S-function Tutorial Matlab R2022a
EECS 461, Winter 2023[1]

Overview

MATLAB S-functions are an effective way to embed object code into a Simulink model. These S-functions can be written in a few languages, C being the one most relevant for our purposes. This tutorial is meant to serve as a guide–almost a walk-through–that explains what all you need to know regarding S functions and how to use them in your final project Simulink models. In lab 1, we wrote a simple adder program. We followed this up in lab 8 where we used Simulink blocks to represent the Digital Input and Output blocks and to shift and add integers. We will now combine concepts from labs 1 and 8 and utilize the best of both worlds. Understanding how and when to use S-functions in your final project is almost guaranteed to make your project significantly more understandable to both you and the GSI who helps you debug your project.

Example: Eight-bit adder

1. The first step is to make sure you are in the correct directory. Make sure you are working from the N drive. Create a folder called `Adder` anywhere on your N drive.

2. The second step is to select a C compiler in MATLAB. Follow the steps below if you are using a CAEN computer

   (a) At the MATLAB prompt, enter `mex -setup`

   (b) Select Microsoft Visual C++ 2015 Professional (C) as the compiler.

   This Microsoft compiler cannot generate object code for the NXP S32K. It should, however, allow you to complete most of your debugging outside the EECS 461 lab. Follow the steps below if you are using an EECS 461 lab computer

   (a) At the MATLAB prompt, enter `mex -setup`

   (b) Select MinGW64 compiler

3. Create a new Simulink model in Matlab 2022a. Insert 8 Constant Blocks, 2 Muxs, 1 Demux, and 5 display blocks. Insert the S-Function Builder block from the Library Browser, figure 1 under User-Defined Functions. Your Simulink model should look something like figure 2.

4. Double-click on the S-Function Builder block. In the text box labeled S-function name, give your new S-Function a suitable name. This name has to be different from your model name. Select C language. When you name the S-sunction, all functions in the S-Function Builder editor change, and the S-function name becomes a prefix to all wrapper functions.

5. In the Ports and Parameters interface specify the names, types and sizes of your function's inputs and outputs. Add ports and parameters using the pull-down menu in the tool bar at the top of the S-Function builder. Create two input ports, `num1` and `num2`. These should be 1-D vectors and have 4 rows, as seen in the figure below. These settings mean that your S-Function will receive 2 vector inputs, each containing 4 elements. Create one output port called `sum` which is a 5 element vector. See figure 3 for reference.

6. Under settings, check the Direct feedthrough box. Since there are neither continuous nor discrete states, no other settings need to be specified.

7. When generating the S-function code, the S-Function Builder block generates a wrapper using the name of your model and a wrapper function in the form of system_name_function_name_wrapper. Enter the code that computes the outputs of the S-function at each simulation time step. The eight-bit adder code is:

[1] Revised November 22, 2022.

