Instrumentation and Microcontrollers Using Automatic Code Generation

Using and Applying Microcontrollers for the Rest of Us!

Marc E Herniter

Table of Contents

List of Demonstrations

List of Exercises

Lab I
Opening the World of Microcontrollers

Hello World!

A. Introduction

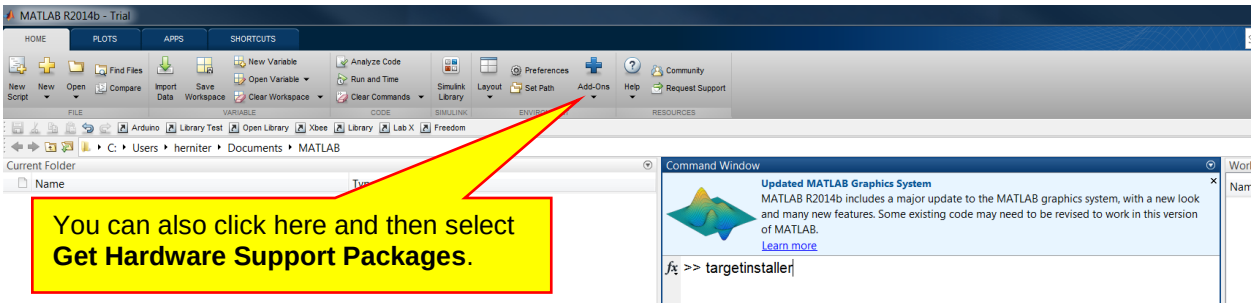
In this lab we will show how to program the Arduino Mega 2650 using automatic code generation facilities provided by the MathWorks Simulink. Instead of learning how to program in C, how to address hardware I/O ports of the Arduino target, and how to download and run the program, we will just draw a block diagram (model) in Simulink and tell Simulink to run the model on the hardware target. The translation of the model to C, compiling the C code, downloading the program, and then running the program are all transparent to us. In higher level courses, you may explore how the process of automatic code generation works. We however, will concentrate on creating some rather complex models to accomplish non-trivial tasks with microcontrollers, and leave the under-the-hood workings of automatic code generation to the higher level courses.

In order to accomplish this task, we must do three preliminary tasks: (1) Install the Arduino software package within MATLAB/Simulink, (2) setup the compilers within MATLAB/Simulink, and (3) examine the Arduino datasheet(s) for pin connections and electrical setup.

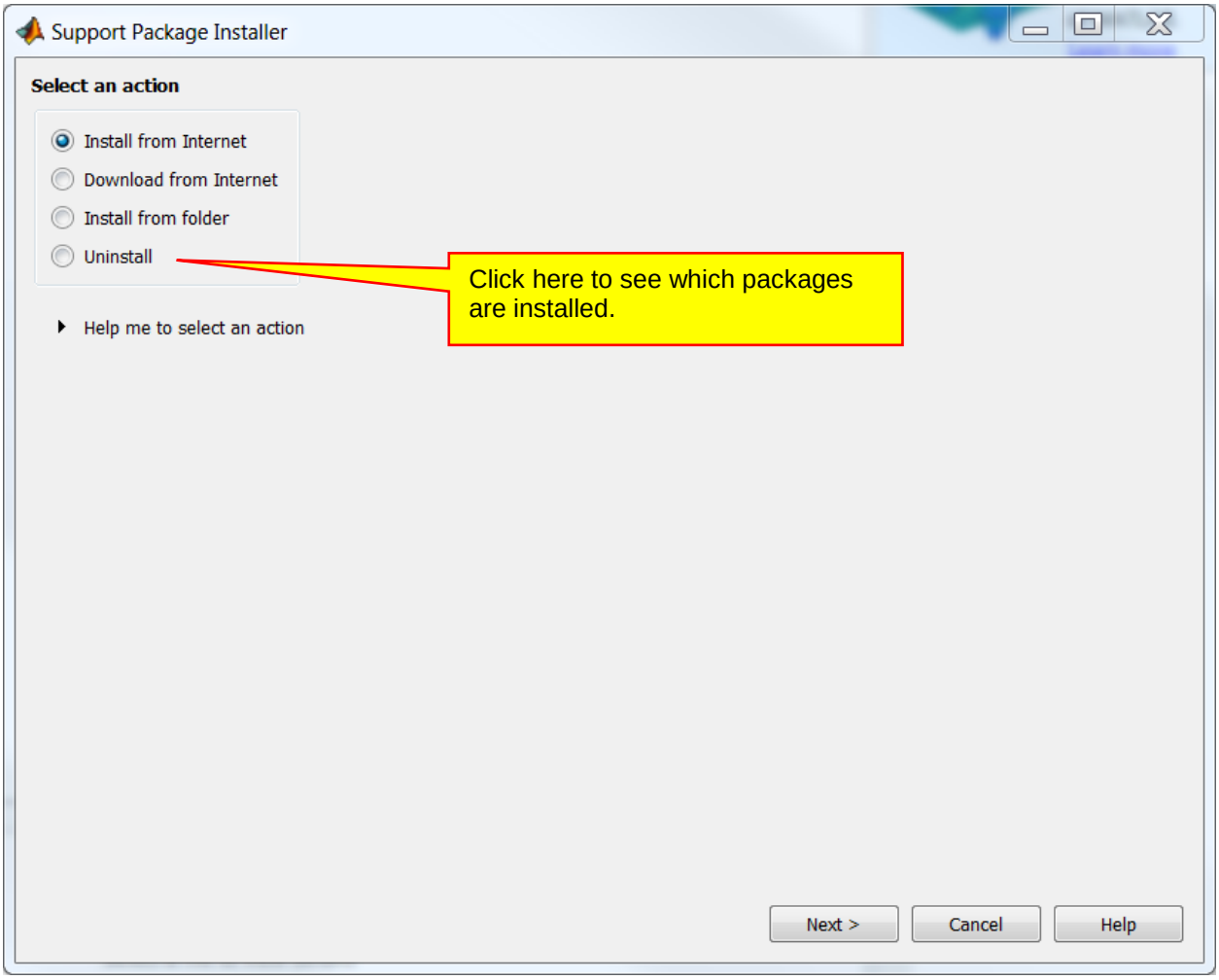
B. Setup

1. MATLAB/Simulink Setup

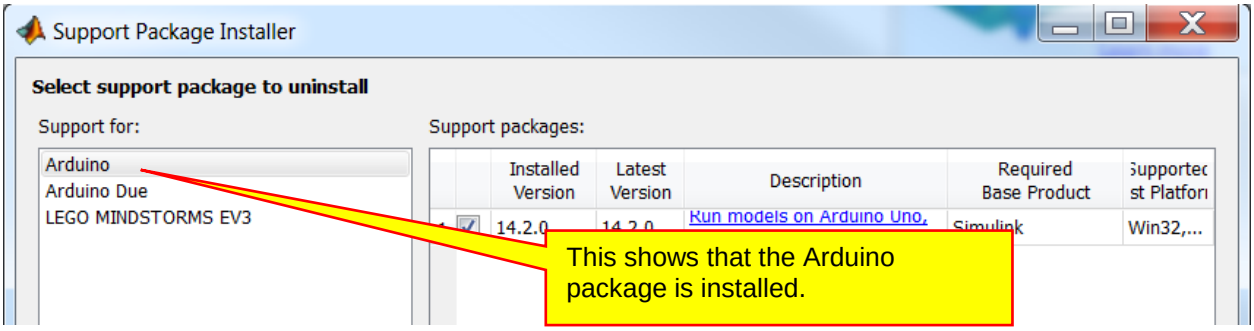
For this setup, we will assume that you are using the 64-bit version MATLAB Version 2014b or higher. The first step is to install the Arduino Target toolbox, but before we do this, we must determine if the Arduino target toolbox has already been installed. From the MATLAB command prompt, type **targetinstaller**:



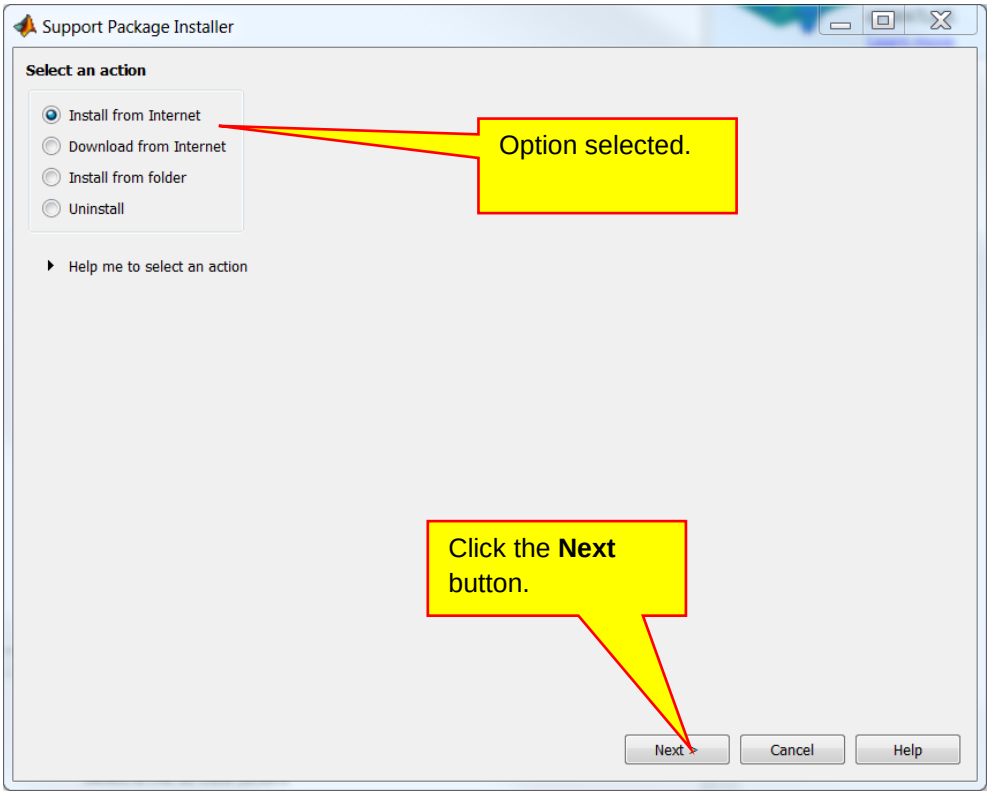
The Support Package Installer should open:



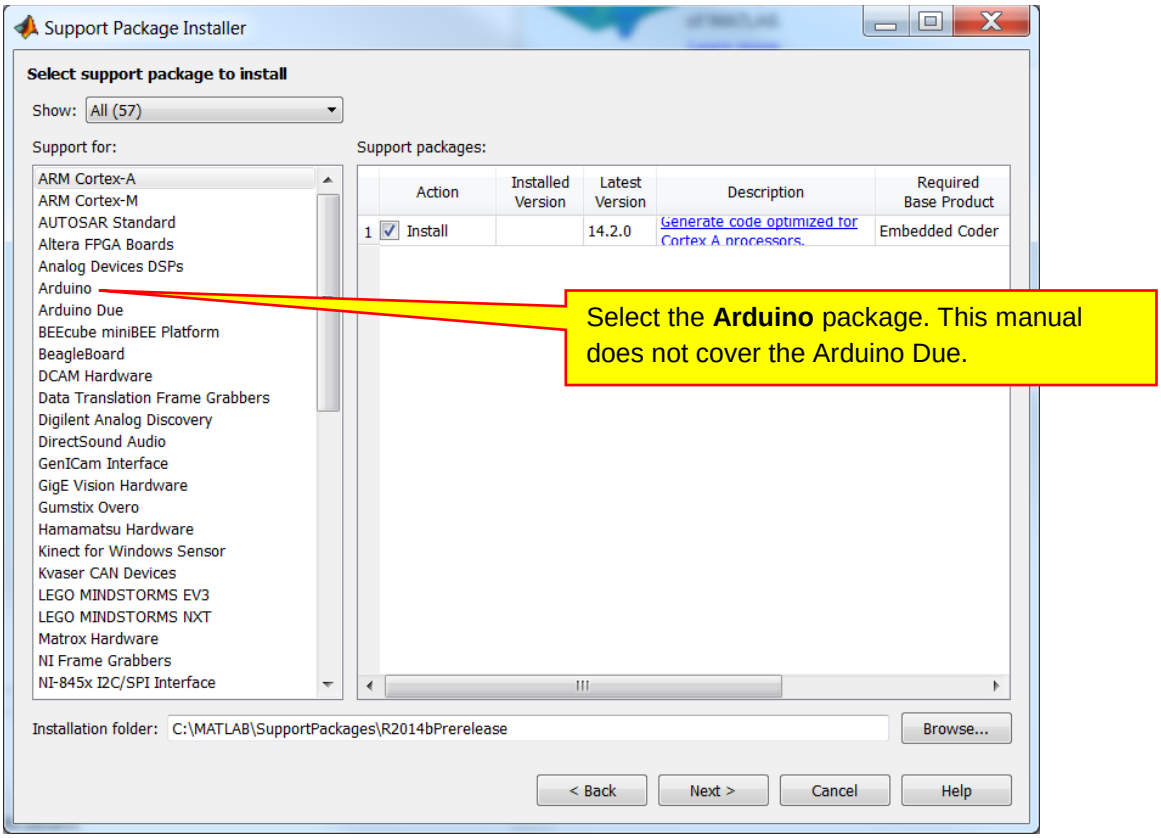
Select the **Uninstall** option and then select **Next** to see which hardware packages, if any, are installed on your computer:



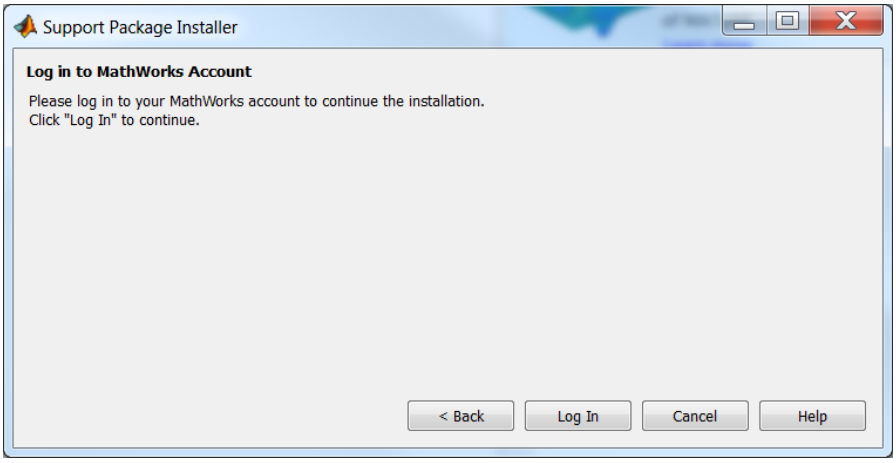
If your window shows that the Arduino support package is installed, then you are done. Click the **Cancel** button and skip to section I.B.2 on page 5. (Do on Uninstall the support package!) If the Arduino support package is not listed, then you will need to install the support package. Click the Back button to return to the previous screen and select the **Install from Internet** option:



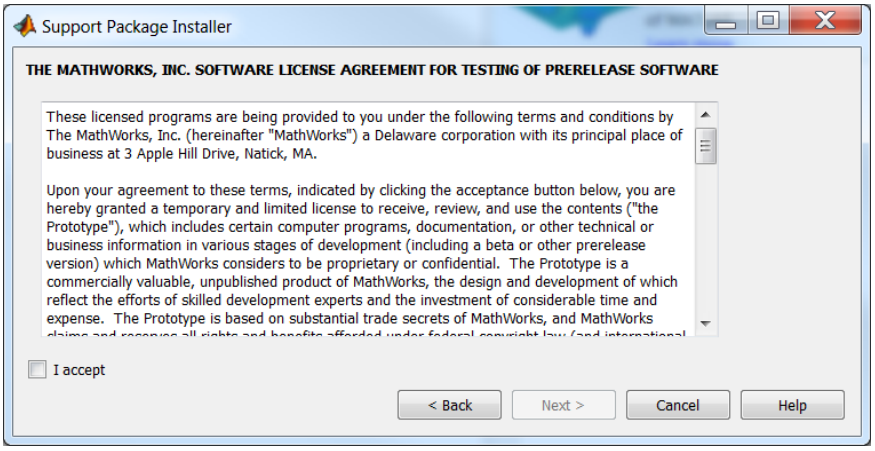
After clicking the next button, the Support Package Installer will list all of the available support packages:



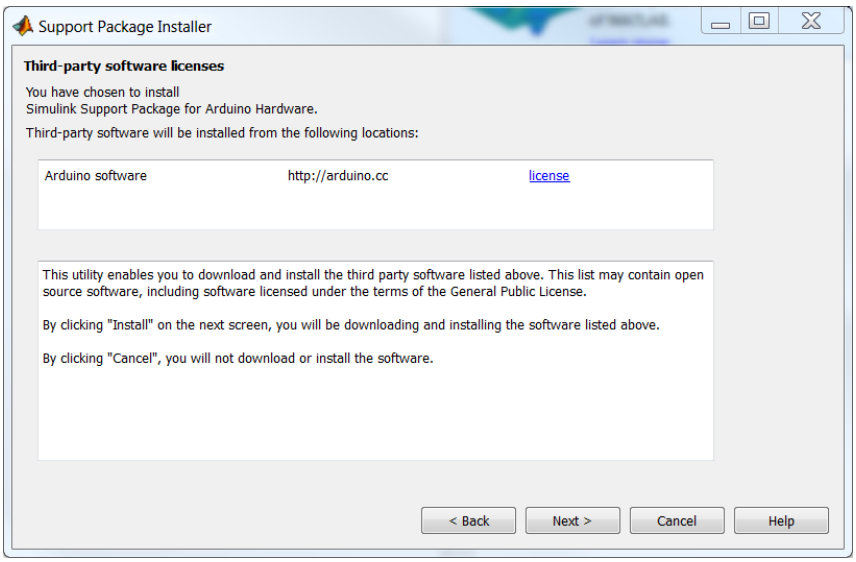
Select **Arduino** package and click the **Next** button:



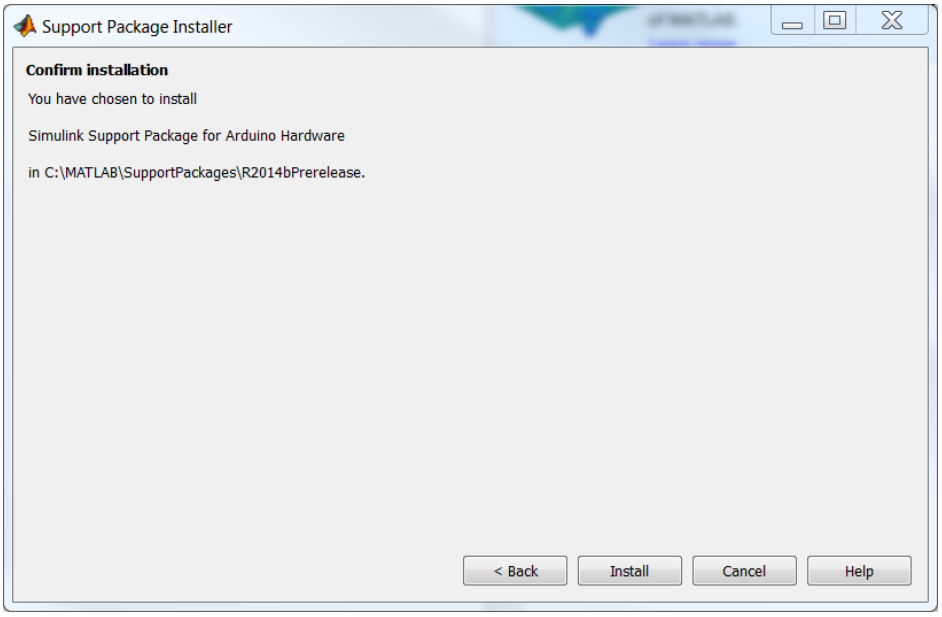
If you do not have a MathWorks account, visit www.MathWorks.com and create an account. Once you have done this, click the **Log In** button and log in to your account. After loggin in, you should see the screen below:



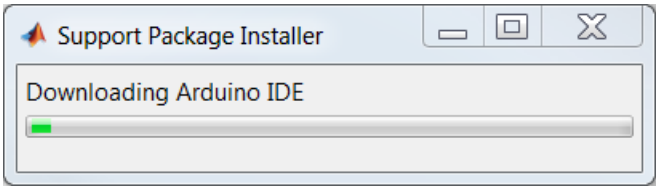
Accept the agreement and click the **Next** button. The window should show that you are attempting to install the Arduino package:



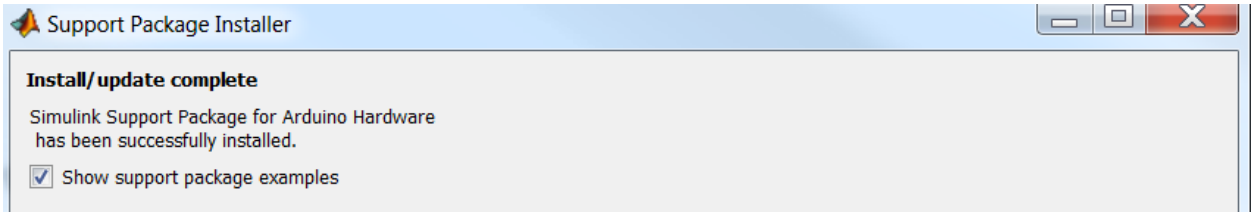
Click the **Next** button to verify that you want to install the Arduino Hardware package:



Click the **Install** button to begin the installation:



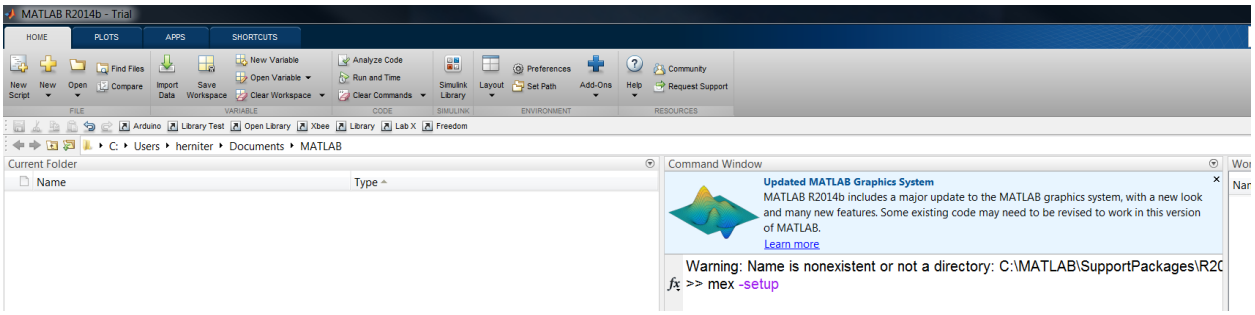
You may be asked for permission by Windows to install the package. The package will be downloaded and then the installation will proceed. After a few moments, you should see the window below:

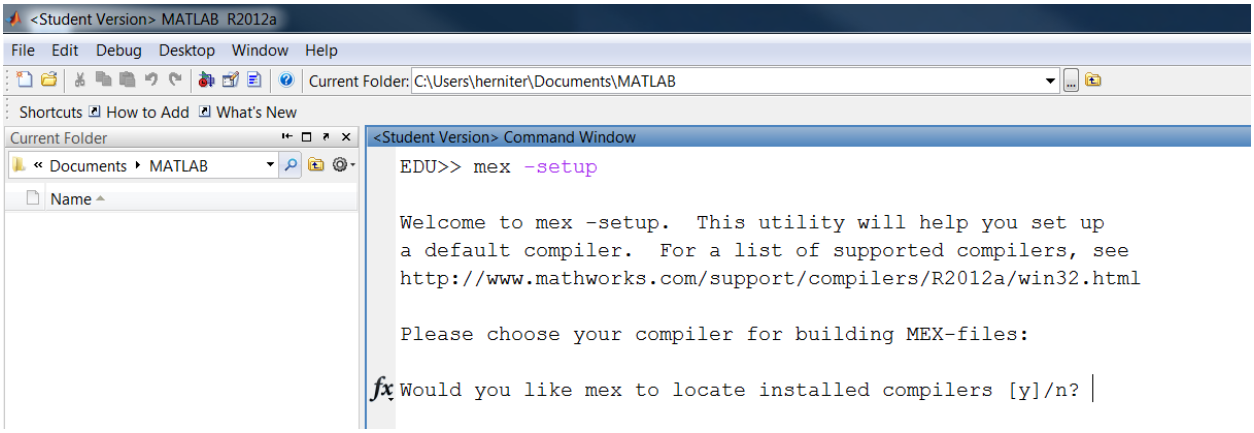


We will skip the **target demos**, so uncheck the box and click the **Finish** button:

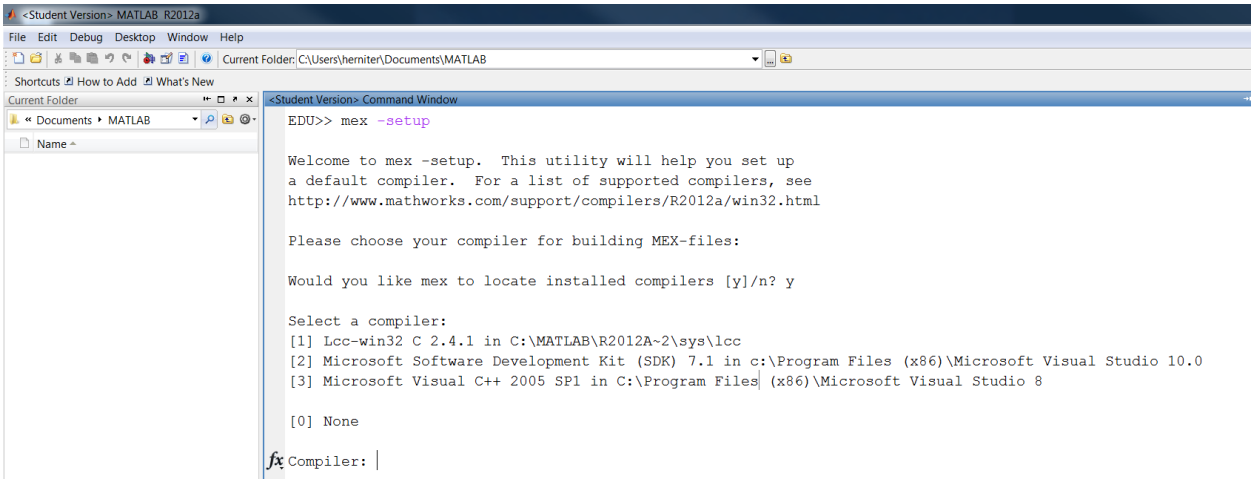
2. MATLAB Complier Setup

In order to translate our Simulink models into C and then run those models on the Arduino Target, we need to specify a compiler. To do this, at the MATLAB prompt type **mex –setup**:

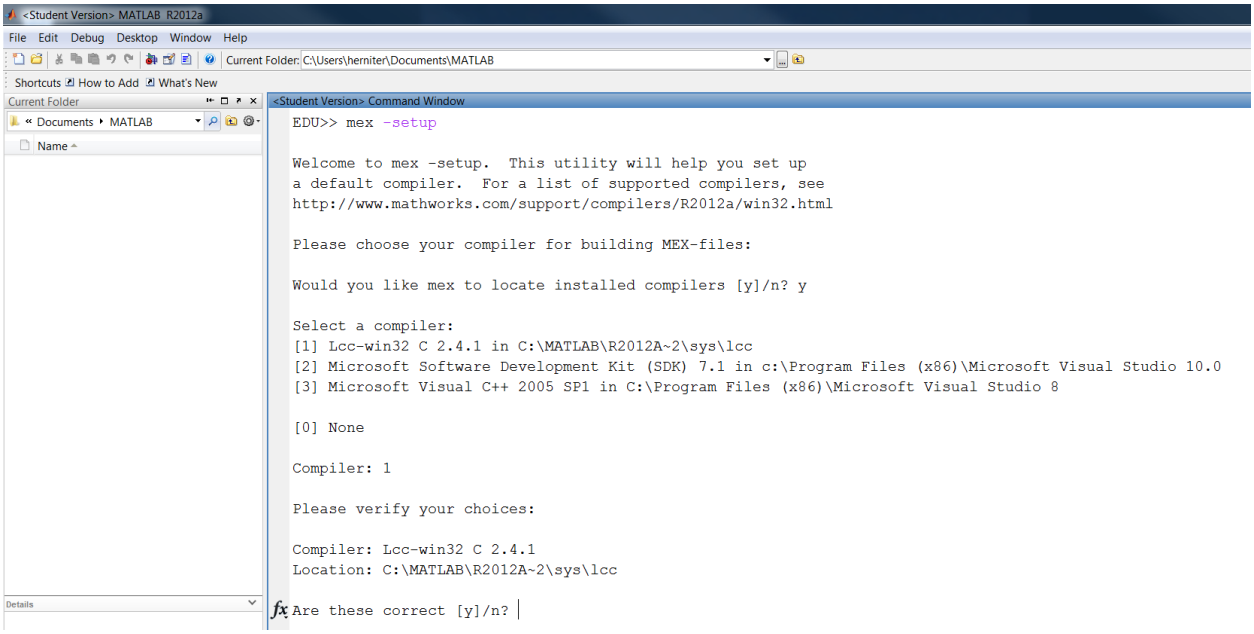




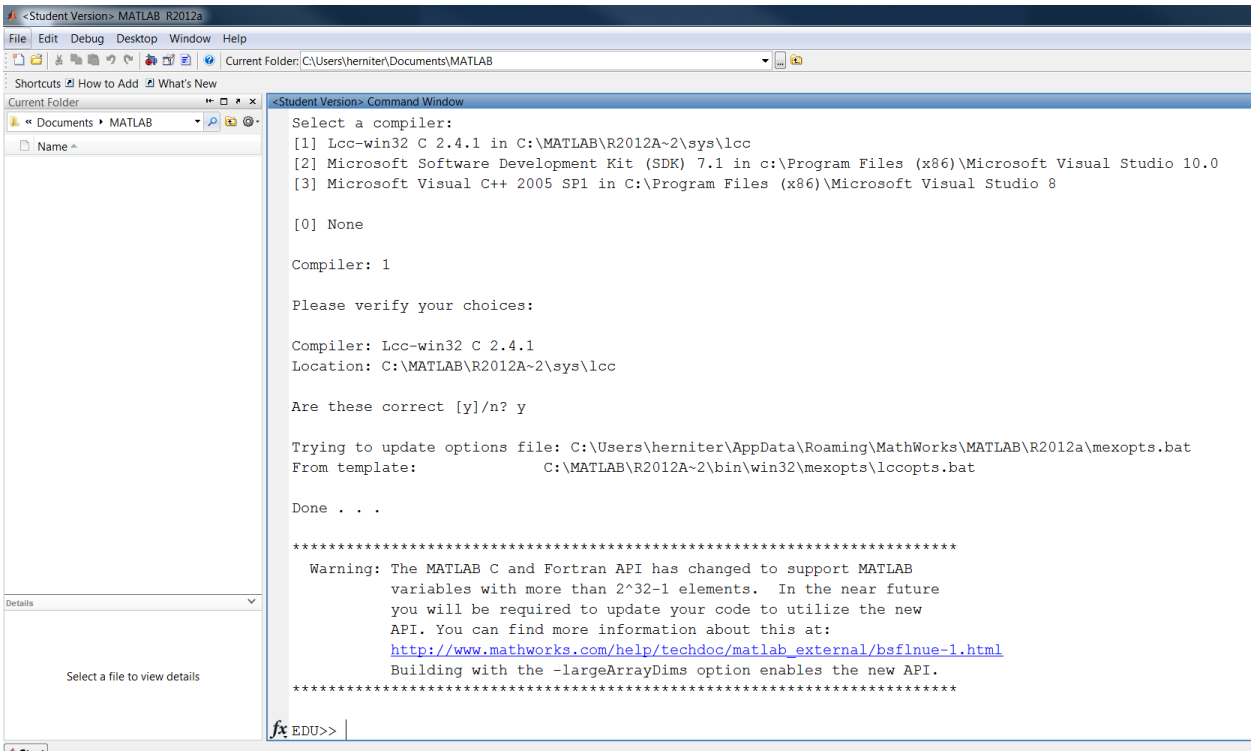
Type **y** to have **mex** search your computer for installed compilers:



Although you may have several compilers installed on your computer, not all of them will be compatible with this version of MATLAB's automatic code generation tools. The compiler that comes with the 32-bit version of MATLAB, is option 1 above in my list (**Lcc-win32…**) Select this option and press the **Enter** key:



Type **y** to install the compiler:



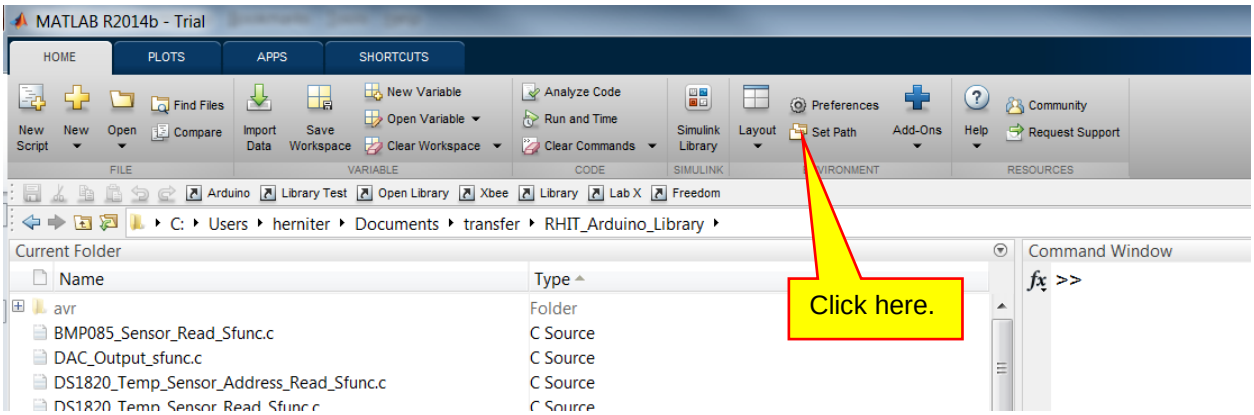
The complier installation is complete.

3. Installing the RHIT Arduino Library

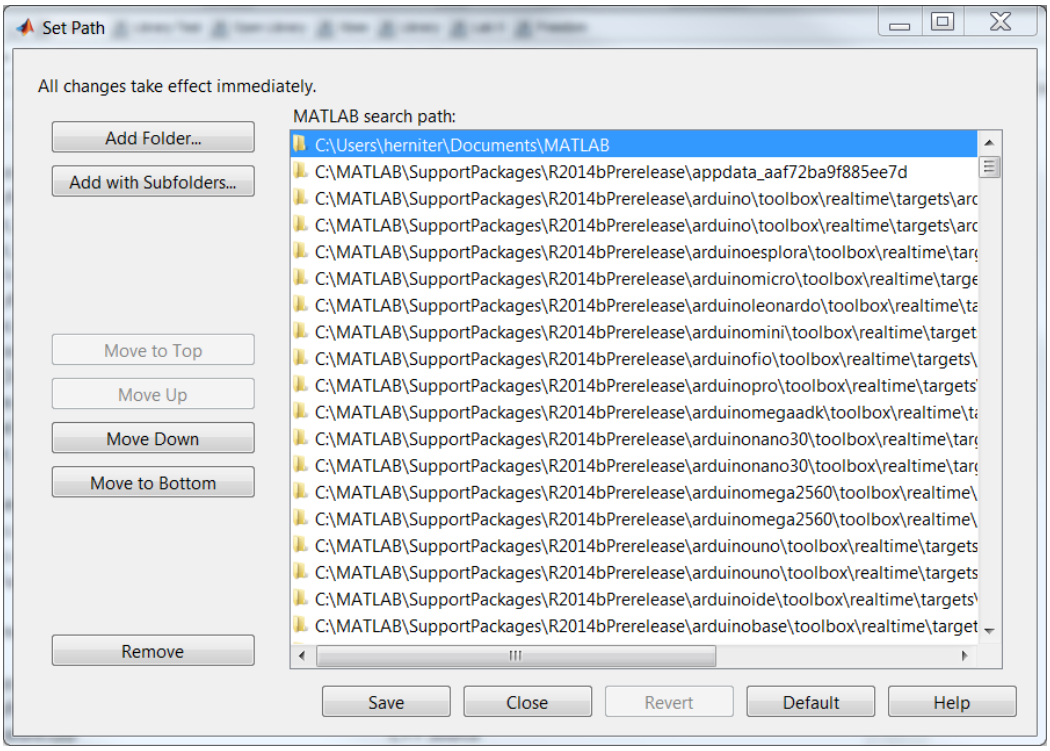
This manual contains several models, functions, and s-functions that can be difficult to implement for people new to MATLAB, C, or Simulink. To make the more advanced functions easier to use, we have created an RHIT Adruino Library for Simulink. We will now install this library:

- Download the library from MathWorks or my website at http://wiki.ece.rose-hulman.edu/herniter/index.php/Instrumentation_and_Microcontrollers_using_Automatic_Code_Generation_%28Arduino_Mega_Version%29. Locate the section of the pages named RHIT Block Library and download the latest version.
- Unzip the file into a directory.
- Open MATLAB click on the **Set Path** button 

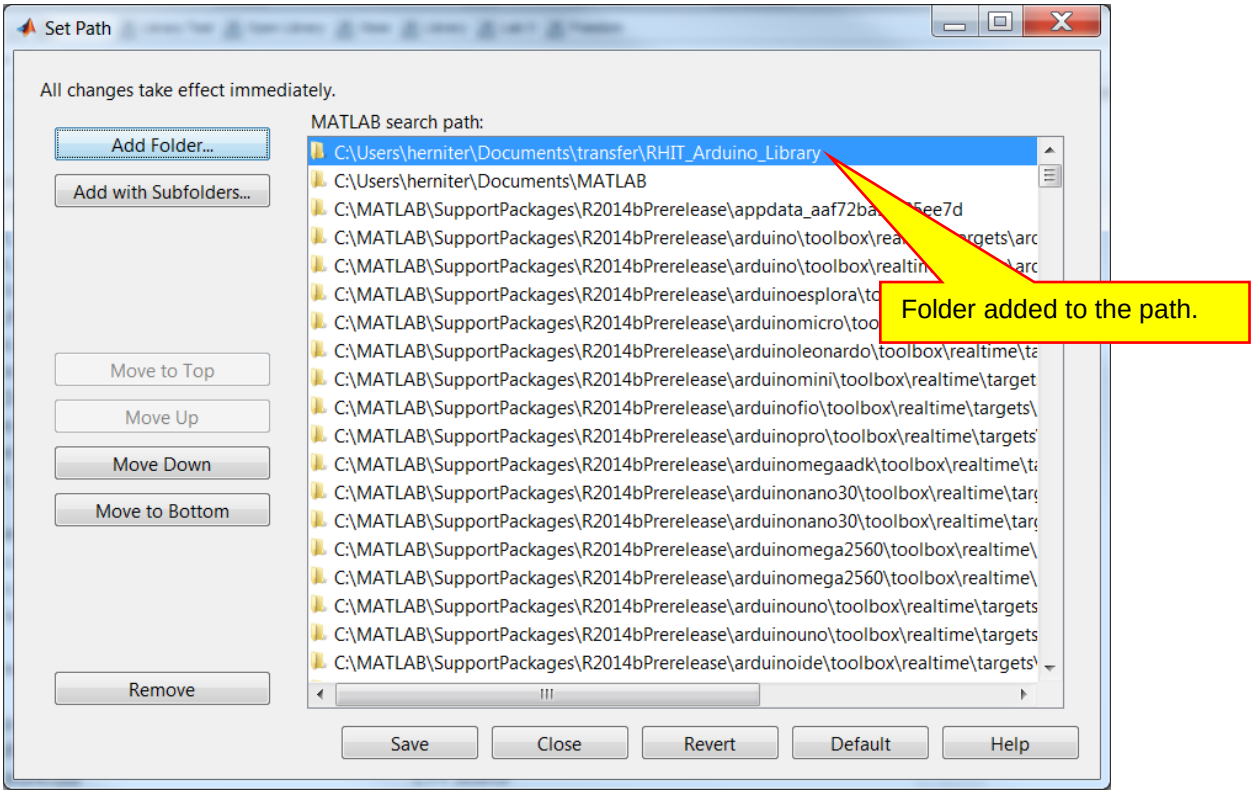
:



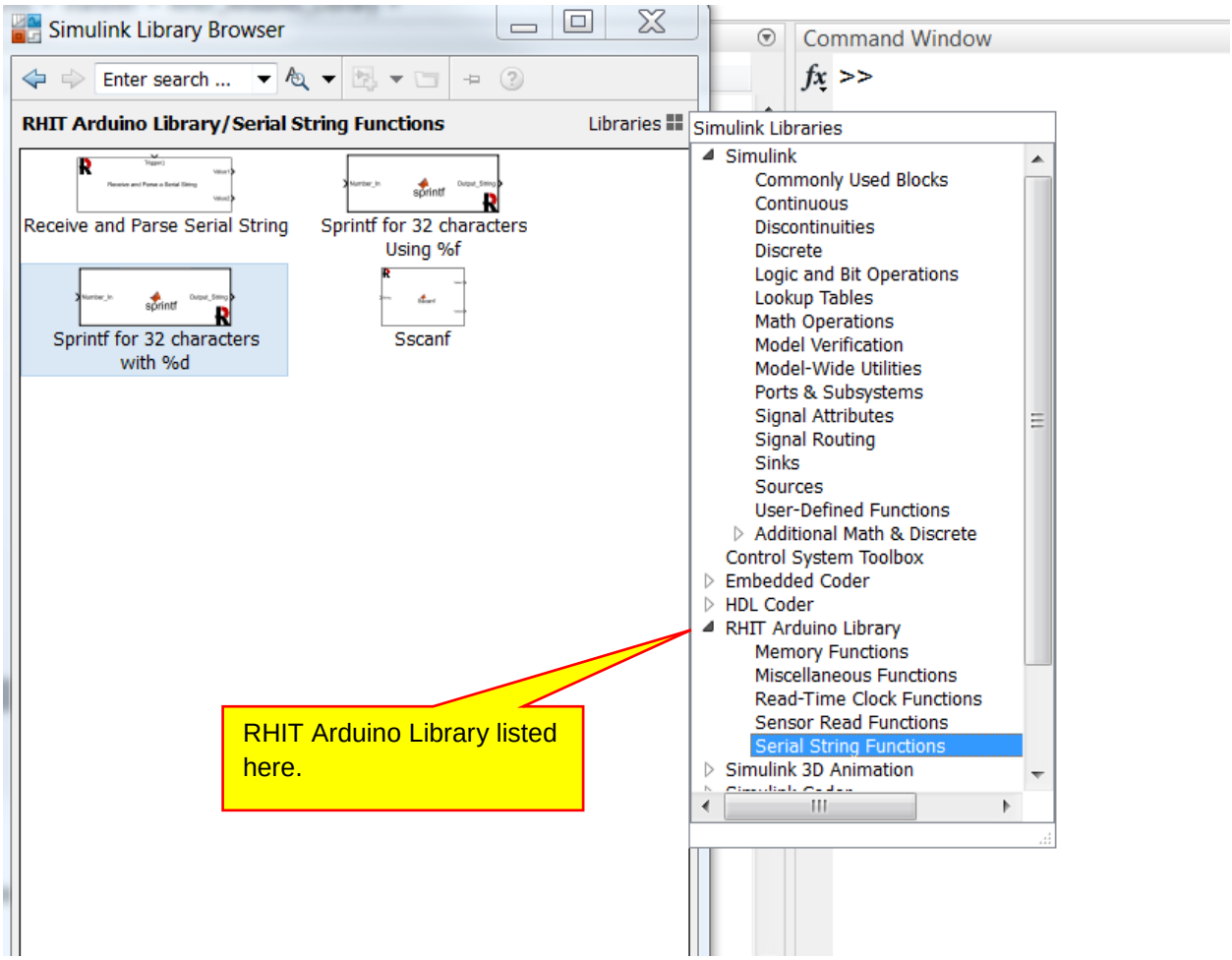
The **Set Path** window will open:



Click on the **Add Folder** button and then locate the directory named RHIT_Arduino_Library and then click on the **Select Folder** button:



Finally, click the **Save** button and the click the **Close** button. Restart Matlab. When you run Simulink, you should see the RHIT Arduino Library as shown:

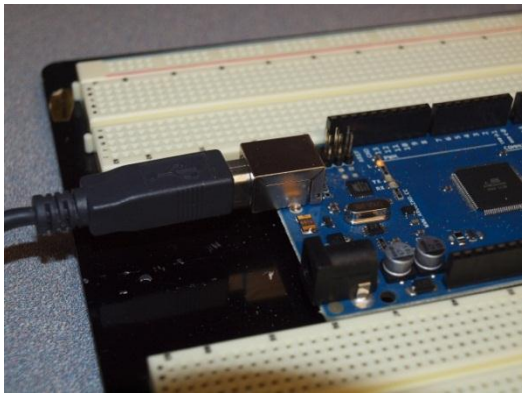


We have not yet covered how to run Simulink yet. If you do not know how to use the Simulink library browser, we will cover it later. At that point, verify that you have the RHIT libraries installed correctly.

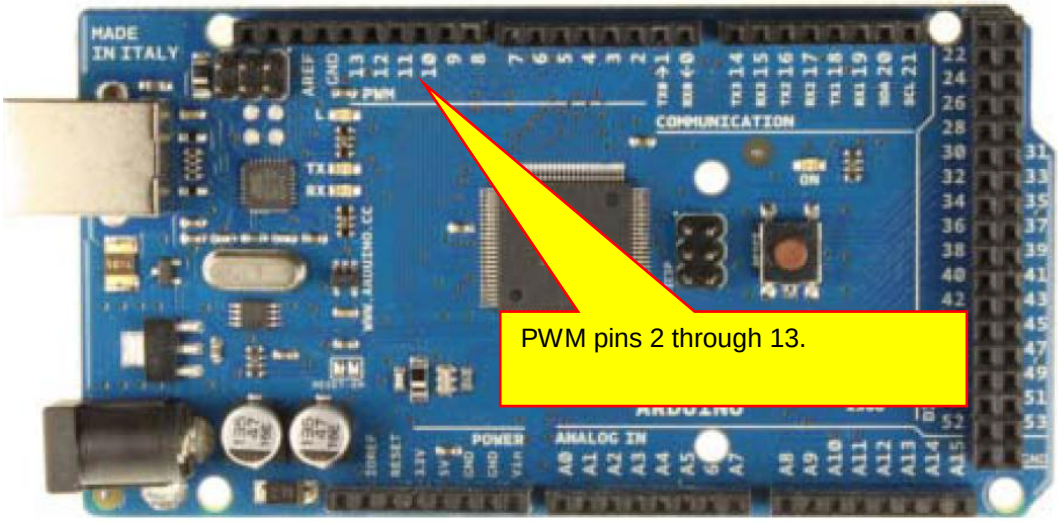
4. Arduino Electrical Setup

The last thing we will look at is the Arduino Mega board schematic. Although many of the pin connections are obvious from the silkscreen on the board, the first model we will build uses the on-board LED and the I/O port connection for this LED is not obvious.

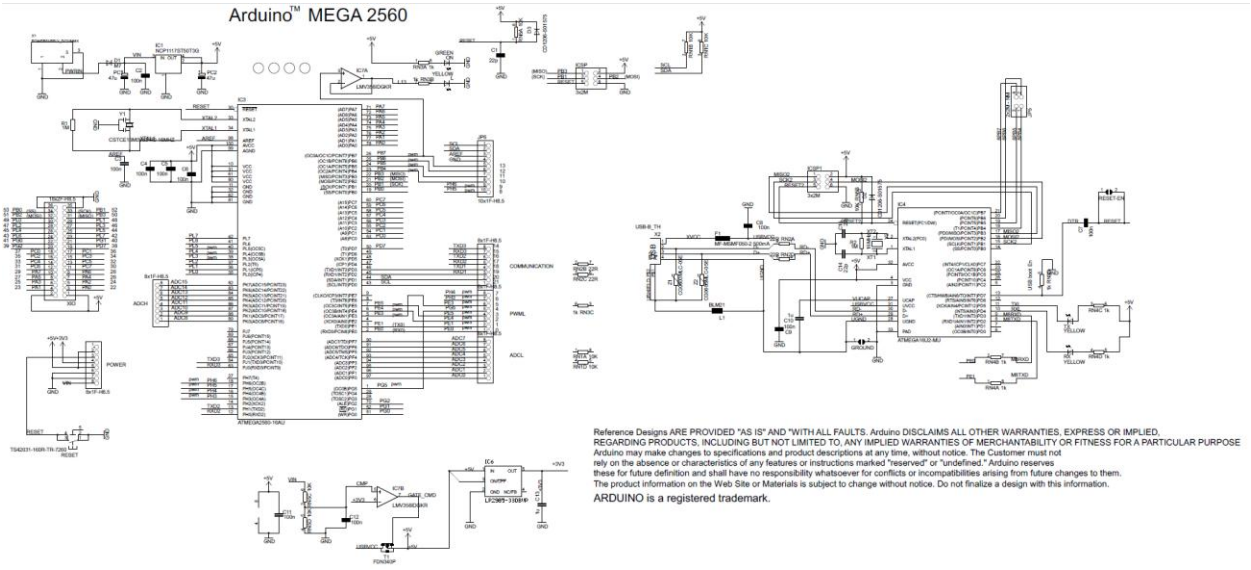
For power connections, the Arduino Mega can be powered from your USB port (as long as the Mega and external hardware does not exceed the power limitations of the USB port) or an external power supply. For the first few labs, we will use the USB supply and we will not even consider power draw. (Though you might want to research just how much current a USB port can supply and how current much the Mega draws…) Plug in the Mega to your computer as shown:



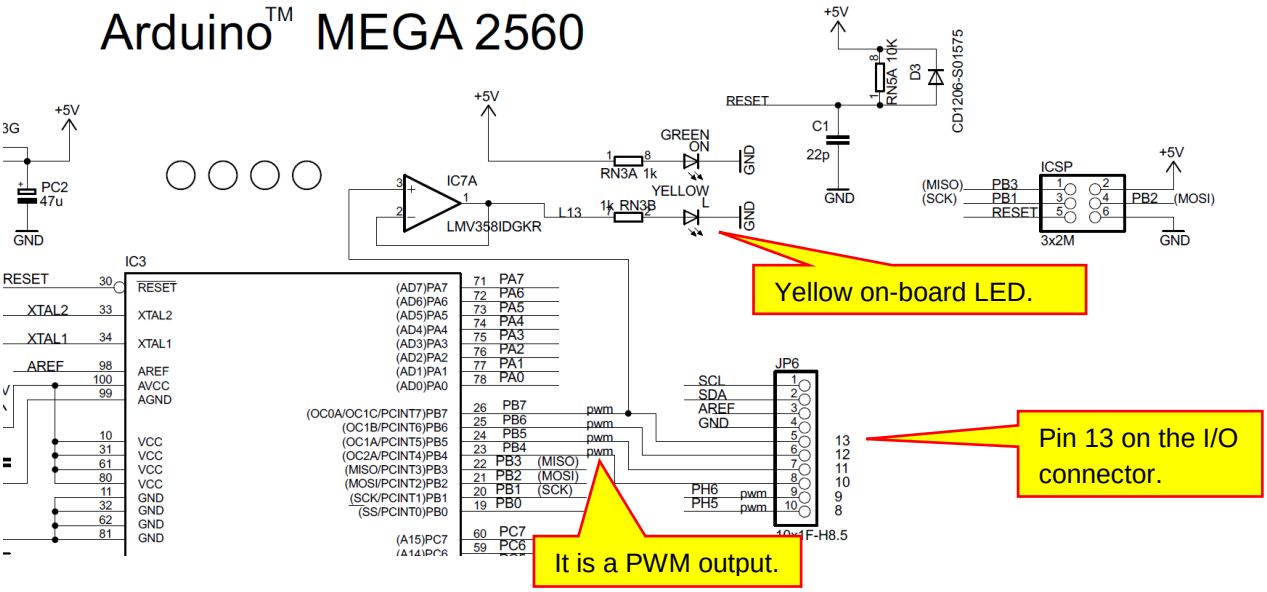
Next, we have to examine the schematic and board layout of the Mega to determine the pin connection of the on-board LED. First, we will look at the silkscreen of the Arduino Mega:



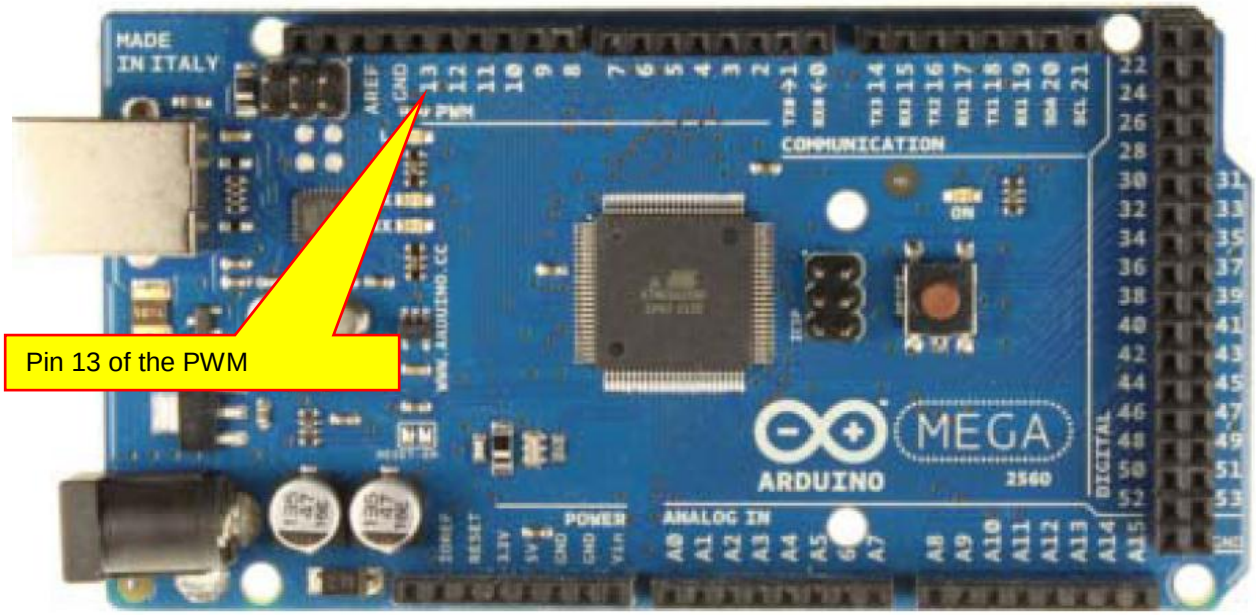
We notice that PWM pins are numbered 2 through 13 above. PWM stands for pulse-width-modulation and these pins can be used for both analog output and digital output. Next, we will look at the Mega Schematic:



This is obviously unreadable in this document, but he schematic is available in the documents provided for this lab. If we zoom in on a portion of the schematic, we can see the LED and determine the I/O pin:



Piecing together the various parts of the schematic, we gather that the LED is connected to pin 13 on the PWM connector:

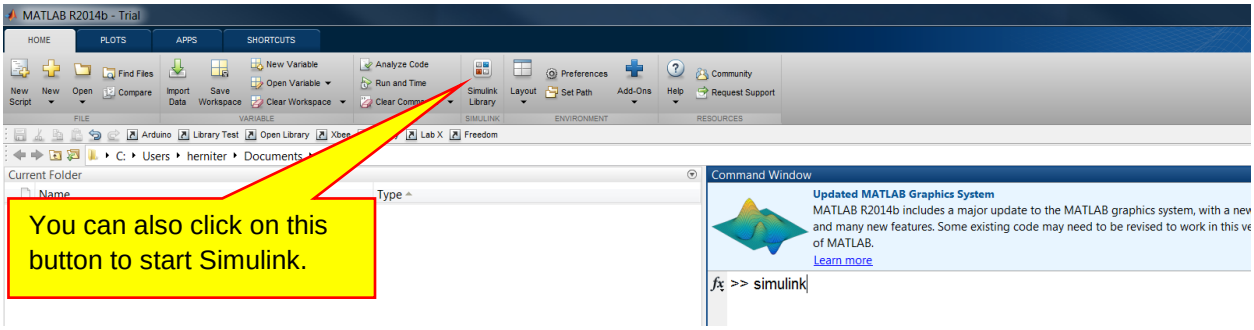


So, when we specify a digital output port to make the yellow on-board LED flash, we will specify pin 13.

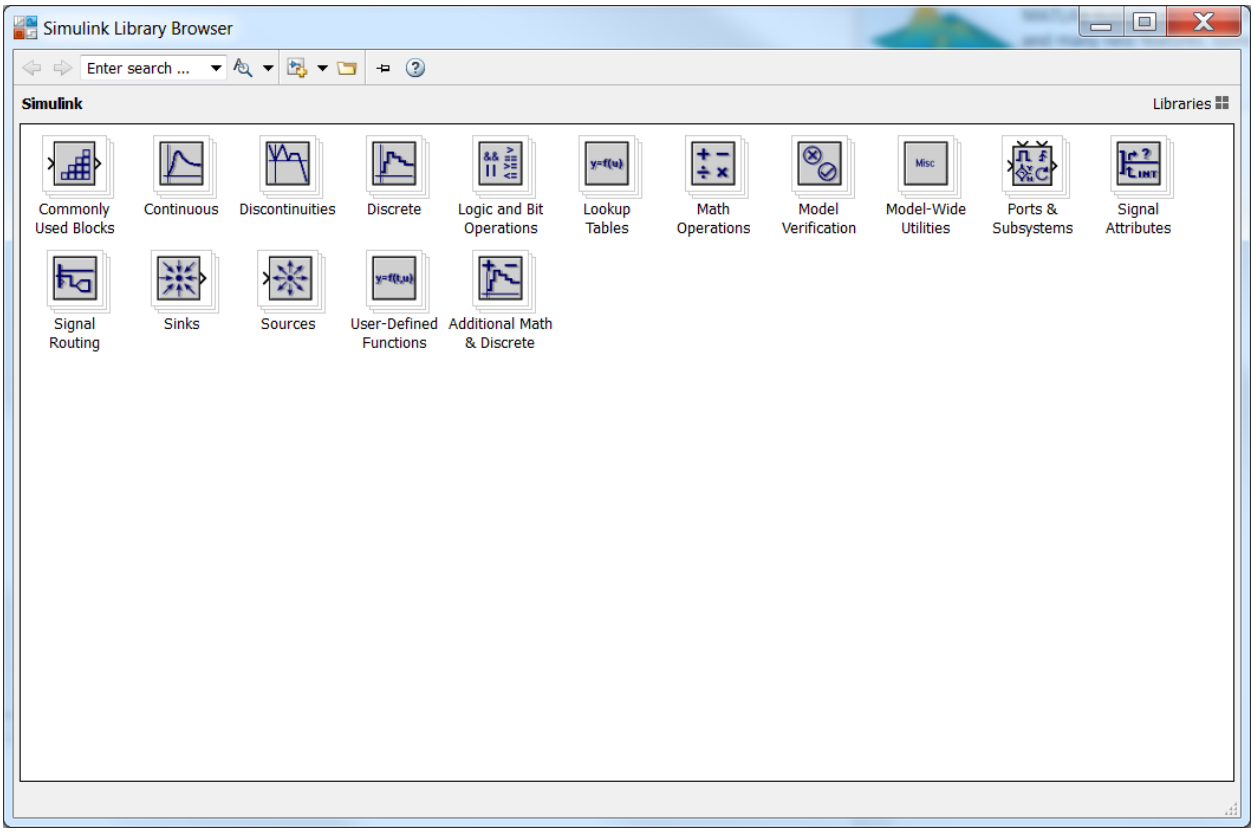
C. Lab Procedure

1. Flip-Flop LEDs

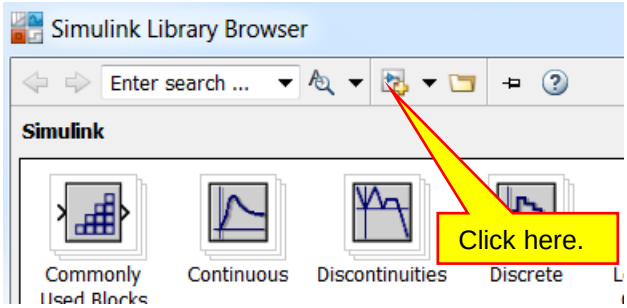
We will now create a simple Simulink model to flash the yellow on-board LED. To start Simulink, type **simulink** at the MATLAB Command prompt:



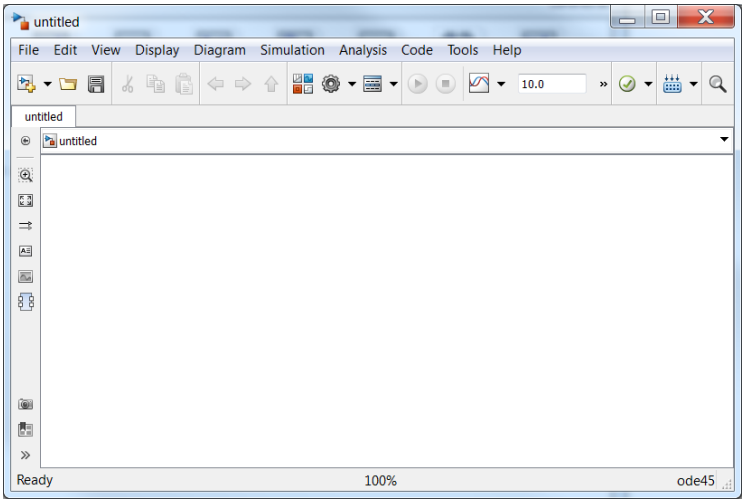
The Simulink library browser will open:



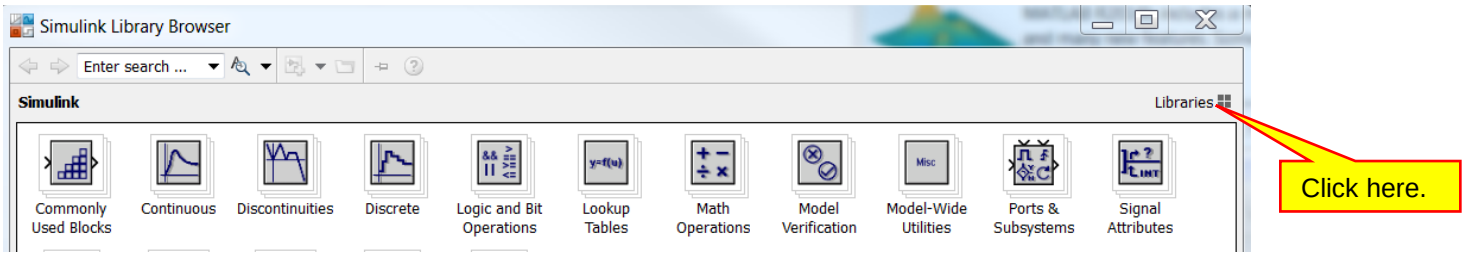
To create a new model, click on the new model button  as shown below:



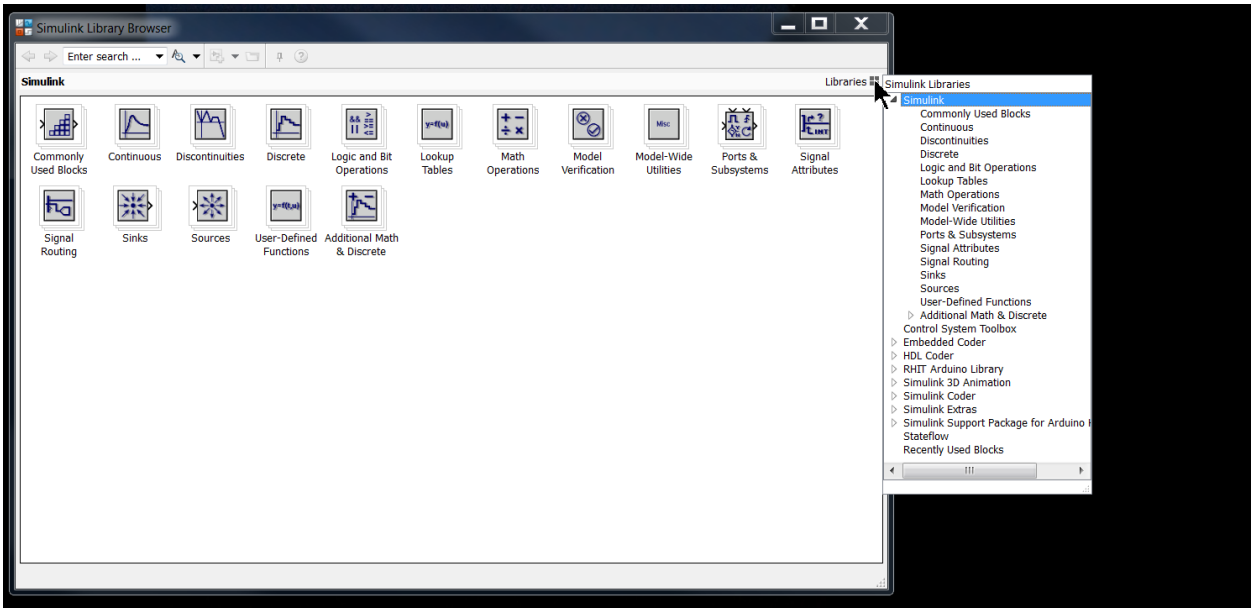
A new and empty model will open:



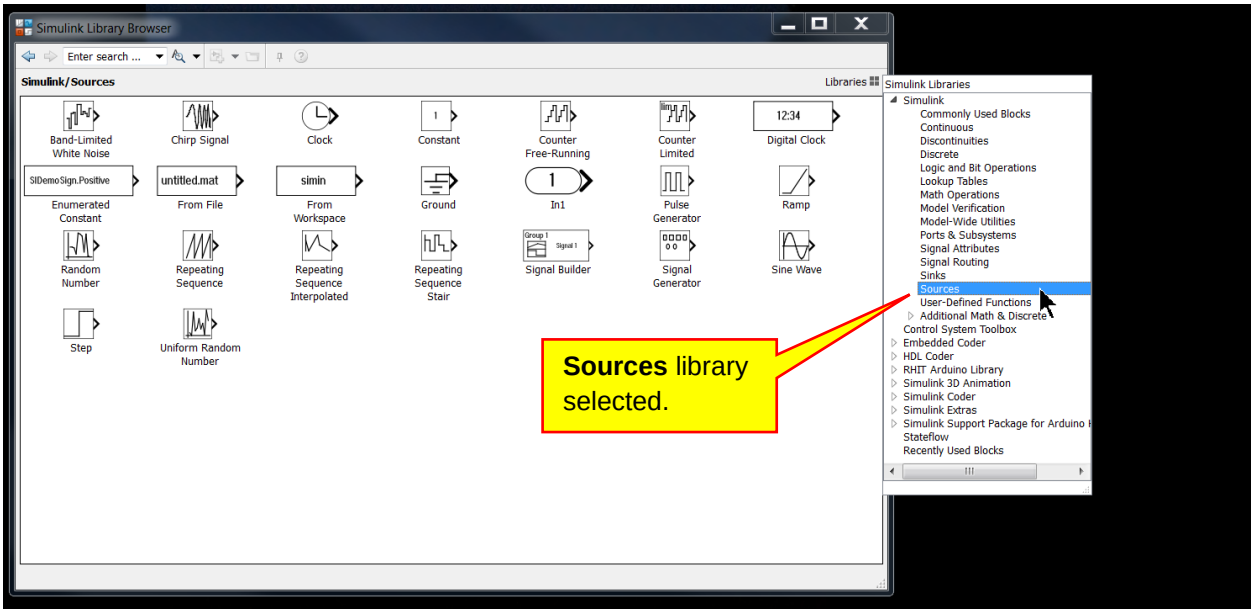
We now need to place a pulse-source in the model. From the library browser, click on Libraries as shown below, to see the list of libraries:



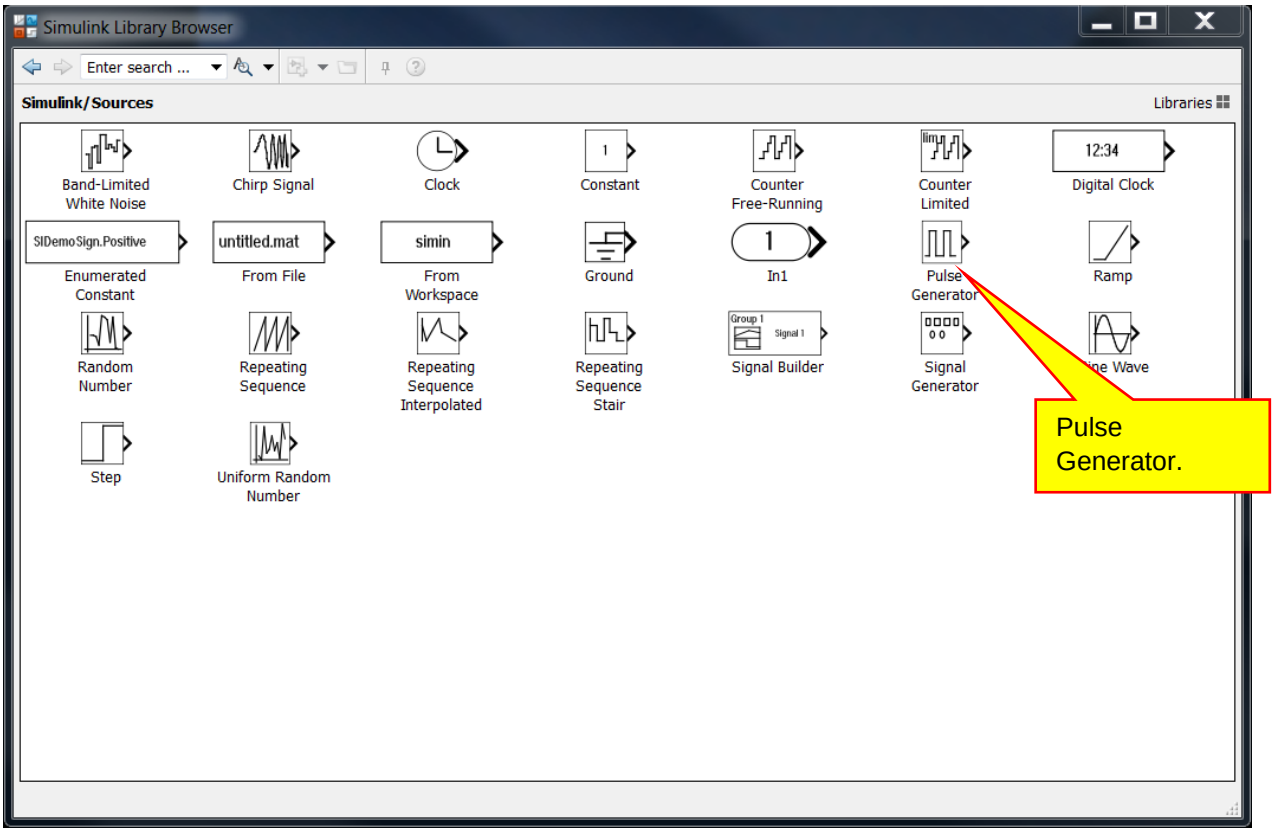
The library menu will appear



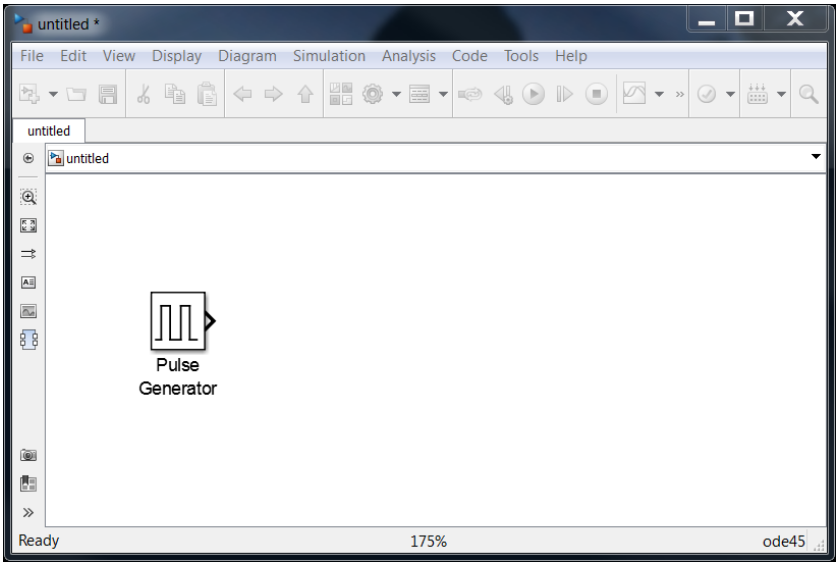
Select the **Simulink/Sources** library as shown below:



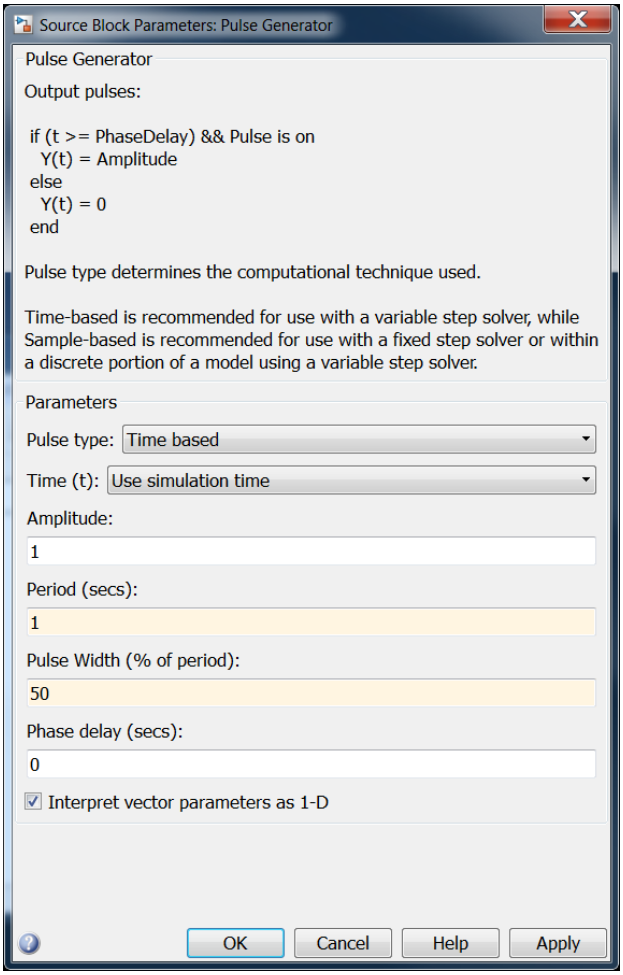
Locate the **Pulse Generator**:



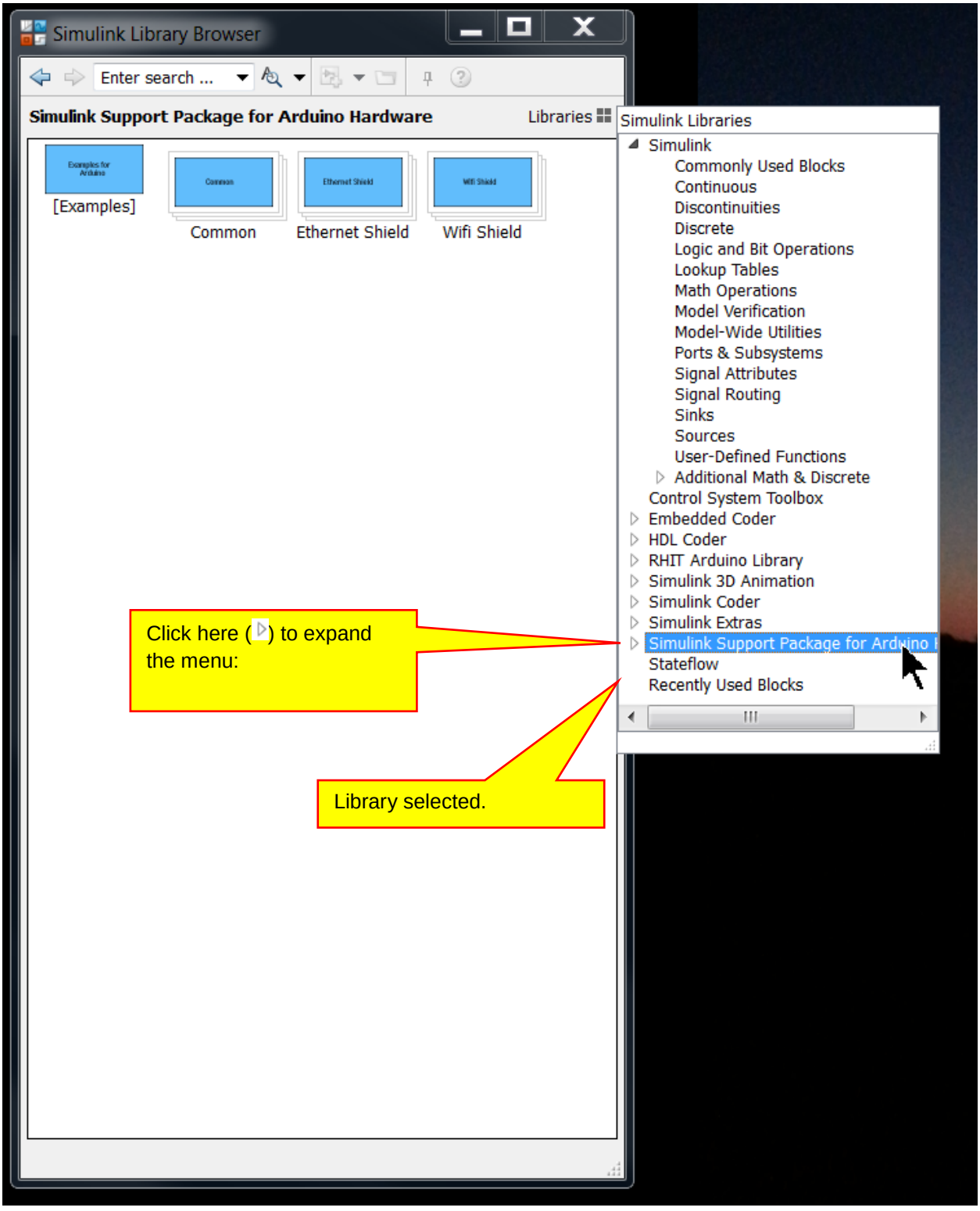
Drag the pulse generator into your model:



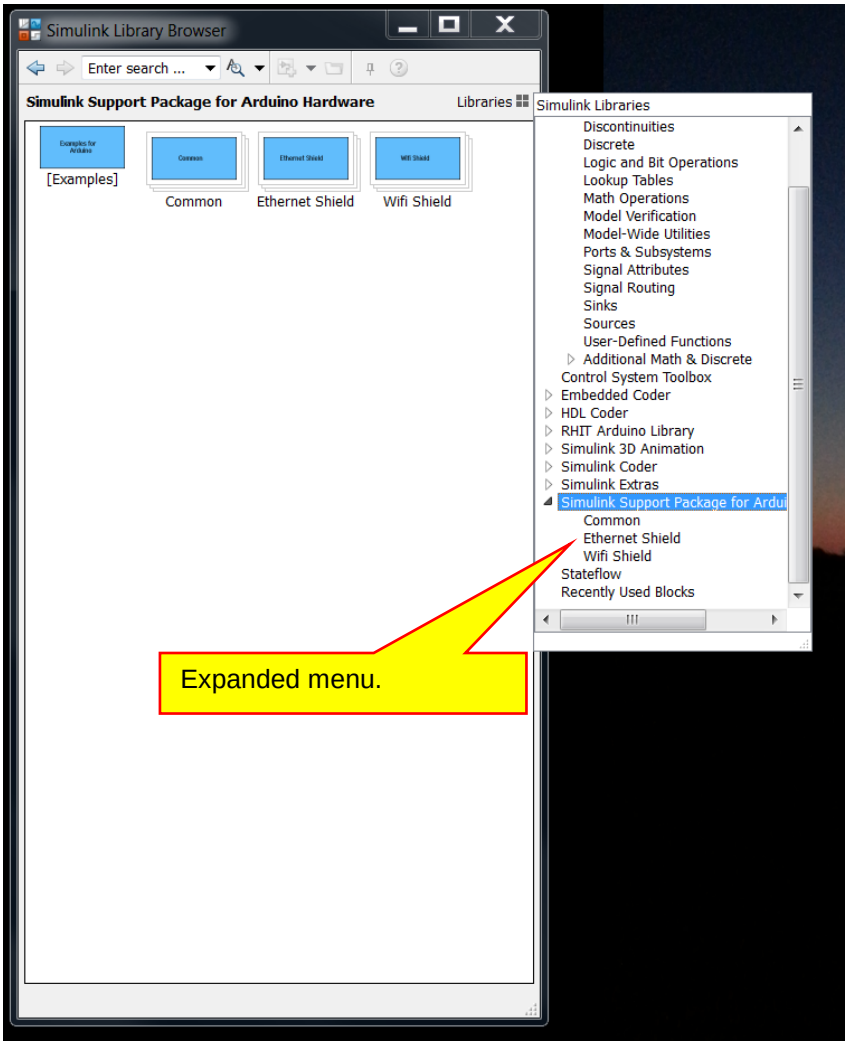
Double-click on the pulse generator parts to edit its attributes. Specify a period of 1 second and a pulse width of 50%. This will cause our LED to flash on and off at a 1 Hz rate with the LED on 50% of the time and off for 50% of the time:



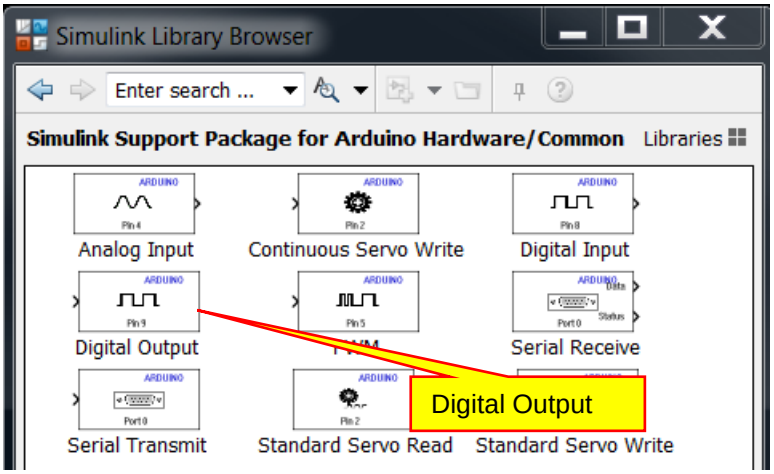
Click the **OK** button to accept the changes. Next, select the **Simulink Support Package for Arduino** library:



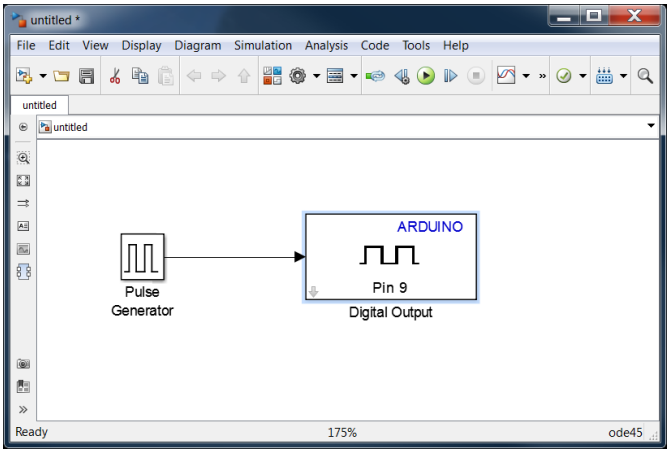
Click on the triangle ( ) as shown above to expand the menu selection:



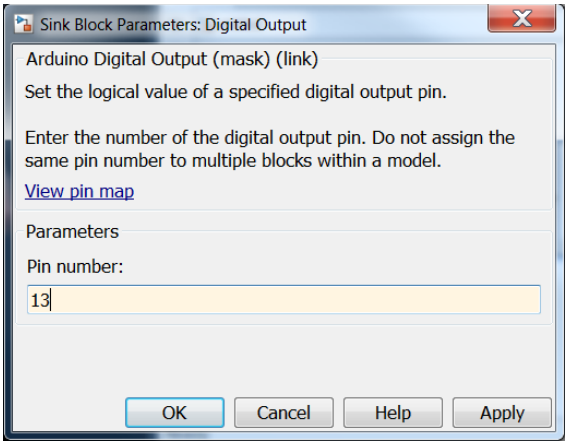
Select the Common Library and locate the **Digital Output** block:



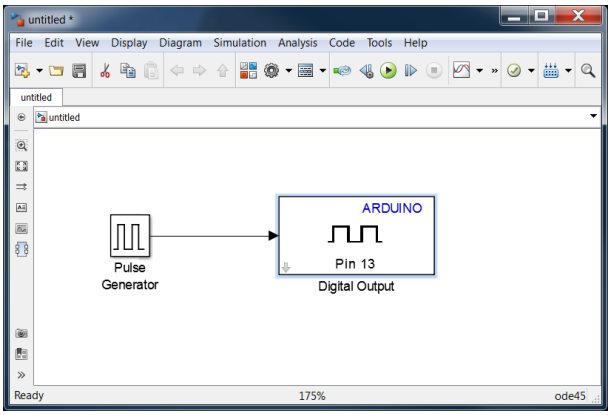
Place a **Digital Output** block in your model. Connect the input of the digital block to the output of the Pulse Generator:



Next, double-click on the **Digital Output** block to open its properties. Change the pin number to **13**:

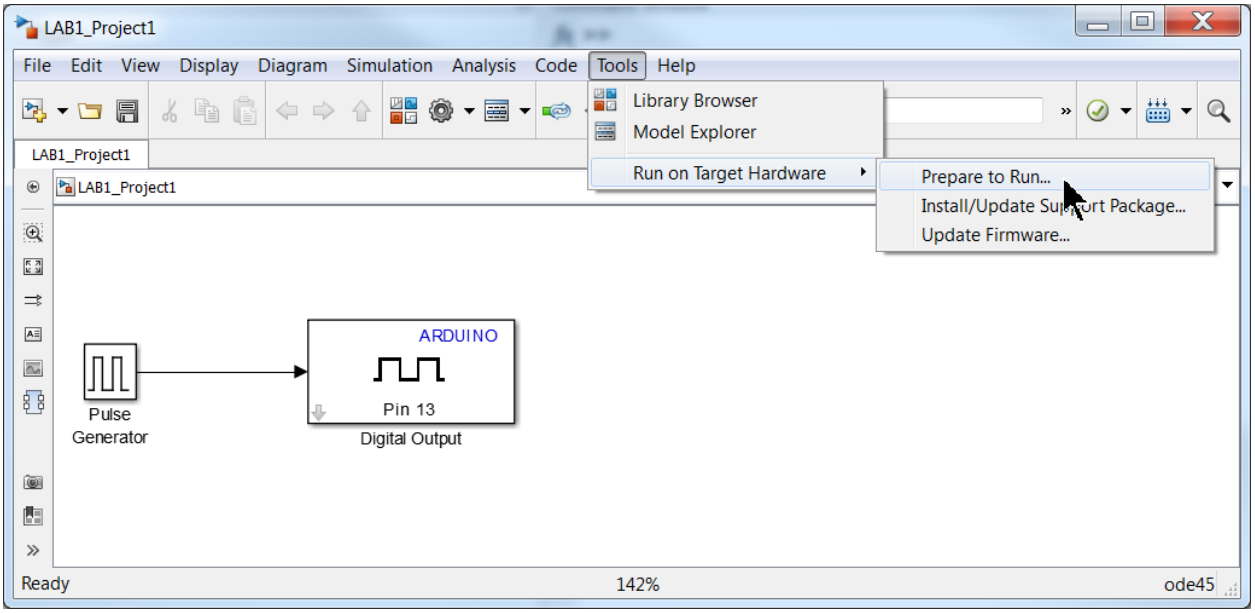


Remember that Pin 13 will drive the on-board LED. Click the **OK** button to accept the changes. Note in the model that Pin 13 will be displayed:

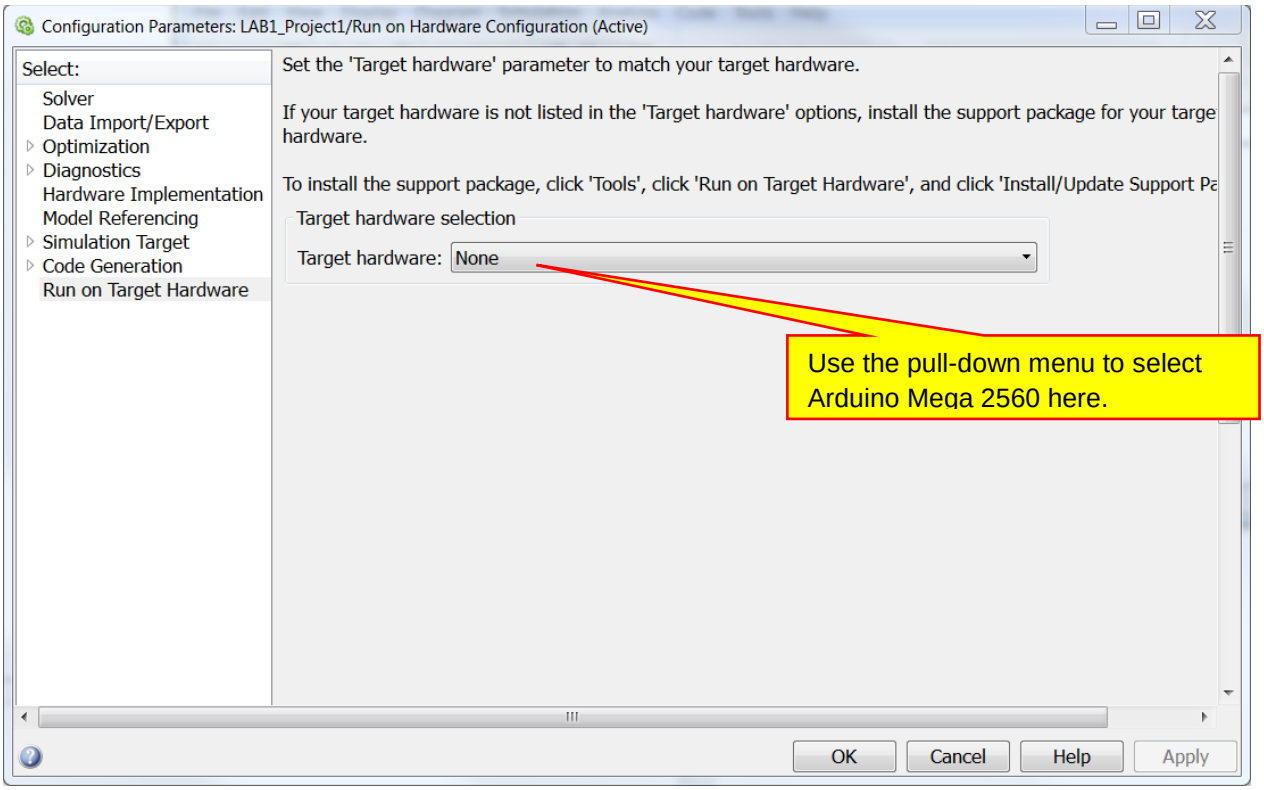


Before continuing, save your model as Lab1_Project1.

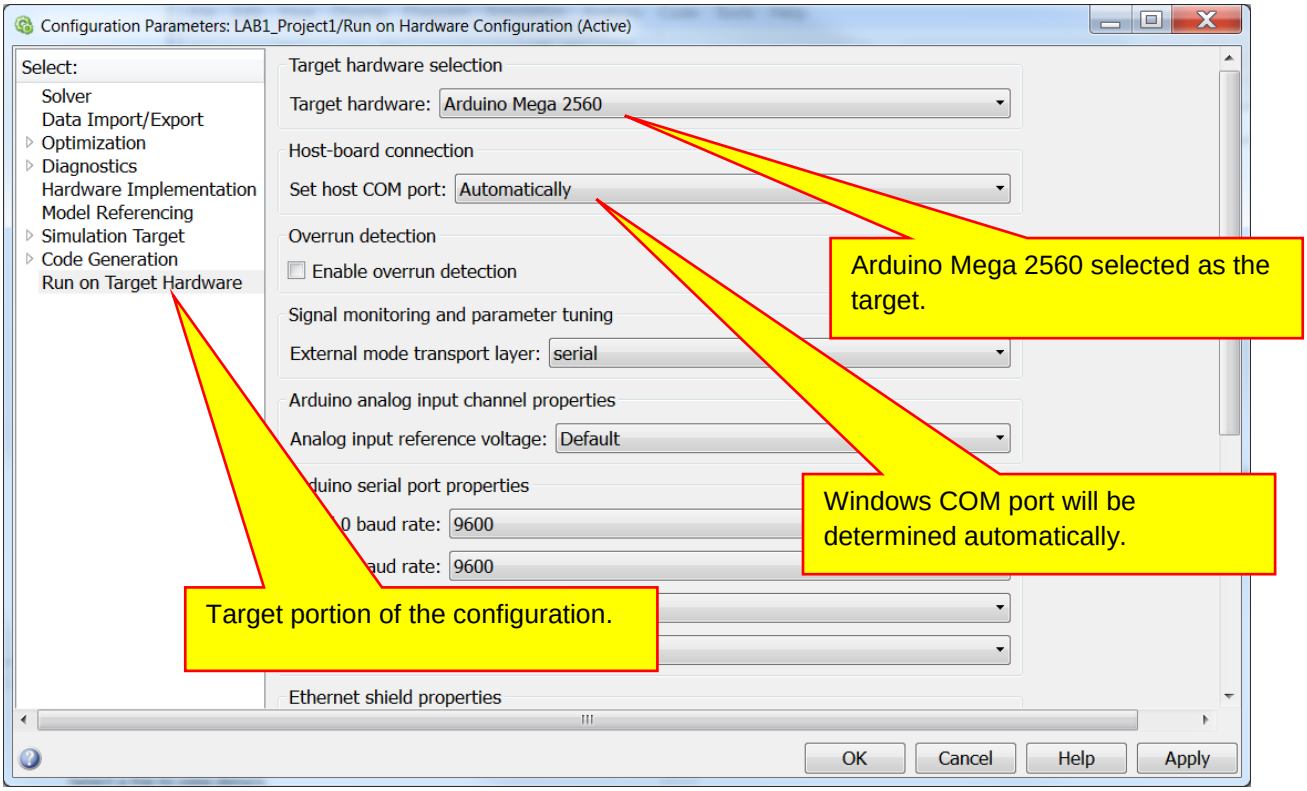
Next, we need to set up the model to run on the Arduino target. From the Simulink menus, select **Tools**, **Run on Target**, and then **Prepare to Run**:



When the **Configuration Parameters** window opens, specify **Arduino Mega 2560** as the **Target hardware**:

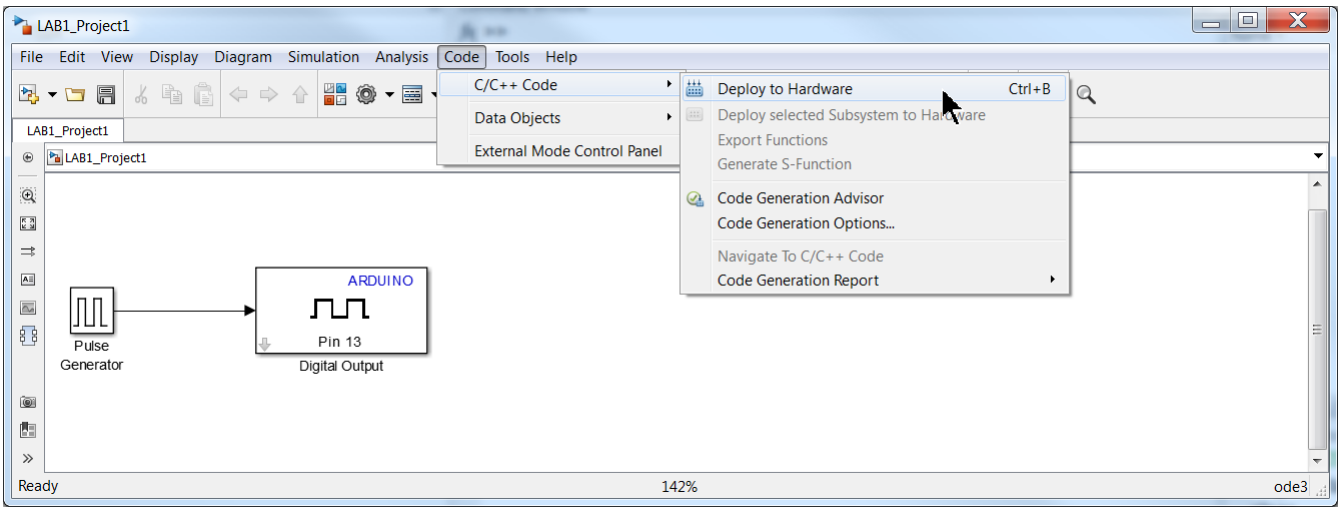


After selecting the target, the window shows you the selected target and that Simulink will determine which COM port on your Windows computer the Arduino Mega is connected to:

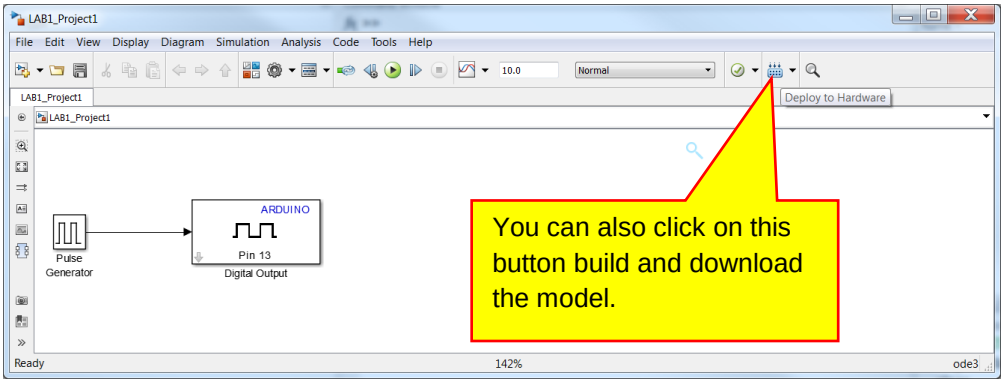


For now, we will not make any more changes. Click the **OK** button to accept the changes. Save your model.

