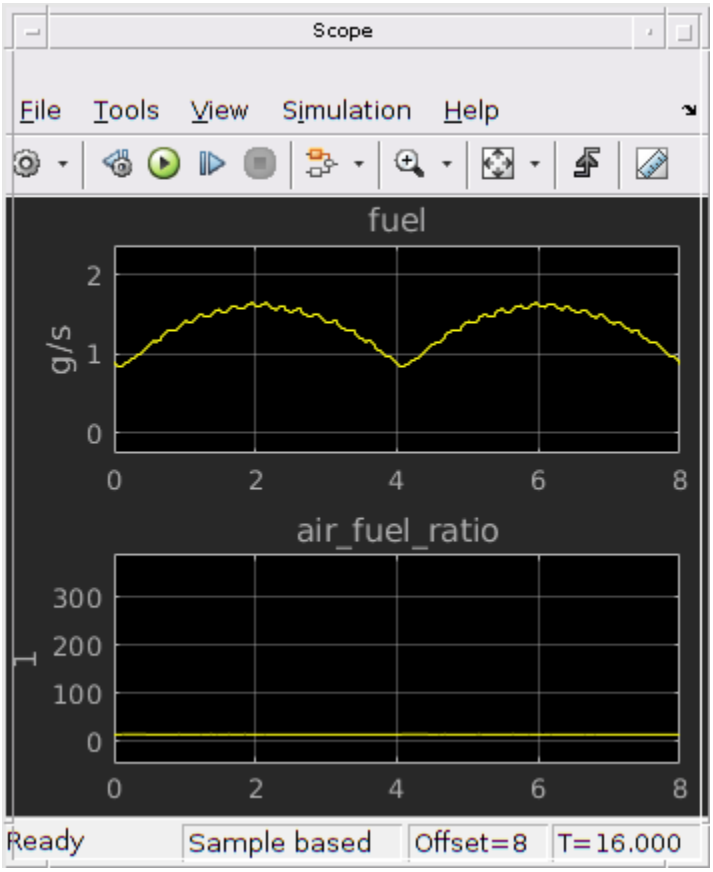
Open the Dashboard subsystem to simulate any combination of sensor failures.



In the `fmuexport_fuelsys_top`, you can open the documentation file packed in the FMU by launching the block dialog for `Fuel Rate Control` and clicking **Open FMU Documentation File**.




```
set_param('fmuexport_fuelsys_top', 'SimulationCommand', 'Update');
set_param('fmuexport_fuelsys_top', 'StopTime', '16');
sim('fmuexport_fuelsys_top');
```




```
bdclose('fmuexport_fuelsys_top');
clear ans sldemo_fuelsys_output;
```

The generated simulation result might be slightly different than “Modeling a Fault-Tolerant Fuel Control System”. This is expected, for more information see, “Co-Simulation Execution”

Integrate Harness Model in Archived Project with Simulink Model

You may also integrate the harness model inside the archived project in `fmuexport_fuelsys_top_harness_model` by unzipping **fmuexport_fuelsys_controller_fmu.mlproj** archived project file in the current directory and executing the following command in the MATLAB command window.

```
%% Script to simulate the extracted harness model in Simulink
open_system('fmuexport_fuelsys_top_harness_model');
set_param('fmuexport_fuelsys_top_harness_model', 'SimulationCommand', 'Update');
set_param('fmuexport_fuelsys_top_harness_model', 'StopTime', '16');
sim('fmuexport_fuelsys_top_harness_model');
```

You can also use the harness model for functional verification in Simulink. For more information, see “Create Test Harnesses from Standalone Models” (Simulink Test).

Export Simulink Model to Standalone FMU with Source Code

This example shows how to export Simulink® component to standalone Co-Simulation FMU 2.0 with source code using Simulink Compiler™ and Simulink Coder™. The source code is packed inside the **sources** folder of the Standalone FMU. A documentation file, **index.html** is also generated. It lists the steps to regenerate the binaries on another platform. This documentation file is located inside **documentation** folder of the Standalone FMU. The source code can be used for cross platform workflow.