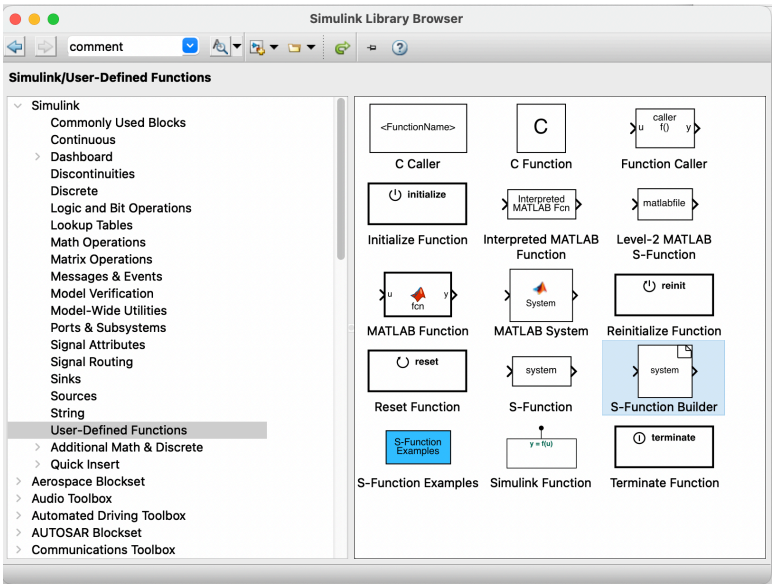

Figure 1: Simulink S-function Builder block

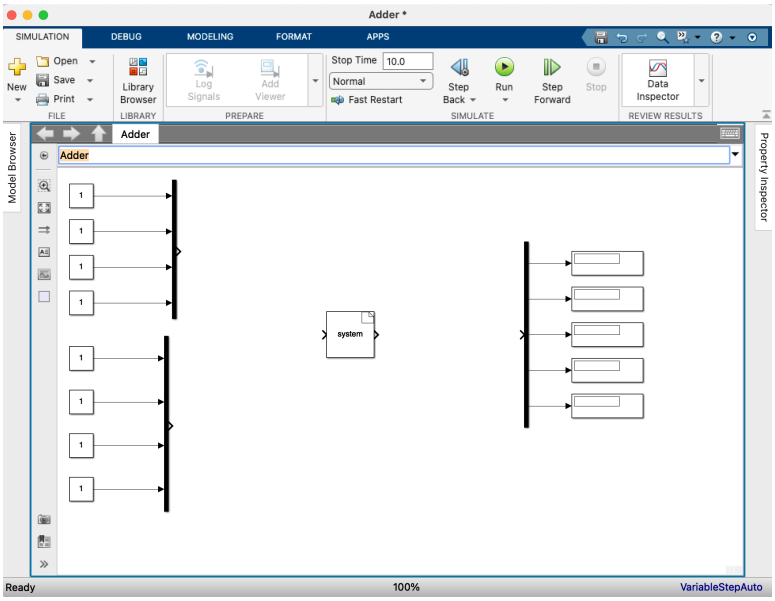

Figure 2: Eight bit adder Simulink model

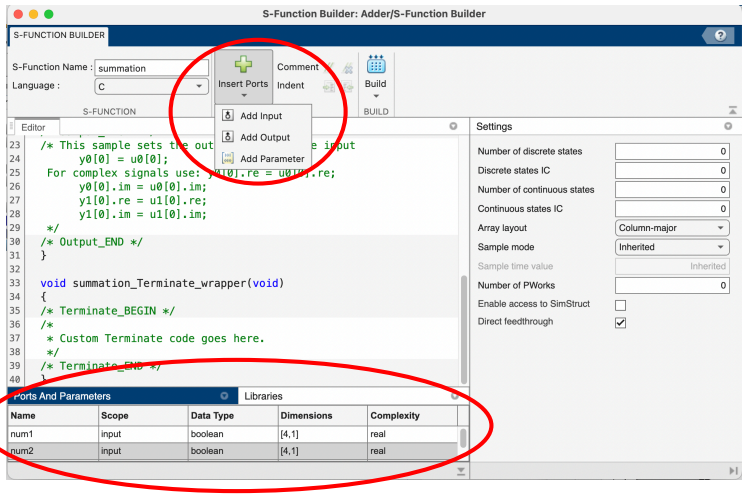

Figure 3: Add input and output ports. Input ports are 4-element vectors of type Boolean. The output port (not shown) is a 5-element Boolean vector.

```
int bit = 0;
unsigned int number1 = num1[0] + (num1[1]<<1)+(num1[2]<<2)+(num1[3]<<3);
unsigned int number2 = num2[0] + (num2[1]<<1)+(num2[2]<<2)+(num2[3]<<3);
unsigned int result = number1 + number2;
for (bit = 0; bit<5; bit++) sum[bit] = (result & (1 << bit)) >> bit;
```

This code should be included in the

`void sfun_sum_Outputs_wrapper`

as shown in figure 8.