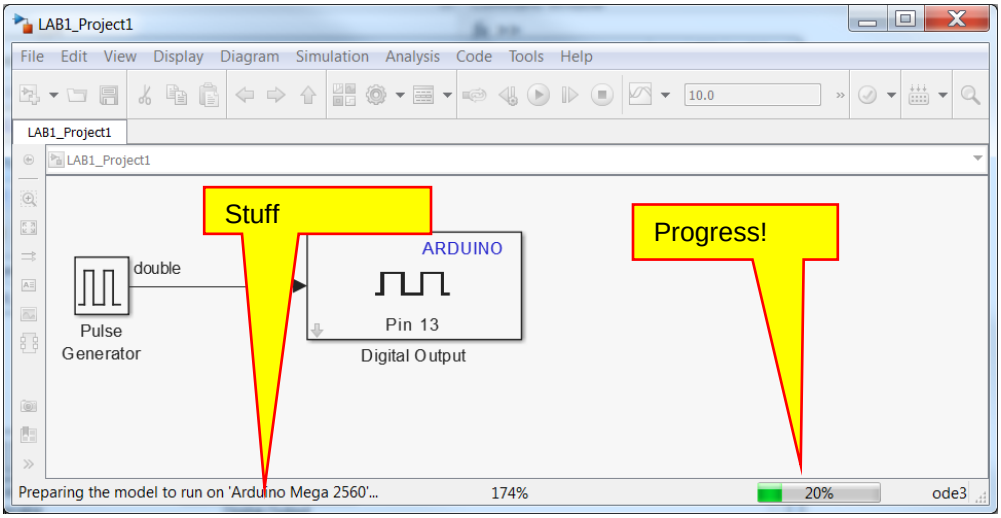
We are now ready to run our first model on the Arduino Microcontroller target. To compile the model, download it to the Arduino Target, and run the model on the target, select **Code**, **C/C++ Code**, and then **Deploy to Hardware**:



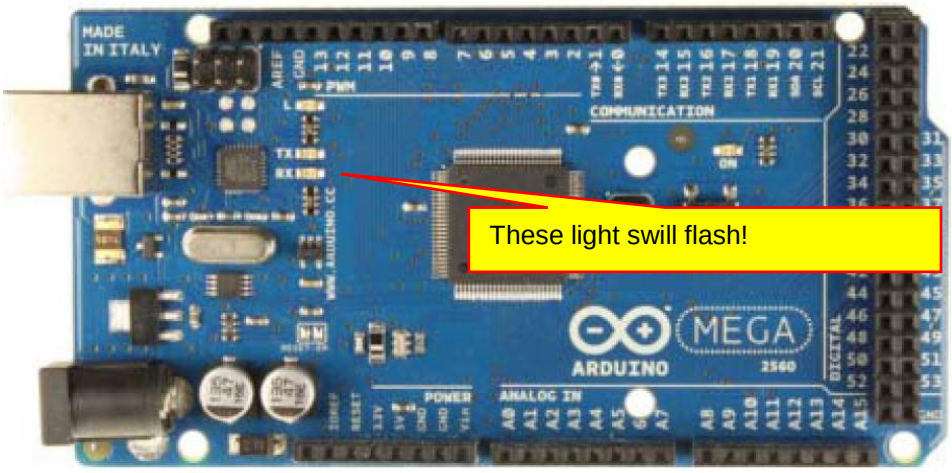
Note that you can also type **CTRL- B** or click on the **Deploy to Hardware** button ( ) shown below:



A bunch of stuff will happen:

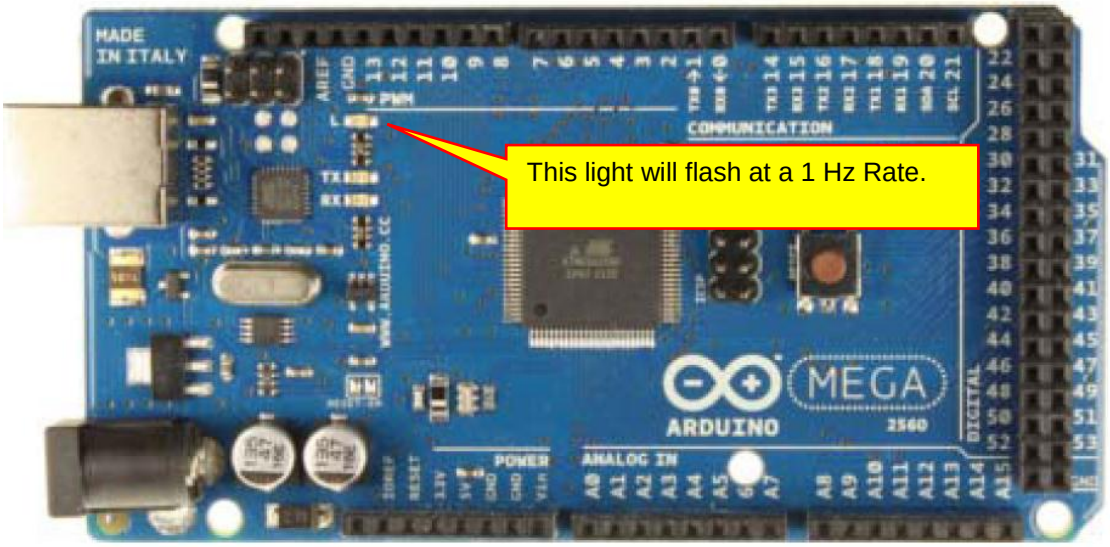


When the build is finished, the Tx and Rx lights on the Arduino Mega will flash as the model is downloaded. The lights indicate that you do have communication with the Arduino Mega.



When the download is complete[1], the **L** light will flash on and off at 1 Hz:

[1] If the download fails, the driver for your Arduino board may not have installed itself automatically. You will need to manually install the device driver for the Arduino Mega.



Note that the model is written to the Arduino flash memory so that if you cycle the power, the model will remain in memory and restart at power up or when you press the reset button.

a) Student Exercises and Demos

Demo I.1: Show the Arduino LED flashing at a 1 Hz Rate.

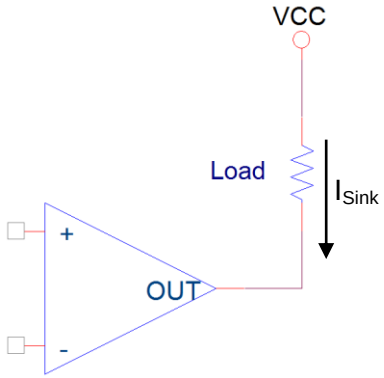
Exercise I.1: Change the model so that the LED flashes at a 2 Hz rate. Demo to your instructor.

Exercise I.2: Change the model so that the LED flashes at a 10 Hz rate. Demo to your instructor.

Exercise I.3: Change the model so that the LED switches back and forth between flashing at a 1 HZ rate and a 2 Hz rate. The rate should change every 5 seconds. Hint: In the Simulink **Commonly Used Blocks** library, use a part called **Switch**.

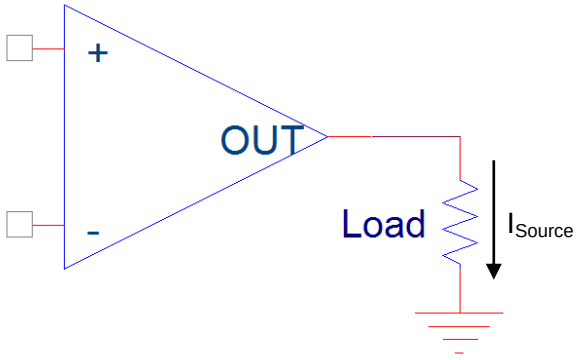
2. Driving a Single Off-Board LED

Next, we will learn how to drive a load directly from the Arduino digital output ports. Whenever you connect a load to the output of an electronic device, you must find out how much current that output can source or sink. When a device sinks current, current flows into the device output pin as shown below. In this example, if the output voltage of the OP-AMP is less than the supply voltage Vcc, current will flow into the OP-AMP output as shown, and the OP-AMP will “sink” current:



Typically, a device will sink current if one side of the load is tied to the positive supply (often referred to as the "positive rail" or the "positive supply rail." In this case, when the output of the device (the OP-AMP shown above) goes low, current will flow into the device.

When a device sources current, current flows out of the device output pin as shown below. In this example, if the output voltage of the OP-AMP is greater than ground (zero volts), current will flow out of the OP-AMP output as shown, and the OP-AMP will "source" current:



Typically, a device will source current if one side of the load is grounded. In this case, when the output of the device (the OP-AMP shown above) goes high, current will flow out of the device. The datasheet for the Arduino states that the "DC Current per I/O pin" is 40 mA:

Summary

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 14 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA

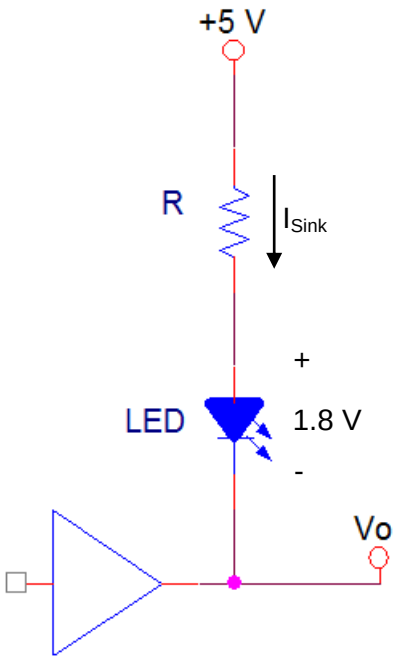
The only problem with this information is that we do not know if that is source or sink current. Further on down in the document, we find a bit more information:

Input and Output

Each of the 54 digital pins on the Mega can be used as an input or output, using pinMode(), digitalWrite(), and digitalRead() functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50 kOhms. In addition, some pins have specialized functions:

This paragraph tells us that the Arduino outputs can source or sink 40 mA of current.

We will now drive an LED from the Arduino digital output. We have two choices, we can turn on the LED when the digital output is low and have the output sink current, or we can turn on the LED when the digital output is high and have the output source current. In both cases, we will design the circuit to have about 10 mA of current flow through the LED. We will also assume that when the LED is on, the voltage drop across it is about 1.8 V. The circuit below turns on the LED when the digital output goes low:

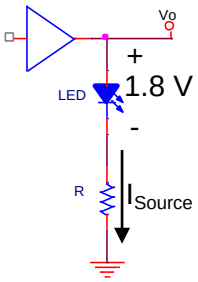


When V_O goes low (0 V), we would like the current, I_{Sink}, to be 10 mA. The equation for the resistor is

$$R = \frac{5\ V - 1.8\ V}{I_{Sink}} = \frac{3.2\ V}{10\ mA} = 320\ \Omega$$

The closest standard 5% resistor is 330 Ω, so we will use that value.

We can also use the circuit below turns on the LED when the digital output goes high (5 V):

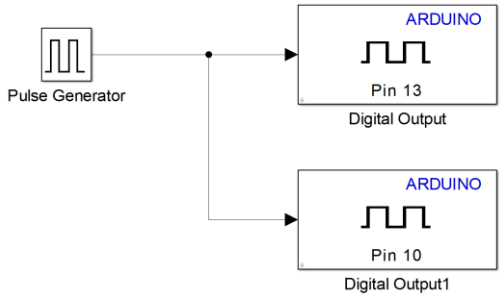


When V_O goes high (5 V), we would like the current, I_{Source}, to be 10 mA. The equation for the resistor is

$$R = \frac{5\ V - 1.8\ V}{I_{Source}} = \frac{3.2\ V}{10\ mA} = 320\ \Omega$$

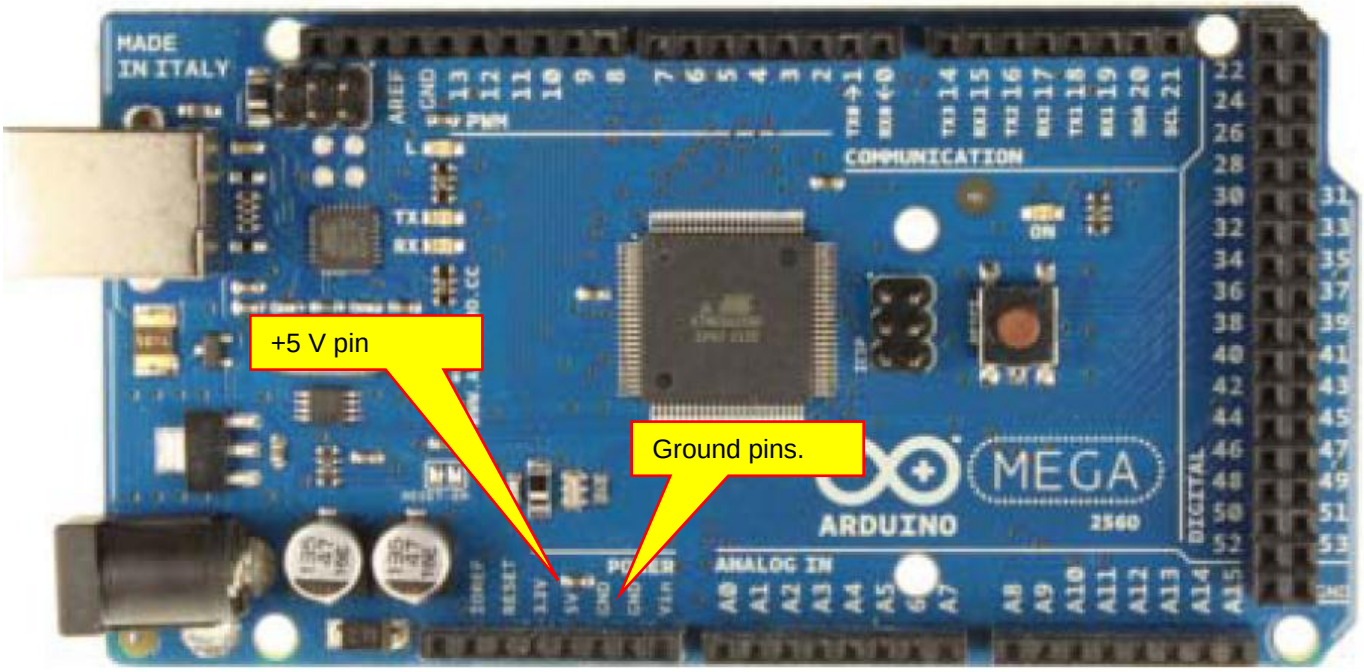
Once again, the closest standard 5% resistor is 330 Ω, so we will use that value. Thus, in both cases, we use the same resistor. Note that in many cases, although the logic power supply is 5 V, the digital outputs do not go to the rail, and V_O is not 5 V, but somewhat less.

We will use the second method show above to drive the LED. This time we will use pin 10 of the Arduino to turn on the LED when the output is high. Modify the Simulink model as shown:

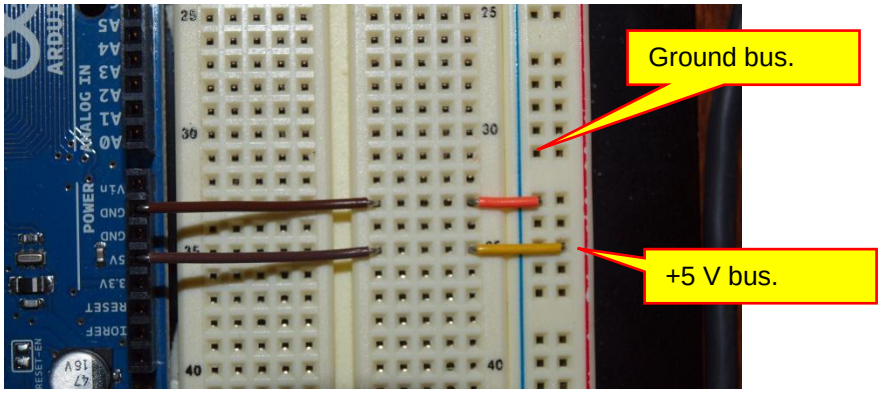


Note that we will still use Pin 13 to drive the onboard LED because we know this works, and if our new LED circuit does not work, this onboard flashing LED will at least tell us that the board is working and running our model.

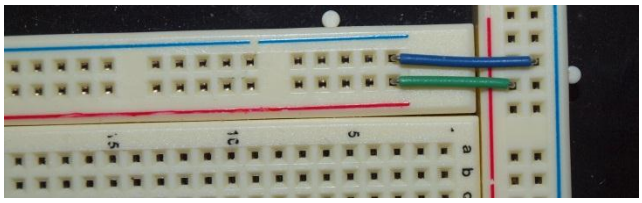
For this and future labs, we are going to need many connections to ground and the + 5V supply. These pins are available on the Arduino board as shown below:

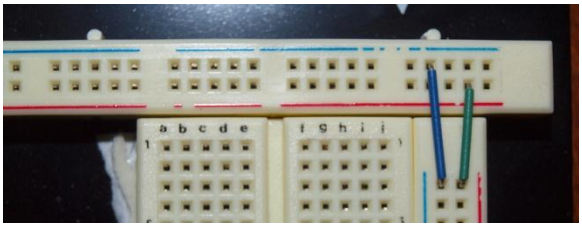


We will route these to the supply busses as shown below:

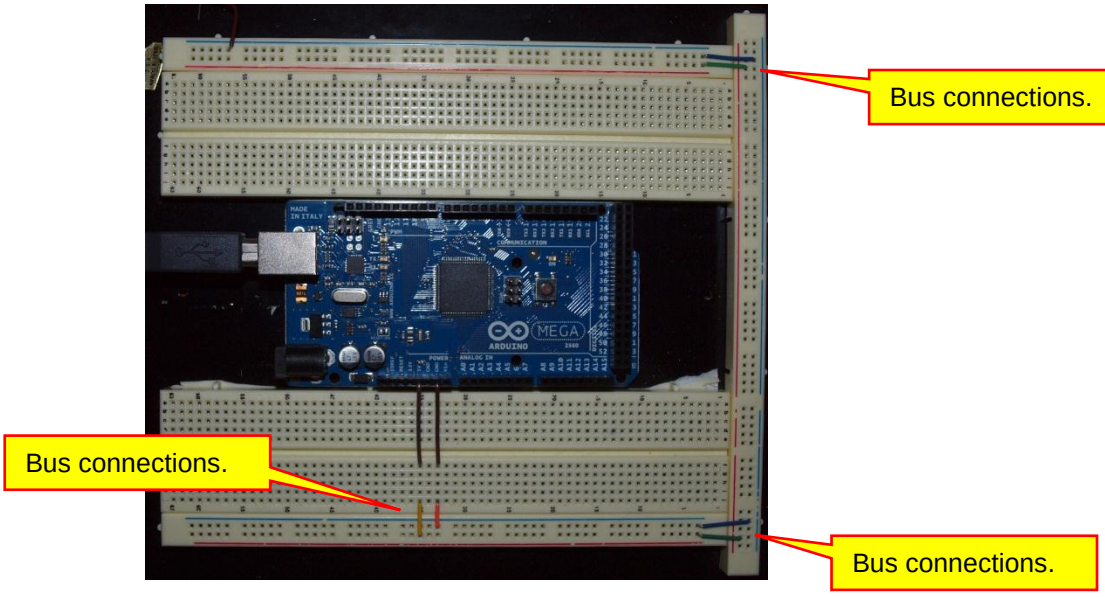


We will use the blue bus as the ground bus and the red bus as the +5 V supply bus. If you have other busses on your board, you can connect them as shown:





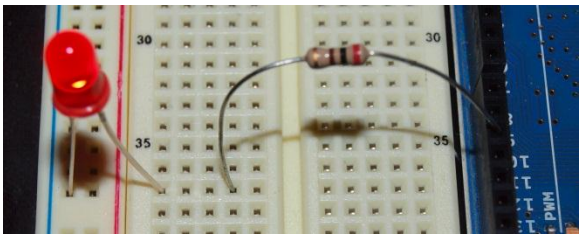
Your entire board should look as shown:



On the board shown, the blue bus is the ground bus and the red bus is the +5 V supply bus, and both busses make a horse shoe pattern around the board.

The last thing we need to do is to wire up the LED circuit. Note that the longer lead on the LED is the positive lead of the Anode:

Wire the circuit on your board as shown below (or something close):



When your model is complete and you have wired your circuit, connect your Arduino to your computer with the USB cable. Click the **Build** button  to build your model and then download and run it on the Arduino target. If all goes well, your LED should Flash on and off at the same rate as the onboard LED.

Demo I.2: Show the off-board LED flashing at a 1 Hz rate.

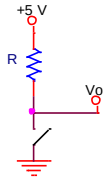
Lab II
Counters, Digital Input, Stateflow, Truth Tables

In this lab we will learn how to use the digital input block to read a 1-bit input, and also look at the many different facilities available in Simulink for implementing both simple and complex logic functions. We will illustrate the facilities by creating a few simple counters, and then adding functionality to those counters.

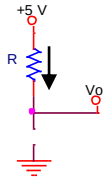
A. Digital Input Block

The Arduino Mega has 54 pins that can be used as digital inputs (all of these pins can be used for digital outputs, pins 2-13 can be used for PWM/analog output, and pins 0-1, can also be used for serial transmit and receive). Typically, when the input voltage to a digital input is between 0 and 0.8 Volts, the block will output a logic "low" or "false.". (For the Arduino Mega, it will output a numerical value of zero.) When the input voltage to a digital input is between ~2 and 5 Volts, the block will output a logic "high" or "true.". (For the Arduino Mega, it will output a numerical value of one.) The voltage ranges are "typical" and depend on the logic family. The logic family is the type of electronic circuitry used to realize the function.

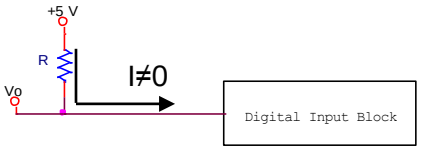
We are going to use the digital input to read the input of a switch. To save cost, a single-pole-single-throw (SPST) switch will be used. SPST is a fancy way of saying that the switch is either "open" or "closed." "open" is short for an open circuit where no current flows – basically the switch connects something to something else and the connection is not made. "Closed" means a connection is made. We will use the circuit below to allow a single SPST switch to output two voltage levels:



When the switch is open, we have the circuit below:

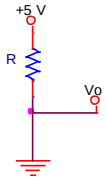


The output voltage is 5 V – I·R. Since the current is zero, I·R is zero, and V_O = 5 V. V_O is said to be "pulled-up" to 5 V, and the resistor in this circuit is called a "pull-up" resistor. When used in an actual circuit, V_O is connected to the digital input as shown below:

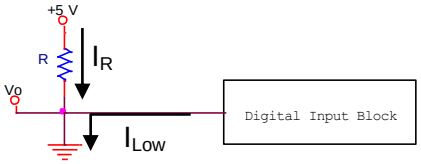


When the input to a digital block is “high” a small amount of current usually flows into the input of that block as shown. The amount of current depends on the implementation of that block, but can be anywhere between zero and 40 µA. This small current will create a voltage drop across the pull-up resistor and result in V_O dropping below 5 V. If the resistor is chosen too large, V_O may drop so low that it is no longer considered a logic “high” input to the digital block. This would obviously be a problem, as the reason for using a pull-up resistor was to pull VO up to a high voltage when the switch was open.

When the switch is closed, we have the circuit shown below:



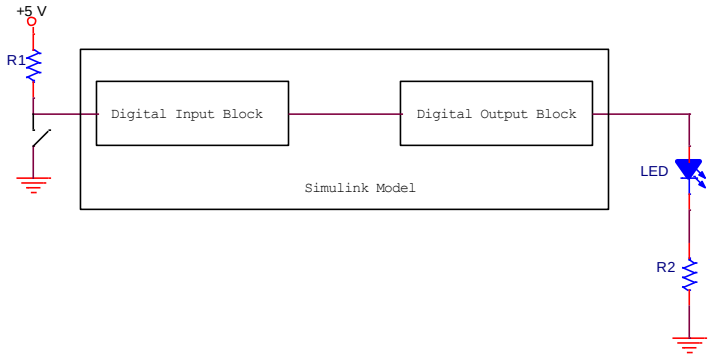
VO is directly tied to ground, so VO is zero volts, a good “low” logic input. When connected to a digital block, we have the circuit shown below:



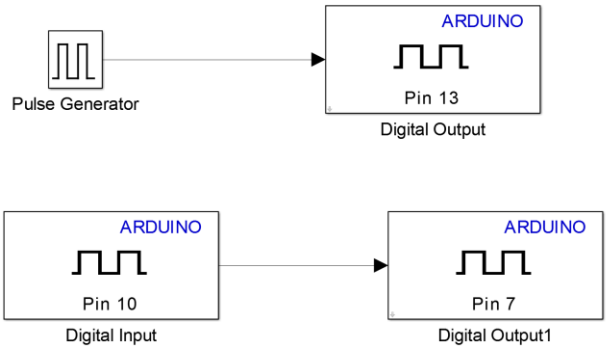
When the input to some digital blocks is tied low, as shown in the figure, current will flow out of the input. In our case, I_{Low} flows directly to ground through the switch and does not affect the voltage V_O. For this circuit, neither the value of the resistor nor the current I_{Low} affect the input voltage to the digital block, V_O. The only effect the choice of the resistor has is the current through the resistor when the switch is closed. This is a power draw from the +5 V supply. The smaller the resistor, the more power it draws from the supply.

Thus, we see that the choice of the pull-up resistor is a trade-off. We want the resistor to be small when the switch is open so that we get a good logic “high” input to the digital block. When the switch is closed, we want a large value for the resistor so that it does not draw too much current from the supply. A good compromise for the value of a pull-up resistor is 10 kΩ.

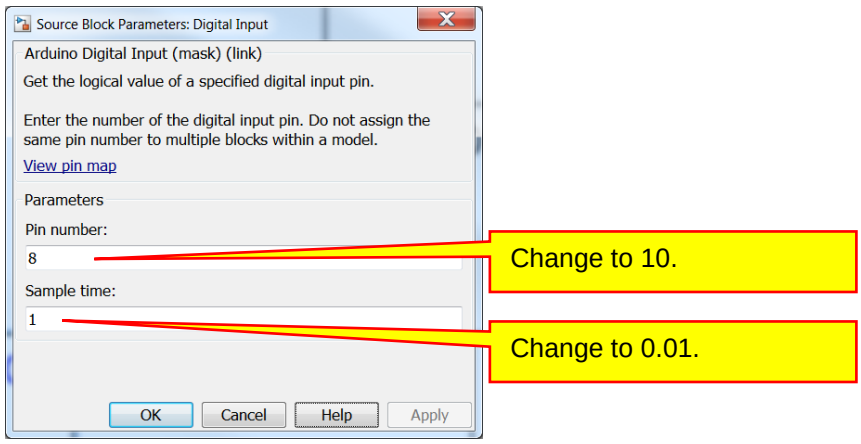
We will no create the circuit/model below:



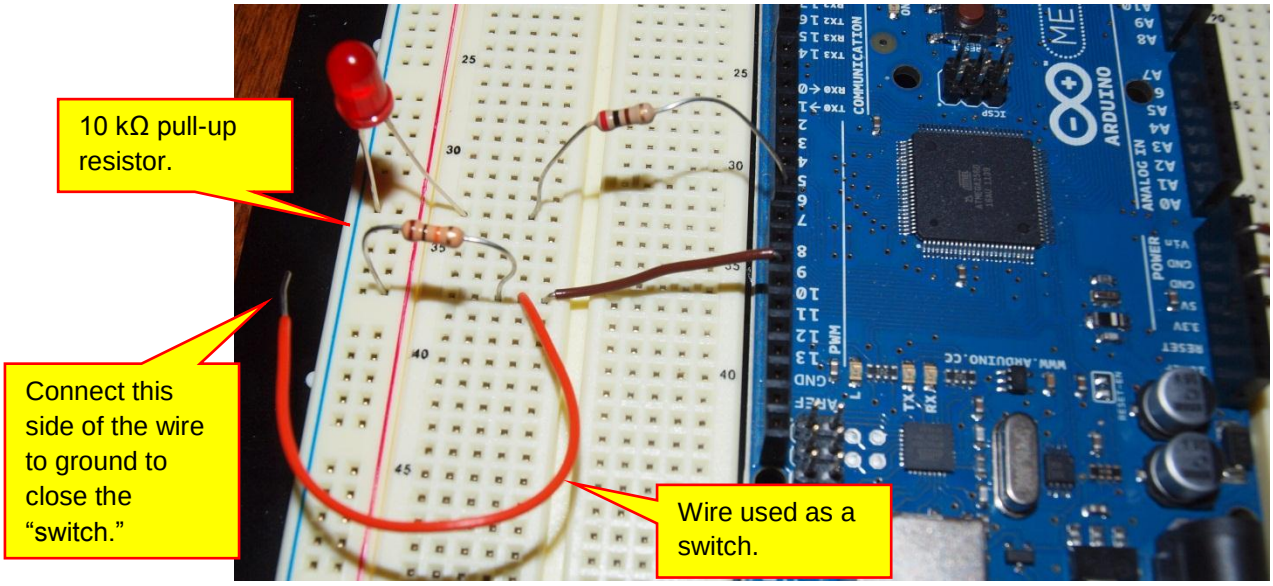
All we are doing is reading a single digital input, and passing that information directly to a digital output block. When the switch is closed, the LED should turn off. When the switch is open, the LED should turn on. (We don't need a microcontroller for this, but hey, why not make things as complicated as possible.) We will keep the flashing on-board LED's from Lab 1 in all of our models so that we know the controller is working. The complete model is shown below:



We will use pin 10 as the input and pin 7 as the output. When you double-click on the **Digital Input** block, you will notice a **Sample time** parameter:



This parameter specifies how often the block reads a new input from the pin. The default is a sample every second, which is very slow. Change the parameter so that a sample is taken every 10 ms. Next, wire up the circuit on your breadboard. Note that you can use a piece of wire as a switch if you do not have a real switch:



Demo II.1: Show the operation of this circuit to your instructor. When the switch is closed the LED should be off. When the switch is open, the LED should be on.

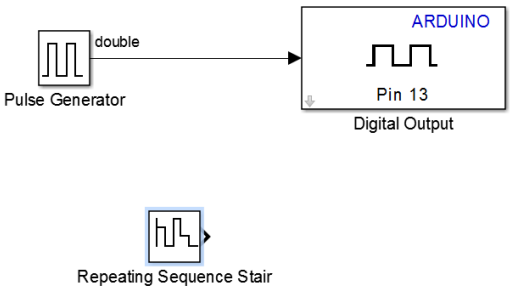
B. Counters with Basic Logic Blocks

1. Ring Counter

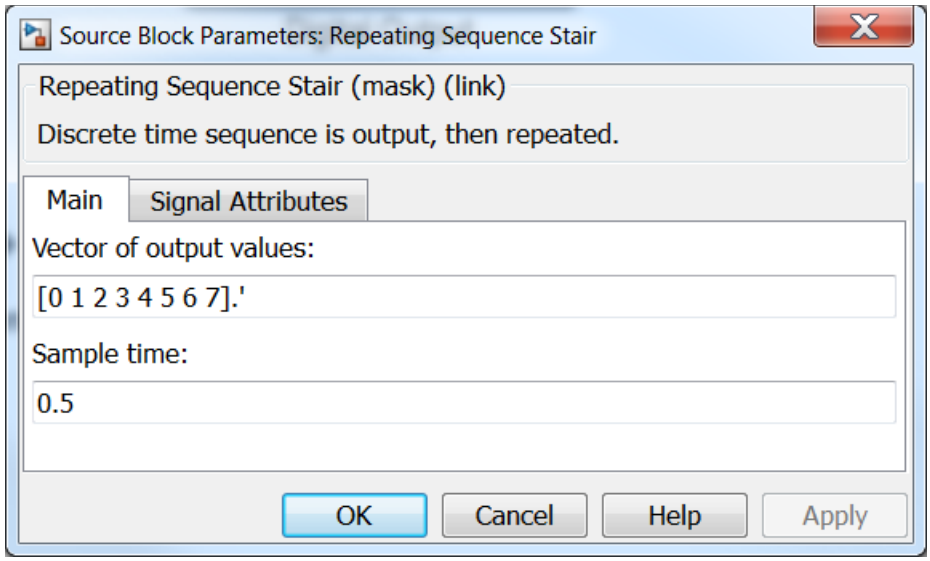
We will now create an 8-bit ring counter that lights up 8 LEDs with the sequence shown below:

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

Only one LED is illuminated at a time, and the illuminated light shifts from left to right. We would like this ring counter to change states every half second. This counter has eight different states, thus, we need a Simulink counter that has eight different values. We will use the Repeating Sequence Stair block Simulink block found in the **Commonly Used Blocks** / **Sources** library:

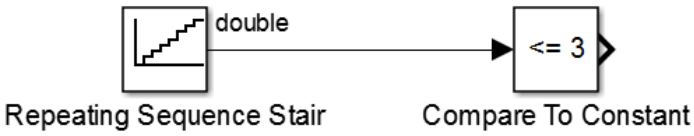


Fill in the properties of the Repeating Sequence Stair block as shown:

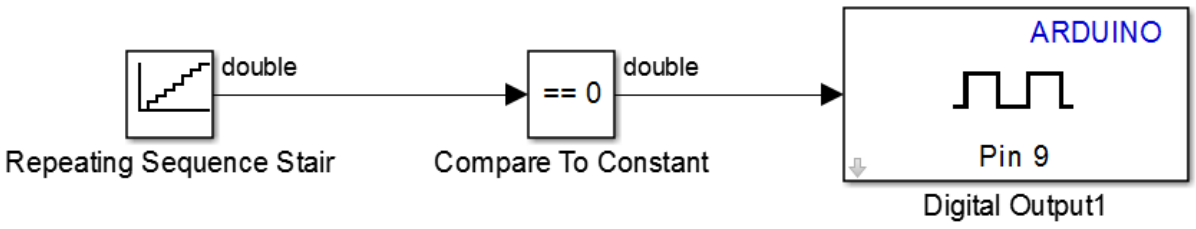


We actually could have used any sequence of numbers as long as the numbers in the vector were unique. (You will see why in the next step.)

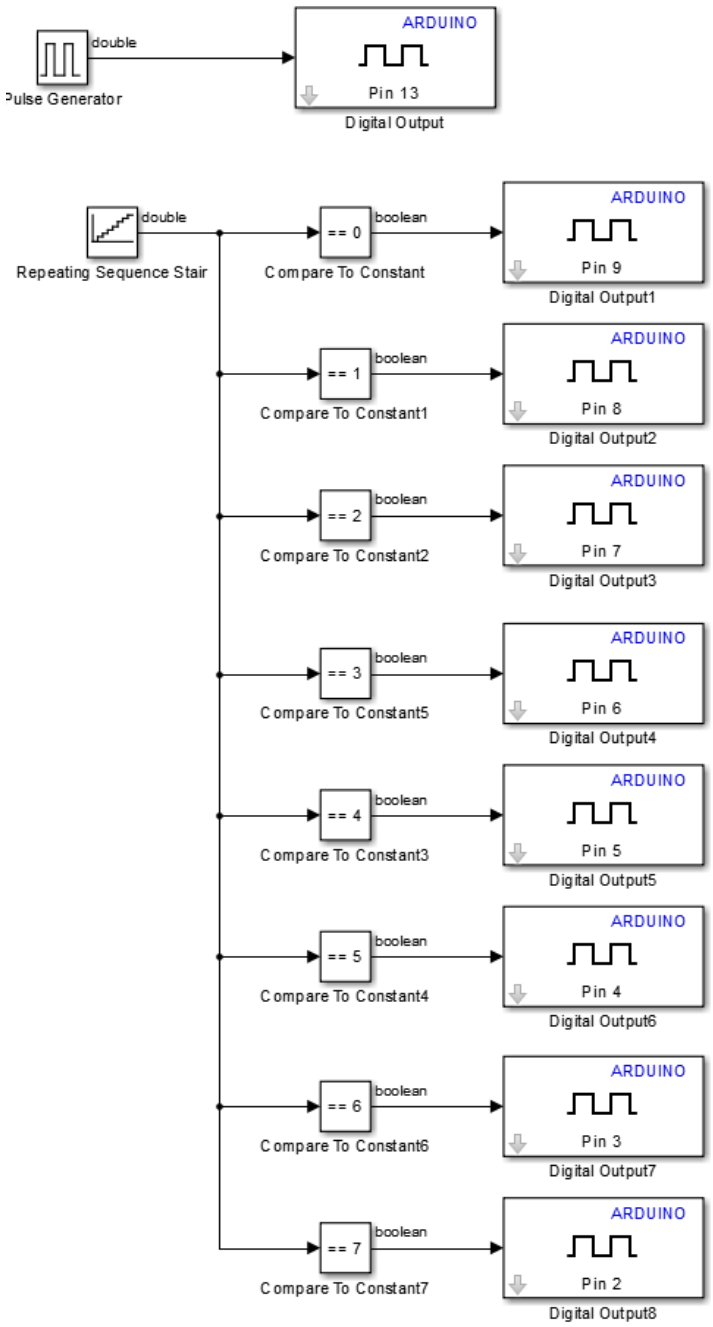
Next, we will create a simple decoder using the **Compare to Constant** block located in the **Simulink** / **Logic and Bit Operations** library:



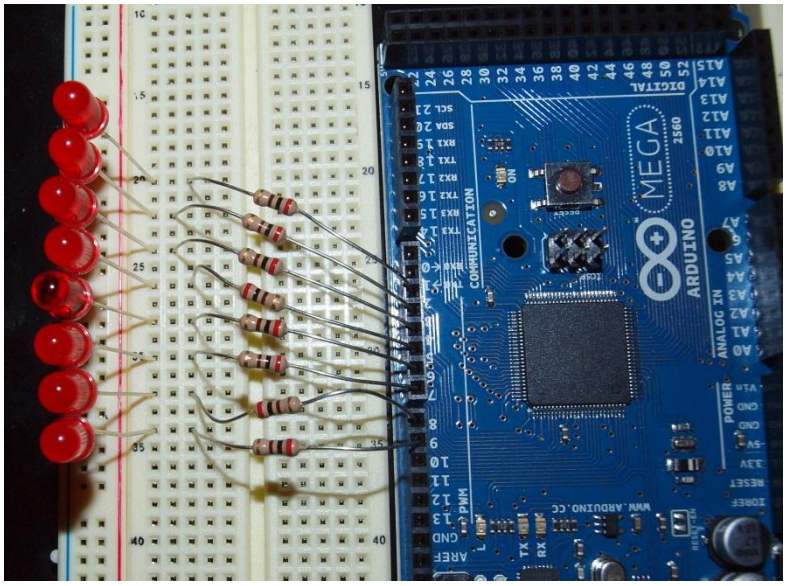
Double-click on the **Compare to Constant** block and change its properties to compare to zero, and add a digital output block:



Note that the output data type for the **Compare to Constant** block was type Boolean. This is an acceptable data type for the **Digital Output** block, so we can just connect the two. Complete the circuit for the 7 other states as shown:



We will use this arrangement of LEDs for the remainder of the lab.



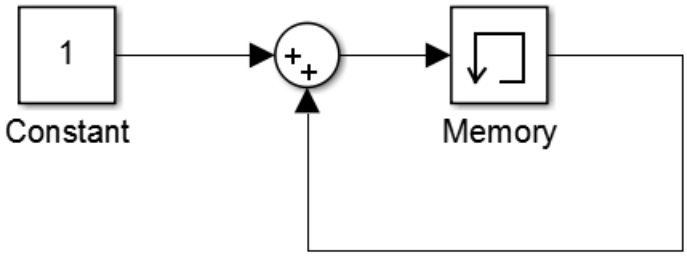
Demo II.2: Wire up 8 LEDs to your chosen pins and demo the ring counter.

Exercise II.1: Create a model that illuminates the LED's in the sequence shown below:

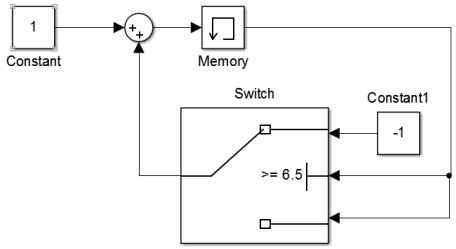
Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0

2. The Digital Memory Block

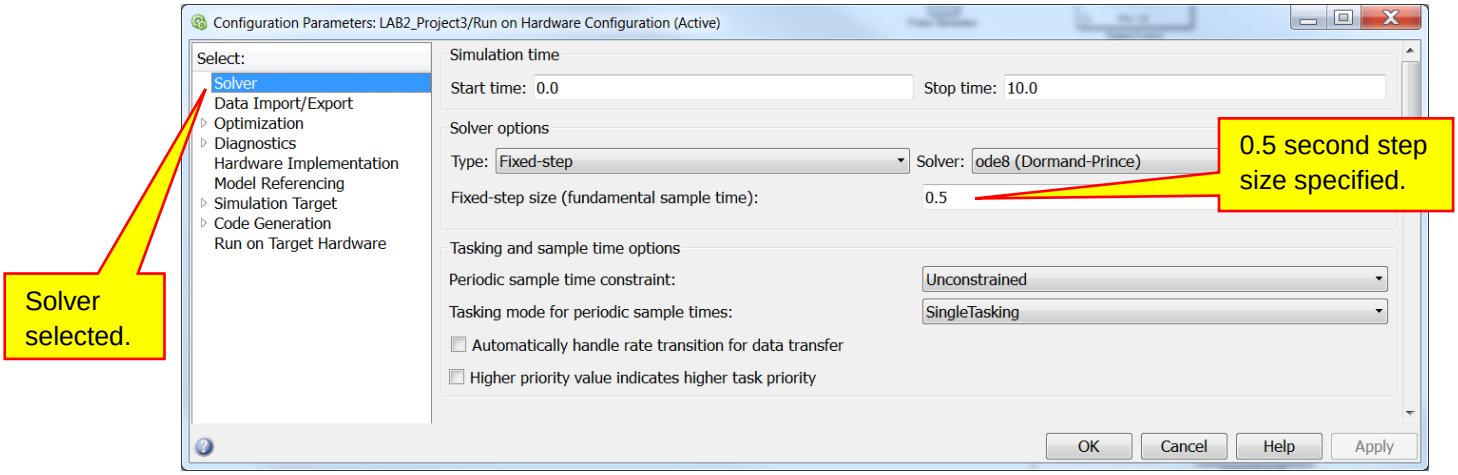
We will now create our own counter. Implementing a counter requires a memory block. We will use the Memory block located in the **Simulink** / **Discrete** library. The output of this block is the input from the previous time step. We can implement a counter using the Sum, Memory, and Constant blocks as shown:



The counter will start at zero and count up forever. The counter starts at zero because the default initial condition for the Memory block is zero. To reset the counter to zero, we need to use a switch to change the feedback to a -1 when the counter reaches 7. The Switch lock is located in the **Simulink** / **Commonly Used Blocks** library:



This counter will count up every time the model executes, which could be fast depending on the time step. To control this, we have to specify the time step for the model. Select **Simulation** and then **Model Configuration Parameters** from the Simulink menus (or type CTRL-E or click the **Model Configuration Parameters** button), and specify the **Fixed-step size** as 0.5 secnds:



Run the model. You may have to fix a few issues with the Pulse generator that we created by changing the fixed time step. The advantage to using a counter of this type is that we can have full control over the operation of the counter, such as pausing the counter or having it count up or down on command.

Demo II.3: Demo the ring counter using a counter implemented with the Memory block.

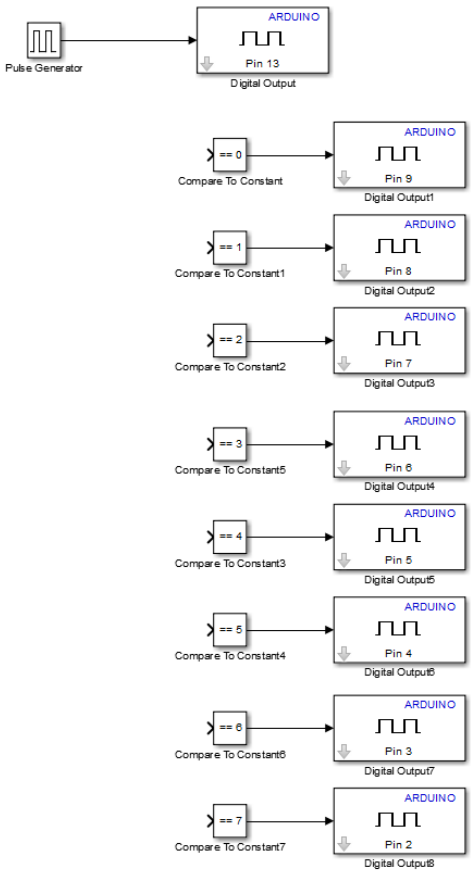
Exercise II.2: Using a digital input, modify the previous model so that the counter holds its value (stops counting) when the switch is closed.

Exercise II.3: Using a digital input, modify the ring counter model so that the counter counts up when the switch is open and counts down when the switch is closed. The counter is not allowed to jump or skip a count when the switch is pressed.

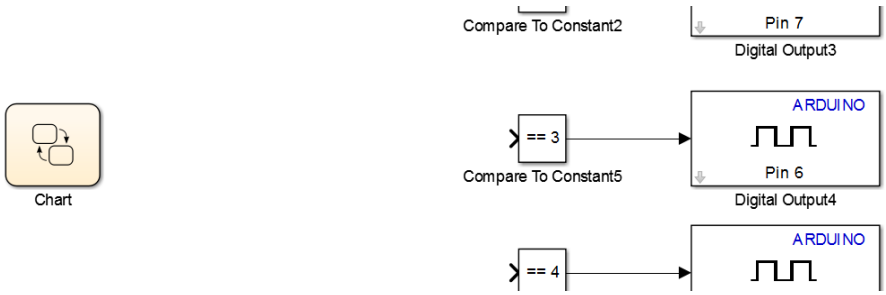
C. Using a Stateflow Chart to Implement State Diagrams

A Stateflow chart is a programming facility that allows us to graphically set up a state diagram. A state diagram has memory (the states) and allows us to use rules of logic to transition between those states. With memory, we know where we have been. With logic transitions, we can determine where we are going. With Stateflow we can create logic based on history or a sequence of events, and use the rules of logic to determine the next state.

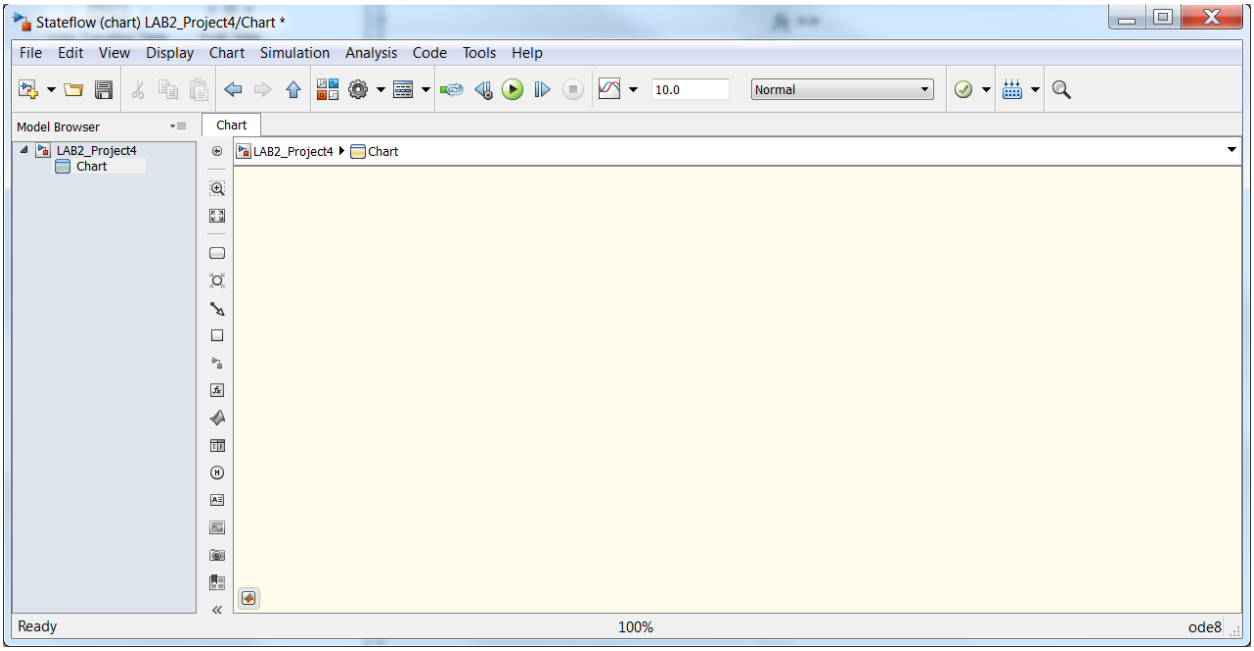
We will start by creating a simple counter. We will start with the previous project, and delete the counter we created with the **Memory** block:



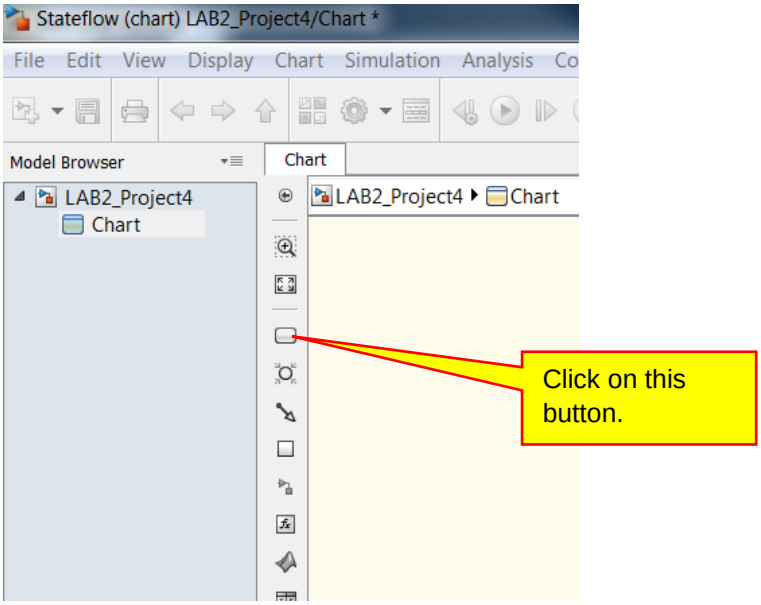
Next, add a **Chart** from the **Stateflow** library to your model:



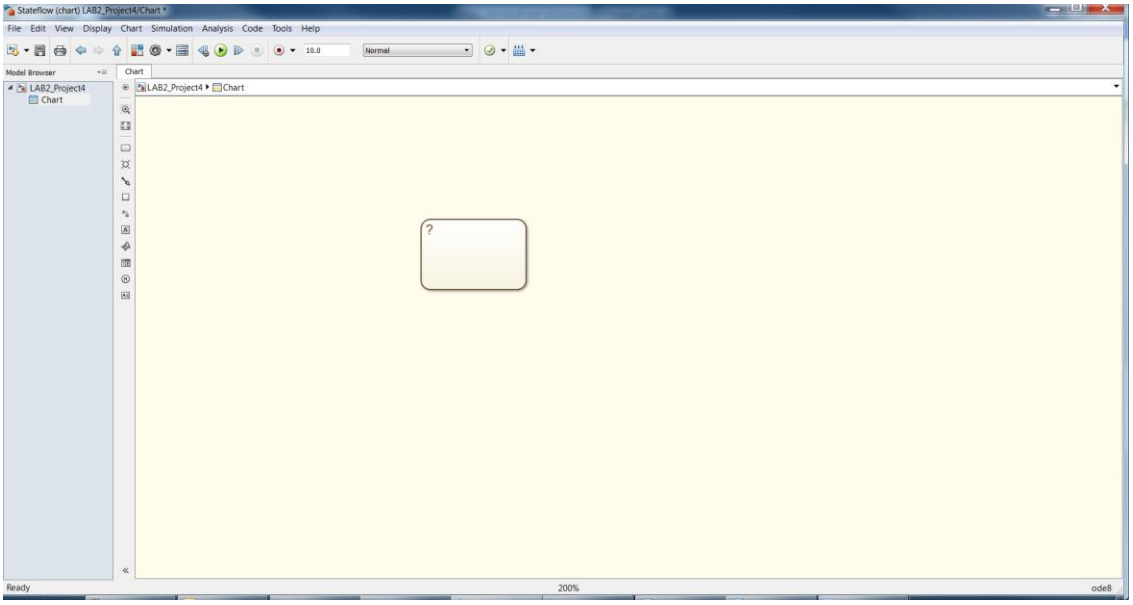
Double-click on the chart to open it. You will have an empty Stateflow chart as shown:



Next, drag a state into the chart. To do this, click the left mouse button on the State button  as shown:

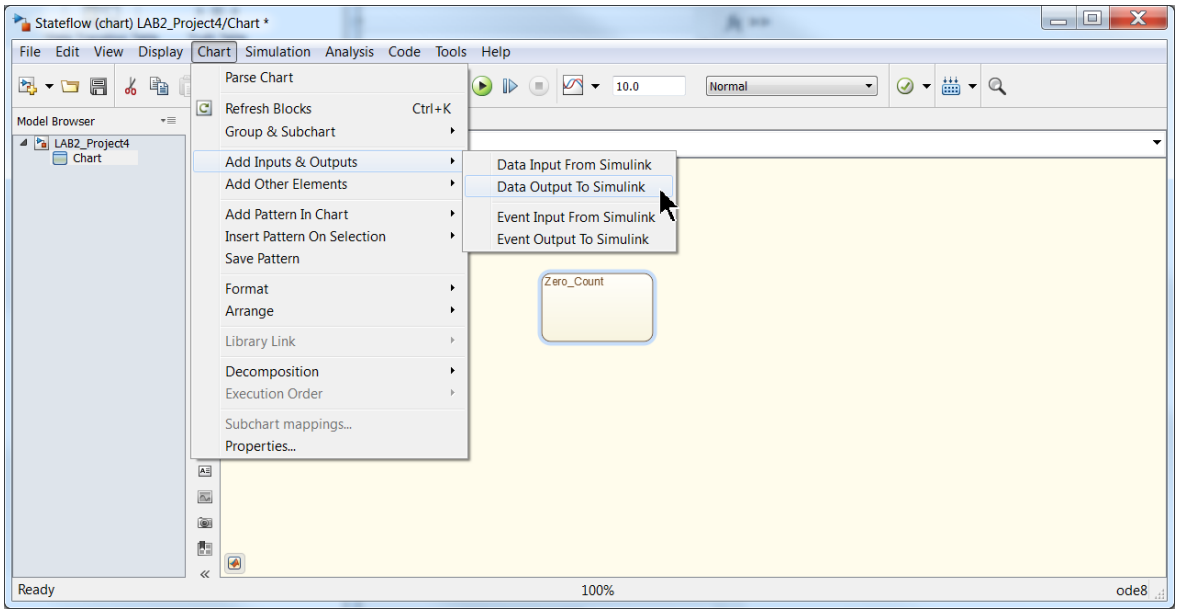


An empty state will be attached to your mouse pointer. Place it in your chart:

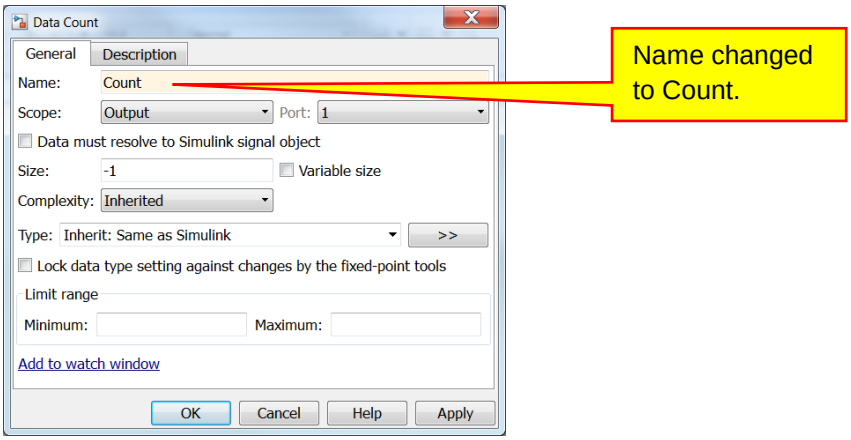


Click on the question mark to highlight it, and then type a name for the state. Call the state **Zero_Count**. Note that there can be no spaces in the name of a state:

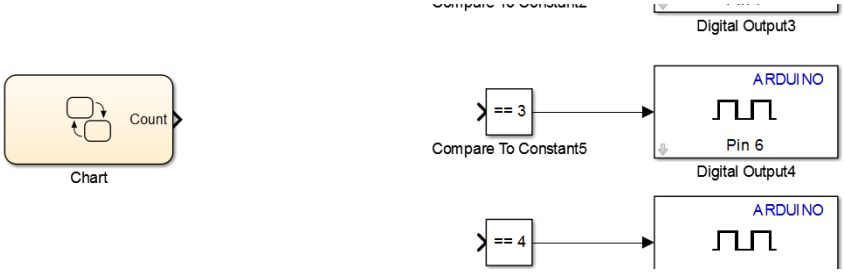
This state will be used to initialize the count to zero. First, we need to create an output for this chart. This output will keep track of the count and also be fed to the remainder of the Simulink model. To create an output, select **Chart**, **Add Inputs & Outputs**, and then **Data Output to Simulink**:



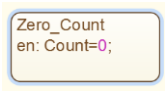
Change the name of the output to **Count**:



Since we specifically created an Output variable, the **Scope** is set to **Output** for us. Click the **OK** button to create the output. You will not notice any difference in the Stateflow chart. However, if you flip back to the Simulink diagram, you will notice that the chart now has an output:

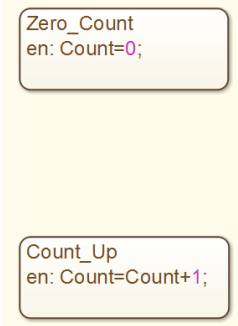


We will continue with the Stateflow chart. Now that we have defined an output, we can use it. In the Zero_Count state, add a new line with the text **en: Count=0;**

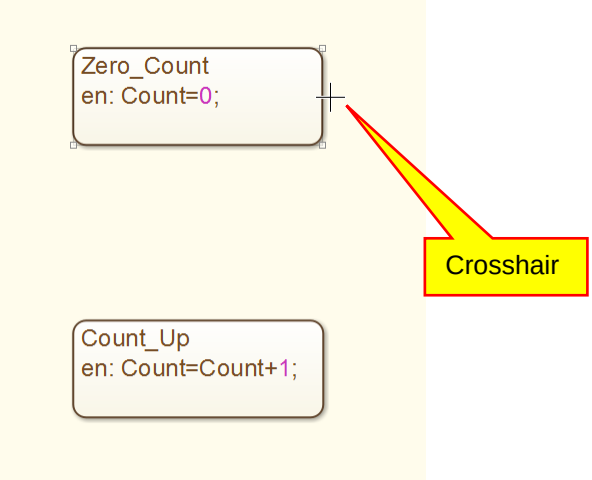


The text **en:** means that when we enter the state, execute the following command. The command is not executed while we are in the state, or when the state is exited. Thus, the line **en: Count=0;** sets the counter to zero when we enter the state, and only when we enter the state. Other available commands are **du:** and **ex:**. The **ex:** command executes the statement when a stated is exited. The du: command repeatedly executes a statement while it is in the state (if you want the value to change while in the same state).

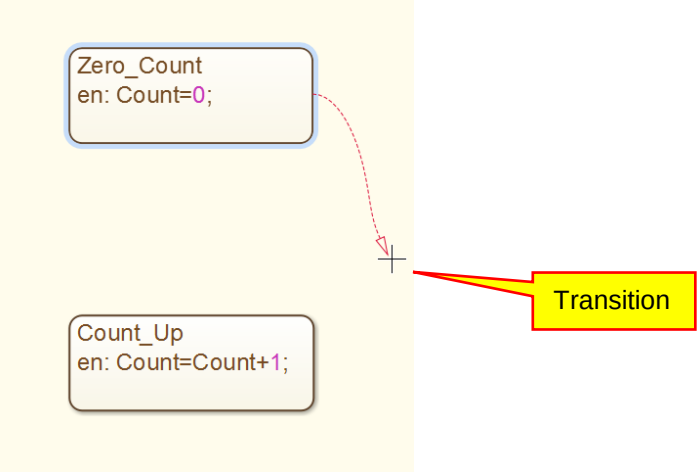
Next, we need to add a second state that counts. You can drag in a new empty state or right-click and copy the Zero_Count state. Fill in the state as shown:



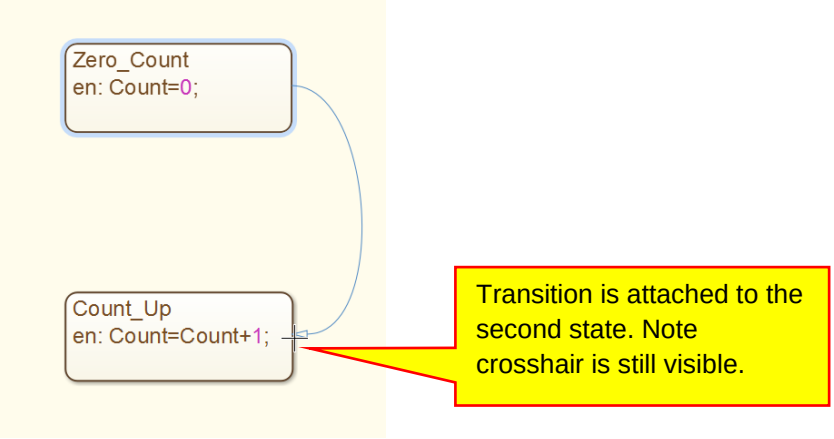
We only need two states for this chart. One state to reset the counter to zero, the other to count up. The next things we need are transitions to determine how we go between states. To create a transition, place the mouse pointer over the edge of the top state. The cursor will be replaced with a crosshair:



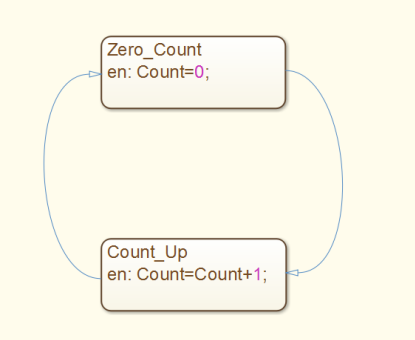
When you click the left mouse button: a transition from the first state becomes attached to the pointer:



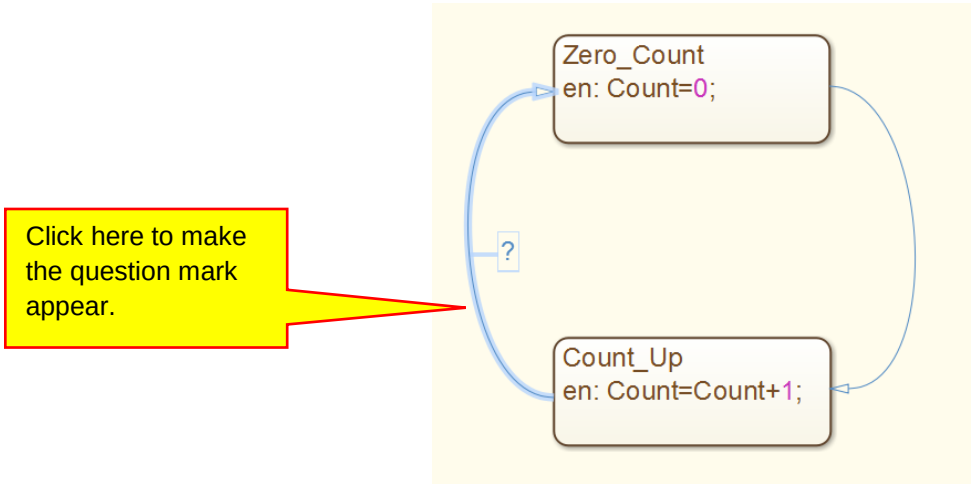
Place the mouse pointer over the right edge of the **Count_Up** state. The Transition will become attached to the state:



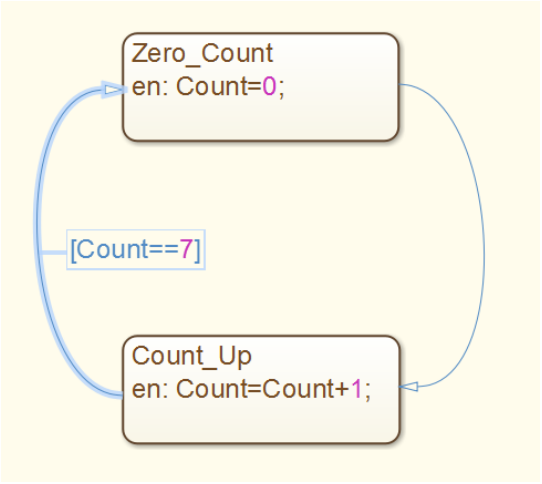
When you release the mouse button, the transition will become attached to the state. Repeat the process to add a transition from the **Count_Up** state back to the **Zero_Count** state:



These transitions are called unguarded transitions in that the transitions will always be taked unless we add a "guard." The only time we want to be in the zero state is when the count has reached 7. Once we reach 7, we want to go back to the Zero_Count state to reset the counter. Thus, we only want to take that transition when the count is 7. Click on the left transition. A question mark will appear:

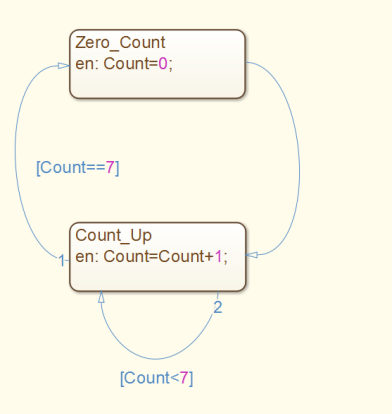


Click on the question mark and enter the text [Count==7]. Note the double equals sign (==):

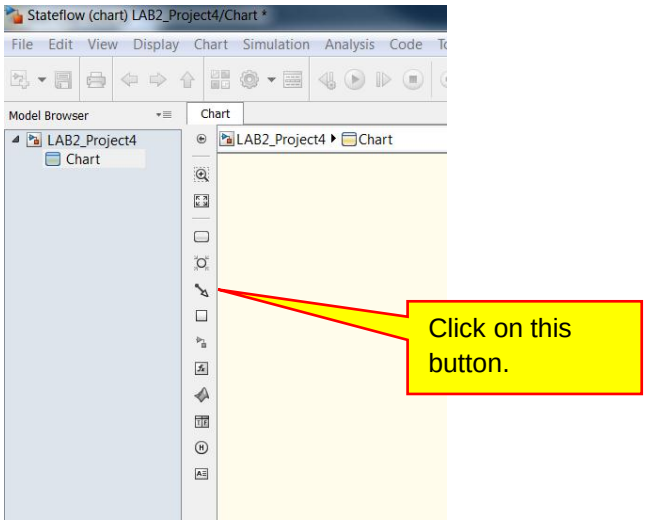


This guard means that the left transition will only be taken when the count reaches 7.

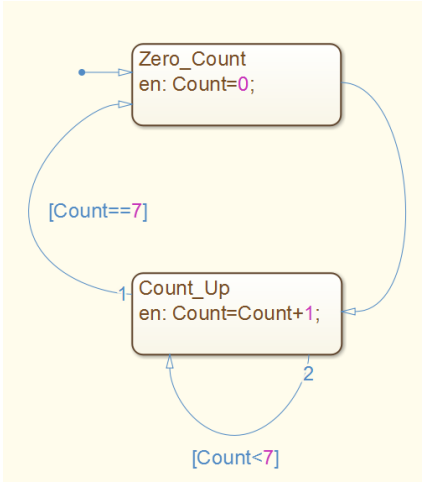
Since the count up state only counts up when the state is entered (because we used the command **en:Count=Count+1;**), we need to add a transition in the **Count_Up** state to itself:



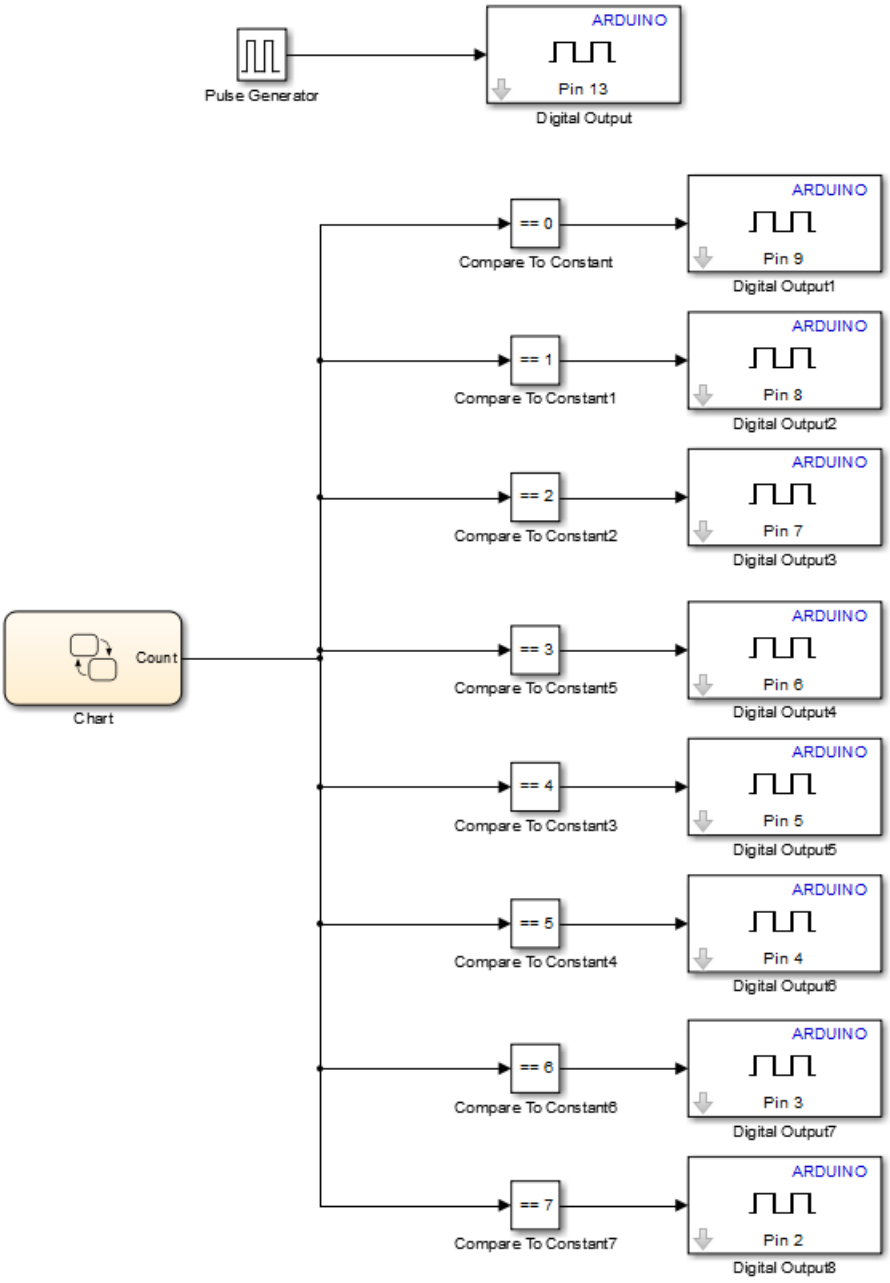
The last thing we need to specify is which state the chart starts in when the controller powers up. This is done by adding a default transition. Click on the default transition button  as shown below and drag in a default transition:



Connect the default transition as shown:



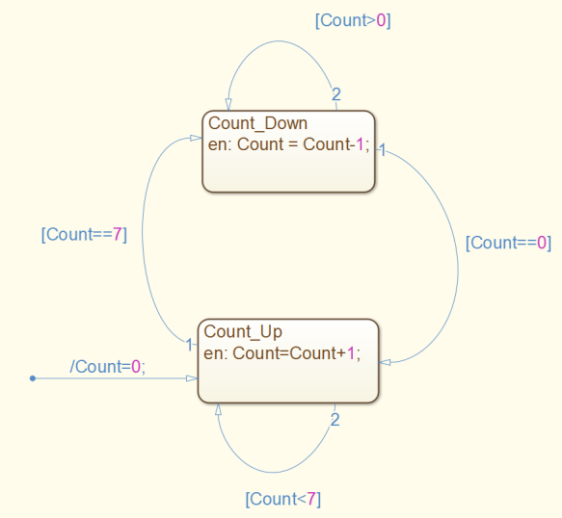
The chart is now complete. Switch back to the model and connect the chart to the remainder of the Simulink model:



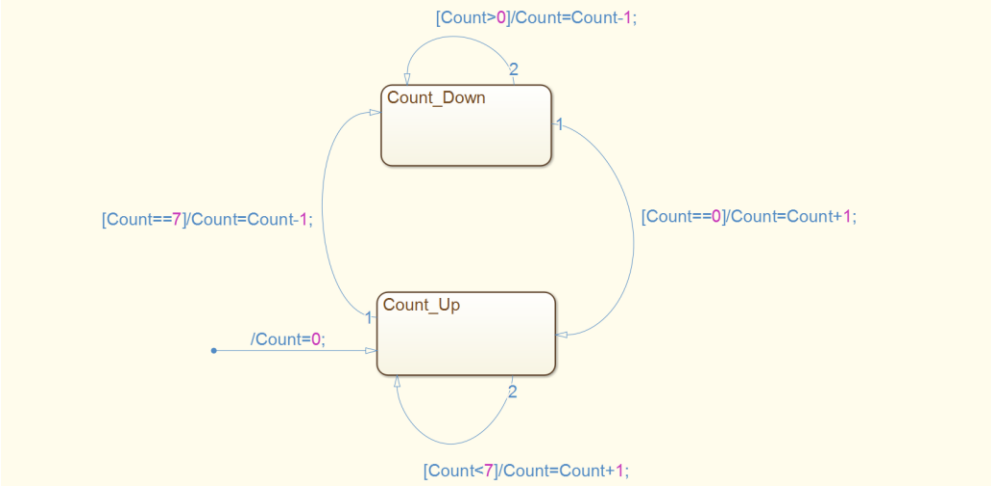
Note that the chart executes once every time step. Note that in the previous model, we specified a fixed time step of 0.5 seconds, and this model started as a copy of the previous model. Thus, the transitions are checked each fixed time step, or 0.5 seconds in this example.

Demo II.4: Demo the ring counter using a counter implemented with the Stateflow chart.

As another example, we will create an up-down counter, where the counter counts up from zero to 0, and then from 7 back down to zero. The Stateflow chart for this model is shown below:



We see one new item in this chart (besides the added logic) which is a transition action. In the default transition we see the text **/Count=0;**. This command is different from then **en:**, **ex:**, and **du:** commands mentioned earlier which execute a statement upon entering, exiting, or while in a state. **/Count=0** is called a transition action. The action **Count=0;** is executed when the transition is taken and happens before it enters the next state. Here, we are using the transition action to initialize the counter when the controller first starts. Note that we can implement the same logic function using transition commands for all of the transitions:



Demo II.5: Demo the up-down ring counter using a counter implemented with the Stateflow chart.

Exercise II.4: Using Stateflow and a digital input, create a ring counter that shifts to the right when a switch is open and shifts to the left when the switch is closed.

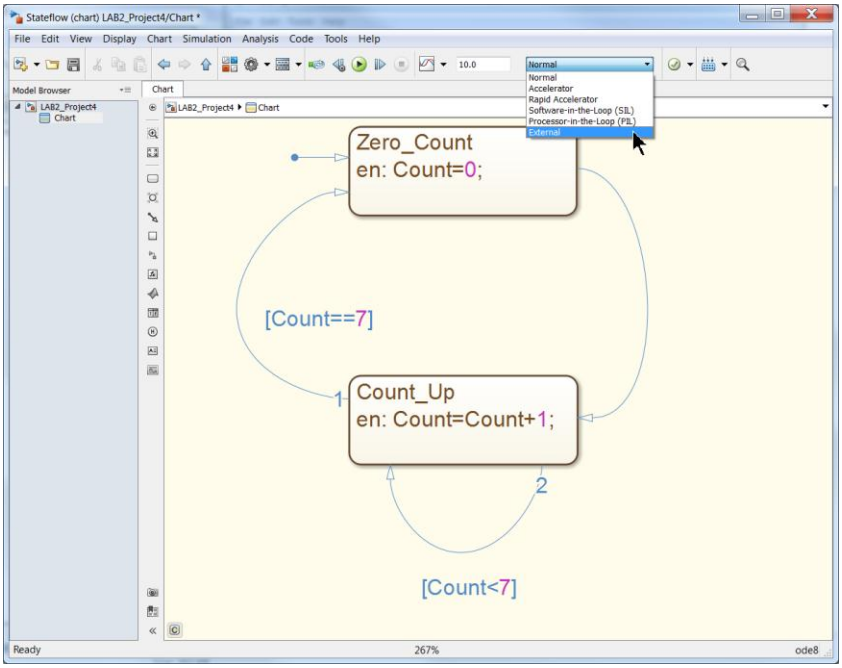
Exercise II.5: Using Stateflow and a second digital input, modify your solution to Exercise II.4 by adding a switch that will hold the count at its present value. The first switch will still determine the direction. Note that '&&' is the logical AND in Stateflow syntax.

1. External Mode

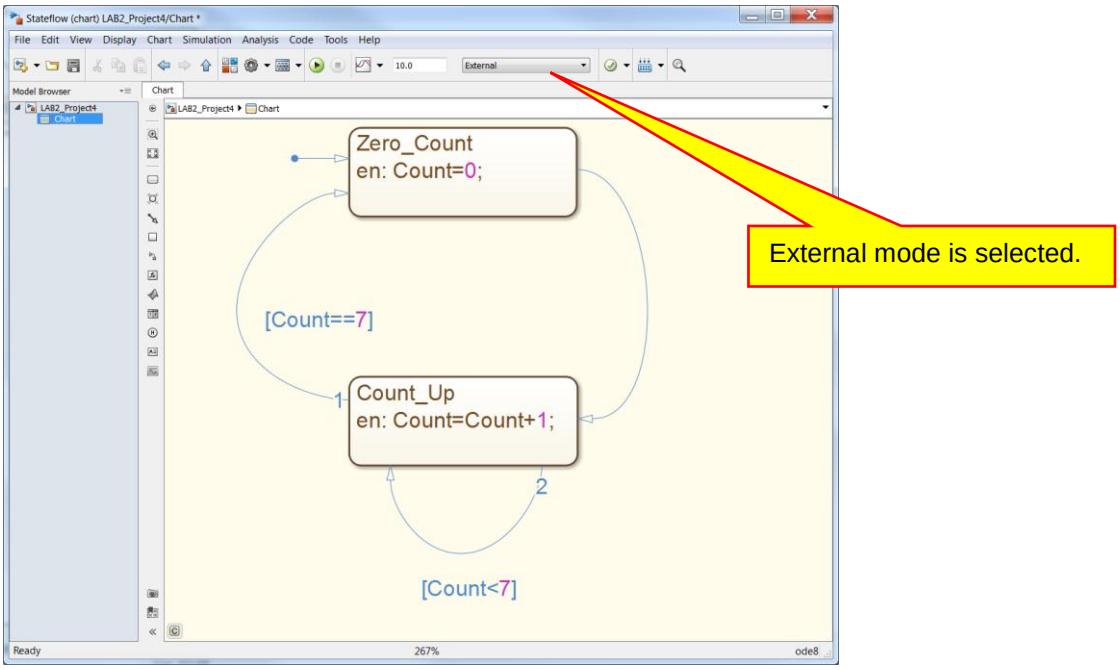
External model will allow us to run a model on the Arduino and monitor the operation of the Arduino using the Simulink model in real-time. This is a great debugging tool in that it is the only way to look inside the Arduino controller and observe its logical operation. The problem with this method is that so much data is being

transferred between the Arduino and Simulink that only small models with a larger fixed time steps will run. Thus, we will demonstrate the tool here, but we need to be careful when using it in more complex models.

We need to specify that we will be using External mode. Click on the menu as shown and select External Mode:



External should be displayed as shown:



Enabling External mode tells Simulink to add code to the model that enables communication between the Arduino and Simulink. The problem is that data for every signal in the model is sent over the serial

communication line creating a large amount of overhead. This limits the use of External mode to large time steps and simple models.

a) Overrun Detection

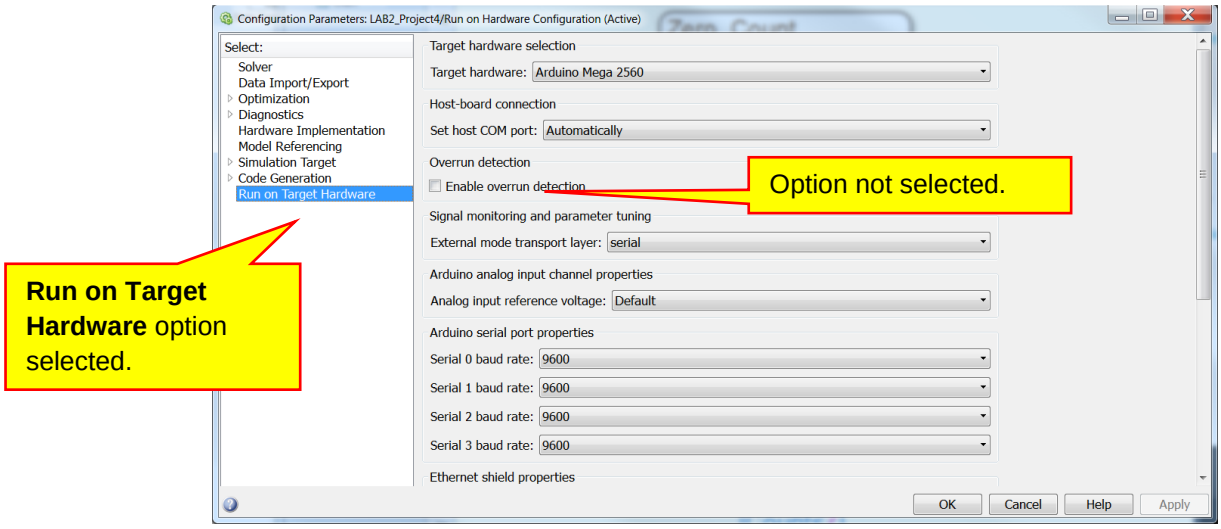
Inherently, a Simulink model is executed repeatedly as a loop. Blocks in the model execute in an order determined by when data becomes available in the model. All blocks in the model are executed once (unless there are conditional blocks and repeated loops), and then then the entire model is repeated again. The fixed time step we specify in Simulink is the rate at which the model execution is repeated. All calculations in the model must be completed in an amount of time less than the fixed time step for the model to run in real time. Note that the fixed time step is not the clock rate at which instructions are executed on the Arduino. Instead, it is the time we have allotted to the controller to execute all calculations in the model. If the time step is longer than the time needed to perform the calculations, the controller will wait before it loops and executes the model again. Here we say that the calculations finish on time. When the calculations finish on time, we know how often the model execution repeats; it repeats at the fixed time step. In this case, the model runs is real time.

If the time step is too small and the model calculations take longer to complete than the fixed time step, the calculations are said to have finished late, and we have what is called an overrun. When the calculations finish late we have no idea how often the model execution repeats; it repeats at a rate determined how long the calculations take to complete. In this case, the model does not run in real time.

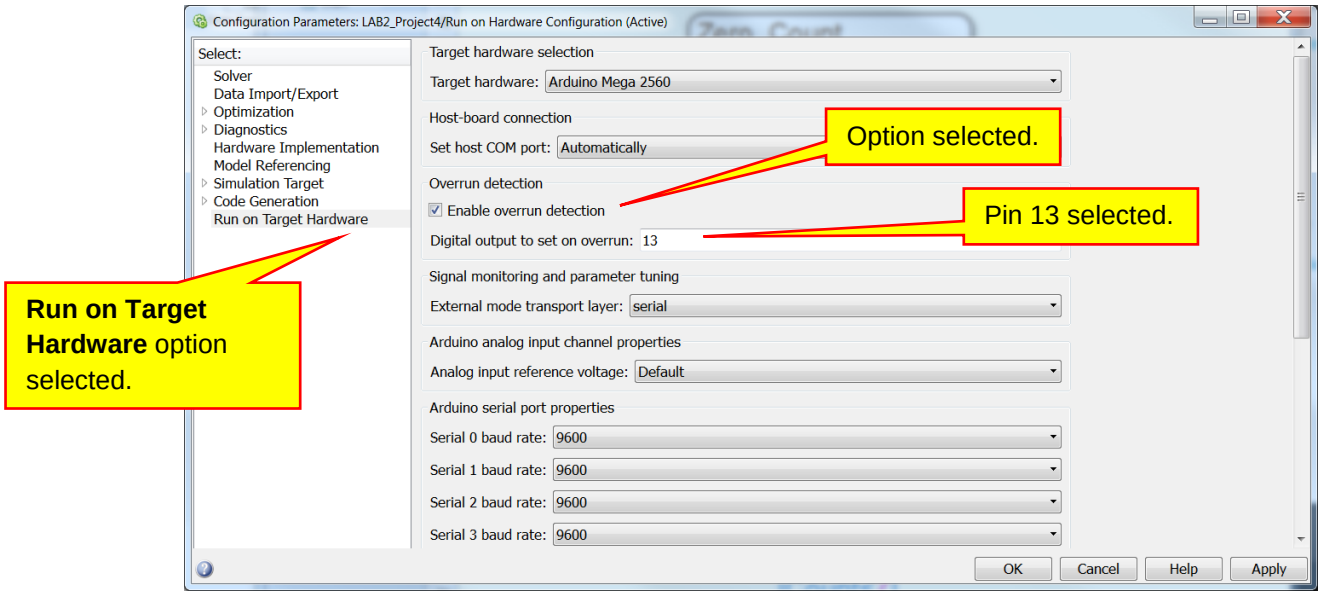
Having a fixed time step that is too small can have three consequences: (1) The best scenario is that the model runs, but it does not run in real time, which will make the application of the controller incorrect. This is dangerous and sometimes hard to determine immediately. (2) The model runs, but the logic is not correct. (3) The controller just hangs up and completely dies.

Because these problems can be quite serious, the Arduino has overrun detection so that we can easily determine if our model overruns. Also note that since External Mode uses so much serial communication, it tends to make many of our models overrun. When we enable overrun detection, the Arduino will make a specified digital output high when an overrun occurs. We can use this digital output to drive an LED to visually indicate when an overrun has occurred.

To enable overrun detection, select **Simulation** and then Model **Configuration Parameters** to obtain the Configuration Parameters dialog box. Select the **Run on Target Hardware** option:

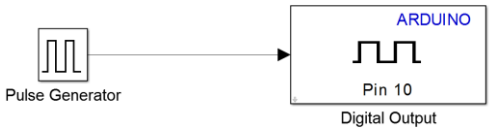


Enable the overrun detection in the Configuration Parameters dialog box and specify pin 13 as the digital output:



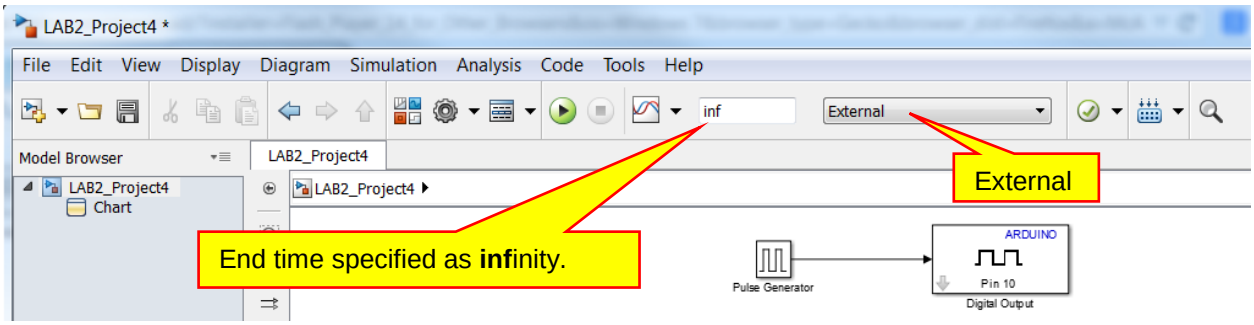
Note that pin 13 uses the on-board LED on the Arduino Mega board. This selection makes it easy to use overrun detection. Note that if the LED illuminates, we have an overrun. If the model runs in real-time, the LED should never turn on. **You should use overrun detection in all of your models.** When a model does not run properly, the first thing you should check is if there is an overrun. Click the OK button to return to the model.

Because we are using pin 13 for overrun protection, we will need to change the digital output for our flashing LED and add our own LED to the board. Change the digital to a different pin:



And wire up an LED and resistor as described in Section I.C.2.

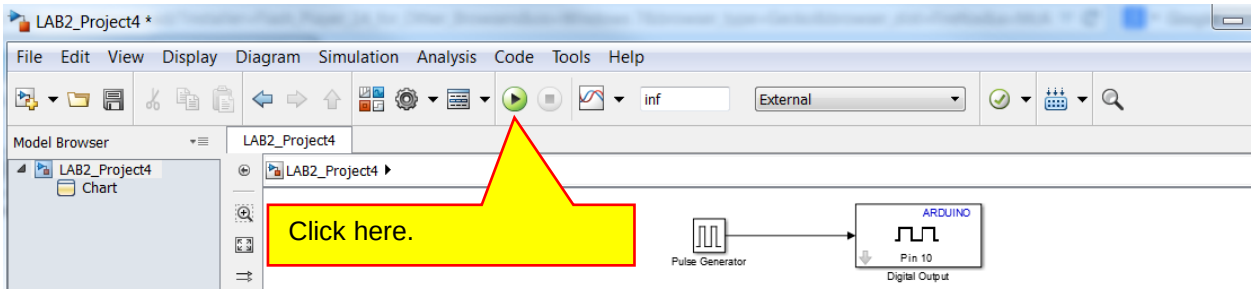
In the Simulink model, we need to specify that the model runs for ever:



When we run a model on the Arduino controller board, it will run forever. (Forever being when we turn off the power or reprogram it.) When we run a Simulink simulation, it will run to the specified end time. By default, the

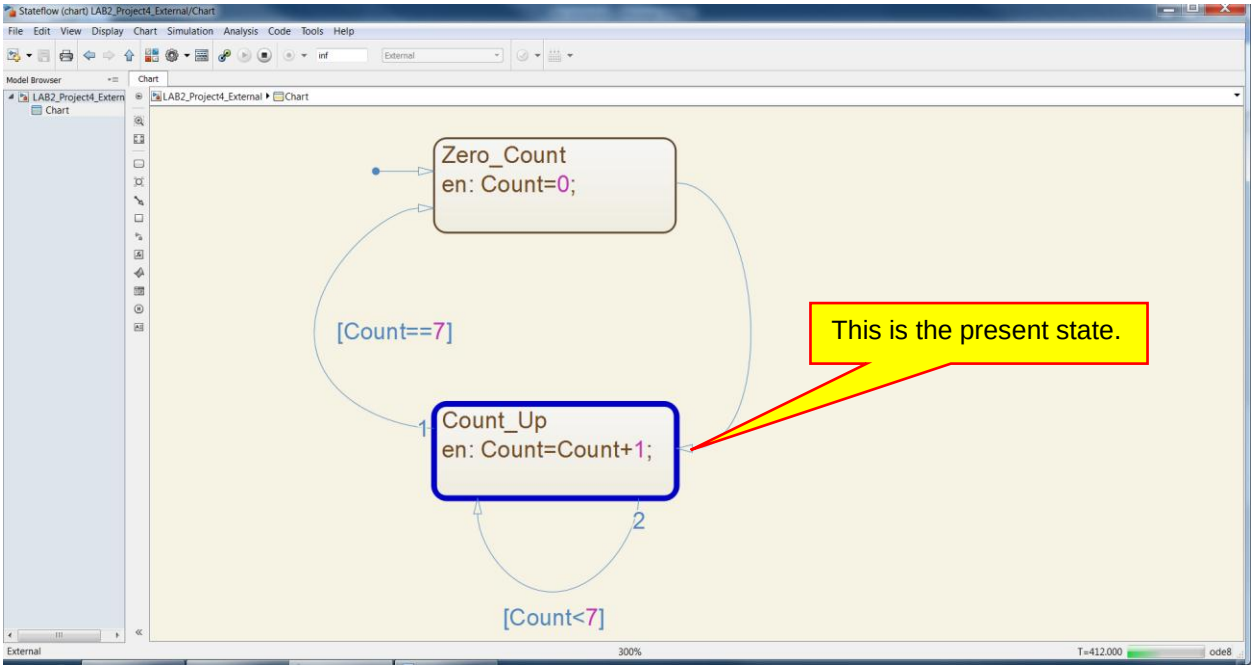
end time is 10 seconds. When we use External mode, both the Simulink model and the model running on the Arduino must execute at the same time. When Simulink model stops running, communication will stop as well. Since we do not know how long we will be testing the Arduino model, we will specify an end time of infinity, which means the Simulink model will run until we stop it.

To run the model in external mode, click on the Run button  :

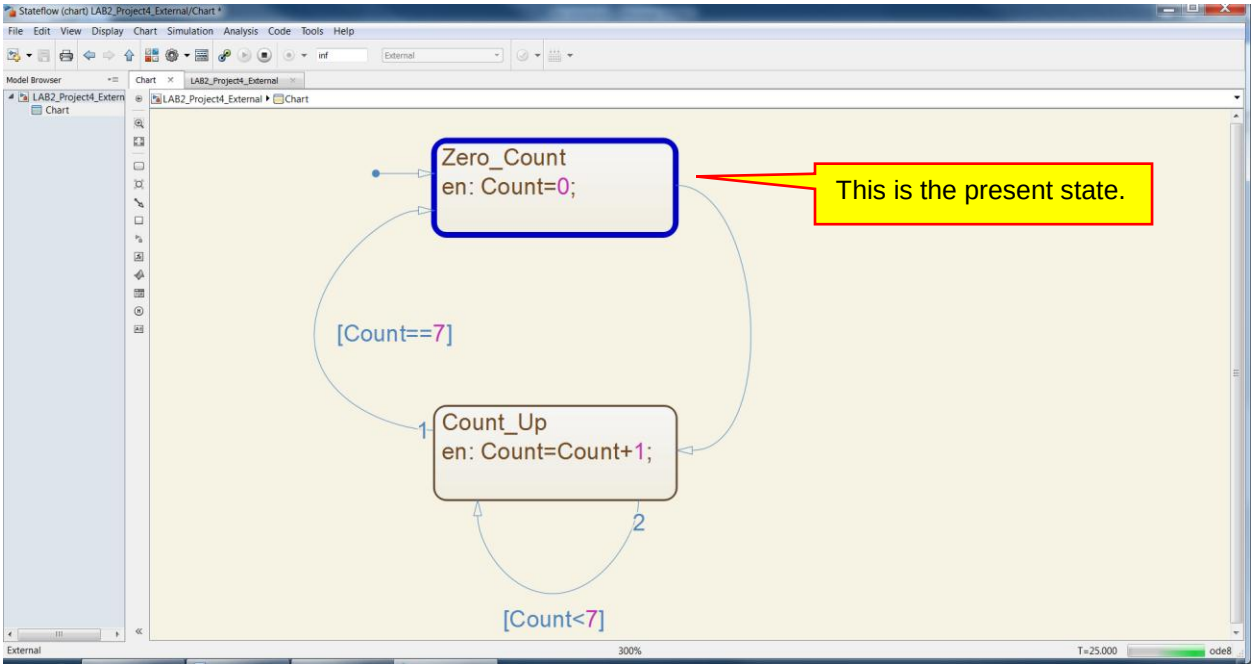


After the model is built and downloaded, the TX and RX lights should flash continuously on the Arduino board illustrating the communication between the Arduino controller and Simulink in External mode. Since there is so much communication going on between the Arduino and the Simulink model, the on-board LED will illuminate indicating that we have an overrun. This is a side effect of external mode. Most of the time we use it, it will cause an overrun. We will ignore the error as we are demonstrating the use of external mode and do not require the model to run in real-time.

If you open the Stateflow chart, you will see the state transitions in real-time. The present state is highlighted in blue:



After a few moments, the state changes:



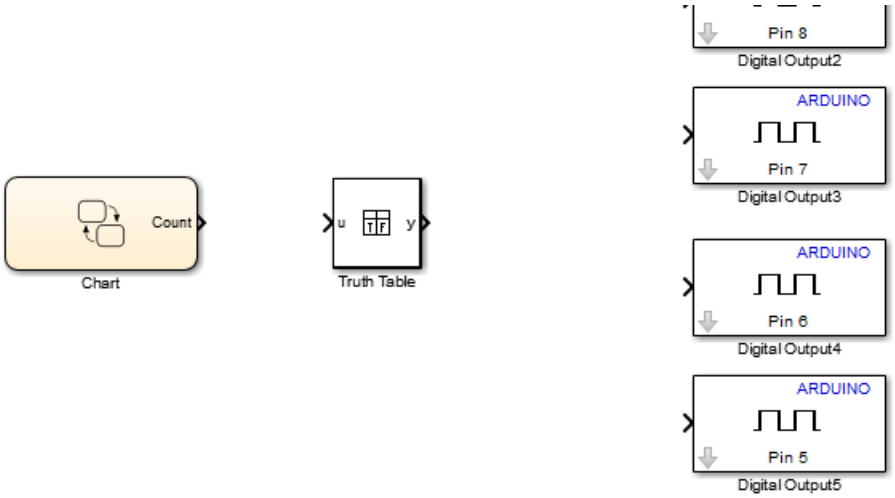
We can use this tool to observe and debug the logical operation of our Stateflow charts. In more charts, this becomes a very useful tool.

Before continuing, you must terminate External mode by clicking on the **Stop** button .

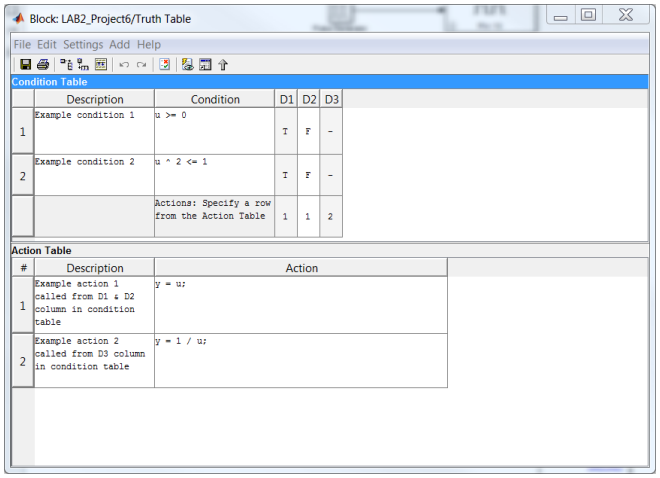
Demo II.6: Show external mode displaying the Stateflow chart.