A user may desire to export Simulink component to standalone FMU with source code for the following scenarios:

- Cross-platform FMU support.
- Hardware-in-Loop (HIL) simulation.

In this example, the aircraft flight control system is composed of two simulink models:

- Aircraft Longitudinal Flight Control Plant Component: `fmuexport_aircraft_flight_control_plant` and
- Top-level model: `fmuexport_aircraft_flight_control_top`.

`fmuexport_aircraft_flight_control_plant` models flight control for the longitudinal motion of an aircraft. First order linear approximations of the aircraft and actuator behavior are connected to an analog flight control design that uses the pilot's stick pitch command as the set point for the aircraft's pitch attitude and uses aircraft pitch angle along with pitch rate to determine commands. A simplified Dryden wind gust model is incorporated to perturb the system.

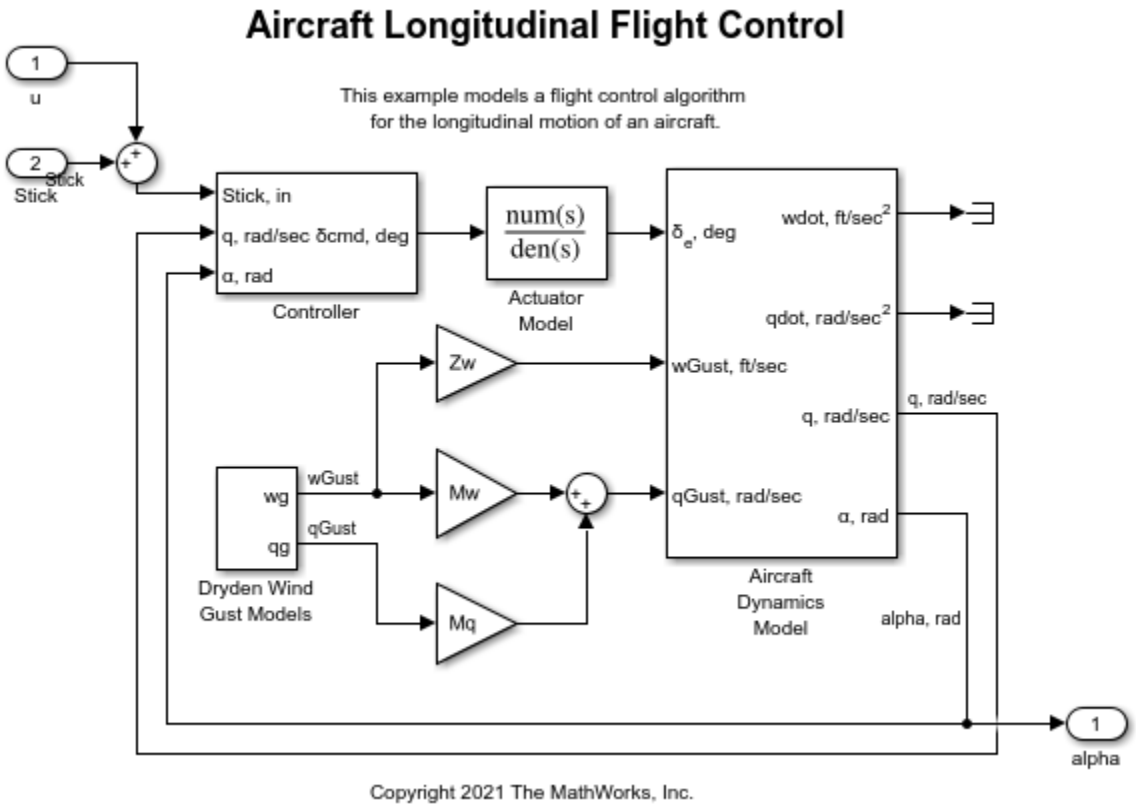
`fmuexport_aircraft_flight_control_top` is used to observe the change in the angle of attack reported by Aircraft Longitudinal Flight Control Plant Component with respect to the applied pilot's stick pitch command.