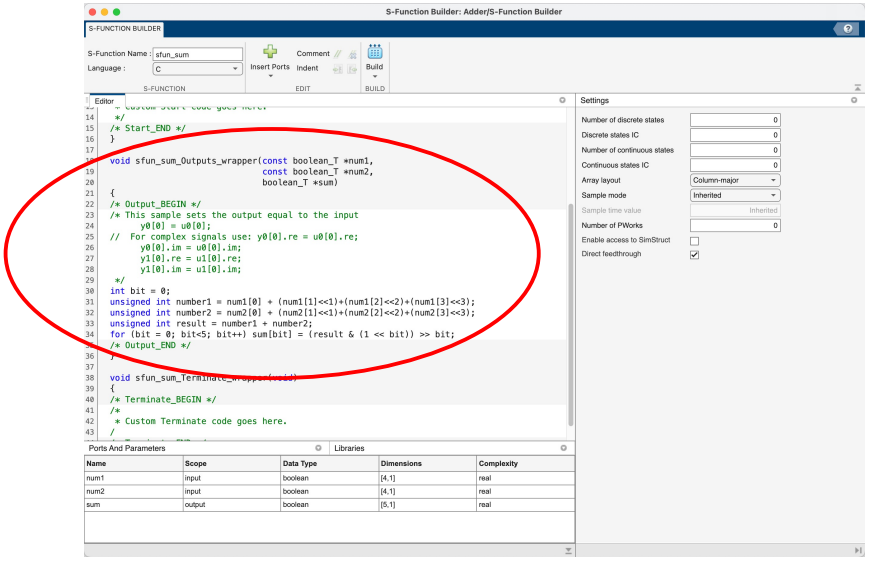

Figure 4: Eight-bit adder code is written in the generated sfun_sum_outputs_wrapper.

8. You are now ready to build your function. In the pull-down build menu in the toolbar, check the boxes Show compile steps, Create a debuggable MEX-file and Generate wrapper TLC. Click Build. See figure 5.

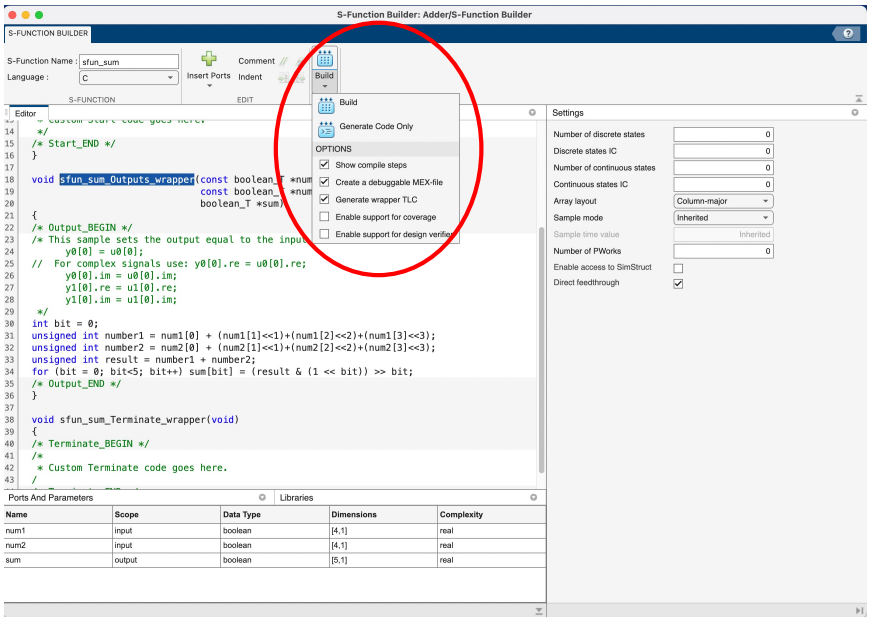

Figure 5: Eight-bit adder code is written in the generated sfun_sum_outputs_wrapper.

9. Your S-function block should now be updated with the input and output ports.Complete the model as shown in figure 6. Change the data types of the constant blocks to Boolean and run the simulation.

Example: First order system

The previous example was a simple arithmetic operation with no dynamics. Let us now consider a simple first order dynamic system with transfer function

$$\frac{y_0(s)}{u_0(s)} = \frac{1}{\tau s + 1}.$$

Although this is easily accomplished with a standard Simulink block, we'll use S-Function Builder as illustrated in figure 7. Note that in the settings pane, the number of continuous states is set to 1 and the direct feedthrough box is unchecked.