D. Using Truth Tables to Implement Logic

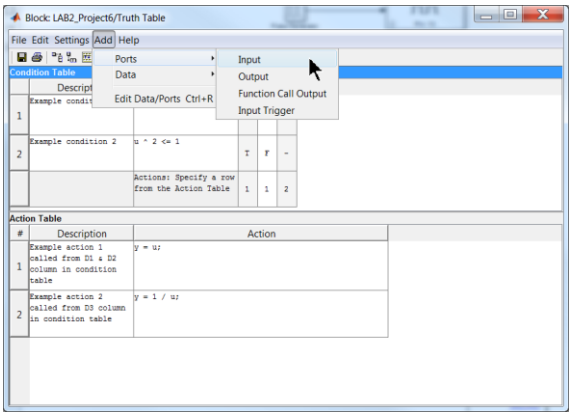
Truth tables provide a convenient way to document and implement logic functions. They do not provide memory the way a Stateflow chart does. However, they allow us to specify a large number of conditions, and allow us to manipulate combinations of those conditions. As a simple example, we will use the last model of the up-down counter with the Stateflow chart and replace the **Compare to Constant** blocks with a truth table. The **Truth Table** block is located in the **Stateflow** library:



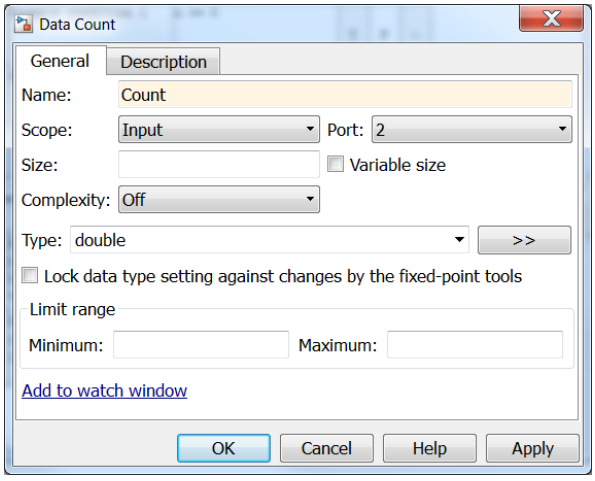
Double-click on the **Truth Table** block to open it:



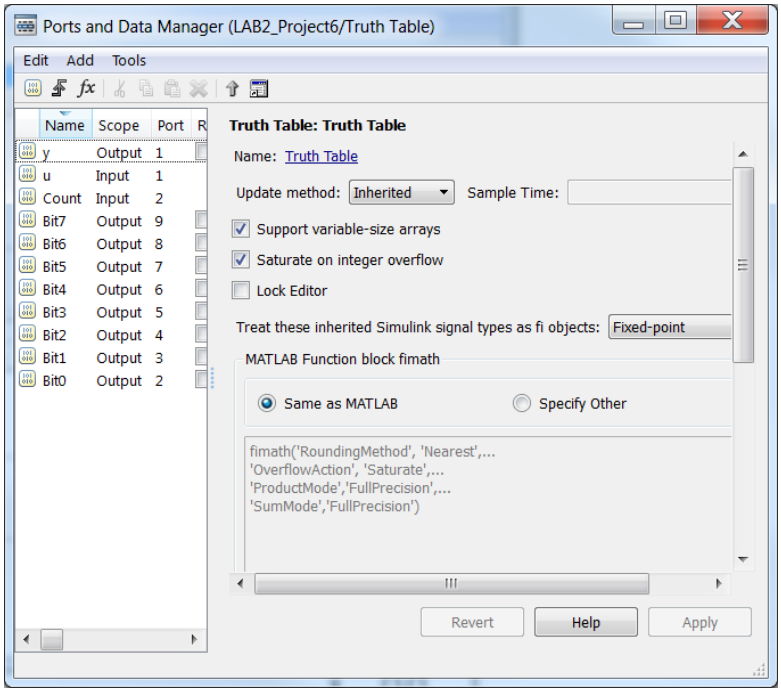
First we must add the input variable, which we will call Count. Select **Add**, **Ports**, and then **Input** from the Truth Table menus:



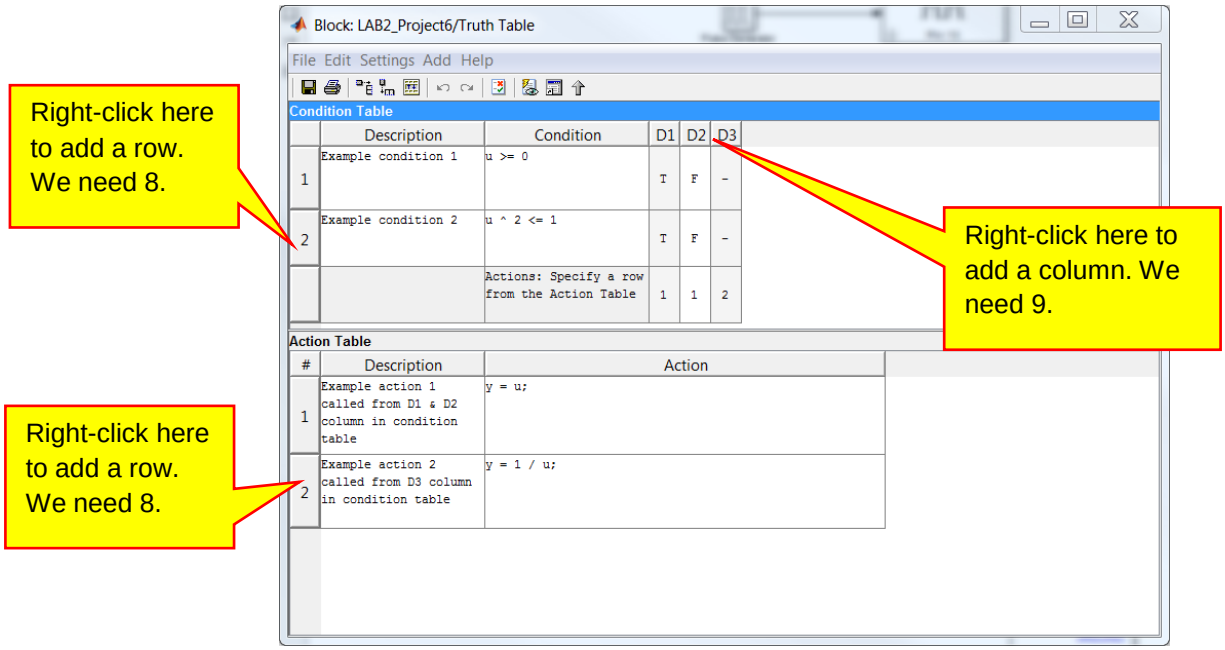
Specify the name of the variable as Count:



Click the **OK** button. We will have 8 outputs called Bit0, Bit1, Bit2, …, Bit7. Use the same procedure but specify the data as an output. When we placed the Truth Table, it had a default input parameter called u and a default output parameter called y. We need to delete these. From the Truth Table menus, select **Add** and then **Edit Data/Ports**:



The Ports and Data Manager allows us to modify and delete any of the ports in the Truth Table. Delete the y output and the u input and close the Port and Data Manager. We need to add rows and columns to our table:



After adding rows and columns, fill in the table as shown:

Block: LAB2_Project6/Truth Table

File Edit Settings Add Help

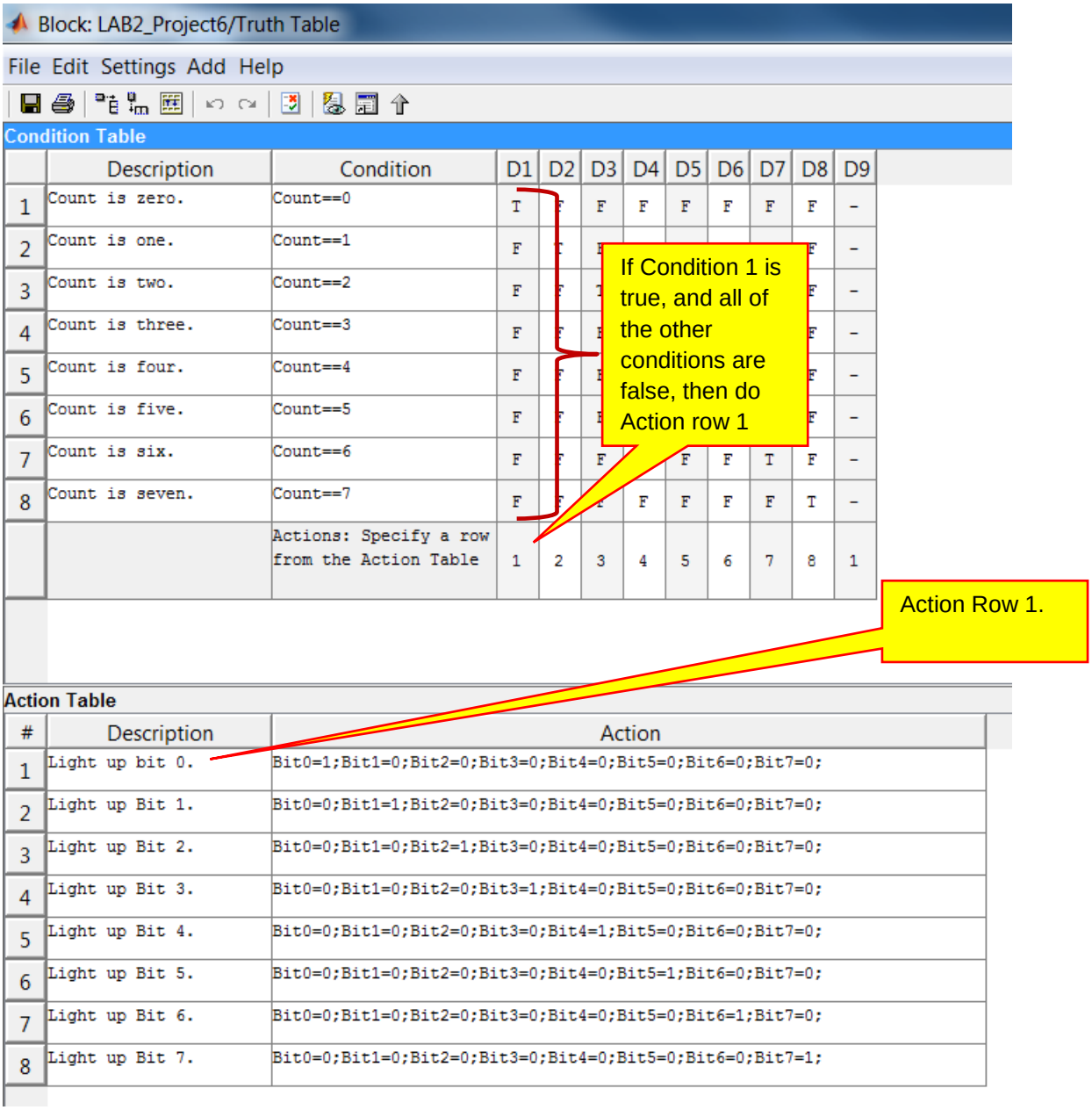
Condition Table

	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8	D9
1	Count is zero.	Count==0	T	F	F	F	F	F	F	F	-
2	Count is one.	Count==1	F	T	F	F	F	F	F	F	-
3	Count is two.	Count==2	F	F	T	F	F	F	F	F	-
4	Count is three.	Count==3	F	F	F	T	F	F	F	F	-
5	Count is four.	Count==4	F	F	F	F	T	F	F	F	-
6	Count is five.	Count==5	F	F	F	F	F	T	F	F	-
7	Count is six.	Count==6	F	F	F	F	F	F	T	F	-
8	Count is seven.	Count==7	F	F	F	F	F	F	F	T	-
		Actions: Specify a row from the Action Table	1	2	3	4	5	6	7	8	1

Action Table

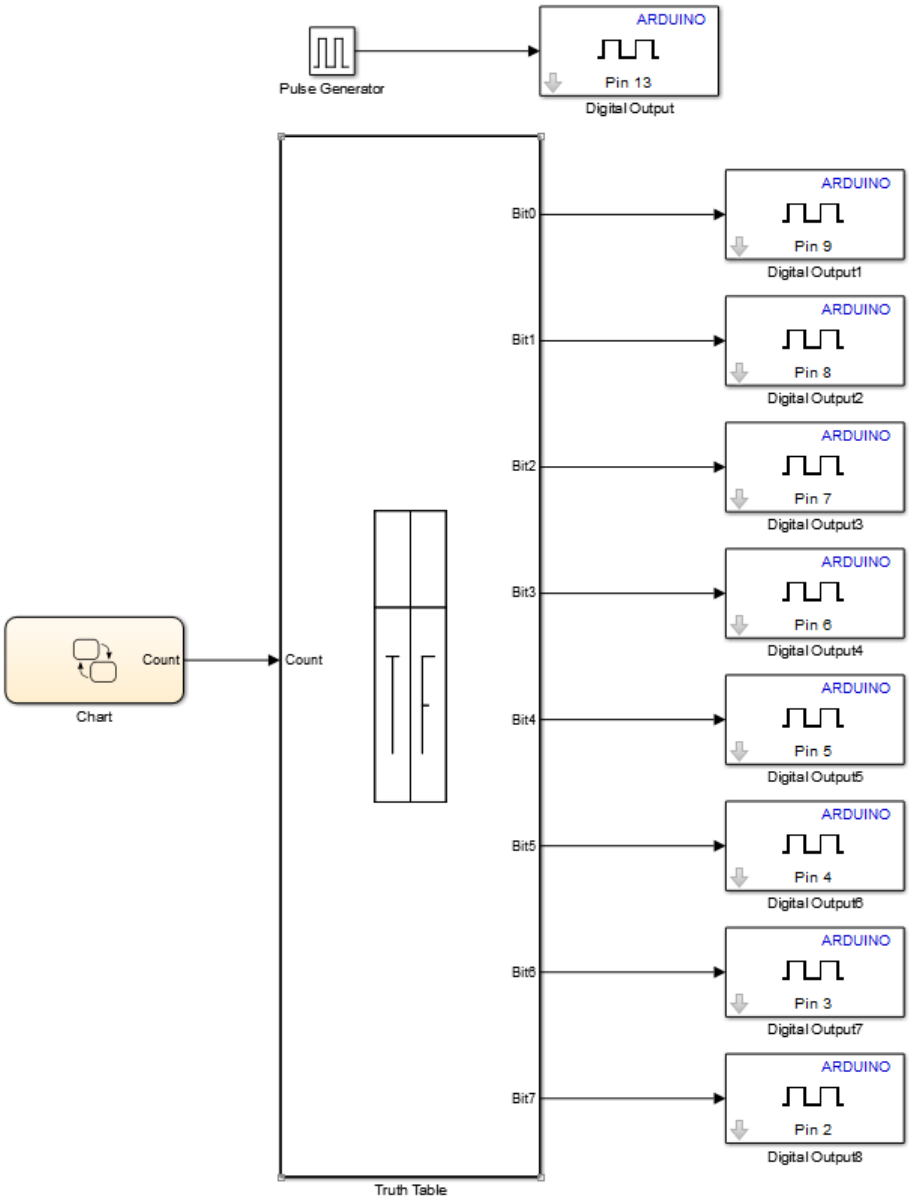
#	Description	Action
1	Light up bit 0.	Bit0=1;Bit1=0;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
2	Light up Bit 1.	Bit0=0;Bit1=1;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
3	Light up Bit 2.	Bit0=0;Bit1=0;Bit2=1;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
4	Light up Bit 3.	Bit0=0;Bit1=0;Bit2=0;Bit3=1;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
5	Light up Bit 4.	Bit0=0;Bit1=0;Bit2=0;Bit3=0;Bit4=1;Bit5=0;Bit6=0;Bit7=0;
6	Light up Bit 5.	Bit0=0;Bit1=0;Bit2=0;Bit3=0;Bit4=0;Bit5=1;Bit6=0;Bit7=0;
7	Light up Bit 6.	Bit0=0;Bit1=0;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=1;Bit7=0;
8	Light up Bit 7.	Bit0=0;Bit1=0;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=1;

Note that the descriptions are plain English and for documentation only. We have simple conditions and they can be more complex using logical AND (&&) and logical OR (||) constructs. Note that columns D1 through D9 are “decisions.” For example column D1 will be used if condition 1 is true and conditions 2 through 8 are false. If these criteria are met, then Action 1 will be taken, which sets Bit1 to a one and all the other bits to a zero:



Similarly for Decision column 2, if Condition 2 is True and all of the other conditions are false, action row two will be used. The only other column that is a bit strange is Decision column 9. Note that we have 8 conditions and each of those can be true or false. From a logic point of view, and of the conditions can be true or false independent of the other conditions. This is not possible for our case because of the function of the counter, but Truth Tables don't know the limitations of the inputs and the conditions. Since we have 8 conditions, and any condition can either be true or false, there are 2^8 possible combinations of how those conditions can be true or false. Do cover them all, we would actually need 2^8 decision columns. Since we know that none of these are possible for our counter, we can catch these all in Decision column 9, which is the default decision. Having all dashes means that if we have a combination of conditions that are not covered by the decisions in D1 through D8, it will use D9, which specifies action 1. For our example, D9 should never occur. In practice, the default column can be used to catch an error, such as, we know that this should never happen, but if it does, have it take the appropriate action.

After saving the Truth Table, return to Simulink and resize the table and connect it as shown:



Demo II.7: Demo the up-down ring counter using a counter implemented with the Stateflow chart and a truth table as a decoder.

Exercise II.6: Modify the Truth Table from Demo II.7 to light up the odd numbered LEDs when the count is odd and light up the even numbered LEDs when the count is even.

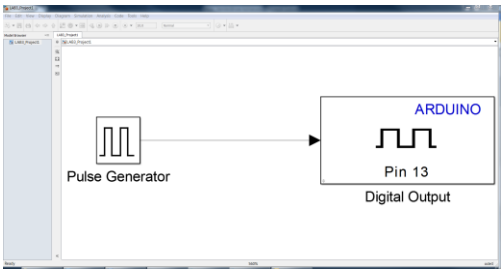
Exercise II.7: Modify the Truth Table from Demo II.7 so that two adjacent LED's travel up and down the LED display. Two LED's must be illuminated at all times.

Lab III
Analog Input, Sensors, Subsystems, and Lookup Tables

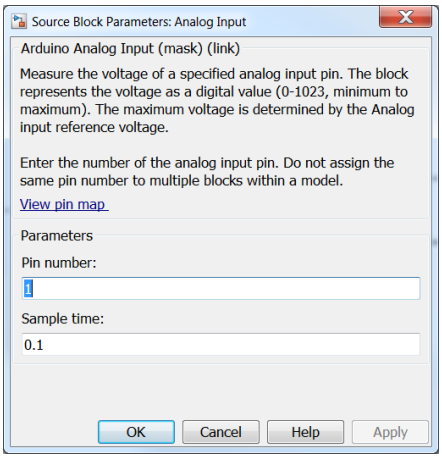
The Arduino **Analog Input** block uses a 10-bit analog to digital converter to convert a positive analog voltage to a number between 0 and 1023. The voltage conversion range can be changed and depends on a setting we can specify. We will use the Analog Input for two applications. The first is a linear control reference where we can specify a parameter with an analog input. The second is to read analog sensors. Many sensors output an analog voltage, and we can read these sensors with the analog input.

A. Analog Input Block

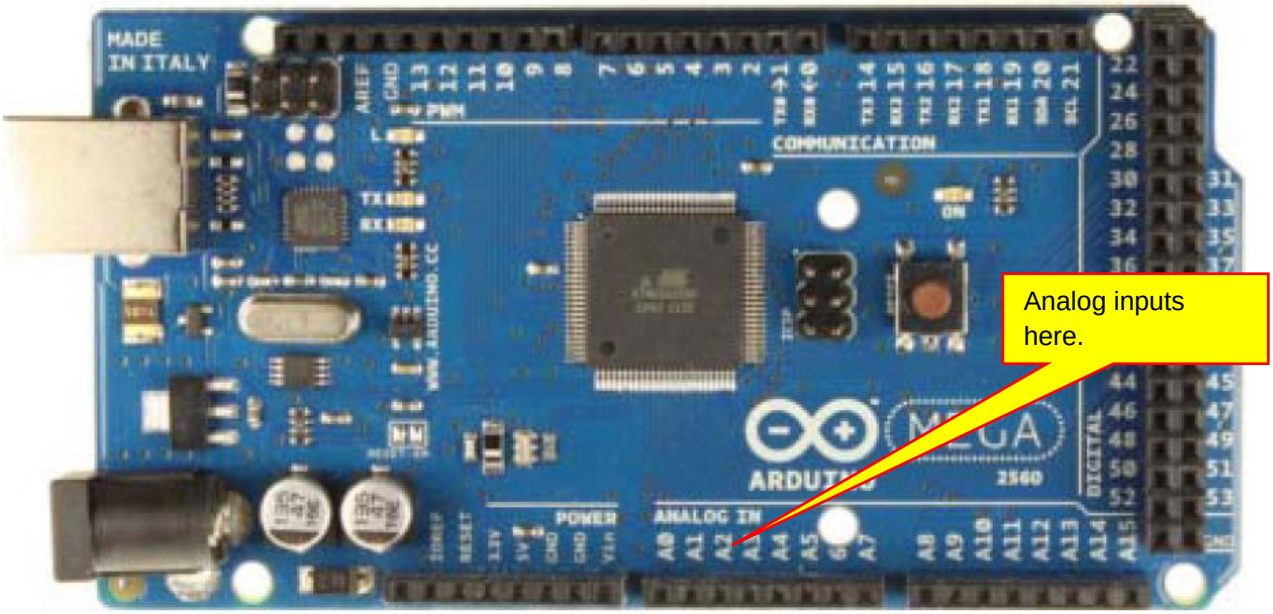
The first thing we will do is try to observe the output value of the Analog Input block directly. We will do this using the External mode of Simulink. First, we will start with the simplest model we can, the one from Lab 1, demo one where we flashed the onboard LED. Open this model and save it as Lab3_Project1:



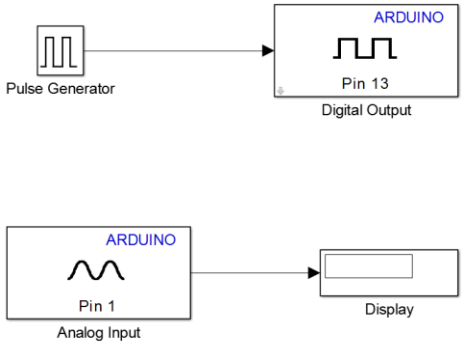
Add an **Analog Input** block located in **Target for Use with Arduino Hardware** library to your model. Double-click on the Analog Input block and specify pin 1 as the **Pin number** and set the **Sample time** to be 0.1 seconds:



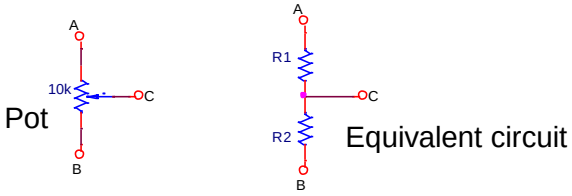
Important note! Analog input pin 1 is labeled as pin **A1** on the Arduino mega board. Make sure you use **ANALOG IN** pin **A1** on the Arduino Mega board and not pin 1:



The **Sample time** is how often the block will read and convert a new value of the input. Click the **OK** button to return to the model. Add a **Display** block located in the **Simulink / Sinks** library to your model and connect it to the Analog Input block:



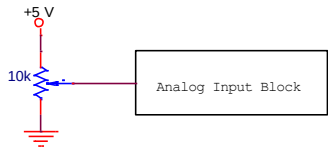
Lastly, we need to provide an analog input voltage to **Pin 1**. We will do this with a potentiometer (pot). A potentiometer can be thought of as a variable voltage divider:



The potentiometer can be thought of as two variable resistors in the equivalent circuit provided that:

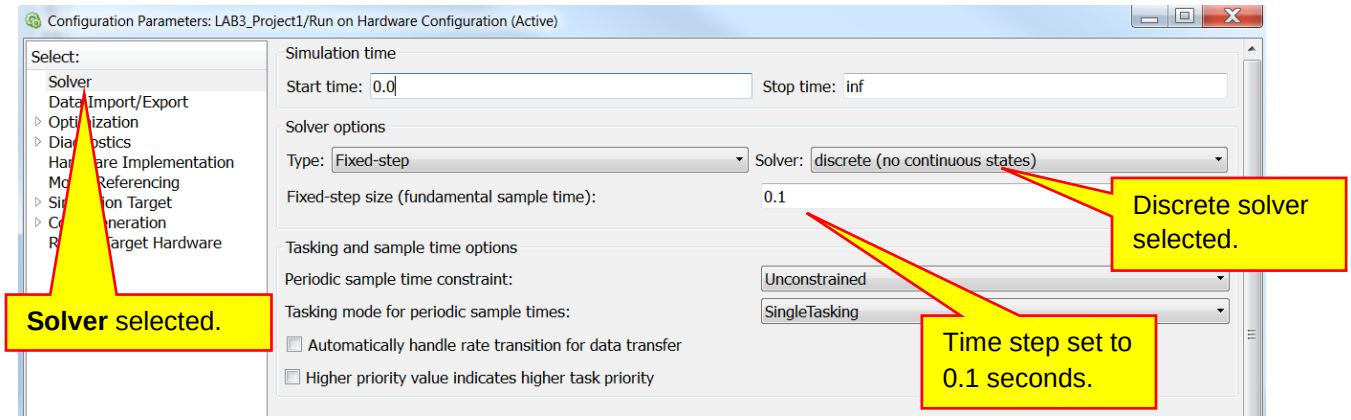
$$R_1 + R_2 = 10\ k\Omega$$
$$0 \le R_1 \le 10\ k\Omega$$
$$0 \le R_2 \le 10\ k\Omega$$

As we turn the knob/when/screw on the potentiometer, one resistor increases and the other decreases such that the sum is constant. We will use the potentiometer in the circuit below:



The potentiometer will provide a 0 to 5 V signal that we will use as a test input to the Analog Input block.

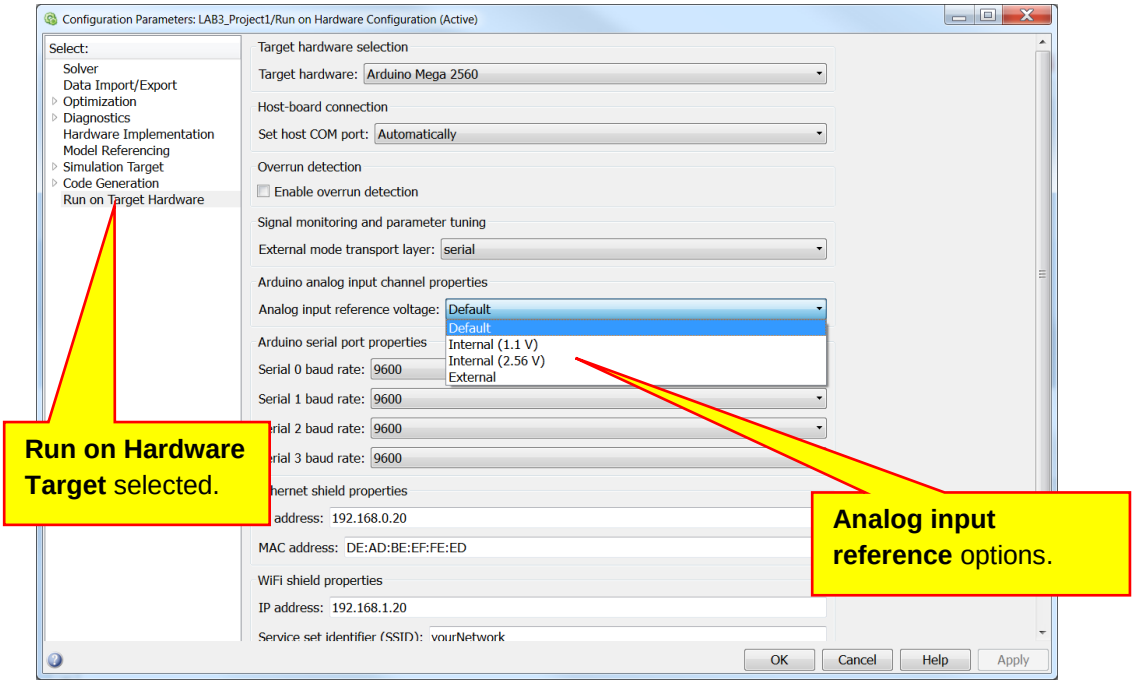
Before we set up external mode, we will look at a few of the Simulink setting at apply to the Analog input block. Select **Simulation** and then **Model Configuration Parameters** from the Simulink menus. Select the **Solver** item. Fill in the items as shown:



We have selected a discrete solver. This implies that values for the next time step are calculated with information from the previous time step. This uses the least amount of calculation to calculate the next set of values. Other solvers use information from several of the previous times steps to calculate the next set of values. For example, ODE4 is a 4th order Runge-Kutta method that calculates the next value basted on values from the previous 4 time steps. This method is more accurate but also requires more computation. If the model is too complex and requires too much computation, the calculations for the next time step cannot be calculated within the fixed time step, and the calculation will be invalid. The result is that the controller will either hang or produce erroneous outputs. The Discrete solver is the fastest solver and produces the fastest models. However, using the Discrete solver limits us to discrete time functions, which is not yet a problem for us, but could be in the future. We will not be able to use Laplace transforms, continuous time filters, or analog integrators (the 1/s block). If you need such functions, you will need to use the discrete-time equivalent functions. (Sorry, you might have to take digital signal processing…)

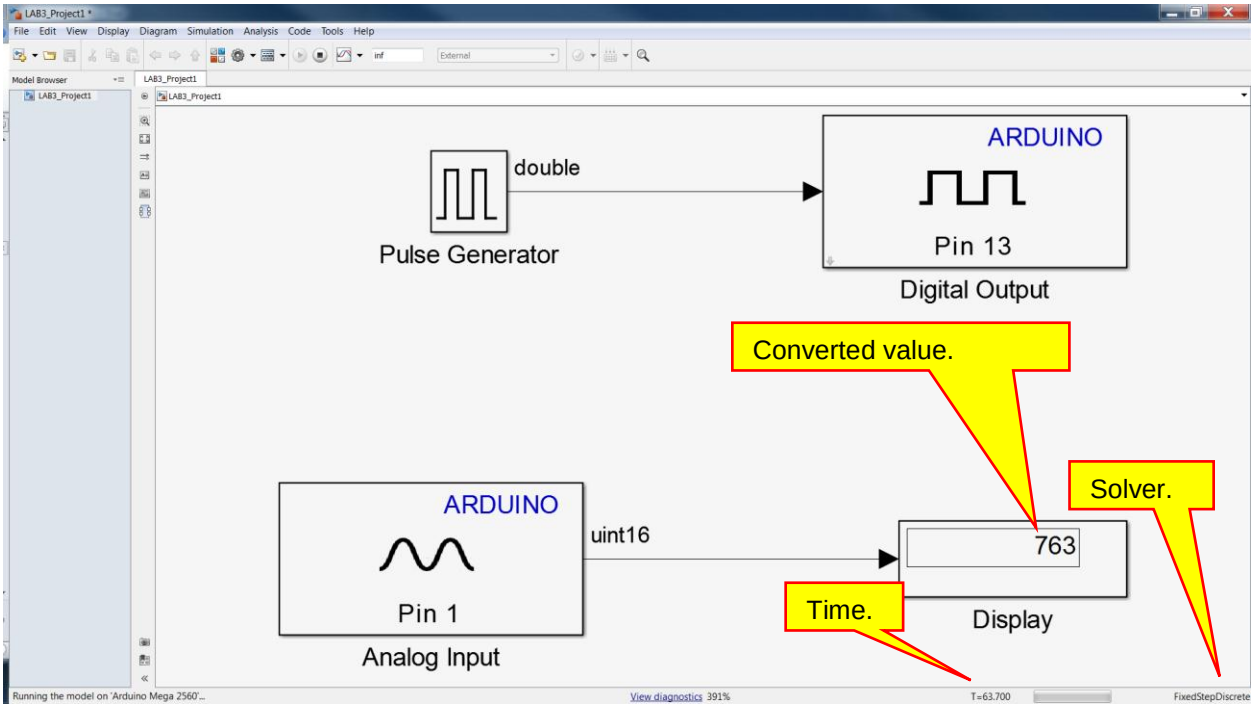
The fixed time step is specified as 0.1 because we specified a sample time of 0.1 seconds in the Analog Input block. The fixed time step must be 0.1 seconds or less. We cannot make it too much smaller because the time step will be too fast for the Arduino in External mode, which we will look at next.

Next, select the **Run on Target Hardware** selection and then click on the **Analog input reference voltage** pull-down menu:



This menu specifies the reference for the Analog Input block. The Default value is 5 V, which means that an analog input voltage from 0 to 5 V will be converted to a digital value from 0 to 1023. If we select a different reference, say 2.56 V, than an analog input voltage from 0 to 2.56 V will be converted to a digital value from 0 to 1023. Choose the **Default** menu selection (which should have already been selected) as our input if from 0 to 5V. Click the OK button to save the changes.

We now want to run the model in External mode. See section II.C.1 on page 44 for setting up and running a model in external mode. When you click the run button  , after the model builds and downloads, the display should indicate the value of the converted analog input, a number between 0 and 1023:



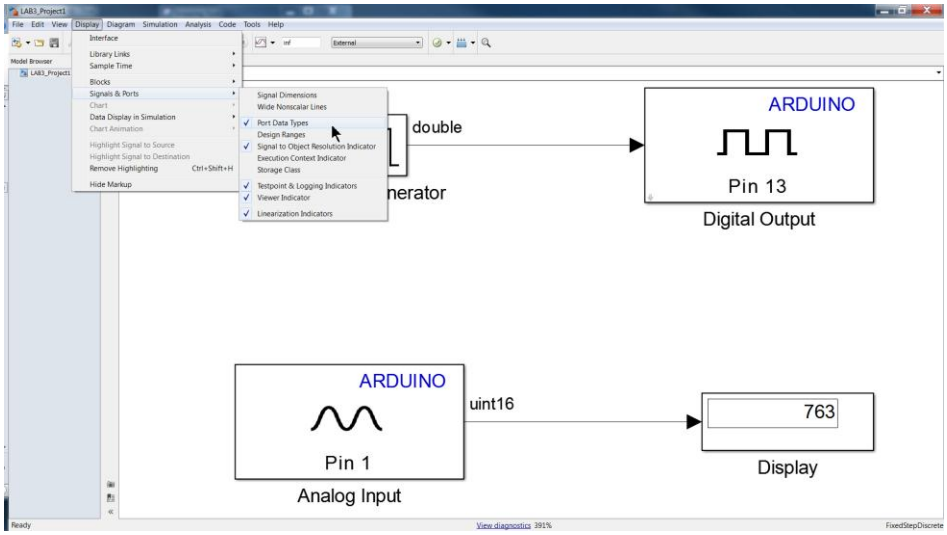
As you change the pot, you should see the value change between 0 and 1023. Before continuing, you must terminate External mode by clicking on the **Stop** button  .

Demo III.1: Show external mode with the **Analog Input** block and a **Display** block.

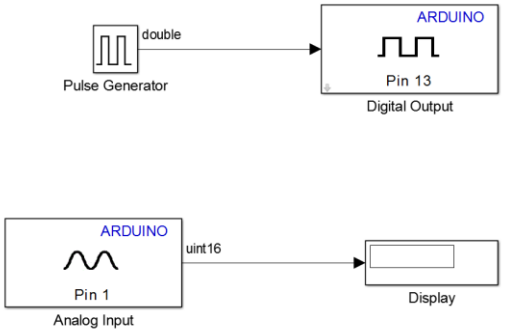
1. Data Types

To keep things simple, we will do everything in double precision floating-point calculations. For non-ECE types, this basically means that we do not need to worry about number systems, data representations, or loss of accuracy due to fixed bit lengths. The downside is that using double precision calculations takes more time than single precision calculations, and a lot more time than integer calculations. If we are looking to make our models run fast, then double precision is not the way to go. Also note that the Arduino does not have native double precision calculations so the calculations are done in software (which is why it is slow). However, using double precision makes the implementation easy for us, which is the goal of using Simulink models in the first place.

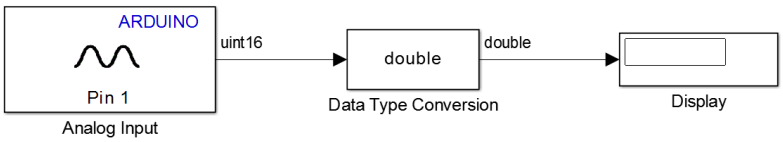
What we will do here is use a facility to determine the data type of a signal, and if it is not a double precision data type, convert it to double. In advanced courses, we will look at making calculations with data types other than double precision. In our model, select **Display**, **Signals & Ports**, and then **Port Data Types** to turn on the diagnostic:



When you return to the model, type **Ctrl-D** to update the diagram. Since we enabled **Port Data Types**, the data types should be displayed on the signal lines:



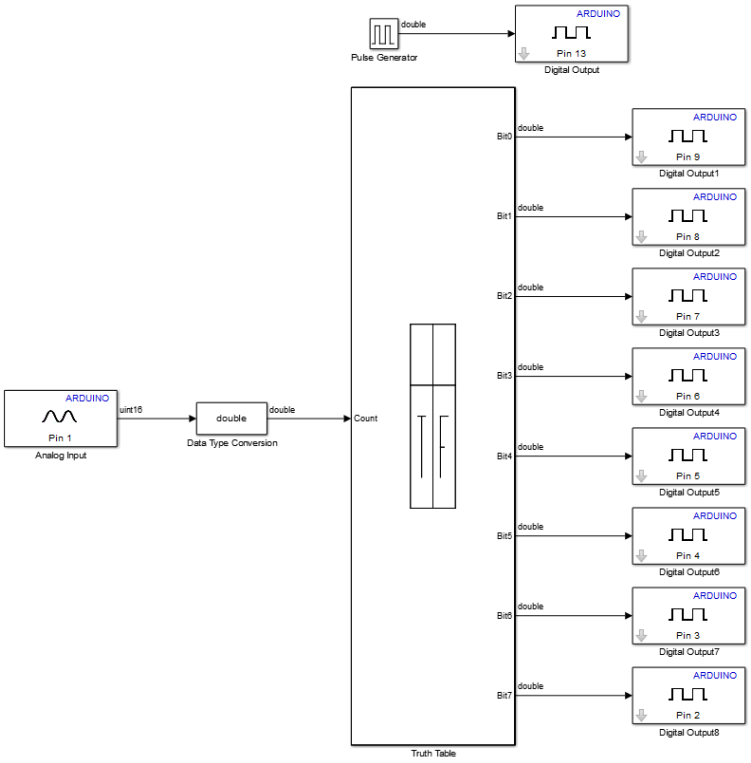
We see that the output of the **Analog Input** block is an unsigned 16-bit integer (**uint16**).If we perform calculations between a uint16 and another data type, double for example, the result can be unpredictable. To eliminate this possibility, we will convert the data type to double precision. Place a **Data Type Conversion** block from the **Simulink / Commonly Used Blocks** library in your model as shown below. Open the block and specify the **Output data type** as double. When you update the diagram, the output data type of the block should be **Double**:



The uint16 and the double signals will have the same numerical values, just different representations. Now that that we have the value as a double precision representation, we can use it in calculations with other signals that also have a double precision representation.

2. Analog Voltmeter

We are now ready to use the Analog Input in a circuit that does something more significant. In this case, we will make an analog volt meter where the output voltage is displayed by lighting up a number of LED's in proportion to the voltage. We will start with the up/down counter we demonstrated in Demo II.7. Open the model and save it as Lab3_Project3. Delete the Stateflow chart and add the analog input we used in the previous model. Do not forget the Data Type Conversion block. Change the **Fixed-step size** to 0.1 seconds and specify the **Discrete** solver. Display port data types to make sure that all non-double data types are converted to double:



The truth table was set up to check for an integer count between 0 and 7. We now have a number between 0 and 1023. Modify the Truth Table as shown:

Block: LAB3_Project3/Truth Table

File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8	D9
1	Count is zero.	Count>=64	T	T	T	T	T	T	T	T	-
2	Count is one.	Count>=192	F	T	T	T	T	T	T	T	-
3	Count is two.	Count>=320	F	F	T	T	T	T	T	T	-
4	Count is three.	Count>=448	F	F	F	T	T	T	T	T	-
5	Count is four.	Count>=576	F	F	F	F	T	T	T	T	-
6	Count is five.	Count>=704	F	F	F	F	F	T	T	T	-
7	Count is six.	Count>=832	F	F	F	F	F	F	T	T	-
8	Count is seven.	Count>=960	F	F	F	F	F	F	F	T	-
		Actions: Specify a row from the Action Table	1	2	3	4	5	6	7	8	9

Action Table

#	Description	Action
1	Light up bit 0.	Bit0=1;Bit1=0;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
2	Light up Bit 1.	Bit0=1;Bit1=1;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
3	Light up Bit 2.	Bit0=1;Bit1=1;Bit2=1;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
4	Light up Bit 3.	Bit0=1;Bit1=1;Bit2=1;Bit3=1;Bit4=0;Bit5=0;Bit6=0;Bit7=0;
5	Light up Bit 4.	Bit0=1;Bit1=1;Bit2=1;Bit3=1;Bit4=1;Bit5=0;Bit6=0;Bit7=0;
6	Light up Bit 5.	Bit0=1;Bit1=1;Bit2=1;Bit3=1;Bit4=1;Bit5=1;Bit6=0;Bit7=0;
7	Light up Bit 6.	Bit0=1;Bit1=1;Bit2=1;Bit3=1;Bit4=1;Bit5=1;Bit6=1;Bit7=0;
8	Light up Bit 7.	Bit0=1;Bit1=1;Bit2=1;Bit3=1;Bit4=1;Bit5=1;Bit6=1;Bit7=1;
9	Light up nothing!	Bit0=0;Bit1=0;Bit2=0;Bit3=0;Bit4=0;Bit5=0;Bit6=0;Bit7=0;

Note that we added a new row and that the default transition chooses that row. The lights should illuminate linearly as the input voltage varies.

Demo III.2: Demonstrate the working linear analog volt meter.

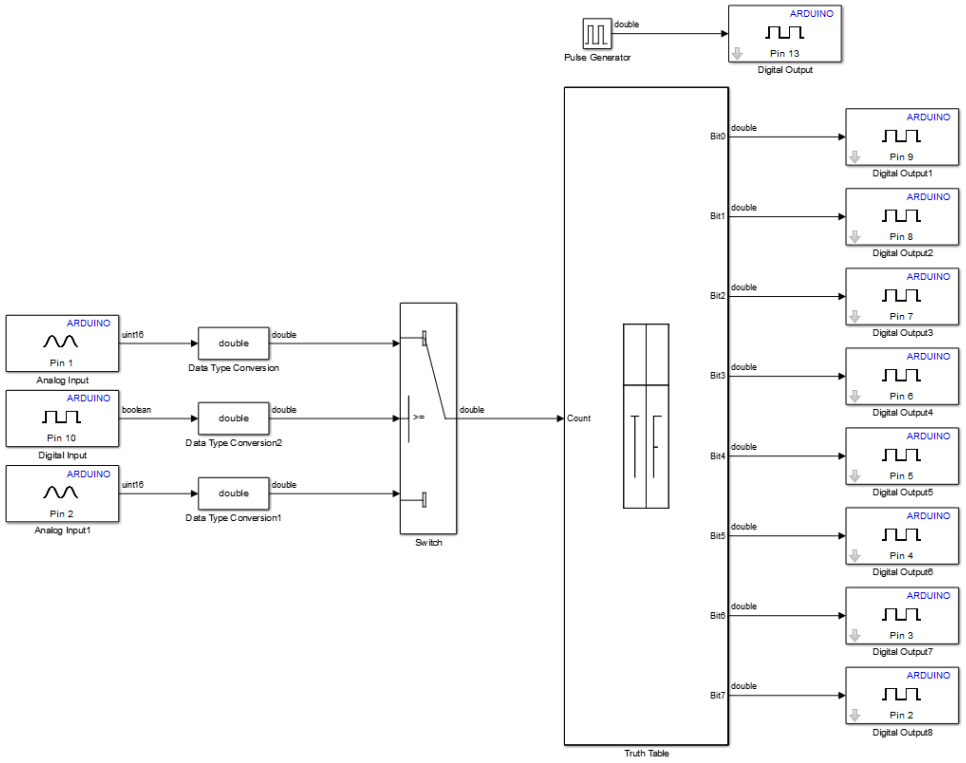
Exercise III.1: Modify the linear voltmeter of Demo III.2 to use a base-2 logarithmic scale. That is, the next LED should only light up when the size of the signal doubles. This makes the bar graph very sensitive for small signals and less sensitive for larger signals. You can do this in two ways: (1) Use the Simulink **Math Function**

block located in the **Simulink / Math Operations** library and remember that $Log_2(x) = Log_{10}(x)/Log_{10}(2)$. Or (2) modify the Truth Table.

3. Sampling Time

In the previous example we set the sampling rate of the Analog input block to 0.1 seconds. If we have multiple Analog input blocks in our model, each can be set to a different sampling rate. This is beneficial because different sensors will output signals at different frequencies. Since we can specify different sampling rates, we can sample slowly varying signals less often (say a temperature sensor) and higher frequencies (say an accelerometer) more often. We can also partition our model so that different portions execute at different speeds allowing us to make more efficient use of the microcontroller.

To illustrate the sample time, we will create a model that uses a switch to select between two different analog inputs. One samples every 100 ms, the other every 2 seconds. The model is shown below:

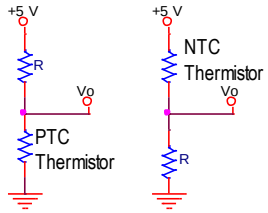


Both analog inputs are connected to the same potentiometer. When the slow input is chosen, there is a large delay between when the input changes and the LEDs respond. Although this model is not all that profound, you should always choose an appropriate sampling rate for the signal you are measuring.

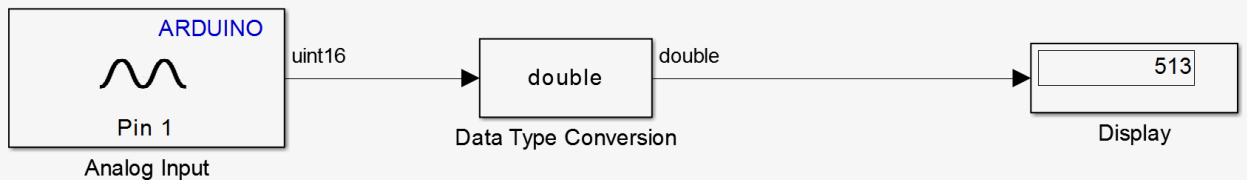
Demo III.3: Demonstrate the working linear analog volt meter with different sampling rates.

B. Temperature Sensor

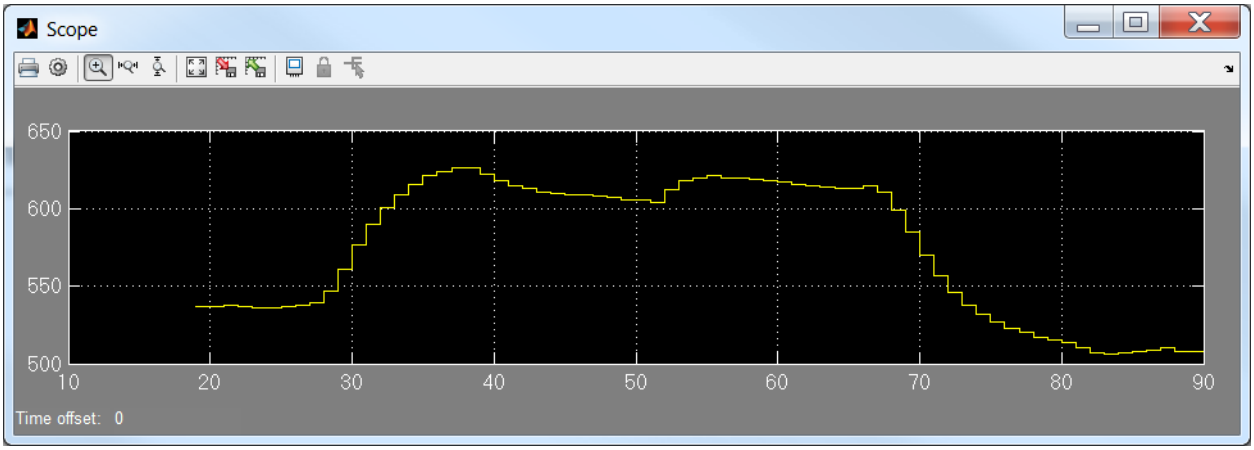
As an example of a sensor, we will measure the room's temperature using a thermistor. A thermistor is a resistor who's resistance varies with temperature. Thermistor's come with positive or negative temperature coefficients (PTC or NTC). With a positive temperature coefficient a thermistor's resistance will increase with temperature. For a negative coefficient, the resistance goes down as temperature increases. For this example, we want the sensor voltage to increase as temperature increases. We can use one of the two circuits below:



Use the appropriate circuit for the thermistor you have. We will use a resistor that is equal in value to the value of the resistance of the thermistor. My thermistors have a room temperature resistance of 10 kΩ, so I will use a resistance of 10 k. We will first use Simulink in External mode so that we can see the output of the sensor. We will use the model shown below:



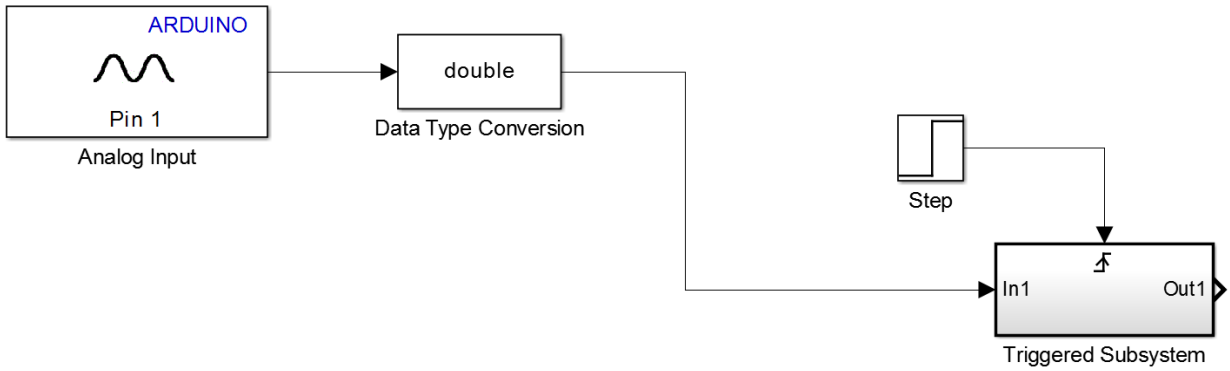
The sample time was set to one second as was the fixed step size. What we see is that the temperature sensor does change its voltage with temperature, but it does not vary greatly and certainly does not use the full range of 0 to 5 V available to the Analog Input block. They was to fix this to get the most accuracy would be to use an analog circuit to convert the sensor voltage to 0 to 5 volts and take advantage of the 10-bit accuracy of analog to digital converter. We will do this in a later lab. Here, however, we will fix the problem with Simulink. Before we do this, we will generate a plot of the values produced by sensor. We will let the sensor sit at room temperature for a while, and then warm it up by heating it with a hair dryer. We will use a Simulink Scope block to generate a plot.



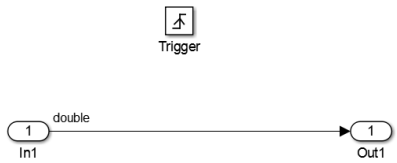
The sensor only uses a small portion of the converter range. To fix this problem, we will do the following. We will sample the sensor when we first start our controller. We will use this value as a baseline. We will then subtract this value from the real-time temperature measurement to generate a value that is the difference in temperature from the baseline. We will then amplify this difference so that it lights up all the LEDs over a specified range. When cold, no LEDs will illuminate. When we heat the sensor with a hair dryer, eventually all the lights will illuminate.

1. Triggered Subsystems

We can sample the sensor at start-up by using a triggered subsystem. By default, all blocks in our models execute once every fixed time step. We can vary the execution of specific portions of our model by placing them in triggered subsystems. The blocks within the subsystem only execute when the subsystem is triggered. Place a **Triggered Subsystem** and **STEP** in your model as shown:

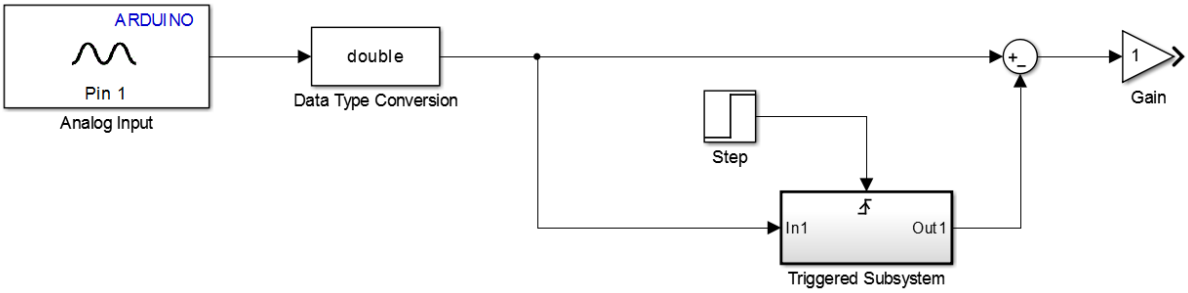


Note that by default, this subsystem is triggered by a rising edge. We will use a step function to trigger this subsystem. Note that the default settings of the Step is that the output is initially zero, and then increases to a value of one at a time of one second (a unit step at a time of on be second). Since the Step function stays at 1 for ever, the **Triggered Subsystem** is only triggered once. We will not place anything inside the **Triggered Subsystem**:

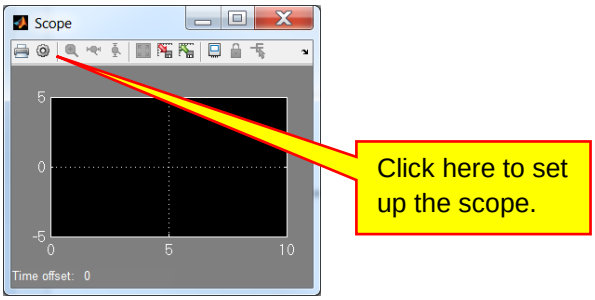


When the subsystem triggers, the output will be set to the input at the moment of the trigger. After that, the output of the Triggered Subsystem will hold the output value until it receives another trigger. Since a step provides one positive edge, the subsystem will hold the value until we restart the microcontroller.

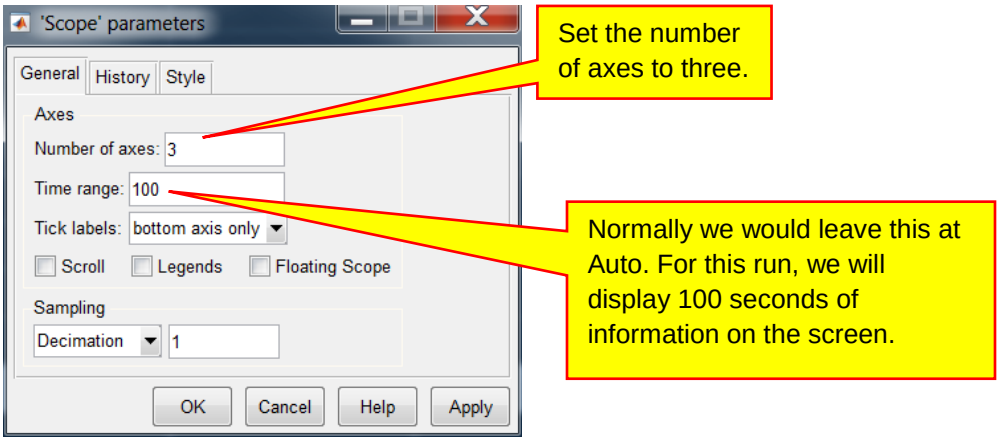
Next, we will subtract the reference from measured signal to create a signal that starts at zero and then increases as the temperature goes up. Finally, we will add a gain block to amplify the difference so that we can light up all of the LEDs:



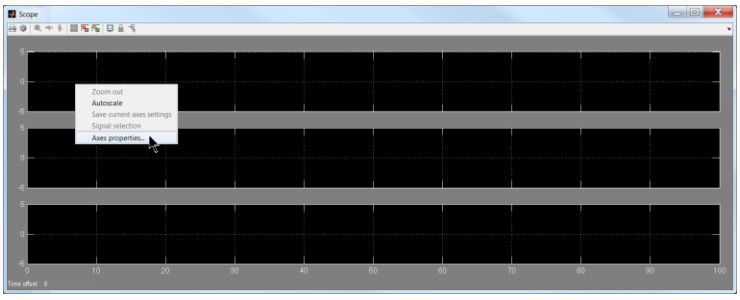
To test this sensor signal conditioning model, we will plot the three signals on a scope using External mode. Place a **Scope** in your model (**Simulink** / **Sinks** library) in your model. (Note - don't place a Floating scope.) Double-click on the scope to open the scope window:



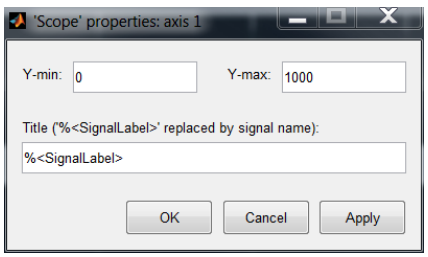
Click on the **Parameters** button to set up the properties of the scope. Make the changes as shown:



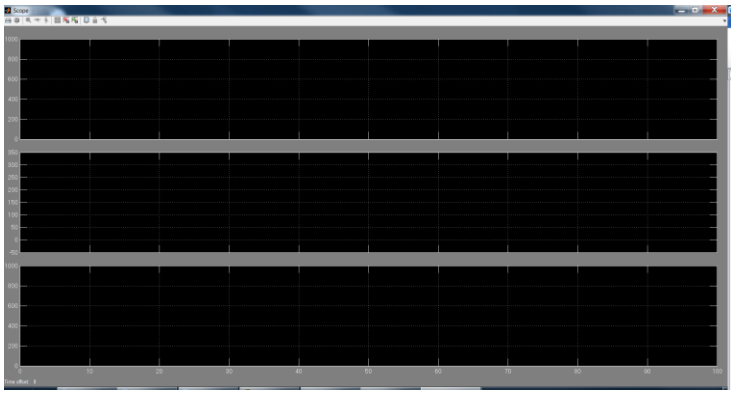
Click the **OK** button and return to the scope Window. Enlarge the window as shown. Right-click on the top scope plot and select **Axes Properties**:



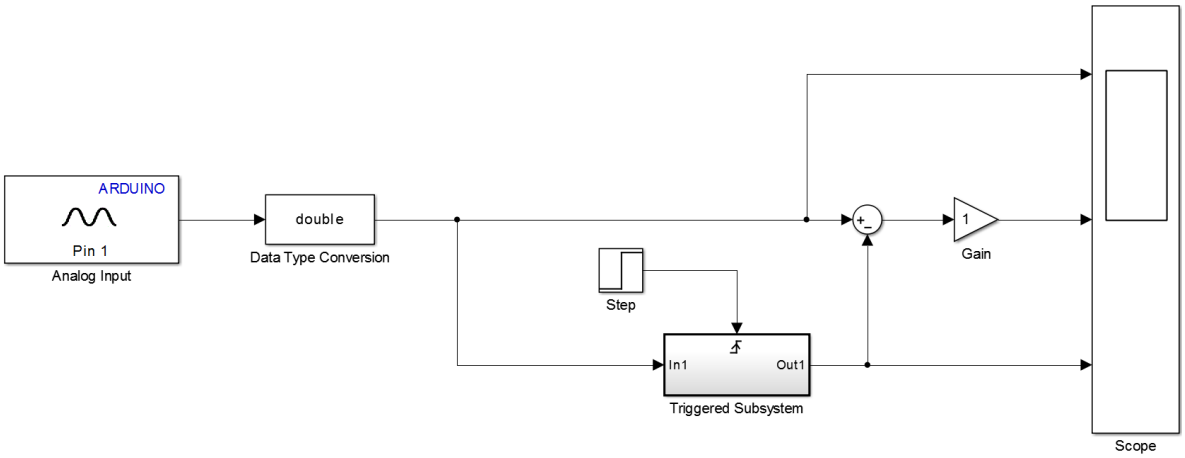
A dialog box that allows us to change the y-axis range for the top plot will open. Change the range to 0 to 1000:



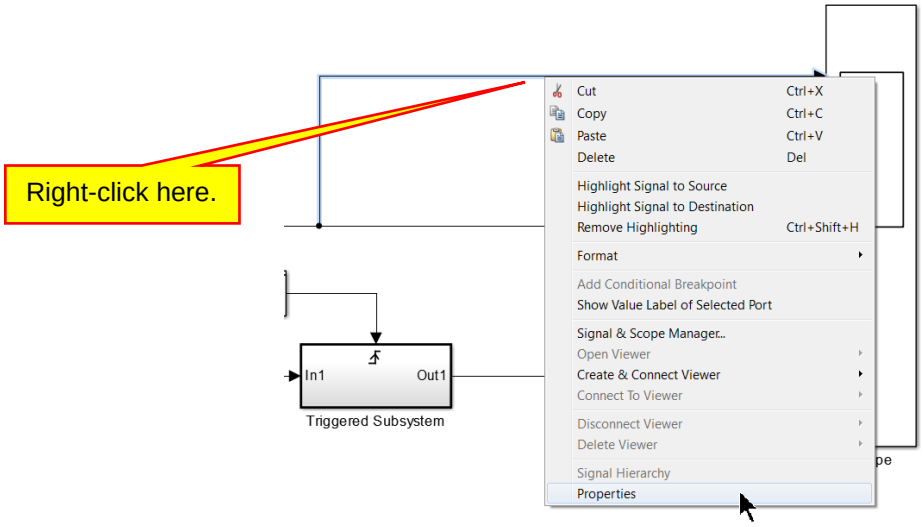
Click the OK button to accept the changes. Change the range of the middle plot to -50 to 350 and the bottom plot to 0 to 1000:



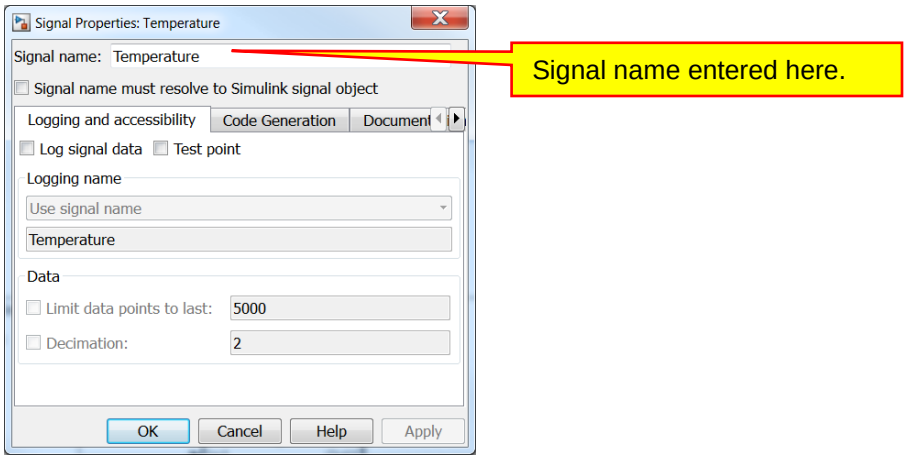
When you return to the Simulink model, you will notice that the scope now has three inputs. Resize the scope and connect it as shown:



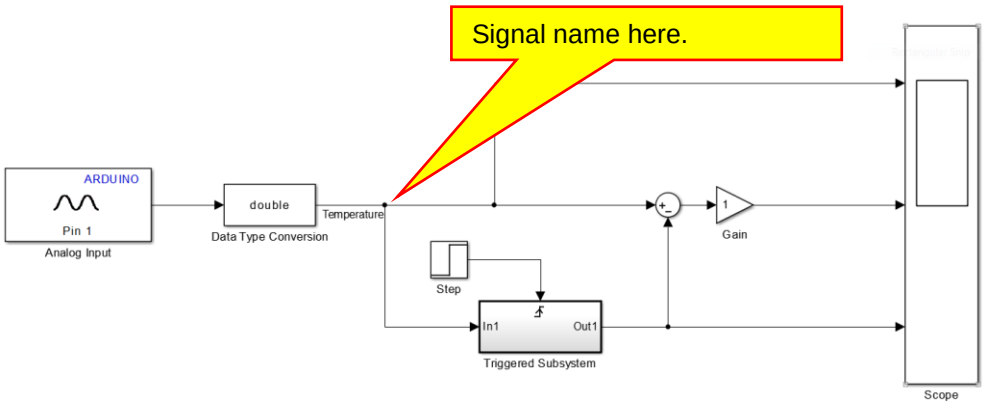
Next, we will label the signal wires which will flow through to the plot as well. Right click on the signal wire as shown and select **Properties** from the menu:



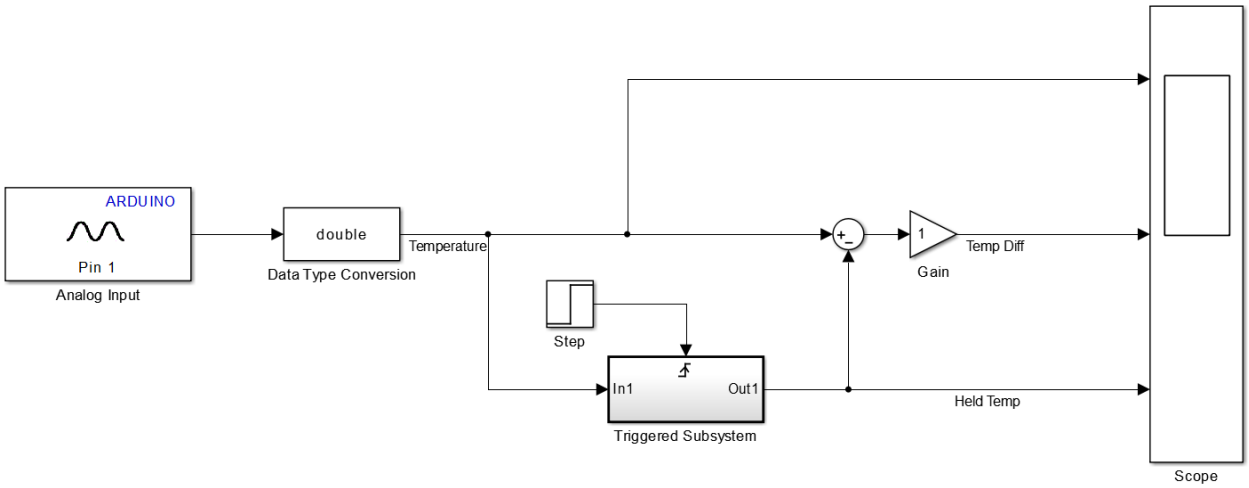
After you select **Properties**, the dialog box below will appear. Fill in the **Signal name** as shown:



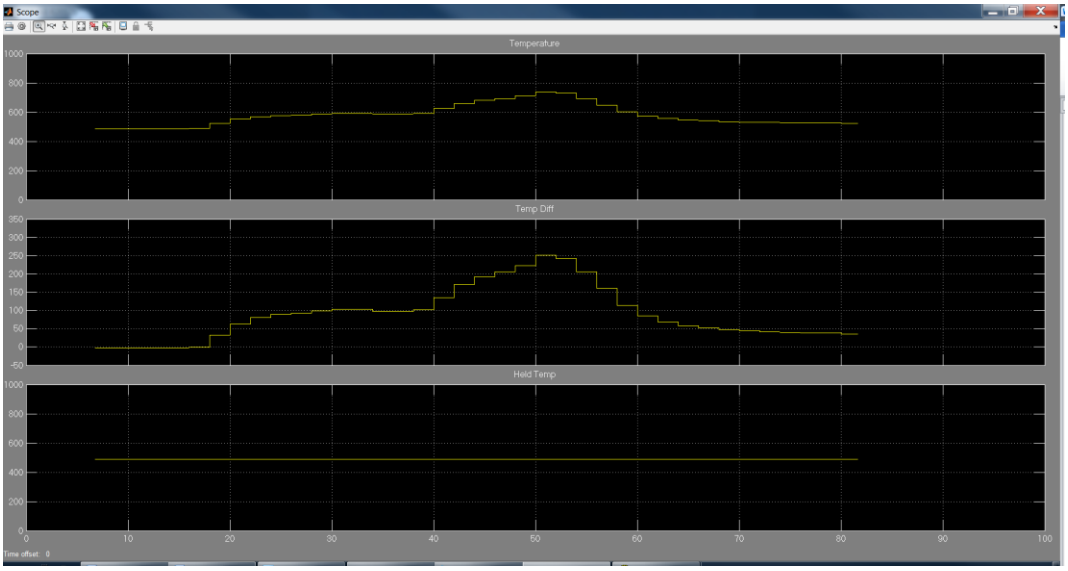
When you click the OK button, you will notice that the **Signal name** appears attached to a wire:



Name the output of the gain block “Temp Diff” and the output of the Triggered Subsystem as Held Temp:

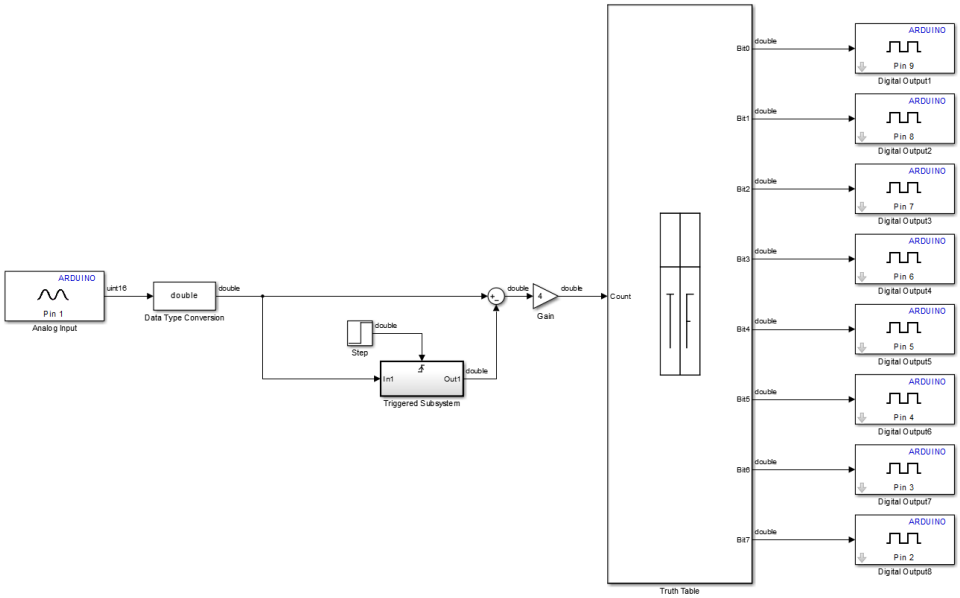


Make sure that External mode is enabled, save the model, run the model on the target, and connect to the target. If you heat up the sensor with a hair dryer and then let it cool down, you will see a plot similar to the one below:



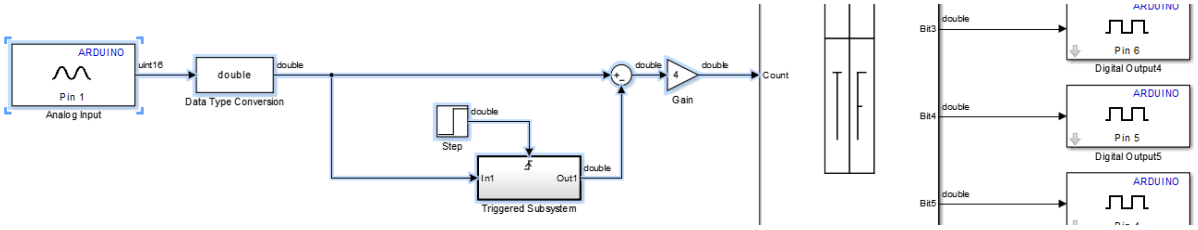
2. Temperature Meter

We are now ready to connect our sensor to the Analog Voltmeter developed earlier:

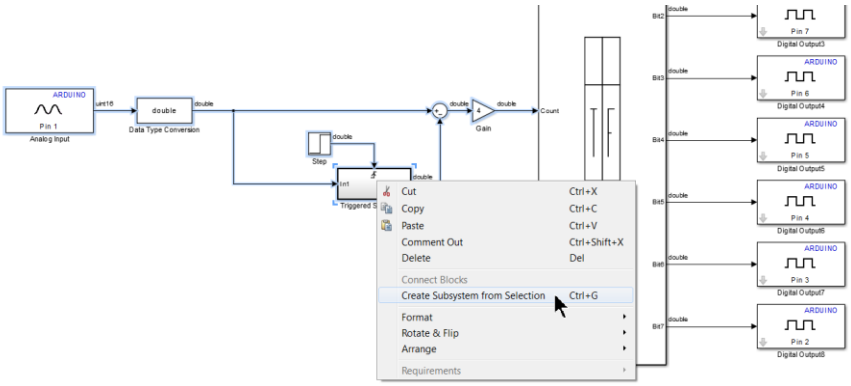


Adjust the value of the gain block so that all 8 LEDs will illuminate when the sensor reads a maximum temperature that your heating source provides. (Your gain will not be 4.)

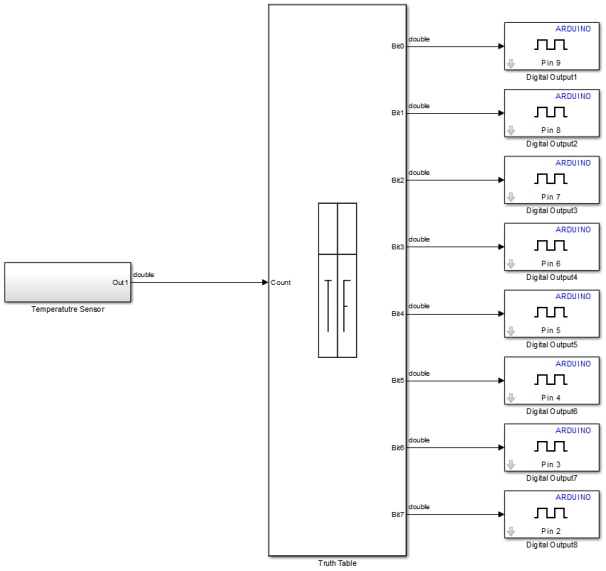
The last thing we will do is clean up our model by placing all of the blocks that condition the sensor signal in their own subsystem. Select all of the blocks used in conditioning the temperature sensor signal:



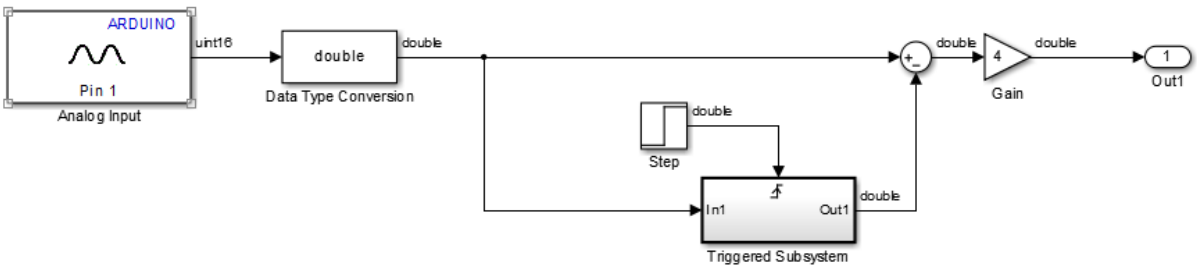
Next, right-click on the selection and select **Create Subsystem from Selection** from the menus:



The selected blocks will be placed in a subsystem. Rename the subsystem “Temperature Sensor:”



If you look inside the Temperature Sensor subsystem, you will see the selected blocks:



Demo III.4: Demonstrate the working linear temperature meter.

Exercise III.2: Modify the temperature meter so that when it zeros out, 4 LED’s are illuminated. Then when the sensor is warmed up to the maximum temperature, all 8 LED’s are illuminated. If the sensor becomes colder than the reference, less than 4 LEDs are illuminated. This is essentially a sensor that indicates temperatures both above and below the reference point.

Exercise III.3: Modify the temperature meter of Demo III.4 to use a pushbutton to sample the temperature rather than the step source. This will allow the user to zero the meter at any time, rather than zeroing the meter automatically at power-up.

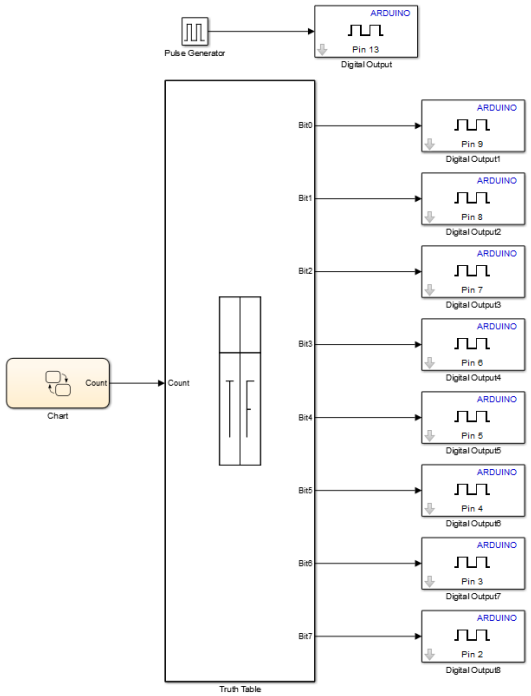
Exercise III.4: Modify the temperature meter of Demo III.4 to use a pushbutton and the step source to sample the temperature. This will automatically zero the meter at power up as well as allow the user to zero the meter at any time.

C. Lookup Tables

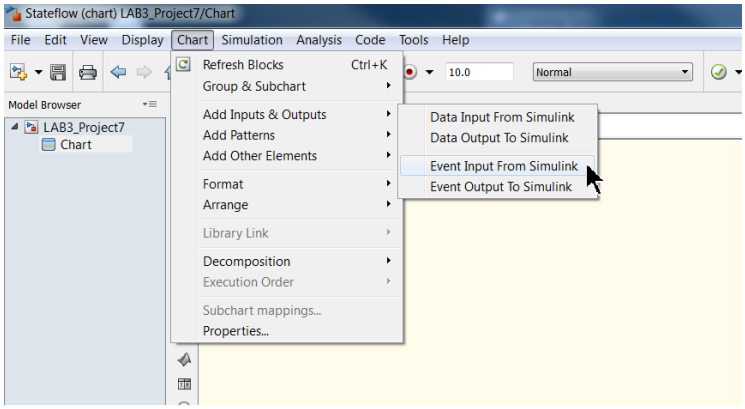
The last project we will do in this lab is to make a variable speed up-down counter whose counting frequency varies linearly with the analog input from 1 Hz to 10 Hz. We will do this by making a ramp and square wave whose frequency is dependent on the input voltage. The only problem is that out analog input voltage is interpreted as a number between 0 and 1023 and we want to map this to a number between 1 and 10. We could do this with an algebraic function using the Simulink **Fcn**, **MATLAB Function**, or **Polynomial** blocks or other Simulink primitive blocks. Instead, we will use a lookup table to map the input values to the needed output range.

1. Stateflow Charts with Events

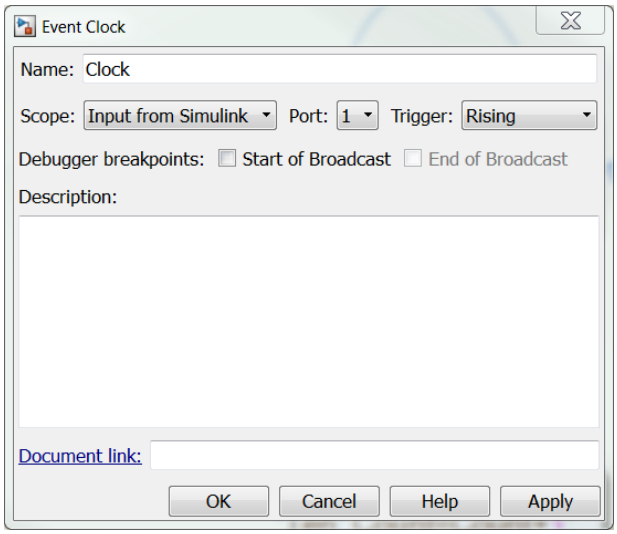
We will start with the model of the up-down counter that was implemented with a state flow chart and truth table (Demo II.7). Open the model, and save it as Lab3_Project7. This model uses a SF chart that executes at the fixed-step rate, 0.5 seconds in this model. This is because the chart has no event triggers. We will add an event so that we can control how often the chart executes.



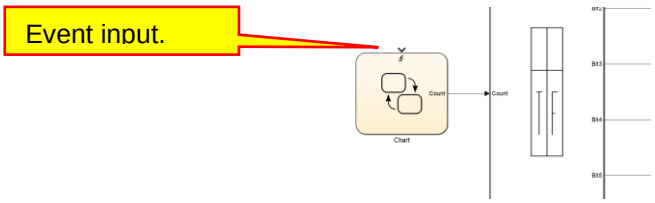
Open the chart and then select **Chart**, **Add Inputs & Outputs**, and then **Event Input From Simulink**.



Name the event **Clock**:

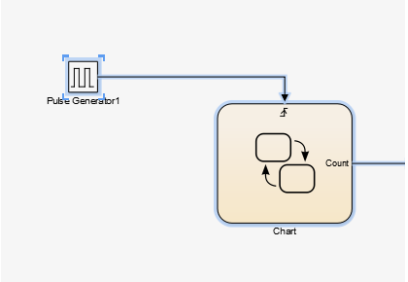


Click the **OK** button to return to the Stateflow chart. Then switch to the Simulink model. You will notice that the Stateflow chart now has an event input at the top:



We also notice that the event is positive-edge triggered. Every time the input has a rising edge, the Stateflow chart will execute. We will provide a square-wave input for Clock event, and change its frequency to see how the model reacts.

Since we will be trying frequencies between 1 and 10 Hz, we need a smaller fixed time step. Change the Fixed-time step to 0.01 seconds (**Simulation**, **Model Configuration Parameters**). Also specify a discreet solver. Finally, add a Pulse Generator and specify the **Amplitude** as 1, **Period** as 1, and **Pulse Width** as 50 percent. Connect the **Pulse Generator** to the **Event** input of the chart:



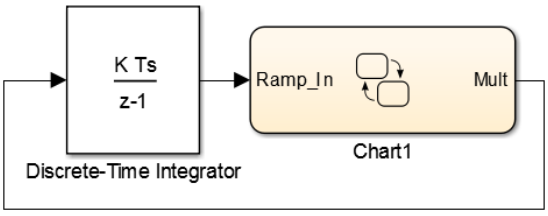
Set up your model to run in External mode and run the model. While the model is running on the Arduino, and Simulink is connected to the Arduino, you can open the pulse generator and change its values. For the step fixed step size, we can specify a period of 0.06 seconds or larger. As you change the period, you should see the up-down counter vary its counting rate. External mode allows us to vary parameters like these in real-time. If you were doing this for a feed-back controller, this would be called parameter tuning, and Simulink in External mode will allow you to tune the controller in real-time.

Demo III.5: Demonstrate using External mode to change the counting frequency in real-time between a period of 0.1 and 1 second.

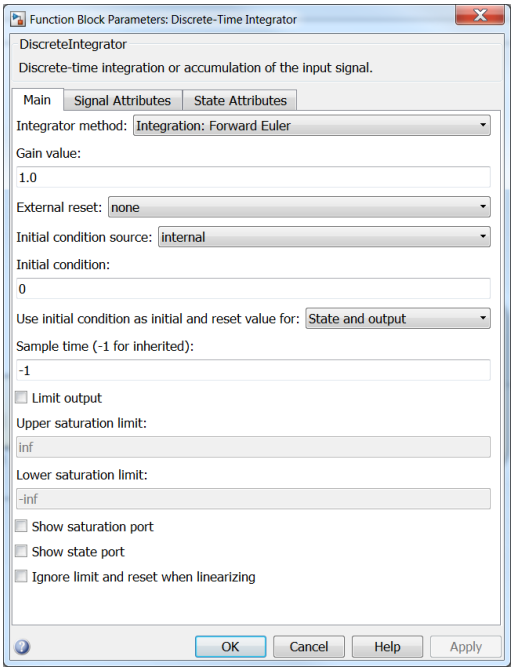
2. Integrators, Timing, and Signal Generators

There are several ways to generate timing functions in Simulink: (1) We can use the Stateflow temporal commands (**after**, **before**, **at**, **every**, etc). (2) we can use some of the Simulink sources and specify a clock time for the sources. (3) We can use Stateflow and count a number of events that occur. Instead, we will use math and on the way generate a triangle wave and a square wave.

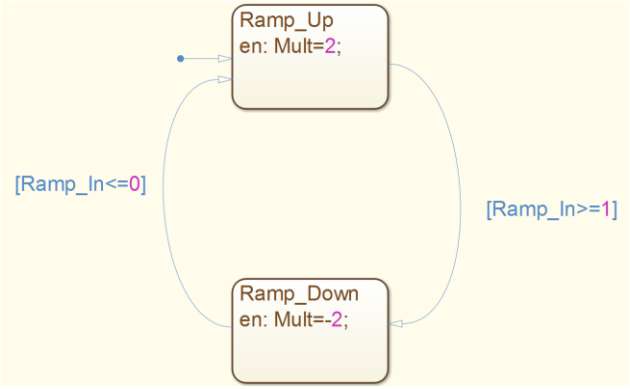
The integral of a constant is a ramp, $\int_{t_o}^{t} a \cdot dt = a(t - t_o)$. This is the equation of a straight line of slope *a*. If *a* is a positive constant, then the ramp increases at a rate of *a*. If *a* is a negative constant, then the ramp increases at a rate of *a* (or -|a|). We will start by creating a ramp of slope 2 by integrating a constant value of 2. The ramp will start at zero and then integrate positive. When the ramp hits a value of 1, we will change the input of the integrator to -2 so that the integrator ramps down. When the ramps hits zero, we will again change the input to +2 and ramp up once again. This process will repeat indefinitely. For the integrator we will use a discrete integrator because we are using discrete solver. Note that integration can take a lot of computation time. We chose a discrete solver to make the models execute as fast as possible and high accuracy on a ramp generator is not necessary. We will use a Stateflow chart to toggle to track the ramp and output a 1 and -1 as needed. The added blocks are shown below:



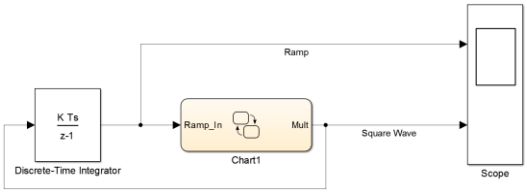
The settings of the integrator are shown below:

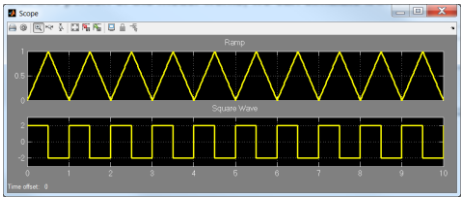


The Gain is set to one which means that the output is just the time integral of the input, and the initial condition is zero which means that the ramp starts at zero. The Stateflow chart is shown below:

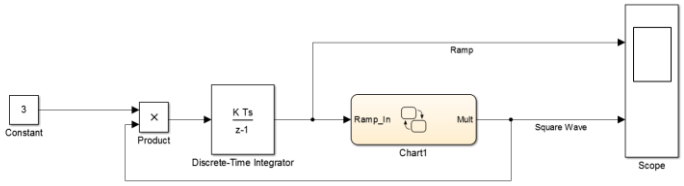


The reason we have a slope of two instead of 1 is because we wish to generate a frequency of 1 Hz, which means that the ramp has to integrate up to 1 in ½ second and integrate back down to zero in ½ second. To verify that this portion of the model works, we can simulate it in Simulink as a separate model:

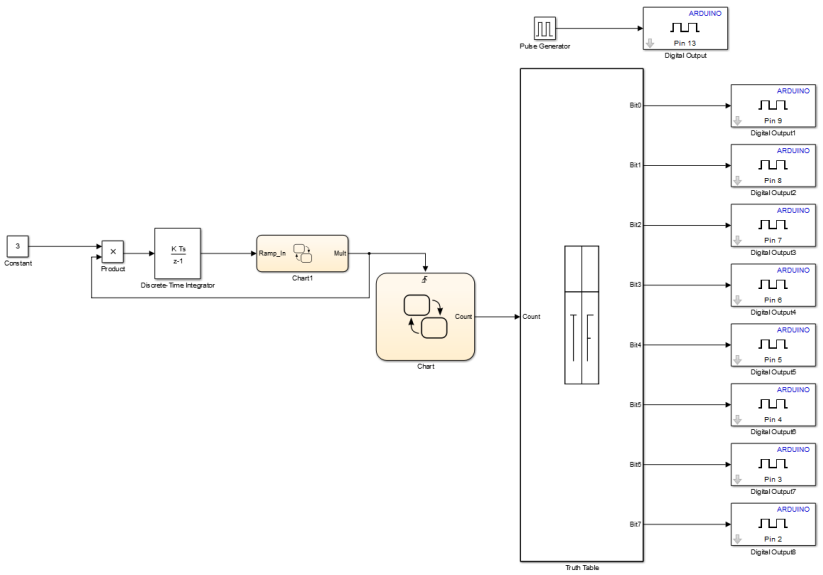




To make a variable frequency signal generator, all we need to do is vary the input to the integrator as shown:



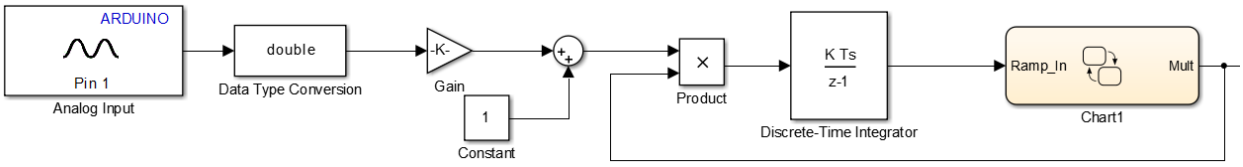
As a check, we will place this portion into the full model, and then verify its operation using external mode, where we can change the value of the constant in real-time:

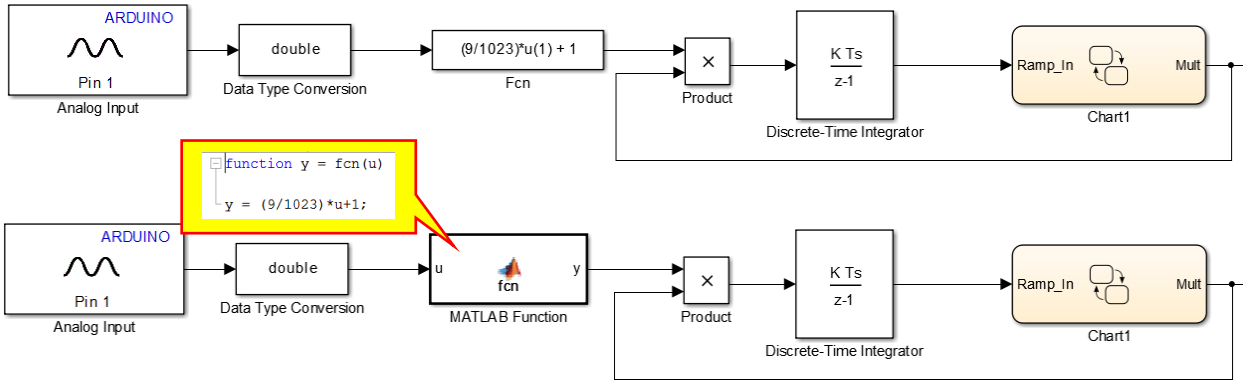


Demo III.6: Demonstrate that the up-down counter can vary between a counting frequency of 1 and 10 Hz by using external mode to change the value of the constant in real-time.

3. Lookup Tables

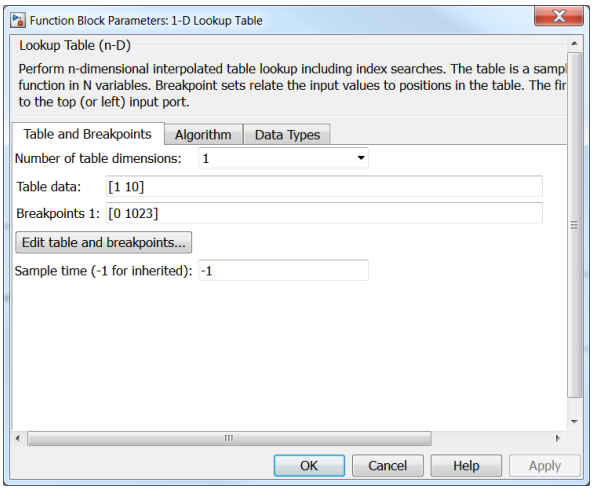
The last thing we need to do is to replace the constant block by a set of blocks that reads the analog input and scales that value to a range of 1 to 10. If you recall, the Analog Input block reads an analog voltage between 0 and 5 V and outputs a number from 0 to 1023. We need to scale this output linearly to a range of 1 to 10. We could do this by implementing the function $y = \frac{9}{1023}x + 1$ in many different ways in Simulink. Examples are shown below:

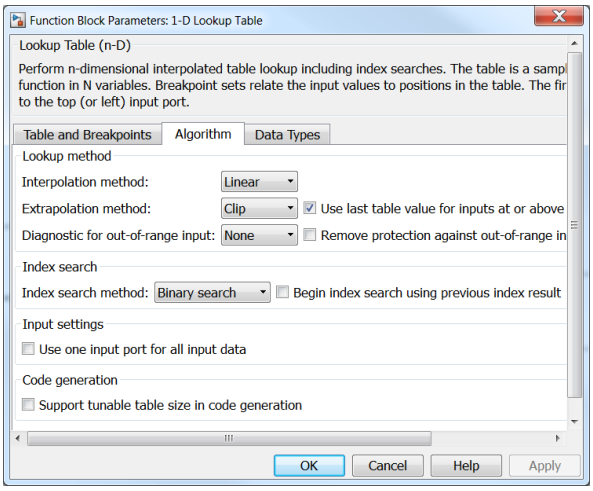




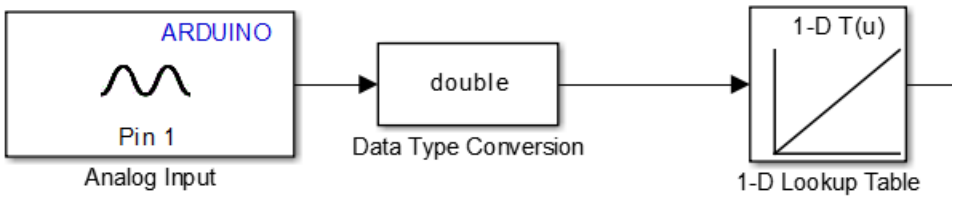
Instead of using one of these methods, we will use a lookup table. With a lookup table, a set of input and output points are specified. The points are connected by straight-line segments. If the input matches a point, then the output is set to the output coordinate of the point. If the input is between points, then linear interpolation is used to approximate the output between the points. (Basically, a point on the straight line connecting the two points.) Lookup tables are typically used to model measured data. For example, a torque curve measured from an engine. The lookup table would contain data of output torque versus input rpm. Lookup tables can contain many points of just a few, depending on the complexity of the function you want to model, or the amount of measured data. Most lookup tables are used to represent data that is highly nonlinear or very hard to represent mathematically. Our lookup table is straight forward. We have two points connected by a straight line. The input ranges from 0 to 1023 (represented by a vector [0 1023]), and the corresponding output is 1 to 10 (represented by a vector [1 10]).

Place a 1-d lookup table in your model. Double-click on the lookup table and fill it in a shown:



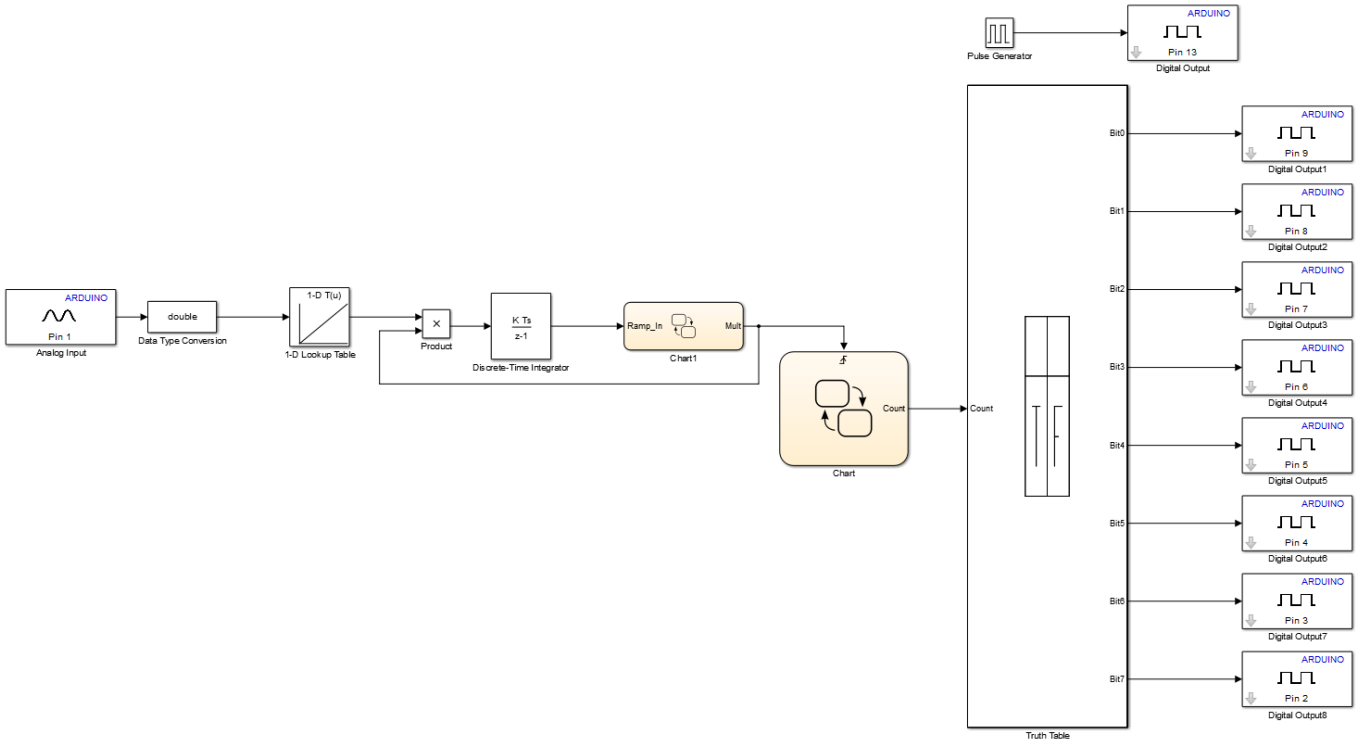


The **Table data** is the vector of output values and the **Breakpoints** is the vector of input values. If you think of these vectors as coordinates on the x-y plane with y being the output and x being the input, then we are specifying two points, (0,1) and (1023,10). If the input is between 0 and 1023, then the output is calculated along the straight line connecting the two points. The **Interpolation method** specifies that if the input is between the two points, a linear approximation (straight line) should be used to calculate the value of the output. The **Extrapolation method** specifies what should happen if the input is outside the range of 0 and 1023. The chosen options specify that if we are outside the range, the endpoint values should be used. When you click the OK button, a plot of the data points will be displayed on the block. Notice that it is a straight line:



4. Variable Speed Ring Counter

We are now ready to test the entire model. Note that the sample time of the Analog Input should be set to -1 so that it uses the fixed-set time of 0.01 seconds:



Demo III.7: Demonstrate the operation of the variable-speed up-down counter.

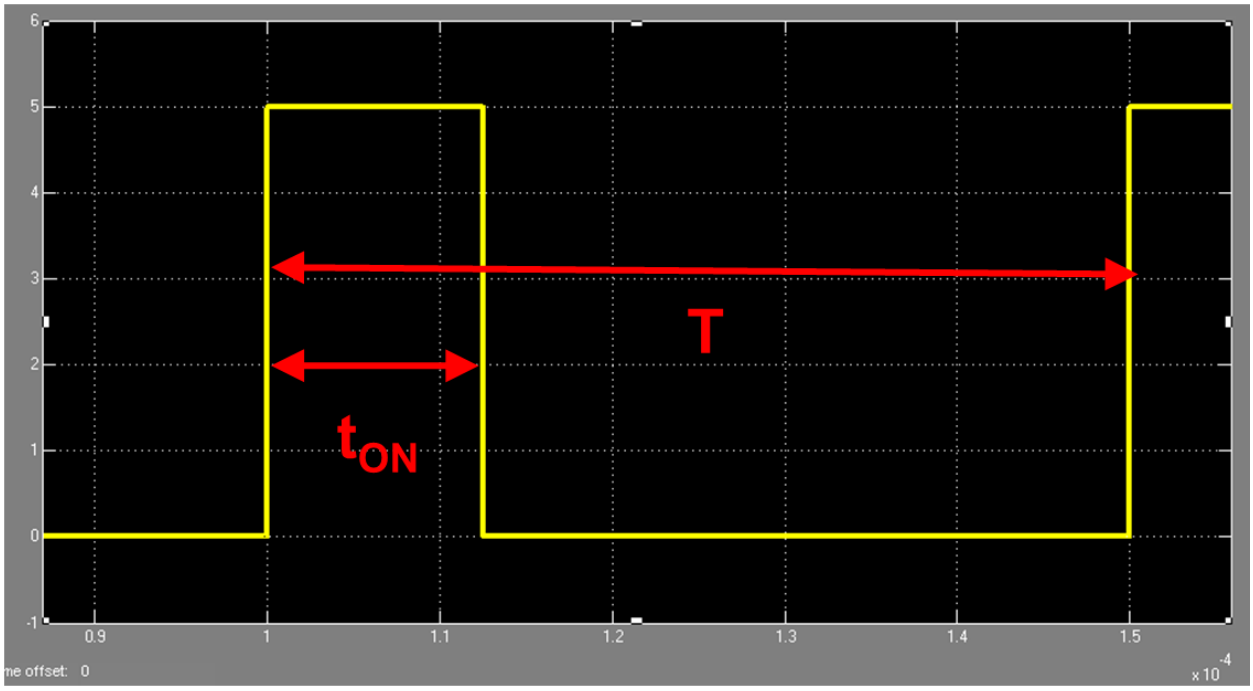
Exercise III.5: Create a model that switches between three modes of operation for the up-down counter. When you push and release a push-button, the mode of operation changes. The three modes are: (1) constant 1 Hz frequency, (2) constant 5 Hz Frequency, and (3) Variable frequency between 1 and 10 Hz based on an analog input.