This examples exports `fmuexport_aircraft_flight_control_plant` to Standalone Co-Simulation FMU 2.0 with source code and list steps to compile the source code on 64-bit Linux® platform to generate binaries for reuse in `fmuexport_aircraft_flight_control_top`. For a list of tools that support FMI, see: https://fmi-standard.org/tools/.

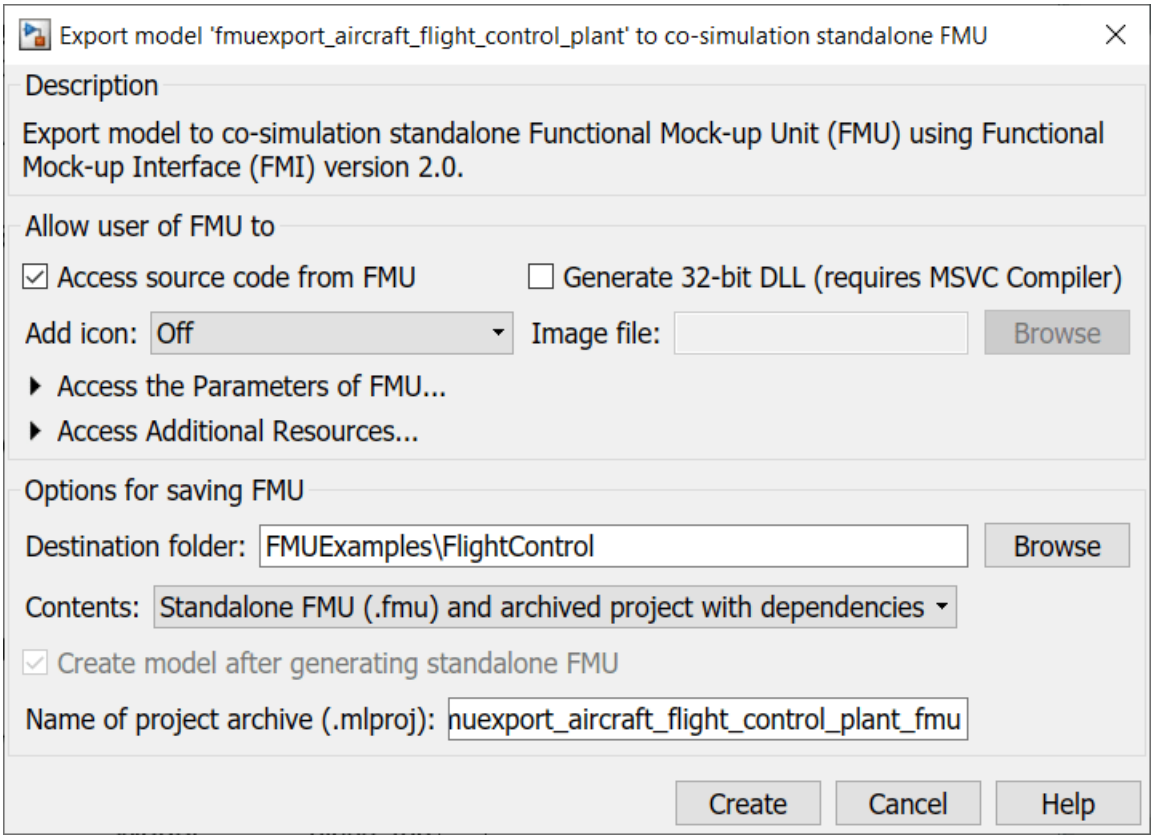
Export Aircraft Longitudinal Flight Control Plant Component to FMU with Source Code

Open the `fmuexport_aircraft_flight_control_plant` example model.

```
open_system('fmuexport_aircraft_flight_control_plant');
```