S-function dynamic systems must be written in state-space form, using specific notation. Our system in state-space is

$$\dot{x} = Ax + Bu_0$$
$$y_0 = Cx$$

where $A = -\frac{1}{\tau}$, $B = \frac{1}{\tau}$ and $C = 1$. In S-functions, derivatives are written `dx` and states are written `xC`, so our state-space system with $\tau = 0.10$ is

```
dx[0]=-10*xC[0]+10*u0[0];
y0[0]=xC[0];
```

The code is shown in context in figure 8. After building the S-function as described in the first example, the simulation results are shown in figure 9.

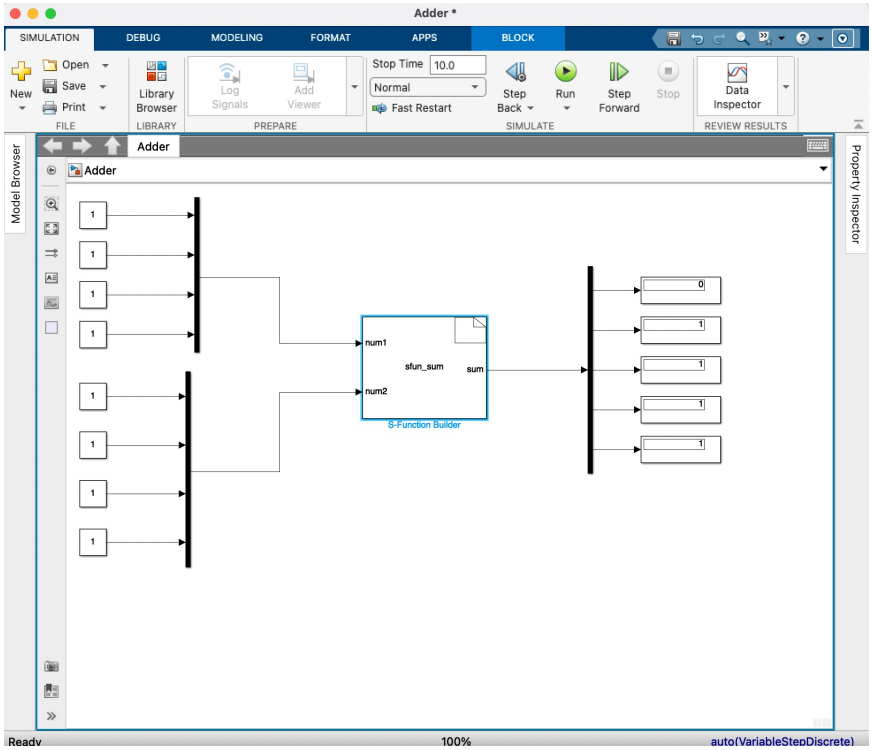

Figure 6: Eight-bit adder S-function input and output.