Lab IV
Analog Output (PWM)

The Arduino uses pulse-width modulation (PWM) to generate analog output voltages. This method requires a low-pass filter to convert the PWM waveform to an analog output. The benefits of this method are that it is inexpensive to implement on a microcontroller, so there are many PWM outputs on the Arduino (twelve possible PWM outputs using pins 2 to 13). Another benefit is that many devices can be directly controlled with a PWM output (provided they use a high current driver) and do not require a low-pass filter. These devices are typically slow and respond to the average of the PWM waveform. Examples are motors that have a large inertia and light bulbs. The down side of using PWM outputs is that if you want a true analog output voltage, the cutoff frequency of the low-pass filter must be well below PWM frequency, placing an upper limit on the frequency of the desired analog output. Note that the PWM output frequency of the Arduino is approximately 490 Hz.

A. Pulse-Width Modulation (PWM)

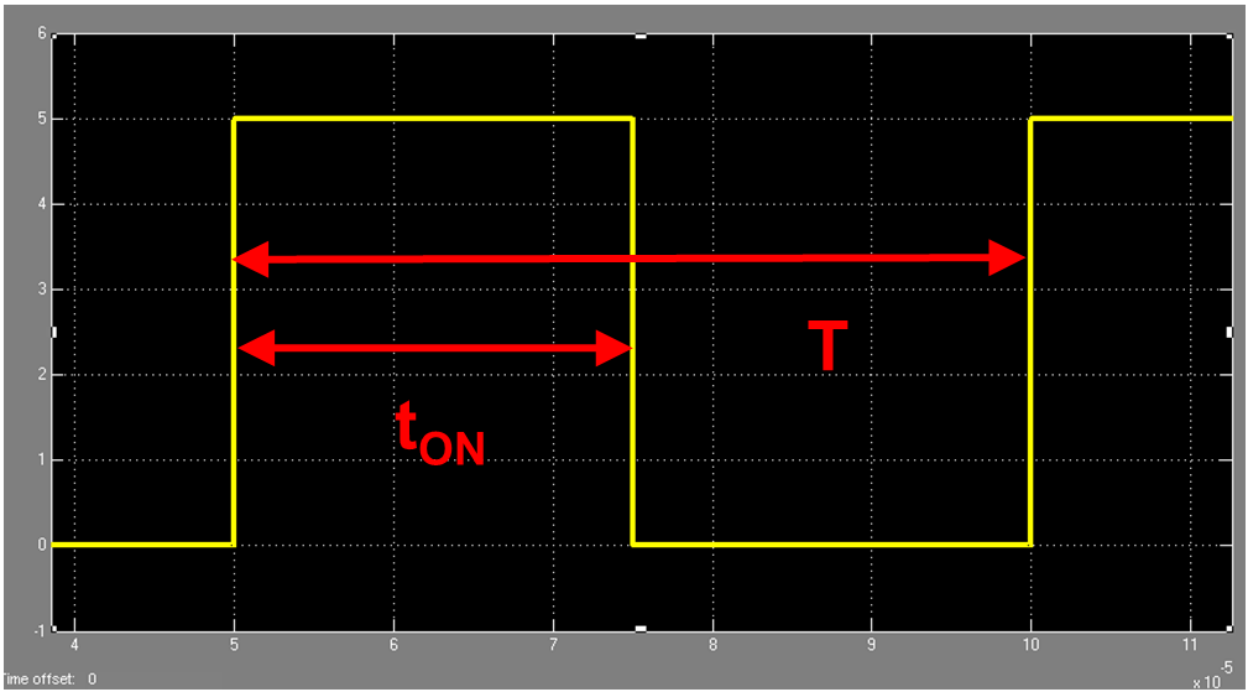
A pulse-width-modulated waveform is a square wave of constant amplitude and period (or frequency), with varying on – time (t_{on}). In the waveform below, the period is 50 µs, the amplitude is 5 V, and the on-time is 12.5 µs:



The the average voltage of the waveform is:

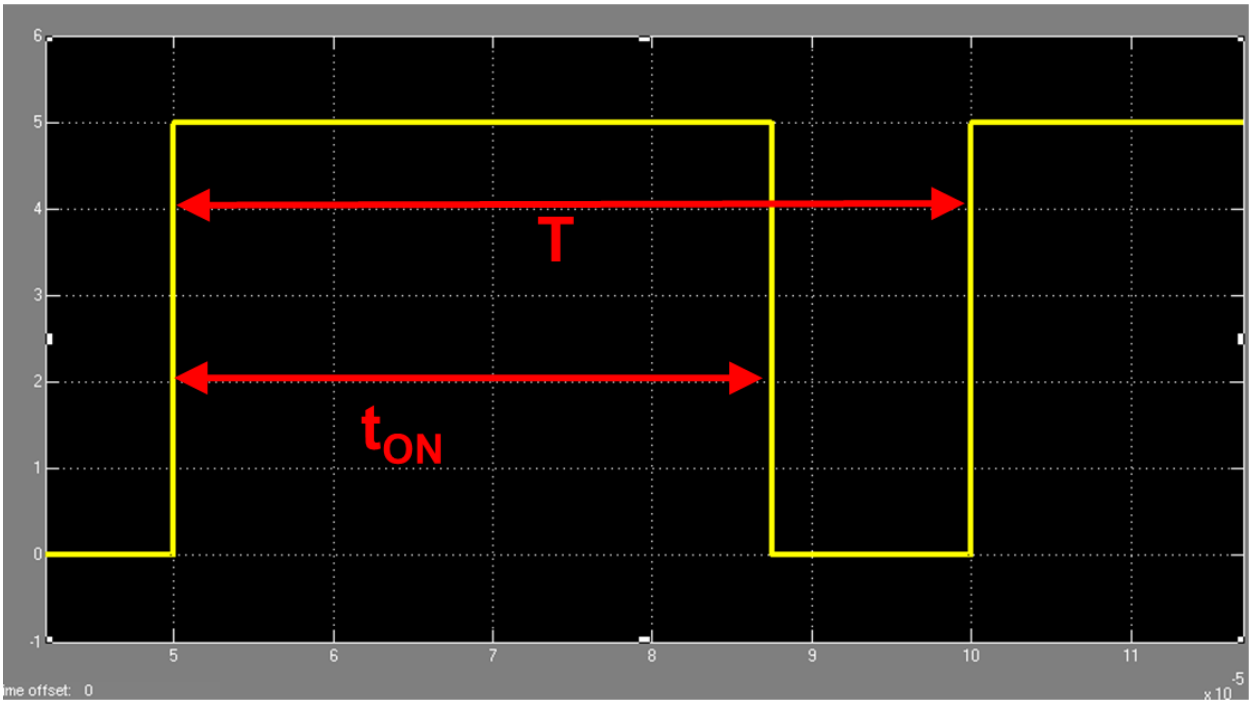
$$V_{avg} = \int_{T} V(t) \cdot dt = \int_{0}^{t_{on}} 5\,V \cdot dt + \int_{t_{on}}^{T} 0\,V \cdot dt = 5\,V\left(\frac{t_{on}}{T}\right) = 5\,V\left(\frac{12.5\,\mu s}{50\,\mu s}\right) = 1.25\,V$$

Note that the average output voltage is the “analog” output voltage we are after. For a PWM waveform, we keep the amplitude and period constant. The waveform below shows the output for an increased on-time, yielding a different average output voltage:



$$V_{avg} = 5\ V\left(\frac{t_{on}}{T}\right) = 5\ V\left(\frac{25\ \mu s}{50\ \mu s}\right) = 2.5\ V$$

And just to be totally redundant:

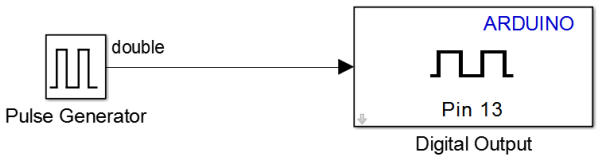


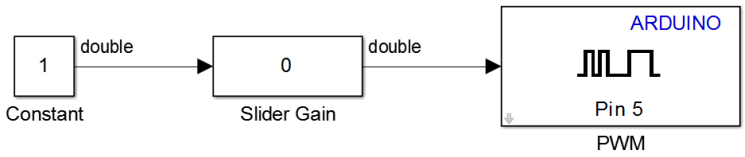
$$V_{avg} = 5\ V\left(\frac{t_{on}}{T}\right) = 5\ V\left(\frac{37.5\ \mu s}{50\ \mu s}\right) = 3.75\ V$$

Thus we see that we can obtain an analog output voltage between 0 and 5 V by varying the on time (ton) from 0 to T. Also note that $\left(\frac{t_{on}}{T}\right)$ is referred to as the duty cycle (D) and is expressed as a percent. As the duty cycle varies between 0 and 100 %, the output varies between 0 and 5 V.

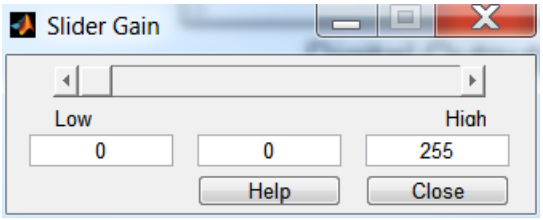
1. Arduino PWM output Block

We will observe one of the Arduino PWM outputs to get a better idea of how PWM waveforms behave. Note that the frequency of the Arduino PWM output is approximately 490 Hz, corresponding to a period of 2.04 ms. The PWM block requires an input between 0 and 255. Zero corresponds to a duty cycle of 0 % and 255 corresponds to a duty cycle of 100 %. We will use the model shown:

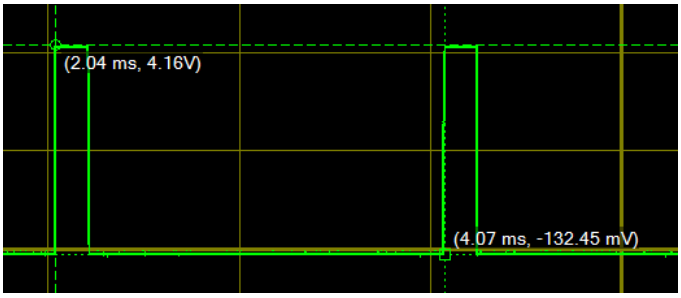




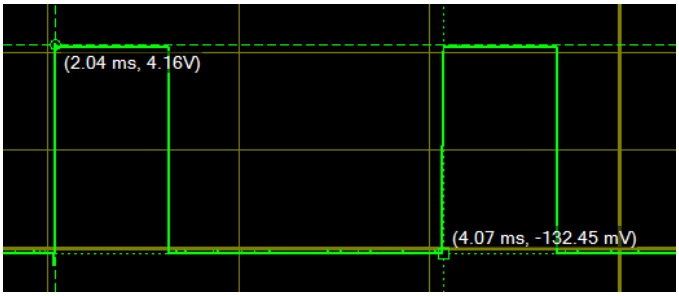
There is a new block called a slider gain which is located in the **Simulink** / **Math Operations** library. Double-click on the block and set the limits to 0 and 255:



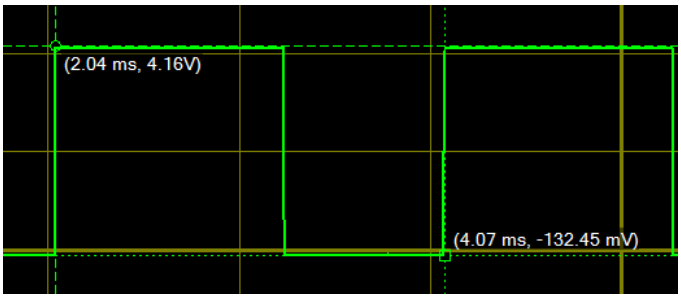
In external mode, this block will allow us to easily vary the duty cycle. Setup the model to run in external mode and observe the PWM output with an oscilloscope. Shown below are a few waveforms:



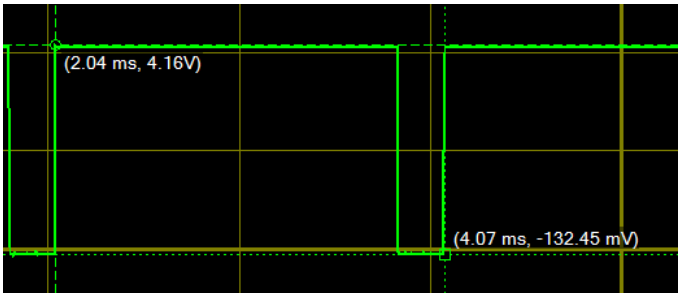
- PWM input = 22



- PWM input = 75



- PWM input = 150

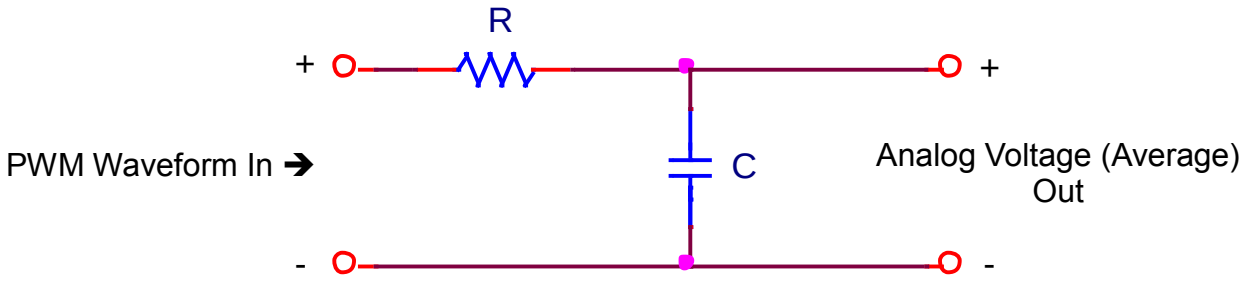


- PWM input = 225

Demo IV.1: Demonstrate the PWM output of the Arduino on the oscilloscope.

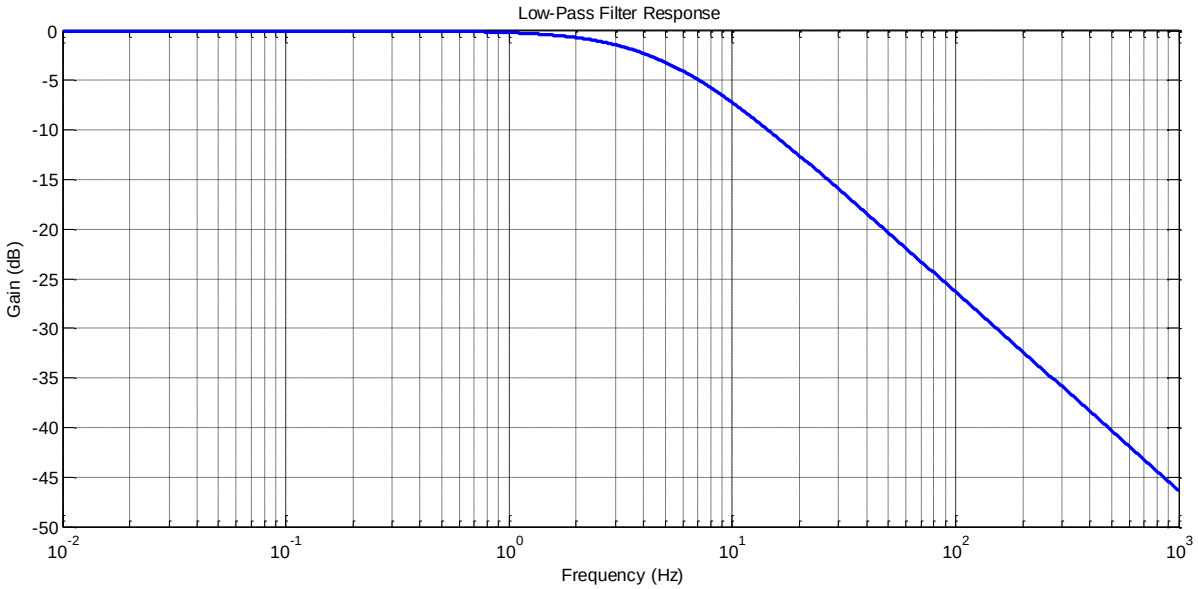
2. Low-Pass Filter

We can extract the average (DC) value of the waveform using a low-pass filter. For this course, we will use the first order low-pass filter shown below:



This filter has a cut-off frequency of $F_C = {}^{1}/_{2\pi RC}$. Frequencies below the cutoff frequency are passed through the filter. Frequencies above the cutoff frequency are attenuated (reduced in amplitude). The more a frequency is above the filter cutoff frequency, the more it is attenuated. Our PWM waveform has a frequency of 490 Hz. This means that its fundamental frequency if 490 Hz and it contains higher order frequencies. We would like our filter to have a cutoff frequency of at least a factor of 100 below the PWM frequency, so we will design our

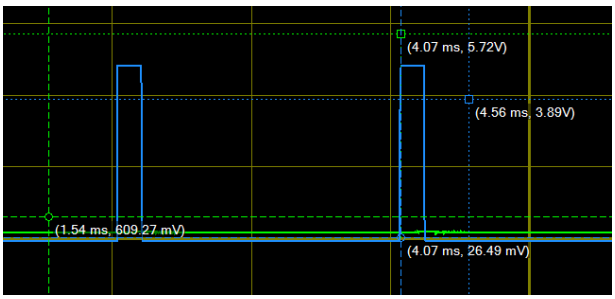
filter for a cutoff frequency of 5 Hz. Choose values of R and C that you have available in the lab to make a filter with a 5 Hz cutoff frequency. A frequency response plot for this filter is shown below:



The plot is in Decibels. The gain plotted is, $Gain_{dB} = 20log_{10}\left(\frac{V_{out}}{V_{in}}\right)$. Note that a gain of 0 dB corresponds to $\left(\frac{V_{out}}{V_{in}}\right) = 1$, a gain of -20 dB corresponds to $\left(\frac{V_{out}}{V_{in}}\right) = 0.1$, and a -40 dB corresponds to a gain of $\left(\frac{V_{out}}{V_{in}}\right) = 0.01$. From the plot we see that DC, which is a frequency of 0, will go through the filter attenuated. Low frequencies in the range 0.01 Hz to 1 Hz are also passed through attenuated. At the cutoff frequency of 5 Hz, the gain is –3 dB, or $\left(\frac{V_{out}}{V_{in}}\right) = 0.707$. At frequencies above the cutoff, the gain decreases rapidly (- 20 dB per decade).

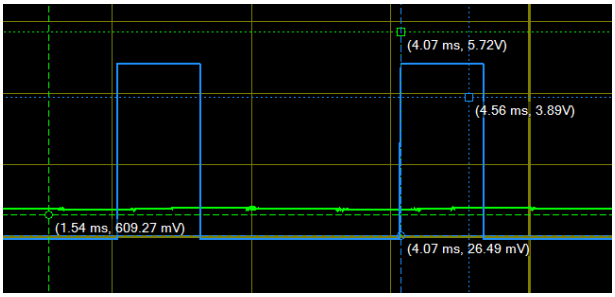
3. DC Output

We will now use a scope to observe the filter input and output. The input should be the PWM waveform of varying duty cycle. The output should be a DC voltage. Screen captures below for varying inputs to the PWM output block are shown:



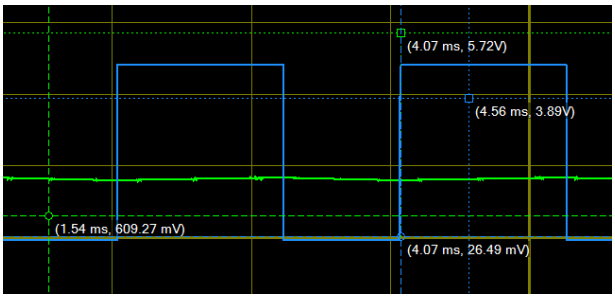
DC Voltage.

- PWM input = 22



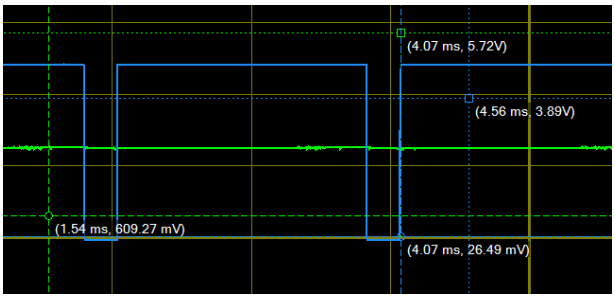
DC Voltage.

- PWM input = 75



DC Voltage.

- PWM input = 150



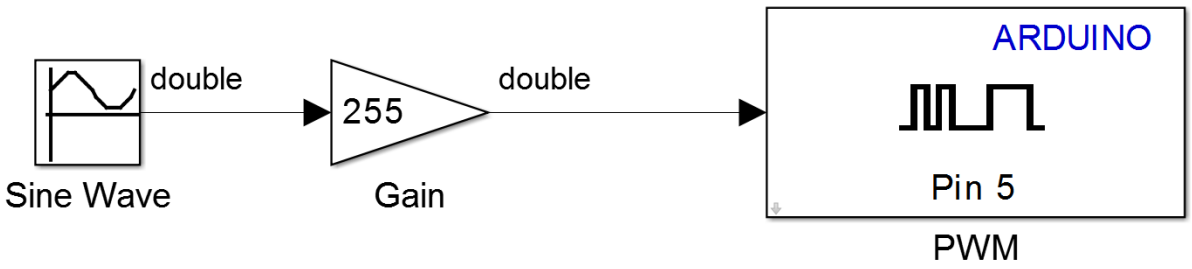
DC Voltage.

- PWM input = 225

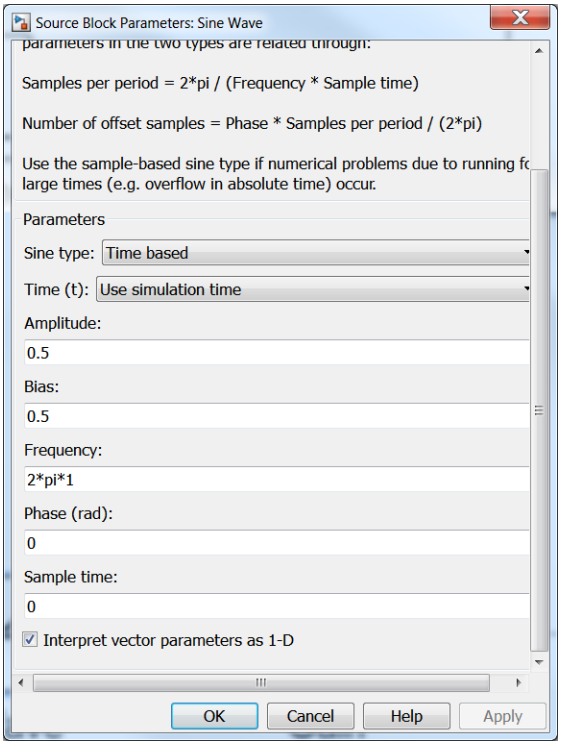
Demo IV.2: Demonstrate the PWM and DC output of the Arduino on the oscilloscope.

4. Sine wave Output

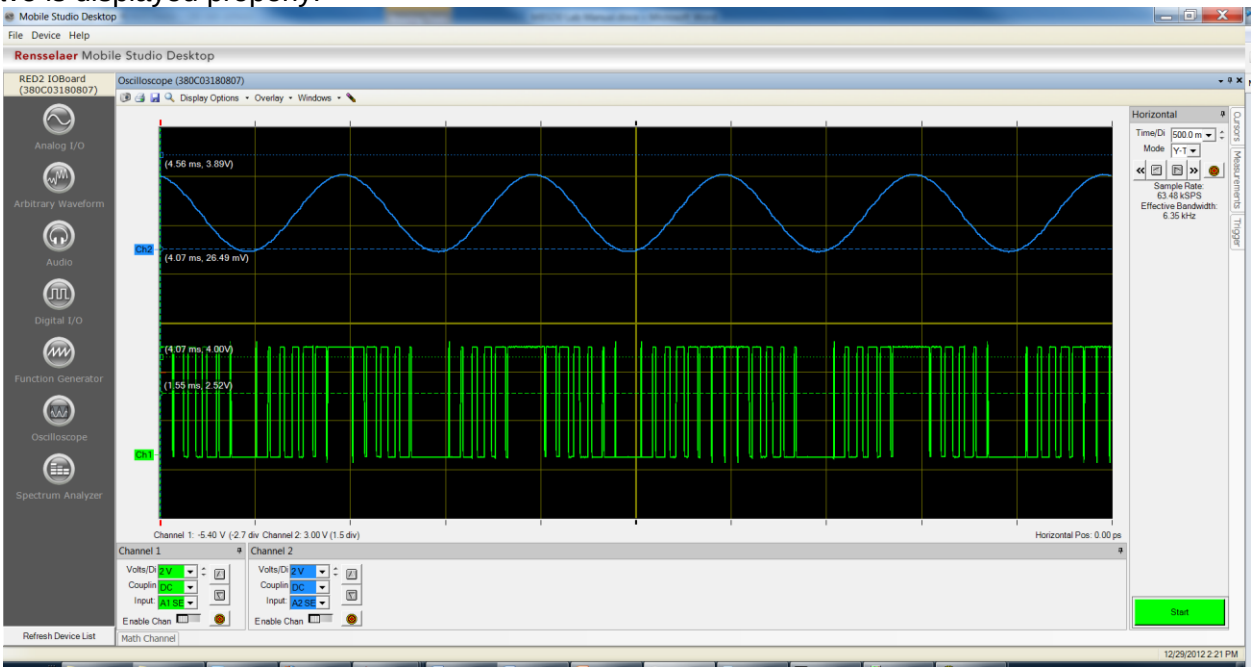
We would now like to see some analog waveforms produced by the Arduino and the low-pass filter. From the frequency plot of the filter, we should be able to pass frequencies of 1 Hz or less. We will use the model below:



Note that the **Sine Wave** block is set to produce a waveform from 0 to 1 at a frequency of 1 Hz. Note that the frequency you specify in the block is in rad/sec, so you must convert Frequency in Hertz to frequency in radians per second by the equation, $\omega=2\pi F$. The parameters for the Sine Wave block are:



The output waveforms are shown below. The PWM frequency is too fast to see it on a time scale where the sine wave is displayed properly:



Demo IV.3: Demonstrate the PWM and sine wave output of the Arduino on the oscilloscope.

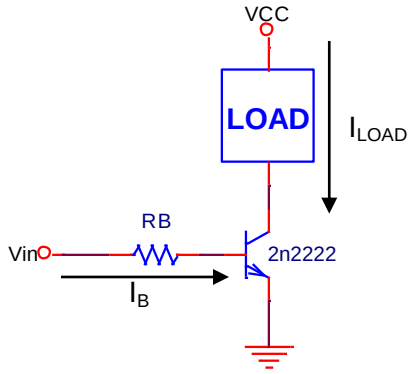
Exercise IV.1: Create a model that uses a push-button to switch between the following waveforms: (1) A sine wave at 1 Hz, (2) a sine wave at 0.5 Hz, (3) a 1 Hz sawtooth, and (4) a 1 Hz triangle wave. All waveforms are 0 to 1 V in amplitude.

B. Variable Brightness Light Bulb

As our next example we will drive a slightly larger load. PWM modulation can be used to drive a large variety of loads, as long as that load is slow enough to respond to the average rather than waveform. The requirement is that the PWM frequency be high enough so that the load cannot follow the waveform. An example is an old incandescent light bulb in your house (if you still even have one). These bulbs are powered from the 60 Hz line. You cannot see the light

intensity vary at a 60 Hz rate for two reasons. One is that your eye does not respond light changing that fast. The second is that the amount of light is dependent on the temperature of the filament, and the filament has a large enough thermal mass that the temperature does not vary at a 60 Hz.

We will demonstrate driving a 5 V incandescent light from our PWM output. The intensity will vary with the Duty cycle, and you eye will never notice that the current and voltage are actually a 490 Hz pulsed waveform. We cannot just hook the light bulb up to the PWM output because the bulb requires 60 mA of current and the Arduino cannot drive that large of a load. Although this is not a huge amount of current, we will show how to drive larger loads using an npn bipolar junction transistor (BJT) as a switch, or a high current driver. The circuit below is used to drive currents up to a few amps:



You can use any transistor that meets your current requirement. Typically the load current is specified by the manufacturer of the load you want to drive. In this case the load current is 60 mA, so we need to find a transistor that can handle 60 mA. A second parameter that we will need to know is called the current gain, and is referred to as h_{fe}, or h_{FE}. h_{fe} is referred to as the small signal current while h_{FE} is called the DC current gain. Portions of the ON Semiconductor PN2222A data sheet are shown below:

ON Semiconductor™

Amplifier Transistors
NPN Silicon

P2N2222A

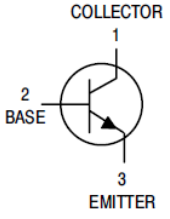
MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Collector–Emitter Voltage	V_{CEO}	40	Vdc
Collector–Base Voltage	V_{CBO}	75	Vdc
Emitter–Base Voltage	V_{EBO}	6.0	Vdc
Collector Current — Continuous	I_C	600	mAdc
Total Device Dissipation @ T_A = 25°C Derate above 25°C	P_D	625 5.0	mW mW/°C
Total Device Dissipation @ T_C = 25°C Derate above 25°C	P_D	1.5 12	Watts mW/°C
Operating and Storage Junction Temperature Range	T_J, T_{stg}	–55 to +150	°C

CASE 29–11, STYLE 17
TO–92 (TO–226AA)

THERMAL CHARACTERISTICS

Characteristic	Symbol	Max	Unit
Thermal Resistance, Junction to Ambient	$R_{\theta JA}$	200	°C/W
Thermal Resistance, Junction to Case	$R_{\theta JC}$	83.3	°C/W



We would like a specification for the value of h_{FE} for a collector current near 60 mA. A portion of the data sheet is sown below:

P2N2222A

ELECTRICAL CHARACTERISTICS (T_A = 25°C unless otherwise noted) (Continued)

Characteristic	Symbol	Min	Max	Unit
ON CHARACTERISTICS				
DC Current Gain	h_{FE}			—
(I_C = 0.1 mAdc, V_{CE} = 10 Vdc)		35	—	
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc)		50	—	
(I_C = 10 mAdc, V_{CE} = 10 Vdc)		75	—	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, T_A = –55°C)		35	—	
(I_C = 150 mAdc, V_{CE} = 10 Vdc)(1)		100	300	
(I_C = 150 mAdc, V_{CE} = 1.0 Vdc)(1)		50	—	
(I_C = 500 mAdc, V_{CE} = 10 Vdc)(1)		40	—	
Collector–Emitter Saturation Voltage(1)	$V_{CE(sat)}$			Vdc
(I_C = 150 mAdc, I_B = 15 mAdc)		—	0.3	
(I_C = 500 mAdc, I_B = 50 mAdc)		—	1.0	
Base–Emitter Saturation Voltage(1)	$V_{BE(sat)}$			Vdc
(I_C = 150 mAdc, I_B = 15 mAdc)		0.6	1.2	
(I_C = 500 mAdc, I_B = 50 mAdc)		—	2.0	

We see that at a collector current of 10 mA, the minimum value of h_{FE} is 75. For a collector current of 150 mA, the minimum value of h_{FE} is 100. We do not know what h_{FE} is at 60, but the plot below shows us that the curve is well behaved and smooth between 60 and 150 mA:

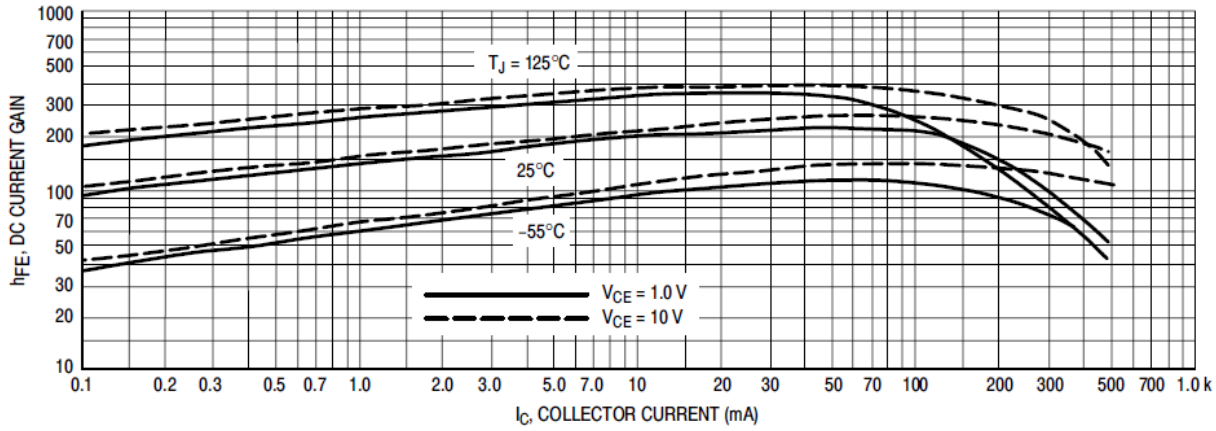


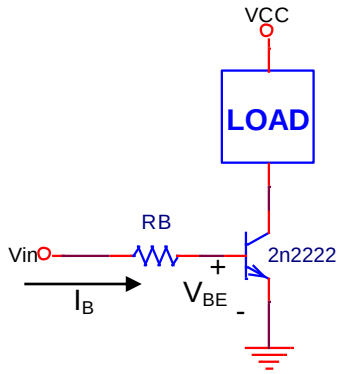
Figure 3. DC Current Gain

Thus, we will use a minimum value of 75 for h_{FE}.

The design equations for our BJT driver are as follows. To guarantee that the switch is on when Vin is high, we need $I_B \geq I_{LOAD}/H_{FEmin}$. To satisfy this constraint, we will choose

$$I_B = 1.1\left(\frac{I_{LOAD}}{H_{FE_{min}}}\right) = 1.1\left(\frac{60\ mA}{75}\right) = 880\ \mu A$$

Next, we need to calculate I_B from the circuit, which is repeated below:



From the bas loop, we have:

$$I_B = \frac{V_{in} - V_{BE}}{R_B} \geq 880\ \mu A$$

The base current flows when the input is high. We will assume that when the PWN output is high, the voltage can be anywhere between 4 and 5 V. Remember that larger base current guarantees that the BJT will be an on switch, so if we have to round component sizes or choose parameter limits, we will choose the ones that make the I_B larger. Solving the above equation for the R_B, we get:

$$R_B \leq \frac{V_{IN_{min}} - V_{BE}}{I_B}$$

We chose $V_{IN_{min}}$ because if we meet the minimum value of I_B with the minimum input voltage, I_B will be even larger when V_{IN} is larger. Solving, we get:

$$R_B \leq \frac{4\ V - 0.7\ V}{880\ \mu A} = 3750\ \Omega$$

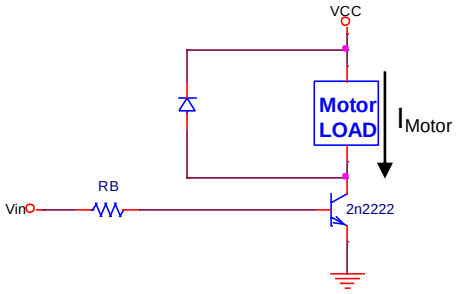
We will round R_B down to the next closest 5% resistor, which happens to be 3.3 kΩ. We will use the same Simulink models as before to control the brightness of the light. Remember to use the transistor driver and not hook the light bulb directly to a pin of the Arduino board.

Demo IV.4: Demonstrate that your PWM output can directly control the brightness of the light bulb by having the brightness follow a 1 Hz sine wave.

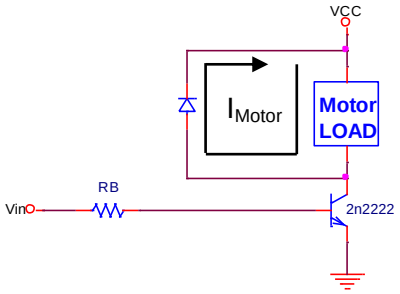
Exercise IV.2: Create a model that uses a push-button to switch between the following waveforms: (1) A sine wave at 1 Hz, (2) a sine wave at 0.5 Hz, (3) a 1 Hz ramp, and (4) a 1 Hz triangle wave. Drive the light bulb with all of these waveforms.

C. Variable Speed Motor

We can drive a motor the same way we drive any large load. However, since the motor is inductive, we need to add a diode as shown:



When the switch is on, the diode is reverse biased and current flows as shown above. When the switch turns off, the current through the switch goes to zero instantaneously. The inductor current, however, cannot go to zero instantaneously and must continue to flow. The diode provides a path as shown below:



The current will circulate as shown until the energy stored in the motor inductance is dissipated, or the switch turns on again. A diode used in this configuration is called a “flyback” diode and is needed whenever we drive an inductive load with a switch. If the flyback diode were not present, when the switch turns off, the inductor will generate a large voltage in an attempt to keep current flowing. If the voltage is large enough, it will damage the switch.

Demo IV.5: Demonstrate that you can control the speed of your motor using the PWM output of the Arduino. You may need to change the driver transistor and the base resistor depending on the current requirements of your motor.

Lab V
Serial Communication

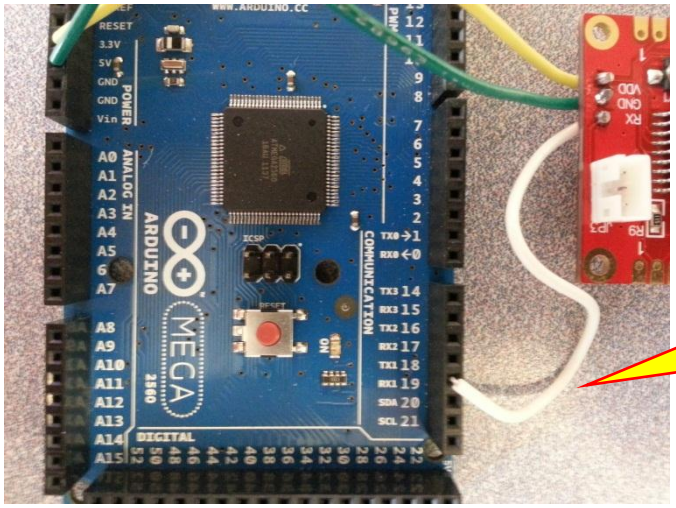
A. Introduction

We will now look at using the serial communication ports of the Arduino Mega. The Mega has four TTL-level serial ports. Traditional RS-232 serial ports have a minimum of three wires for communication: a ground wire, a transmit (TX) wire, and a receive (RX) wire. When connecting two serial devices together, you would connect the grounds together and then connect the RX of device 1 to the TX of device 2, and the TX of device one to the RX of device two. This allows two devices to communicate with each other, but does not allow networking of several devices. Traditional RS-232 devices used +/- 12 V signal levels. The Arduino serial ports differ in that they use TTL level signals of 0 to 5 V rather than +/- 12 V signal levels. We cannot connect an Arduino serial port directly to an RS-232 port as higher signal voltages of the RS-232 port could damage the Arduino board. There are, however, a number of devices that have TTL-level serial ports, and we will use the Arduino to communicate with devices that have these types of ports.

We will use an LCD display that has a TTL-level serial interface, available from SparkFun Electronics. The LCD-09393 is a 16x2 serial-enabled LCD display. 16x2 means that it can display 2 lines of text, each with a maximum of 16 characters. Note that it is slightly different than the model you may have. The datasheet for the serial LCD module is shown in Appendix XVI.A on page 266. Note that the Serial LCD display only receives and displays text, so it only has an RX input port:

To use the serial device, from an electrical interface point of view, all we need to do is hook up power and ground (VDD on the Serial LCD connects to 5 V on the Arduino Mega, and GND on the Serial LCD connects to GND on the Arduino Mega), and connect one of the transmits outputs on the Arduino Mega to the receive input on the serial LCD. We will use the Arduino Mega TX1 to connect to the LCD RX input:

You can use any of the Arduino Mega transmit ports. However, serial port 0 (RX0 and TX0) are also used for communication between the Arduino board and your host computer. To avoid any possible conflicts, we will not use port 0. Connect the TX output of the Arduino port you select to the Serial LCD RX port:



Do not forget to hook up the power and ground for the Serial LCD as well.

Serial RS-232 communication sends text as a series of 8-bit codes referred to as ASCII (American Standard Code for Information Interchange). The Serial LCD will respond to and display many of the ASCII codes. The codes include standard text characters like A through Z, a through z, 0 through 9, punctuation, and the like. ASCII also includes special characters like ESC, carriage return, and bell. The serial LCD will display many of the standard printable characters, but uses a few of the codes as its own control codes.

You can find tables for ASCII codes on the internet and is many archaic text books that were used in the era that the person writing this book attended college. The table below is from Wikipedia [1]:

Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	␣	100 0000	100	64	40	@	110 0000	140	96	60	`
010 0001	041	33	21	!	100 0001	101	65	41	A	110 0001	141	97	61	a
010 0010	042	34	22	"	100 0010	102	66	42	B	110 0010	142	98	62	b
010 0011	043	35	23	#	100 0011	103	67	43	C	110 0011	143	99	63	c
010 0100	044	36	24	$	100 0100	104	68	44	D	110 0100	144	100	64	d
010 0101	045	37	25	%	100 0101	105	69	45	E	110 0101	145	101	65	e
010 0110	046	38	26	&	100 0110	106	70	46	F	110 0110	146	102	66	f
010 0111	047	39	27	'	100 0111	107	71	47	G	110 0111	147	103	67	g
010 1000	050	40	28	(	100 1000	110	72	48	H	110 1000	150	104	68	h
010 1001	051	41	29	)	100 1001	111	73	49	I	110 1001	151	105	69	i
010 1010	052	42	2A	*	100 1010	112	74	4A	J	110 1010	152	106	6A	j
010 1011	053	43	2B	+	100 1011	113	75	4B	K	110 1011	153	107	6B	k
010 1100	054	44	2C	,	100 1100	114	76	4C	L	110 1100	154	108	6C	l
010 1101	055	45	2D	-	100 1101	115	77	4D	M	110 1101	155	109	6D	m
010 1110	056	46	2E	.	100 1110	116	78	4E	N	110 1110	156	110	6E	n
010 1111	057	47	2F	/	100 1111	117	79	4F	O	110 1111	157	111	6F	o
011 0000	060	48	30	0	101 0000	120	80	50	P	111 0000	160	112	70	p
011 0001	061	49	31	1	101 0001	121	81	51	Q	111 0001	161	113	71	q
011 0010	062	50	32	2	101 0010	122	82	52	R	111 0010	162	114	72	r
011 0011	063	51	33	3	101 0011	123	83	53	S	111 0011	163	115	73	s
011 0100	064	52	34	4	101 0100	124	84	54	T	111 0100	164	116	74	t
011 0101	065	53	35	5	101 0101	125	85	55	U	111 0101	165	117	75	u
011 0110	066	54	36	6	101 0110	126	86	56	V	111 0110	166	118	76	v
011 0111	067	55	37	7	101 0111	127	87	57	W	111 0111	167	119	77	w
011 1000	070	56	38	8	101 1000	130	88	58	X	111 1000	170	120	78	x
011 1001	071	57	39	9	101 1001	131	89	59	Y	111 1001	171	121	79	y
011 1010	072	58	3A	:	101 1010	132	90	5A	Z	111 1010	172	122	7A	z
011 1011	073	59	3B	;	101 1011	133	91	5B	[	111 1011	173	123	7B	{
011 1100	074	60	3C	<	101 1100	134	92	5C	\	111 1100	174	124	7C	|
011 1101	075	61	3D	=	101 1101	135	93	5D	]	111 1101	175	125	7D	}
011 1110	076	62	3E	>	101 1110	136	94	5E	^	111 1110	176	126	7E	~
011 1111	077	63	3F	?	101 1111	137	95	5F	_					

Figure V-1: ASCII table of printable characters.

Typically, text strings and the ABS function in MATLAB will do the ASCII conversion for us, so we do not usually need to know the codes. Occasionally, we may need to send a character that does not work out well with text strings, and we will need to specifically insert the ASCII code for that character.

The rate that information is transmitted on a serial line is referred to as the BAUD rate. In serial communication, that can be symbols per second or bits per second. Since an ASCII symbol is represented by an 8-bit code, there is quite a difference between symbols per second and bits per second. For serial ports, speed is usually specified in bits per second (bps). The default speed of both the Arduino ports and the Serial LCD port is 9600 bps, so we will typically not have to change the port speeds. However, if you are having trouble with serial communication, you may want to verify that the speeds on both ports are the same.

Before we start our first model, we need find out to clear the LCD screen. (Though we do not need to do this if we always send 32 characters to the LCD, the unused ones being blanks (ASCII 32 in decimal). Referring to the Serial LCD data sheet, sending the LCD a code of decimal 254 tells the LCD that the next character it receives will be a command code. Following 254 by the code 01 clears the display and positions the cursor at the top left position of the LCD screen. Thus, we will always prepend our text strings with the codes [254 01].

B. Useful MATLAB Commands

Before we use our LCD screen, it will be useful to review some commands that are useful with text strings. First, text strings are encased in single quotes, for example, ‘This is a test.’ Entering this command in MATLAB is shown below:

```
Command Window
>> a='This is a test.'

a =

This is a test.

fx >>
```

The MATLAB ABS command, among other things, converts a text string to an array of ASCII codes:

```
Command Window
>> abs('This is a test.')

ans =

  Columns 1 through 14

    84   104   105   115    32   105   115    32    97    32   116   101   115   116

  Column 15

    46
```

Arrays can be concatenated by placing them inside square brackets and separating the strings by a comma:

```
Command Window
>> ['This is string 1.' , 'This is string 2.']

ans =

This is string 1.This is string 2.

fx >>
```

Or, some slight variations:

```
Command Window
>> Line1 = 'This is the first string. ';
>> Line2 = 'This is the second string. ';
>> [Line1, Line2]

ans =

This is the first string. This is the second string.

fx >>
```

And finally, just for fun:

```
Command Window
>> Line1 = 'This is the first string. ';
>> [Line1, 'This is something else.']

ans =

This is the first string. This is something else.

fx >>
```

We can use the concatenate command to concatenate strings or arrays. We need to send the LCD an array of ASCII codes. To create the array of codes that clears the LCD screen and display the text, ‘This is a test!’ we can use the command below:

```
Command Window
>> [254, 01, abs('This is a test!')]

ans =

  Columns 1 through 14

   254     1    84   104   105   115    32   105   115    32    97    32   116   101

  Columns 15 through 17

   115   116    33

fx >>
```

Or, alternatively:

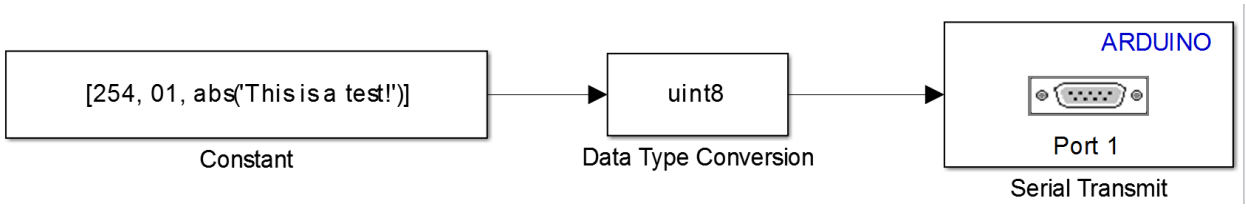
```
Command Window
>> Line1 = 'This is a test!';
>> [254, 01, abs(Line1)]

ans =

  Columns 1 through 14

   254     1    84   104   105   115    32   105   115    32    97    32   116   101

  Columns 15 through 17

   115   116    33

fx >>
```

C. Displaying Text on and LCD Display

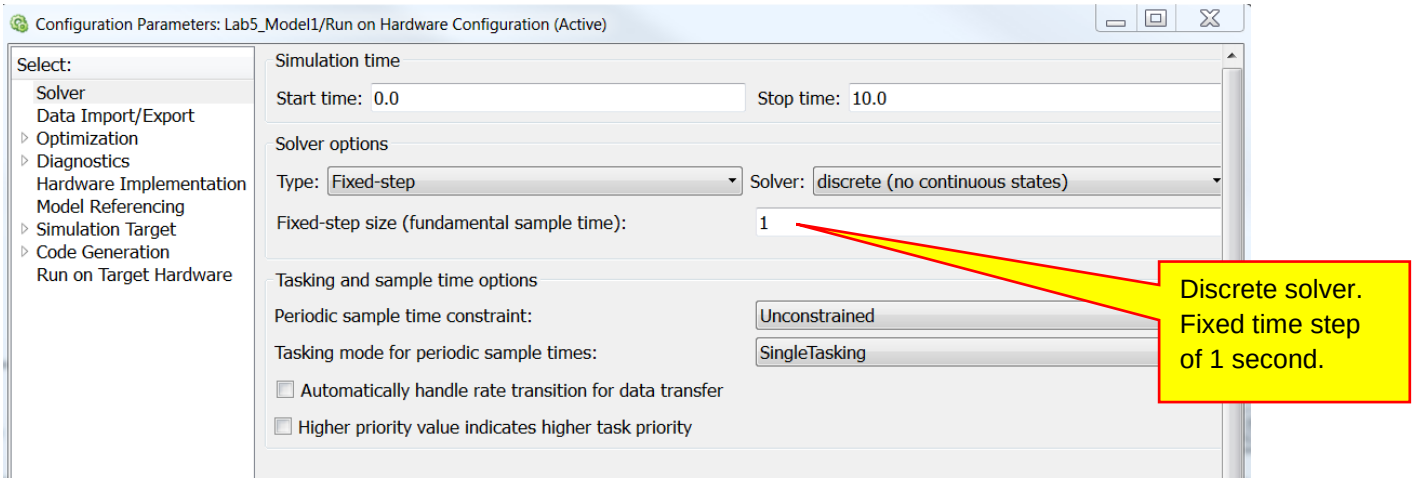
We are now ready to create a model with Simulink that displays text on the.

1. Displaying a Single Line of Text

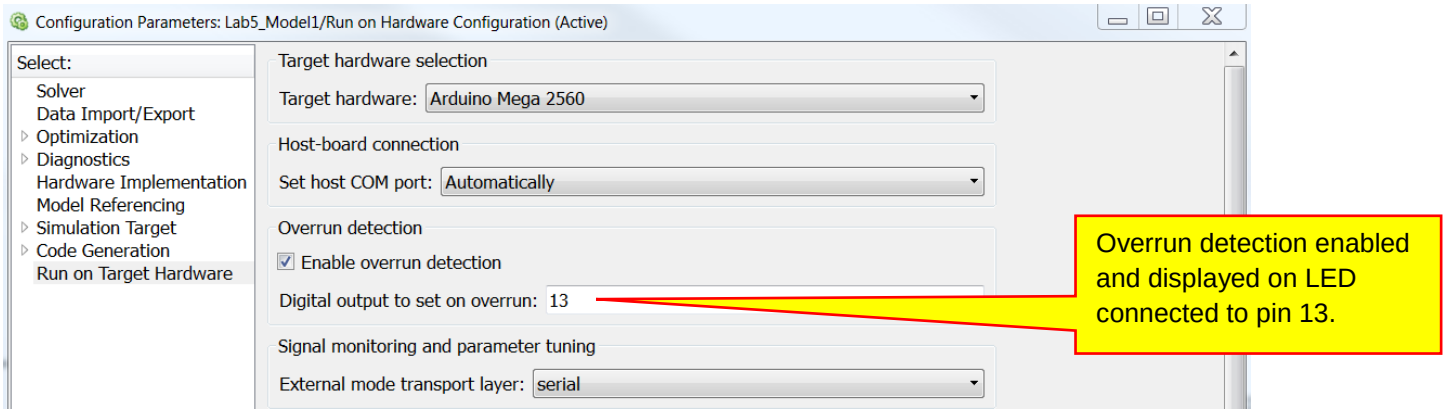
Create the model below:



This model should display the text message ‘This is a test!’ on the LCD screen. Note that the **Constant** block will output a 1-dimensional array of double precision numbers. The **Data Type Conversion** block will convert the array to a 1-dimensional array of 8-bit integers, appropriate for serial transmission. Set up the **Model Configuration Parameters** to use the discrete solver with a fixed time step of 1 second:



In the **Run on Target Hardware** selection, make sure that you **Enable overrun detection** and use the on-board LED connected to pin **13**:



Note that the model will run once every second, meaning that the test message will be resent to the LCD display every second. This is unnecessary because we only need to send it once and the LCD will continue to display the text until a different text message is sent to the LCD or the power is removed from the LCD.

Demo V.1: Display the text, 'This is a test!' on your LCD screen at a fixed step size of 1 second.

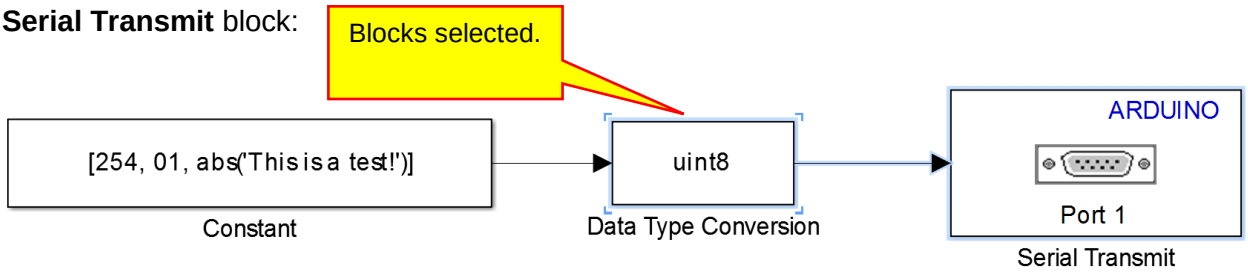
For our models to do any type of serious control or instrumentation functions, we usually need a fixed time step smaller than 1 second. Change the fixed time step to 0.01 seconds and rerun the model. You will notice that the LCD displays 'dimmed out' or jittery text:

The text appears this way because it takes time to transmit the text over the serial line and we are rapidly resending the text string, the first two characters of which clear the screen. Thus, if we have a small fixed time step, we need to find a way to reduce the rate at which we send a text string to the LCD. We actually only need to send the text string once and then never send it again until we need to change the text string. We will fix this problem in the next section by using triggered subsystems.

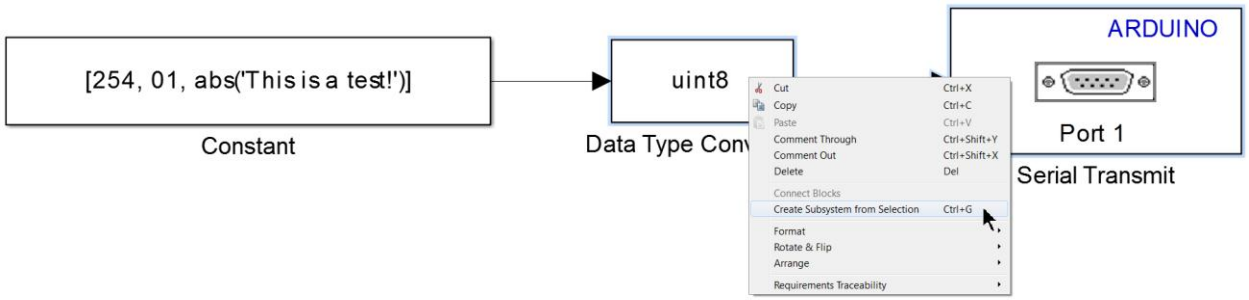
a) Triggered Subsystems – 1 Hz Rate

Typically we would like our Arduino to run with a fast fixed time step so that it can control external systems quickly or respond to sensor inputs quickly. However, we only need to send information to the LCD at a slow rate because humans can only read so quickly and because the LCD display reacts slowly. The ideal solution for this is to place the serial transmit block inside a triggered subsystem. We introduced triggered subsystems in Section III.B.1 on page 64. Triggered subsystems allow us to run different sections of the model at different clock rates. In this case, we will run the complete model with a fixed time step of 0.01 seconds and place the LCD communication portion of the model inside a triggered subsystem such that it runs once every second. This does two things for us. First it allows the LCD module to display text properly, since updating the LCD text string too frequently caused the test to fade out. Second, it frees the processor to work on other tasks.

We will start with the previous model and select the **Data Type Conversion** block and the **Serial Transmit** block:

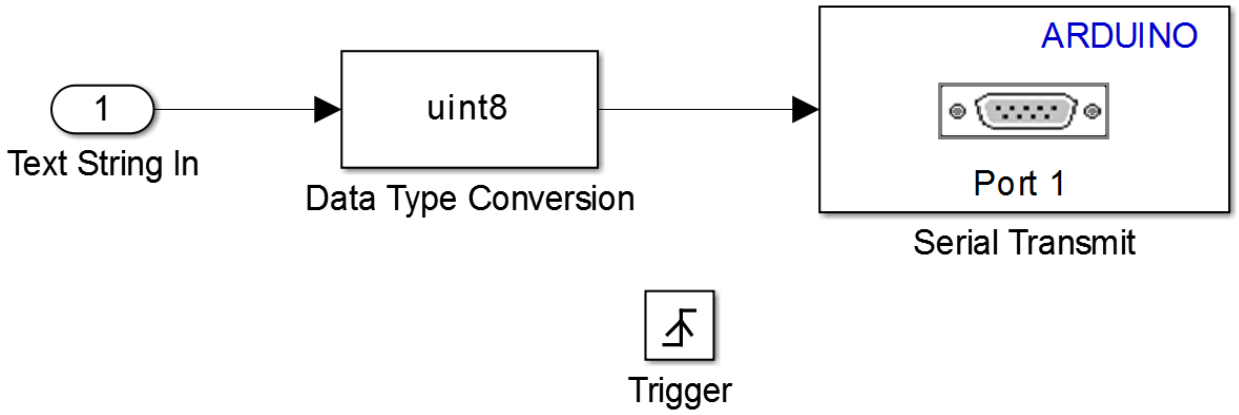


Right-click on one of the selected blocks and select **Create Subsystem from Selection**:

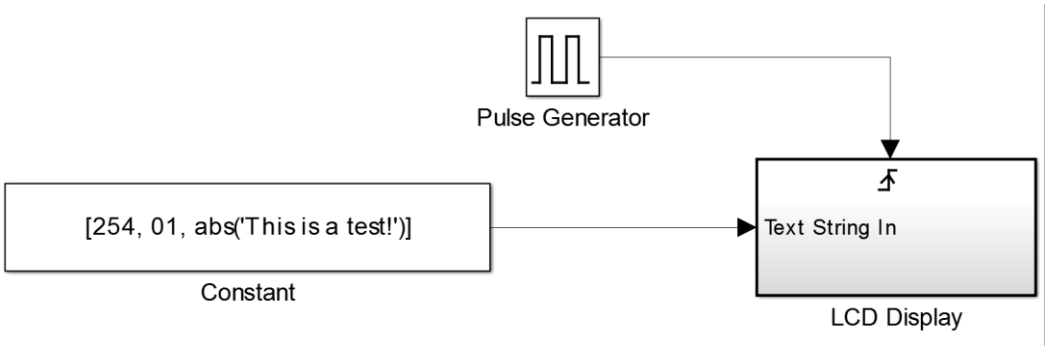


The two blocks are placed inside a conventional subsystem. Rename the subsystem "LCD Display." Double-click on the subsystem to open it and rename the input port to "Text String In." To make the subsystem a triggered subsystem, place a trigger block 

in the subsystem. This block is located in the **Simulink / Ports and Subsystems** library:



By default, the trigger is positive edge triggered. If you want to change the trigger type you can double-click on the block and change the trigger type to positive edge, negative edge, both, or a function call trigger. We do not need to change the trigger type, so navigate back to the top level of the model, and place a **Pulse Generator** in the model and set the period to one second:



With a Pulse Generator period of one second, the LCD display subsystem will run once every second, even though the model uses a fixed time step is 10 ms. Down load and run your model. The LCD display should work properly.

Demo V.2: Display the text, 'This is a test!' on your LCD screen at a fixed step size of 0.1 second and using a triggered subsystem to update the text screen once every second.

Exercise V.1: Using a switch, have the LCD automatically switch at a 1 Hz rate between displaying the text "My name is xxxx." and "This class is fun!" You can use a switch to select between text strings. Note that the text strings must be the same length, so you may need to pad one of the strings with spaces.

2. Displaying Text Based on User Selection

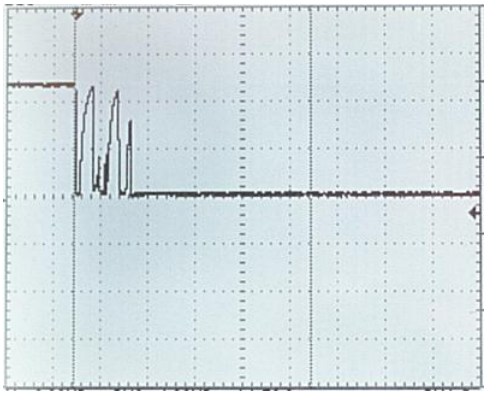
Next, we would like to change the text we display every time we press a switch. Instead of updating the LCD periodically, we will only update the LCD at the time we want to change the text. This is possible because the LCD display remembers the text string we sent and will continue to display the text until it is changed. This has the benefit of freeing up the control to perform other tasks rather than periodically send the same text string to the LCD.

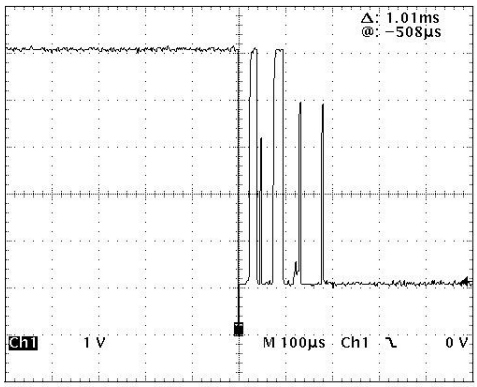
a) Stateflow Switch Debounce

In order to implement this model, we will need to recognize that when a switch changes from high to low or low to high we are responding to the edge of the change rather than the low state or high state. An example of positive and negative edges are shown below:

Some devices or functions are level sensitive, such as an enable or disable signal where the systems perform their function when a signal is high or low. Some are edge triggered where the systems start to perform their function when an edge occurs. Edge triggered functions can be used for timing or sequencing operations.

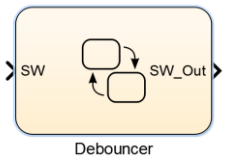
Inside digital systems, an edge is usually pretty reliable. However, we would like to sense the edge of when a push-button is pressed or a toggle switch is thrown. When a switch changed from one state to the other, the output voltage may "bounce" as shown in the figures below:



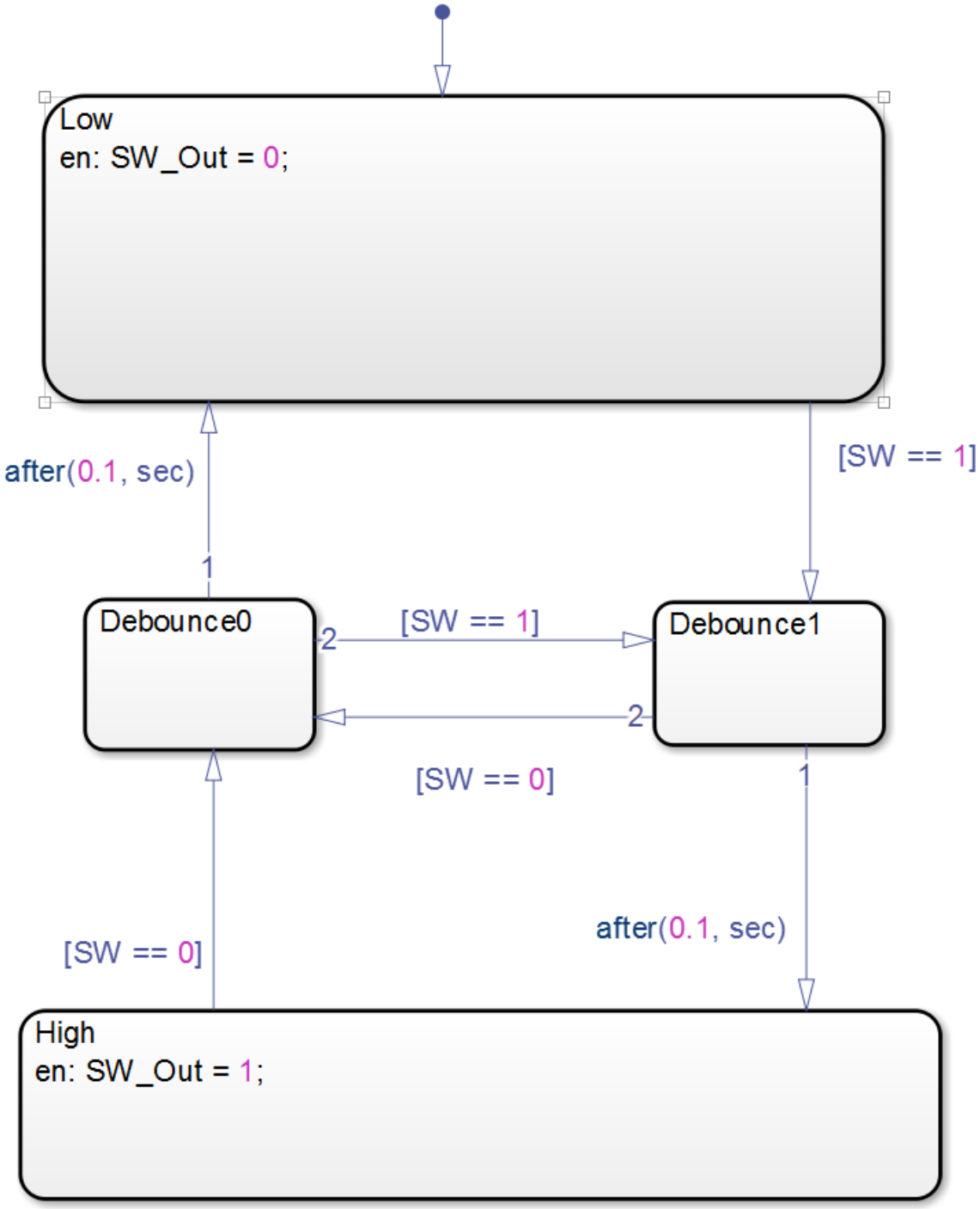


Toggle switches use a spring to throw a “hammer” contact against a metal ball contact. When they collide, the hammer momentarily makes contact with the ball and then mechanically bounces away from the ball. The spring then forces the hammer back towards the ball and the bouncing continues until it dies out. The bouncing causes intermittent electrical contact between the ball and the hammer resulting in the waveforms show above. Switch bounce is a fairly common problem and different logic solutions have been devised to change the bouncing edge into a single edge. Here, we will make a switch debounce using Stateflow logic.

The Stateflow chart shown here is a slightly modified version of the switch debouncer found in the MATLAB help. To find the original version from which this example was derived, type “doc Stateflow debounce” at the MATLAB prompt. The top level of the chart is shown below:



The input to the chart is the undebounced switch voltage. This voltage is usually steady at a high or low level, but when the switch changes state, there may be several edges. The Stateflow chart eliminates the multiple edges and passes the steady high or low value. The logic inside the chart is shown below:



During a transition, the switch changes state, and the chart will enter into the debounce states. During the transition, SW may bounce between 0 and 1 several times before it settles down. While this bouncing occurs, the chart will bounce back and forth between the **Debounce0** and **Debounce1** states. However, while it is bouncing, it remains in the Debounce state and the output of the chart never changes.

The **after** command is referred to as temporal logic and allows us to time how long we have been in a state or how long a condition has been true. In this case, the chart will not transition into the **High** state until the chart has been in the **On** state for 0.1 seconds, or SW has been constant a 1 for 0.1 seconds. SW will not be constant at 1 until the bouncing has stopped. The same temporal logic is used for the transition

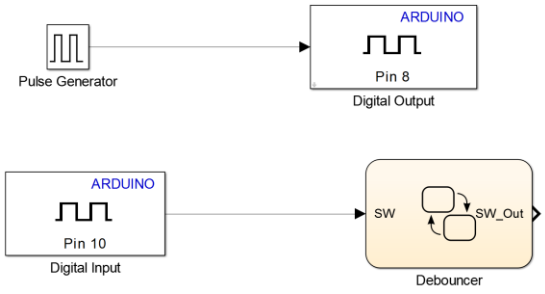
to zero. The chart will not change to the low state until the SW has been at 0 for 0.1 seconds. Thus, while the switch is bouncing, the chart will stay in the debounce states while the switch voltage is bouncing. The chart will only enter the **High** or **Low** states after the switch voltage has been constant for 0.1 seconds.

Note that there are several Stateflow temporal operators: after, before, at, every, and temporalCount. Search for "Temporal Logic in State Actions and Transitions" in the MathWorks help for more information.[2]

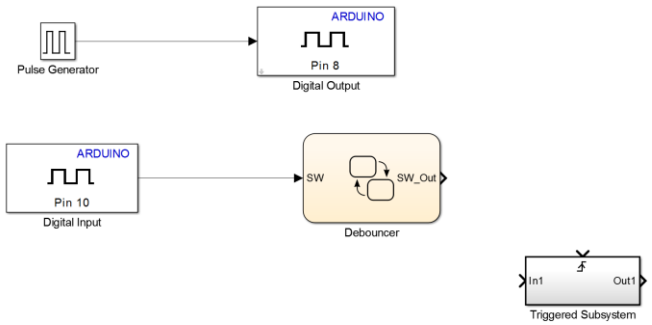
b) Function-Call Triggers

Now that we have debounced a switch, we want to make a Stateflow chart that will switch between displaying one of three different messages when the button is pressed. The text message should only be sent to the LCD once after the button has been pressed, not periodically as in the previous examples. This task requires us to sequence events after the button has been pressed. First, we must increment a counter that selects the message to be displayed. Second, we must select a text message. And third, we must send the text message to the LCD display. All of these are triggered by the pushbutton being pressed. To sequence events in a specific order, we need to use a different kind of trigger called a function-call trigger. Conceptually, they operate the same as an edge trigger; you can have blocks generate a function call trigger, and you can make subsystems execute when they receive a function call trigger. The difference is that we can use a demux to split a single function trigger into several function triggers, all of which execute in sequence.

We will start with the system below, most of which we generated previously:

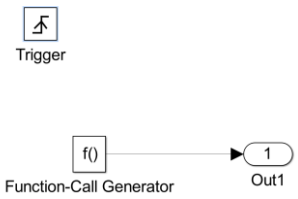


The output of the debouncer is a digital signal that switches between 0 and 1. This will generate positive and negative edges, which we could use to trigger edge triggered subsystems. For this example, we need to convert a positive edge to a function trigger. Place a **Triggered Subsystem** (Simulink / Ports & Subsystems) in your model:

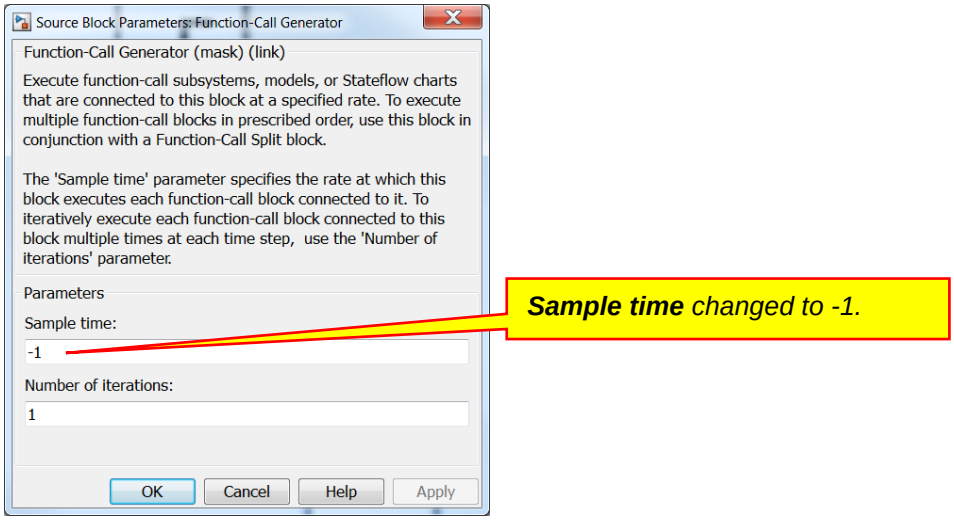


Open the **Triggered Subsystem**, delete the input port and place a part called **Function-Call Generator** (Simulink / Ports & Subsystems) in your model:

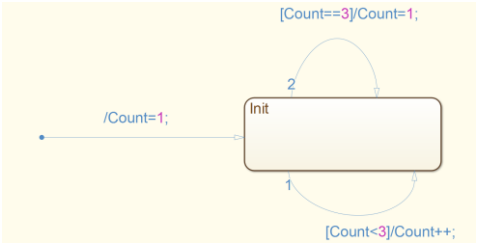
[2] Switch debouncer subsystem are located in the RHIT Arduino library under the Miscellaneous Functions.



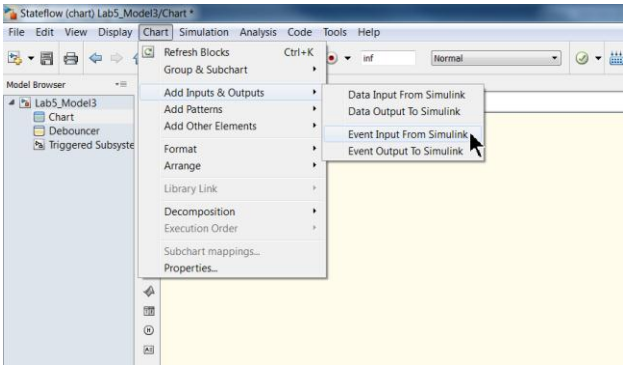
This subsystem executes every time it receives a positive edge. All the subsystem does is generate a function-call trigger. Thus, it converts a rising edge trigger to a function-call trigger. Since this **Function-Call Generator** is inside a triggered subsystem, we need to change its sample time. Open the **Function-Call Generator** part and change its **Sample time** to -1 (inherited):



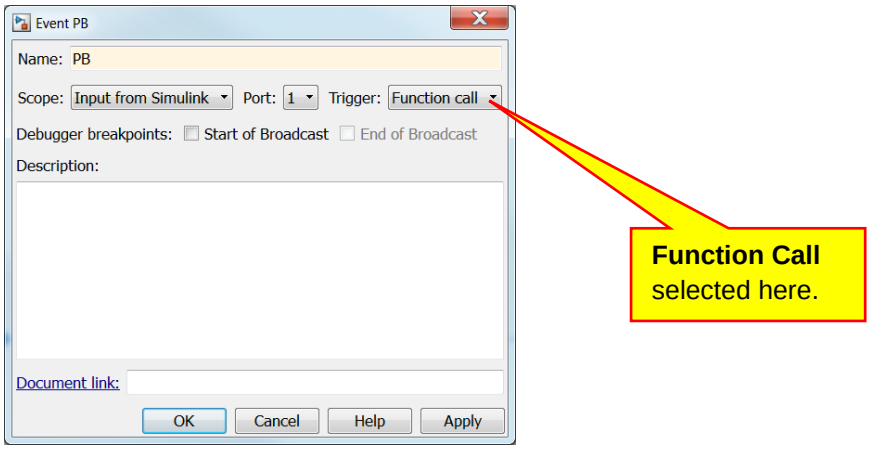
Next, we need to generate a counter that increments every time it receives a function-call trigger. We will do this with Stateflow using the chart shown below:



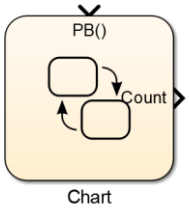
To make this chart execute every time it receives a function-call trigger, select **Chart**, **Add Inputs & Outputs**, and then **Event Input From Simulink**:



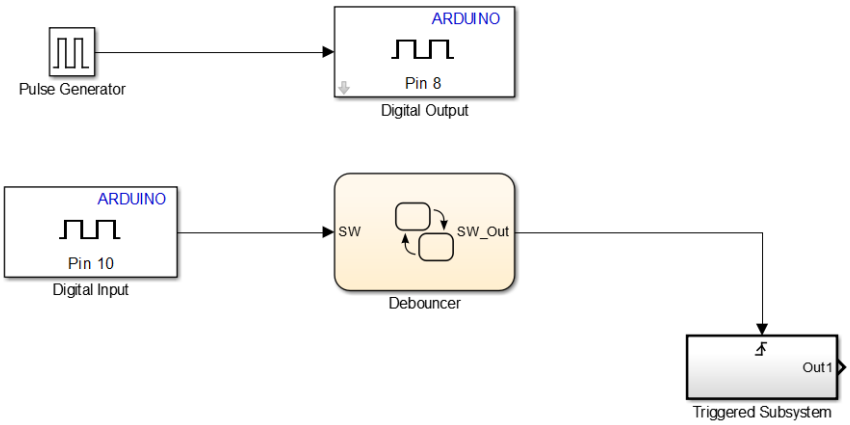
Name the input **PB** and specify the **Trigger** as a **Function call**:

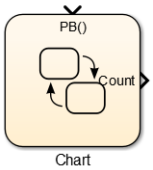


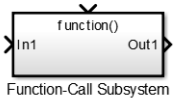
When you return to the top level of the model, you will notice the event displayed as **PB()** indicating that the event is a function-call trigger.



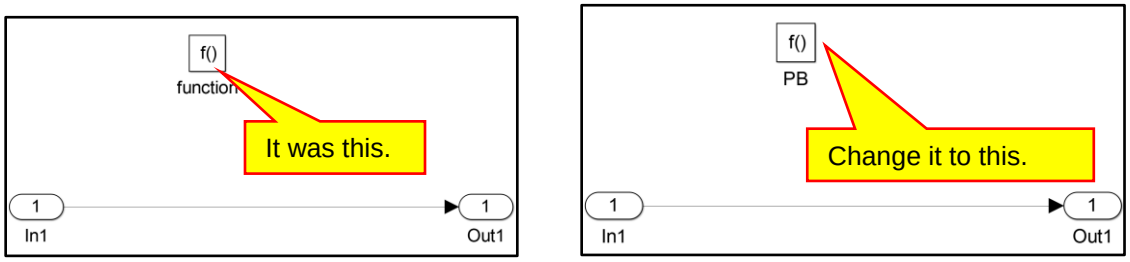
Next, we need to create another subsystem that selects the text message. Place a **Function-Call Subsystem** (Simulink / Ports & Subsystems) in the top-level of your model:







To accomplish the same thing, you could have also placed a Subsystem and added a trigger or placed a Triggered Subsystem and changed the trigger type. Here, we just placed a Function-Call Subsystem to save a few steps. Note that the name of the trigger is function() indicating that it is a function-call trigger. Open the **Function-Call Subsystem** and change the name of the trigger as shown:



We will be using this subsystem to specify the text message. We could use a **Multport Switch** (Simulink / Signal Routing) or even a Stateflow chart. Instead, we will use the Truth Table below:

Block: Lab5_Model3/Function-Call Subsystem/Truth Table

File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4
1	Text 1	u==1	T	F	F	-
2	Text 2	u==2	F	T	F	-
3	Text 3	u==3	F	F	T	-
		Actions: Specify a row from the Action Table	1	2	3	4

Action Table

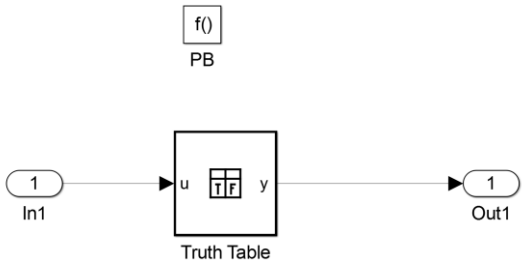
#	Description	Action
1	Test 1	y = [254, 01, abs('This is the first text string. ')]';
2	Test 2	y = [254, 01, abs('A second text string, this is. ')]';
3	Text 3	y = [254, 01, abs('Text string 3. ')]';
4	Error	y = [254, 01, abs('Error... ')]';

Don't forget the transpose operator.

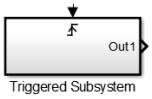
A few notes are important. First is that all of the text strings are 32 characters. Second, the text strings are enclosed in single quotes. Third, the total length of the array is 34 elements. Fourth, we are using the

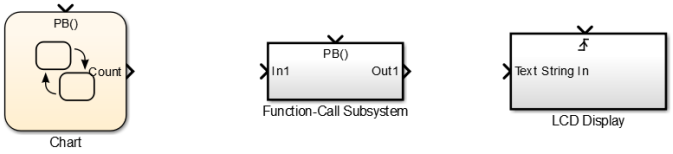
transpose operator (‘) to convert a row vector to a column vector. I’m not sure why we need to convert the array from a row to a column vector, but the string will not be displayed if we don’t.

This truth table emits one of three text messages based on the count being between 1 and 3. If, for some reason, our Stateflow chart has a count outside this range, the truth table will send out an error message. This should never happen if our logic is correct. The complete subsystem is shown below:

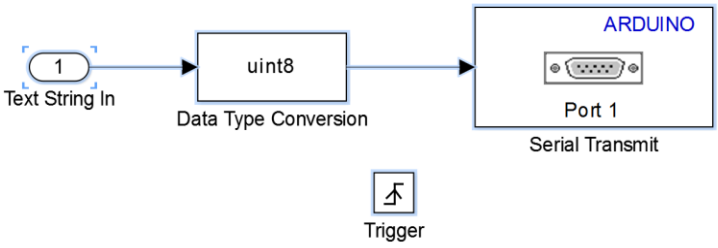


Finally, at the top level, we need to add our triggered serial output subsystem that we created earlier:

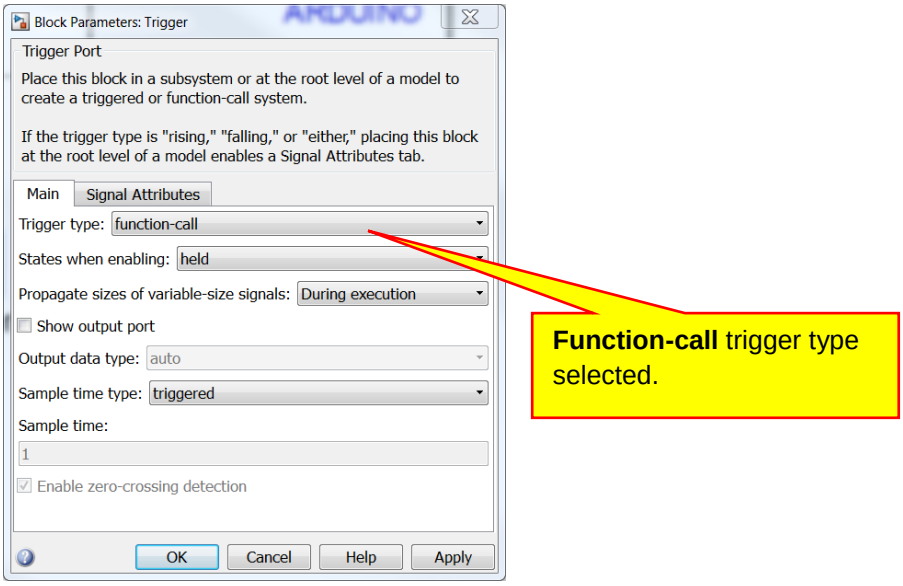




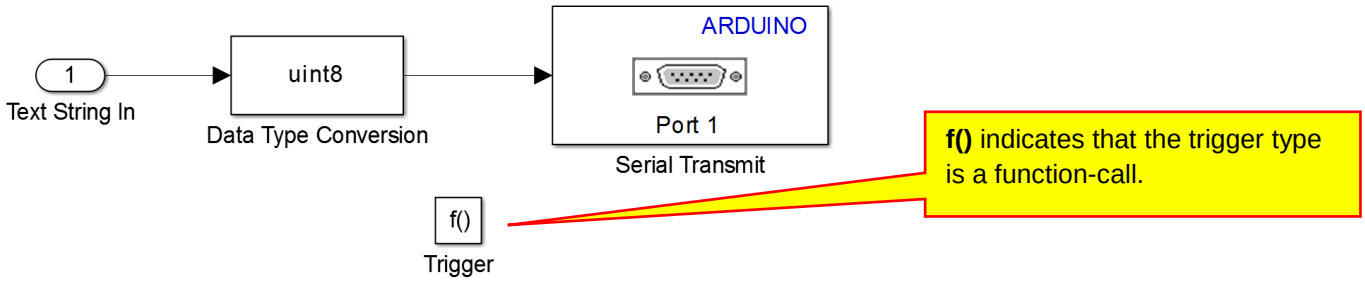
A problem with this system is that it is edge triggered, so we need to change the trigger type. Open the LCD Display subsystem:



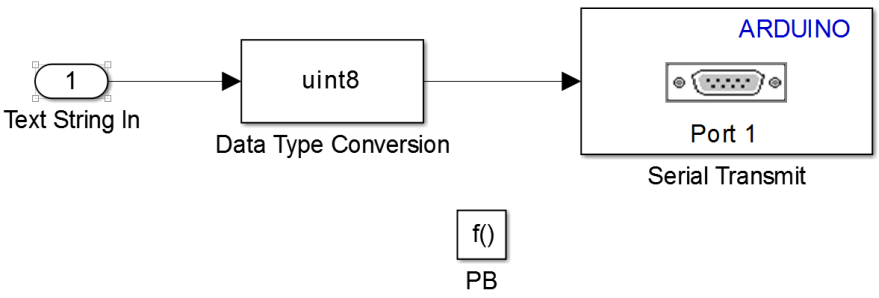
Double-click on the **Trigger** to edit its properties. Change the **Trigger type** to **function-call**:



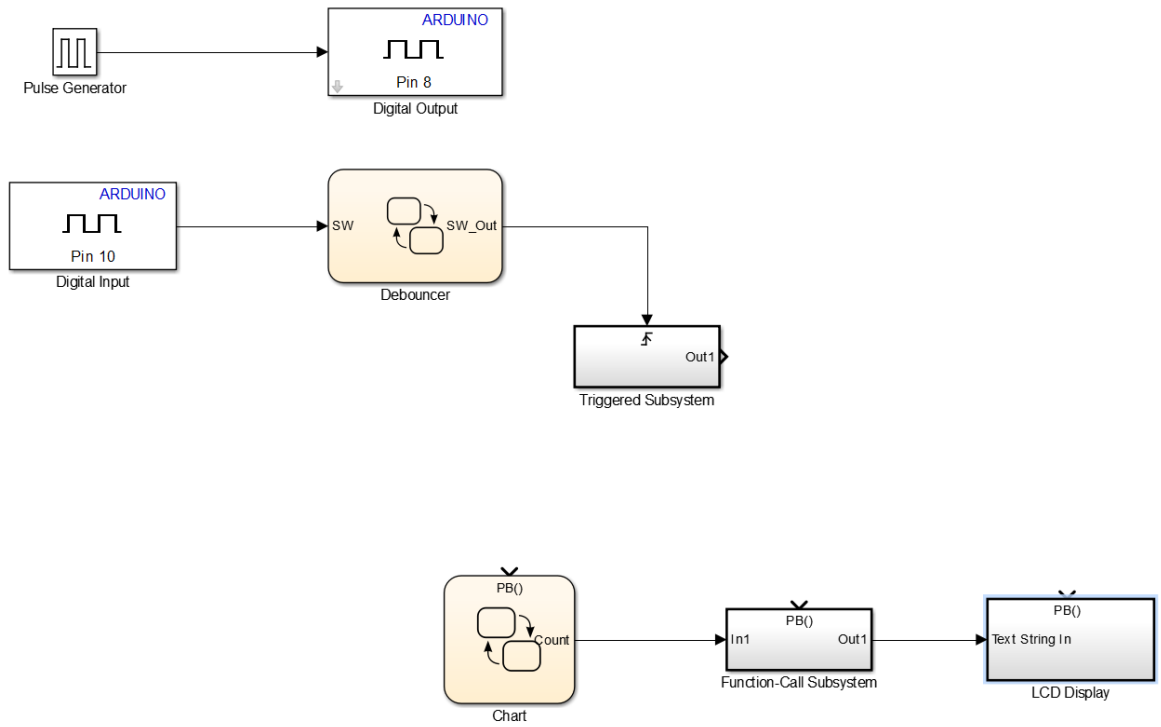
Apply the changes and return to the subsystem. You will see that the trigger type has been changed to a function-call:



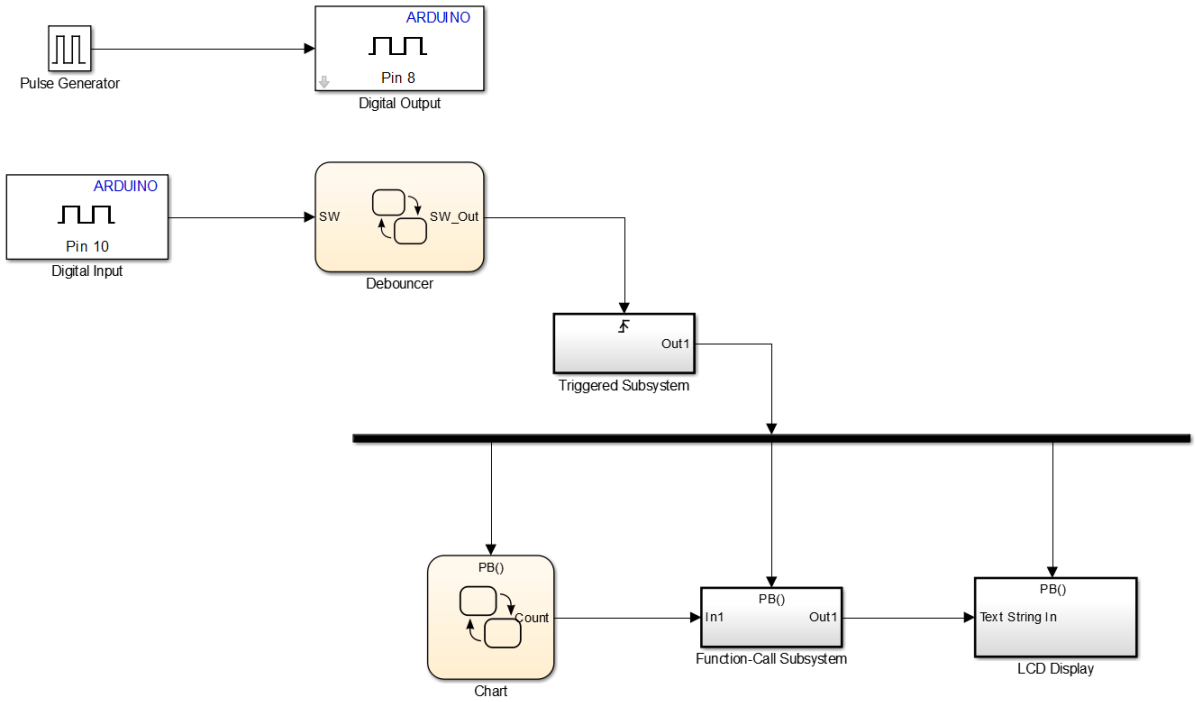
Change the name of the trigger to PB as shown below:



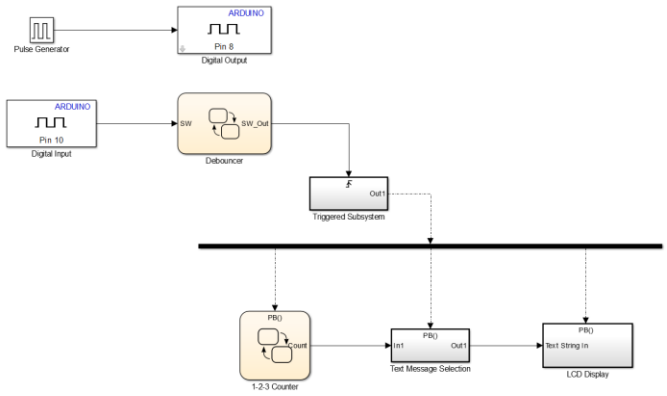
At the top level, we now have three function-call triggered subsystems. Wire them as shown:



The last thing we need to do is sequence the events. Place a **Demux** (Simulink / Commonly Used Blocks) in your model. Rotate it, change the **Number of outputs** to three, and stretch it:



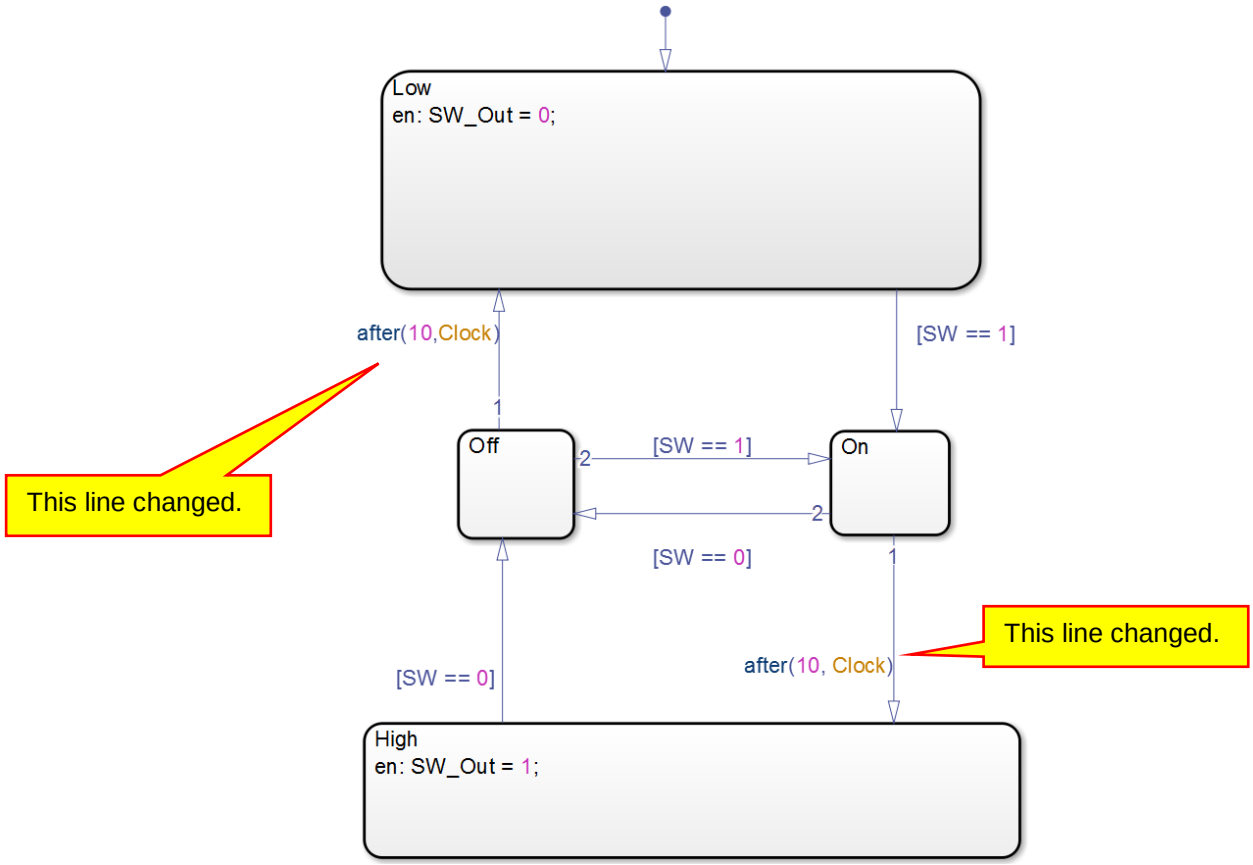
The **Demux** splits the function-call trigger input into three function-call outputs. The triggers are sequenced from left to right, so the chart will receive the trigger first, then the Function-Call Subsystem, and finally the LCD display. Using function-call triggers in this method allows us to specify a sequence of events. Rename the subsytems to document the function of the subsystems:



Download and run the model to verify. We notice that we have to hold the push-button for a significant amount of time and that the text message changes when we release the push-button.

Demo V.3: Demo the working LCD display. The text message should change after the push-button is **pressed** with an inconveniently long delay and cycle through the three text messages.

We do notice that the model takes a while to respond to the push-button, sometimes much more than the 0.1 seconds specified in the Debouncer Stateflow chart. This is because we used the command after(0.1,sec) command for timing. Timing of this type is not always implemented accurately in a real-time system and can be hardware dependent. We will change this method to use our own clock and count the number of events. First, create a pulse generator with a period of 0.01 seconds. This will require us to change the fixed time step for the model to 0.001 seconds. Next modify the Debouncer Stateflow chart as shown below. Note that the after commands have been changed to after(10,Clock):



The clock will be the input from our pulse generator which has a period of 0.01 seconds. This command will wait for ten **Clock** events, or 0.1 seconds. Finally, select **Chart**, **Add Inputs & Outputs**, and then **Event Input from Simulink** to add a new event and name it **Clock**. Run the model. The response when pressing the push-button should be much faster.

Demo V.4: Demo the working LCD display. The text message should change **quickly** after the push-button is pressed and cycle through the three text messages.

Exercise V.2: For the model of Demo V.4, if you cycle the power, you will notice that the LCD displays a blank screen until the user presses the push-button. This could be confusing to the user. Modify the model so that that at power up, the model will wait 2 seconds and then display the text, 'Press the push button to start. ' When the user presses the push-button, it will display the first text message and then cycle through the three text messages when the button is pressed. The message 'Press the push button to start. ' will never be seen again unless the power is cycled.

Exercise V.3: Create a model that switches between three functions when a push-button is pressed. The functions are displayed on the LEDs we used in previous labs. The functions are a ring counter, an up/down counter, and an analog voltmeter. You should have models that solved these problems before. In addition to this requirement, the LCD should display the text, “Ring Counter,” “Up-Down Counter,” or “Analog Voltmeter” as appropriate.