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



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

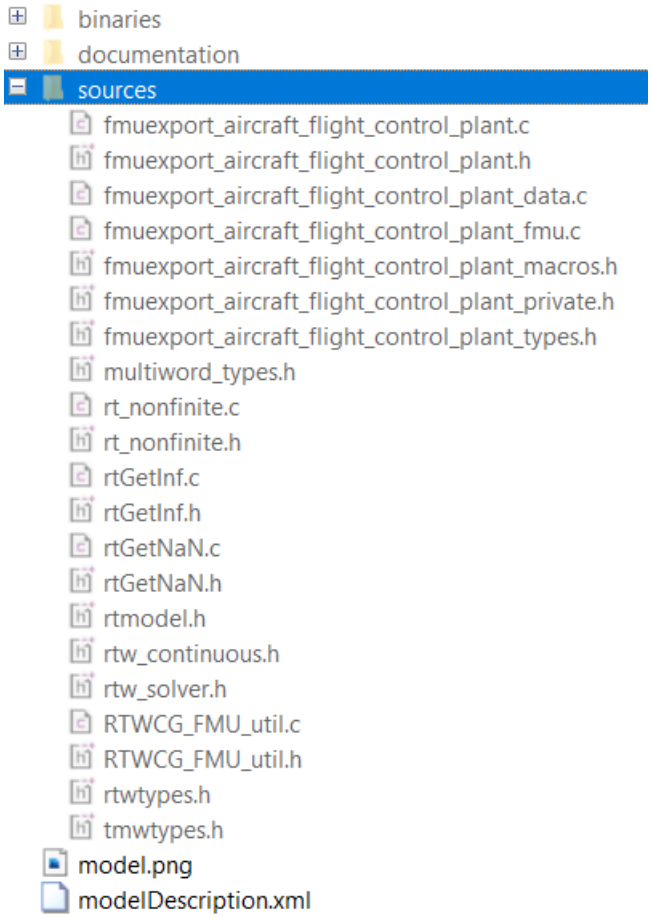
```
restoreval = get_param(0,'AutoSaveOptions');
newval = restoreval;
newval.SaveOnModelUpdate = false;
set_param(0,'AutoSaveOptions',newval);
restoreOC = onCleanup(@()set_param(0,'AutoSaveOptions',restoreval));
% Export model to Standalone Co-Simulation FMU 2.0 with source code
evalc('exportToFMU2CS(''fmuexport_aircraft_flight_control_plant'',''SaveDirectory'', pwd, ''Save
restoreOC.delete;
close_system('fmuexport_aircraft_flight_control_plant', 0);
```

FMU can also be exported with source code using command-line. In the MATLAB® command-line window, use `exportToFMU2CS` command:

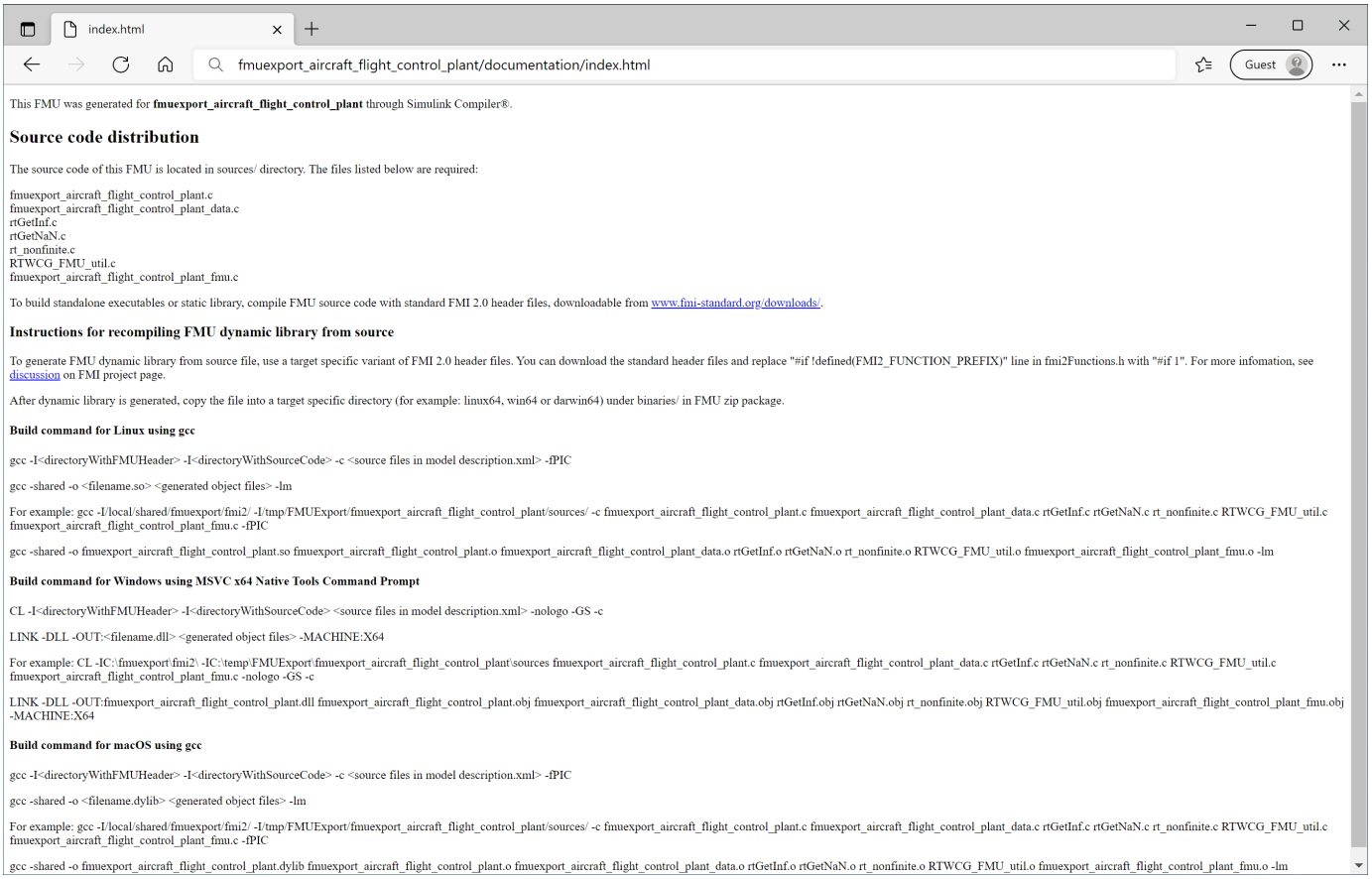
```
% Export model to Standalone Co-Simulation FMU 2.0
exportToFMU2CS('fmuexport_aircraft_flight_control_plant', 'SaveDirectory', pwd, 'SaveSourceCodeT
```

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

The generated standalone Co-Simulation FMU 2.0 has source code in the **sources** folder inside the FMU



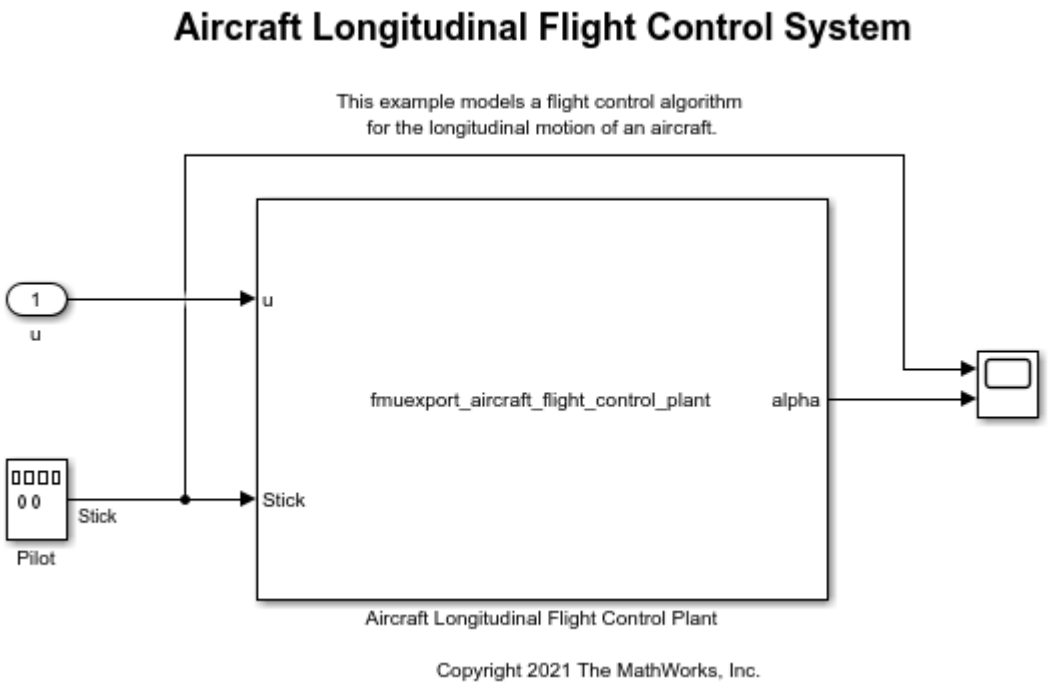
A documentation file(index.html) is also generated with steps to regenerate the binaries on Linux, Windows® and macOS platform. This documentation file is located at documentation/index.html