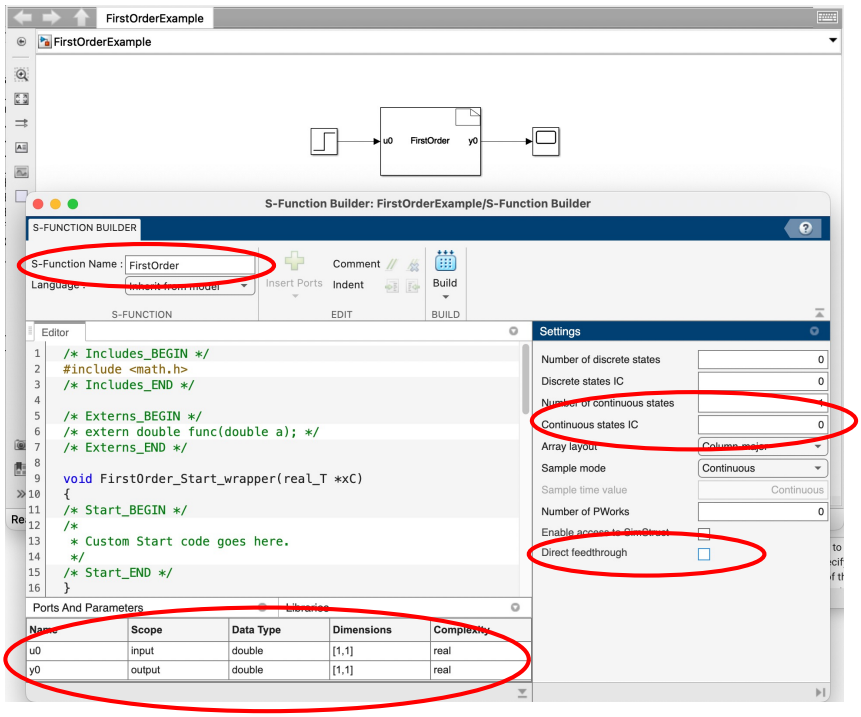

Figure 7: Simple system with one continuous state and no direct feedthough.

Editor

```
1  /* Includes_BEGIN */
2  #include <math.h>
3  /* Includes_END */
4
5  /* Externs_BEGIN */
6  /* extern double func(double a); */
7  /* Externs_END */
8
9  void FirstOrder_Start_wrapper(real_T *xC)
10 {
11 /* Start_BEGIN */
12 /*
13  * Custom Start code goes here.
14  */
15 /* Start_END */
16 }
17
18 void FirstOrder_Outputs_wrapper(real_T *y0,
19                                 const real_T *xC)
20 {
21 /* Output_BEGIN */
22 /* This sample sets the output equal to the input
23       y0[0] = u0[0];
24  For complex signals use: y0[0].re = u0[0].re;
25       y0[0].im = u0[0].im;
26       y1[0].re = u1[0].re;
27       y1[0].im = u1[0].im;
28 */
29
30 y0[0]=xC[0];
31 /* Output_END */
32 }
33
34 void FirstOrder_Derivatives_wrapper(const real_T *u0,
35                                     real_T *y0,
36                                     real_T *dx,
37                                     real_T *xC)
38 {
39 /* Derivatives_BEGIN */
40 /*
41  * Code example
42  *   dx[0] = xC[0];
43 */
44 dx[0]=-10*xC[0]+10*u0[0];
45 /* Derivatives_END */
46 }
47
48 void FirstOrder_Terminate_wrapper(real_T *xC)
49 {
50 /* Terminate_BEGIN */
51 /*
52  * Custom Terminate code goes here.
53  */
54 /* Terminate_END */
55 }
```


Figure 8: S-Function editor with first order system code.

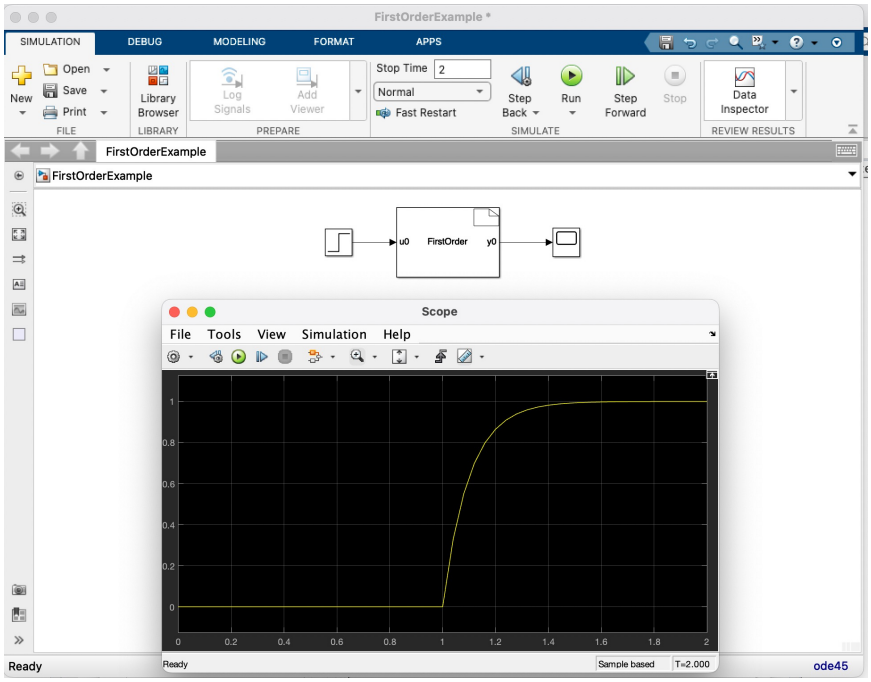

Figure 9: First order system simulation.