Exercise V.4: Use the LCD display to create a bar-graph display representing an analog input voltage. An input of 0 should display 0 bars and an input of 5 V should display 16 bars. Use a potentiometer to supply a variable input voltage to one of the Arduino analog input pins. On the top line of the LCD display, show the text “Bar Graph” with appropriate spacing. On the second line of the LCD display, show the bar graph. To display a square, you can send an ASCII code of 6. To display a blank use an ACSII code of 32. As an example, the array Bar = [6 6 32 32 32 32 32 32 32 32 32 32 32 32 32 32]'; will display 2 bars and 14 blanks. You may also want to use the Simulink block **Vector Concatenate** to concatenate strings. Examples are shown below:

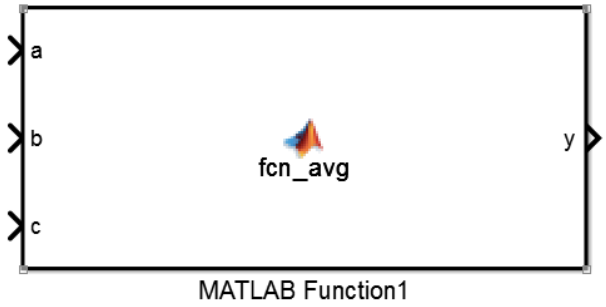
3. Formatted Text Output

Now that we know how to generate text strings and send them to the LCD display, we will look at generating formatted numerical output, such as the “The temperature is 72.4 Degrees” where the number comes from a Simulink signal. Simulink does not have a method for converting a signal to a formatted text string. However, we can use the Simulink **MATLAB Function** block to use MATLAB functions within Simulink. We will use the MATLAB **sprint** function which will make some of MATLAB’s numerical formatting available to us. For these examples, we will use the %d formatting string for converting an integer to a text string. Note that the formatting strings %f, %g, and %e are not available to us. To display a floating point number such as 3.14, we will have to break the number up into its integer and fractional parts.

a) MATLAB Functions

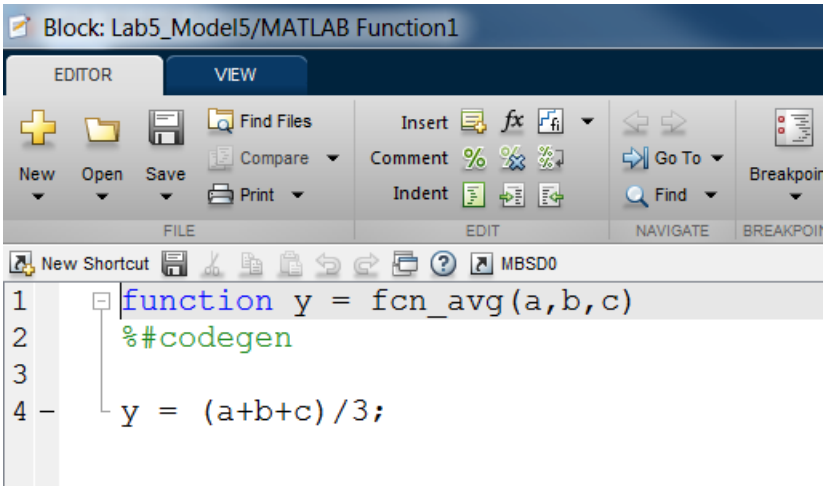
The MATLAB function block allows us to use a subset of the MATLAB language within a Simulink block. The subset of MATLAB functions can be found by searching for “functions supported by code

generation" within the MATLAB/Simulink help system. A simple example is shown next. First, the block is shown below. Note that it has three inputs called a, b, and c, and one output called y:



MATLAB Function1

The code for this block is shown below:

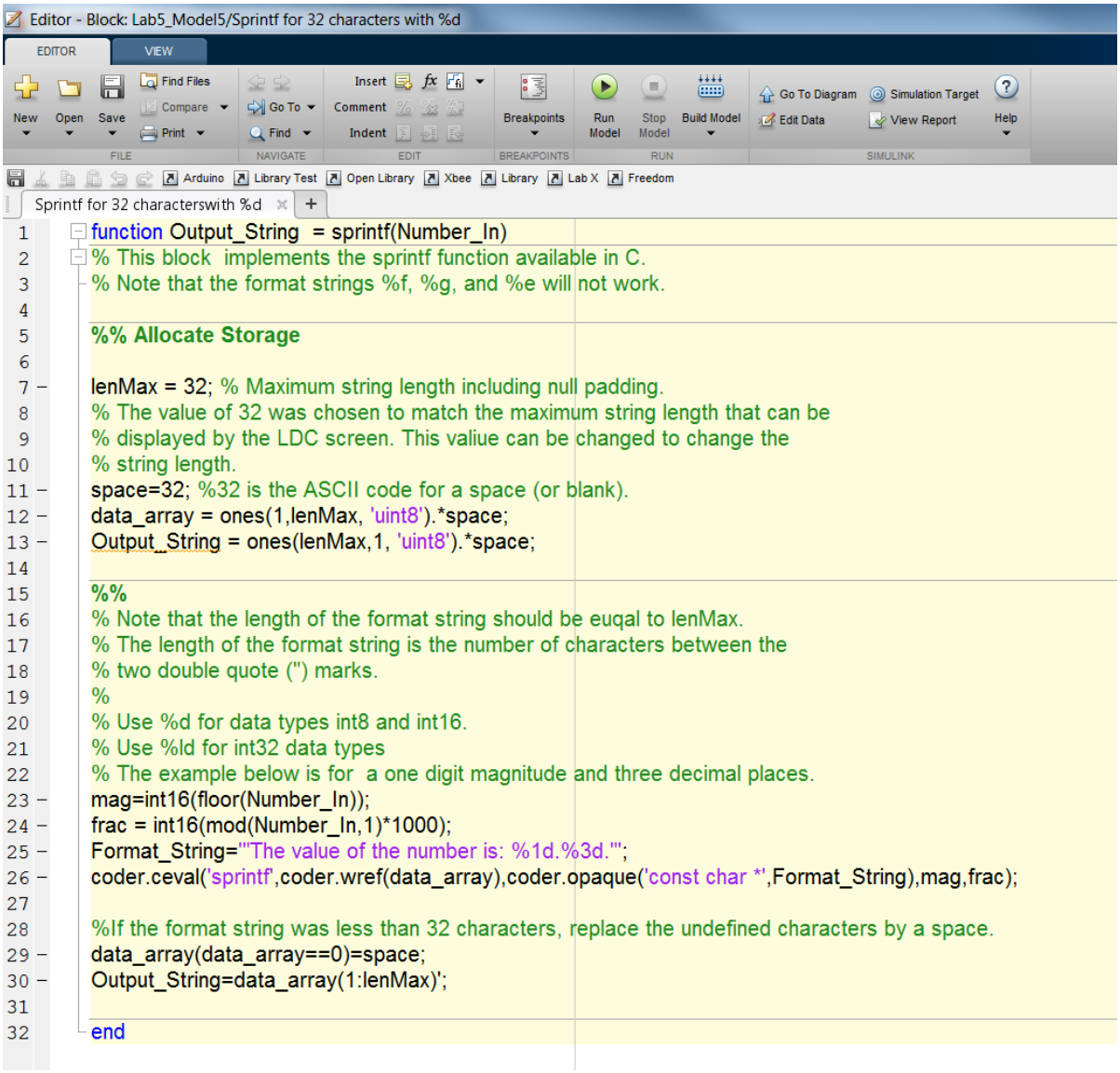


Note that the inputs to the function are a, b, and c, and correspond to the three Simulink input signals. The output of the function is y, and this corresponds to the Simulink output signal. This is a fairly trivial example, and given your comfort level with MATLAB programming, you can make quite complex functions. In a future lab, we will use MATLAB functions to create memory blocks.

a) Displaying Formatted Text Output

It turns out that the sprint function is not one of the supported MATLAB functions. So, what do we do? We call technical support and ask for help. They can write the MATLAB function for us[3]. The function is presented below:

[3] Actually, we call Peter Maloney, a MathWorks Sr. Principal Consulting Engineer, and good acquaintance through the Challenge X, EcoCAR, and EcoCAR2 Hybrid Vehicle competitions. He actually wrote the function presented here.



The input signal name is **Number_In**. The numerical value of this signal will be placed in the text string. Since we are displaying text on the LCD screen, we have limited the size of the text string output to a maximum of 32 characters. The lines **`data_array = ones(1,lenMax, 'uint8').*space;`** and **`Output_String = ones(lenMax,1, 'uint8').*space;`** allocate the storage and initialize the arrays to the value 32, which is the ASCII code for a space.

Since we are required to use the %d format string, we need to break the number into its integer and fractional portions. This is accomplished with the lines

```
mag=int16(floor(Number_In));
frac = int16(mod(Number_In,1)*1000);
```

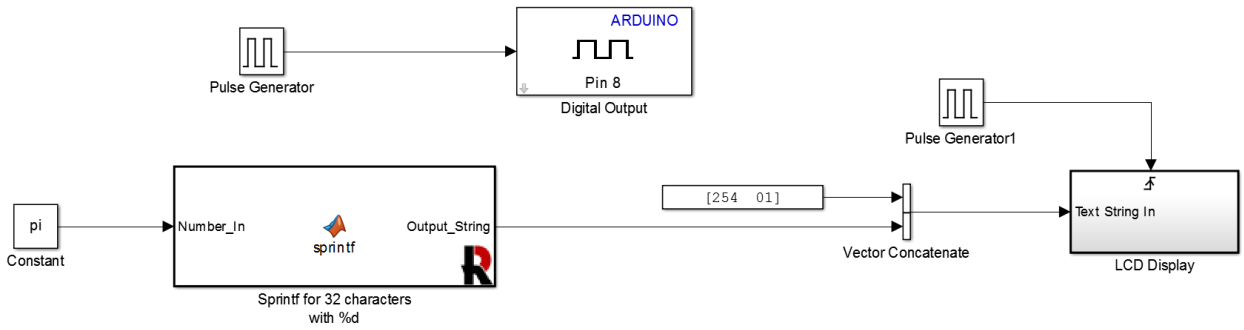
The line **`mag=int16(floor(Number_In))`** rounds Number_In down to the nearest integer and converts it to an int16 data type. The command **`mod(Number_In,1)`** divides Number_In by 1 and keeps the remainder which is the fractional part. The result of the mod command is a double data type so the fractional part is still available. The fractional part is multiplied by 1000 and then converted to an integer. Thus, variable frac contains the first three places of then fractional part of Number_In.

The lines **`Format_String='"The value of the number is: %1d.%3d."';`** and **`coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char`**

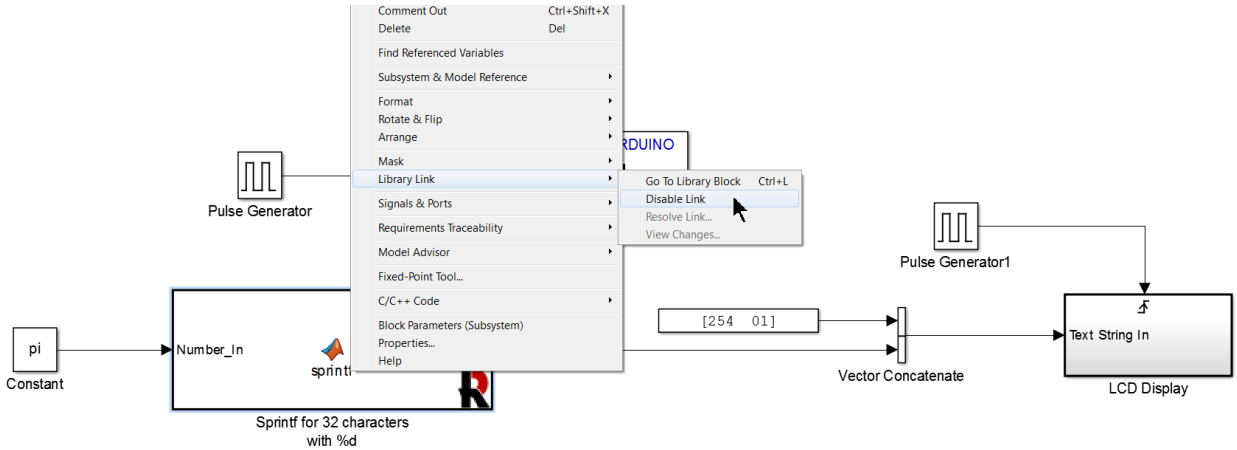
`*',Format_String),mag,frac);` do all of the work. They specify sprintf as the function we are calling and use the text between the double quotes in variable Format_String as the output string. Note that the string must be contained within double quotes, `"The value of the number is: %1d.%3d."`. Also note that this string should be 32 characters long (not including the double quotes). The numerical value of **`mag`** will be placed in the format string %1d and the numerical value of **`frac`** will be placed in the format string %3d. Thus the number itself will take a total of 5 spaces, and be displayed as x.xxx.

If we use more or less than 32 elements, some of the values of the array will be zero. When an ASCII code of zero is sent to the LCD, it displays a strange character, one we do not want. To fix this problem, we add the line, **`data_array(data_array==0)=space;`** This command scans the data_array for zeros and replaces them with the value of 32, the ASCII code for a space. Finally, the line **`Output_String=data_array(1:lenMax)';`** converts the row vector to a column vector. Remember that we used column vectors when we used the truth table for selecting one of three text strings.

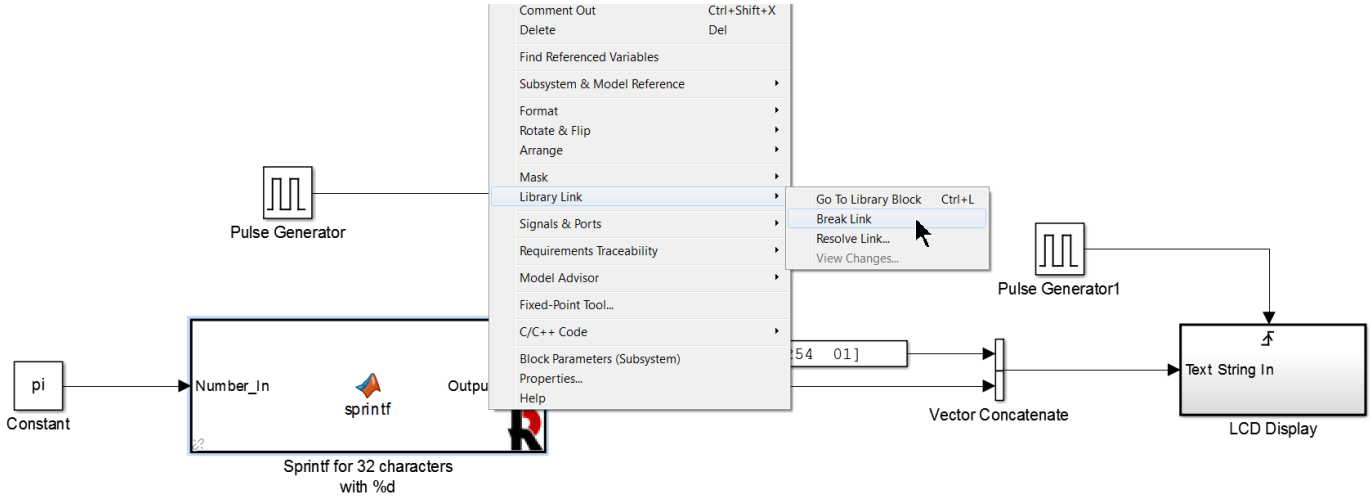
Finally, we note that this function does not prepend the string with [254, 01], the commands to clear the LCD screen. We will do this with the Simulink **Vector Concatenate** block. As a first example, create the model below for displaying the text string, "The value of pi is 3.142.' Note that the Sprintf function is contained in the RHIT Arduino Library. (See section I.B.3 on page 7 to install the library if you do not have this library listed in your Simulink Library Browser.). The sprintf function is located in the **Serial String Functions** section of the RHIT library. Use the block for %d as the %f block is reserved for future use and will not work properly:



Pi is a predefined function in MATLAB and has the value of π. We need to modify the Sprintf block to suit our needs. The block is intended to be used as an example and it is expected that you will modify it. Since the block is linked to our library, we need to unlink it and break the link. To disable the library link, right-click on the sprintf block and then select **Library Link** and then select **Disable Link**:

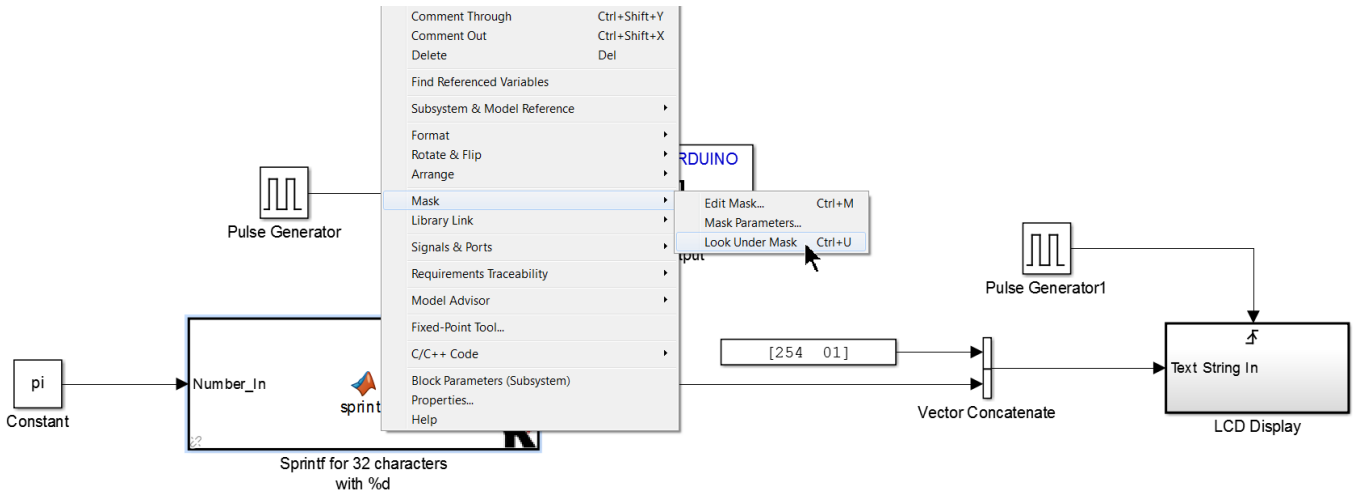


Next, right-click on the sprintf block and then select **Library Link** and then select **Break Link**:

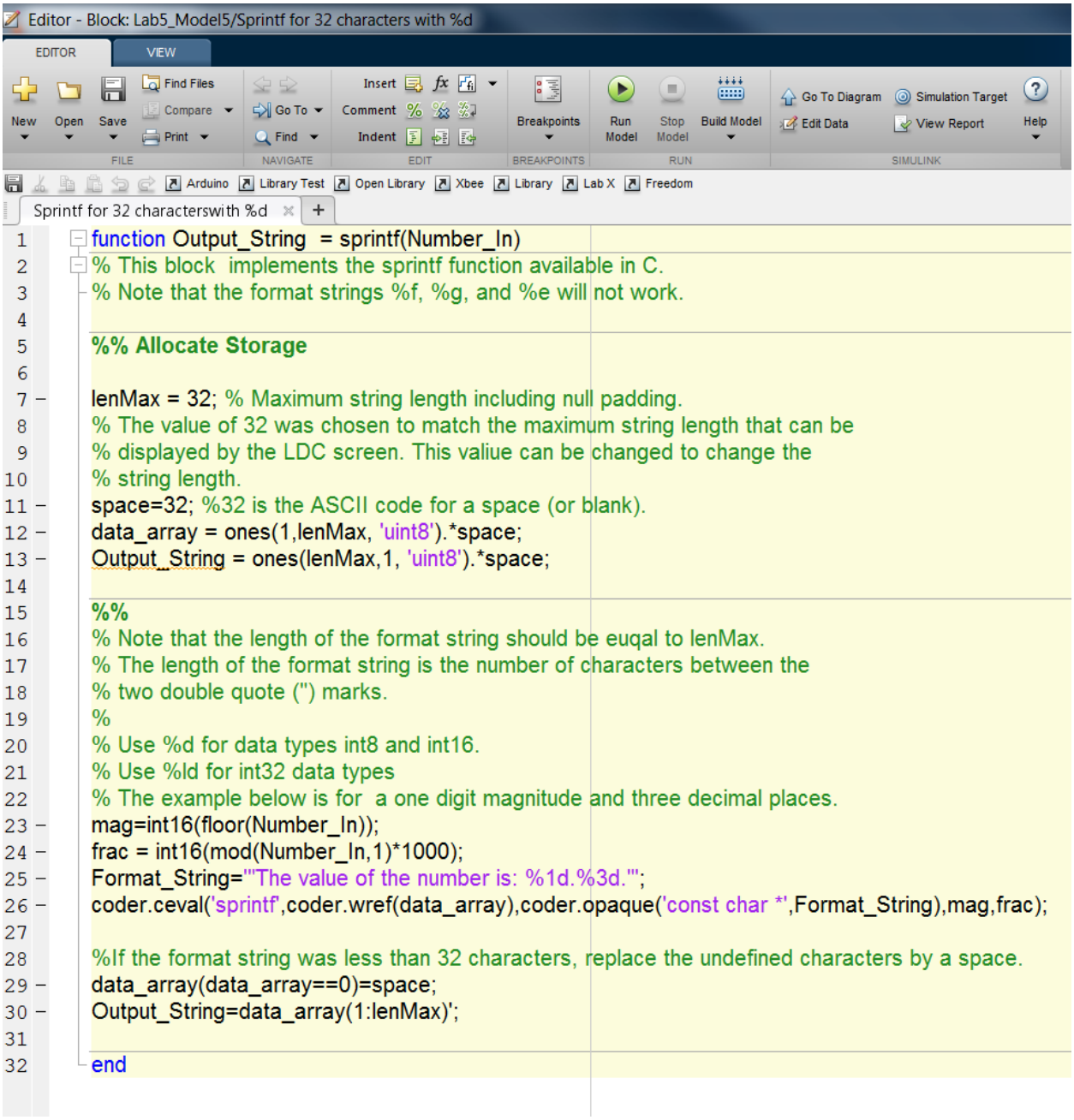


After taking these two steps, and changes we make to the block will not affect the block saved in the library.

We can now edit and modify the sprintf block. Right-click on the sprintf block and then select **Mask** and then select **Look Under Mask:**



The editor will open the MATLAB Embedded function and you can modify it:



You can now modify the function as needed. Typically we will need to change the number of inputs and modify the format string. For this example, since we only have a single input all we need to do is modify the format string. Change the string to **`Format_String='"The value of pi is: %1d.%3d.      "';`**

Demo V.5: Demo the sprint MATLAB Function block displaying the text string, “The value of pi is 3.142.”

Exercise V.5: Create a model that increments a counter from 1 to 10. This value is multiplied by the constant pi. Both numbers are passed to a modified sprint function. The sprint function generates a text string that outputs the following, “The value of x*pi is y.” where x is the counter, and y is the value of the constant times pi. Example outputs are:

The value of 1*pi is 3.14.

The value of 3*pi is 9.42.
The value of 8*pi is 25.13.
The text displayed on the screen should change once every second.

Exercise V.6: Modify the bar graph of Exercise V.4 so that the first line on the LCD displays the text “Bar Graph xxx.x%” where xxx.x is the numerical value of the percent, between 0 and 100. The second line of the LCD should still display the bar graph as in Exercise V.4.

Exercise V.7: The LCD splash screen is the text that the LCD displays at power-up. The text is displayed for half a second and then the LCD goes blank. The datasheet for the LCD gives the procedure for changing the LCD splash screen. This text string should only be sent once. Create a model that sends one text string to the LCD screen such that it changes the splash screen. Change the splash screen so that it displays your name.

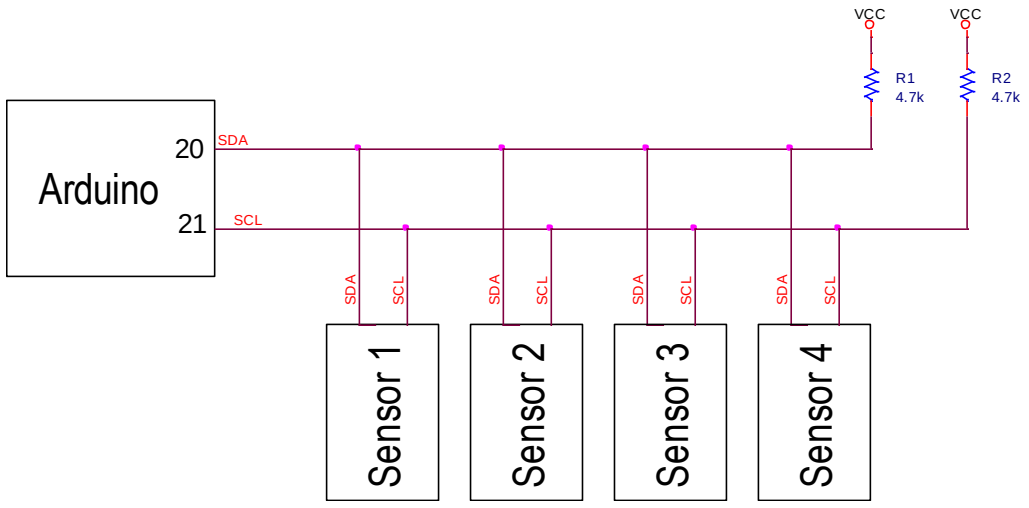
Lab VI
Intelligent Sensors

In this section we will look at intelligent sensors. Typically, these sensors measure some physical quantity, digitize the signal, then perform some signal processing and mathematical transformations on the signal to remove noise and linearize the signal. Typically, analog sensors are non-linear and we need to do some compensation in our circuit to correct the signal. With intelligent sensors, all of the compensation is done inside the sensor. The output signal could be analog or digital. If the sensor output is analog, then we would just read the sensor with one of the Arduino's analog inputs, and we have already demonstrated this with the thermistor. For this section, we will look at sensors that have a digital output and communicate their value as a serial bit stream. In this lab we will study the I^2C bus which is a standard two wire serial data link used by several manufactures for several different types sensors. The RHIT Arduino Library also contains blocks for a Dallas temperature sensor which is a one-wire serial communication method proprietary to Dallas Semiconductor.

Note that serial communication and all of the calculations for filtering and linearization take time. Typical sensors that use a digital serial communication link are much slower than analog sensors. For the digital sensors we demonstrate here, we will poll the sensor once a second or slower. You would typically use Analog sensors if you want higher speed sensors.

A. What is I^2C?

A typical I^2C sensor has four pins, Vcc, ground, SDA and SCL. Vcc and ground are the power pins. Vcc is typically 3.3 V or 5 V. SDA is the bi-directional **s**erial **da**ta line and SCL is the **s**erial **cl**ock line. If you have multiple sensors, all of the SDA lines are connected together and all of the SCL lines are connected together. Pin 20 for the Arduino is the SDA line and pin21 is the SCL line. An example of four sensors connected on a single I2C bus is shown below:



Note that the SCL line and the SDA line require pull-up resistors. Each sensor will have a unique address specified by the manufacturer. You can distinguish between sensors through their address. In our example, the addresses are specified by the manufacturer. If two sensors have the same address, you cannot use them on the same bus.

B. Microchip TC74 Temperature Sensor

Here is where it gets hairy. To use a sensor we need to make a Simulink driver block specific to that sensor. The driver is written in C or C++. Don't panic! The blocks for the sensors used in this manual are contained in the RHIT Arduino Library. For this example, we will use the Microchip TC74.

MICROCHIP

TC74

Tiny Serial Digital Thermal Sensor

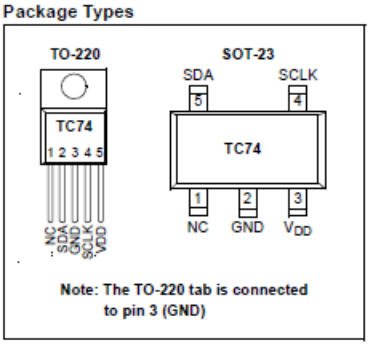
Features

- Digital Temperature Sensing in SOT-23-5 or TO-220 Packages
- Outputs Temperature as an 8-Bit Digital Word
- Simple SMBus/I²C™ Serial Port Interface
- Solid-State Temperature Sensing:
  - ±2°C (max.) Accuracy from +25°C to +85°C
  - ±3°C (max.) Accuracy from 0°C to +125°C
- Supply Voltage of 2.7V to 5.5V
- Low Power:
  - 200 µA (typ.) Operating Current
  - 5 µA (typ.) Standby Mode Current

Applications

- Thermal Protection for Hard Disk Drives and other PC Peripherals
- PC Card Devices for Notebook Computers
- Low Cost Thermostat Controls
- Power Supplies
- Thermistor Replacement

Package Types



Note: The TO-220 tab is connected to pin 3 (GND)

General Description

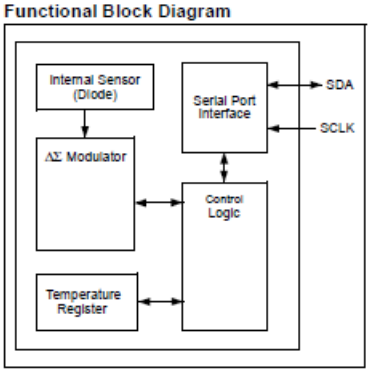
The TC74 is a serially accessible, digital temperature sensor particularly suited for low cost and small form-factor applications. Temperature data is converted from the onboard thermal sensing element and made available as an 8-bit digital word.

Communication with the TC74 is accomplished via a 2-wire SMBus/I²C compatible serial port. This bus also can be used to implement multi-drop/multi-zone monitoring. The SHDN bit in the CONFIG register can be used to activate the low power Standby mode.

Temperature resolution is 1°C. Conversion rate is a nominal 8 samples/sec. During normal operation, the quiescent current is 200 µA (typ). During standby operation, the quiescent current is 5 µA (typ).

Small size, low installed cost and ease of use make the TC74 an ideal choice for implementing thermal management in a variety of systems.

Functional Block Diagram



Datasheets for the TC74 are readily available on-line and are not provided within this manual. For this lab we will be using the TO-220 package as it is large enough for us to easily use on a breadboard. In addition, the metal tab allows us to bolt it to something to measure its temperature.

Note that the TC74 actually has eight possible addresses. Multiple addresses are available so that we can use several TC74 temperature sensors on the same I2C bus. The addresses are specified as part of the part number:

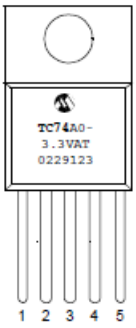
SOT-23 Package Marking Codes

SOT-23 (V)	Address	Code	SOT-23 (V)	Address	Code
TC74A0-3.3VCT	1001 000	V0	TC74A0-5.0VCT	1001 000	U0
TC74A1-3.3VCT	1001 001	V1	TC74A1-5.0VCT	1001 001	U1
TC74A2-3.3VCT	1001 010	V2	TC74A2-5.0VCT	1001 010	U2
TC74A3-3.3VCT	1001 011	V3	TC74A3-5.0VCT	1001 011	U3
TC74A4-3.3VCT	1001 100	V4	TC74A4-5.0VCT	1001 100	U4
TC74A5-3.3VCT	1001 101*	V5	TC74A5-5.0VCT	1001 101*	U5
TC74A6-3.3VCT	1001 110	V6	TC74A6-5.0VCT	1001 110	U6
TC74A7-3.3VCT	1001 111	V7	TC74A7-5.0VCT	1001 111	U7

Note: * Default Address

TO-220 Package Marking Information

TO-220

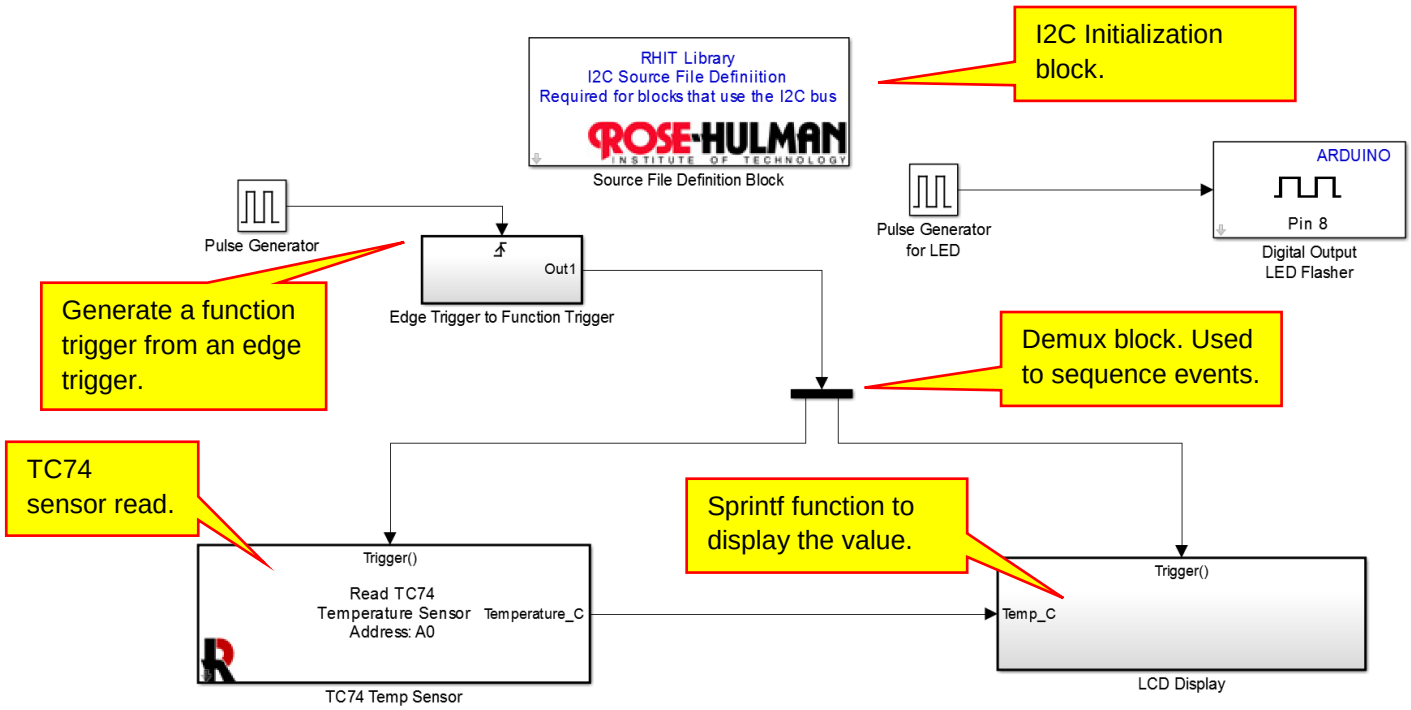


Legend: XX...X Customer specific information*
YY Year code (last 2 digits of calendar year)
WW Week code (week of January 1 is week '01')
NNN Alphanumeric traceability code

Note: In the event the full Microchip part number cannot be marked on one line, it will be carried over to the next line thus limiting the number of available characters for customer specific information.

* Standard marking consists of Microchip part number, year code, week code, and traceability code.

Create the top-level model shown below. Note that a block of the TC74 is located in the **Sensor Read Functions** portion of the **RHIT Arduino Library**:

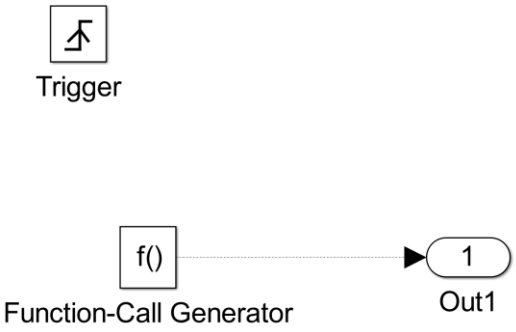


To change the address of the TC74 (addresses are A0, A1, A2, etc.), double-click on the block and specify the address with the pull-down menu. Note that to use the I2C sensors, you need to place one copy (only one copy) of the RHIT Source File Definition Block in your model:

Source File Definition Block

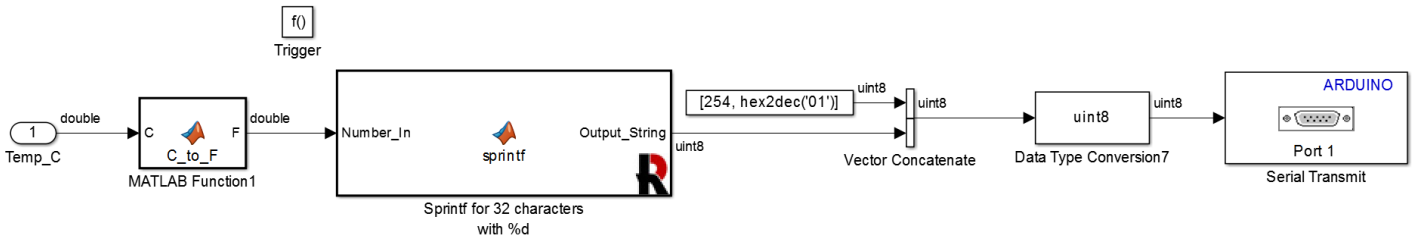
This block defines the libraries for using the I2C bus. You can have more than one I2C sensor in your model. However, only one copy of this block can be placed in your model.

We would like to read the temperature sensor every two seconds, and after reading the sensor, display the value of the sensor on the LCD display. Almost everything in this model has been covered in earlier labs. The contents of the **Edge Trigger to Function Trigger** subsystem is shown below and was covered in Section V.C.2.b) on page 98:

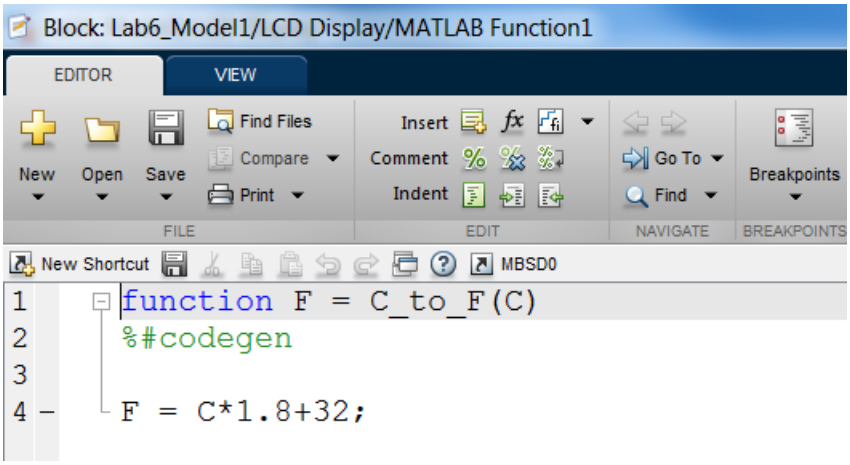


Remember also that we can sequence events using a function trigger and a **Demux** block. (See page 104.) The pulse generator will produce a square wave with a period of 2 seconds (1/2 Hz). The **Edge Trigger to Function Trigger** subsystem will convert the positive edge to a function trigger. The **Demux** will pass the function trigger to the two subsystems, triggering the **Read Temperature Sensor** subsystem first and then the **LCD Display** subsystem second.

The contents of the LCD display subsystem are shown below:

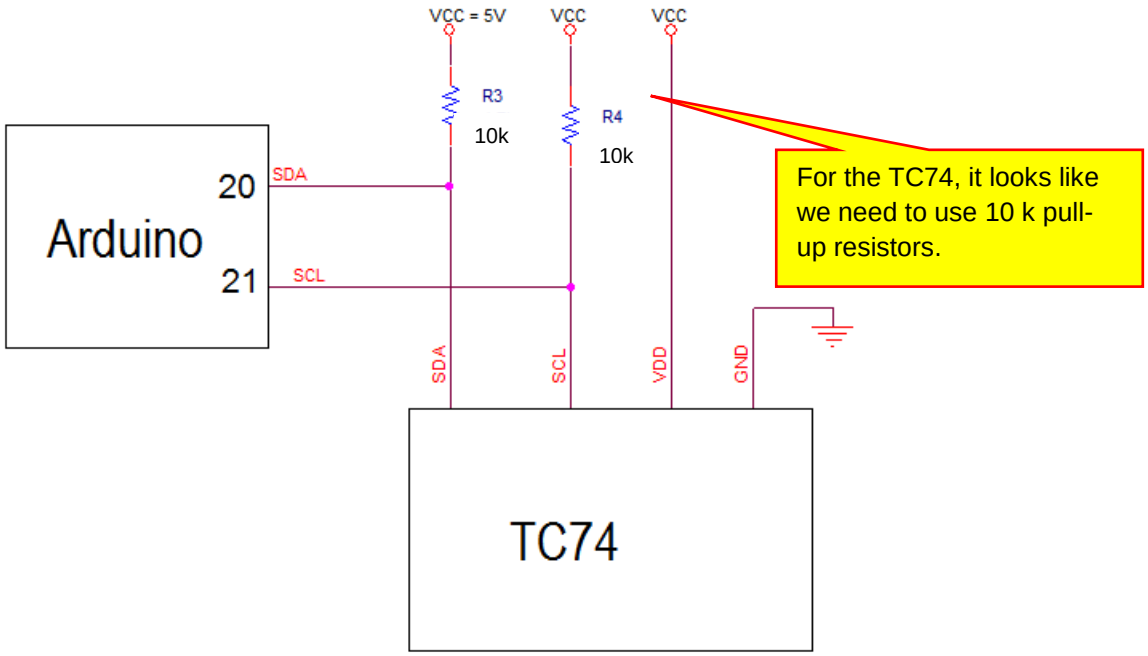


We have seen all of this subsystem previously except for the **MATLAB Function** block that converts a temperature in Celsius to a temperature in Fahrenheit. This subsystem is shown below:



The conversion from Celsius to Fahrenheit could have been done in many different ways. Here we chose to use the MATLAB Function block. The remainder of the LCD Display subsystem has been covered earlier. The format string used in the sprint MATLAB Function block is: `"The temperature is: %3d.%1d F.  "`

We are done with the modifications to the file. Save the changes. You should be able to build and download the model to your Arduino Mega now. Wire your sensor to your Arduino Mega as shown:

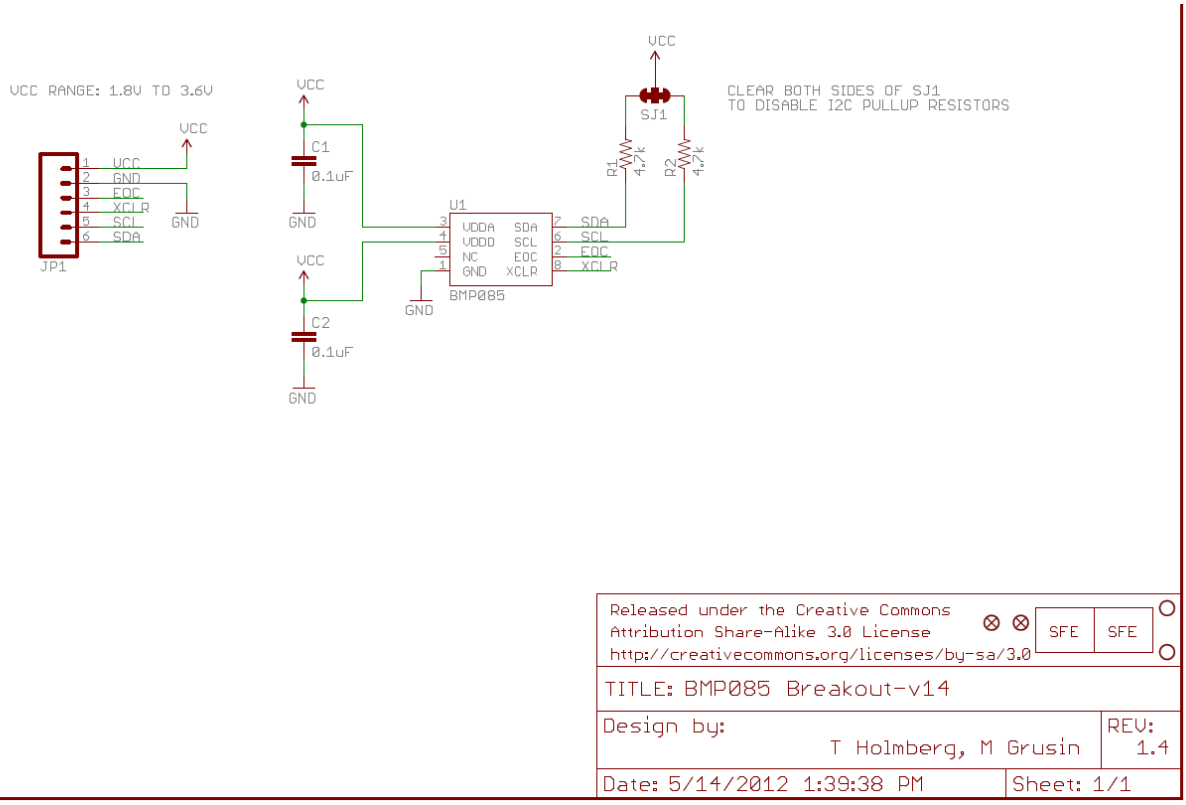


Demo VI.1: Demo of the TC74 temperature sensor with display on the LCD screen.

1. Sparkfun BMP085 Pressure and Temperature Sensors

For a second example of an I2C sensor, we will use a BOSCH BMP085 pressure and temperature sensor. This is a surface mount device, so we need to use a breakout board from Sparkfun Electronics to access the pins on the sensor. A breakout board is basically a small printed circuit board with large pins on it so that we can use surface mount devices that would otherwise be too small for us to work with. The Sparkfun website has a large amount of documentation on the BMP sensor and the breakout board. Also, if you search the web for “Arduino BMP085 library,” you will find a zip file containing code examples and a BMP085.cpp library. For this course, in general, we will use public domain code and adapt it to using with the Simulink S-Function builder block.

First, we will look at the schematic for the breakout board. This schematic was downloaded from the Sparkfun Electronics website:



[2]

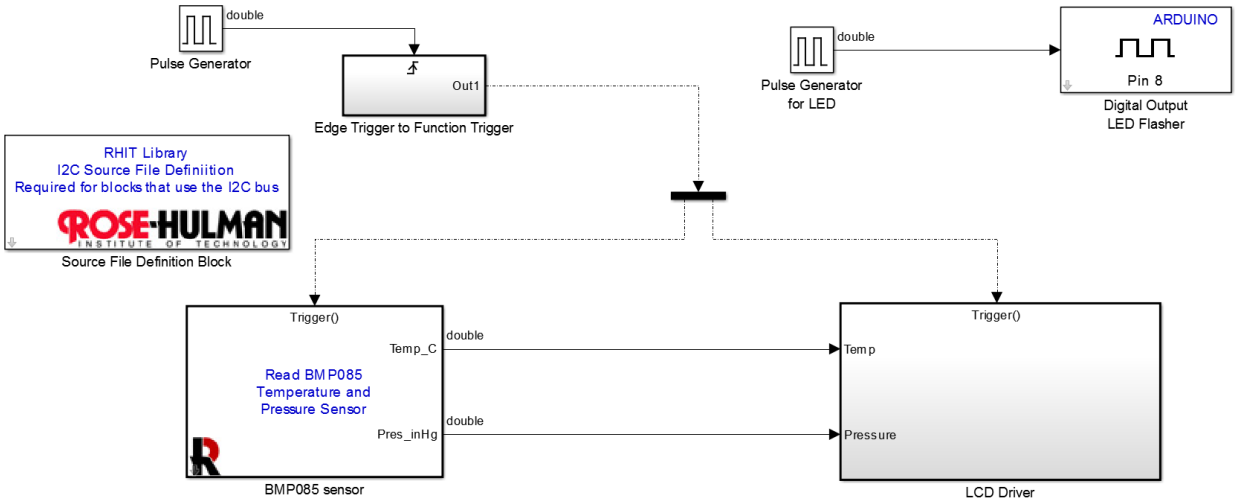
We see that all this board really does is provide access to the sensor so that we can easily hook up power, ground, and the I2C bus wires SCL and SDA. We do note that the board has pull-up resistors already installed, so we do not need to add pull-up resistors. If you place this sensor on an I2C bus with the TC74 we used in the previous section, remember to remove the pull-p resistors we used for the TC74. If we place a bunch of I2C sensors on the same bus, we only need to have one pull-up resistor on each of the bus wires. Also, the maximum sensor voltage for the BMP085 is 3.6 V. **Make sure that you use the 3.3V power supply from your Arduino board.**

Before we create our model, we need to see the format and scaling of the data that the sensor will provide. Referring to the BMP085 data sheet, we see the following:

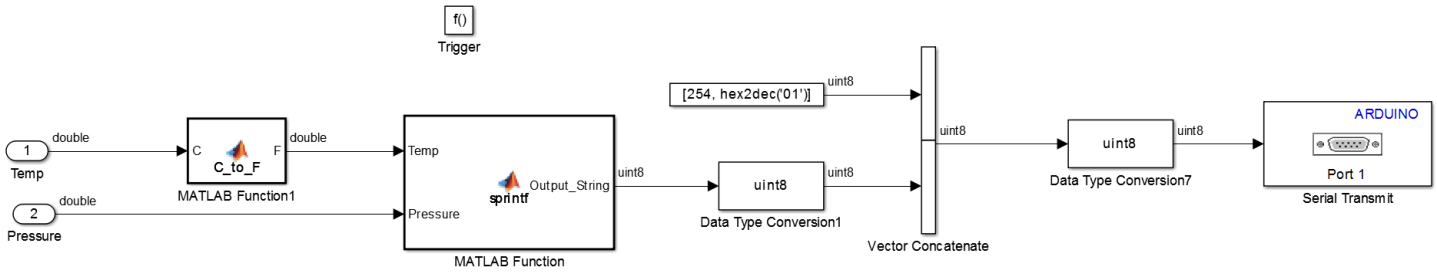
- The temperature measurement is between 0 and 65 °C.

- The resolution of the temperature measurement is 0.1 degrees per bit.
- The pressure measurement is between 700 hPa and 1100 hPa. Note that hPa is hectopascal, where 1 hPA is equal to 100 pascal.
- The resolution of the pressure measurement is 0.01 hPa per bit.

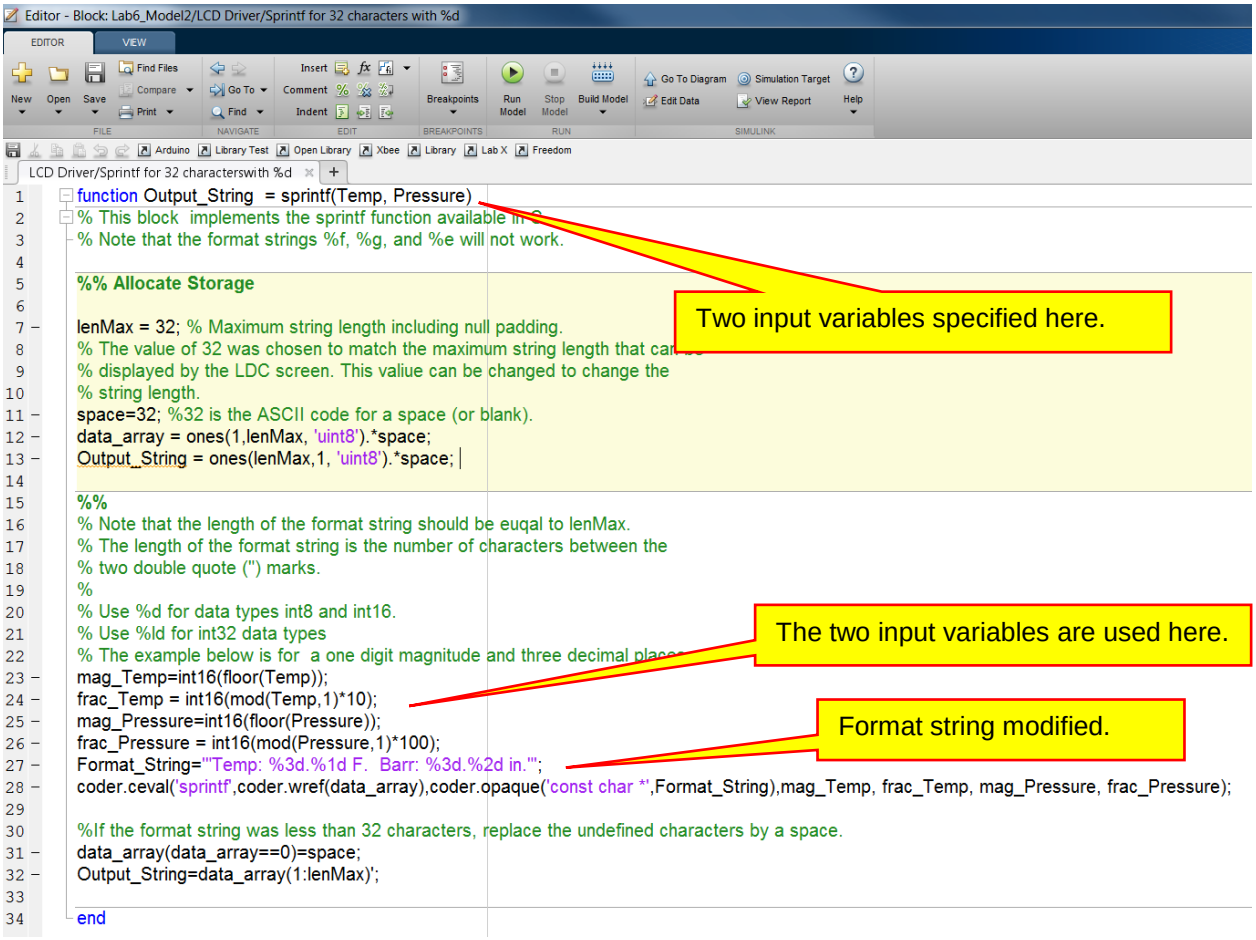
A driver block for the BMP085 is provided in the RHIT Arduino Library. The top level of the model is shown below:



This looks almost the same as the TC74 model except that our sensor has two outputs rather than one. The LCD driver is shown below:



The **sprint** function has changed slightly because we have modified it to convert two numbers rather than one:



You should now be able to download and run your model on your Arduino.

Demo VI.2: Demo of the BMP085 sensor by displaying the temperature and pressure on the LCD screen. The values should update once every 2 seconds.

Exercise VI.1: Create a model that reads the temperature twice, once with the BMP085 and once with the TC74. The model displays both temperatures on the LCD screen for comparison. The display should indicate which temperature is for which sensor.

2. Real-Time Clock

For a final example, we will work with a real-time clock module that communicated through an I2C bus. We will use the Sparkfun Electronics Real Time Clock Module which is a breakout board for the Dallas Semiconductor DS1307 Serial Real-Time Clock / Calendar. This semiconductor uses I2C serial communication to provide date and time information. Two blocks are provided:

- Read Real Time Clock – This function reads the current time. This read will fail if the real time clock has not been set yet.

- Set Real Time Clock – This function writes a set time to the real time clock module. If the RTC has not yet been set, you will need program it with the current date and time. Once set, the RTC breakout board has a battery that will maintain the current date and time. All this block does is program the RTC once and then it does nothing else.

Exercise VI.2: Create a model that programs the RTC with the current date and time. After running this model once, do not run it again as it will reset the date and time to the values you specified the last time you set the clock. Note that you need to specify the current time and date in the block parameters.

Exercise VI.3: Create a model that displays the date and time in the format shown below. The display time should update once a second. You might need to use the %c format string to display a "/" on the LCD display.

```
Time: 10:48:59
Date: 04/14/2012
```

Exercise VI.4: Create a model that reads the three I2C sensors we have studied and flips the display between the information received from each. The LCD display should change screens once a second. You might need to use the %c format string to display a "/" on the LCD display. The alternating screens are shown below:

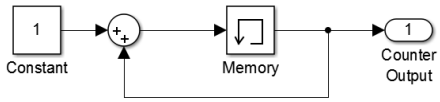
```
Time: 10:48:59
Date: 04/14/2013

Temp: 37.9 F.
Barr: 29.50 in.

TC74 Temperature
Temp: 37.9 F.
```

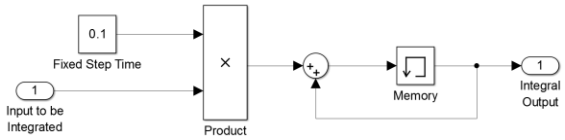
Lab VII
Memory

In this section we will look at ways to store data on the Arduino microcontroller built-in memory. We will use volatile RAM and the static EEPROM for these examples. Volatile means that when we cycle the power to or reset the microcontroller, the information is lost. For the EEPROM memory, we can cycle the power or reset the microcontroller and not lose the stored values. Reading from and writing to the EEPROM memory, however, is very slow

There are several ways to store variables in Simulink. One obvious method is Stateflow. Inherently, a Stateflow chart has memory because the chart remembers where it is and knows here it has been. (How you entered a state can make if a difference if you so decide.) Also, you can define array variables in Stateflow if needed. Simulink has the Memory block 

. The output of this block is the input value from the previous time step. Thus it remembers its last value. Simulink structures like those shown below can be used as counters and integrators:



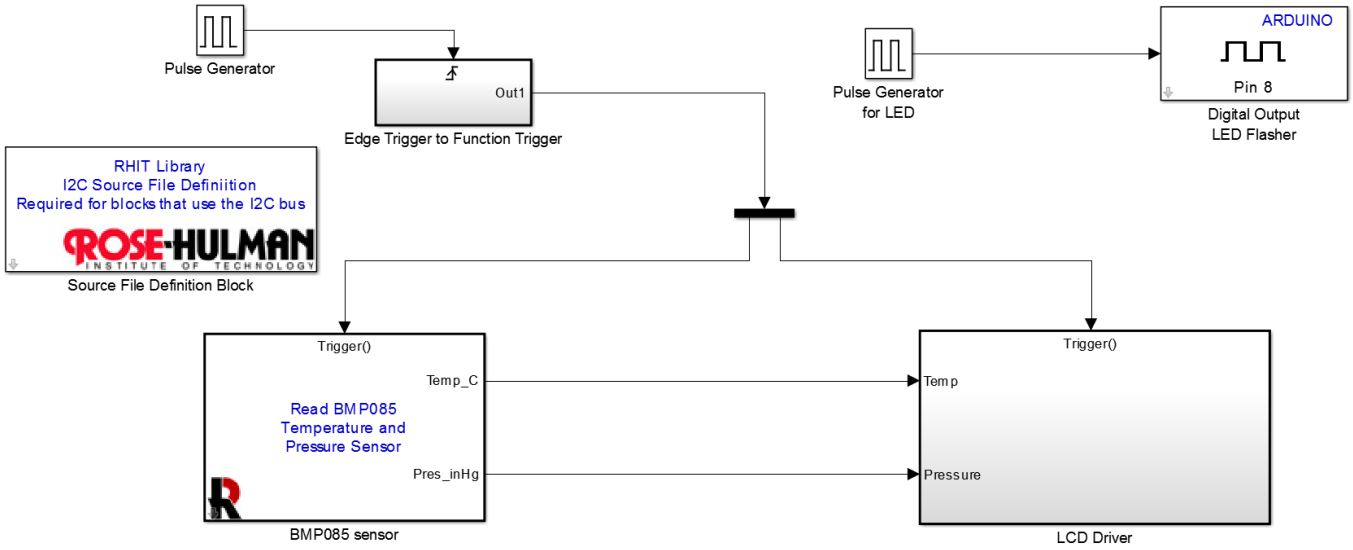


The Simulink **Data Store** blocks also allow us to store data and access them at their current level or below:

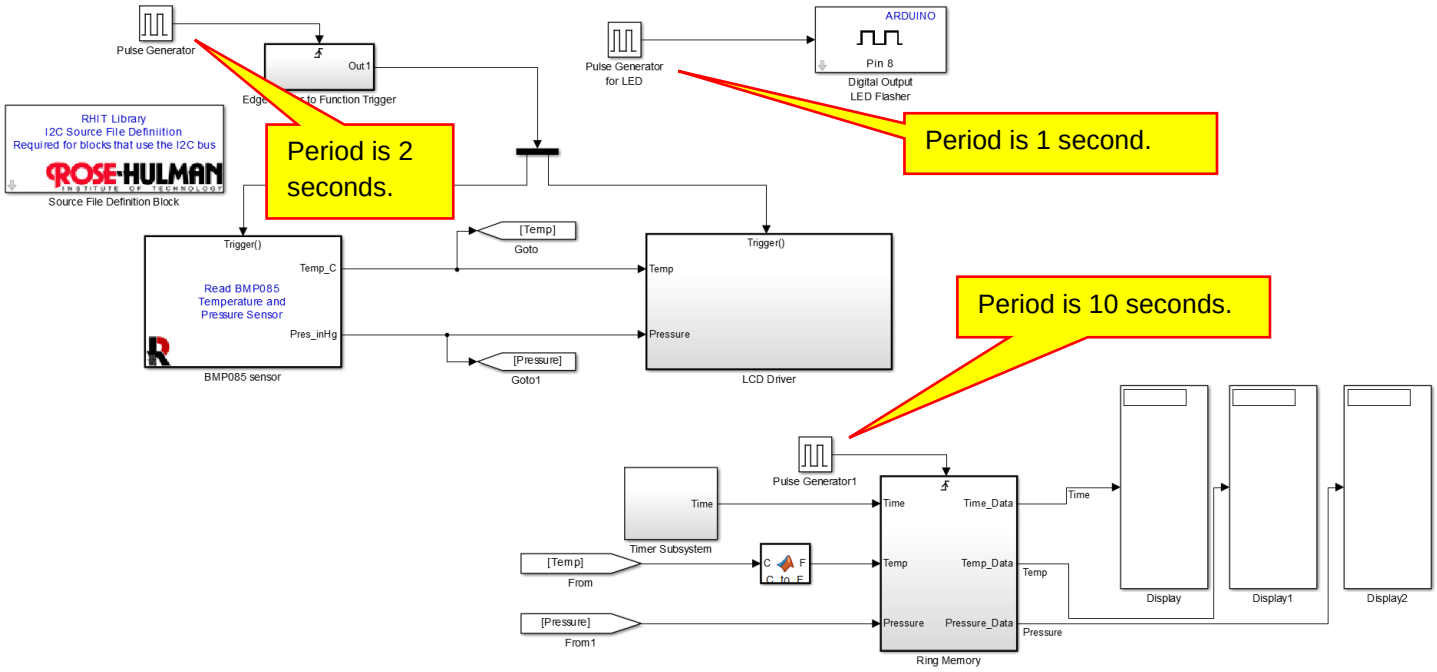
Instead of using these blocks or Stateflow, we will use the MATLAB Function block to create the data structures we need.

A. Array Memory

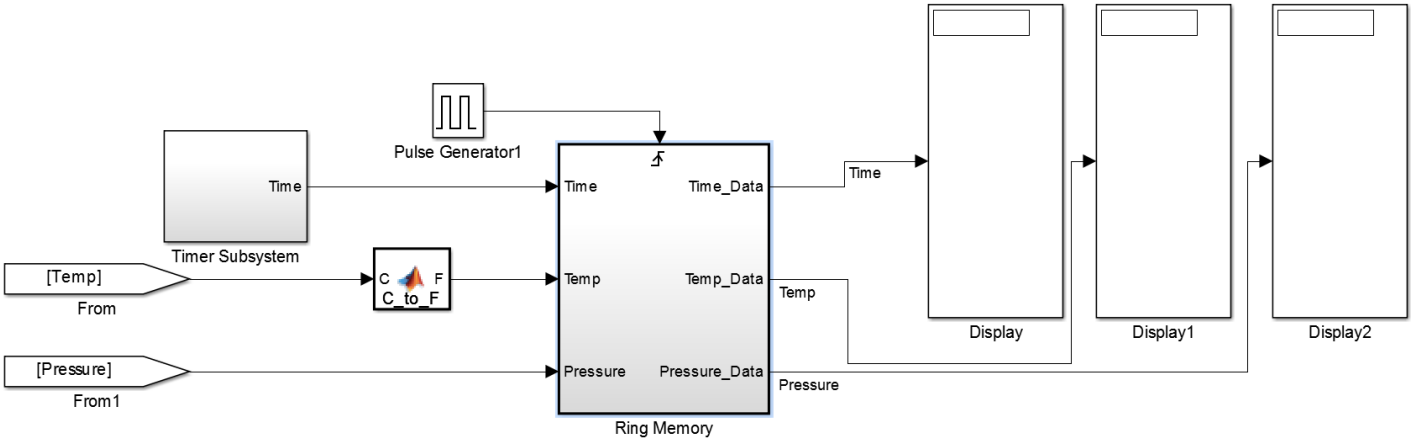
The MATLAB Function block provides access to a subset of MATLAB commands in the Simulink environment. This will allow us to use many of the MATLAB data structures in Simulink. Here we will set up a simple array to store values we read from the BMP085 pressure and temperature sensor we developed in Section VI.B.1. You should be able to reuse parts of those models here. We will start with the BMP085 model developed in Lab 6:



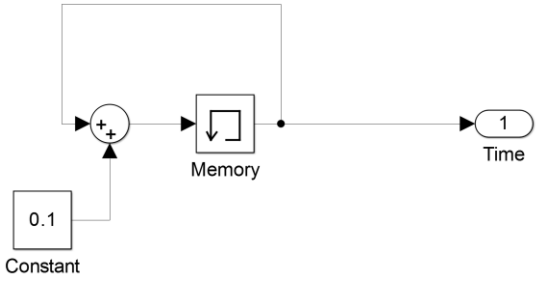
In addition to sending the measured values to the LCD, we will store them in memory. We will record the time, temperature, and pressure signals in an array of ten values. We will create the memory structure using the **MATLAB Function** block. The complete Model is shown below:



The BMP085 sensor and LCD portions of the model are copied from Section VI.B.1. This portion of the model reads the sensor every two seconds and displays the values on the LCD screen. The new portions of the model, detailed in the screen below, records the time, temperature, and pressure every 10 seconds and stores them in static RAM on the controller board:

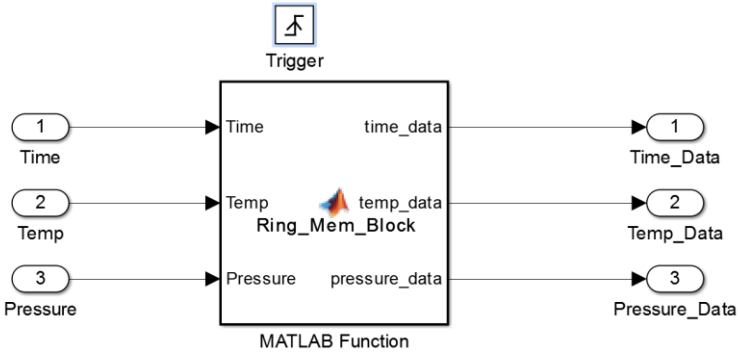


Instead of using the RTC clock module from Section VI.B.2, we will create our own timer using an integrator. The timer subsystem is shown below:

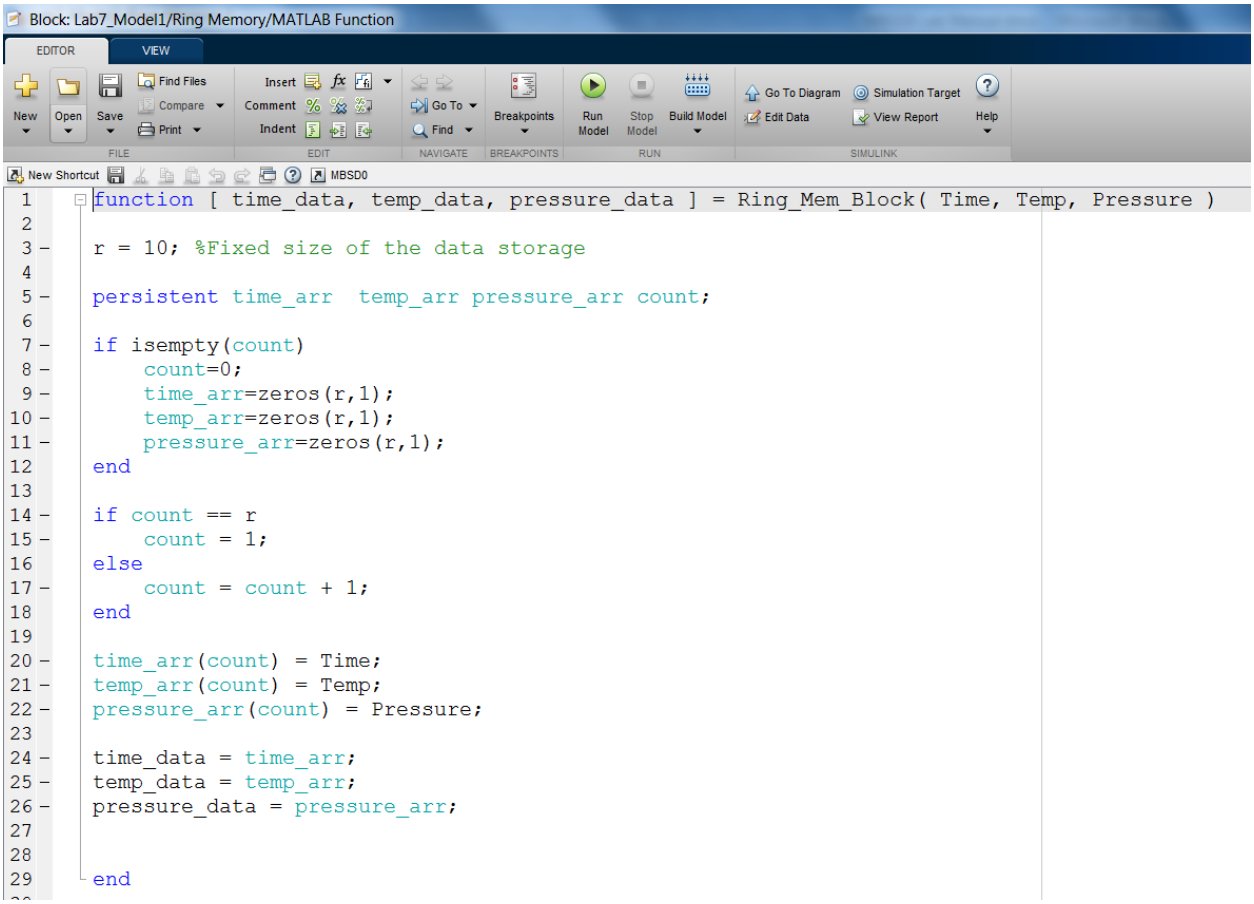


We will set up the model to run with a discrete solver using a fixed time step of 0.1 seconds. Thus, this subsystem will execute once every 0.1 seconds. The output of the memory block is its input from the previous time step. Thus, every time this subsystem runs, it adds 0.1 seconds to the previous output. Thus, as long as the constant is equal to the fixed time step, the output counts up in time. This method is accurate as long as the model runs in real time and the time step is accurate.

The Ring Memory block is an edge triggered subsystem that contains a MATLAB Function block:



The MATLAB Function is shown below:



The line **`r=10;`** defines the size of the memory storage. We are going to create three one dimensional arrays. **`r`** defines the size of these arrays. For this example, each array will have 10 elements. You can change the size of the arrays by changing the value of **`r`**.

The line **`persistent time_arr temp_arr pressure_arr count;`** defines the listed variables as persistent. This means that these variables retain their values between calls. Without this line, each time the MATLAB Function block is called, the variables would be re-initialized and not retain any of their previous values. Effectively, the persistent command allocates our memory storage in RAM and instructs the controller to retain the values until the controller is restarted.

```
if isempty(count)
    count=0;
    time_arr=zeros(r,1);
    temp_arr=zeros(r,1);
    pressure_arr=zeros(r,1);
end
```

The first time the function runs, the persistent variables will not be defined. The code segment below checks to see if the persistent variables are defined. If they are not defined, the arrays are defined, the memory is allocated, and the variables are initialized to zero. If the variables have been previously defined, the lines will be skipped, and the values will not be set to zero. (They will retain the values they had from the previous time the block was executed.)

The code segment below implements a rolling counter.

```
if count == r
    count = 1;
```

```
else
    count = count + 1;
end
```

For **r** elements, the count goes from 1 to **r**. When the **count** hits the value of **r**, the next value of **count** is reset to 1. The **count** then increases again. Thus, **count**, which is the index to our memory counts from 1 to the max value, and then restarts again at 1 and continues circulating until we stop the microcontroller. With this method, the memory records the last **r** data points. We will have to keep track of the beginning and end of the array, but that is the task of a future lab. For now, we will just record values. Also note that you could just have the **count** go from 1 to the end and stop there, recording only the first **r** values. In either case, the goal here is memory storage. With the flexibility of MATLAB, you can design your memory storage for your particular application.

The lines below:

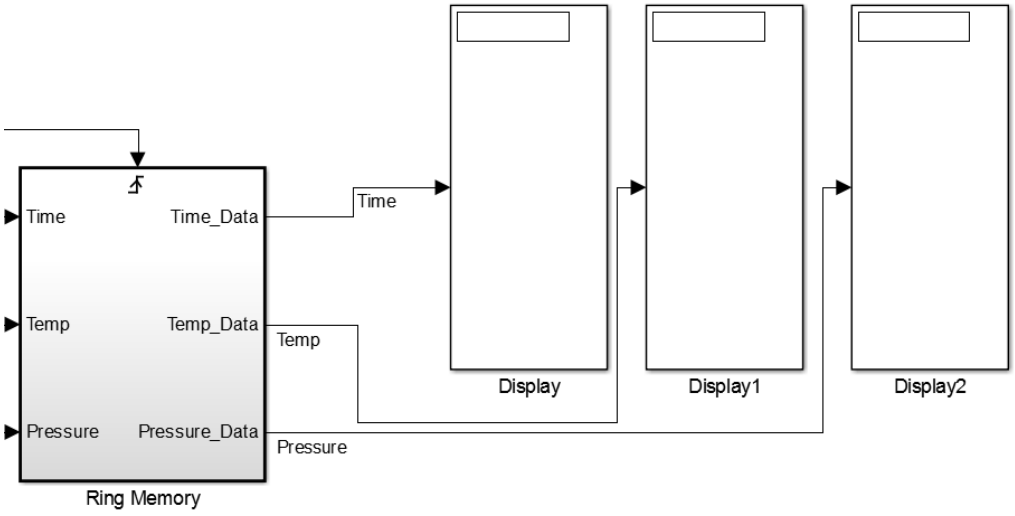
```
time_arr(count) = Time;
temp_arr(count) = Temp;
pressure_arr(count) = Pressure;
```

store the present values in the array. Only one value for each measurement is stored. The next lines specify the output of the function.

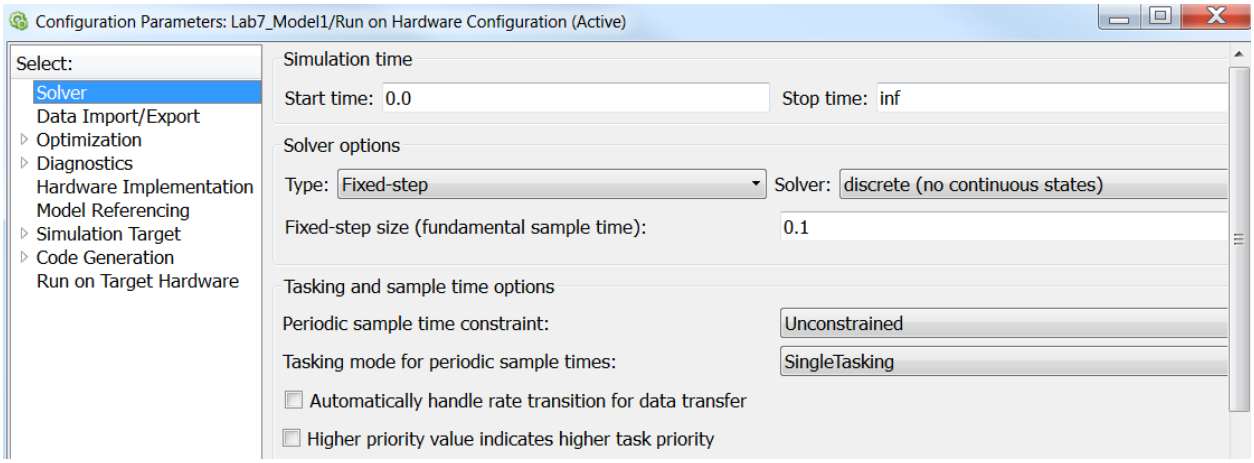
```
time_data = time_arr;
temp_data = temp_arr;
pressure_data = pressure_arr;
```

Note that the outputs of the function are the complete arrays of all of the stored data. Note again that we could select a single point if we wanted.

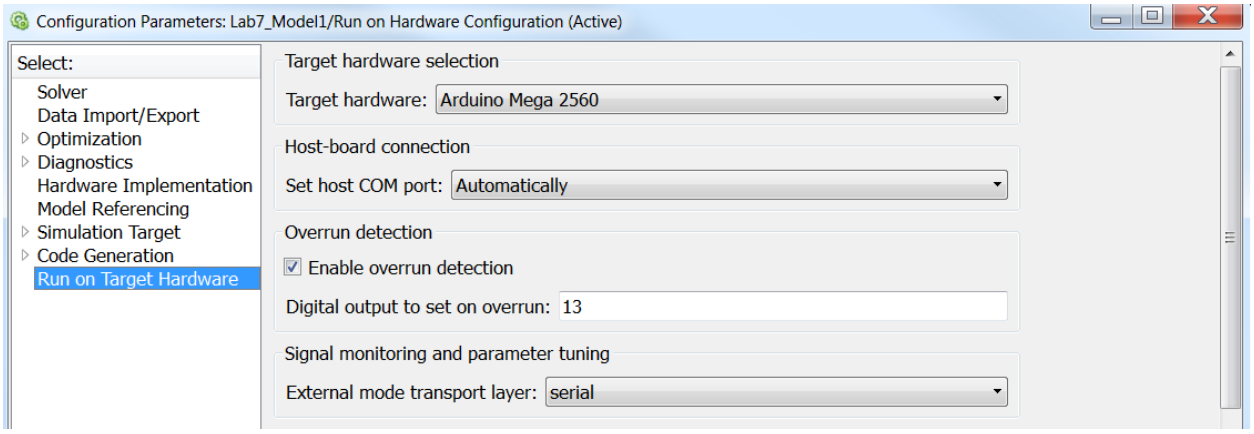
Lastly, we will have the output of the arrays go to **Display** blocks (**Simulink** / **Sinks** library). We have them elongated because each block will display an array of values, in this case 10 elements each.



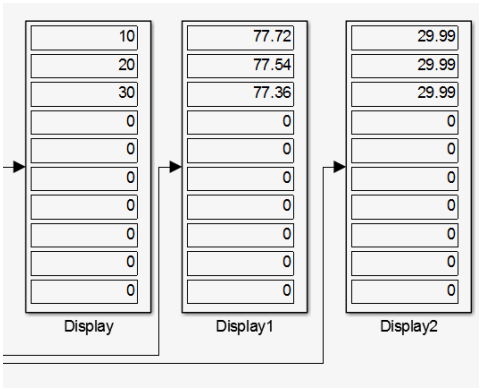
Lastly, we will look at the **Model Configuration Parameters**. First, we are choosing a fixed step size of 0.1 seconds and a discrete solver. Note that if we change the fixed step size, we will also have to change the constant in the **Timer Subsystem** to match the step size:

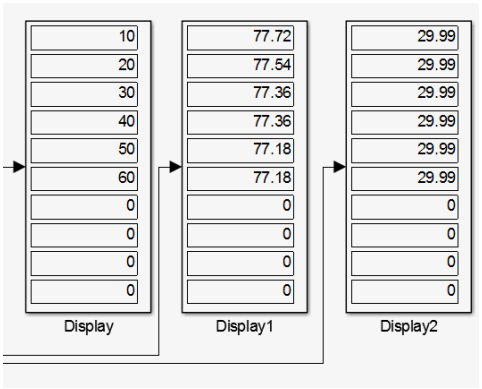


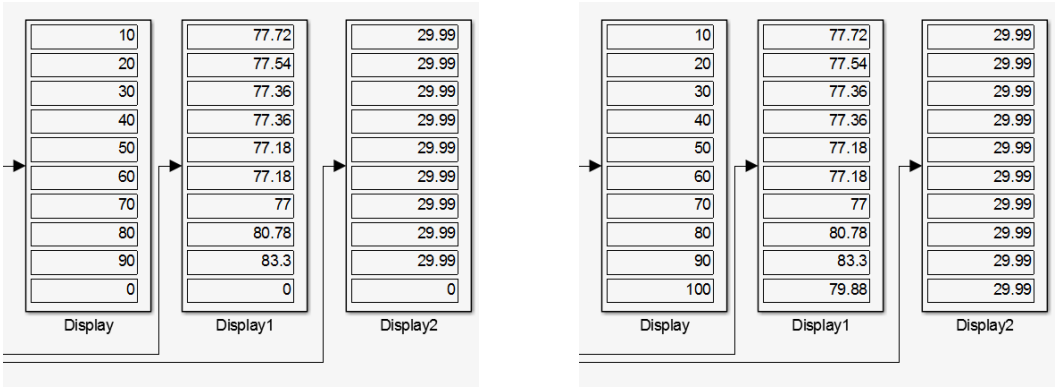
In the **Run on Target** pane, **Enable overrun detection**:



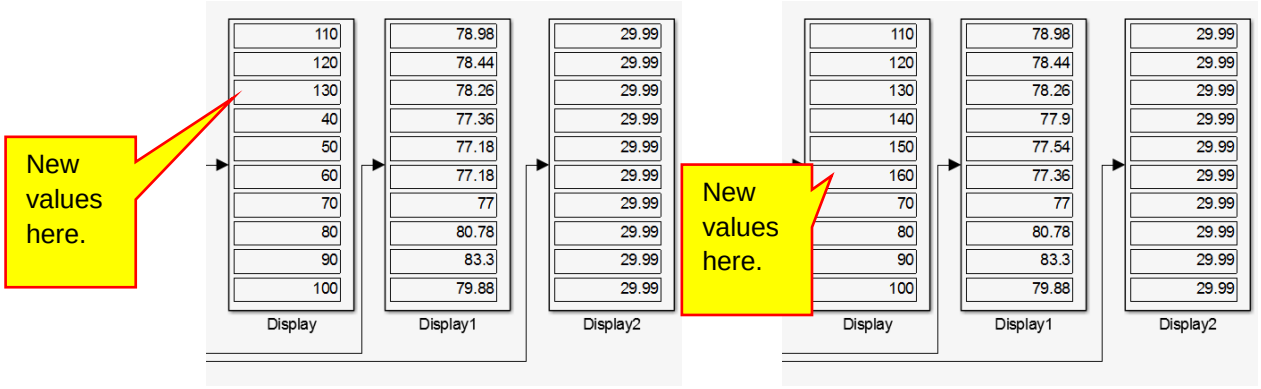
Since this model has a lot of data, we will most likely have an overrun. However, the model will run sufficiently to demonstrate its operation. Run your model on your target and enable external mode. (See Section II.C.1on page 44 to for help.) You will see the memory add a new value every 10 seconds. The LED should flash at a 1 Hz rate. Data values are read from the BMP085 once every two seconds. Data is stored in the memory every 10 seconds. Below are several screen captures showing that the memory is updated every 10 seconds:





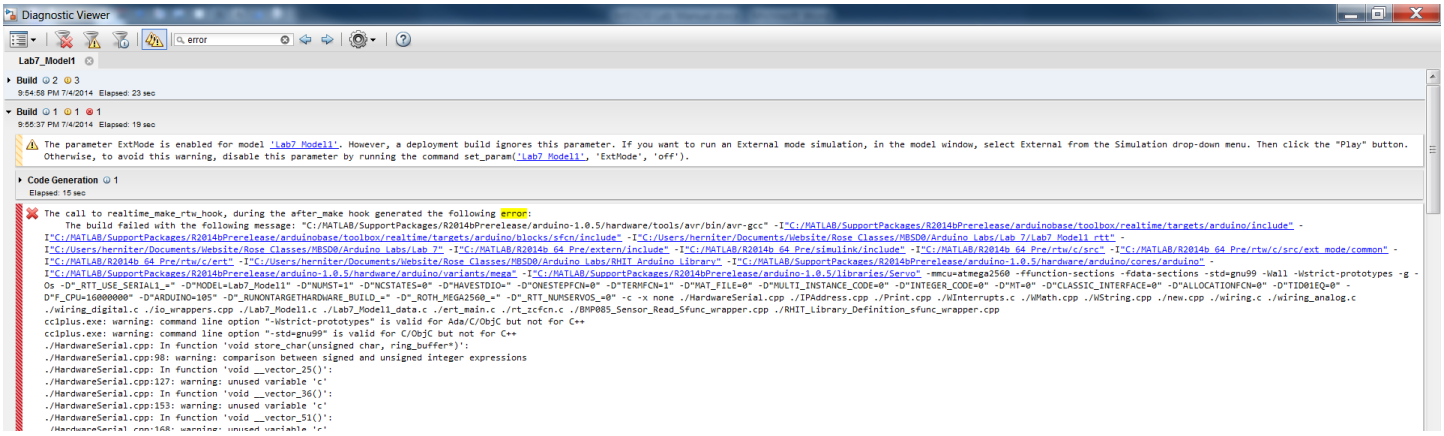


The screen capture below shows that the memory index has rolled over and restarted at the beginning:



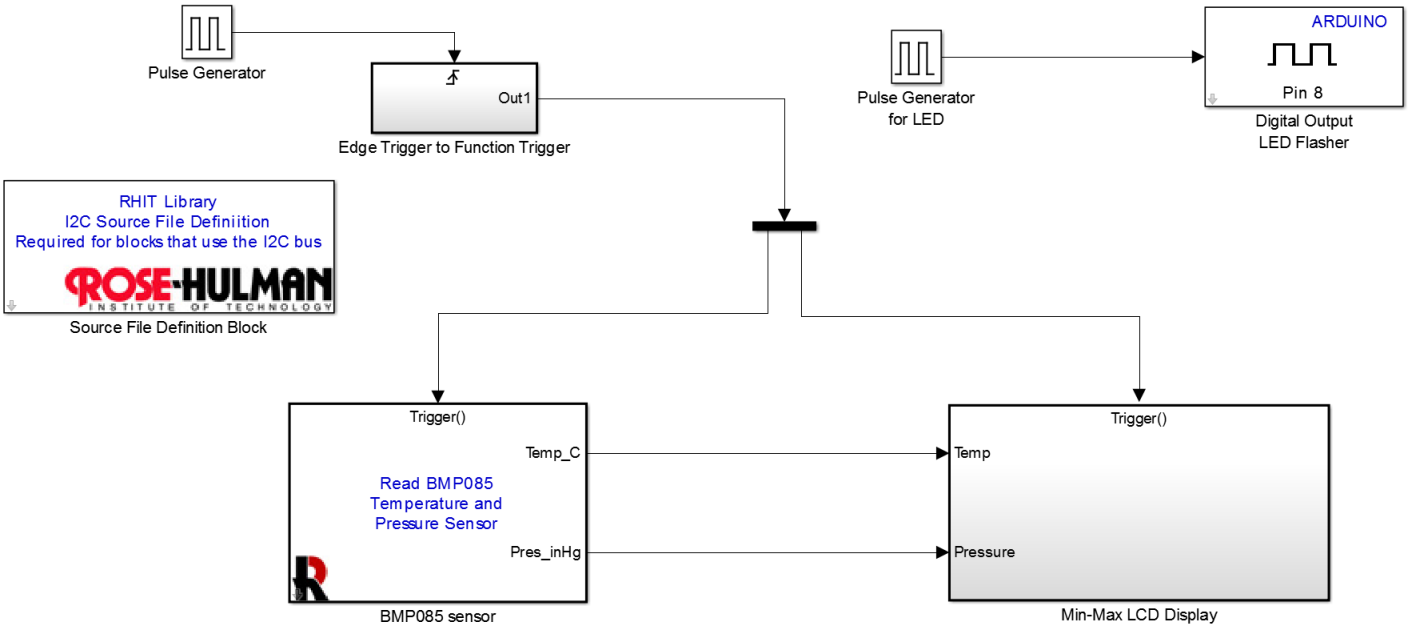
Demo VII.1: Demonstrate the working memory block with 10 elements.

Exercise VII.1: Determine the maximum size of this ring memory that you can use on the Arduino Mega. When the memory is too large, you will get an error screen similar to the one below when you try to run the model on the Arduino.

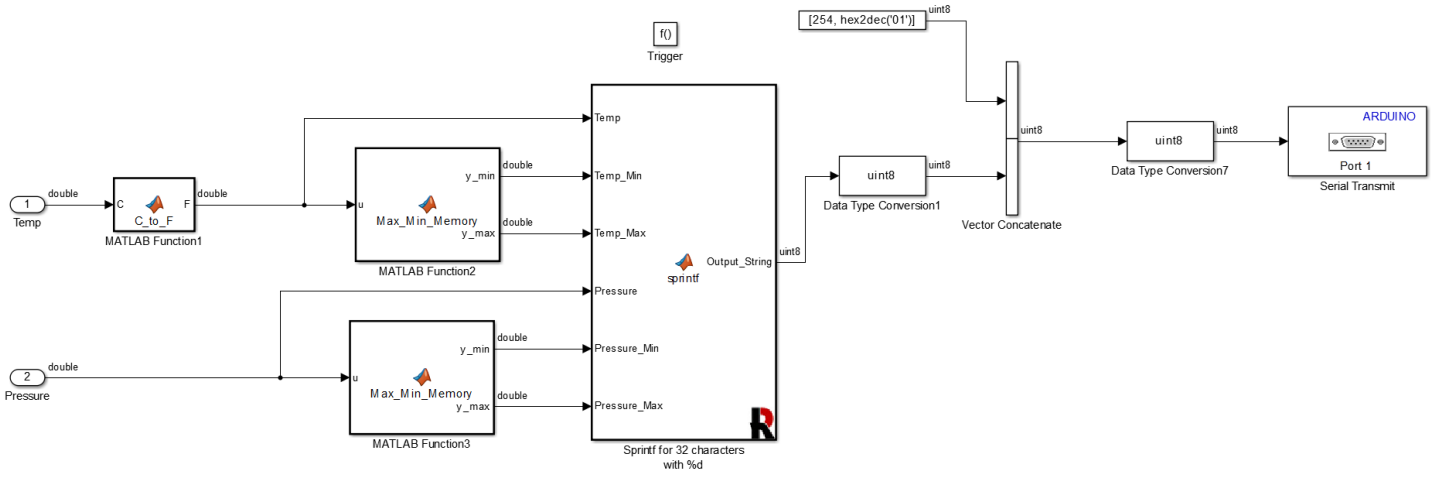


B. Min Max Memory

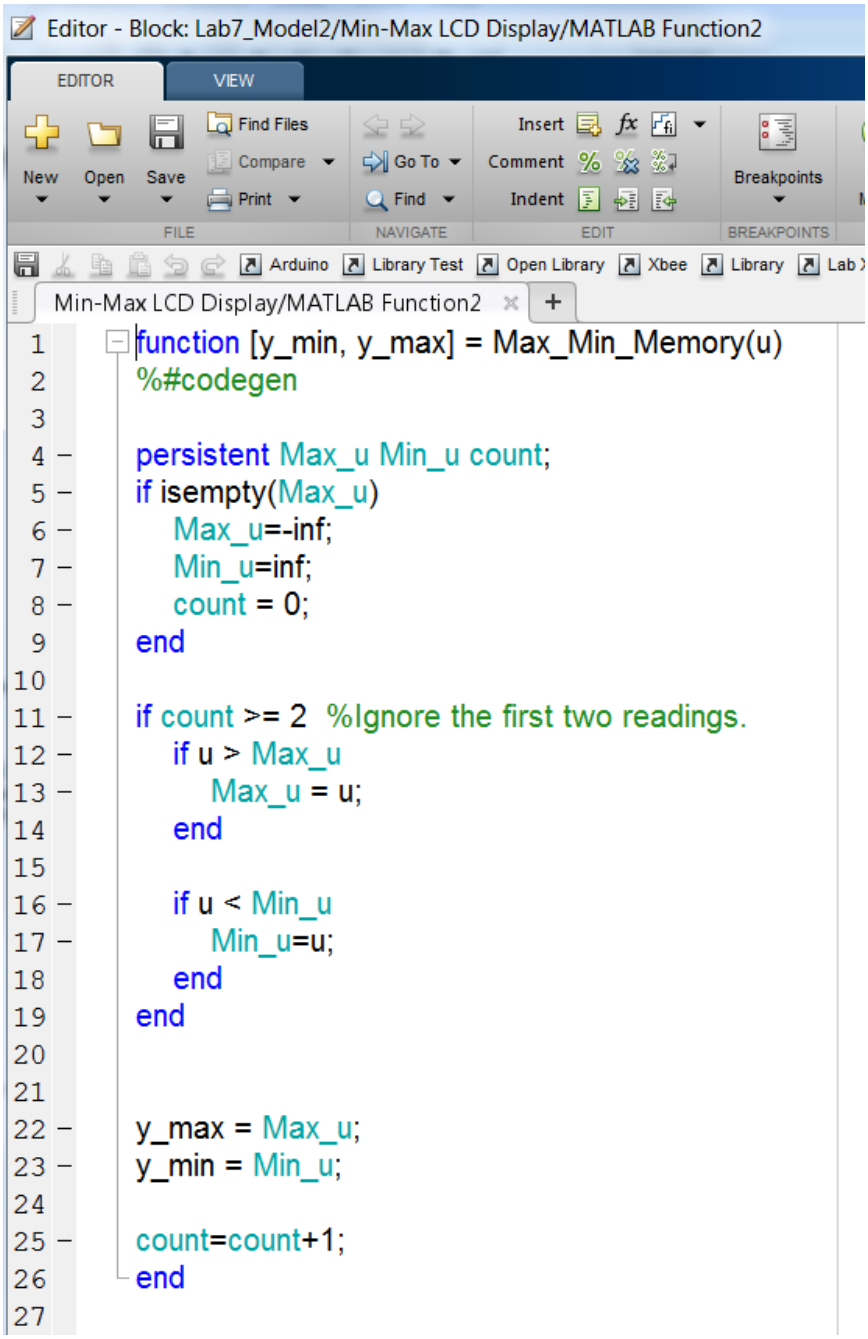
In the next example, we will keep track of the min and max values of the BMP085 sensor outputs using a MATLAB Function block. The top level of the model is shown below:



The model looks similar to what we had before except that the LCD subsystem has been modified. Looking inside the Min-Max LCD Display, we see the following:

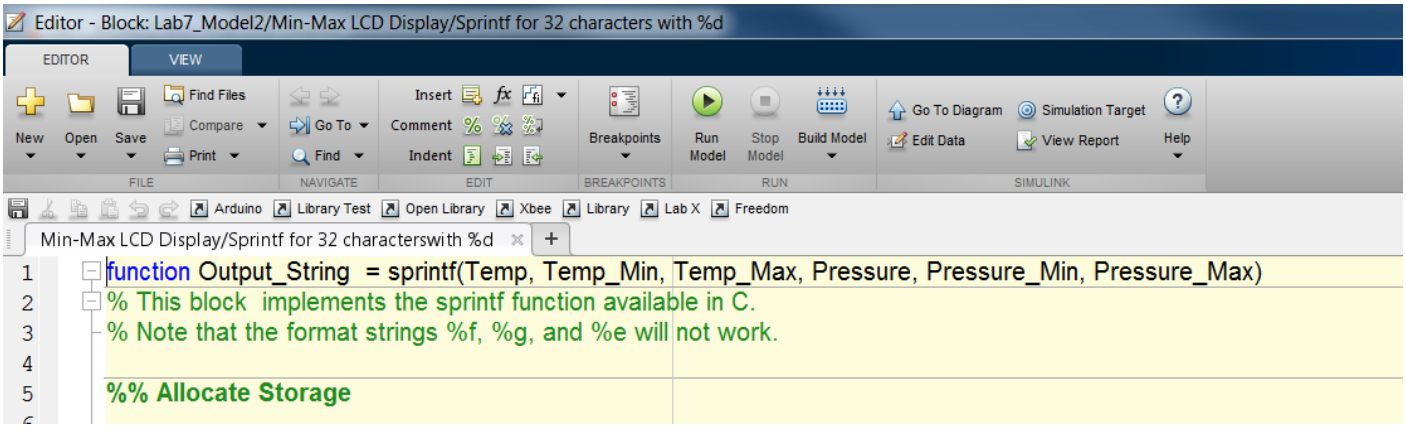


We have added two new MATLAB Function blocks called **Min_Max_Memory** and modified the sprint function. Both **Min_Max_Memory** blocks are the same and shown below:



The function defines **Max_u** and **Min_u** as persistent, so that the function will remember those values. **Min_u** is initialized to infinity so that the next valid reading will be less than the initial value and recorded as the current minimum. **Max_u** is initialized to negative infinity so that the next valid reading will be more than the initial value and recorded as the current maximum. Finally, the first two readings are ignored as the initial values returned by the sensor are not always valid. The block outputs the maximum and minimum values, and these values are held until the Arduino microcontroller is reset of the power is cycled.

The **sprintf** function has been modified to display the minimum value, current value, and the maximum value of the temperature and pressure values:



and

```
mag_Temp=int16(floor(Temp));
frac_Temp = int16(mod(Temp,1)*10);
mag_Temp_Min=int16(floor(Temp_Min));
frac_Temp_Min = int16(mod(Temp_Min,1)*10);
mag_Temp_Max=int16(floor(Temp_Max));
frac_Temp_Max = int16(mod(Temp_Max,1)*10);

mag_Pressure=int16(floor(Pressure));
frac_Pressure = int16(mod(Pressure,1)*10);
mag_Pressure_Min=int16(floor(Pressure_Min));
frac_Pressure_Min = int16(mod(Pressure_Min,1)*10);
mag_Pressure_Max=int16(floor(Pressure_Max));
frac_Pressure_Max = int16(mod(Pressure_Max,1)*10);

Format_String='"%2d.%1d,%2d.%1d,%2d.%1d. %2d.%1d,%2d.%1d,%2d.%1d. "';
coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
   mag_Temp_Min, frac_Temp_Min, mag_Temp, frac_Temp, mag_Temp_Max, frac_Temp_Max,...
   mag_Pressure_Min, frac_Pressure_Min, mag_Pressure, frac_Pressure, mag_Pressure_Max, frac_Pressure_Max);

%If the format string was less than 32 characters, replace the undefined characters by a space.
data_array(data_array==0)=space;
Output_String=data_array(1:lenMax)';

end
```

Demo VII.2: Demo the operation of the BMP085 with the Min_Max_Memory block and the LCD displaying the following

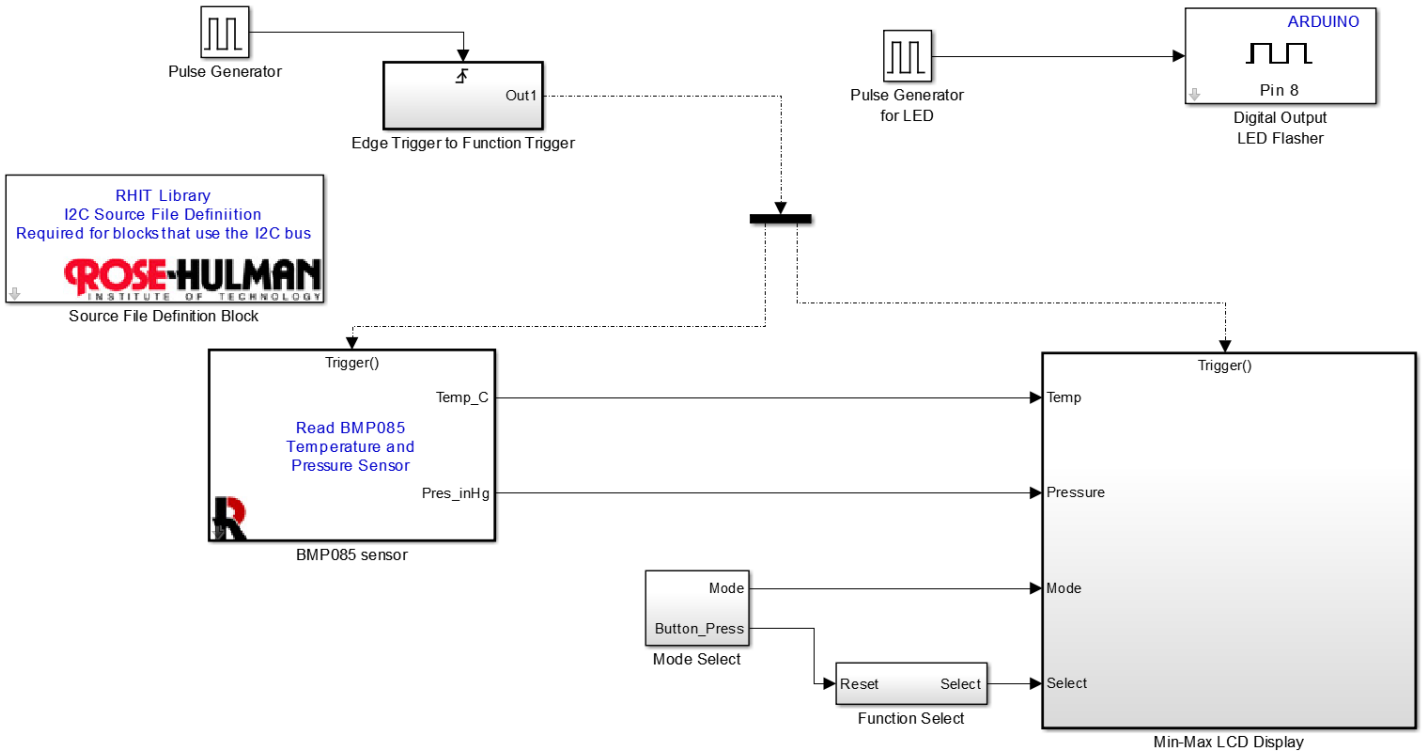
```
Min_Temp,      Actual_Temp,      Max_Temp
Min_Pressure, Actual_Pressure, Max_Pressure
```

Using a %4.1f formagtting string for each numerical value.