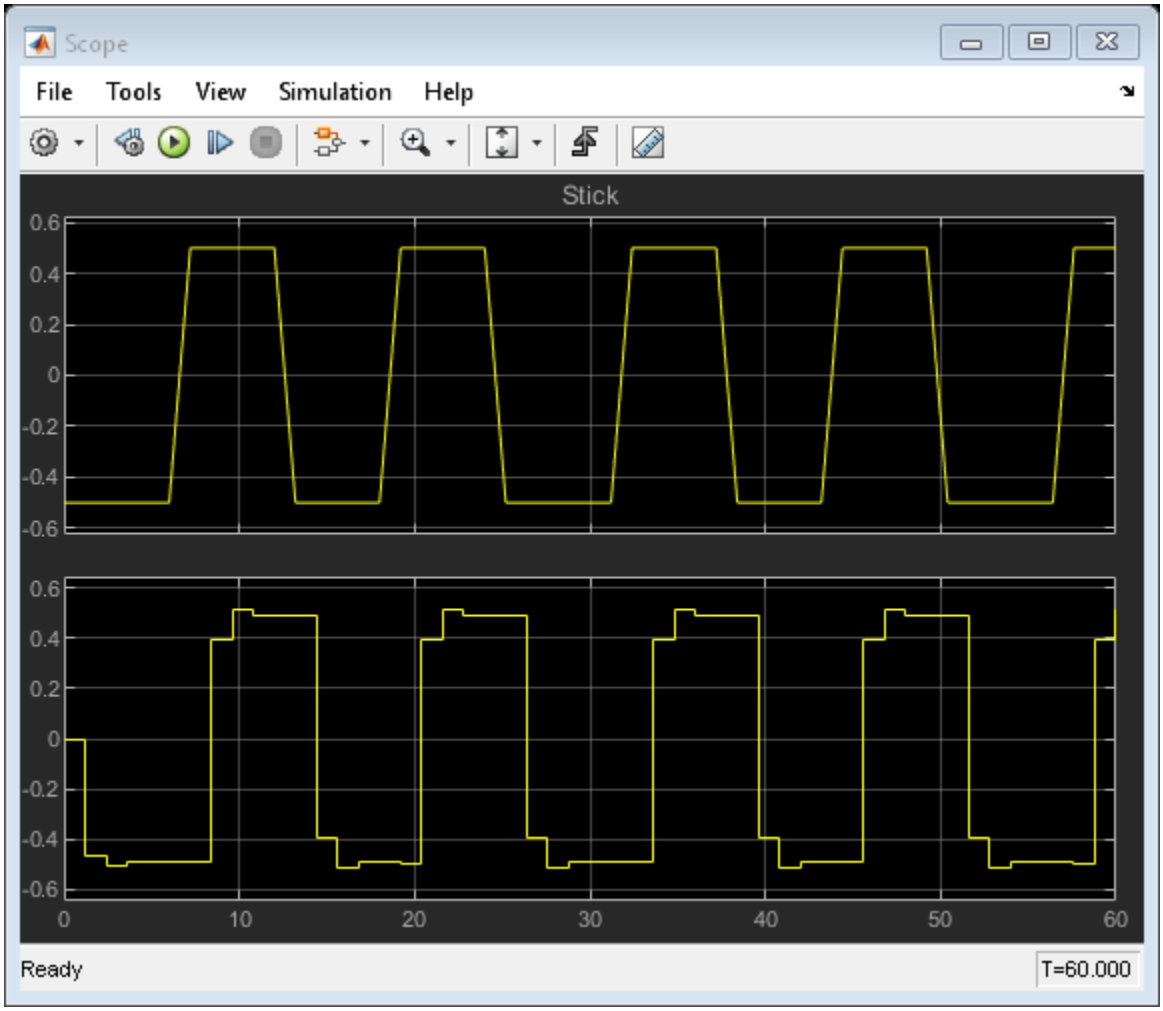


Integrate FMU Components in Simulink

Once the FMU is successfully exported, you may use the top model `fmuexport_aircraft_flight_control_top` to fully integrate the system for testing.

```
open_system('fmuexport_aircraft_flight_control_top');
set_param('fmuexport_aircraft_flight_control_top', 'SimulationCommand', 'Update');
sim('fmuexport_aircraft_flight_control_top');
```


Aircraft Longitudinal Flight Control System
This example models a flight control algorithm
for the longitudinal motion of an aircraft.
1
u
u
fmuexport_aircraft_flight_control_plant
alpha
Stick
Stick
Pilot
Aircraft Longitudinal Flight Control Plant
Copyright 2021 The MathWorks, Inc.




```
close_system('fmuexport_aircraft_flight_control_top', 0);
```

The generated simulation result might be slightly different than simulation results observed for `fmuexport_aircraft_flight_control_plant` model in Simulink. This is expected, for more information refer to “Co-Simulation Execution”