Using Model Based Design toolbox

For the project at the end of the semester, you will have to configure the Eight-bit adder example for code generation and run the adder code on the S32K microcontroller using DIP inputs and LED outputs, similar to what you did in lab 1. Do the following:

1. Incorporate the NXP Model Based Design blocks provided on the lab computers in your model. First, add the MBD_S32K1xx_Config_Information Block to your model. Next, replace all of the Constant Blocks with Digital_Input Blocks. Replace all the Display Blocks with digital_Output Blocks. Select the appropriate pin for each block by double clicking on each block. See figure 10 for reference. You can find the blocks in the Simulink Library Browser: `NXP Model-Based Design Toolbox for S32K MCUs/ S32K14x MCUs/ S32K1xx Core, System, Peripherals and Utilities/ GPIO Blocks`.

2. In the MBD_S32K1xx_Config_Information block, modify the compile options (Build Toolchain → MinGW64→ Compile Options) and append -include stdbool.h. Also add -DRT and -DmwSize=size_t, see figure 11. This is necessary because the S-function builder uses booleans when it creates a .c file (in mux blocks, for example); mwSize is a Mathworks defined type; -DRT specifies that the S-function is being built with the Simulink Coder product for a realtime application using a fixed-step solver.

3. You will need to set a fixed time step. Go to Simulation and select Configuration Parameters For Fixed-step size, enter .001. Click OK.

4. Build your model by pressing *ctrl-b*. This will generate an executable linkable file (.elf). Use the S32 Design Studio IDE to upload the file to the S32K144 as follows:

   (a) In your Matlab file browser you should see a new folder ending in _mbd_rtw (See figure 12).

   (b) Inside this folder you will find a file with extension .elf. If you can't find the folder or .elf file, it's likely that you received compile errors after you pressed *ctl-b*.

   (c) Open S32 Design Studio from the desktop (use the workspace you have been using in lab) and click the drop down next to the little bug and select Debug Configurations (see figure 13).

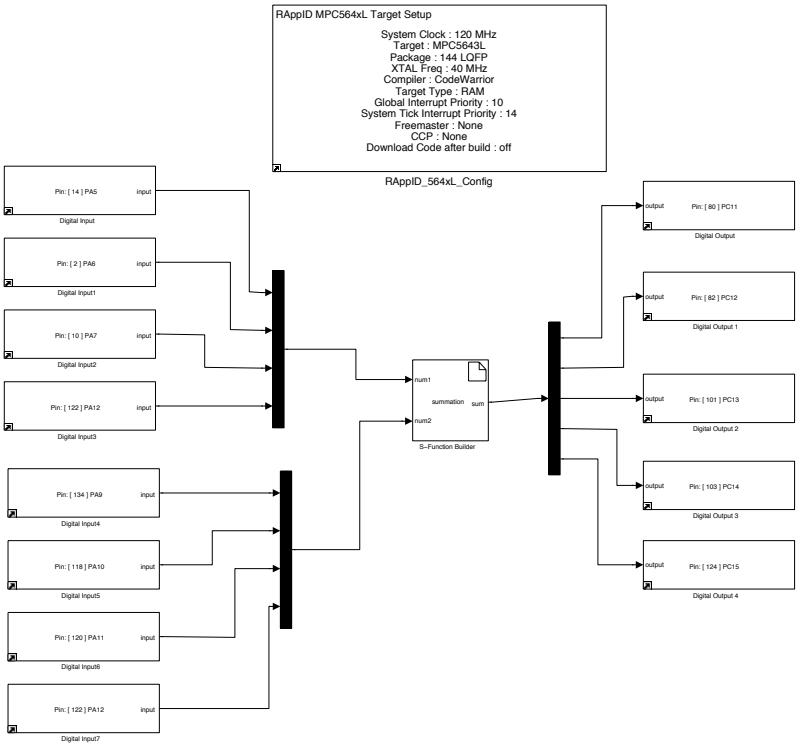

Figure 10: First order system simulation.