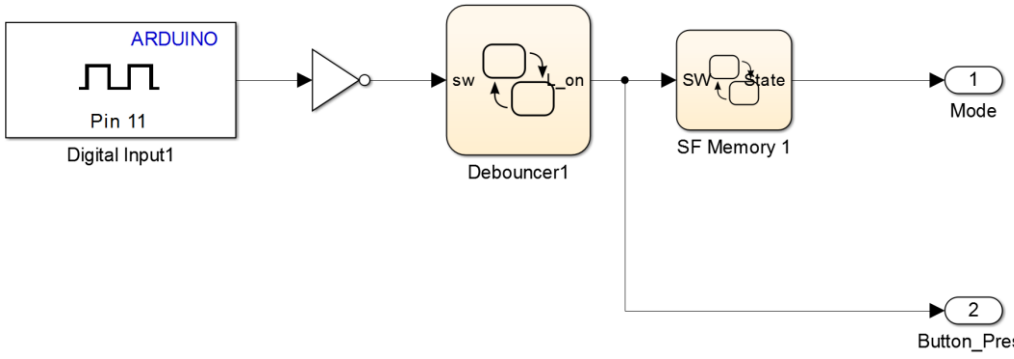
Exercise VII.2: Modify the Min_Max_Memory block so that it has a reset input. When the reset occurs, the minimum and maximum values are reset to the current value. For testing, use a push-button to reset the values.

C. Stateflow Memory and the Switch Case MATLAB Command

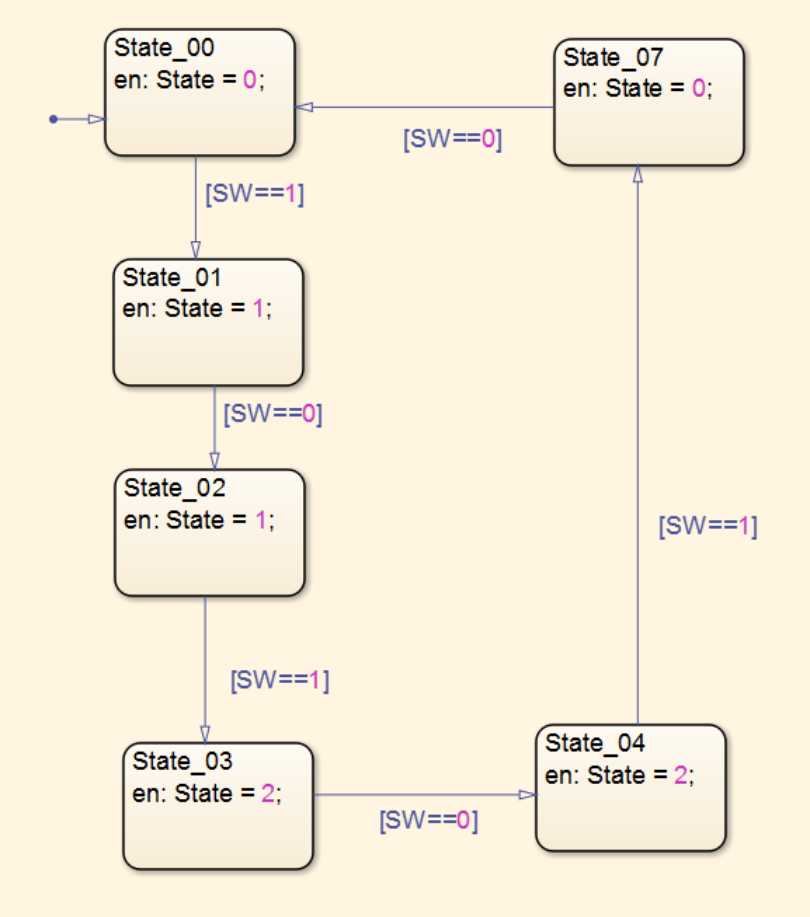
In this section we will look at using a Stateflow chart for memory. We will also look at using some more complicated MATLAB programming structures to display different messages on the LCD display. The top level of the model is shown below:



The **Mode Select** and **Function Select** subsystems use a Stateflow chart to keep track of how many times a pushbutton has been pressed. The **Mode Select** subsystem is shown below:

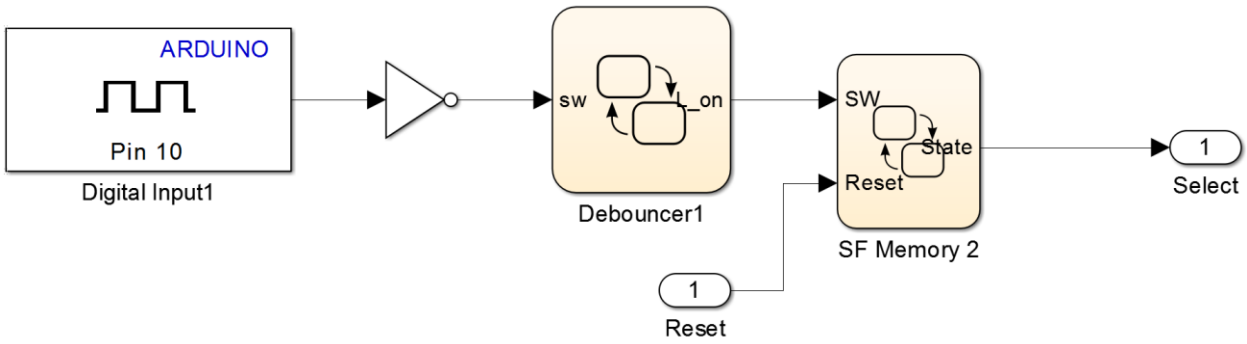


Both of the memory subsystems use a digital input to read the push-button and a debouncer to clean up the signal. Note that this subsystem has a second output called **Button_Press**. Whenever the pushbutton is pressed, this output will change to a 1. This output will be used to reset the second chart we will discuss later. The debouncer was covered in Section V.C.2.a). Here, we are interested in the looking at the chart called **Stateflow Memory 1**:

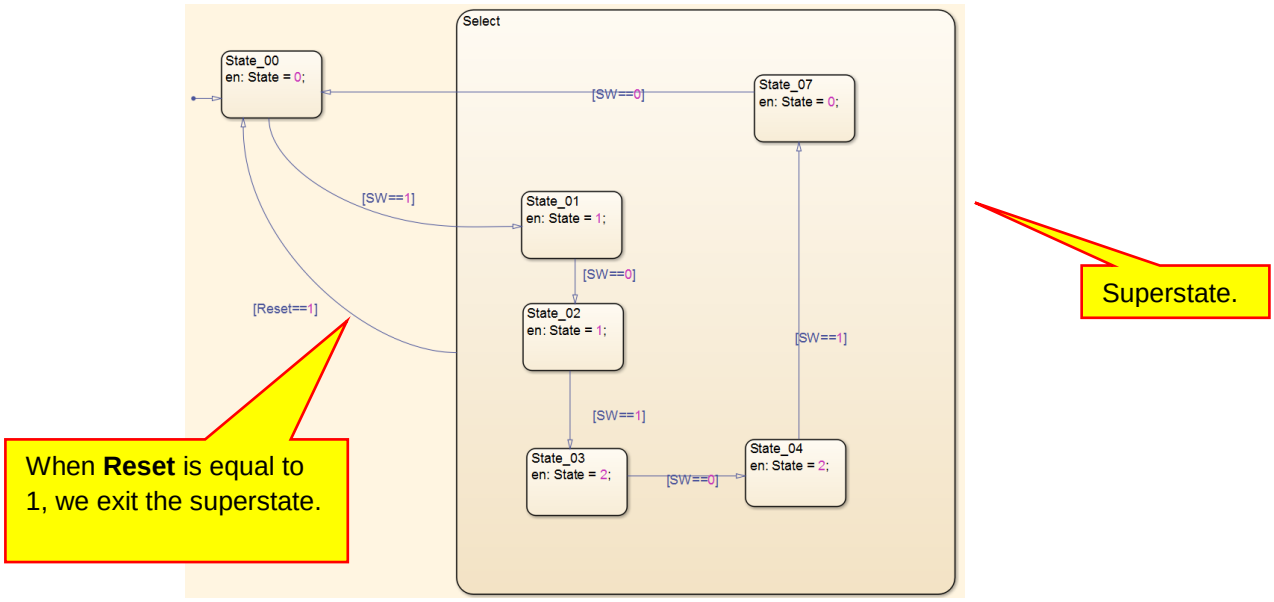


This chart keeps track of the pushbutton. As the pushbutton is pressed and released, the value of the output cycles through one of three values. Note that the value of the output changes when the pushbutton is pressed. When the pushbutton is released, the output stays at its current value. This chart is a simple counter that remembers how many times a pushbutton was pressed.

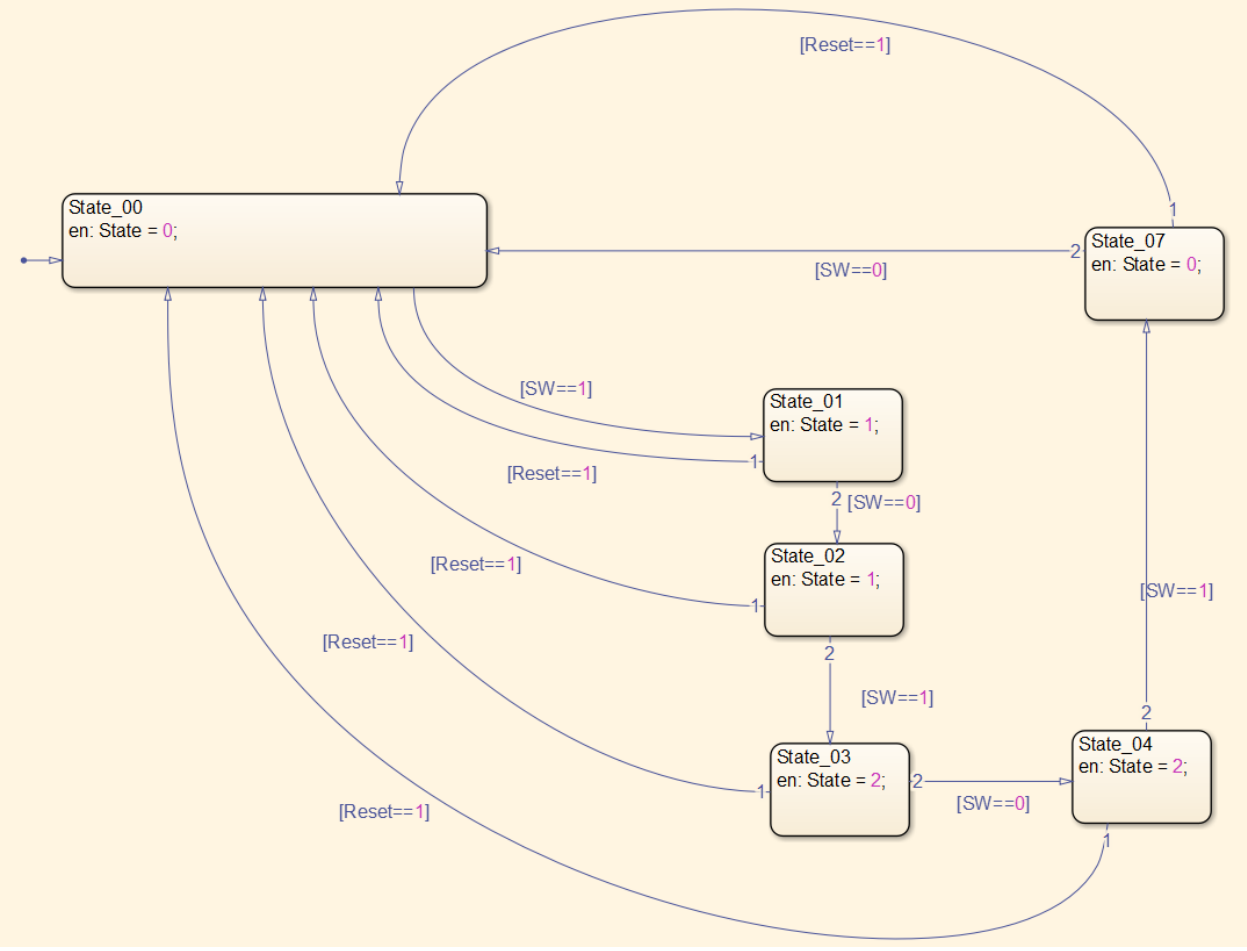
The **Function Select** subsystem is shown below:



The subsystem looks similar to the **Mode Select** subsystem except for the **Reset** input. The **SF Memory 2** chart is shown below:



If we did not have the superstate shown above, this chart would be equivalent to the previous chart. The reason for the super state and the **Reset** signal is that no matter what state we are in, when we receive the **Reset** signal, we return to the initial state, State_00. Logically, the chart above is equivalent to the one shown below:

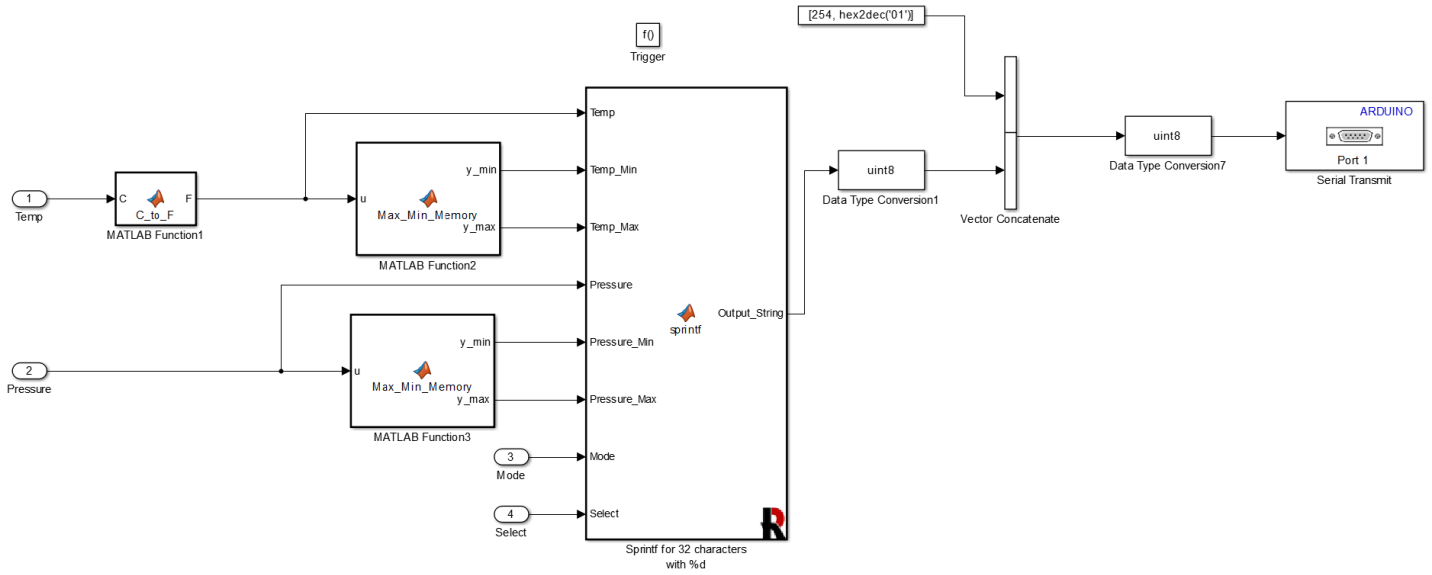


Both implementations work the same. However, the implementation with the superstate is a lot cleaner, easier to understand, and will be easier to modify if we update it or change its function.

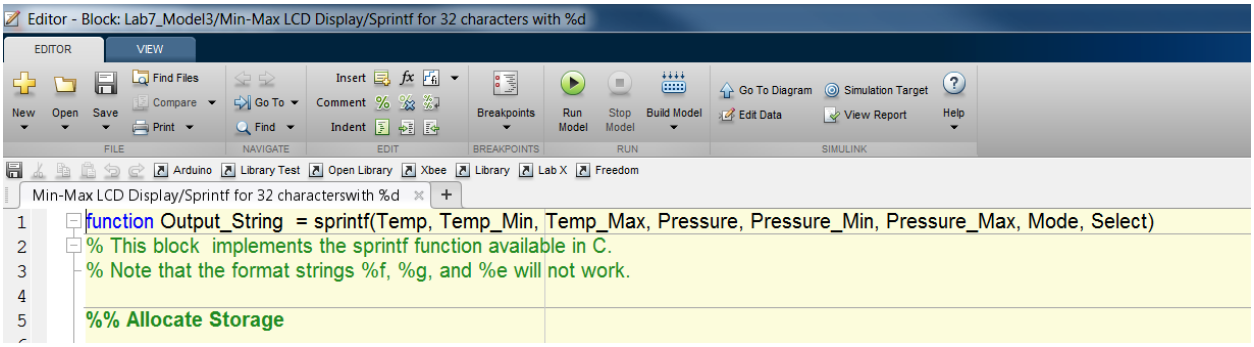
The reason for the two pushbutton subsystems is the following functionality. The Mode select pushbutton selects between displaying the pressure and temperature, the temperature, and the pressure. The Function select switches between displaying the current value, the maximum value, or the minimum value. Whenever you change the mode, the function select is reset to display the current value of whatever it is displaying. Or, as a list of functionality:

- Mode Select 0 – Display the pressure and the temperature on the LCD.
  - Function switch has no effect.
- Mode Select 1 – Display the temperature on the LCD screen.
  - Function Select 0 – Display the current temperature, maximum temperature, and minimum temperature.
  - Function Select 1 – Display the maximum temperature.
  - Function Select 2 – Display the minimum temperature.
- Mode Select 2 – Display the pressure on the LCD screen.
  - Function Select 0 – Display the current pressure, maximum pressure, and minimum pressure.
  - Function Select 1 – Display the maximum pressure.
  - Function Select 2 – Display the minimum pressure.

We have created the Stateflow charts to keep track of the Mode and Function selections. Now we need to modify the **sprintf** function to display different information. The Min-Max LCD display is shown below:



The sprint function has been significantly modified. The first few lines look similar to what we had before except we added **Mode** and **Select** as function inputs:



We have added a switch-case statement to select between several different **sprintf** lines to emit the correct text string:

```
43
44      case 1 %Display Temp
45 -       switch Select
46 -          case 0  % Display All Temps
47 -             Format_String='  Temp: %3d.%1d F H: %3d.%1d L:%3d.%1df ';
48 -             coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
49                  mag_Temp, frac_Temp,  mag_Temp_Max, frac_Temp_Max, mag_Temp_Min, frac_Temp_Min);
50 -          case 1  % Display Max
51 -             Format_String='Max Temperature      %3d.%1dF      ';
52 -             coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
53                  mag_Temp_Max, frac_Temp_Max);
54 -          case 2  % Display Min
55 -             Format_String='Min Temperature      %3d.%1dF     ';
56 -             coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
57                  mag_Temp_Min, frac_Temp_Min);
58 -       end
59      case 2 %Display Pressure
60 -       switch Select
61 -          case 0  % Display All Pressures
62 -             Format_String='Barr: %3d.%02d in.H:%2d.%02d L:%2d.%02d ';
63 -             coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
64                  mag_Pressure, frac_Pressure,  mag_Pressure_Max, frac_Pressure_Max, mag_Pressure_Min, frac_Pressure_Min);
65 -          case 1  % Display Max
66 -             Format_String='  Max Pressure     %3d.%02d inHg   ';
67 -             coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
68                  mag_Pressure_Max, frac_Pressure_Max);
69 -          case 2  % Display Min
70 -             Format_String='  Min Pressure     %3d.%02d inHg   ';
71 -             coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
72                  mag_Pressure_Min, frac_Pressure_Min);
73 -       end
74
75  end
```

Instead of having a single **sprintf** statement, we can use the rich language available in MATLAB to choose between one of several **sprintf** statements. We could have implemented this logic in Simulink with switches, but it is far easier and cleaner to do it in a MATLAB function block, especially since we are required to use the **sprintf** function in a MATLAB Function block. Note also that we could have used nested **if** statements to implement the logic shown above. However, we have illustrated the use of **if** statements in previous **MATLAB Functions**.

Demo VII.3: Demo the BMP085 with the Mode Select and Function Select pushbuttons.

D. All Sensors Project

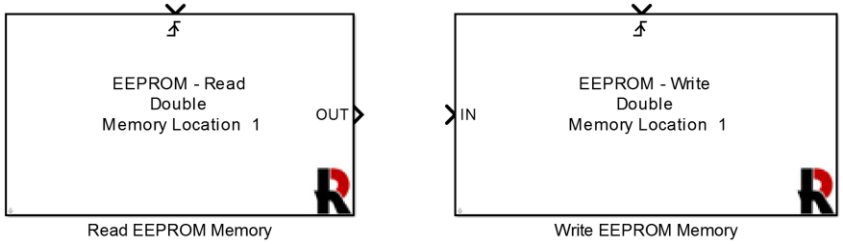
Exercise VII.3: Modify the model created in Demo VII.3 to implement the functionality below:

- Mode Select 0 – Display the pressure and the temperature on the LCD display.
  - Function switch has no effect.
- Mode Select 1 – Display the temperature on the LCD screen from the BMP085 sensor.
  - Function Select 0 – Display the current temperature.
  - Function Select 1 – Display the maximum temperature.
  - Function Select 2 – Display the minimum temperature.
- Mode Select 2 – Display the temperature on the LCD screen from the TC074 sensor.
  - Function Select 0 – Display the current temperature.
  - Function Select 1 – Display the maximum temperature.
  - Function Select 2 – Display the minimum temperature.
- Mode Select 3 – Display the temperature on the LCD screen from the TC074 sensor, the BMP085 sensor, and the average of the two sensors.
  - Function switch has no effect.
- Mode Select 4 – Display the pressure on the LCD screen.
  - Function Select 0 – Display the current pressure.
  - Function Select 1 – Display the maximum pressure.
  - Function Select 2 – Display the minimum pressure.
- Mode Select 5 – Display the time and date as read from the RTC module.
  - Function switch has no effect.

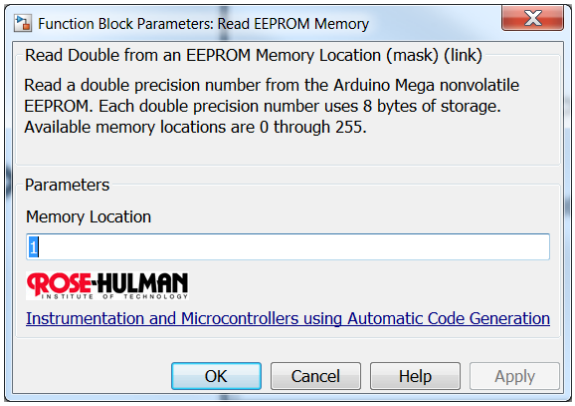
E. EEPROM Non-Volatile Memory

The memory structures we have looked at in the previous sections use volatile memory, meaning that when we cycle the power, all of the information saved in the structures is lost. The Arduino Mega has 4 kB of EEPROM which is non-volatile memory that we can read from and write to. The EEPROM will retain the information written to it after we cycle the power and even if we write a new model to the controller. Thus, important information that needs to be saved, even if the user cycles the power, can be written to the EEPROM.

Two EEPROM blocks are available in the RHIT Arduino Library:



These are masked subsystems that we created for using the EEPROM. A masked subsystem allows you to hide the subsystem below it from unsuspecting users as well as pass parameters to the subsystem below. Most users will not need to see the underlying subsystem, so the mask is used to intentionally prevent the user from seeing unnecessary detail. However, you can look at the mask and look behind the mask if you so desire. First, we will double-click on the **EEPROM – Read block**. Since the subsystem is masked, a window will open that makes the subsystem look like most of the other Simulink blocks we have seen before. (If the subsystem was not masked, double-clicking on the block would open the subsystem.) Double-click on the **EEPROM – Read** block:

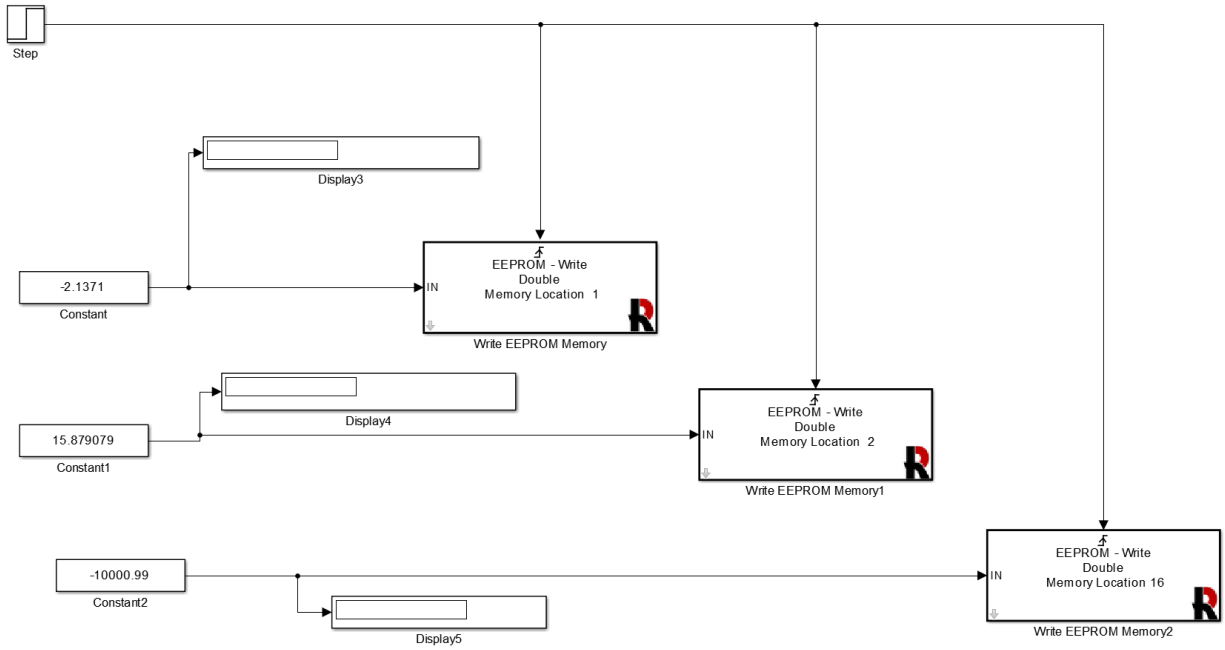


The mask tells us how to use the block and this is all that most people will need to know. The description tells us that this block will save one double-precision value to the EEPROM. All we need to specify is the memory location of where we want to save the value. Note that instead of specifying a variable name, we specify a memory location. If we want to retrieve the value, we will need to read the same memory location.

Note that we can only store 256 double precision numbers with this subsystem even though the EEPROM is capable of storing 512. This is because we used an 8-bit unsigned integer to specify the address (UINT8) which can only have values between 0 and 255. Click the **Cancel** button as we will not make any changes to the memory location address..

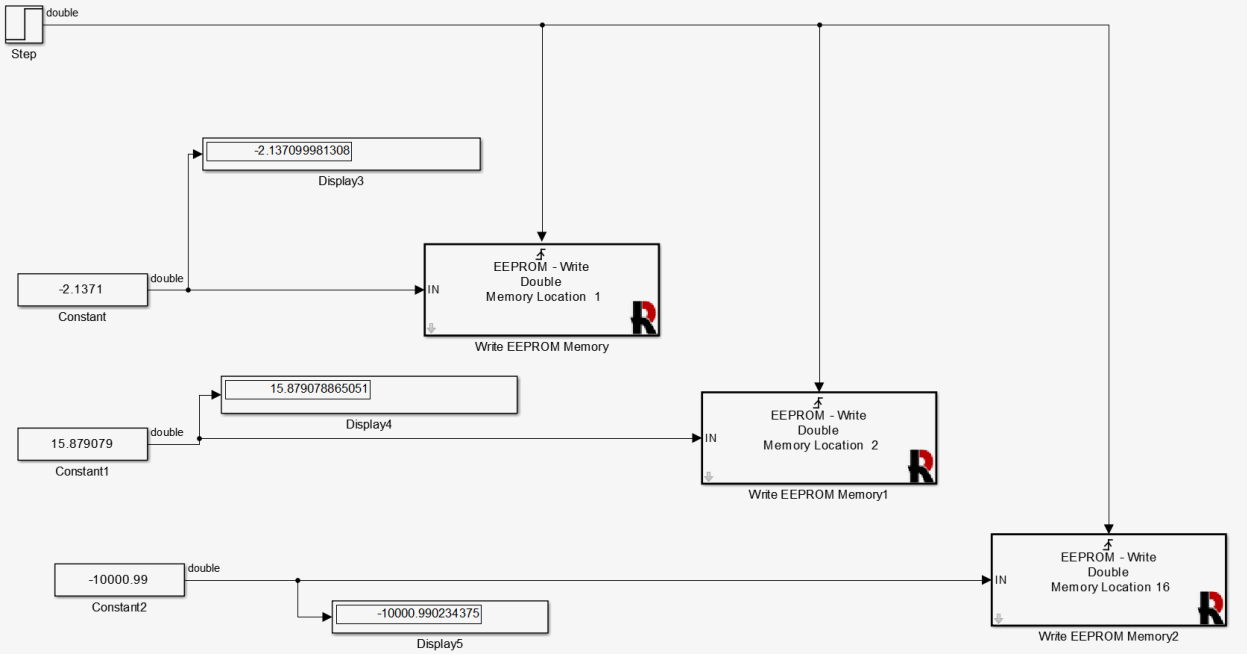
Writing to EEPROM is slower than writing to volatile RAM. Typically, we will read the EEPROM at power up, and only write to EEPROM when we want to record important values. Triggered subsystems allow us to read and write to the EEPROM at specified times. If you need a different trigger type, you can modify the subsystems.

For our example of writing to the EEPROM, we will use the Subsystem shown below:



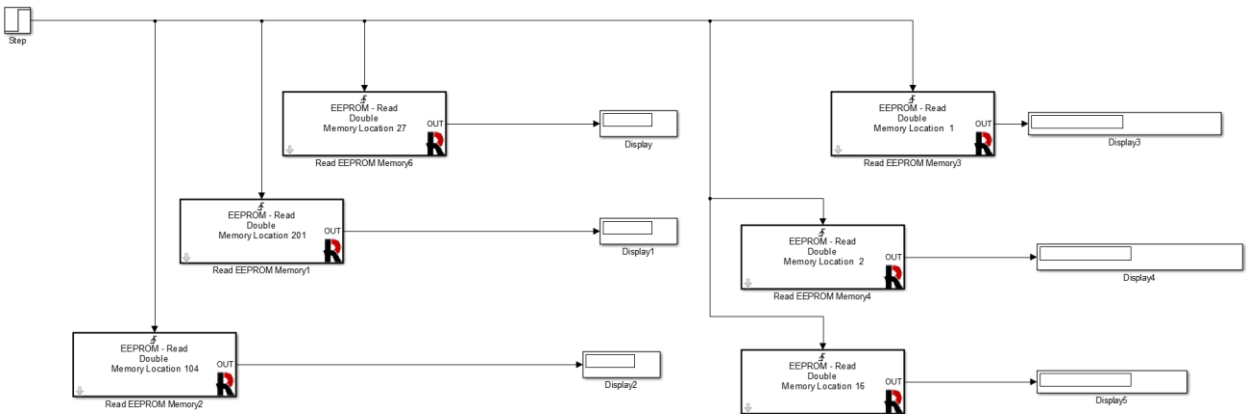
Note that we are using external mode and **Display** blocks so that we can see how the values specified by the constants are displayed in the **Display** blocks. We will read and display the EEPROM values in a second

model using external mode and display blocks as well. By using external mode in the model above, we eliminate the problem of the **Display** blocks displaying the value of the constants in a different format than shown in the constant blocks. When we run the model and connect to it in external mode, we do see that the display blocks show the value of the constants slightly differently than in the constant blocks:

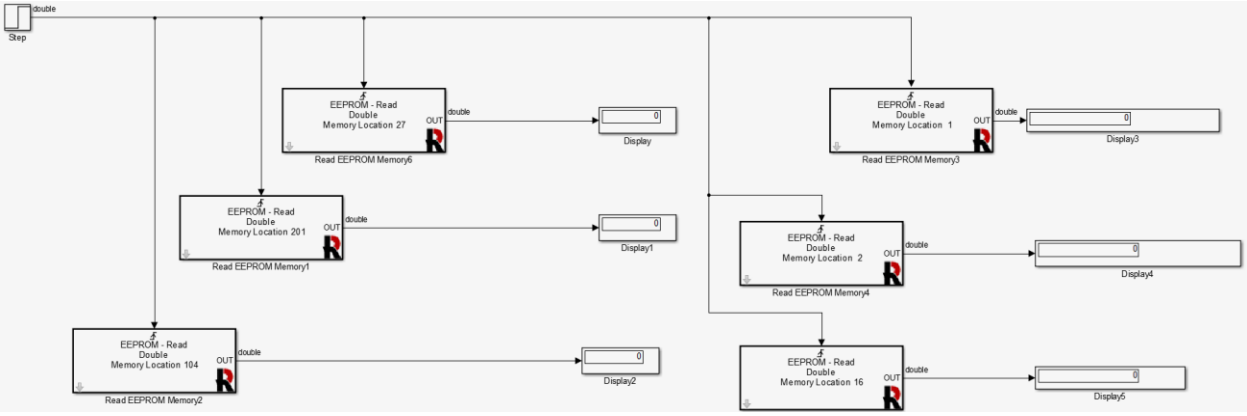


The step function is defined to generate a 0 to 5 step at after 2 seconds. Thus, the values are written to the EEPROM once, two seconds after the model begins execution. Run this model on your Arduino. Note that we are using three memory locations, 1, 2, and 16.

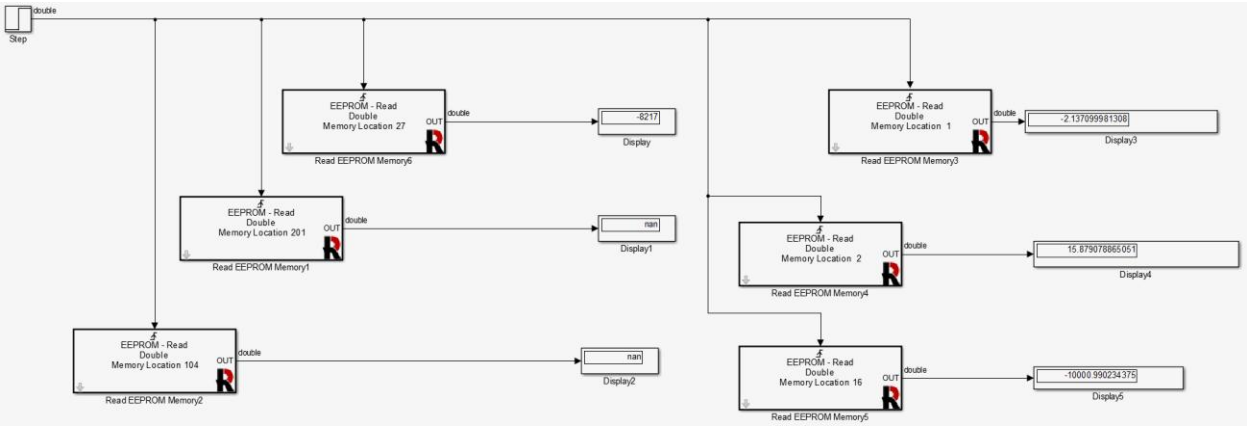
Next, create the model below to read the EEPROM:



A few notes are needed about this model. First, we are using external mode so that we can observe the operation of the read function. Second, the step occurs at 10 seconds so that we can see the values before and after the read. Third, we are reading memory locations that we did not write to using the EEPROM write function. Build the model, download it to your Arduino, and quickly connect to your board using external mode. Before the step occurs, the EEPROM Read block outputs zeros:



After the step occurs, the blocks display the results below:



We notice two things. First, for the memory locations that we wrote to (1, 2, 16), the value displayed matches exactly with the display shown in the EEPROM Write model. Thus, our write and read blocks work properly. Second, when we read a memory location that we have not initialized with the EEPROM Write block, it has the value of NaN, which means Not a Number. Finally, note that memory location 27 had a value of -8217. This is probably just luck that it contains a real number, or it was a value that was placed there by an EEPROM write command previously. The value of -8217 has nothing to do with this model. However, since I did not initialize the memory location, it had an invalid value.

As a last test, disconnect from the target to end external mode and then cycle the power on your Arduino Mega board. Then reconnect your board and connect to your board. You should have the same display as shown in the previous screen capture, showing that the Arduino Mega retains the values in the EEPROM even after cycling the power.

Demo VII.4: Show that the Arduino Mega retains the values in the EEPROM after cycling the power. (Display the results of the model shown above.)

Exercise VII.4: Create a counter that starts at zero and increments every second. The counter should remember its value when the power is removed or the controller is reset. When controller restarts, the counter should increment once a second, starting from the value the count had when the controller was reset or the power was removed. Include a pushbutton that resets the count to zero if the pushbutton is pressed. Display the value of the count on the LCD screen.

Lab VIII
Collecting Data 1

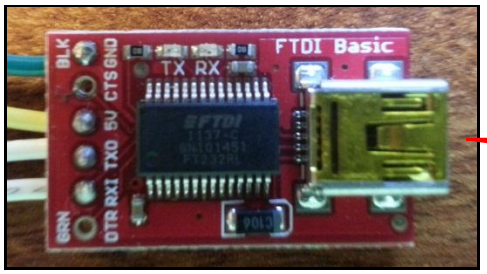
In this lab and Lab IX we will look at collecting data with a sensor and storing that data. In this lab we will record data on the Arduino controller using the volatile RAM. The data will be sent over a serial data line to a Windows computer. We will use MATLAB to read the data from the serial port and store in in the data in a MATLAB matrix. Once read, the data will be saved in the MATLAB workspace where it can be saved to the hard disk, plotted, or manipulated with MATLAB to suit our needs.

This method has two major limitations. First, the data is stored in RAM on the Arduino. If the board loses power or the board is reset before we transmit the data to the Windows computer, all of the data stored in RAM will be lost. Second, since we are using a serial data link, data takes time to transmit and is error prone. Furthermore, we are limited in the amount of data we can collect because of the errors and the amount of time it takes to transmit the data. These limitations are eliminated in Lab IX when use an on-board SD card to record the data. Although this method is not reliable for collecting a large amount of data over a long period of time, the methods here can be used to create a monitor that will allow us to view the status of our controller by sending serial information.

For this lab, we will develop a single large model that collects data from a sensor, saves the data in the controller RAM using a memory structure similar to what we developed in Lab VII, displays the most recent sensor data on the LCD screen, and transmits that over a serial line when a push-button is pressed. We will also develop a MATLAB script to read and record the data.

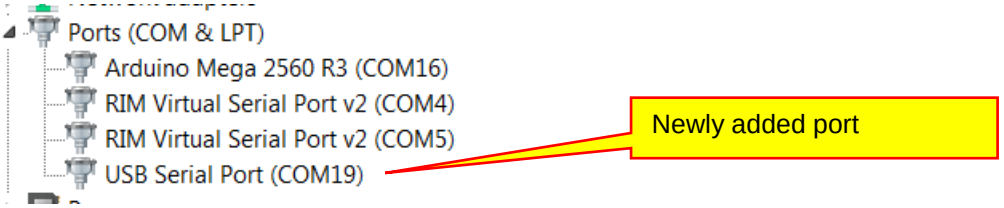
A. Serial Communication with Simulink

We will use the Sparkfun FTDI Basic USB to TTL serial converter shown below:



This board plugs into a USB port and adds a serial communication port to your computer. When you plug this board into a USB port, a driver should be installed and then a new serial communication (COM) port should be created. You will need to use the Windows Device Manager[4] to see that the device was added and determine the name of the newly added port. On my system, the port was labeled as COM19:

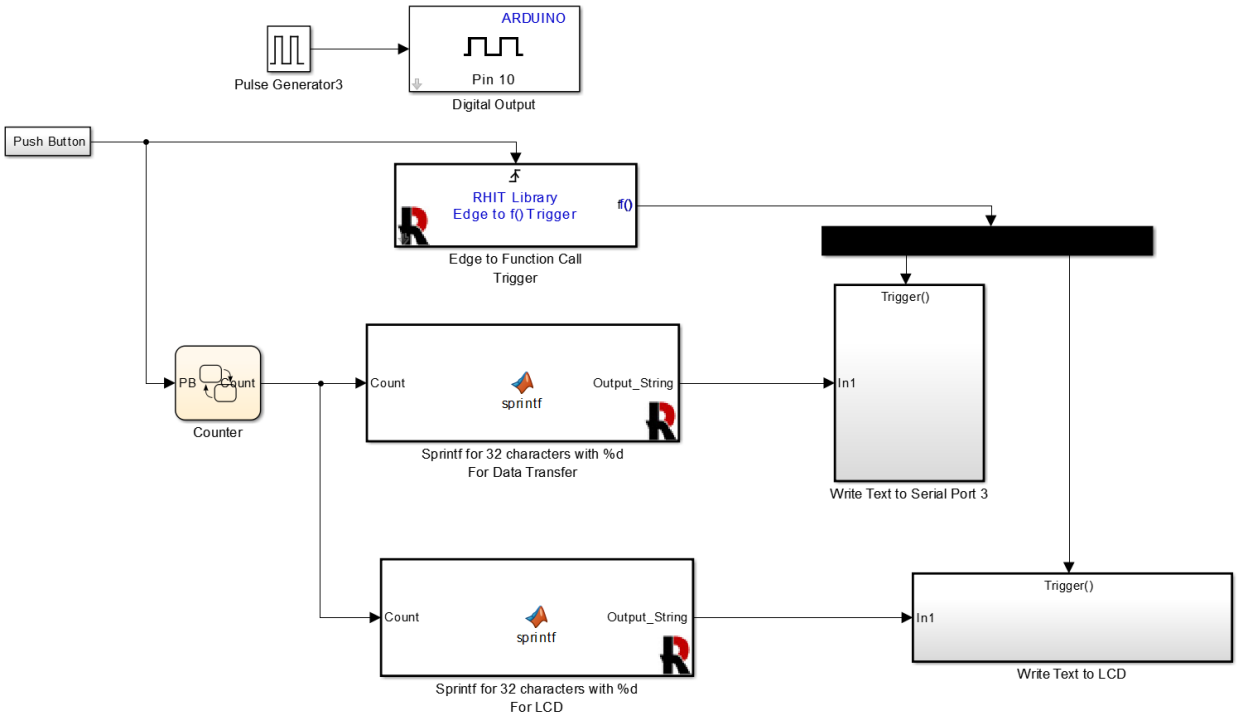
[4] To open the Device Manager in Windows 7, open the **Start** menu, right click on **Computer** and then select **Properties**. The System window will open. In the upper left corner, click on **Device Manager** to open it.



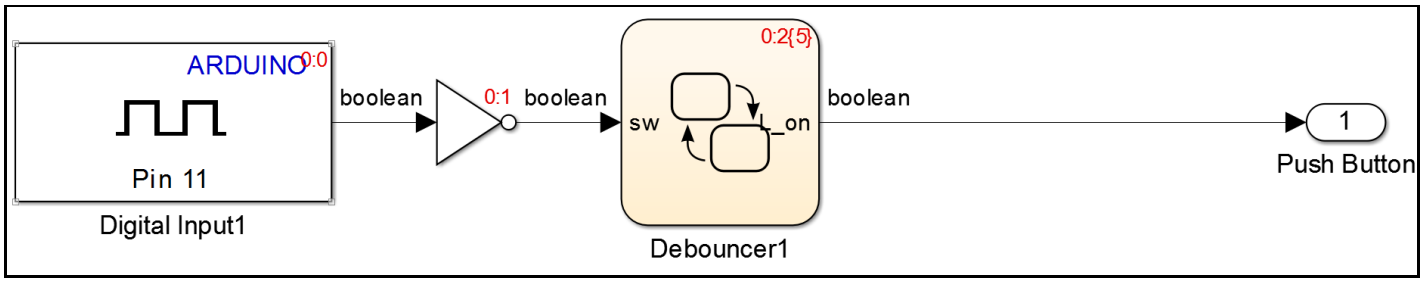
We will need to know the COM port number when we read data with MATLAB.

We will connect the USB serial port to serial port 3 of the Arduino. When connecting two RS-232 serial devices together (it's not really RS-232 because the voltage levels are TTL compatible), the RX line of one device must connect to the TX line of the other device. Connect the serial port to your Arduino board as shown below:

Next, we will create a Simulink model that sends out a single line of text. The top level is shown below and consists of many of the blocks and subsystems we have used previously:

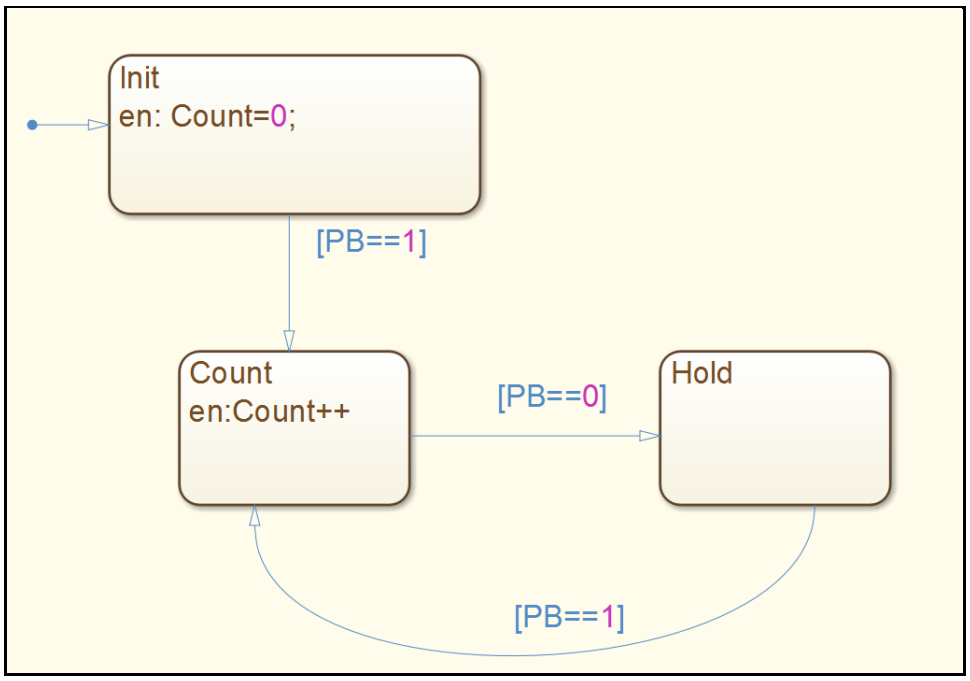


When the pushbutton is pressed, the model will send out a single line of text on serial port 3 (which will be read by MATLAB) and almost the same line of text will be sent out on serial port 1 (to be displayed on the LCD screen). We are using a pushbutton so that we can send out a single line of text at the time of our of choosing. The **Push Button** subsystem is shown below:

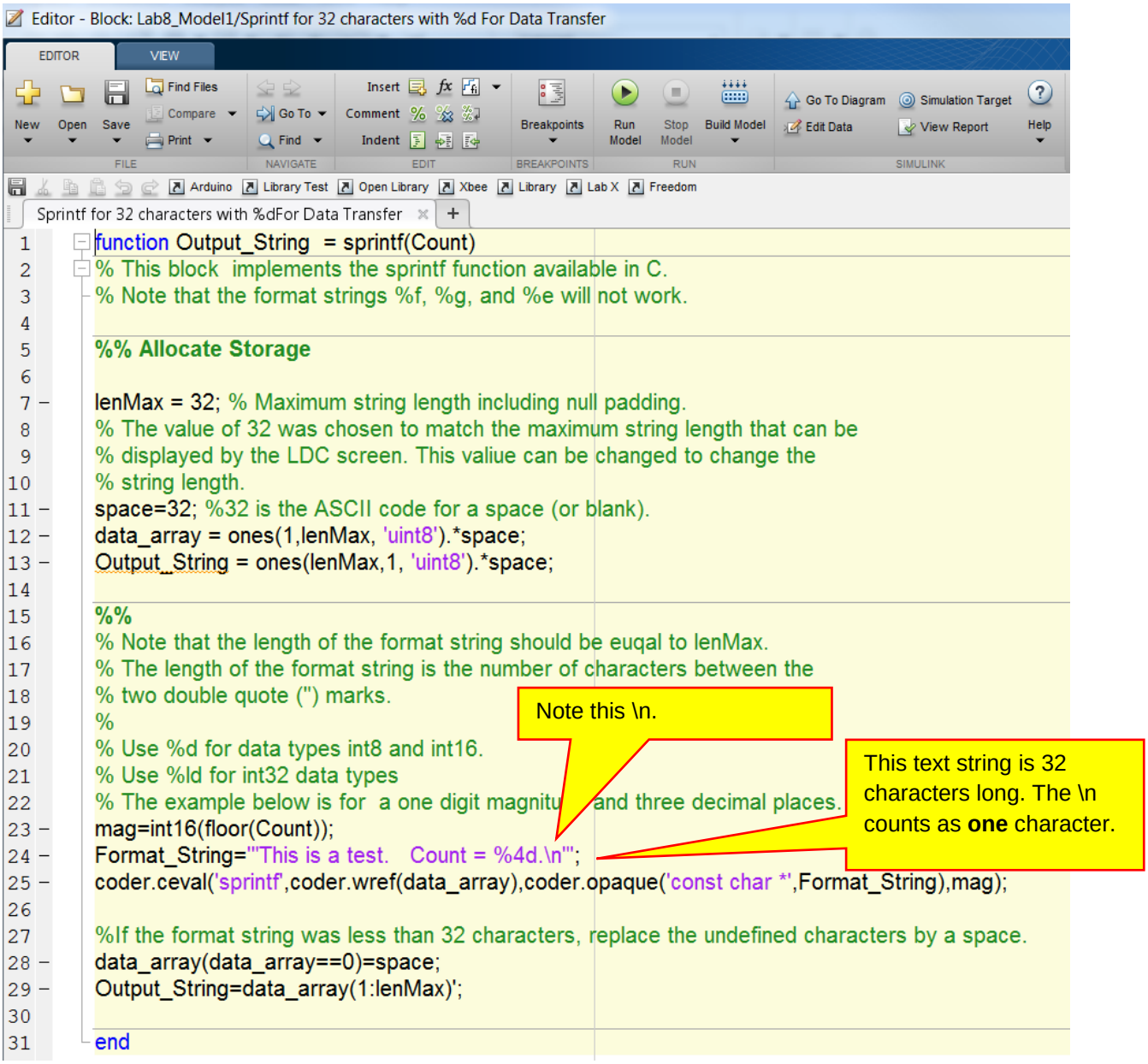


It uses the switch debouncing Stateflow chart developed in Section V.C.2.a). When the pushbutton is pressed, the output will go high. When the pushbutton is released the output will go low. We are using the debouncer because our subsystems are edge triggered, and we want to guarantee that each press only generates a single edge.

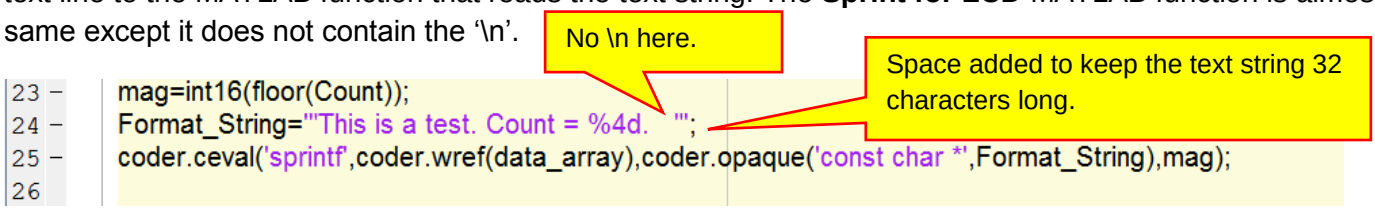
The **Counter** chart is a simple counter that starts at zero and increments every time the pushbutton is pressed. We use this so that we can tell that we are sending out a new and unique text string every time we press the pushbutton:



The two sprintf MATLAB embedded functions are nearly identical. The only difference between the two is that we need to tack on a '\n' character when the data is to be read by MATLAB. The **Sprintf for Data Transfer** MATLAB Function is shown below:



Note that we have added a '\n' to our output string. You may recall that in formatted output with the sprintf or fprintf functions, a '\n' in the text string generates a new line. For this application, the '\n' will indicate the end of the text line to the MATLAB function that reads the text string. The **Sprint for LCD** MATLAB function is almost the same except it does not contain the '\n'.

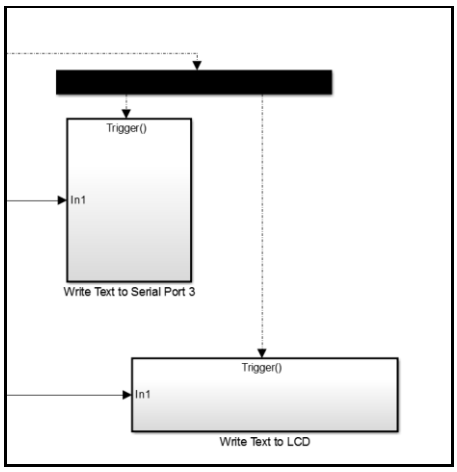


It is important to note that both embedded functions output a text string that is 32 characters long. Make sure that you pad your text with spaces so that each is 32 characters in length.

The **Edge to Function Call** subsystem is an edge triggered subsystem that outputs a function call trigger:

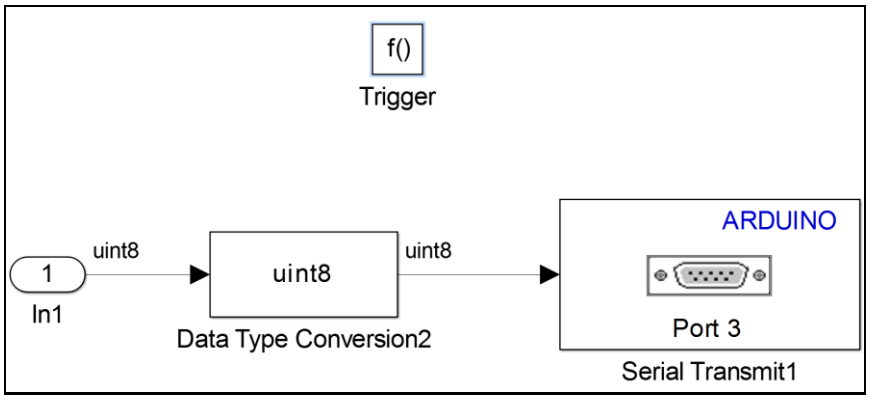
We use this subsystem to convert an edge trigger to a function call trigger. We convert edge triggers to function triggers so frequently that we added a block to the RHIT library to perform this function.

Putting function call triggers into a **Mux** allows us to sequence subsystems in time. When the **Demux** below receives a function call trigger, the **Write Text to Serial Port 3** subsystem executes first. When it is done, then the **Write Text to LCD** subsystem will execute:

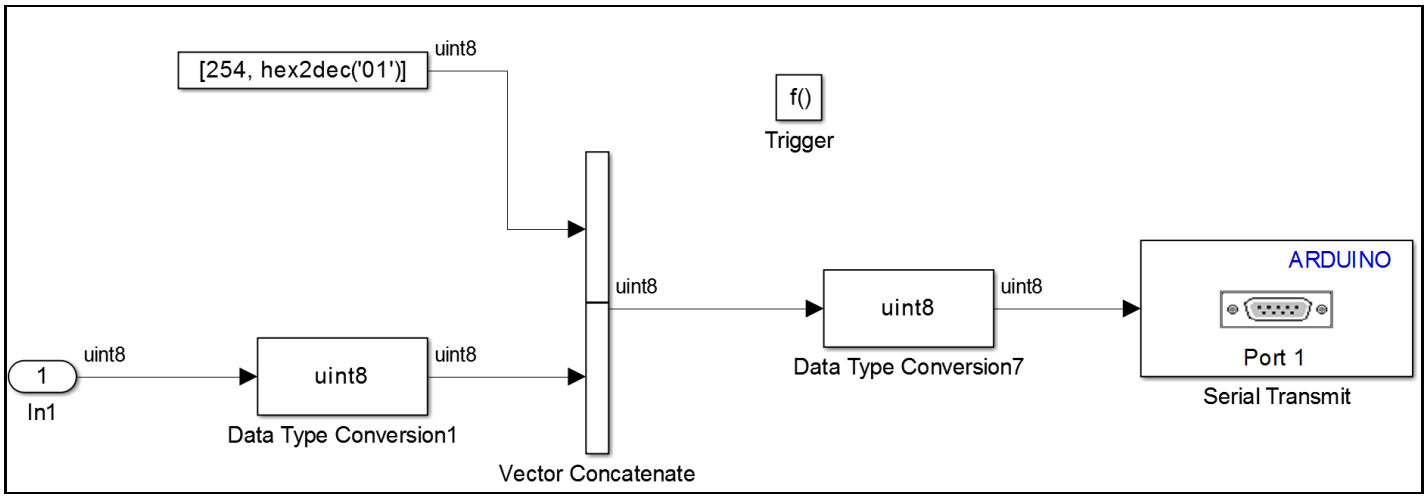


When used with a function call, the Mux generates a function call for the left most subsystem. When it has completed its task, the Mux generates a function call for the next subsystem closest to the left. The systems are executed from left to right. Note that this method works for function call triggers only, and the subsystems attached to the Mux must use a function call trigger (not an edge trigger).

Subsystem **Write text to Serial Port 3** just converts the text string to a Uint8 data type and then outputs it on serial port 3:

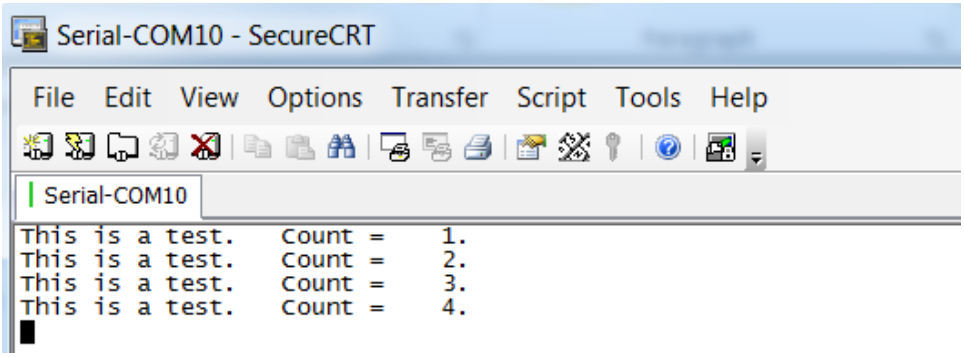


Note that there are no control characters prepended to the text string as we did with the LCD. The **Write Text to LCD** subsystem is shown below:



This is the same function we have used previously to drive the LCD. Note that the control characters [254, hex2dec('01')] that we prepend to the text string instruct the LCD to clear the screen and place the cursor at position 1 of the LCD. We do not want these control characters prepended to the text string that we send to serial port 3 as they contain no data.

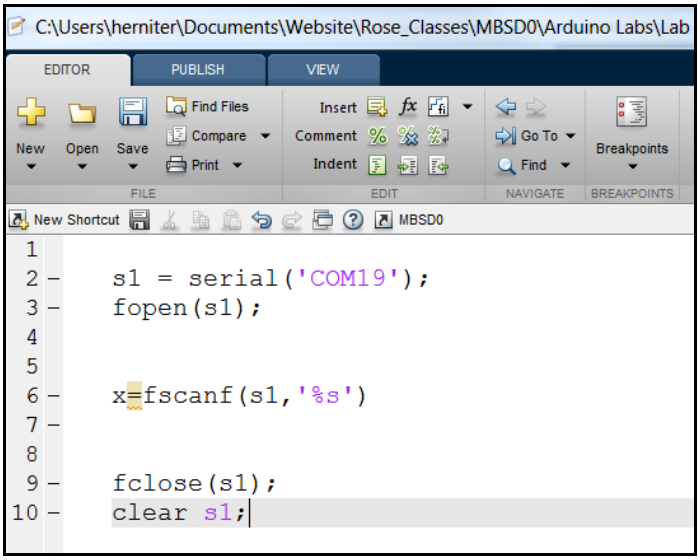
Demo VIII.1: Now that we have a model that outputs a text string to a serial port, we can test that model using a serial communication program in Windows. Connect the Sparkfun FTDI Basic USB port to a USB port on your computer. Run the Windows communication program (if you have one installed on your computer) to verify that the model is sending out text and that your hardware is correct. The LCD display and serial communication program should display a new line of text every time you press the push button. You should see something similar to that shown below:



When done, exit your serial communication program because it will conflict with the next section. (Both will attempt to use the same COM port.)

B. Serial Communication with MATLAB

Now that we can output a text string to the serial port of the Arduino, and that serial port is connected to the USB port of our Windows computer, we need to read the data on the port with MATLAB. We will do this with the code segment below:



When we looked at the Windows Device Manager, we noticed that the USB to serial port device was labeled as COM 19. The line **`s1 = serial('COM19');`** creates a serial port object (s1) associated it with COM19. The line **`fopen(s1);`** opens the port for reading and writing. The line **`x=fscanf(s1,'%s')`** reads a text string from COM19 and stores it in variable x. The fscanf command will read every character in the COM port buffer until it encounters a \n character, where it will stop. If there are more characters in the buffer after the \n is read, we will need to call the fscanf command one or more times to read the remaining characters. The line **`fclose(s1)`** closes the COM port and the line **`clear s1`** removes variable s1 from the MATLAB workspace. After we issue the commands **`fclose(s1)`** and **`clear s1`**, we can reopen the port using the commands **`s1 = serial('COM19');`** and **`fopen(s1)`**. If we do not close the port and clear s1, we will get an error if we try to reopen the port again.

The code segment:

```
s1 = serial('COM19');
fopen(s1);
x=fscanf(s1,'%s')
fclose(s1);
clear s1;
```

opens the COM port, reads a single text string (up to the first \n character), and closes the comport. This is not how we want to use this facility. We want to open the COM port. Then every time we press the pushbutton to send a text string, we want to issue the command **`x=fscanf(s1,'%s')`** to read the text string. We will keep pressing the pushbutton and using the **`x=fscanf(s1,'%s')`** command to read the test strings until we have convinced ourselves that the system works. When we are done, we can use the commands **`serial('COM19');`** and **`fopen(s1)`** to close the port.

Run the script and then press the pushbutton the value of x should be:

```
x =

Thisisatest.Count=1.
```

Note that spaces have been removed from the text string. This is because the fscanf with the %s format string removes the spaces from the text strings. Thus, the communication works as it should. The second time we press the pushbutton, we get:

```
x =

Thisisatest.Count=2.
```

Note that if you issue the command **`x=fscanf(s1,'%s')`** before data is present, it will wait for data on the port. In our case, it will sit there and wait for you to press the pushbutton. If you wait too long, it may time out and generate an error, as shown below:

```
>> x=fscanf(s1,'%s')
Warning: Unsuccessful read: A timeout occurred before the Terminator
was reached.
x =
''
```

Finally, if we use the %c format string, we will read the spaces in the text string as well:

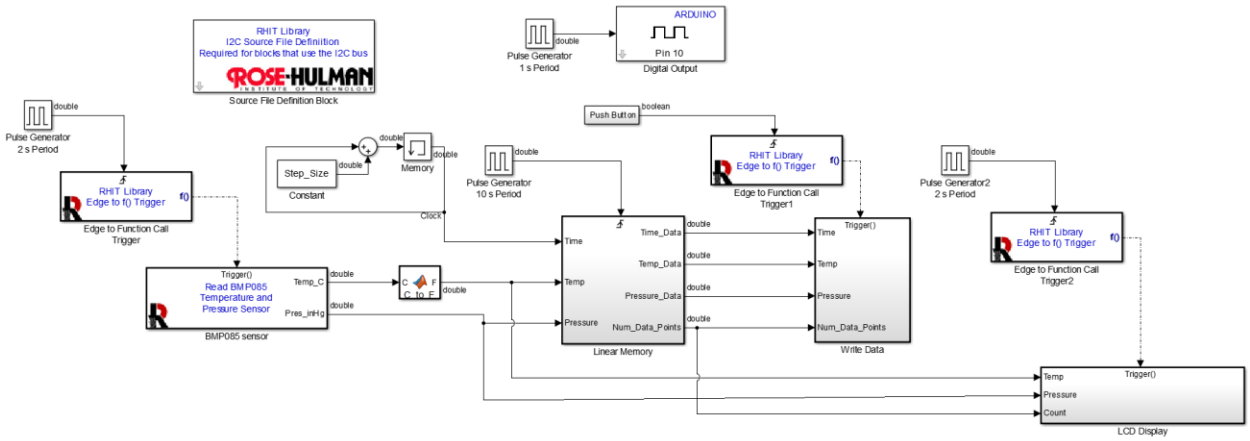
```
>> x=fscanf(s1,'%c')
x =
This is a test.   Count =    8.
>>
```

Note that when we use the %c format string, x is a 1x32 array of type char. When we use the %s format string, x is a text string. (If you look in the workspace, you will notice that they are listed differently.) We will be using text strings (%s format) in the next example because spaces in the data are not necessary and will be ignored.

Demo VIII.2: Demonstrate the operation of sending serial text with the controller and receiving the text with MATLAB.

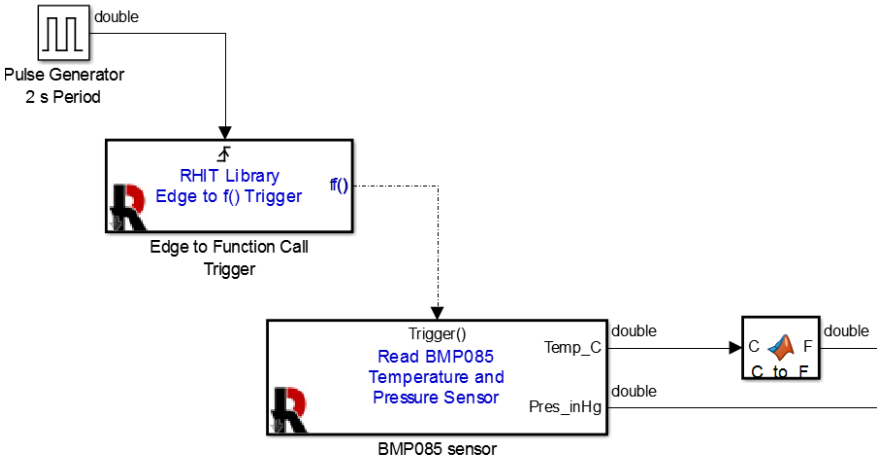
C. Data Collection Simulink Model

We will now create a model for measuring data from a sensor, storing that data in a memory structure, and then transmitting that data over a serial port. The complete model is shown below:

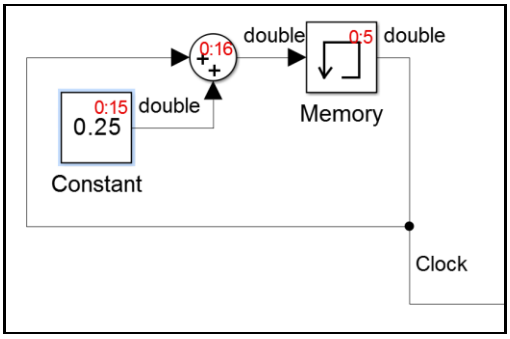


Many portions of this model have been used previously, and you should make use of models that you have developed previously. First, we need to measure some data. For this example, we have chosen the BMP085 to

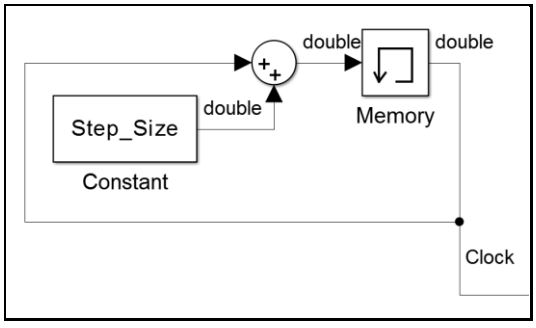
measure pressure and temperature. The block was copied directly from Section VI.B.1. We will sample the sensor every two seconds:

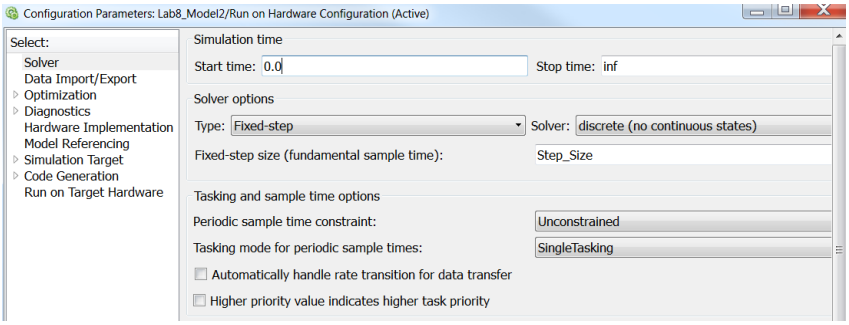


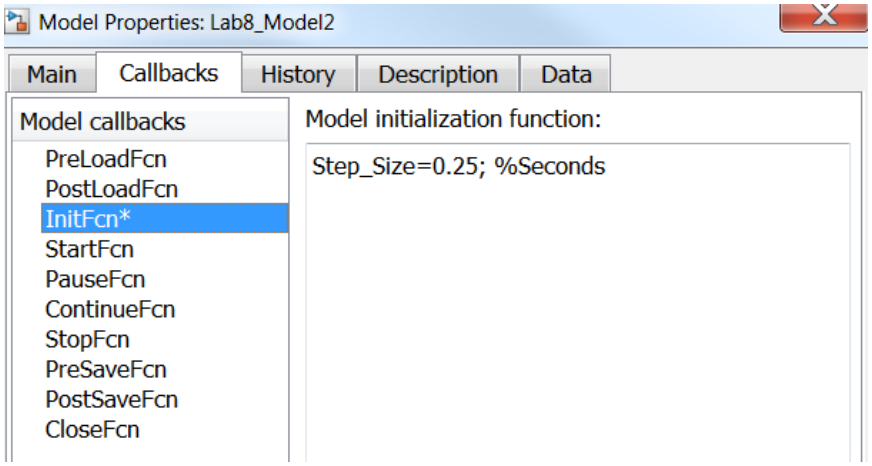
This sensor gives us temperature and pressure. We would also like to record the time at which the samples were taken. The blocks below show a simple way to generate a time signal:



The memory block is initialized to zero, and then every fixed time step, 0.25 seconds is added to the sum. The constant value of 0.25 is chosen to match the fixed step size of the model, which is specified in the Configuration Parameters dialog box. The output of the memory block will be equal to the real-time as long as we do not have an overrun and the constant value matches the **Fixed-step size**. It would be good practice to declare a workspace variable for the step size and then use the variable to specify both values, as shown below:



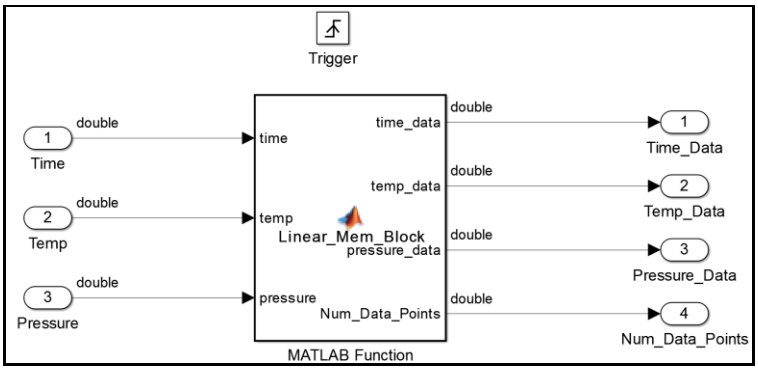




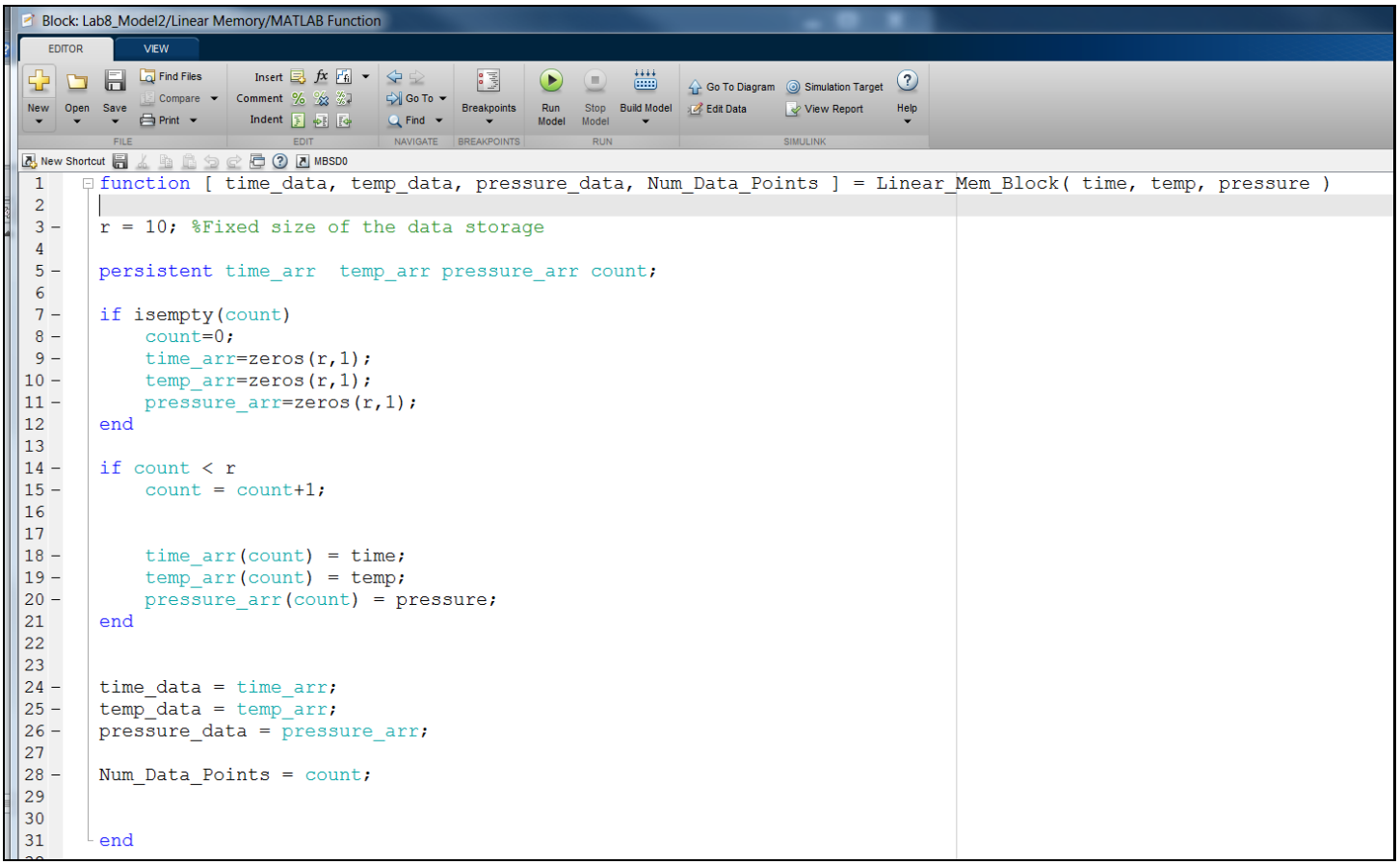
To monitor that the system is working, we continue to use the flashing LED and we also display the most recent sensor values on the LCD display. The display updates every 2 seconds. The LCD will also display the number of samples collected, which we will discuss later. The **LCD Display** subsystem is the same as we have used numerous times with the slightly modified format string shown below:

```
23 -    mag_Count=int16(floor(Count));
24 -    mag_Temp=int16(floor(Temp));
25 -    frac_Temp = int16(mod(Temp,1)*10);
26 -    mag_Pressure=int16(floor(Pressure));
27 -    frac_Pressure = int16(mod(Pressure,1)*10);
28 -    Format_String='"Temp:%2d.%01dF,I=%3dBarr:%3d.%02d in. "';
29 -    coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
30          mag_Temp, frac_Temp, mag_Count, mag_Pressure, frac_Pressure);
```

We will store the data in a linear memory similar to what we did in Section VII.A . **Note that we only store a value in the memory every 10 seconds even though we sample the sensors every 2 seconds.** The block is a MATLAB Embedded function:



The memory uses the MATLAB function below:



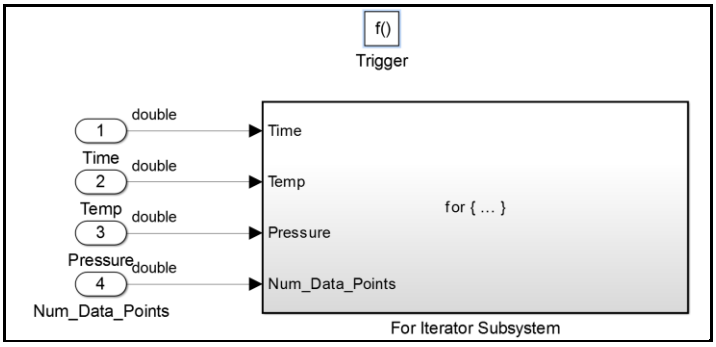
The memory stores ten rows of data (r=10). The value of ten was chosen to limit the amount of data we will transmit as a block. If the data block is too large, an overrun will occur. The data is stored in column arrays, in this case arrays of dimension 10 rows by 1 column. Note that the outputs of the function are the complete data arrays, not a single value.

Finally, we are also outputting the index into the array. If we use arrays larger than 10 elements, we might not use the entire memory structure. The variable Num_Data_Points is the number of rows of data that we have collected. We will use this value to only transmit the data we collected, and not transmit an array full of zeros.

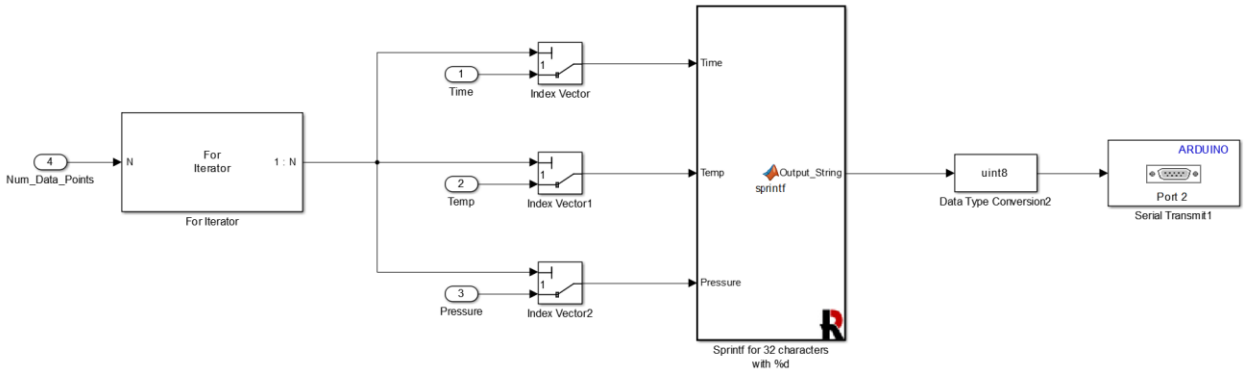
Important Note: you can change the size of the memory array if you would like. However, if the array is too large, when you go to transmit the data, an overrun may occur. To avoid this problem, we will use the SD Card memory we will develop in Lab VIII.E. (And also the model of Exercise VIII.1.)

The new part of this model is the **Write Data** subsystem. When a pushbutton is pressed, this subsystem will transmit the collected data over serial port 3. The pushbutton is the same as in Section VIII.A and will not be covered here.

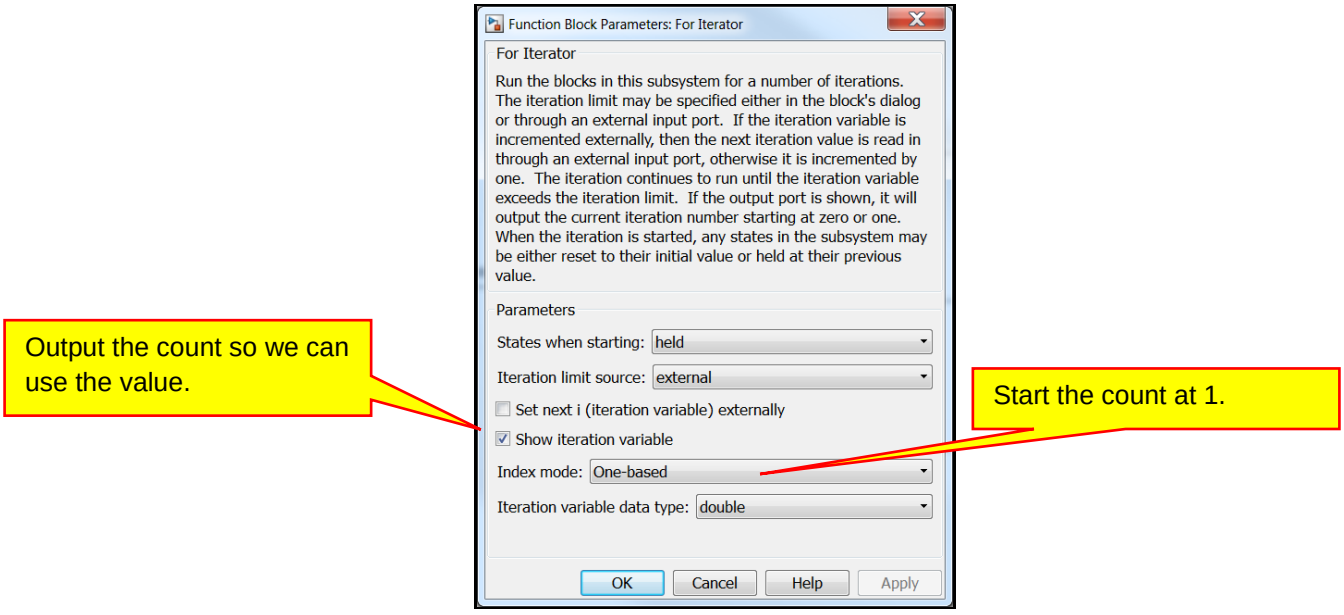
The **Write Data** subsystem is a **For Iterated** loop that repeats for the number of rows of non-zero data saved in the array:



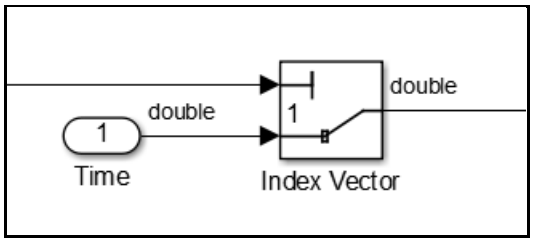
Inside the **For Iterator Subsystem**, we have:



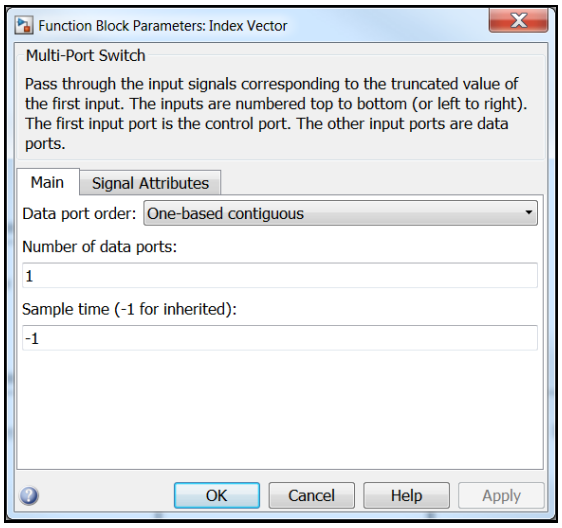
The **For Iterator** starts at 1, and then every block in the subsystem is executed. Then the **For Iterator** increments, and the blocks in the subsystem are executed again. The process repeats until the **For Iterator** hits the maximum count. Thus, the **For Iterator Subsystem** is used to repeat a set of blocks for a specified number of times. In this application, the **For Iterator** counts from one to the number of rows of non-zero data (saved in signal **Num_Data_Points**). The **For Iterator** block outputs the value of the count so that we can use it to index the arrays:



The **Index Vector** block



allows us to address a specific value stored in a vector. In our case each vector holds up to 10 values. The Index Vector allows us to pick out a specific element. In this case, we will pick out the value indicated by the **For Iterator** counter. Since the **For Iterator** counts from 1 to the number of rows of data, this subsystem allows us to step through all of the data and transmit one line at a time. The **Index Vector** block should also specify ones based indexing:



The output of the **Index Vector** blocks is one row of data for the time, temperature, and pressure. We then transmit these on serial port 3 with the following format:

```
23 -    mag_Time=int16(floor(Time));
24 -    frac_Time = int16(mod(Time,1)*10);
25 -    mag_Temp=int16(floor(Temp));
26 -    frac_Temp = int16(mod(Temp,1)*10);
27 -    mag_Pressure=int16(floor(Pressure));
28 -    frac_Pressure = int16(mod(Pressure,1)*100);
29 -    Format_String='"%7d.%01d;%7d.%01d;%7d.%02d;\n"';
30 -    coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
31          mag_Time, frac_Time, mag_Temp, frac_Temp, mag_Pressure, frac_Pressure);
32
```

Note that we output the data using the floating point format and separate the values using semicolons. In this application, the semicolon (;) is called the delimiter and will be used to indicate individual values. Also remember that the line is terminated with a \n. The reader we will use will separate the numerical values based on the delimiter (;) and determine the end of a line by the \n. Note that the format of the data is three real numbers separated by semicolons (;).

Note that this **For Iterator** loop takes a different amount of time to execute based on the number of rows of data. Furthermore, for the model to run in real-time, the model, including the **For Iterator**

Subsystem, must finish within the fixed step time. This is troublesome in this application because the **For Iterator Subsystem** takes a different amount of time to complete based on the size of the array we specify and the amount of data collected. If you increase the size of the array and collect variable amounts of data, sometimes you may get an overrun, sometimes not. This is not the best programming practice, so we will keep the size of the arrays small (r=10 in this case).

D. Data Collection MATLAB Script

The script for reading the data is shown below:

```
s1 = serial('COM19');
fopen(s1);
dat=zeros(1,3);
data=[];
str=input('Send the data, then hit the Enter key.\n', 's');
while s1.BytesAvailable>0
    x=fscanf(s1,'%s');
    a=textscan(x,'%f%f%f', 'delimiter',';');
    for j=1:3
        dat(j) = a{j};
    end
    data=[data;dat];
end
fclose(s1);
clear s1;
```

The lines **`s1 = serial('COM19')`** and **`fopen(s1)`** create a serial port object and open the port. These lines were discussed in detail in Section VIII.B. The line **`dat=zeros(1,3)`** initializes the array named dat and fills it with zeros. This variable will be used to store one row of transmitted data. **`data=[]`** initializes variable data to an empty matrix. This variable will be used to store all of our data and will be constructed by using concatenation. We will grow the array as we add data.

The line **`str=input('Send the data, then hit the Enter key.\n', 's')`** prompts the user with the text line and then waits for user input in the form of a text string. We will not use the response entered by the user. We just use this line to sequence events. The sequence of events is the following. The model will run on the Arduino Mega and collect data. This could be a long or short time depending on how much data we want to collect. When we want to transfer the data to MATLAB, we run the script, which will output the text string and wait for us to hit the enter key. We will then press the pushbutton on the Arduino Mega to start the serial transfer. We will then press the enter key to receive the data.

Serial port object s1 has several fields. **`s1.BytesAvailable`** tell us how much data is available at the port. The **while** loop will repeat until there is no more data available, which means that all of the data has been read.

The next few lines read the data and save it as numeric data in variable **`data`**. The line **`x=fscanf(s1,'%s')`** reads text from the serial port until it encounters a new line character (\n), where it stops. Remember that x will be a text string with the spaces removed. The line **`a=textscan(x,'%f%f%f', 'delimiter',';')`** scans through text string x and parses it into three floating point numbers using a semicolon (;) as a delimiter. Variable **`a`** is a structure with three values. The lines

```
    for j=1:3
        dat(j) = a{j};
    end
```

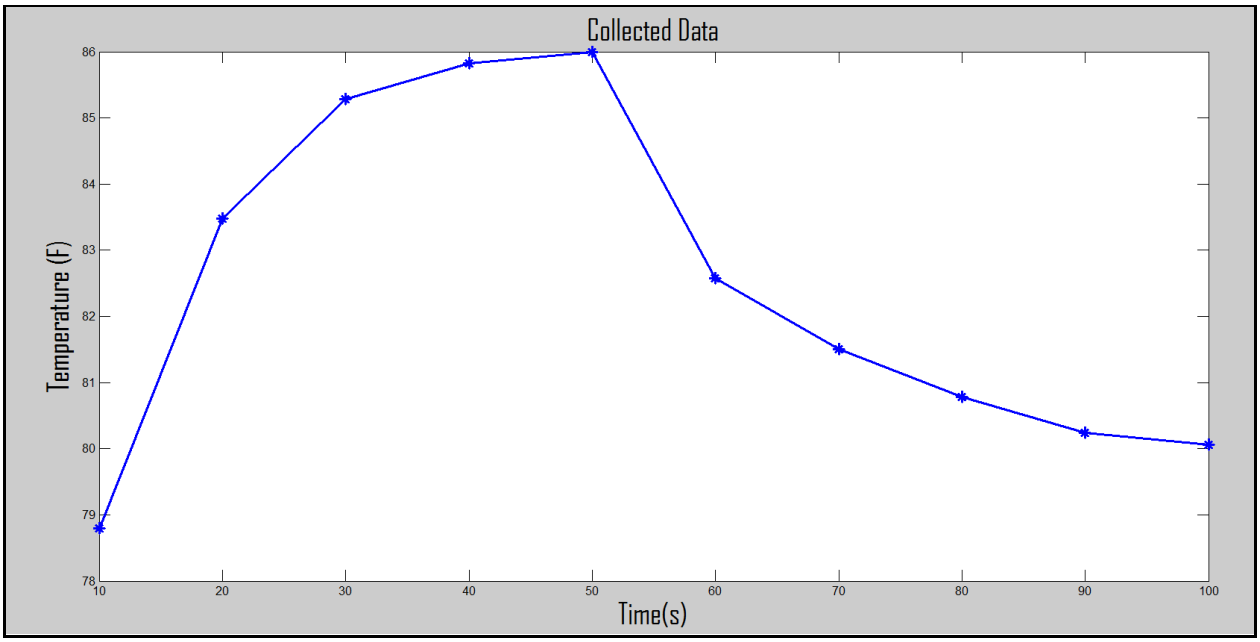
take the values in structure **a** and places them in the row victor **dat**. Finally, the line **`data=[data;dat]`** builds the ,matrix named **data**. It does this using a process called concatenation. The new value of **data** is the old

value of **data** with a new row added to the bottom. The new row being the data contained in row vector **dat**. Note that matrix **`data`** grows each time we add a new row to the matrix. This type of matrix is called a dynamic matrix because it changes in size as the program runs. In general, this method is discouraged because the process of growing a matrix means that we have to allocate memory each time the matrix grows. This process is very slow. The method has the benefit that the matrix has the correct size when we are done, not too big and not too small. A different way of doing it would have been to declare data at the beginning, such as **`data=zeros(1000,3)`**. The problem with this is that we need to pick a size that we know is greater than or equal to the amount of data we will collect. In this application, it is easy to size this array. In a real world application, we may need to pick a giant array to accommodate the largest amount of data we anticipate collecting. When all of the data on the port has been read, the while loop will exit. We then close the port and clear s1.

An example of the collected data is shown below:

```
>> Lab8_MATLAB2
Send the data, then hit the Enter key.
>> data
data =
   10.0000   78.8000   29.4926
   20.0000   83.4800   29.4920
   30.0000   85.2800   29.4905
   40.0000   85.8200   29.4908
   50.0000   86.0000   29.4887
   60.0000   82.5800   29.4923
   70.0000   81.5000   29.4941
   80.0000   80.7800   29.4920
   90.0000   80.2400   29.4893
  100.0000   80.0600   29.4938
>>
```

We can now use MATLAB to generate plots and do cool stuff with the data:



Demo VIII.3: Demonstrate the operation of your data collection routine. Collect sensor data every 10 seconds, store a total of 10 rows of data, transfer the data to MATLAB, show the contents of the data array, and then plot the temperature data versus time.

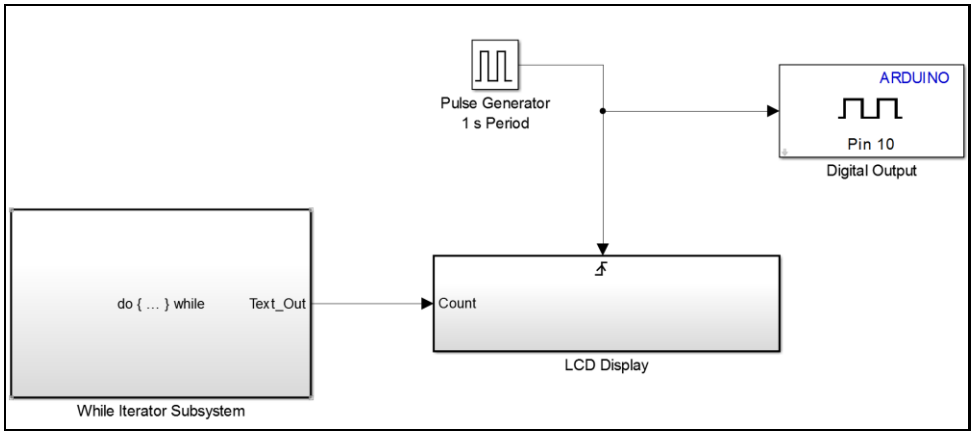
E. Receiving Serial Data

Next, we will look at sending a serial text string from MATLAB to the Arduino Mega. The ultimate goal will be for MATLAB to send a command to the Arduino, the Arduino responds by sending the collected data to MATLAB, and then the MATLAB script records the data. As a first step, we will send a text string from MATLAB to the Arduino, and the Arduino will display the text message on the LCD screen. We will use the script below to send the text string:

```
s1 = serial('COM19');
fopen(s1);
str='';
while ~strcmp(str,'End')
    str=input('Enter a Text String. (''End'' to terminate.)\n', 's');
    fprintf(s1, str);
end
fclose(s1);
clear s1;
```

The script prompts the user for an input string, and then outputs that string on the COM port. Since we want to test the serial transmission with several different texts strings, we use a while loop to repeatedly ask for and send a text string until the user types 'End.' Note that you should use the string compare function (**`strcmp`**) instead of **`str=='End'`**, because you cannot use == if two text strings are of different lengths. If the strings are of different lengths, you will get a runtime error and the script will terminate.

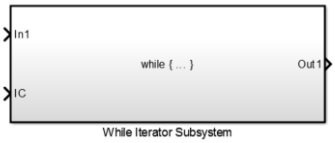
We will use the Simulink model shown below:



One of the reasons for doing this example is to introduce the **While Iterator Subsystem**. Logically, you can use this subsystem to implement a while loop or a do-while loop. Both forms repeatedly execute the blocks in the subsystem until the condition becomes false. With a while loop, the condition is tested before the blocks in the subsystem are executed. This allows for the possibility that the blocks are never executed if the condition is false at the beginning. For a do-while loop, the blocks within the subsystem are executed before the condition is tested. This imposes that the blocks within the loop be executed at least once, and the exit condition is

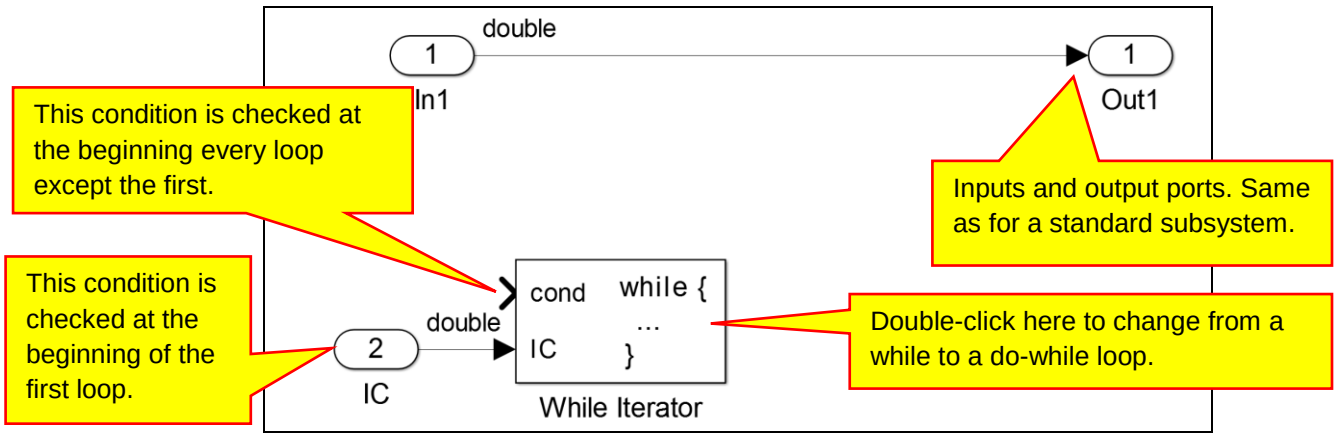
tested after the blocks have performed the task. Note that the blocks within the loop must somehow make the condition false or an infinite loop will result[5].

When placed in your model, a **While Iterator Subsystem** looks like:



The in1 and out1 ports are the same as in any subsystem. You can have as many or as few inputs and outputs as you would like. The **IC** port is for use when you are using the subsystem as a while loop. **IC** is the initial value for the condition test. If the value of **IC** is false, none of the blocks in the While Iterator Subsystem will execute. If the value is true, the blocks will execute once, and then the condition will be checked internally. The IC input is needed because the condition must be checked first, before any of the blocks inside the subsystem are executed. Note that **IC** signal must come from a signal outside of the While Iterator Subsystem, because it is checked before any of the blocks inside the subsystem are executed, so they could not possibly produce the signal.