Reuse the Standalone Co-Simulation FMU with Source Code for Cross Platform Workflow

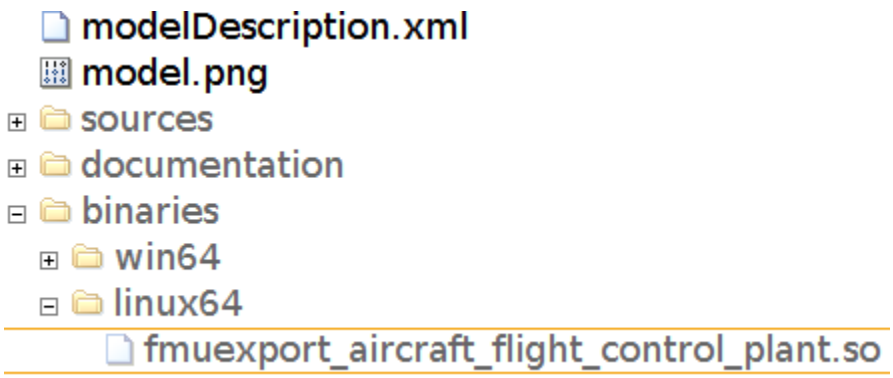
Users can reuse the standalone co-simulation FMU with source code on another platform by regenerating the binaries and packing the generated binaries in the FMU. Given below are the steps to reuse the FMU on another platform.

1. Unzip the standalone co-simulation FMU with source code and compile the source code to generate the platform specific binary files.

```
%Example
gcc -I<directoryWithFMUHeader> -I<directoryWithSourceCode> -c fmuexport_aircraft_flight_control_
gcc -shared -o fmuexport_aircraft_flight_control_plant.so fmuexport_aircraft_flight_control_plan
```

2. Move the generated binary file in the binaries folder.

```
% Example of folder structure when binaries were packed in FMU on 64-bit
% Linux platform
```



3. Repackage the files in FMU for cross-platform use