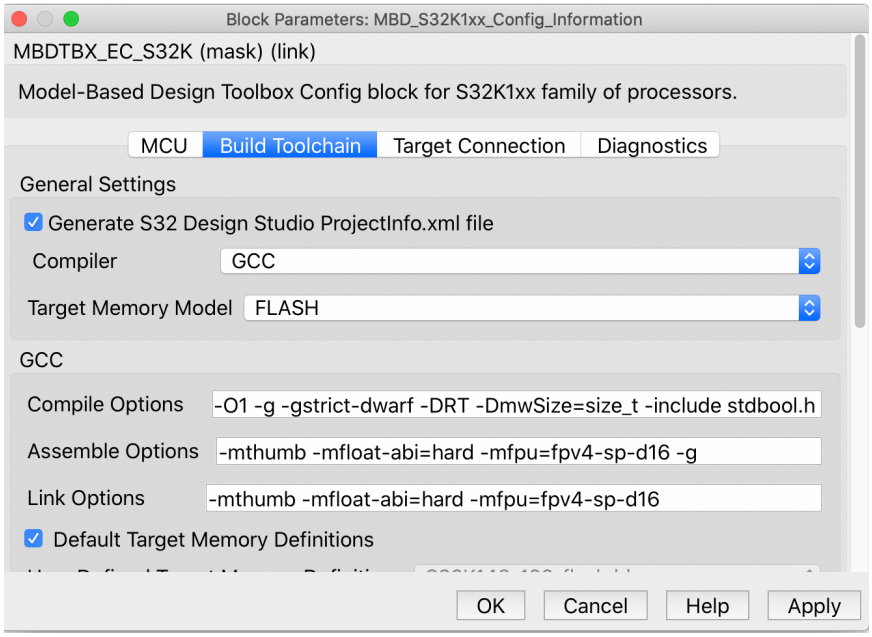

Figure 11: Block parameters: Model Based Design Toolbox.

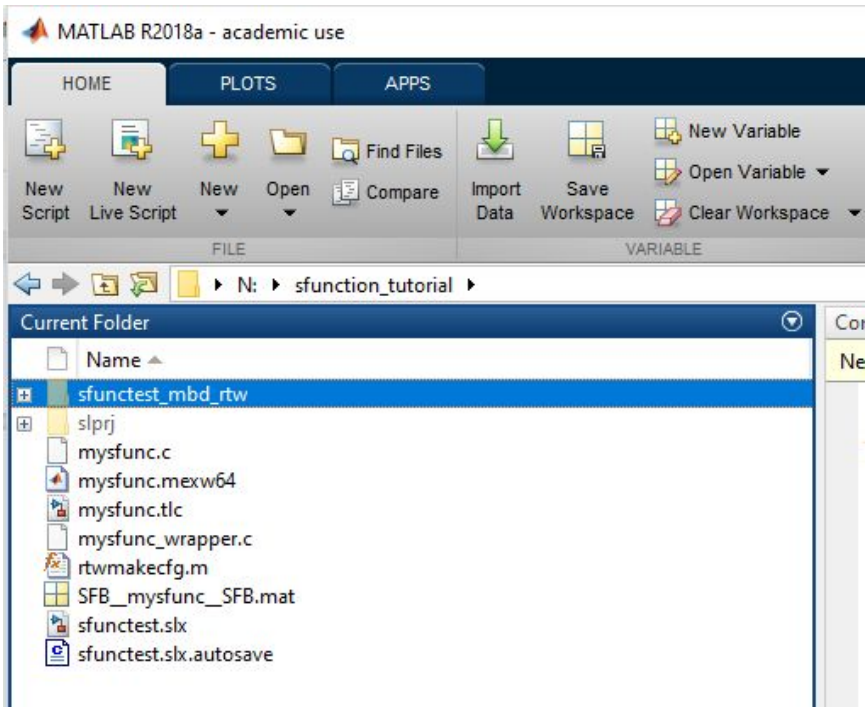

Figure 12: Matlab file browser.