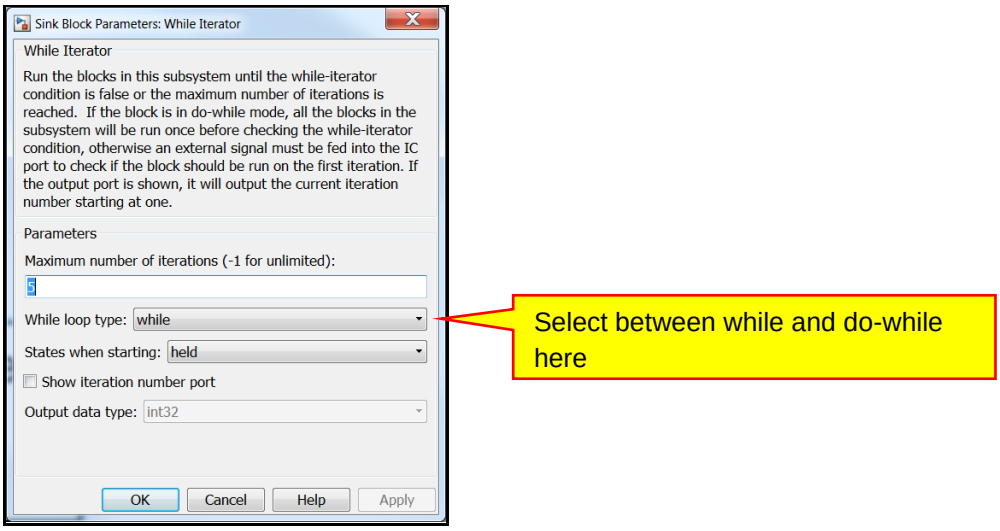
Inside the While Iterator Subsystem, you will see the following:



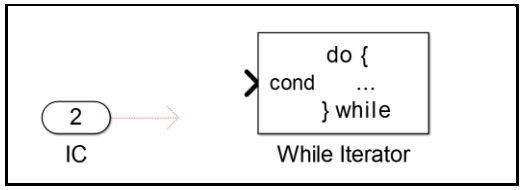
The **While Iterator** block controls the looping. The first loop, and possibly more loops, will occur if the IC signal is true. If the first loop occurs, the condition will be checked, and looping will continue until the condition is false. Remember that the condition is checked before any blocks in the subsystem are executed.

Double-click on the **While Iterator** block to open it:

[5] It takes the Arduino a very long time to complete an infinite loop!

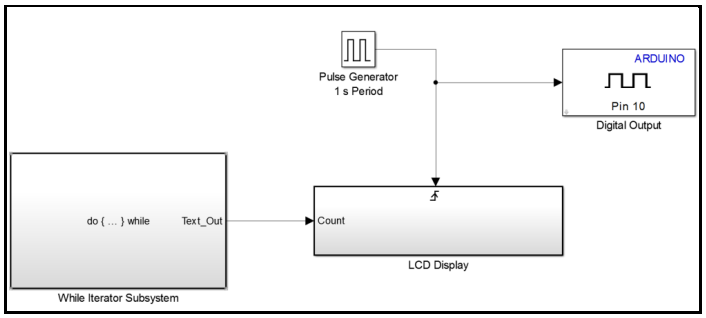


Change the type to a do-while and return to the subsystem:

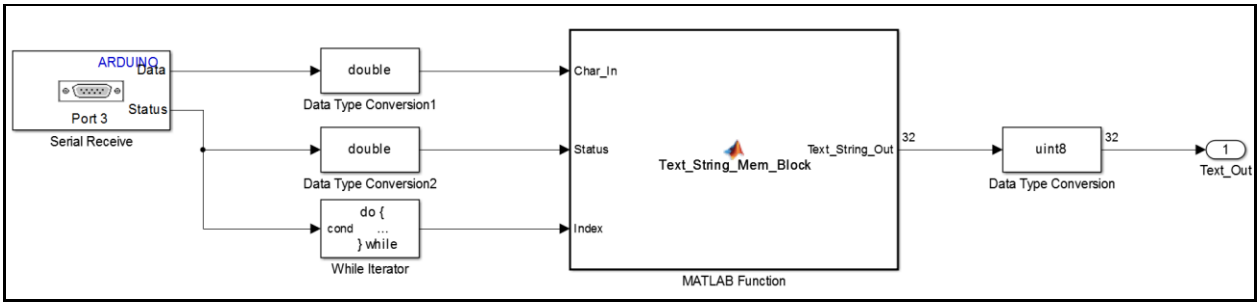


Notice that the **While Iterator** no longer has an IC input. This is because the condition is checked after all blocks in the subsystem have executed, so the condition can always be generated within the subsystem. We can thus, delete the **IC** input port because it is not used by the While Iterator.

Now that we know a little about While and Do-While loops, we can create our model. Our top-level model looks as shown:



Notice that we are using a do-while loop. Our **While Iterator Subsystem** is shown below:



We will use the Arduino **Serial Receive** block and specify port 3. This block is slightly different than the Serial Transmit block. When the status signal is 1, data is available on the port. When data is not available the status signal is zero. We will use this status signal for the exit condition for the while iterator so that we will loop until all of the data is read from the port. Every time the Serial Receive block executes, it will read one byte of data from the serial port (one character). If no data is available, it will output a code of 255 so that we can test and ignore it. If data is available, it will read a single byte (one ASCII character). To read all of the data, we need to loop until the port status says that not more data is available. Thus, we use a do-while loop to read all of the characters in the serial port buffer.

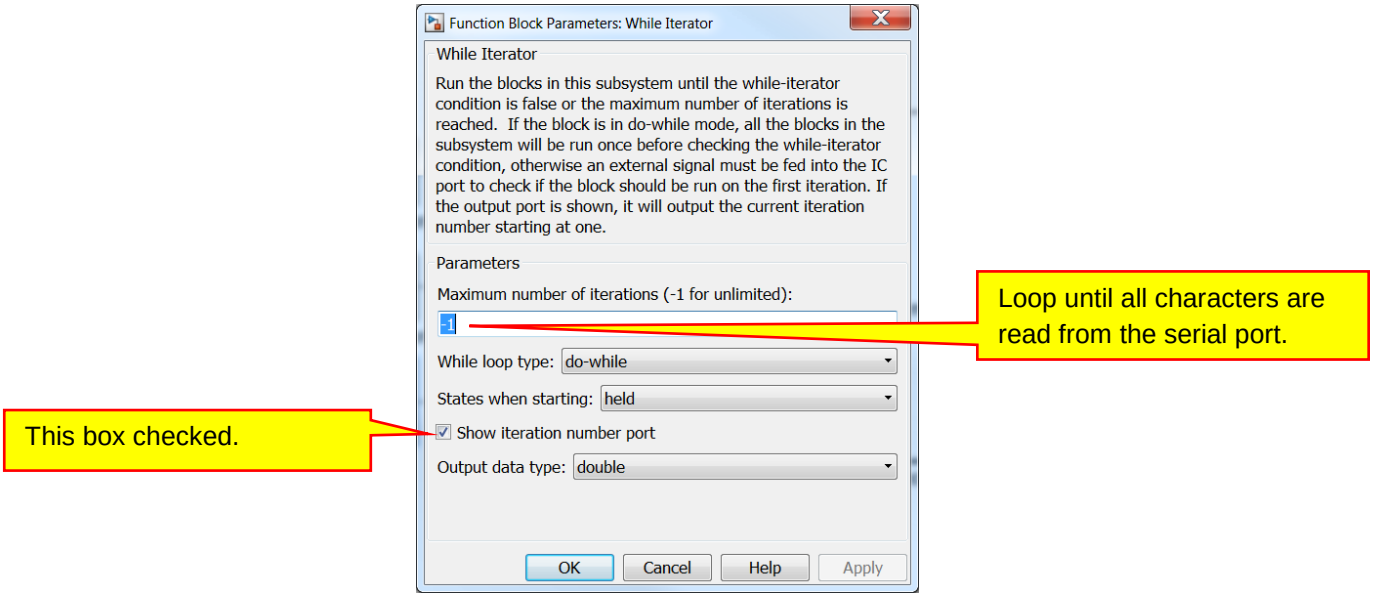
Our goal is to read a text string and output it to the LCD display. Thus, if more than 32 characters are available on the serial port, we will read all of them but only save the first 32 characters. We will use a MATLAB function to create a memory block to store the 32 characters. If the do-while loop retrieves more than 32 characters, we will ignore the extra characters. Note that to read 32 characters, the while loop will have to loop 32 times. The MATLAB function is similar to memory functions that we have used before except that we have a few conditions on how we store the data:

```
function Text_String_Out   = Text_String_Mem_Block( Char_In, Status, Index )


persistent Text_String;

if isempty(Text_String)
    Text_String='                                ';
end

if Status == 1 && Char_In ~= 10
    if Index ~= 1
        if Index <= 32
            Text_String(Index) = Char_In;
        end
    else
        Text_String(1) = Char_In;
        Text_String(2:32)='                                ';
    end
end

Text_String_Out = abs(Text_String)';
end
```

Don't forget this transpose operator (').

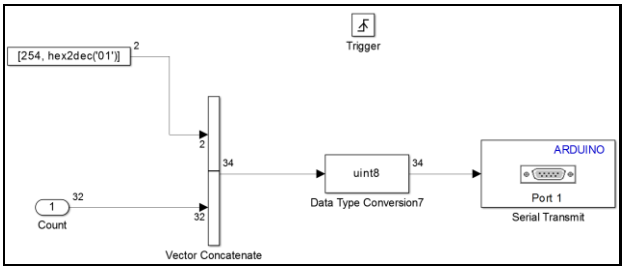
- If this is the first character in the string, save the character and replace the remaining characters by spaces. Remember that the data storage is persistent. If this is the second text string we have received, then we need to erase what was saved previously.
- If the index is between 2 and 32, save the character in the text string.
- Only change the saved data when Status ==1. (Only change the saved data when data is available on the port, which means the user sent a new text string.)
- The MATLAB script sends a new line (\n) character at the end of our text strings. This character has the ASCII code of 10. If we receive this code, ignore it.
- Ignore all characters after the 32 characters. Read them and remove them from the serial port input buffer, but don't add them to the array.

The index for the MATLAB Function is obtained from the **While Iterator** block:



The iteration port is a counter that outputs the current iteration number. It starts at one and increases. We use this as the index to our data storage.

The LCD Display subsystem is shown below.



All it does is prepend the text string with the code to clear the LCD display and set the pointer to the first character. The subsystem is edge triggered and executes once a second.

Demo VIII.4: Show the operation of your serial receive subsystem. Test it for any length text string including 1 character, 32 characters, more than 32 characters, and the 'End' text string.

Exercise VIII.1: Create a Simulink model and MATLAB script that accomplish the following task. The Simulink script collects time, temperature and pressure data the same as in Demo VIII.3 except that it stores sensor data every 5 seconds, and when the memory is full with 10 rows of collected data, the data is automatically transmitted on the serial port. The MATLAB script waits for the data and automatically concatenates the 10 rows of data to the data array with no user intervention. After receiving the data, the MATLAB script sends out the character "R" and then waits for more data to be sent. When the Simulink model receives the "R" character, it sets the memory index back to 0 and collects another 10 rows of data. When the memory is full again, the Simulink model automatically transmits the data again. The MATLAB script automatically receives the data and concatenates it to the data array. This process should continue until the script collects 100 rows of data, all of which are saved in a single array. The MATLAB script should output a status message each time it receives data so that we can monitor its progress. An example would be:

```
Received 10 rows of data.
Received 20 rows of data.
Received 30 rows of data.
```

Lab IX
Collecting Data 2

We will continue our quest to collect data in this lab and also expand our toolkit to include storing data on an SD card. The SD card will give us the option of storing gigabytes of data in non-volatile flash memory. This solves the two problems in countered in Lab VIII of storing data in volatile RAM and transmitting the data over the serial port. Here, we will write the data to a MicroSD card in a fashion that we will not lose data if the Arduino resets or loses power, and we transfer data to another computer by removing the MicroSD card from the Arduino and inserting it into another computer.

For this application, we will be using a MicroSD Breakout board from Adafruit Industries [3]. The one difference between this example and previous examples is that this breakout board uses an Serial Peripheral Interface (SPI) BUS rather than the I^2C Bus covered earlier.

A. The SPI BUS

The SPI bus is a 4-wire bus, as compared to the I^2C bus which uses two wires. There will be one master device and then one or more slaves. Unlike the I^2C bus that requires an address to determine the slave that is being addressed, the SPI bus uses a dedicated Slave Select (SS) line. The lines are usually referred to by the following terminology [4]:

- SCLK – Serial clock – provided by the master.
- MOSI – Master output, slave input
- MISO – Master input, slave output
- SS – Slave select (active low and output from the master)

In our case, the master will be the Arduino Mega and the slave will be the MicroSD breakout board. The terminology for the MicroSD card is slightly different:

- CLK – Serial clock – (Should connect to SCLK on the master.
- DO – Data out – (Output data of the slave; connect to MISO of the master.)
- DI – Data In - (input data of the slave; connect to MOSI of the master.)
- CS – Chip select (active low and output from the master.)

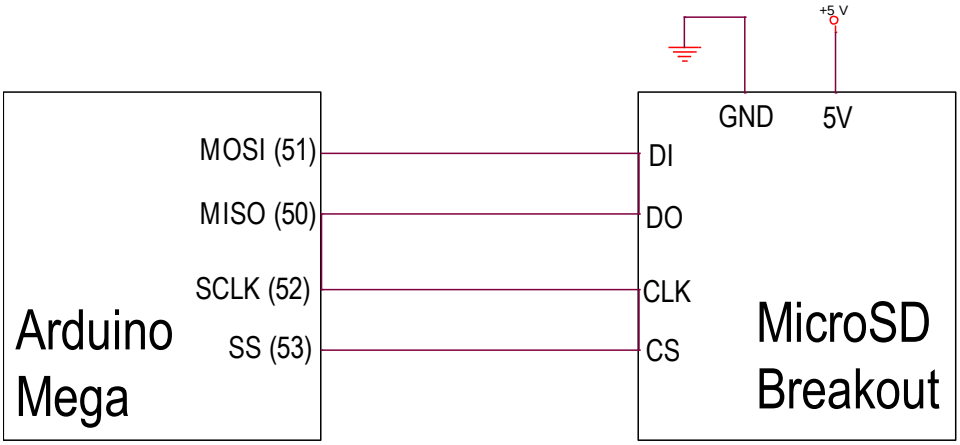
The SPI pin connections for the Arduino Mega are detailed on the Arduino website [5]:

- MISO – Pin 50
- MOSI – Pin 51
- SCK (SCLK) – Pin 52
- SS – Pin 53

Note that the SS pin on the Arduino does not have to be pin 53, and we specify a pin number in our S-Function. Typically, pin 53 is used for this purpose, so we will use it in this lab. However, we could specify a different pin if need. MISO, MOSI, and SCK are always pins 50, 51, and 52 for the Arduino Mega.

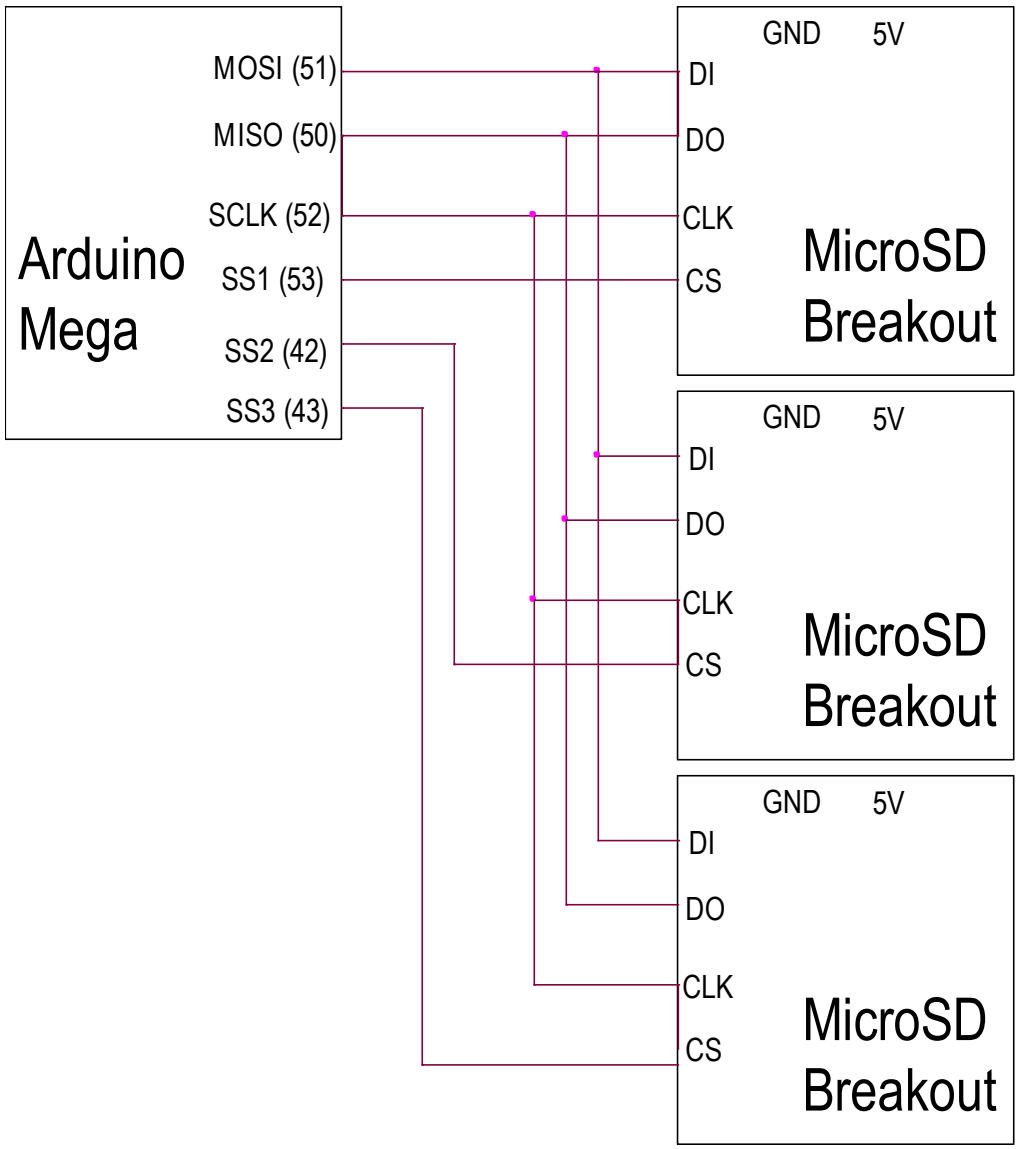
The pin connections for the MicroSD breakout board are silkscreened on the board:

DO of the MicroSD board should go to MISO of the Arduino Mega and DI of the MicroSD should go to the MOSI of the Arduino Mega. We will use the wiring diagram shown below:



The CD pin on the MicroSD card is the Card Detect pin and is pulled low when an SD card is inserted. If you wish to use this pin, you will need to use a pull-up resistor and then connect the CD pin to one of the digital inputs of the Arduino.

Our example only has a single slave. The idea of a bus is that it can connect the master to several slave devices. The slave select line determines which slave device the master will communicate with. Just for illustration purposes, if we had several slaves, all of which were MicroSD Breakout boards, our wiring diagram would look as shown below:



Notice that the DI, DO, and CLK lines are all connected together and form the SPI bus, and that each slave uses a dedicated CS line.

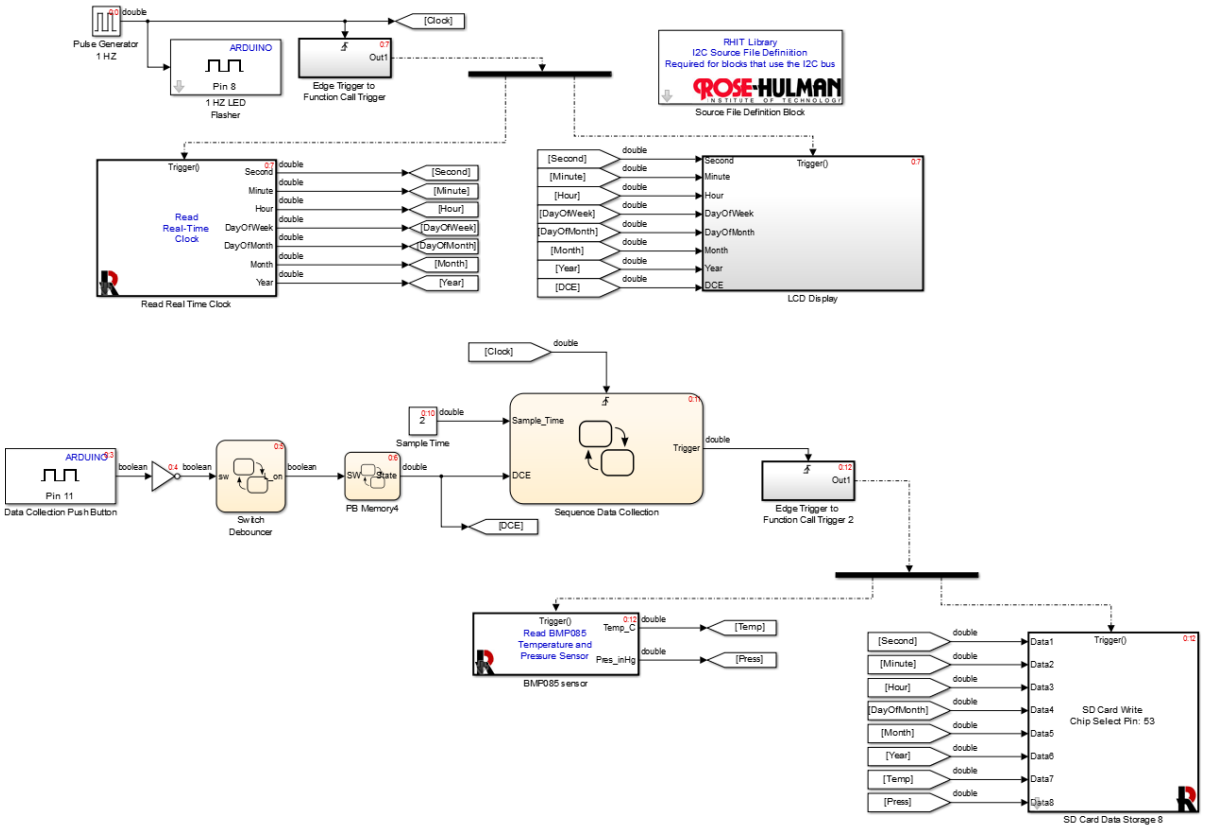
B. Data Collection Model

Now that we know how to wire the MicroSD breakout board to the Arduino Mega, we will make a model that measures data with a sensor and then writes that data to the SD Card. The model will implement the following features:

- 1 Hz Flashing LED to show that the Arduino is alive.
- Overrun detection.
- A pushbutton to enable and disable data collection. Data collection can be enabled and disabled as many times as desired by the user.
- The LCD displays the current date and time, and indicates whether data collection is active. (Data collection being active means that data is being saved to the MicroSD card.)
- The data collected includes a time/date stamp, the current temperature at the time/date stamp, and the current pressure at the time/date stamp.

- Data should never be lost if the Arduino is reset or if the Arduino loses power.

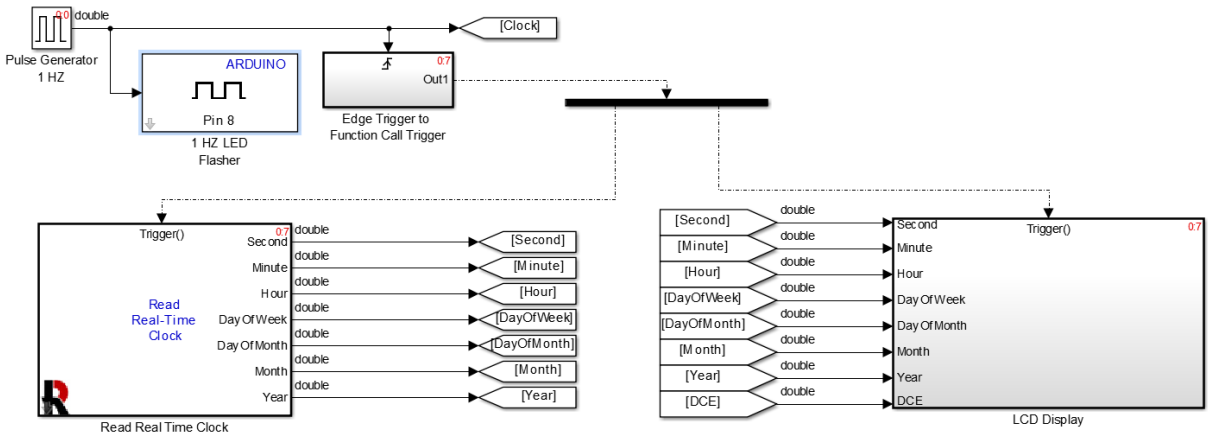
The complete model is shown below:



1. Data Collection Model Sorted Execution Order

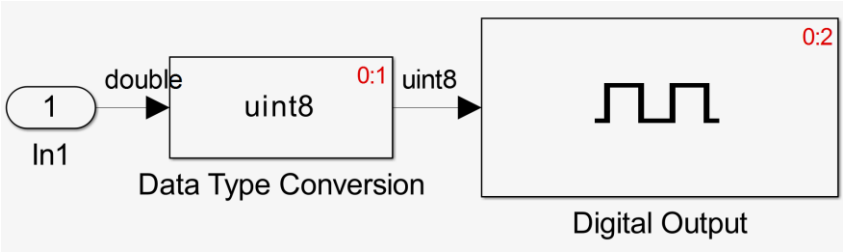
We have displayed the sorted execution order so that we can determine the sequence in which the blocks are executed. (Select **Display**, **Blocks**, and then **Sorted Execution Order**.) For an example like this where we are collecting data and applying time stamps to the data, the order that we execute blocks can be important as we would like the time stamp and the time at which the data was collected to be the same. (Or, In this case, as close to each other as possible.)

We will zoom in on the real-time clock and the LCD display:



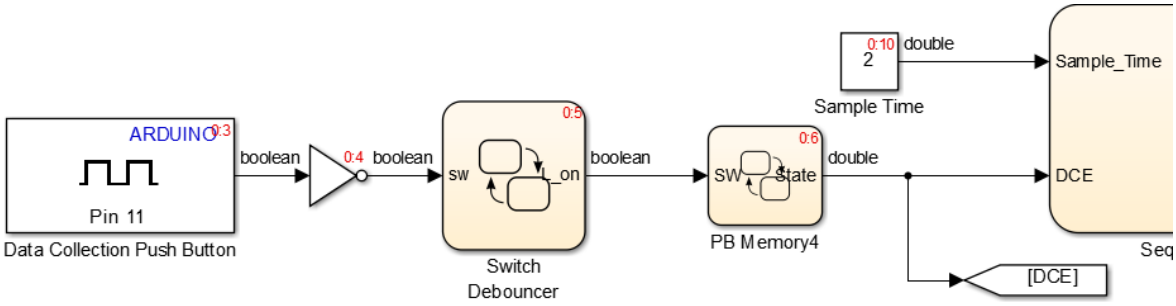
The first block to be executed in the model is the 1 Hz pulse generator, as this is indicated by the sorted order notation **0:0**. The first zero indicates the system index. Note that a system index of 0 is the root level of the model. The second zero is the block position and indicates the order the block is executed within it system. Thus, **0:0** indicates that the 1 Hz Pulse Generator is the first block executed in the model as it is at the root level and has the lowest block position.

The next block to be executed is within the 1 Hz LED Flasher subsystem, although we can't tell by looking as the sorted execution order is not displayed on the block. If you look at the subsystem below the mask, you will see the following:



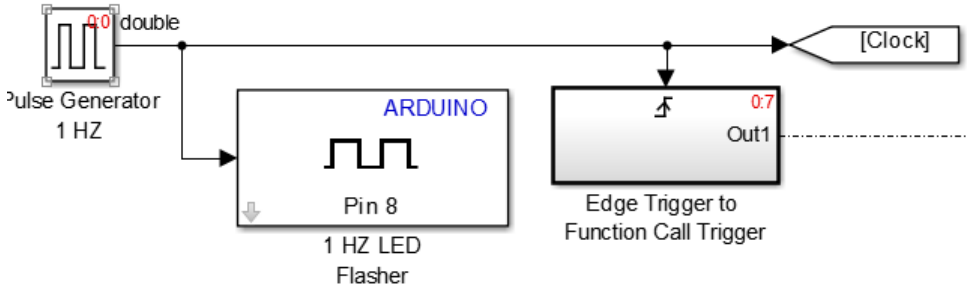
We see that although the blocks are within a subsystem, they are still at the root level. This is because a basic subsystem is just a convenient method of grouping blocks that would otherwise be at the root level. We see that the **uint8** block is at the root level and executes after the 1 Hz Pulse generator. The **Digital Output** block is also at the root level and executes third.

The blocks that execute next are blocks indicated by 0:3, 0:4, 0:5, and 0:6, and these are part of the model that will be discussed later:



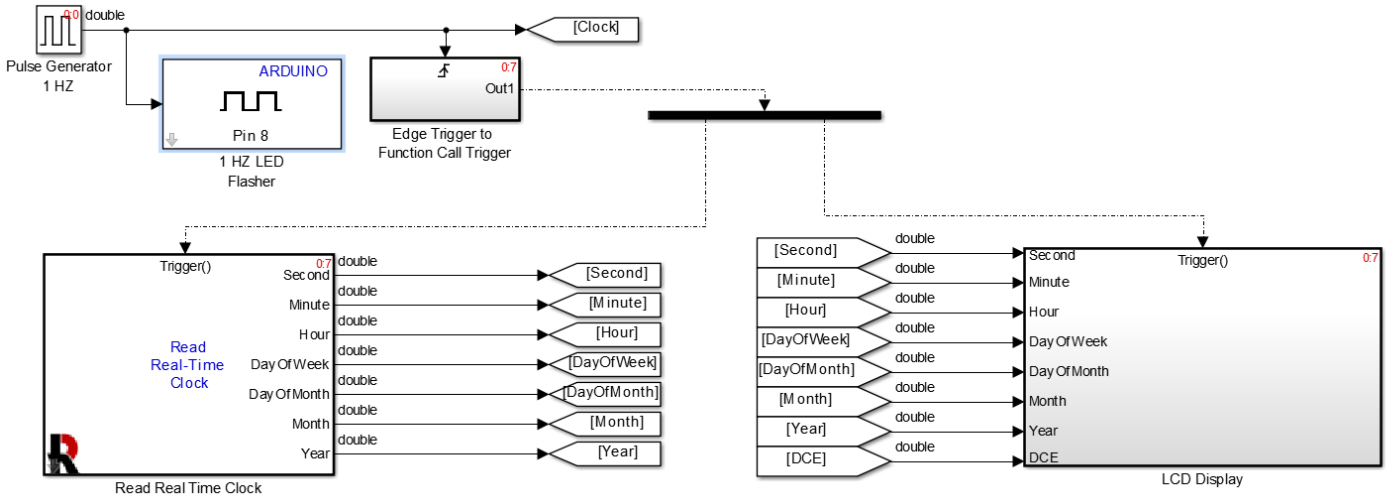
The order of the above blocks does not affect the portion of the system we are looking at.

The Edge Trigger to Function Call Trigger block executes next as it is system index **0**, block position **7**:

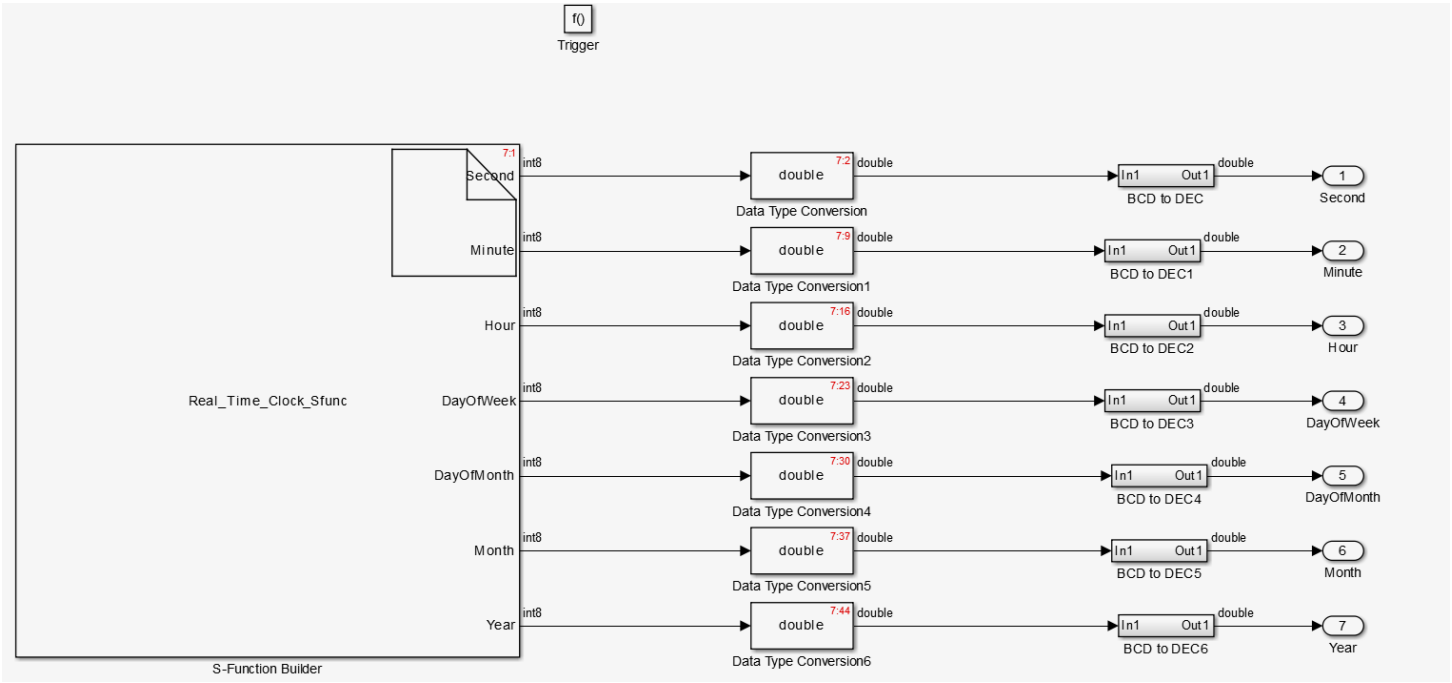


Notice that this block is at the root level (system index 0) and executes 8th.

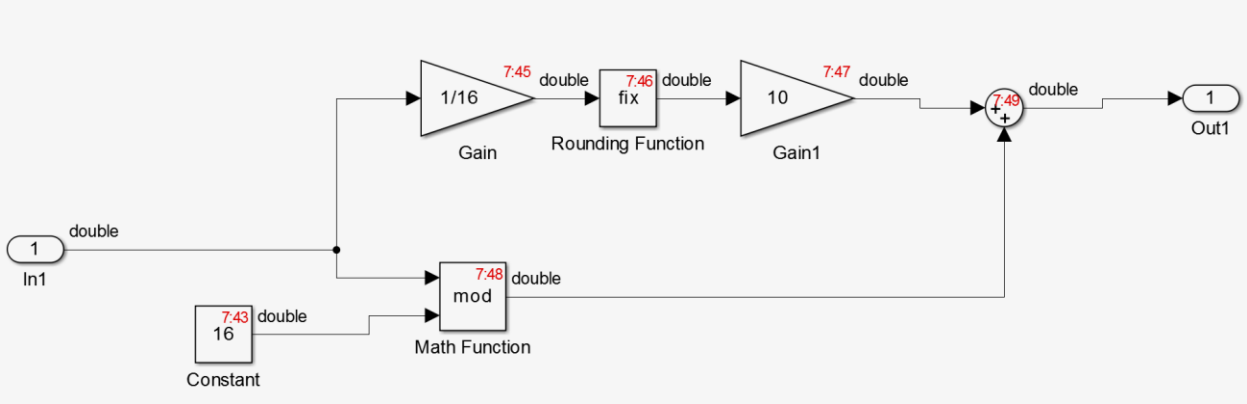
Returning to the root level, we see that the **Edge Trigger to Function Call Trigger**, **Real Time Clock** and the **LCD Display** all execute 7th:



Because we are using a Demux, the Real Time Clock and LCD Display subsystems execute as a group (indicated by 0:7) The Demux specifies that the **Real Time Clock** block executes before the **LCD Display** block executes. If we look inside the **Real Time Clock** subsystem, we will see that all the blocks are part of the system referred to by system index 7:



If you expand all of the BCD subsystems, you will see that all of the blocks are referenced by system index 8:



Including all of the blocks in the BCD subsystems, there are a total of 49 blocks

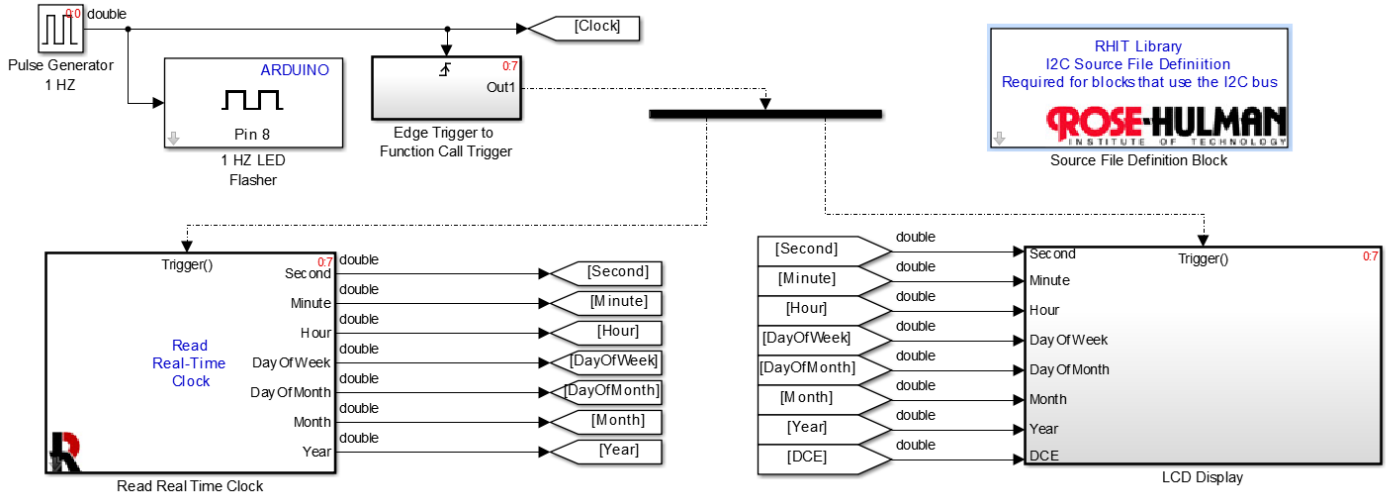
Looking at the sorted execution order numbers for our complete model, shown on page 164, we can determine that the blocks in the model are executed in the following order:

0. 1 Hz Pulse Generator (0:**0**)
1. Data Type Conversion (Within the 1 Hz LED Flasher) (0:**1**)
2. Digital Output (Within the 1 Hz LED Flasher) (0:**2**)
3. The Data Collection Push Button (0:**3**)
4. The inverter (0:**4**)
5. The Switch Debouncer (0:**5**)
6. PB memory4 (0:**6**)
7. Edge Trigger to Function Call Trigger (0:**7**)
8. Real Time Clock ***7***
9. LCD Display ***7***
10. Sample Time (Constant Block) (0:**10**)
11. Sequence Data Collection (Chart) (0:**11**)
12. Edge Trigger to Function Call Trigger 2 (0:**12**)
13. Pressure and Temp ***12***
14. SD Card Data Storage ***12***

What we get out of all of this is that although the Pulse generator is the master 1 Hz clock for the entire system, we read the real time clock at step 8 and we read the data from the pressure and temperature sensors at step 13. Thus, the time we read the clock and the time we measure the data are not the same. Furthermore, there are a number of significant and time consuming operations being executed between steps 8 and 13. Because we are sampling data at a 0.5 Hz rate (or slower) and it only takes a few milliseconds to execute the code between steps 8 and 13, the time difference for this application will be trivial. If we needed high speed data collection, the delay between the time stamp and the data sample may become significant, making this method inappropriate.

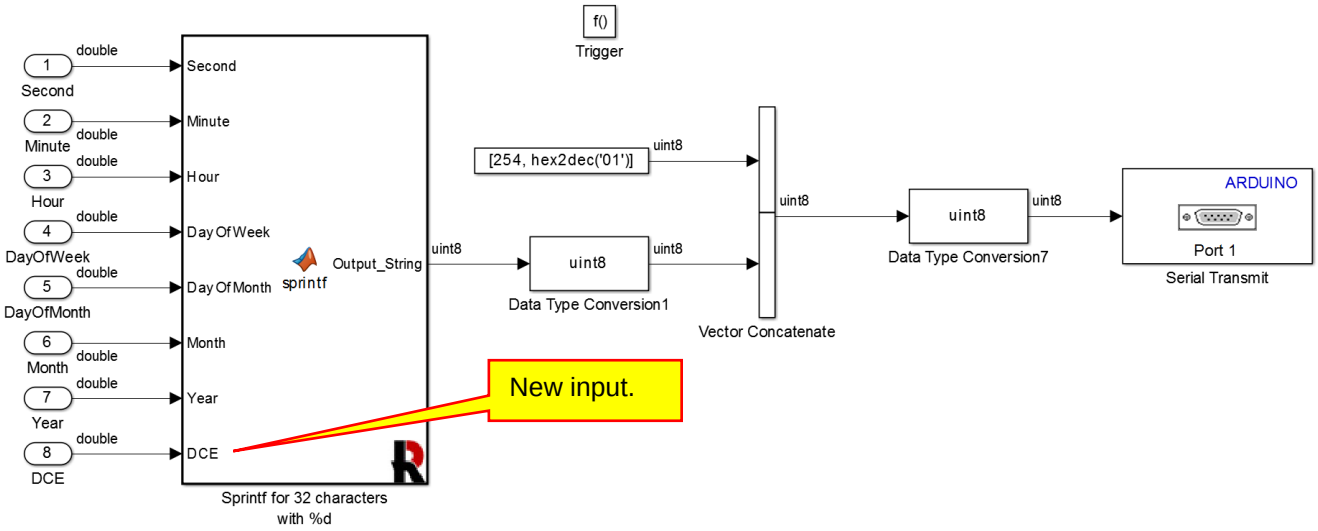
2. Data Collection Model Logic

We will now discuss the logical design of this model. Many of the blocks have been used before and will only be referenced in this discussion. The top portion of the model reads the **Real Time Clock** module every second and displays the information on the LCD screen:



The **Edge Trigger to Function Call Trigger** was discussed on page 165. It runs every time it receives a positive edge and the only thing it outputs is a function call trigger. This function trigger, in combination with the **Demux**, causes the **Real Time Clock** and **LCD Display** to run in a specific sequence. The Real Time Clock Runs to read the data, and then the LCD Display runs to display the data. The Real-Time Clock module was provided as a library module and covered in Section VI.B.2.

The LCD Display has been used numerous times. We do note that the subsystem has an input called DCE, which stands for “Data Collection Enable.” This signal is provided by the pushbutton input, and is used to turn on and off data collection. When data collection is disabled, the LCD will display the date and time. When data collection is enabled, the LCD will display the date, the time, and the word “Data.” The only modification to the LCD is in the sprintf MATLAB Function and the added input called DCE:



The MATLAB Function has been changed as shown below:

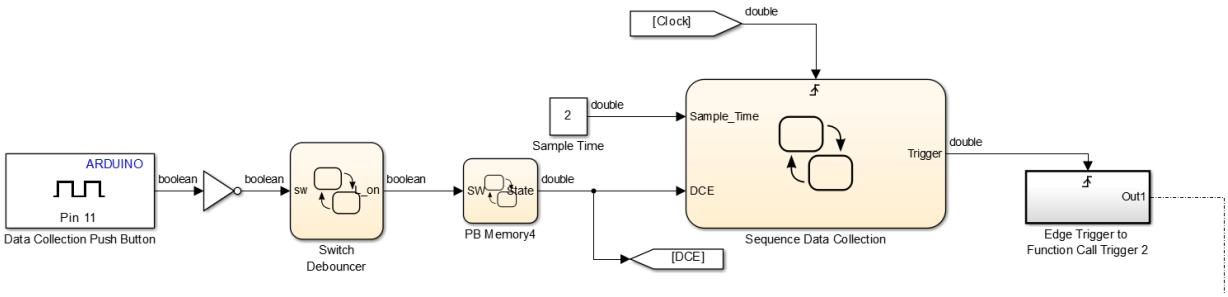
```
23 -    Sec=int16(floor(Second));
24 -    Min=int16(floor(Minute));
25 -    Hr=int16(floor(Hour));
26 -    DOM=int16(floor(DayOfMonth));
27 -    Mo=int16(floor(Month));
28 -    Yr=int16(floor(Year));
29
30 -    if DCE==1
31 -       Format_String='"Time: %02d:%02d DataDate: %02d%c%02d%c20%02d"';
32 -       coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
33            Hr, Min, Mo, '/',DOM, '/', Yr);
34      else
35 -       Format_String='"Time: %02d:%02d     Date: %02d%c%02d%c20%02d"';
36 -       coder.ceval('sprintf',coder.wref(data_array),coder.opaque('const char *',Format_String),...
37            Hr, Min, Mo, '/',DOM, '/', Yr);
38      end
```

Test “Data” displayed here.

This ‘/’ goes in this %c.

We notice that the **if** statement chooses between two different **sprintf** commands. If DCE is equal to 1, the text “Data” will be displayed as the last four characters of the first line of the LCD display. We also note that in order to display the date as month/day/year, we need to use the %c format string to specify a single character and specify a / as we would a string variable, ‘/’. Also note that the %d format strings are specified as %02d rather than %2d. Both strings mean two total spaces with no decimal places. However %02d means display a number like 5 as 05. %2d still uses two spaces, but leading zeros are not displayed. This format is needed to specify time and dates correctly such as 08:05 rather than 8:5 and a date as 08/22/2003 rather than 8/22/203.

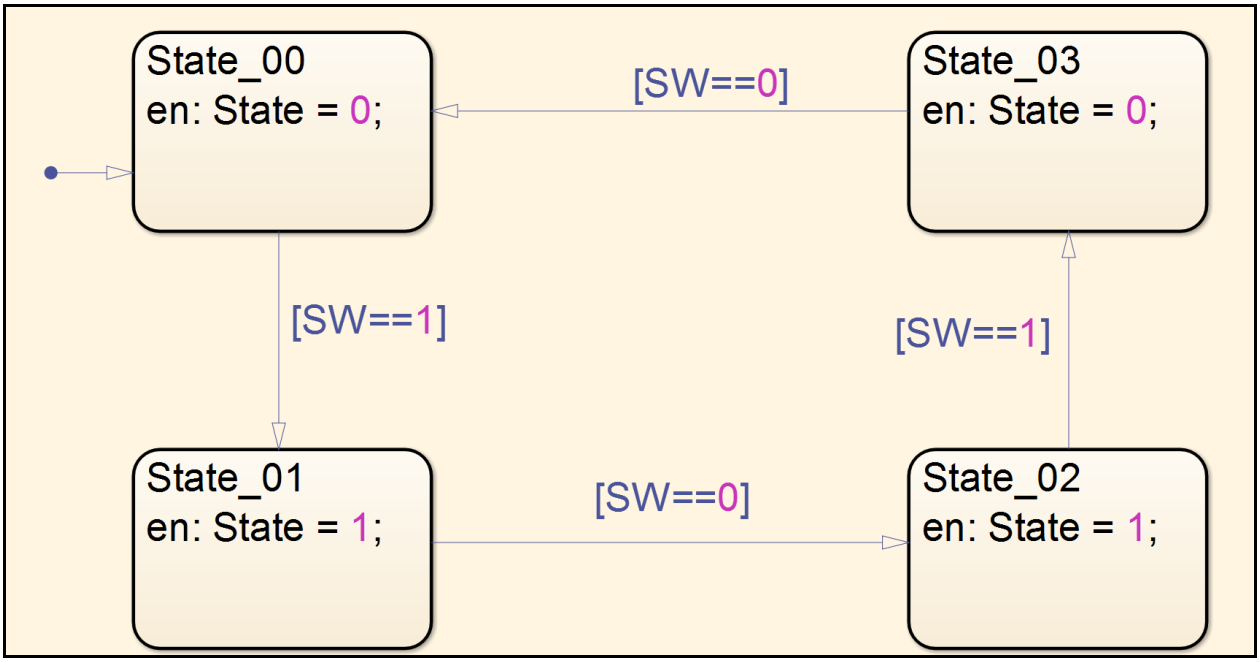
Next, we will look at the pushbutton portion of the model:



We have seen the Switch Debouncer in Section V.C.2.a). The PB Memory4 Stateflow chart implements the following logic:

1. The initial value of State is 0.
2. When we press the pushbutton, State changes to 1.
3. When we release the pushbutton, State remains at 1.
4. When we press the pushbutton, State changes to 0.
5. When we release the pushbutton, State remains at 0.
6. The sequence from 2 through 5 then repeats.

The purpose of this logic is that the state changes when we press the pushbutton, but in order to change from a one to a zero and a zero to a one, the pushbutton must be released before it is pressed again. The Stateflow chart is shown below:



The **Sequence Data Collection** Stateflow chart is used to sequence when data is written to the MicroSD card. Note that it is triggered by the 1 Hz clock, so this chart is run once a second. The purpose of the chart is to generate an edge trigger to read the sensors and then write the data to the MicroSD card. The constant **Sample Time** is, during normal data collection, how often a sample is taken. In the example, the Sample Time is 2, so a data point will be logged every two clock cycles. Since the Clock has a frequency of 1 Hz, a data point will be logged every two seconds. We chose this fast sample rate so that we can collect some data in a short amount of time. In a real application, say measuring the air temperature throughout the day, a sample time of 10, 100 or even longer could be used. We could go to a faster sample rate as well, but we do have limitations in the Arduino speed and the fact that we have a fixed time step (which you need to choose).

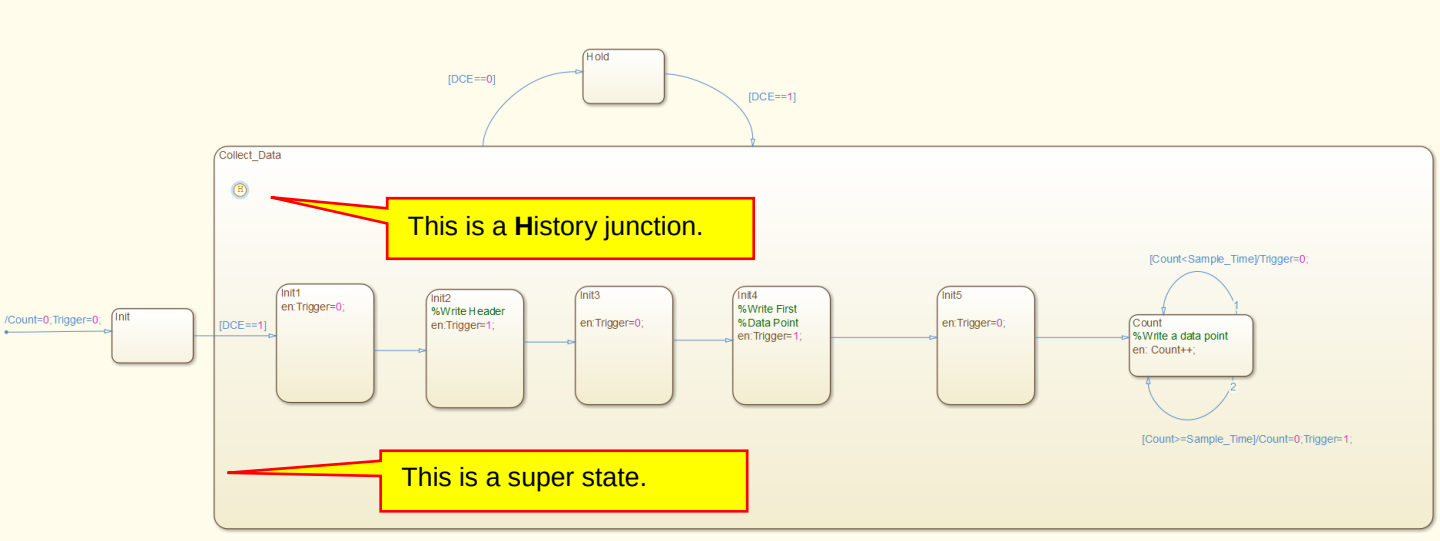
The Sequence data Collection chart implements the logic below:

1. After power-up or a system reset, hold. Don't do data collection. Hold until the data collection is enabled.
2. When data collection is enabled for the first time:
    a. Leave the Trigger at 0.
    b. Wait one second.
    c. Set trigger to 1.
        i. This will trigger the sensor S-Function for the first time which will run the initialization portion of the S-Function.
        ii. This will trigger the MicroSD card S-Function the first time which will run initialization routines and write the header to the data file.
    d. Wait one second.
    e. Set the trigger to 0.
    f. Wait one second.
    g. Set trigger to 1.
        i. This will trigger the sensor S-Function and measure data.
        ii. This will trigger the MicroSD card S-Function which will write a data point to the MicroSD card.

h. Wait one second.
i. Set the trigger to 0.

3. When the sequence above completes, toggle the trigger between 0 and 1 at the rate specified by the constant Sample Rate.
4. If, at any point during the sequence listed above, data collection is disabled, the trigger holds at its present value. When data collection is re-enabled, the system will continue exactly where it left off.

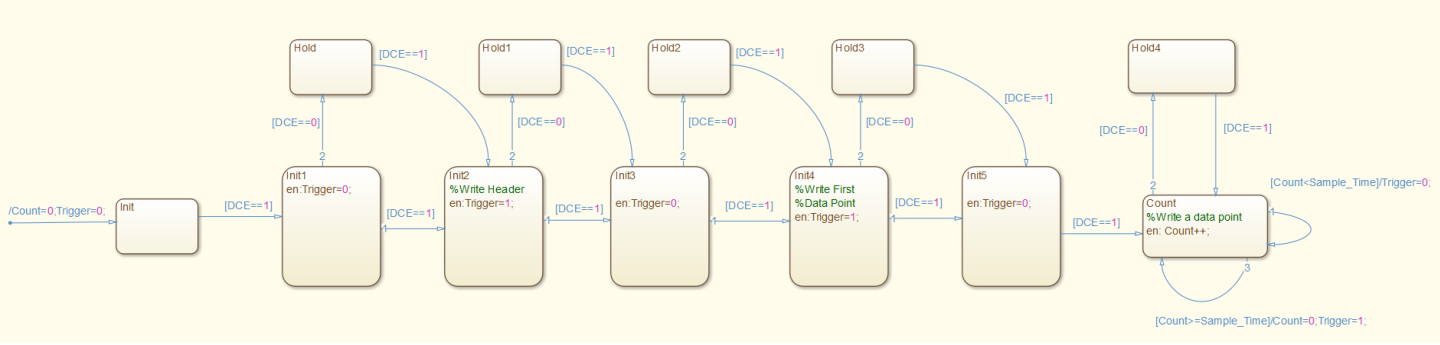
The purpose of this sequence is to write the file header and then take a data point immediately. Then collect data at the specified sample time. This method is used because we are designing this system for long sample times. Suppose the sample time is ten minutes. When you first enable data collection, nothing would happen for 10 minutes. It would make more sense to take a data point immediately and then wait 10 minutes for the next sample. The Stateflow chart is shown below:



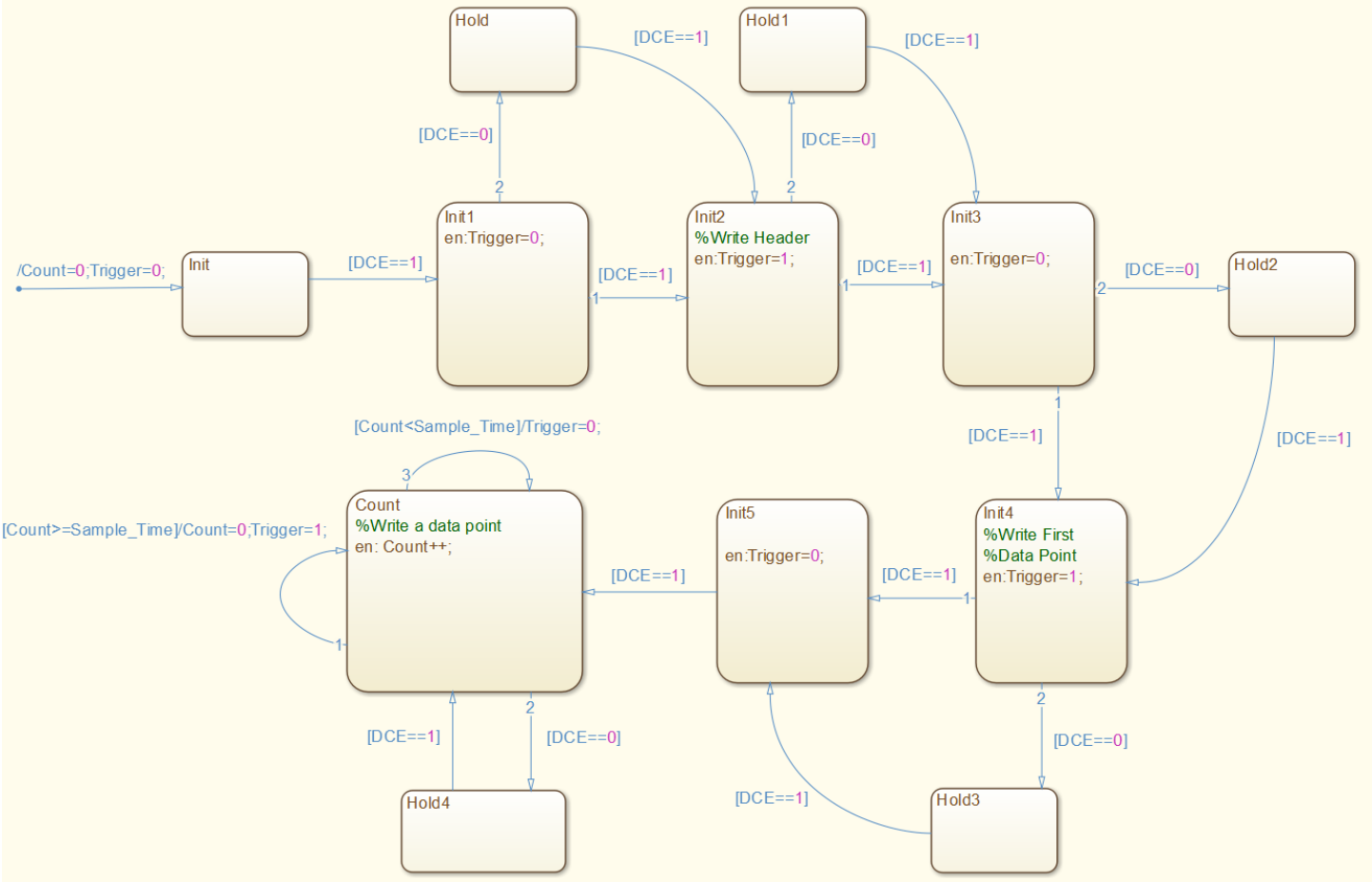
Notice the history junction  . We will get back to it in a moment. Also notice the **Collect_Data** super state and that it contains many states within it, **Init1**, **Init2**, and so on. Superstates were covered on Page 134. This superstate makes two things easier for us. The chart follows the path **Init1**, **init2**, **Init3**, **Init4**, **Init5**, and then **Count**. If we are in any of these states and DCE is 0, we need to hold. Having the transition [DCE==0] on the super state gives us an easy way to do this. The history junction remembers where we were in a superstate when we exited the super state. Suppose we were going through the sequence of states init1, Init2, Init3, and so on, and then data collection becomes disabled. The chart exits the superstate and goes to the Hold state. When data collection is re-enabled, because we are using a history junction together with the superstate, the chart will jump back into the superstate where it left off and then continue. Thus, the history junction allows us to pause and return to the sequence of steps.

The chart below performs the same function without using super states and history junctions[6]:

[6] I think it does. I did not test it…And I don't want to. It's ugly…

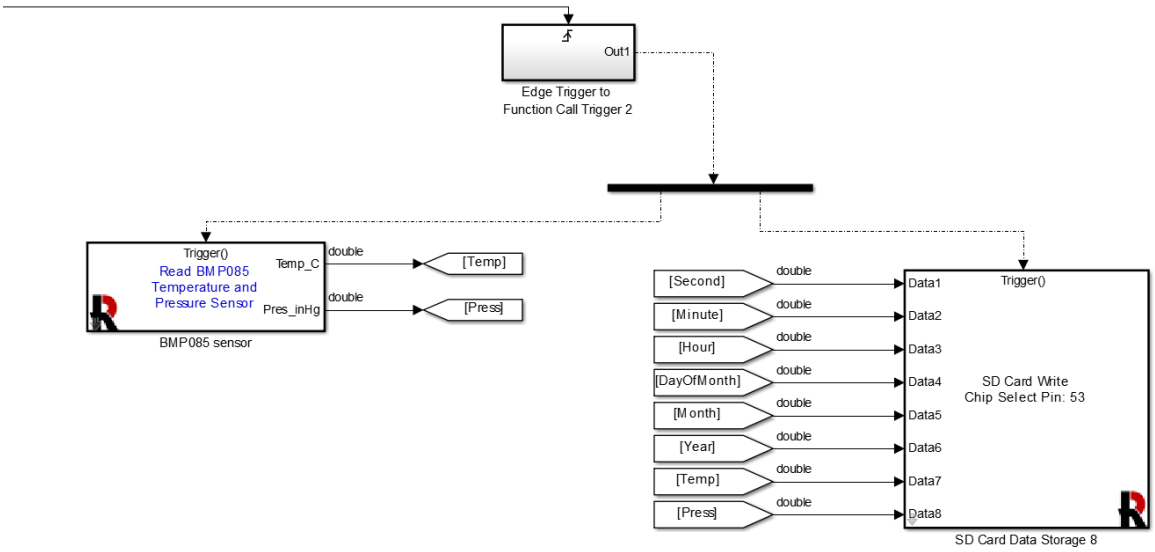


The chart has the same linear progression, but is unreadable in the screen capture, so the same chart is rearranged below:



Clearly the superstate and the history junction make for a more compact and easy to understand chart than the one shown above.

The Sequence Data Collection chart triggers the sensors and then the MicroSD card:



The SD Card Write block writes data to an SD Card using an SD card reader/writer on the SPI bus. The block was originally developed using the SD Card reader available from adafruit (http://www.adafruit.com).Values Data1 through Data8 are double-precision values. The values are saved on the SD Card in a file named Datalog.csv. The file format is a CSV file (comma separated values). The values are written with two decimal places of accuracy. Unused inputs should be set to zero with a constant.
The block opens the data file, writes the data to the MicroSD card with each number separated by a comma, and then closes the data file. Note that the data file is closed immediately after writing the data. This is done to minimize the possibility that the data file is not closed properly when something bad happens. Something bad could be a power failure or accidental reset. If these happen while the file is open, the file could be corrupted and we might lose data. For long sample times, using this method keeps the file closed most of the time. It is only open for the fraction of a second it takes to write the data file.

An example data file is shown below:

```
3.00, 1.00, 2013.00, 9.00, 3.00, 36.00, 70.50, 29.46
3.00, 1.00, 2013.00, 9.00, 13.00, 37.00, 70.50, 29.46
3.00, 1.00, 2013.00, 9.00, 23.00, 36.00, 70.90, 29.46
3.00, 1.00, 2013.00, 9.00, 33.00, 36.00, 71.20, 29.46
```

Note that the data values are separated by commas. This format is referred to as a comma separated values (CSV) file, and can be read easily with most spreadsheet programs and with MATLAB.

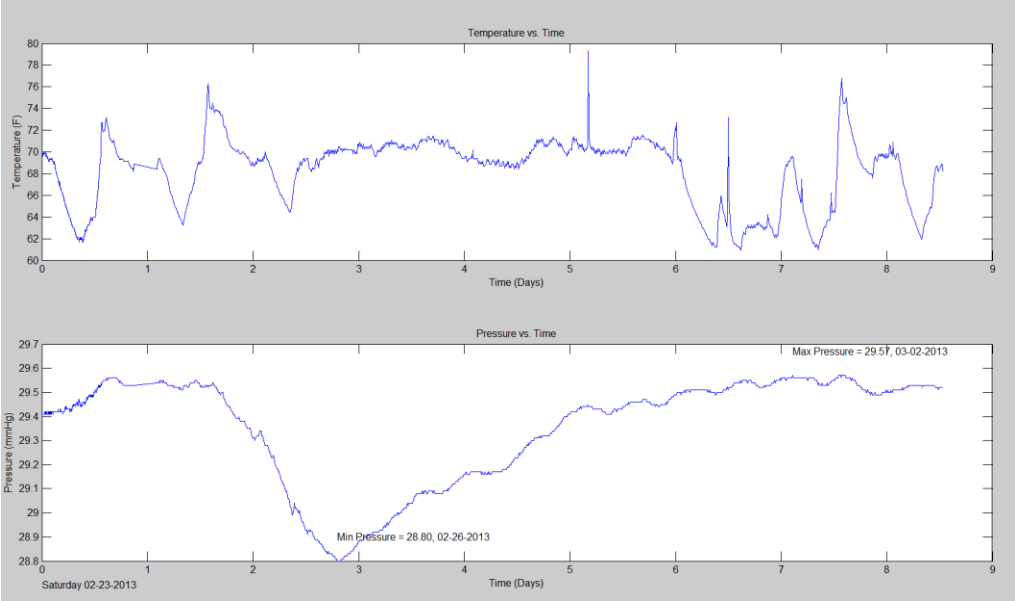
Demo IX.1: Demonstrate the operation of your Data collection model:

- Data collection can be turned on and off.
- The LCD shows the current time, date, and if data collection is on or off.
- The SD Card collects data.
- Show the contents of a collected data file.

Exercise IX.1: Write a MATLAB script that reads your data and generates a figure containing two plots with the following information:

- A plot of temperature versus time.
- A plot of pressure versus time.
- Display the starting date on your plot.
- Prints the max pressure and date on the plot. (Date is optional.)
- Prints the min pressure and date on the plot. (Date is optional.)

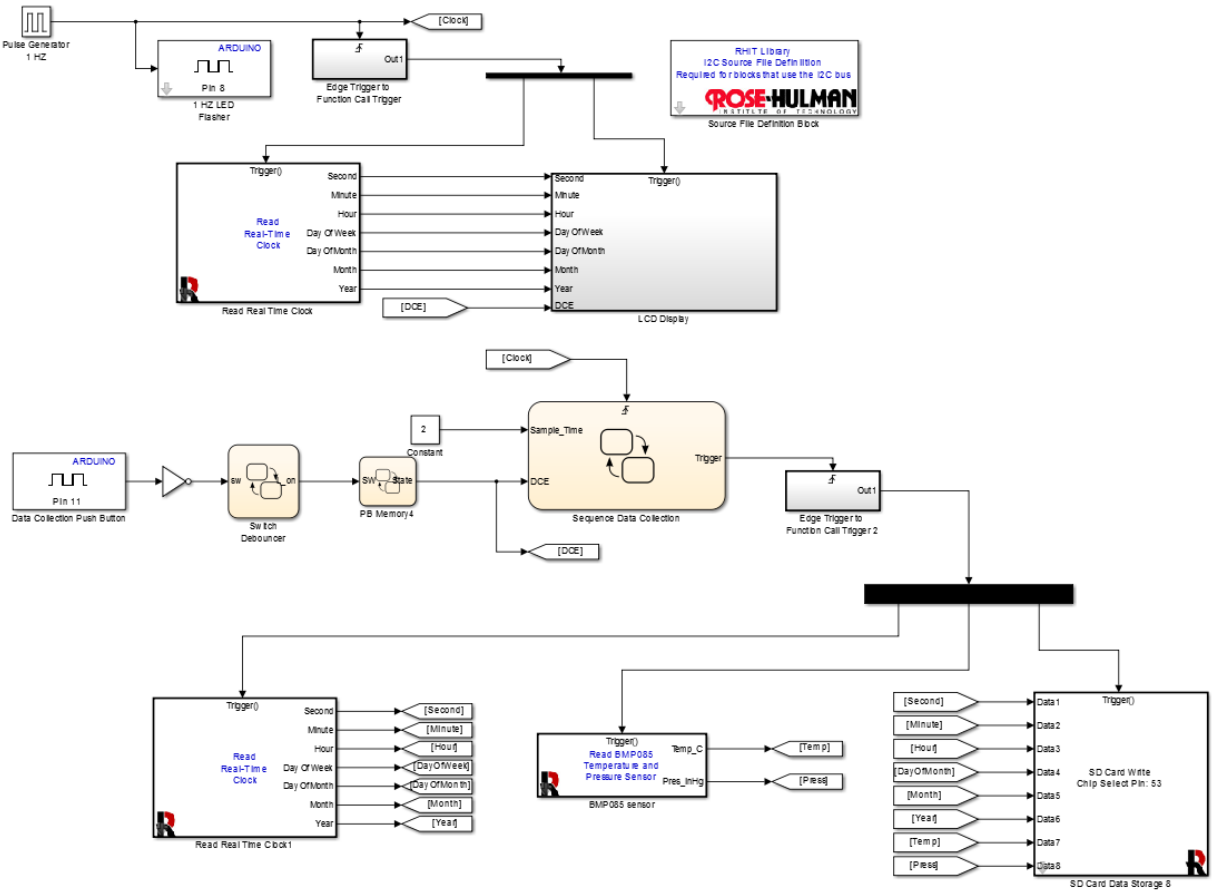
You may want to use the following MATLAB commands: **`uigetfile, csvread, datenum, datestr, subplot, figure, text, sprintf, max, min, xlabel, ylabel, title`**. An example plot is shown below:



You don't need to let your data collection run for 9 days.

3. Improved Data Collection Model

The full model shown on page **Error! Bookmark not defined.** has the problem that a significant amount of time could elapse between when we read the real-time clock and when we sample the sensor, meaning that the time stamp is not the actual time the sensors were sampled. To "fix" this problem, we use the model shown below:



We see that we read the Real-Time Clock twice in the model. Once for when we display the clock on the LCD and again immediately before we sample the sensors. This reduced the time delay between the sample and the time stamp. Note that it does not eliminate the problem; it just reduces the time delay. Note that the block **Real Time Clock1** is not executed if we are not collecting data. With this model, we need the other Real Time Clock block to read the time because we want to display the time on LCD even when we are not collecting data.

Exercise IX.2: Modify your data collection Simulink model to use the Chip Detect (CD) pin of the MicroSD breakout board. You model must enforce the logic listed below:

- Data collection cannot be started if the card is not inserted.
- If data collection enabled and then the card is removed, data collection should stop independent of the pushbutton state. It cannot be restarted until the card is reinserted and the user presses the push button.
- If the card is removed and then reinserted, and while the card is being reinserted the user also holds down the pushbutton, data collection will not resume until the user releases the pushbutton and then presses it again.
- The correct order for starting data collection is that the card must be inserted with the pushbutton released, and then the pushbutton should be pressed to enable data collection.

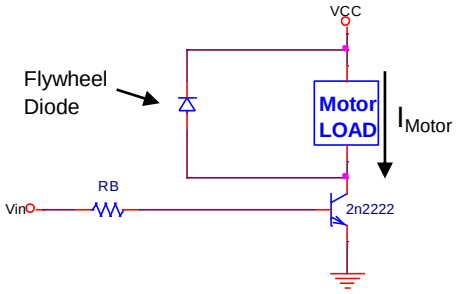
Exercise IX.3: Create a model that has the same functionality as the model of Exercise IX.2 with the addition that LCD toggles every two seconds between displaying the time and date or the temperature and pressure. This is a good opportunity to use the Simulink **IF** and **IF Action Subsystem** blocks. These blocks are located in the Ports & Subsystems library. For both displays, the LCD should still display if data collection is enabled or not.

Lab X
Motors

In this lab we will cover the use of three basic motors, a DC motor where we could control speed and torque, a stepper motor where we will control speed and direction, and a servo motor where we will control the angle or position. This section is only basic and introductory. Motors are used everywhere from powering toys, precision speed control for disk drives, to high power torque sources for hybrid and electric vehicles. This is the first step into a world where microcontrollers combine with power electronics to enable the control of huge amounts of power intelligently. Although we will be working with tiny motors, microcontrollers are used to control motors both large and small.

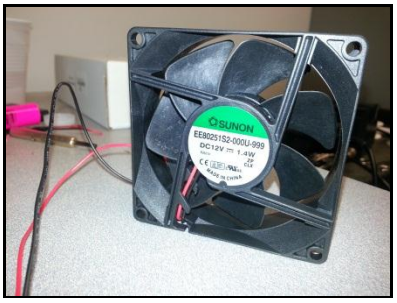
A. DC Motor

We covered the speed control of a DC motor in Section IV.C where we used pulse-width modulation (PWM) to control the speed of a tiny DC motor. We will use this same method here to control the speed of a cooling fan and add some logic for more intelligent control. Note that cooling fans come in all shapes and sizes. Some fans are AC and some DC. Here we will use a small DC fan as an example. The fan requires a larger current and voltage than can be provided by the Arduino logic output, so the high current driver from Section IV.C will be used again, and is repeated below:



All motors are inductive, so a flywheel diode is required as shown[7]. The only difference between the motor load and the fan load is the supply voltage and the amount of current the fan motor draws. Note that this driver allows us to drive a load that uses a different voltage than the controlling signal from the microcontroller. Signal Vin will be a digital output of the Arduino Mega and has a range of 0 to 5 V. The motor voltage in this example can be any positive DC voltage. For a high power traction motor, the voltage could be 200 V or higher. (Note that the high current driver will be more involved, but the principles are the same.) In our case, the motor voltage will be 12 V DC. When Vin is high (5 V), the transistor will turn on and current will flow through the motor, causing it to spin. If we leave the motor on long enough, it will eventually reach full speed. When Vin is zero, the transistor is off and the motor will slow down. If we leave the motor off long enough, it will stop. If we repeatedly turn on and off the transistor fast enough, we can control the fan speed as it will respond to the average of the amount of time we turn on and off the transistor. This method is called pulse-width modulation and was covered in detail in Lab IV. The only difference in this section is that the DC motor uses 12 V and the fan current is higher than our last motor example. We will use the DC fan shown below:

[7] See Section IV.C for an explanation of flywheel diodes.



Note that for this specific example, the fan we are using is a SUNON EE80251S2 and is rated at 1.4 W maximum. (Your fan may be different. Consult the data sheet or name plate for your specific fan.) This is the peak power that the fan will ever draw, and is the locked rotor current and the start-up current. We will design the driver for this power even though in steady-state the motor will draw a lower current. 1.4 W divided by 12 V yields a fan current of 117 mA, which we will round up to 120 mA. Note that the data sheet for this fan shown on page 270 specifies a current of 120 mA for this fan, which agrees with our calculation.

Looking at the PN2222A data sheet on page 271, we see that at a collector current of 150 mA the minimum value of DC current gain (h_{FE}) is 100:

P2N2222A

ELECTRICAL CHARACTERISTICS (T_A = 25°C unless otherwise noted) (Continued)

Characteristic	Symbol	Min	Max	Unit
ON CHARACTERISTICS				
DC Current Gain	h_{FE}			—
(I_C = 0.1 mAdc, V_{CE} = 10 Vdc)		35	—	
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc)		50	—	
(I_C = 10 mAdc, V_{CE} = 10 Vdc)		75	—	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, T_A = –55°C)		35	—	
(I_C = 150 mAdc, V_{CE} = 10 Vdc)(1)		100	300	
(I_C = 150 mAdc, V_{CE} = 1.0 Vdc)(1)		50	—	
(I_C = 500 mAdc, V_{CE} = 10 Vdc)(1)		40	—	

Min h_{FE} is 100 at 150 mA.

We will be using the PN2222A at around 120 mA. Figure 3 of the PN2222A data sheet shows that the value of h_{FE} at 150 mA is lower than the value of h_{FE} at 100 mA:

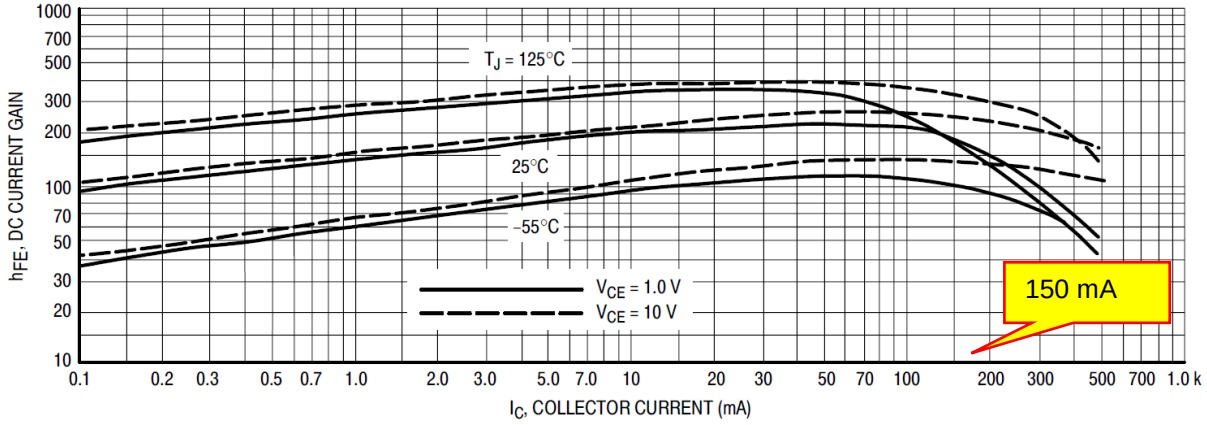


Figure 3. DC Current Gain

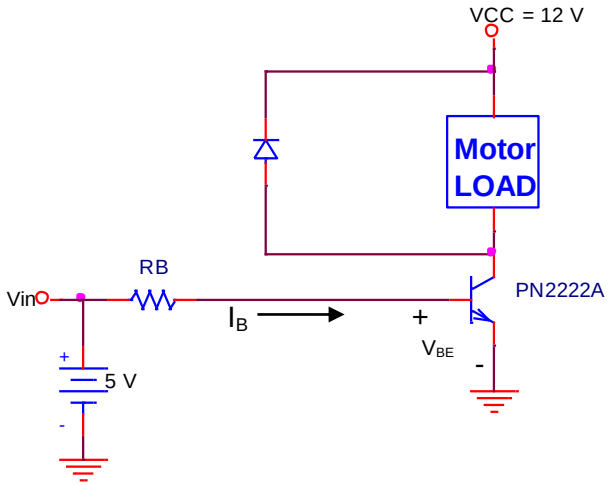
Thus, we expect the minimum value of h_{FE} to be larger than 100. However, using a smaller value of h_{FE} will make our base current larger, and this over design tends to turn on the switch more when it is on, which is OK.

For a collector current of 100 mA, we need a base current higher than of 120 mA divided by the current gain. We will choose a current that is 10% higher than this value to guarantee that the transistor switch is on:

$$I_B = \frac{1.1 \cdot I_{Load}}{h_{FEmin}} = \frac{1.1 \cdot 120mA}{100} = 1.32\ mA$$

So, when Vin is 5 V we would like the base current to be at least 1.32 mA. (Note that the Arduino digital output can source a lot more than 1.32 mA, so there is no problem driving the motor with this driver. The Arduino could not, however, drive 120 mA, so the transistor driver is required.)

We need to choose the base resistor R_B in the circuit below so that when V_{IN} is 5 V, the base current is at least 1.32 mA.



When V_{IN} is 5 V, the base current I_B is:

$$I_B = \frac{V_{IN} - V_{BE}}{R_B}$$

We want the value of base current to be greater than or equal to 1.32 mA, so

$$1.32\ mA \leq \frac{V_{IN} - V_{BE}}{R_B}$$

Solving for R_B:

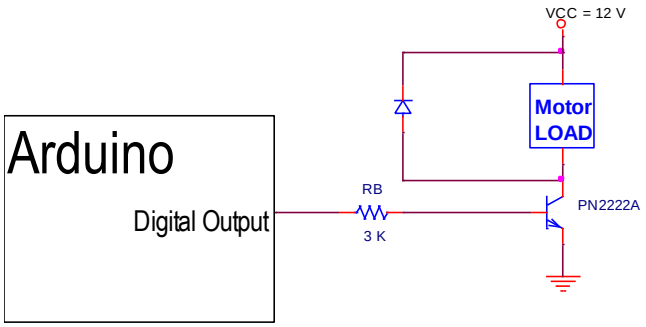
$$R_B \leq \frac{V_{IN} - V_{BE}}{1.32\ mA}$$

When the transistor is on, V_{BE} is approximately 0.7 V:

$$R_B \leq \frac{5\ V - 0.7\ V}{1.32\ mA}$$

Or, R_B = 3.25 k. Choosing the next lower standard 5% resistor gives us a value of R_B = 3.0 k, ±5%. Note that all of our design choices tended toward making R_B smaller, which yields a larger base current that turns on the transistor harder when it is on.

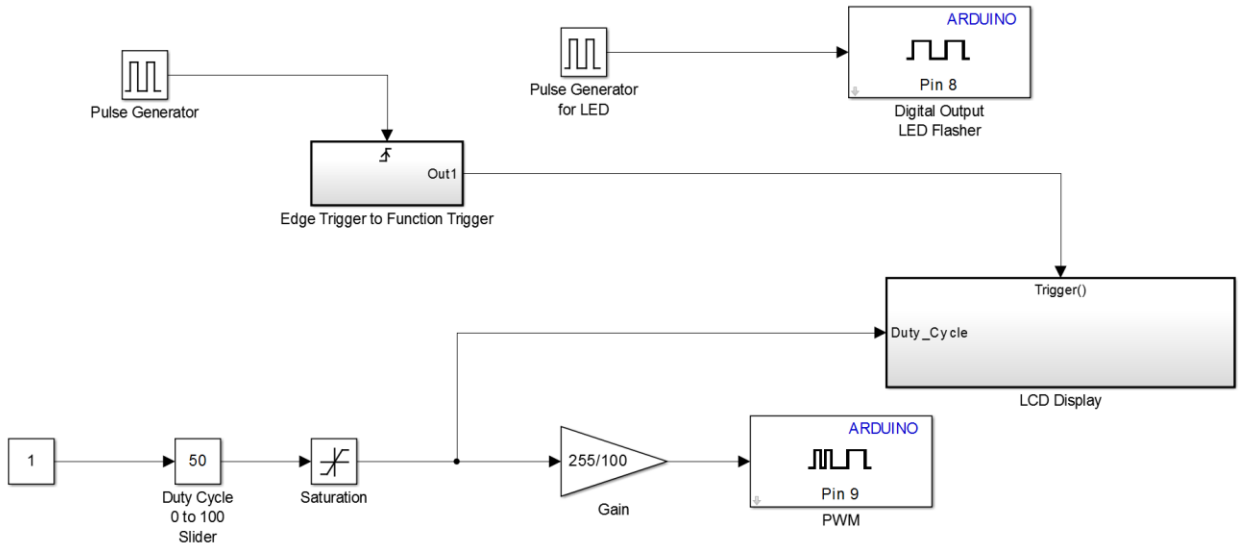
When the output of the controller is low (digital 0), the transistor will be off and there is no design as far as the circuit goes. Out final circuit is shown below:



The remainder of our design will be controlling the fan logically from the microcontroller.

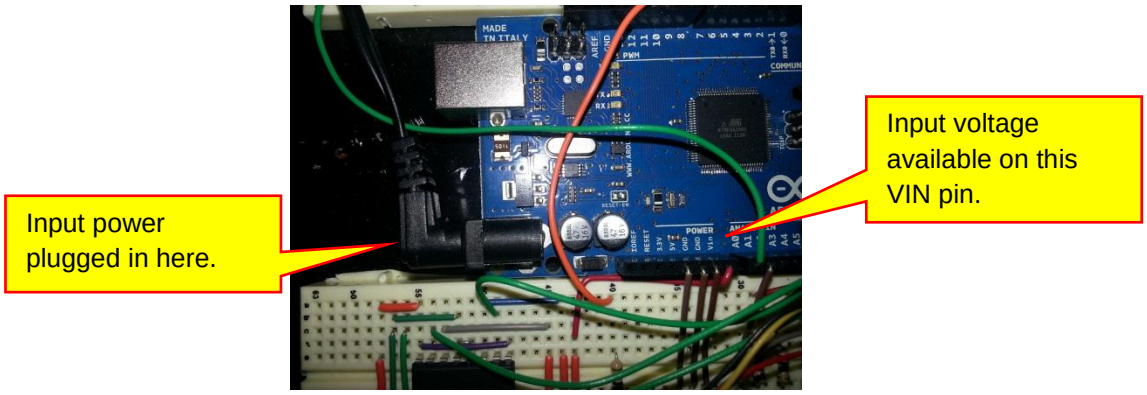
1. Fan Duty Cycle Response

Before we make a fan controller, we would like to determine the response of the fan to a specified duty cycle. When the fan is stationary, it will take a larger duty cycle to start the fan spinning than it will to keep the fan spinning. Create the model shown below:

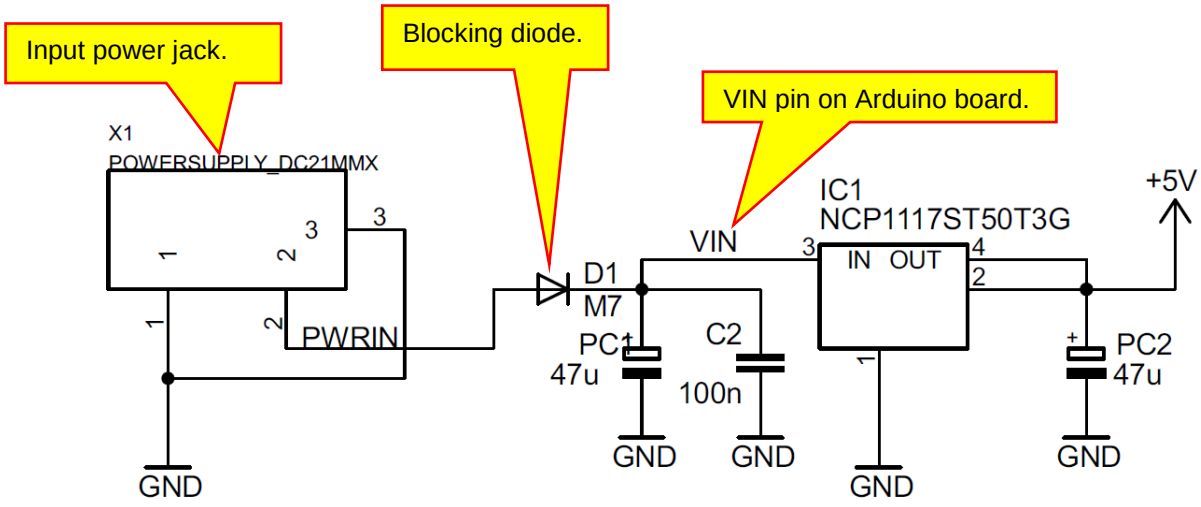


With this model we can set the duty cycle with a slider and then observe the response of the fan.

To power the fan, you will need to use an external 12 V DC power supply. Your lab kit has a power supply that you can plug into the wall and connect the 12 V output into the Arduino Mega power supply port:



Note that this supply will power your board as well as make a higher voltage available for use in your circuit. The power supply is connected to a pin called VIN on your board through a blocking diode:



The blocking diode prevents the board from being damaged by accidentally hooking up the wrong polarity voltage source to the power supply input. The voltage available on the VIN pin is the power supply voltage less a diode drop. We can use VIN as the 12 V source for the fan.

Demo X.1: Show that the fan speed can be controlled by changing the duty cycle. Determine: (1) The minimum duty cycle to start the fan when the fan initially is not rotating. (2) The value of duty cycle where the fan stops after it was initially rotating.

Exercise X.1: Create a fan controller that turns the fan off for temperatures less than 70 °F and full on for temperatures greater than 80 °F. Between temperatures of 70 °F and 80 °F, the fan speed varies “linearly.” Use the LCD to display the duty cycle and temperature and use one of the temperature sensors covered in a previous lab.

Exercise X.2: Create a two-speed fan controller that implements the following algorithm. When the temperature is greater than or equal to 75 °F, the fan spins at 50% duty cycle. If the fan is on at 50% duty cycle, the fan cannot turn off until the temperature is less than or equal to 70 °F. When the temperature is greater than or

equal to 80 °F, the fan spins at 100% duty cycle. If the fan is on at 100% duty cycle, it will not slow down to 50% duty cycle until the temperature is less than or equal to 75 °F.

B. Stepper Motor

For our next motor example, we will control the speed of a stepper motor. Every time you give a stepper motor a pulse, the motor will turn a fixed number of degrees. By controlling the pulse frequency, we can control the speed of the motor. Because of this characteristic, stepper motors are used where precise speed is required. A stepper motor has two or more windings, and rotating the stepper motor shaft requires that we pulse the various windings in the appropriate sequence. Also note that we can change the direction of rotation by changing the pulse sequence.

For this example, you can use any small stepper motor as long as you can find the data sheet that specifies the wiring diagram and the pulsing sequence. Here, we will use the bipolar Portscap 20M020D1B shown below:

This motor has a step angle of 18° which means it takes 20 steps to complete one revolution. Note that the complete data sheet is shown in the Appendix on page 275. The detailed specifications are shown below:

CAN STACK STEP MOTORS | 20M SERIES

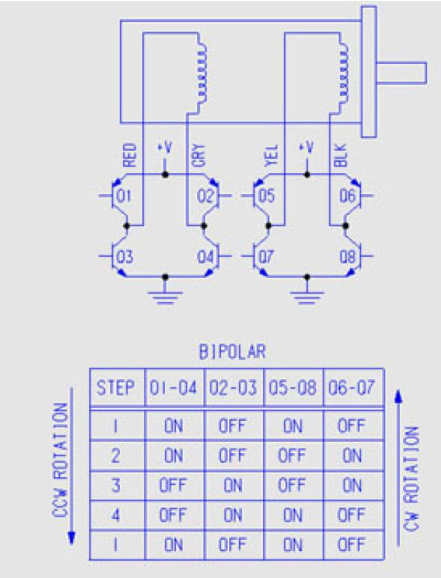
TECHNICAL SPECIFICATIONS

	UNIPOLAR		BIPOLAR	
Part Number	20M020D1U	20M020D2U	20M020D1B	20M020D2B
DC Operating Voltage	5	12	5	12
Resistance per Winding (ohms)	20	115.2	20	115.2
Inductance per Winding (mH)	3.9	20.3	7.8	52.8
Holding Torque* (mNm/oz-in)	7.77 / 1.10	7.77 / 1.10	11.30 / 1.60	11.30 / 1.60
Rotor Moment of Inertia ($g.m^2$)	4.1×10^{-5}	4.1×10^{-5}	4.1×10^{-5}	4.1×10^{-5}
Detent Torque (mNm/oz-in)	3.53 / 0.50	3.53 / 0.50	3.53 / 0.50	3.53 / 0.50
Step Angle	18°	18°	18°	18°
Step Angle Tolerance*	± 1.5°	± 1.5°	± 1.5°	± 1.5°
Steps per Revolution*	20	20	20	20
Max. Operating Temperature	100°C	100°C	100°C	100°C
Ambient Temperature Range				
Operating	-20°C to 70°C	-20°C to 70°C	-20°C to 70°C	-20°C to 70°C
Storage	-40°C to 85°C	-40°C to 85°C	-40°C to 85°C	-40°C to 85°C
Bearing Type	Sintered bronze sleeve	Sintered bronze sleeve	Sintered bronze sleeve	Sintered bronze sleeve
Insulation Resistance at 500Vdc	100 megohms	100 megohms	100 megohms	100 megohms
Dielectric Withstanding Voltage	450 ± 50 VRMS, 2 sec	450 ± 50 VRMS, 2 sec	450 ± 50 VRMS, 2 sec	450 ± 50 VRMS, 2 sec
Weight (g/oz)	23.5 / 0.83	23.5 / 0.83	23.5 / 0.83	23.5 / 0.83
Leadwires	28 AWG, UL Style 1429	28 AWG, UL Style 1429	28 AWG, UL Style 1429	28 AWG, UL Style 1429

* Measured with 2 phases energized

We see that this is a 5 V motor and that the winding resistance is 20 Ω. Thus 5 V pulses will be sufficient to drive the motor. We will be applying 5 V pulses across the windings. Note that if the winding resistance is 20 Ω, when we apply 5 V across the winding, the winding will draw 250 mA. This is more current than the Arduino digital outputs can supply, so we will need to use a driver integrated circuit as a buffer between the Arduino and the motor. This portion of the datasheet also lists the mechanical properties of the motor such as the inertia and torque. Since we are not driving a mechanical load with this motor, we will not consider these specifications in our design.

The fourth page of the datasheet gives the wiring diagram and the pulsing sequence:



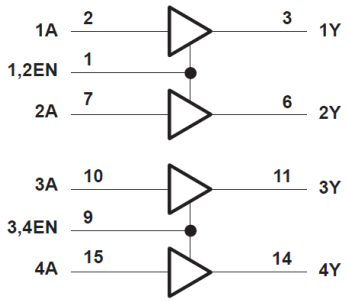
Note that when Q_1 is on, the RED wire will be connected to +V which is the positive supply, 5 V in this example. When Q_4 is on, the GRY wire will be connected to ground. In this configuration, 5 V is applied across the winding with the RED wire positive and the GRY wire negating. When Q2 is on, the GRY wire will be connected to +V. When Q3 is on, the RED wire will be connected to ground. In this new configuration, 5 V is applied across the winding with the GRY wire positive and the RED wire negating, Note that in these two configurations, the polarity of the voltage across the winding has changed.

Q1 through Q8 can be thought of as switches. Q3, Q4, Q7, Q8 are los-side switches that connect a motor winding to ground. Q1, Q2, Q5 and Q6 are high side switches and they connect a motor winding to the positive supply. Switches in series should never be on at the same time. For example, Q1 and Q3 should never be on at the same time. If they were, the combination of Q1 and Q3 would place a direct short across the 5 V power supply, and this could damage the driver. It is possible to turn on transistors that are across from each other at the same time. For example, Q1 and Q2 can be on at the same time as can Q7 and Q8. In both cases, turning on the pair forces the winding voltage to be zero. The winding voltage would be zero so the winding draws no current.

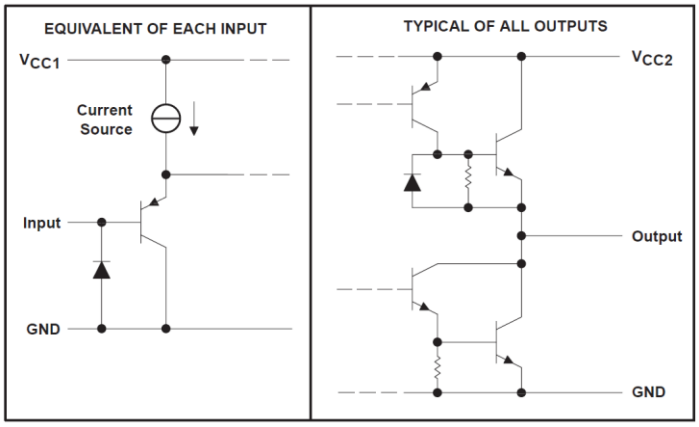
The transistor arrangement of Q1 through Q4 is called an H-bridge. It allows us to apply a +/- 5 V swing to a load from a single 5 V supply. When Q1 and Q4 are on, +5 V is applied across the motor winding. When Q2 and Q3 are on, -5 V is applied across the same winding. Note that Q1 and Q3 form a half bridge or Half-H bridge. Two half bridges form an H-bridge. An H-bridge is a common circuit topology used in power electronics.

We will not use discrete transistors for our H-bridge. Instead we will use the L293 quad Half-H driver integrated circuit. Schematically, the device is shown as four buffers:

logic diagram

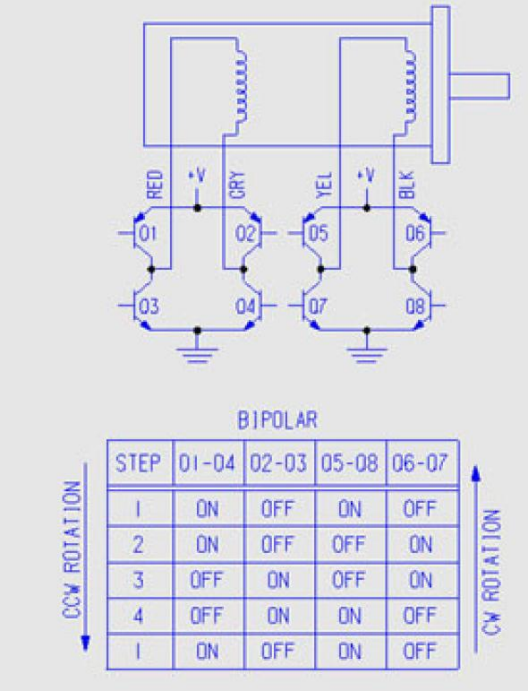


Each buffer is actually a high-current half-H. Note that each pair of drivers has an enable line (pins 1 and 9). Each half bridge is shown as a buffer, which means that the output voltage is equal to the input voltage. Half-bridges are actually current amplifiers, so the output current can be much higher than the input current. The input impedance of the buffers is large, so we can connect the Arduino output directly to the buffer's input. Note that each buffer output is a high-current half bridge and each input is a high impedance emitter-follower :



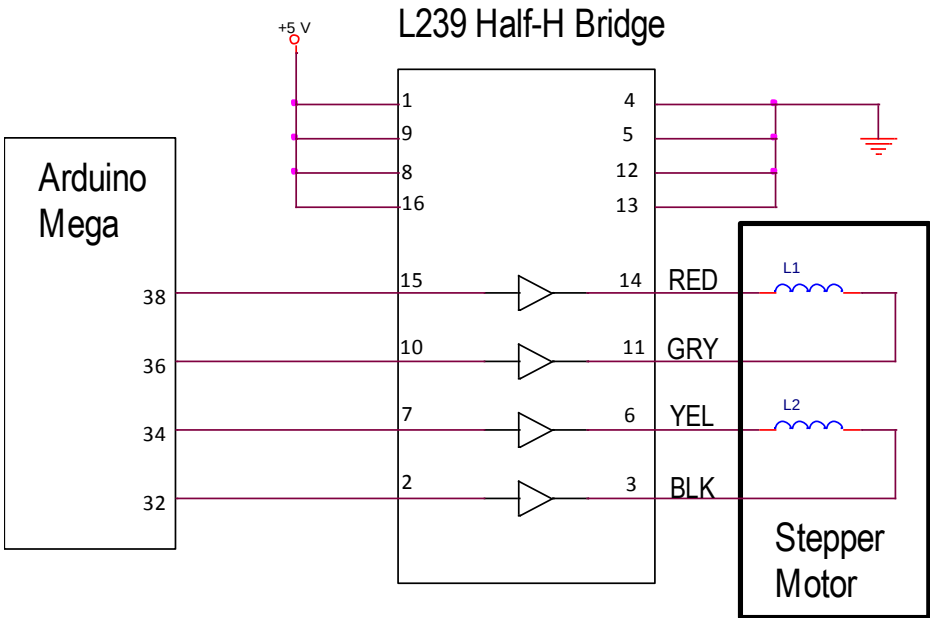
We do not really care how the buffer is implemented. (We see that it is more involved than just two switches.) All we need to know is that it is a high current output with a high impedance input. The complete data sheet for the L293 is shown on page 281.