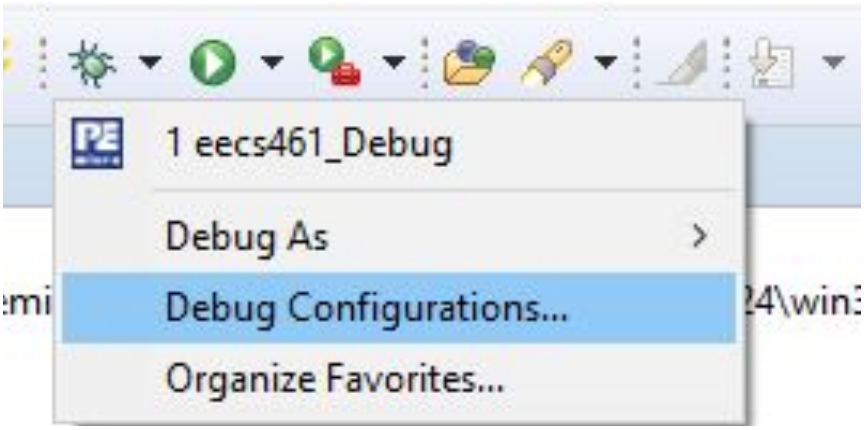

Figure 13: S32 Design Studio debugger.

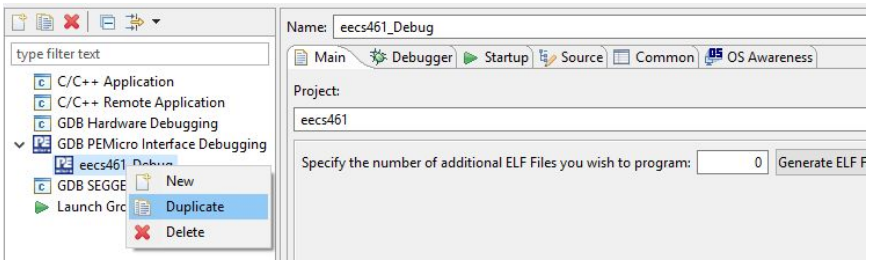

Figure 14: Right click and duplicate eecs461_Debug.

(d) Right click and duplicate eecs461_Debug as in figure 14.

(e) Change the name from eecs461_Debug (1) to eecs461_Debug_elfUpload.

(f) Under C/C++ Application click browse and select the .elf file we found earlier.

(g) Disable auto build and click Apply. The code is now running on your board. See figure 15.

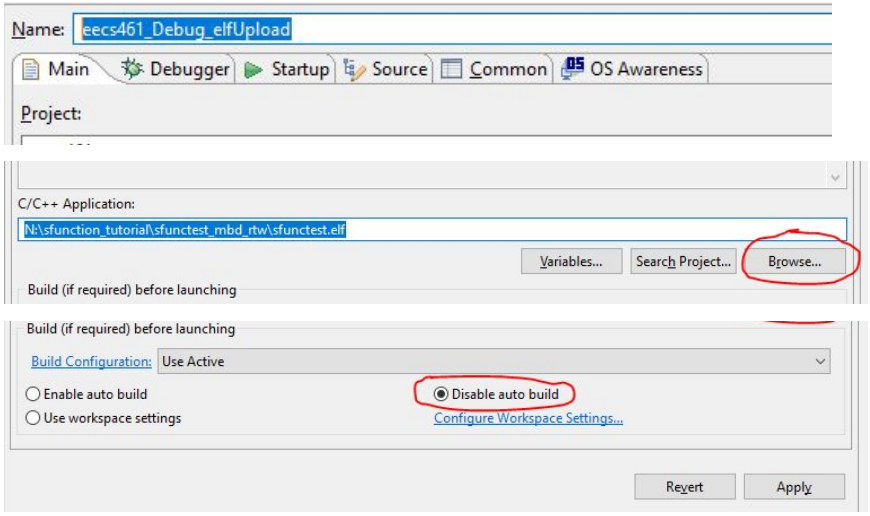

Figure 15: eecs461_Debug_elfUpload.