Instead of using the table shown previously with switches listed, we can translate the transistors switches being "on" as a colored wire being high or low. The table is shown below:

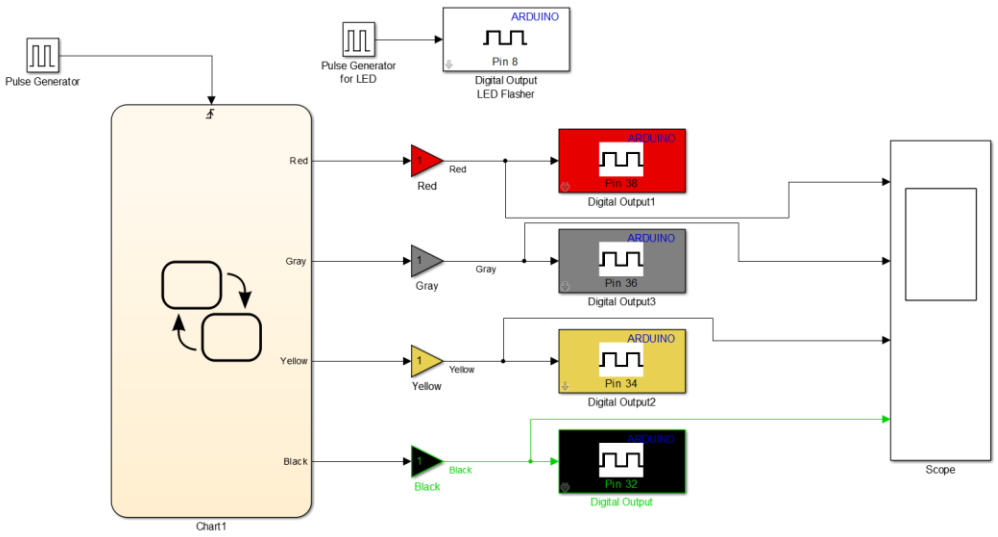


Step	RED	GRY	YEL	BLK
1	H	L	H	L
2	H	L	L	H
3	L	H	L	H
4	L	H	H	L
1	H	L	H	L

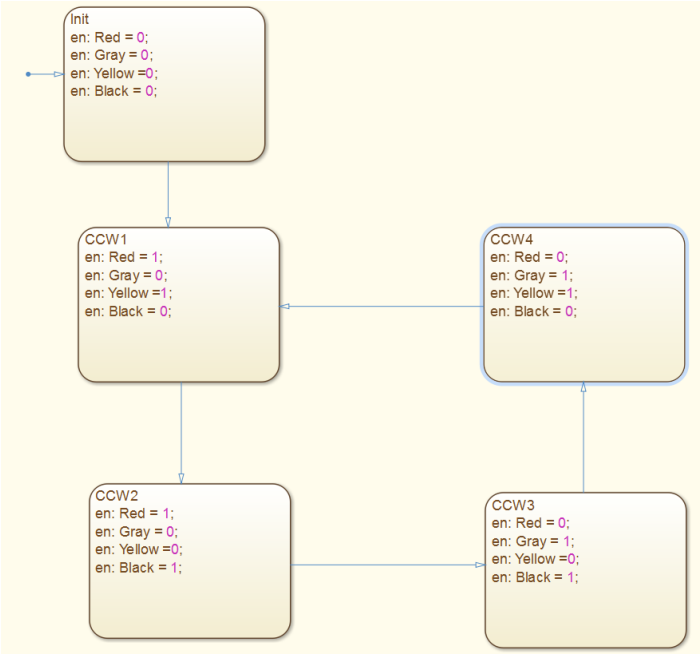
You will need to choose 4 digital outputs from the Arduino mega and select a buffer from the L293 for each wire on the stepper motor. A suggested schematic is shown below. The only required wires are the ones shown for the enable, power, and ground pins. You can rearrange the buffers if so desired:



As a first model, we will just generate the pulse sequence and spin the motor. A good method to generate the sequence is a state flow chart:

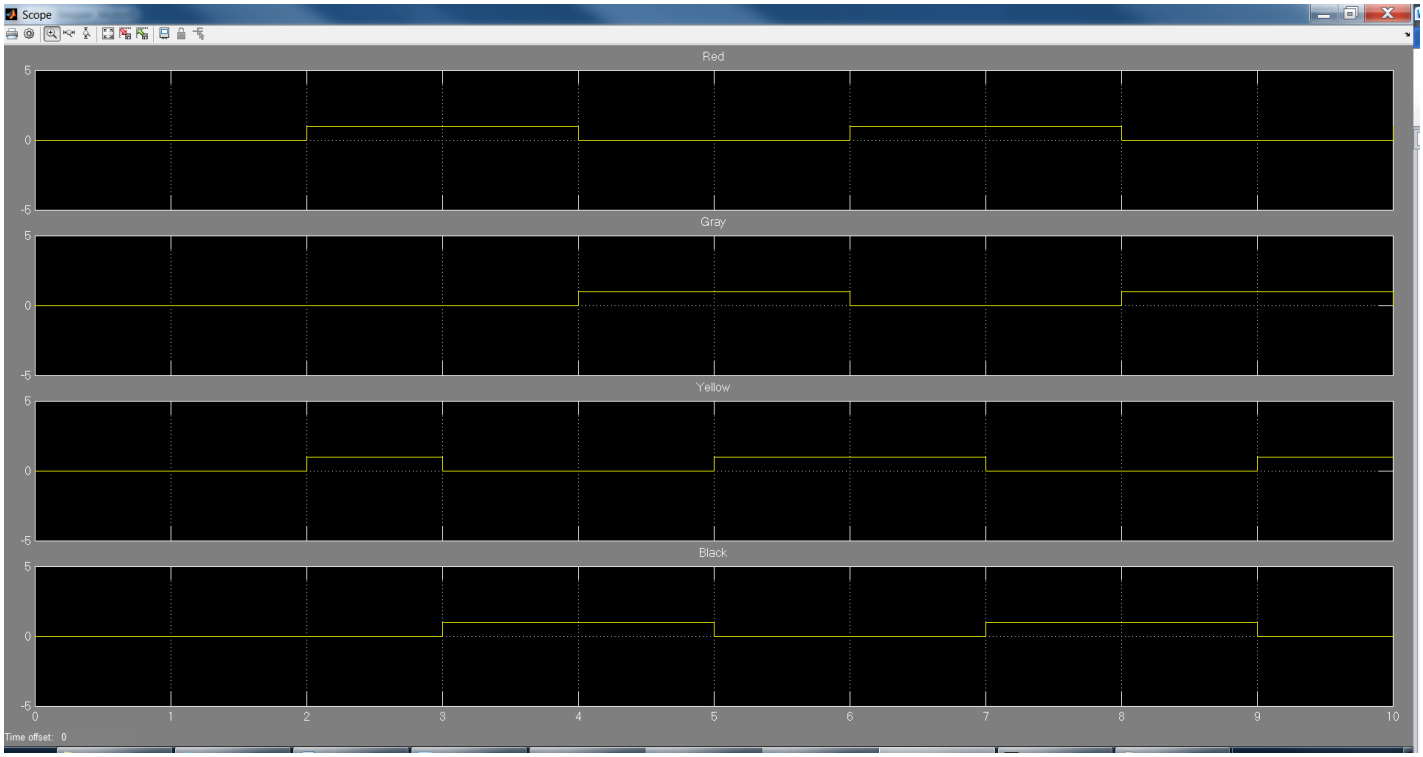


The contents of the Stateflow chart step the outputs through the counter-clockwise sequence of steps:



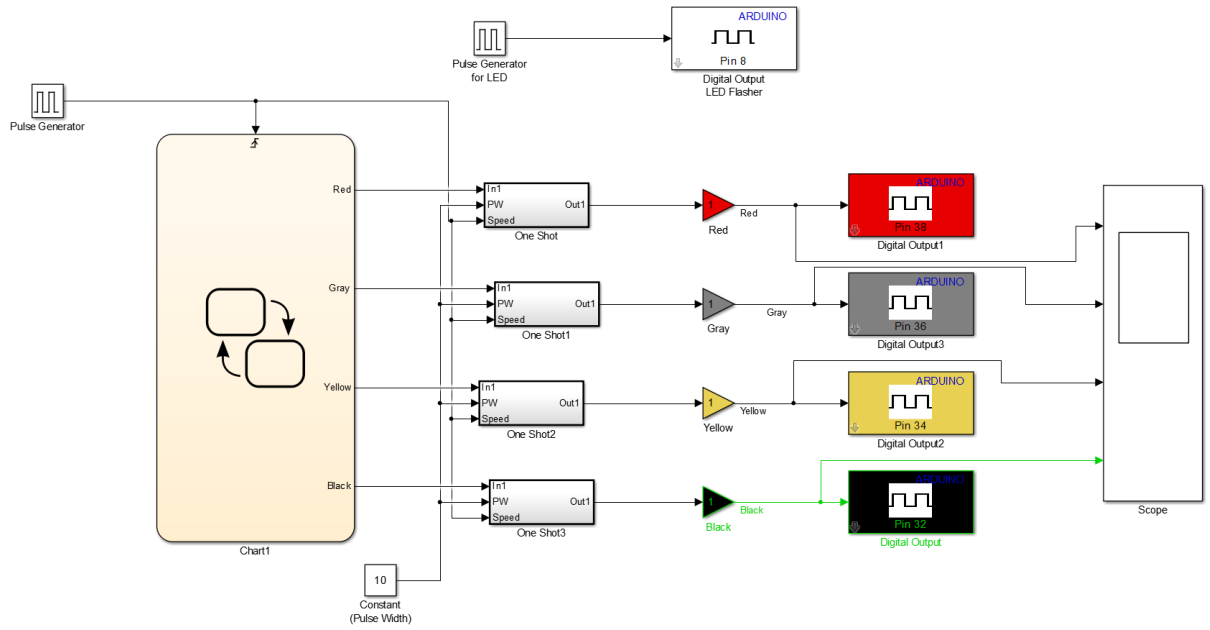
Notice that the sequence of states goes through the counter-clockwise sequence and that the arrows in the Stateflow chart point the state transitions in the counter-clockwise direction. This was done to make understanding the chart easier. Each time the chart transitions to a new state, the stepper motor will take a single step. The pulse generator in the model controls the speed of the motor. Every time the pulse generator generates a positive edge, the Stateflow chart will transition to a new state, and the stepper motor will take a step. Thus, the pulse generator frequency is also the step frequency of the motor. (This is not the rotation frequency as 20 steps are required for a single rotation with this motor.)

This mode has a problem which we can see when we run a simulation. The pulse outputs are shown below for the first 15 seconds. Note that the pulse generator frequency is 1 Hz so the motor will take one step per second:

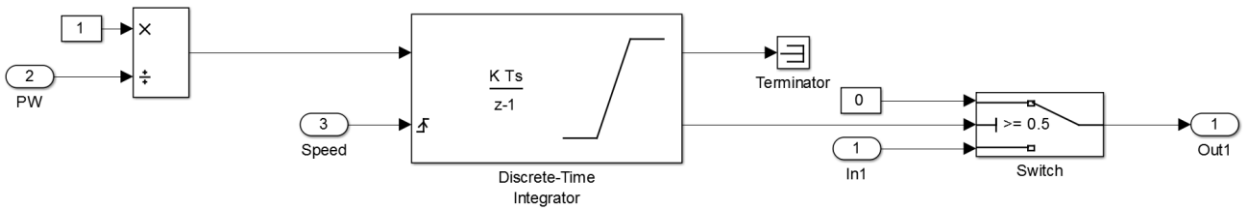


We see that for a slow rotation where the motor is taking one step per second, a pulse across the motor winding will last for 2 seconds. (Except or the first step where we started from the **Init** state in the Stateflow chart.) For a slower pulse generator frequency, the steps will be even longer. We are effectively placing a constant voltage across the motor windings. A motor winding looks like an inductor, which will look like a short and draw a large current for these long pulses. The current through the winding is limited only by the wire resistance of the winding. We saw from the stepper motor data sheet that this winding resistance was 20 Ω, so placing 5 V across the winding will draw 250 mA. Since both windings always have voltage applied, the motor will draw a total of 500 mA. This is a significant current that will heat up the voltage regulator on the Arduino Mega board. This is wasteful because when we pulse the motor, the motor will take a step in about 10 ms. (The value of 10 ms was found experimentally, and will vary for different motors.) Once the motor has moved in response to a step, the motor will not move again even though the voltage is still being applied. Thus, the pulse makes the motor move in the first 10 ms, and for the rest of the pulse, the voltage is applied to the motor winding and the winding draws current, but the motor does not move. To fix this problem we will generate pulse outputs that last for a short amount of time. That time is just enough to allow the motor take a step. This method reduces the power dissipation of the motor and allows the regulator on the Arduino board to run cooler.

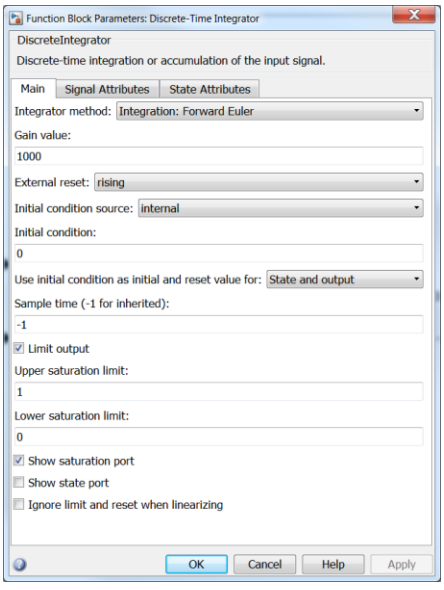
The model has been modified by adding a new subsystem called “One Shot” as shown below:



The contents of the One Shot subsystems are shown below:

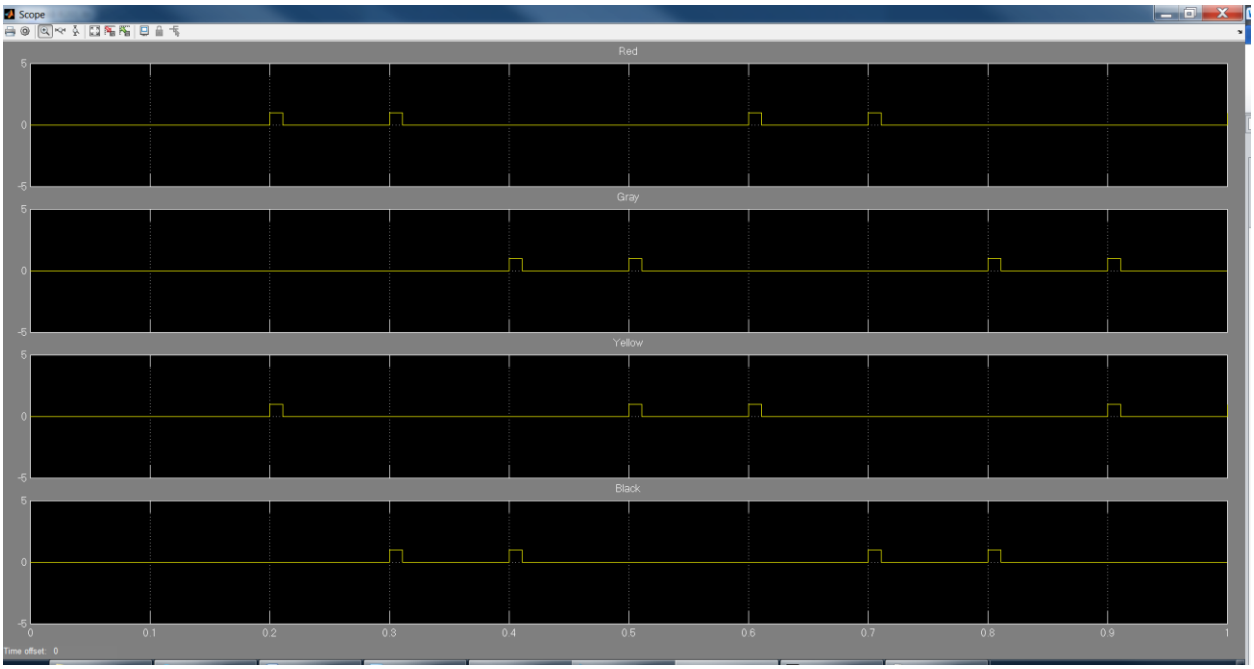


A positive edge on the **Speed** signal resets the integrator output to zero. This is connected to the pulse generator in the previous level, so the integrator is reset each step. This integrator integrates one divided by the pulse width. If you open the integrator block, you will see that the gain is set 1000, the saturation limit is set to 1, and we have enabled the saturation port:



Note that the integration output of the block is terminated and it is the saturation port that goes to the switch. The way the saturation port works is that when the integrator is not saturated, the saturation port output is zero. When the integrator is saturated, the saturation port output is one. The switch is used to invert the saturation port output.

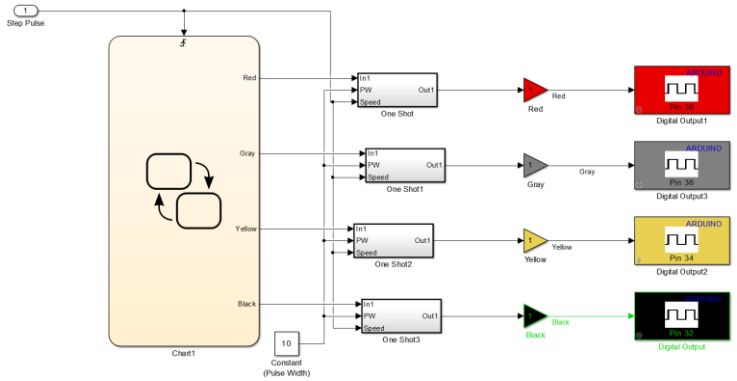
The integrator integrates 1000 divided by the pulse width. Note that the pulse width is provided by a constant in the top level of the model. While integrating, the output of the switch (and thus the output of the subsystem) is one. When integrator reaches one, the integrator will saturate and the switch output goes to zero. The integrator starts at zero and integrates the value of 1000/PW until it reaches a value of 1. The amount of time it takes to reach 1 is $\int_0^t \frac{1000}{PW} dt = 1$, or $\frac{1000}{PW} t = 1$, or $t = \frac{PW}{1000}$. Thus the width of the pulse emitted by this block is the value of signal PW in milliseconds. (If PW = 10, the pulse width will be 10 ms.) A simulation with the pulse width set to 10 ms and the pulse generator period set to 100 ms is shown below:



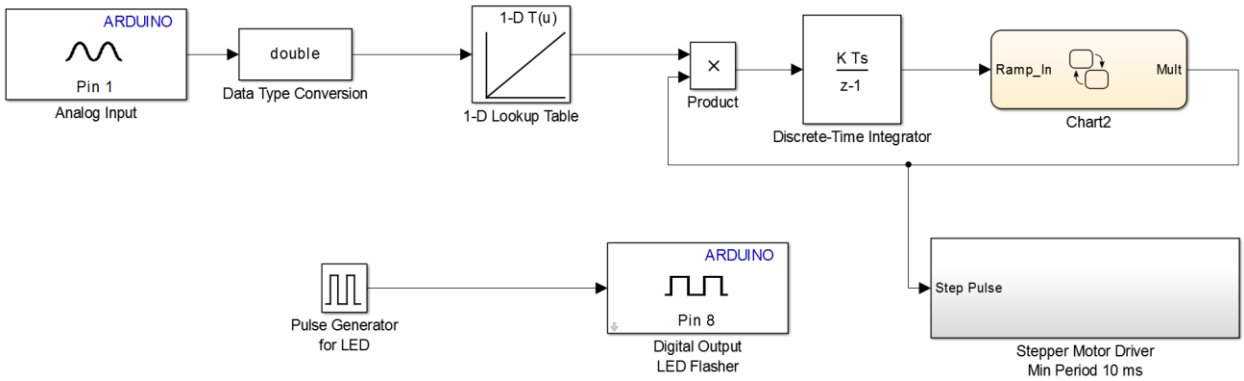
Note that if we change the pulse generator frequency, the pulse width will remain the same. Thus, we can run the motor fast or slow, and the pulse width to the motor will be constant at the specified pulse width.

Demo X.2: Show the working stepper motor. Use external mode so that you can vary the pulse frequency by changing the pulse generator frequency. Show that you can vary the pulse generator period from 10 ms to 1 s, and the motor responds appropriately.

Now that we have created a model that implements basic stepper motor operation, we can add logic to control the motor in interesting ways. Well… At least more complicated ways than in the previous example. First, we will create a subsystem out of the previous model that drives the stepper motor. The subsystem input will be an edge trigger that moves the motor one step counter-clockwise. The outputs of the subsystem are the digital outputs that drive the L293 stepper motor driver integrated circuit:



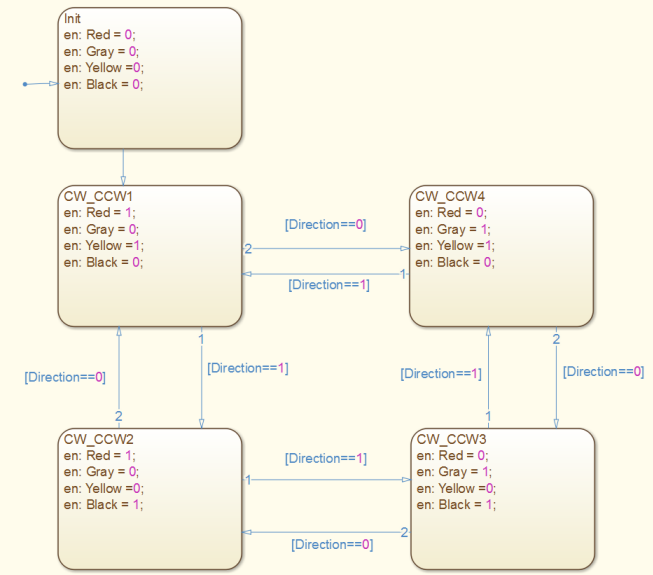
Now that we have this subsystem, we will create a model that drives the motor anywhere from 0 to 100 pulses per second (pps) based on a potentiometer connected to an analog input. Note that since it takes the motor 20 pulses to make one revolution, the motor speed will vary between 0 and 5 revolutions per second. The top level model is shown below:



The additional portions of the model are taken from earlier labs and will not be discussed here. You are encouraged to reuse earlier models and adapt it to example. Note that the output of the Stateflow chart is a variable frequency square wave proportional to the analog input.

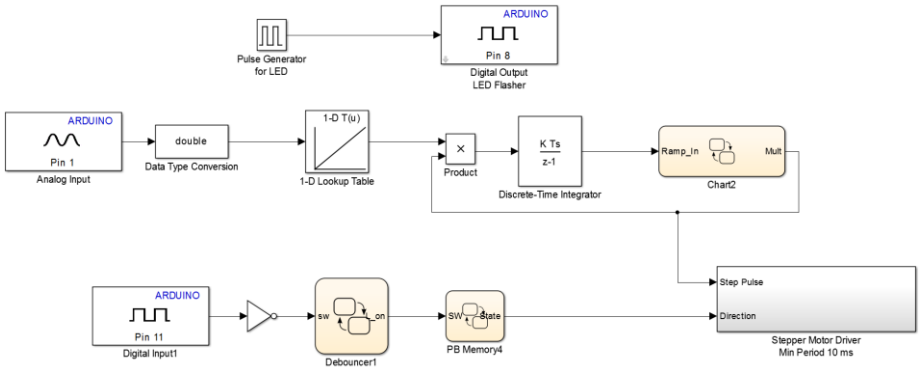
Demo X.3: Show the working stepper motor with a variable frequency between 0 and 5 revolutions per second. The speed should depend linearly on a potentiometer read with an analog input.

Now that we have a variable speed stepper motor controller, we would like to be able to change the direction of rotation. From the stepper motor data sheet and the stepper motor truth table shown on page 184, to change the direction of the stepper motor rotation all we need to do is reverse the order of the pulses. This is easily accomplished by modifying the Stateflow chart that controls the order of the pulses. We will add a new input to the Stateflow chart called “Direction.” This variable determines if the motor rotates clockwise or counter clockwise. The modified chart is shown below:



We see that if **Direction** equals one, the chart changes states in the counter clockwise direction. If **Direction** equals 0, the chart changes states in the clockwise direction. The way we set up the chart initially makes this change easy to manage and visualize.

The top level model is shown below:

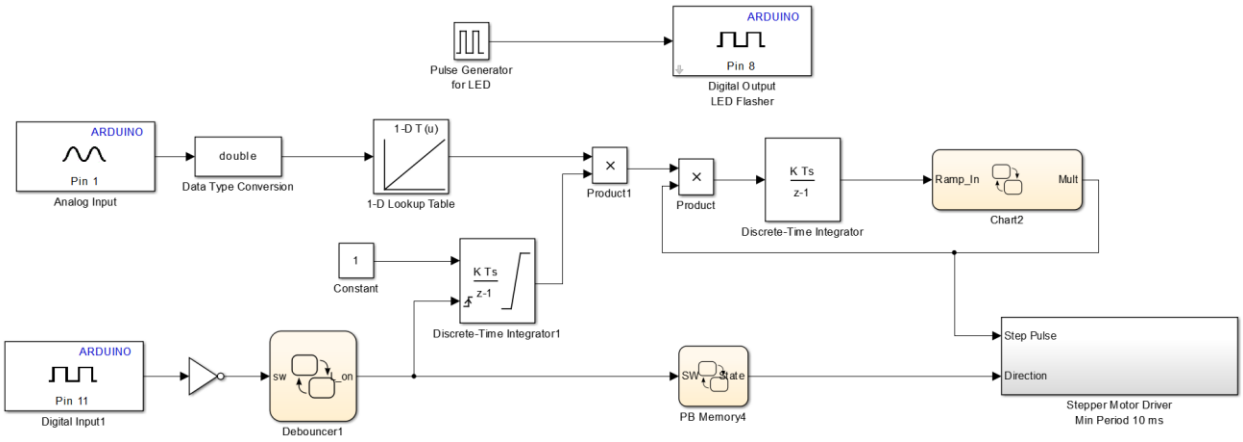


We have added logic to read a pushbutton with a digital input, debounce that input, and then add some memory to remember if the pushbutton has been pushed. The logic is designed so that when we push the button, the output changes from a zero to a one. When we release the pushbutton, the output remains a one. When we push the button again, the output changes from a one to a zero. When we release the pushbutton, the output remains a zero. This operation forces the user to push and release the pushbutton to make a change. Thus, Stateflow chart **PB Memory 4** outputs a zero or a one. The user must push and release the pushbutton before it can be pushed again. The output changes state when the pushbutton is pressed.

Demo X.4: Demo the stepper motor with variable speed and direction change. A pushbutton should change the motor direction. The variable speed control should work in any direction.

We notice that the direction change pushbutton does not always seem to work right. As a fix, whenever the direction pushbutton is pressed, we will set the speed to zero and then slowly allow the speed to return to the setting specified by the analog input. We will do this with an integrator. The integrator will be reset to zero

every time the pushbutton is pressed. The pushbutton will integrate from zero to one in one second, and then saturate at 1. The model is shown below:

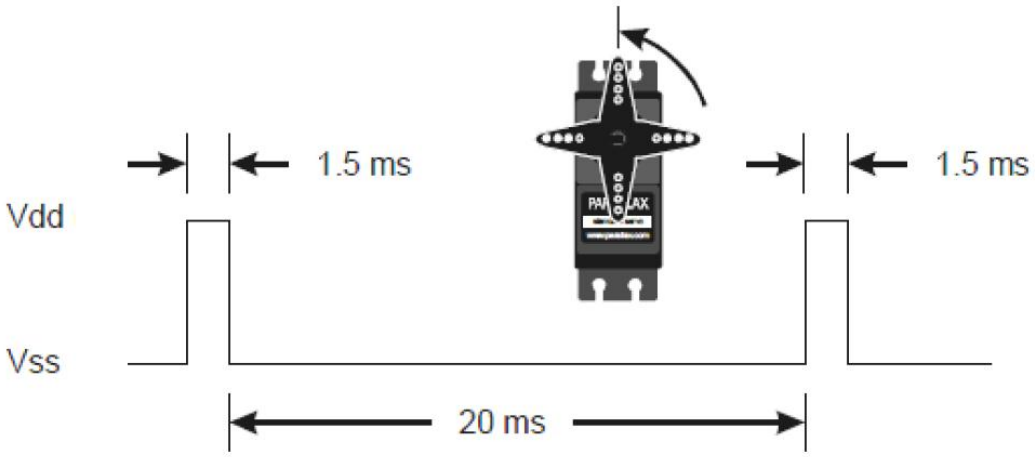


Demo X.5: Demo the stepper motor with variable speed, direction change, and speed ramping when the direction pushbutton is pressed. A pushbutton should change the motor direction. When the direction is changed, the speed should be set to zero and then ramp up linearly to the speed set by the analog input in one second. The variable speed control should work in any direction.

Exercise X.3: Add an LCD display to the stepper motor model. The LCD will display the direction (CW or CCW) and the speed in rpm. Example text would be "**`Direction: CCW  Speed: 193 rpm`**"

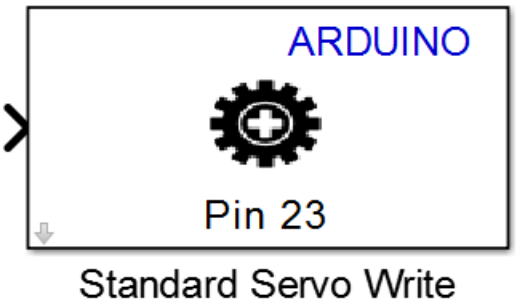
C. Servo Motor

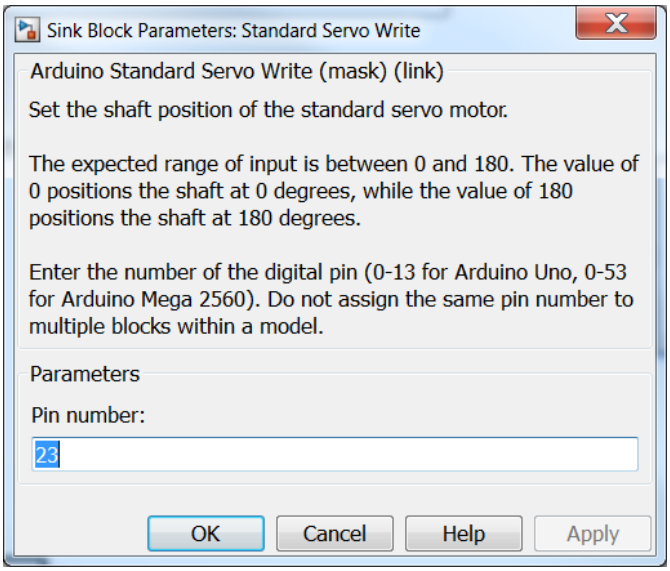
Servo motors are position control motors. The input signal commands the motor to turn to a specified angle, and the motor holds that position. If an external force is applied to the motor arm, the servo motor will apply an opposing torque to hold the arm at the same position. Servo motors are typically commanded by sending it a repeating pulse with a specified pulse width. For analog servo motors, such as we will use in this lab, the pulse frequency is typically 50 Hz corresponding to a period of 20 ms. For the motors we will use in this lab, a pulse width of 0.5 ms will position the motor arm at 0 degrees, a pulse width of 1.5 ms will position the arm at 90 degrees, and a pulse width of 2.5 ms will position the arm at 180 degrees. Different motors may have slightly different requirements on the pulse waveform. A typical waveform for a control pulse is shown below:



This type of waveform is referred to as a pulse-width modulation and was used to control the speed of our DC motor. Here, we have a PWM frequency of 50 Hz and small duty cycles ranging from 2.5% to 12.5%. For servo motors, the duty cycle controls the position.

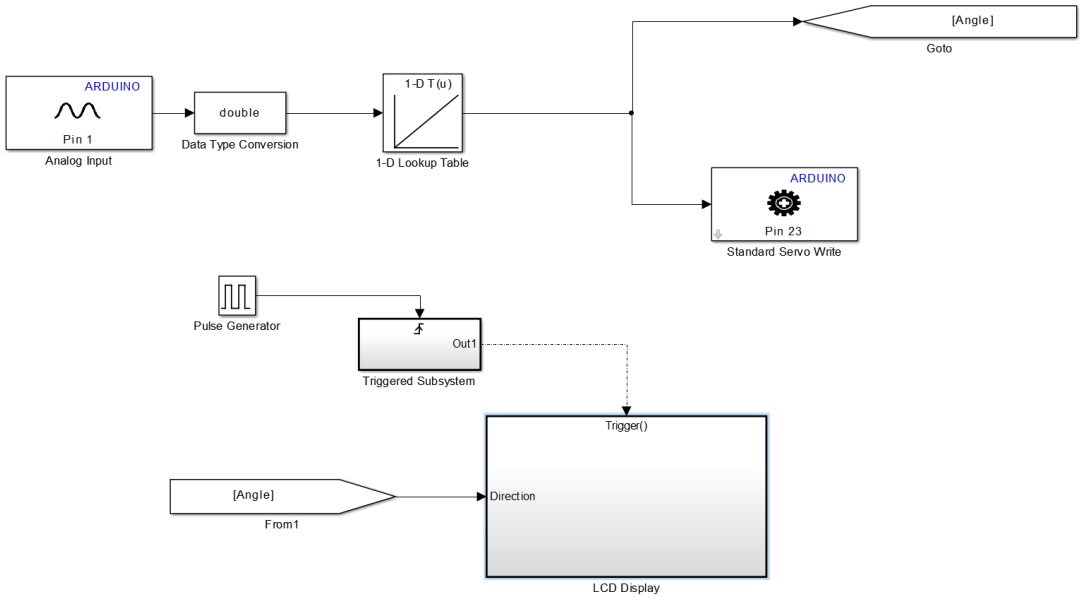
The Arduino toolbox in Simulink has a block for generating waveforms to drive a standard analog servo motor:





We specify a pin number for the output. The input to the block is a number between 0 and 180 specifying the angle. The block generates the appropriate waveform to position the motor.

For a first example, we will create the model below where the motor position is determined by a potentiometer read with an analog input. The lookup table is used to convert the values returned by the analog input block to a number between 0 and 180. We will use the LCD display to display the value of the angle:



For a typical servo motor, the brown or black wire is ground, the red wire is the positive power supply, and the third wire is the control signal.

Demo X.6: Show that you can control the motor position using the analog input and position the motor between 0 and 180 degrees. Display the value of the angle on the LCD display.

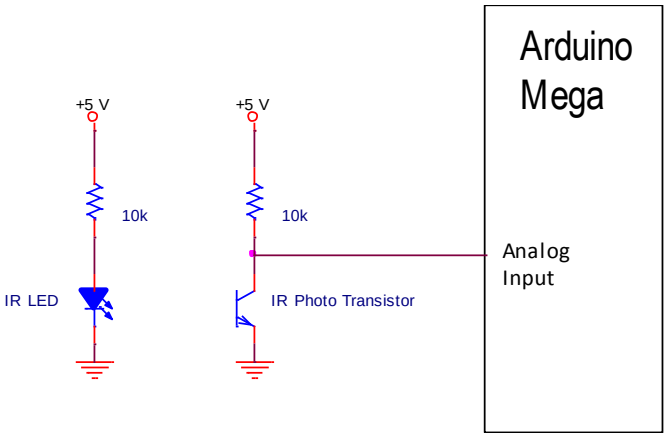
Now that we can control the position of the servo motor, we can make a bunch of interesting and fun controllers to make the motor do want we want it to do.

Exercise X.4: Create a model that slowly rotates the motor from 0 to 180 degrees (or whatever a safe maximum is for your hardware), and then slowly rotates it back to zero. The motor will continually rotate back and forth. The speed of rotation should be constant.

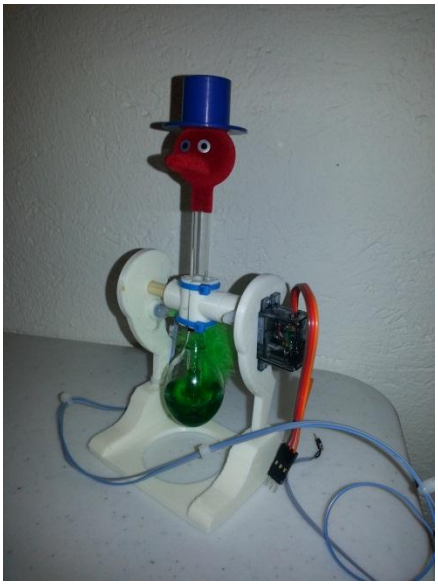
Exercise X.5: Create a model that rotates the motor from 0 to 180 degrees (or whatever a safe maximum is for your hardware), and then rotates it back to zero. The motor will continually rotate back and forth. The speed of rotation should be determined by an analog input. The speed to rotate between 0 and 180 degree should vary between 1 and 10 seconds as determined by an analog input. The speed to rotate back from 180 to 0 degrees should be the same as the speed to rotate from 0 to 180 degrees. If the analog input is constant, the speed should be constant.

Exercise X.6: Create a model that rotates the motor from 0 to 180 degrees (or whatever a safe maximum is for your hardware), and then rotates it back to zero. The motor will continually rotate back and forth. The speed of rotation should be determined by an analog input. The speed to rotate between 0 and 180 degree should vary between 1 and 10 seconds as determined by an analog input. The speed to rotate back from 180 to 0 degrees should be the same as the speed to rotate from 0 to 180 degrees. Make the speed follow a sine wave profile so that the speed increases in the middle of the rotation and slows near the ends of the rotation. (You can use a look up table for this, or a MATLAB embedded function, or a mathematical function.

Exercise X.7: Create the drinking bird model. The bird continually rotates back and forth. When rotating back, the position stops at zero degrees. When rotating forward, an analog input reads an infrared sensor. When the sensor is blocked by the bird the rotation stops. Make sure that you put a hard limit on the rotation in case your sensor does not work. While rotating, the speed of rotation should follow a sine wave to make the bird look “natural.” The overall speed of rotation should be variable and determined by an analog input. (But it should still follow a sine wave.) Verify that the sensor stops the rotation. An example circuit diagram is shown below.



For this sensor, the IR LED will point directly at the IR photo transistor. When the IR radiation is not blocked, the transistor will be on and the sensor voltage will go down. When the bird rotates such that the sensor is blocked, the voltage will increase. You will need to read and calibrate the sensor to determine when the sensor is blocked and not blocked. The drinking bird is shown below:



A video of the drinking bird operation is shown at http://wiki.ece.rose-hulman.edu/herniter/images/4/4e/Drinking_Bird_Video.mp4.

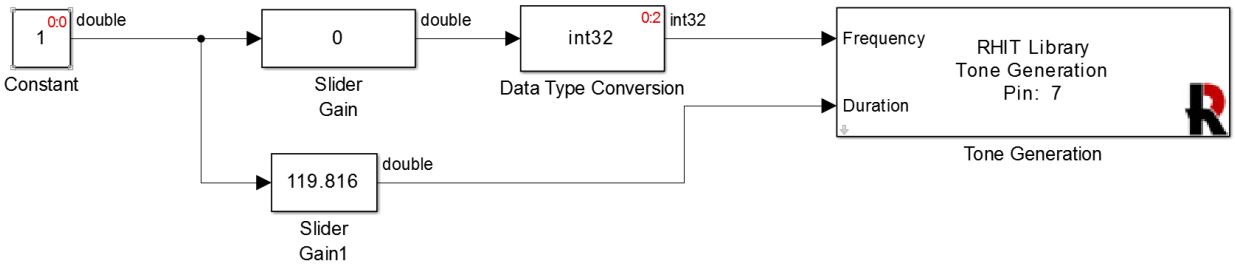
Lab XI
Tone generator

One of the strengths of using the Arduino is that a large number of functions have been implemented by other people and then the code is made public for us to use. In Lab XV we show how to import code and include it in a model using the S-Function Builder. The Tone generator function is just such an application except that we have already ported the code to Simulink and placed a block in the RHIT Arduino library. For this lab, we will just use the block. Keep in mind that this block was generated using the S-Function builder covered in Lab XV. After you have completed Lab XV, you can look at the Tone generator block as see how it was created.

We will test the tone generator using a piezo speaker. A typical Arduino digital output can directly drive a Piezo speaker, thus you can hook one end of the piezo speaker to the Arduino digital output and the other end to ground. In most cases, this will generate enough noise to annoy the other students in the lab. However, if you would like more noise, you can use a regular speaker and use the motor driver circuit shown on page 179 and replace the motor load by a speaker. For this lab, we will connect an Arduino digital output directly to the piezo speaker.

A. Playing Simple Tones

The first model we will create will introduce us to the Tone Generator Block is shown below:



The default pin for the Tone Generator block is digital pin 7. You can select a different pin by double-clicking on the block. The frequency input to the block specifies the frequency of the output tone in Hertz. Note that data type of this signal must be int32. The duration of the tone is how long the tone will last in milliseconds.

We will use **Slider Gain** blocks to vary the Duration input from 0 to 1000 (which will generate tones from 0 to 1000 ms) and the Frequency input from 0 to 20,000 Hz[8]. Set the solver type to discrete and set the fixed step size to 1 second. This will set the model (and therefore the Tone Generator block) to be executed once a second. Double-click on the sliders to change their limits. Run the model in External mode and demonstrate the operation of the Tone Generator.

[8] I will take your word for it that it emits tones all the way up to 20 kHz, and my ears have a limited frequency response.

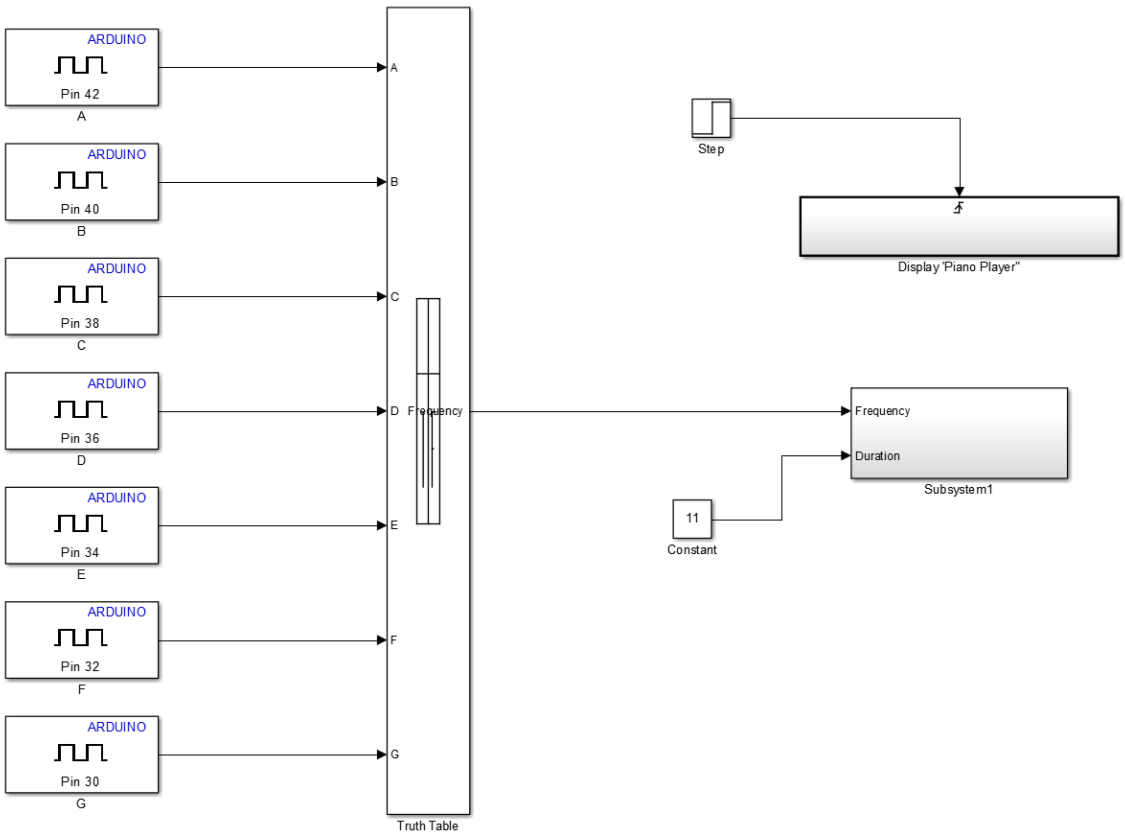
Demo XI.1: Demonstrate the operation of Tone generator. Show the effects of varying the frequency with a constant duration, and varying the duration with a constant frequency. Determine the settings required to generate a continuous sound output. Determine the settings required to obtain no sound output.

Exercise XI.1: Create a model that has continuous sound output and the frequency varies between 200 and 1200 Hz in a sinusoidal pattern at a 1 Hz rate. The change in frequency should sound as if it is continuous and the listener should not notice that the frequency is changing in small steps.

Exercise XI.2: Add a pushbutton to the model of Exercise XI.1 that enables or disables the sound output. (Other people in the lab will greatly appreciate this.) At startup, sound is disabled. When the push-button is pressed the first time, sound turns on. When the push-button is released, the sound remains on. When the push-button is pressed the again, sound turns off. When the push-button is released, the sound remains off. Further use of the push-button will toggle the sound on and off.

B. Piano Player

As a second example of what we can do with the tone generator, we will create the piano player shown below:

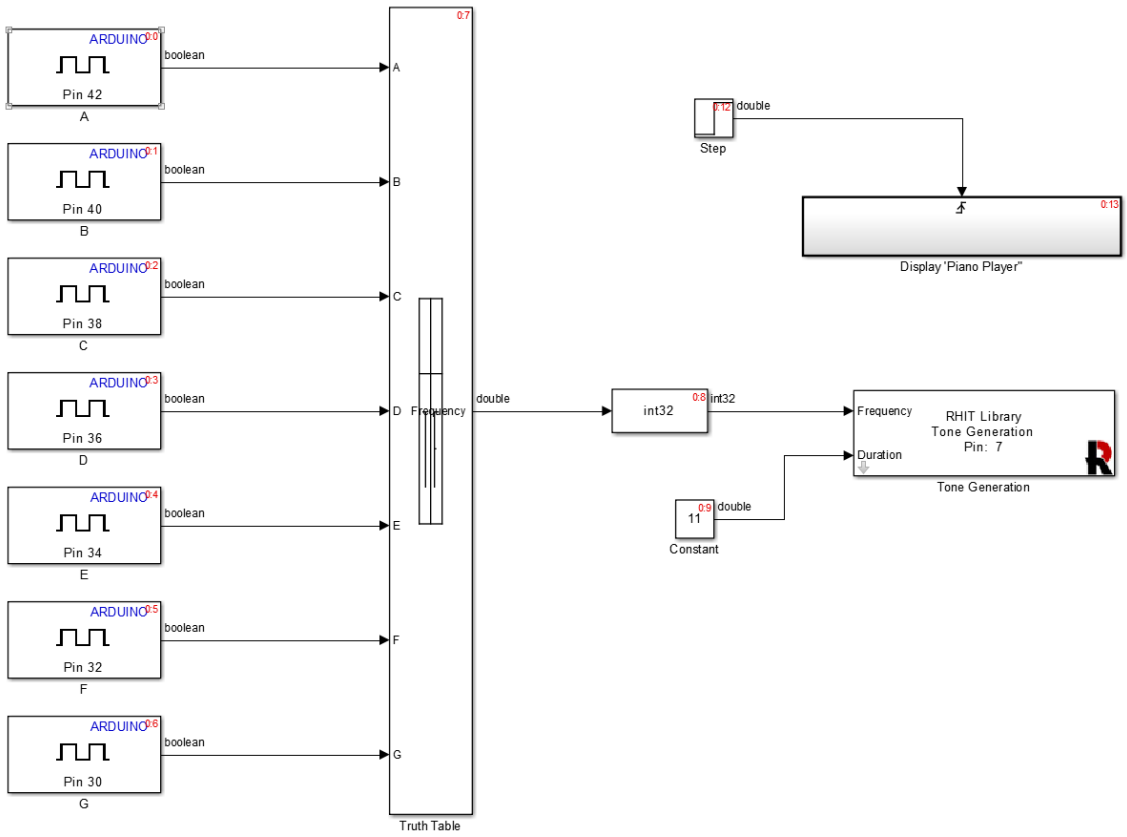


The piano has the following characteristics:

1. At start-up, the LCD displays the text “Piano Player.” This text message is sent to the LCD screen only once.
2. The piano has 8 keys which you will implement with 8 separate push-button switches. The keys select the notes A, B, C, D, E, F, and G. The frequency for A is 440 Hz. You can find the rest of the notes on-line.

3. Pressing one button causes the selected note to play.
4. Pressing more than one key causes an error sound of 220 Hz to be played.
5. Pressing no keys causes no sound to be generated.
6. A note is played as long as the key is held down. The sound is terminated as soon as the key is released.

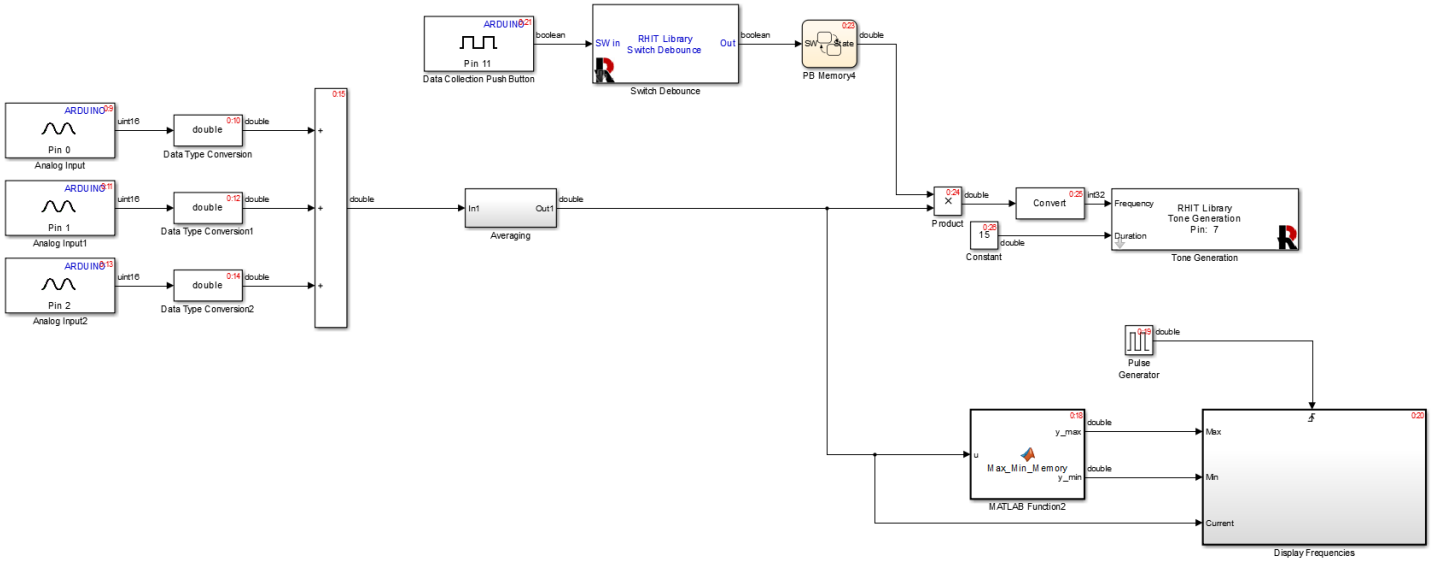
You can implement the frequency output using a truth table or the equation [6] $f(n) = 2^{\left(\frac{n-49}{12}\right)} \times 440\ Hz$. An example model is shown below:



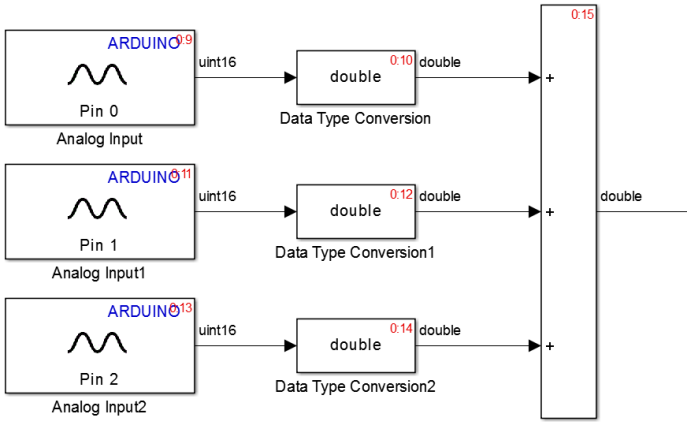
Exercise XI.3: Add two pushbuttons to the piano player of Demo XI.1. One push-button should be physically mounted to the left of the 8 pushbuttons used for the piano keys. The other push-button should be physically mounted to the right of the 8 pushbuttons used for the piano keys. When neither of the two new pushbuttons is pressed, the piano plays the same notes as in Demo XI.1. When both of the new pushbuttons are pressed, the piano plays the same notes as in Demo XI.1. When the left-side pushbutton is pressed, the eight piano keys will play the notes A through G, but an octave lower than in Demo XI.1. When the right-side pushbutton is pressed, the eight piano keys will play the notes A through G, but an octave higher than in Demo XI.1.

C. Accelerometer Whistler

In the next project, we would like to generate a tone based on the values output by an accelerometer. Available from several on-line vendors are analog accelerometers mounted on a breakout board. These boards are usually powered by a 3.3 V supply (available on the Arduino) and have three analog outputs. We will read these three outputs and generate a tone based on the output of all three channels. The sound emitted is up to you, but the sound should change based on the orientation of the board and how fast the board is shaken, and be sensitive to all three axes (z, y, and z). The complete model is shown below:

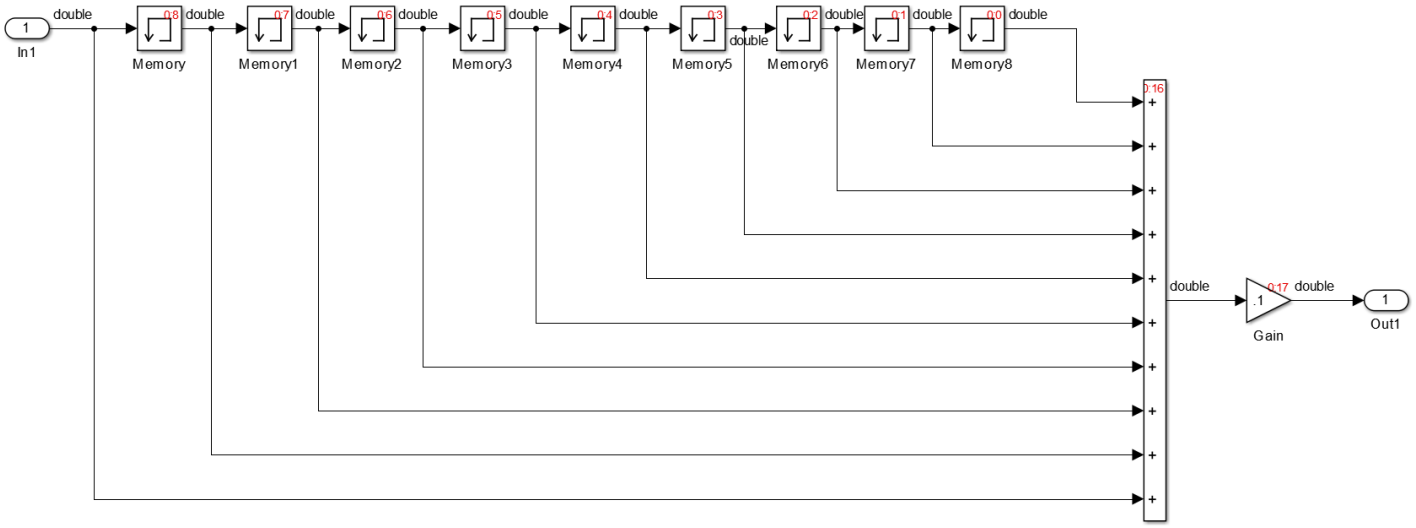


Here we just show a sum to create the frequency signal:



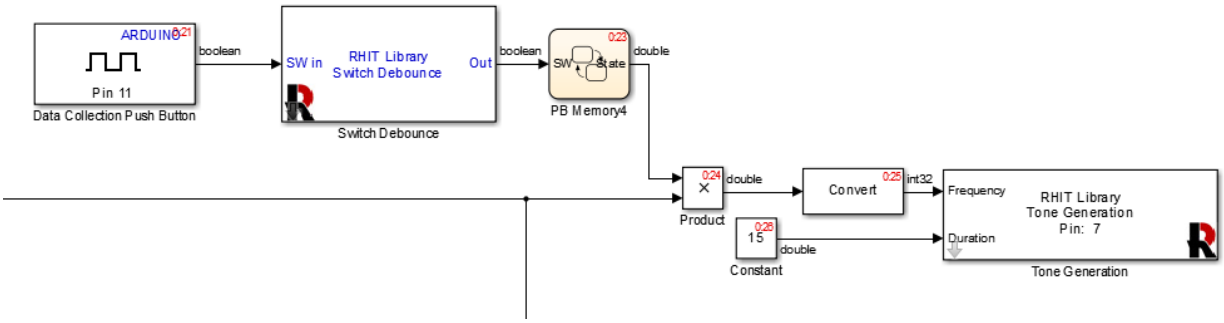
You may have to add some scaling (a gain block or look-up table) to the frequency signal to tune the sound to your liking. Here we just us the straight sum of the three outputs.

Because the output of the accelerator changes so rapidly, we created a time average block that averages the last 10 samples:



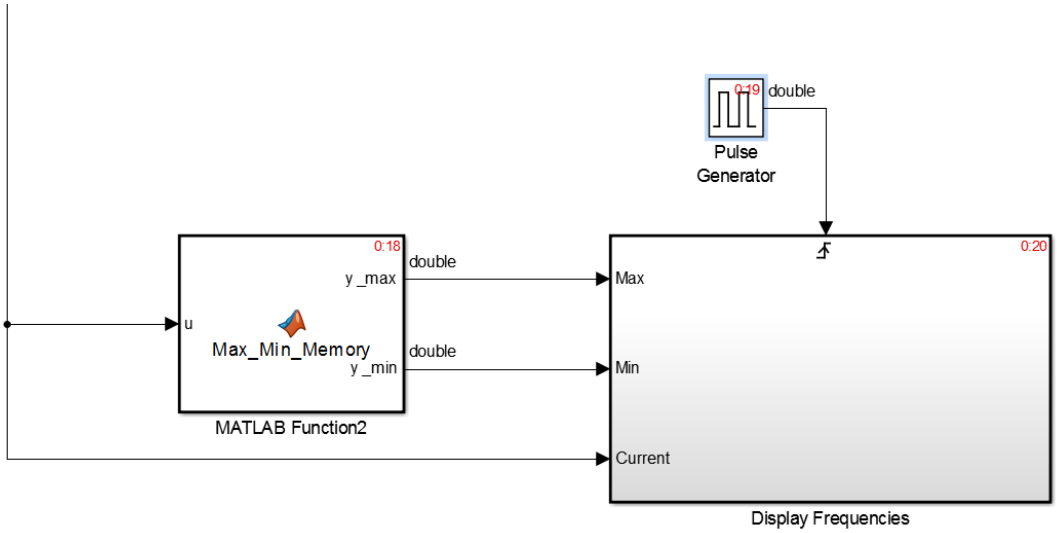
Note that the fixed time step that we are using is 0.01 seconds. With an average of 10 samples, the frequency output will change “slowly” over the time average of 100 ms.

As this project will annoy other people in the lab, we will use a push-button to turn on and off the sound:



You can use the same method that we used in Exercise XI.2 to silence the sound.

As a debussing tool, we will display the max frequency, min frequency, and the current frequency on the LCD display. This display should update every 300 ms:

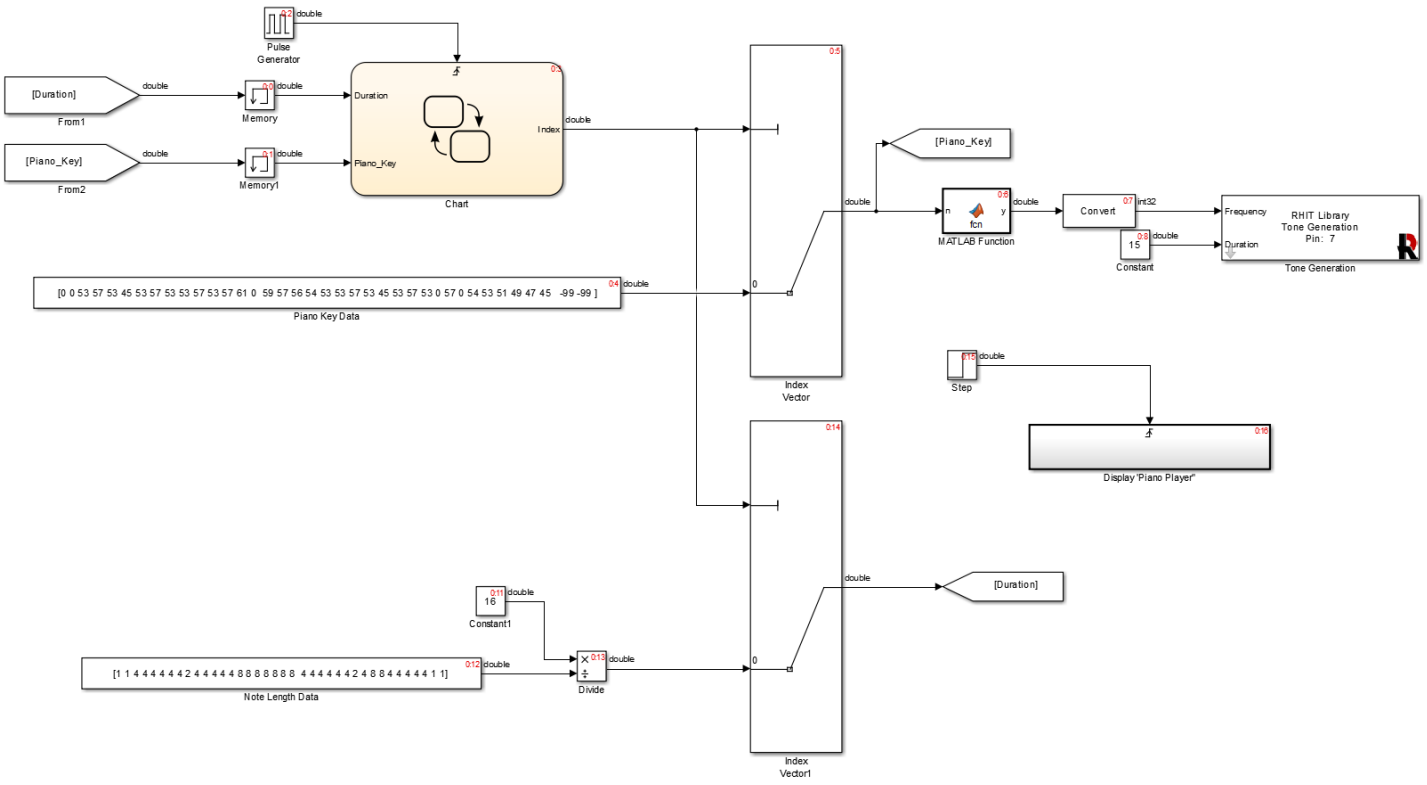


You can use the Max_Min_Memory block developed in Section VII.B on page 128.

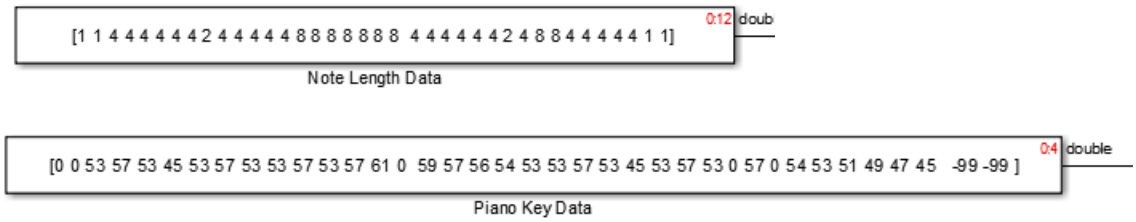
Demo XI.2: Demonstrate the operation of the accelerometer whistler.

D. Music Player

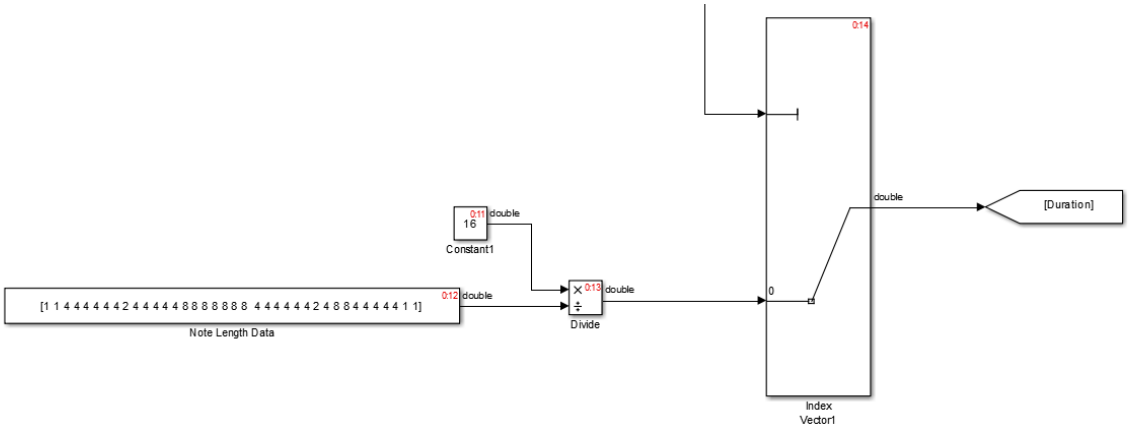
As a last demonstration of the Tone Generator, we will create a music player that plays the Jeopardy theme song. [7] The entire model is shown below:



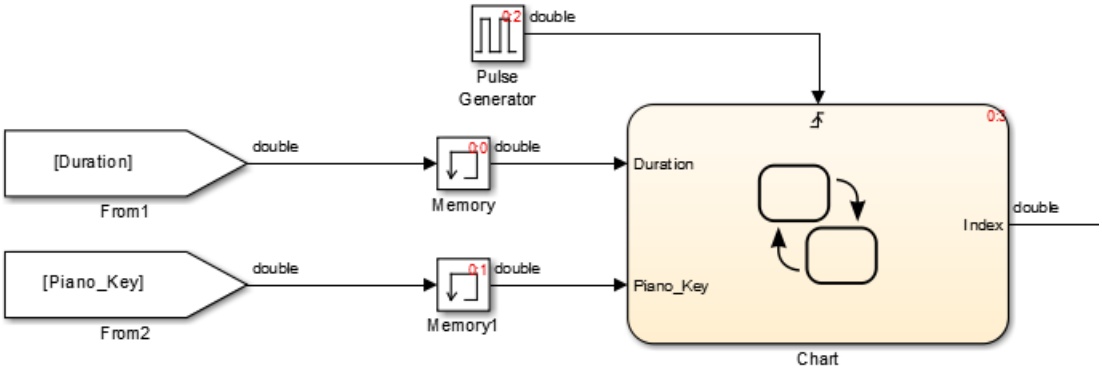
The model contains two arrays of data:



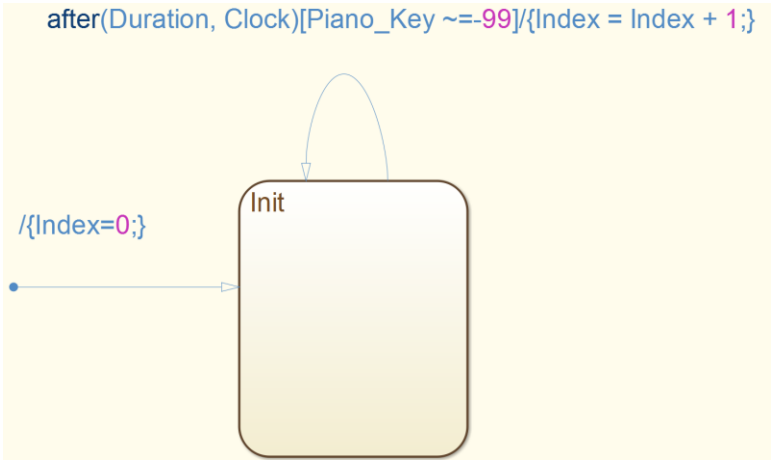
Both of these are constant blocks that that use a row vector rather than a single value. Individual values within the array are accessed using the Index Vector block.



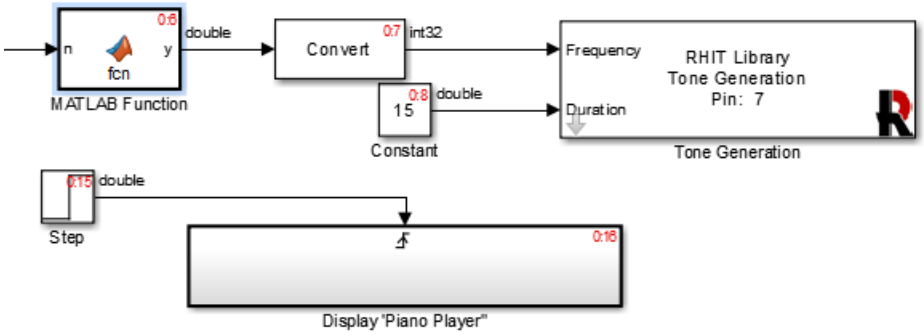
This block outputs a single value from the array based on the index input. The index is in this case is zero based, meaning the first element has an index of 0. The index is generated with a Stateflow chart:



The Stateflow chart starts at 0 and counts up. The speed at which it counts up is based on the duration of the note being played. The count holds when the piano key is equal to -99. The Stateflow chart is shown below:



The notes are decoded into frequencies using the equation [6] $f(n) = 2^{\left(\frac{n-49}{12}\right)} \times 440\ Hz$. The equation is realized using a MATLAB function block:



Finally, the text “Piano Player” is displayed on the LCD screen. This message is displayed only once when the model starts.

Demo XI.3: Demonstrate the operation of the music player with the Jeopardy theme song.

Exercise XI.4: Add a push button switch to the music player of Demo XI.3. If the music is playing, the push button has no affect. If the music player has completed the song, depressing the push-button causes the music player to immediately start playing the music. The music will continue until the entire theme is played.

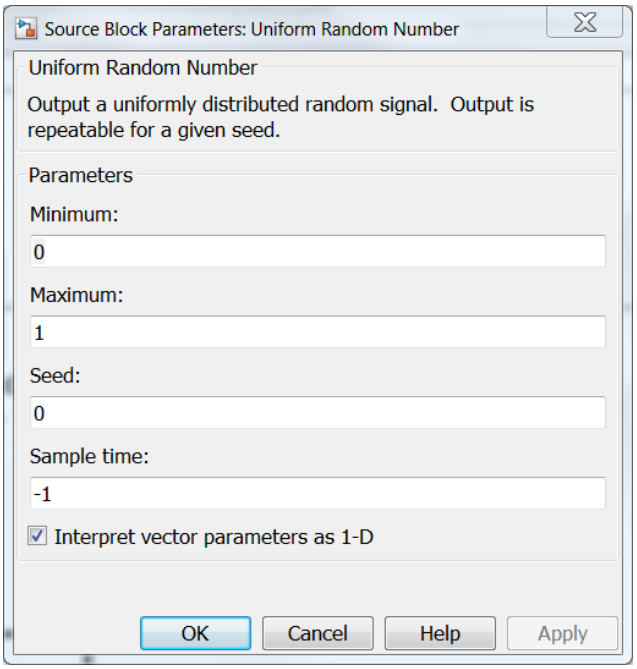
Exercise XI.5: Add a second push button switch to the music player of Exercise XI.4. When the push-button is pressed once, the music holds and the output is silent. When the pushbutton is pressed again, the music resumes where it left off.

Lab XII
Dice Game Project

For this lab, we would like to simulate rolling of a single die. We want to simulate how a die behaves as you roll it, so it will start out rolling fast, slow down, and then eventually stop on a random number between 1 and 6. There are several things that are random. The length of time the die spends rolling, the numbers that you see on the die as it rolls, and the value it displays as it comes to rest. We will simulate this with a tone generator and a 7 segment LED display. When the die is rolling fast, we will have the tone generator emit short high frequency tones. As the die slows down, the length of the tone will get longer and the frequency of the tone will be lower. As the die is rolling, changing random values will be shown on the LED. The length of time the value is shown on the LED is the same as the length of the tone.

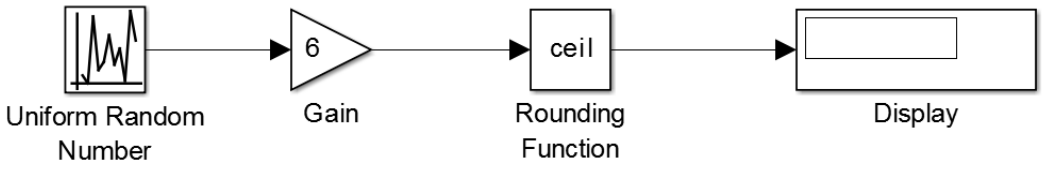
A. Random Numbers

This lab requires that we generate random numbers between 1 and 6. This can be done in two ways in Simulink. One way is to use the **Uniform Random Number** source in the **Simulink / Sources** library. The settings for the Uniform Random Number generator are:



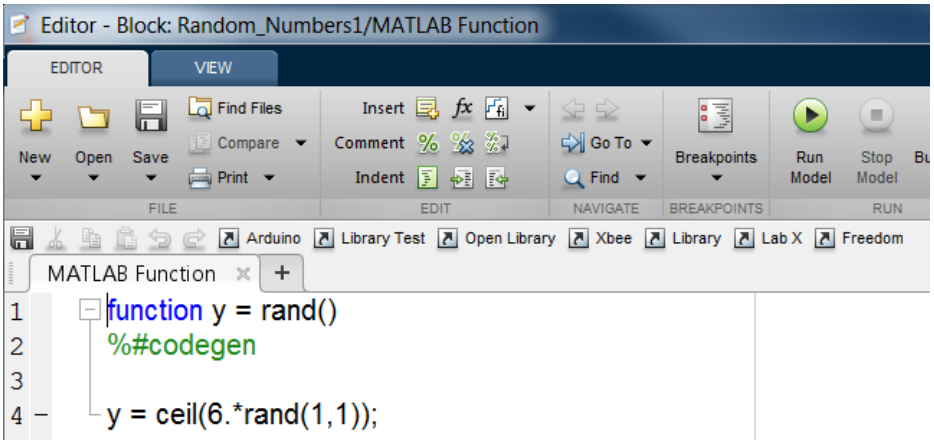
It is setup to generate a random number between 0 and 1. Note that there is a **Seed**, which we shall discuss later.

The generated random number is a real number between 0 and 1. To convert the number to an integer between 1 and 6, we multiply the random number by 6 and then round the number up using the ceiling function:

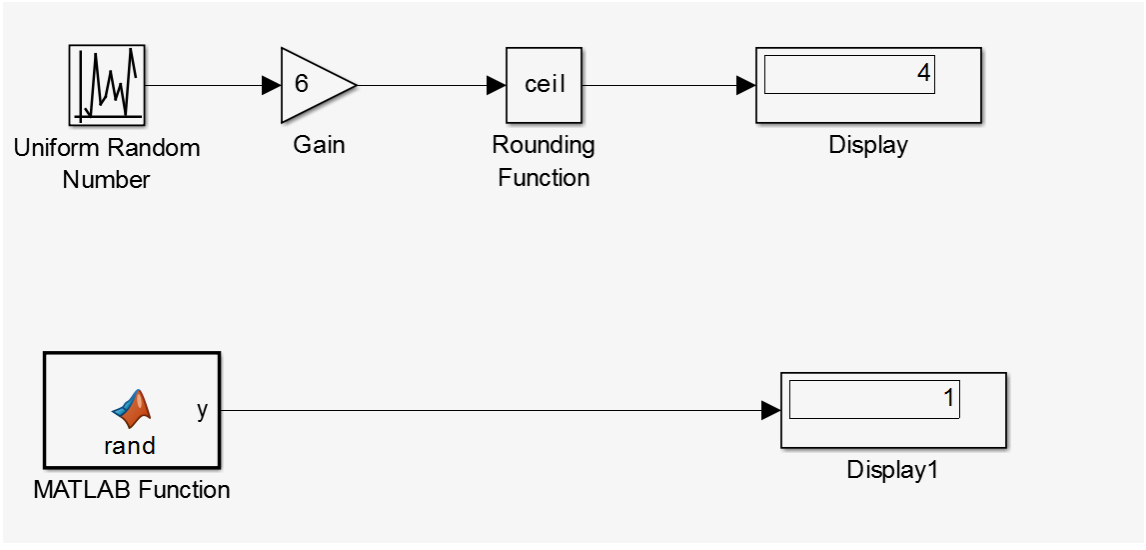


A second want to generate a random number is to use the MATLAB rand function. The MATLAB command **`rand(r,c)`** generates a r by c matrix and fills the matrix with uniformly distributed random numbers between 0 and 1. To duplicate the random number generated with the Simulink model shown above, we can use the command **`ceil(6.*rand(1,1))`**. In Sumulink, we can place this code inside a MATLAB Function block:

The MATLAB function is shown below:

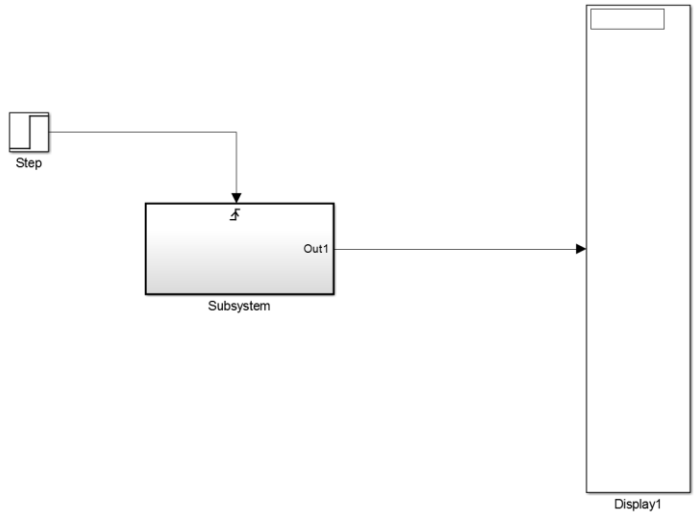


We will now run the model on an Arduino mega in external mode using the discrete solver and a fixed time step of 1 second:

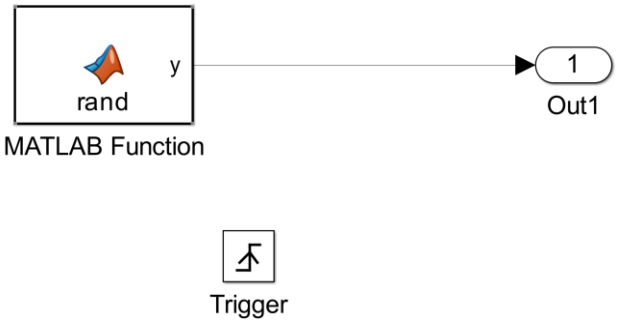


The model should display random integers between 1 and 6. We can use either of these methods to generate our random numbers.

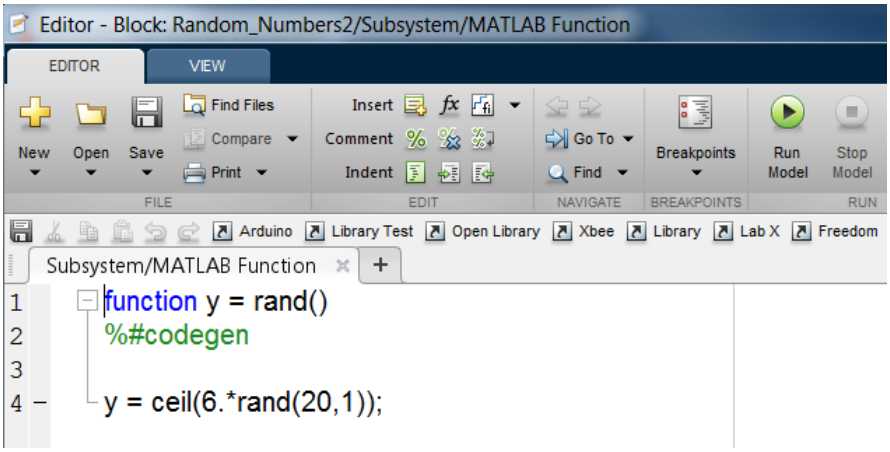
As a second example, we will use the model below to generate 20 random numbers using the MATLAB Function block. We will use the triggered subsystem below so that the model runs only once:



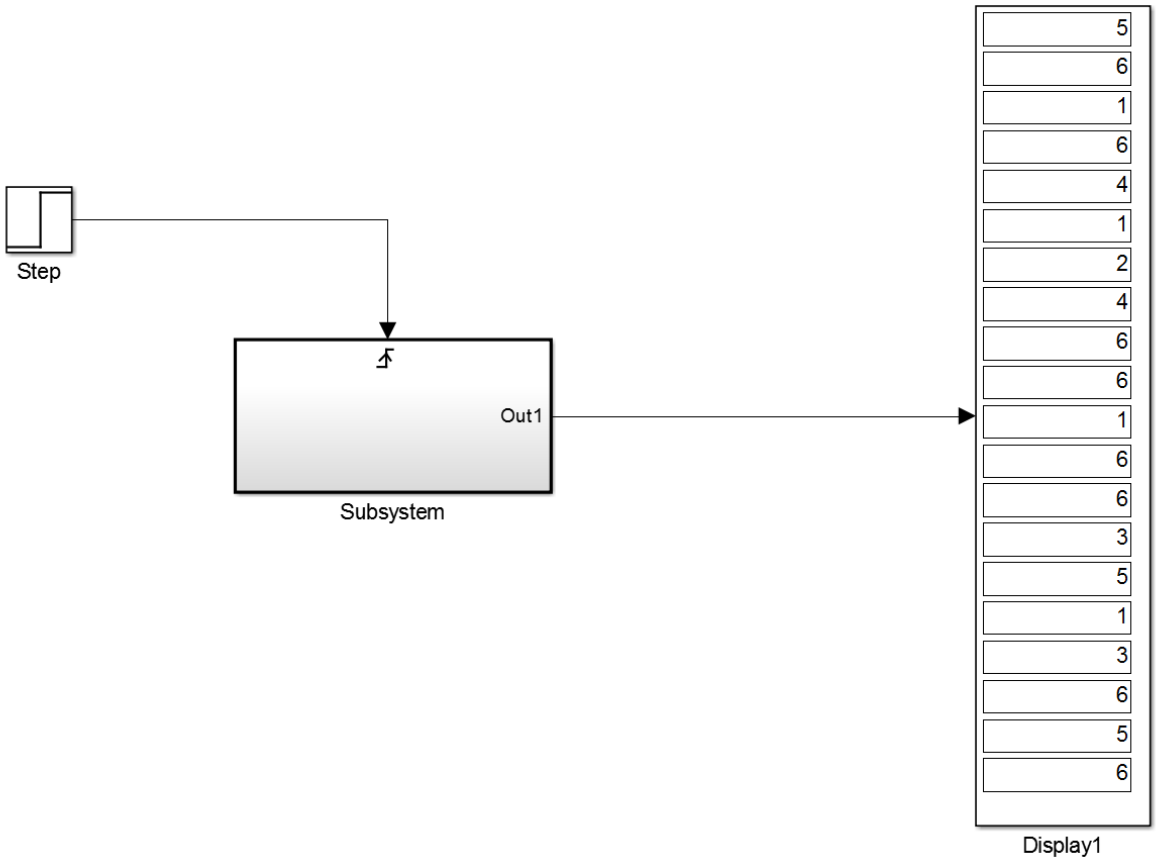
The triggered subsystem contains a random number generator MATLAB Function similar to the one we just covered:



The MATLAB function is slightly different than before in that it generates a column vector of 20 random numbers:

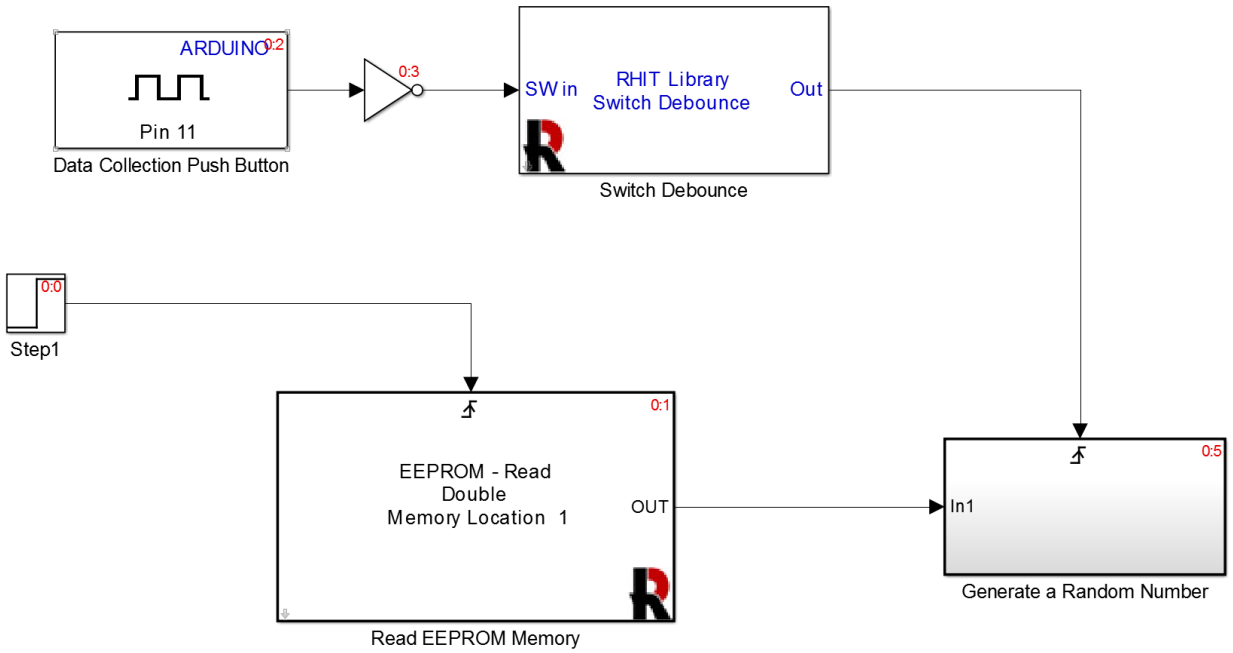


When you run the model in external mode, you will see the display fill with 20 random numbers:

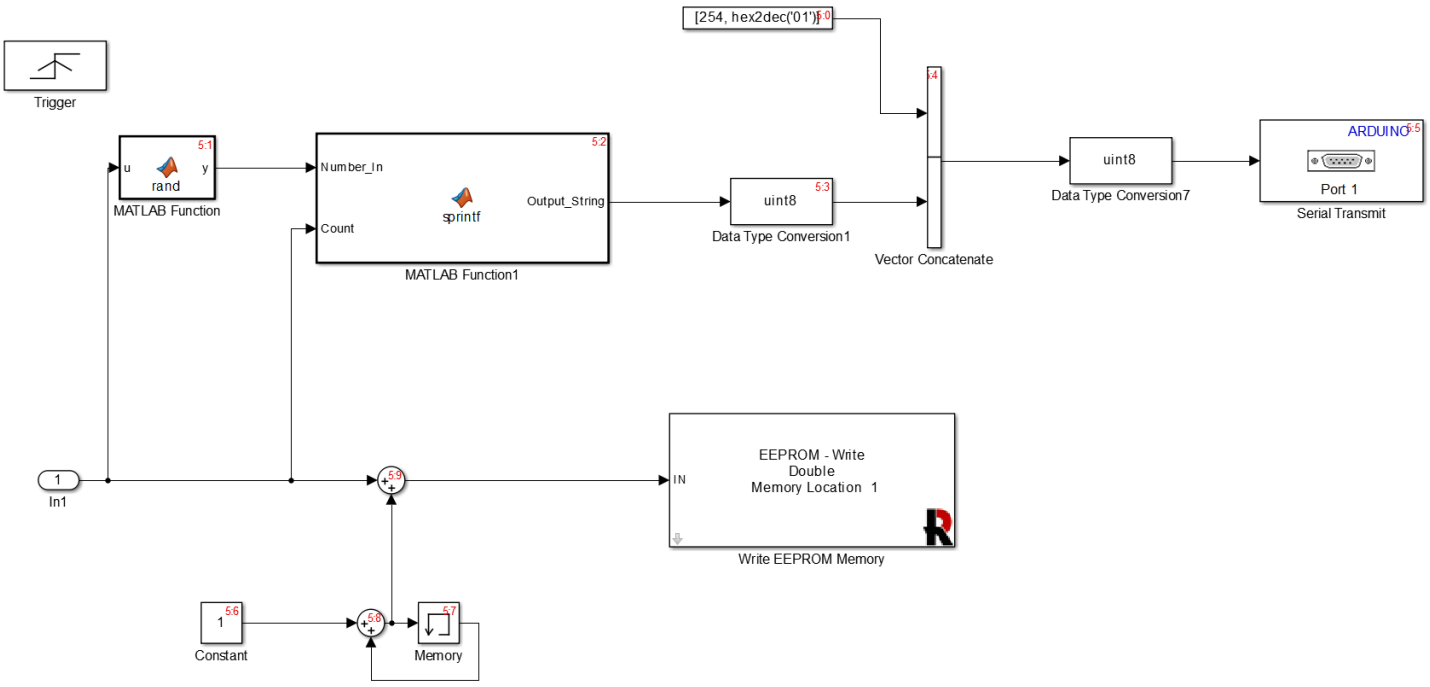


You may notice something strange about this output. First, why is your random number sequence the same as my random number sequence? Second, why is your random number sequence the same as your neighbors' random number sequence? And third, why are the random numbers the same every time you run your model? The answer is that the rand function generates a sequence of numbers. From one number in the sequence to the next, the numbers appear to be random. However, the sequence is always the same. Every time you restart your model, you are starting the sequence at the beginning of the sequence, as are your neighbors. To fix this problem, we need to give the random number generator a seed. The seed tells the random number generator where to start in the sequence. If we give the seed a random value, we start the sequence at a random location and we do not need to worry about the sequence repeating every time we start the microcontroller.

A typical way to generate a random seed is to generate the seed from the real time clock and base the seed on the time of day. The change of the controller restarting at the same time in a day is quite remote. For this lab, we do not have a real-time clock. Because the Uniform Random Number block does not have a way to change the seed, we will use the MATLAB **`rng(seed)`** function. This function specifies the seed for the random number function. Since we do not have a real-time clock, we will increment a counter every time we pick a random number, and then save that counter in the EEPROM. When we first start the Arduino, we will read the EEPROM and then call **`rng(seed)`** function once to seed the random number generator. The Model for doing this is shown below:

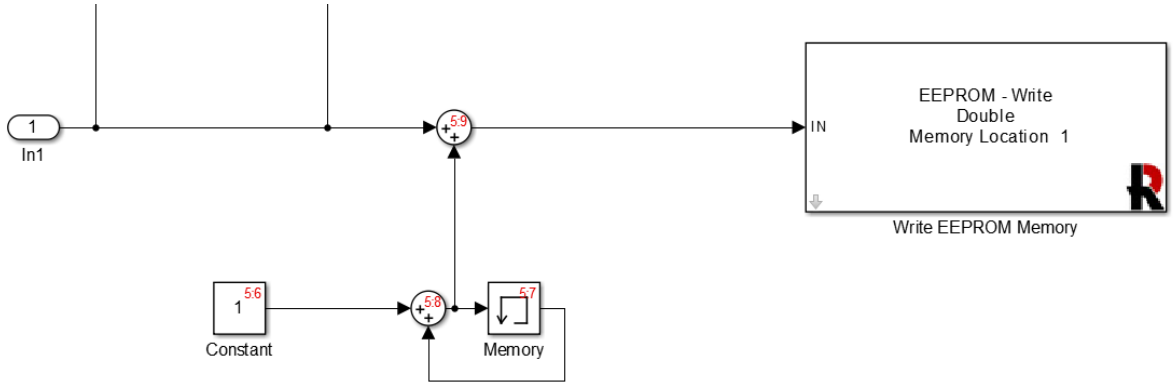


The model reads EEPROM memory location once when the model first starts running. The model will then generate a random number every time a pushbutton is pressed. Subsystem Generate a Random Number is shown below:



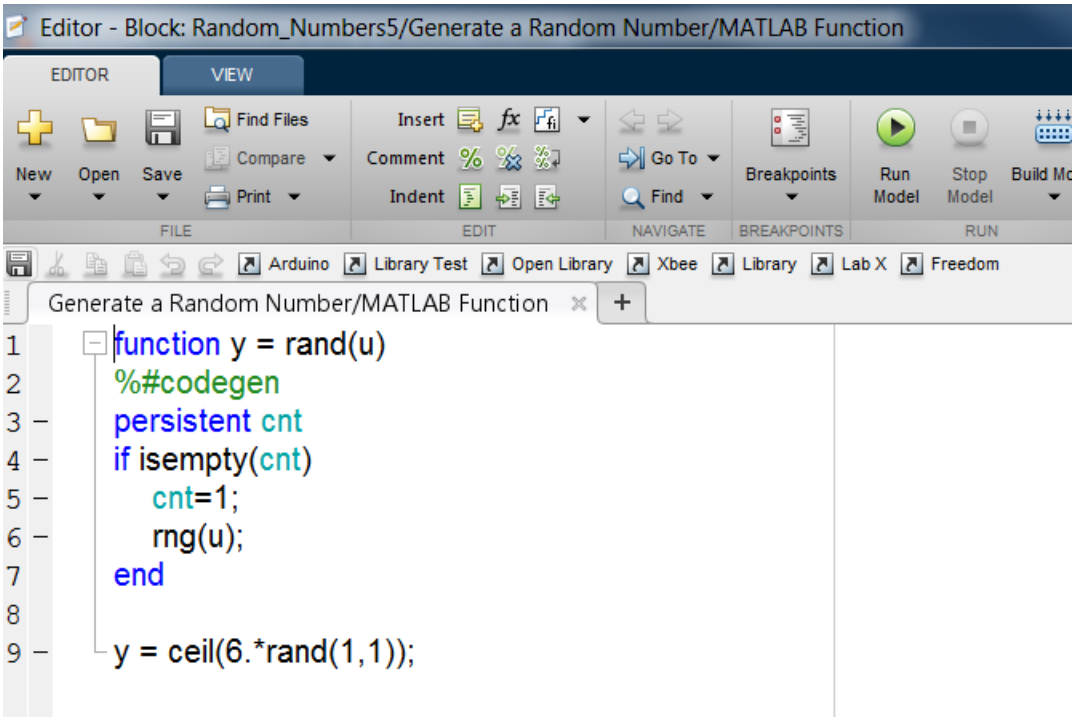
The signal on port **IN1** is the value we read from the EEPROM. This value remains the same the entire time the model is running. The only time it changes is when we reset the Arduino or cycle the power to the Arduino and the model reads the value from EEPROM memory location 1.

The portion of the model below is a counter:



Every time the push-button is pressed and this subsystem is executed the counter increments. The value of the count is added to the value of input IN1 (which is the value read from EEPROM memory location 1 at startup) and then stored in EEPROM location 1. Thus this counter stores how many random numbers we have generated, and saves that value in EEPROM location 1. When we restart the model, we will read the count and use it as a seed for the random number generator.

The MATLAB function to generate a random number has changed slightly:



Variable **`cnt`** is used to determine if this is the first time the function has been executed or not. The first time the function is executed, variable **`cnt`** is undefined and the function **`isempty(cnt)`** is true. Thus, the first time this function is executed, cnt will be set to 1 and the rng(u) function will be executed, giving the random number generator the seed value **`u`** (which is equal to the count stored in EEPROM memory location 1).

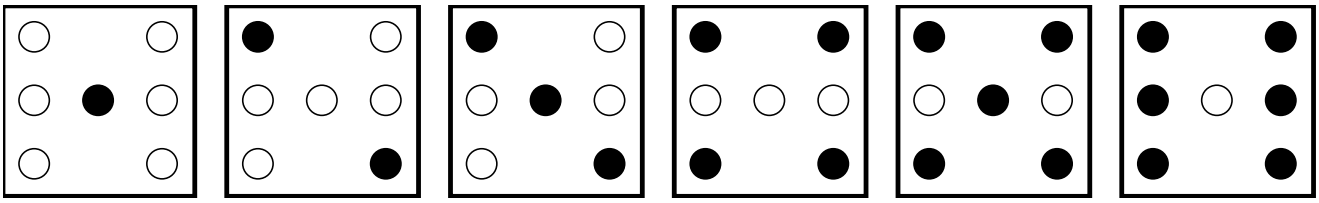
Once we set **`cnt`** to 1, variable **`cnt`** is defined and the **`rng`** function will not be called again until we reset the Arduino. Note that every time the function is called, it generates a random number between 1 and

6. The remainder of the model displays the random number and the value of the count read from EEPROM memory location 1.

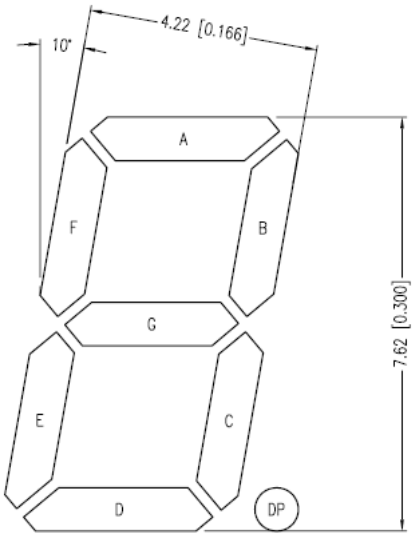
Demo XII.1: Demonstrate the operation of the random number generator. The LCD should display the random number and the seed value. The random number should change every time you press the push-button. The seed should only change after you reset the Arduino or cycle the power to the Arduino **AND** press the push-button. Note that you may need to create a separate model that initializes EEPROM memory location to 1.

B. Seven-Segment Display Decoder

Now that we have a way of generating a random number, we would like to display. We could use 7 LEDs arranged like the face of a die, and then turn on the appropriate LED's depending on the roll:



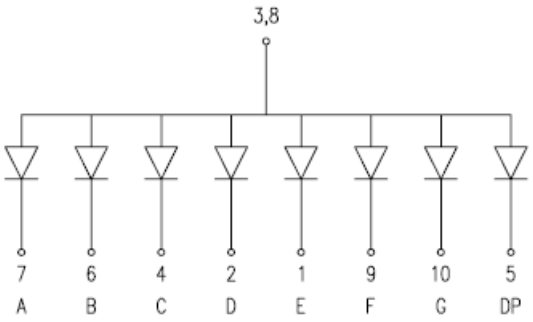
This would be fun, and use the same technique that we would present here. Instead, we will use a 7-Segment display and light up the appropriate segments to form Arabic numbers. An example of a 7-segment display is shown below:



[8]

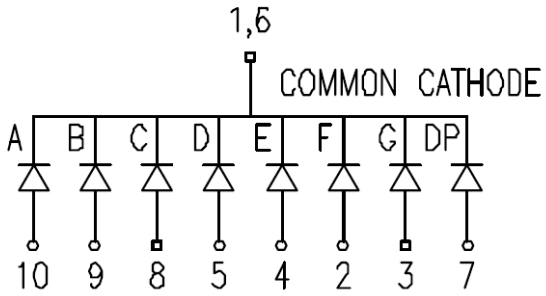
To display a number all we need to do is to turn on the appropriate LEDs. For example, to display the number 1 we will turn on segments B and C, to display the number 4, we turn on segments F, G, B, and C, to display the number 2 we turn on segments A, B, G, E, and D, and so on. Note that typical 7-segment displays have a decimal point as well, which we will not use in this application.

Seven-segment displays come in common anode configurations:



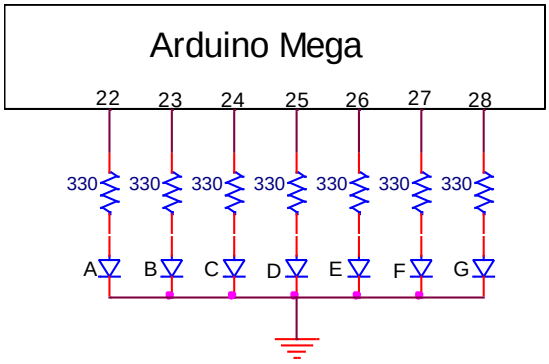
[8]

And common cathode:

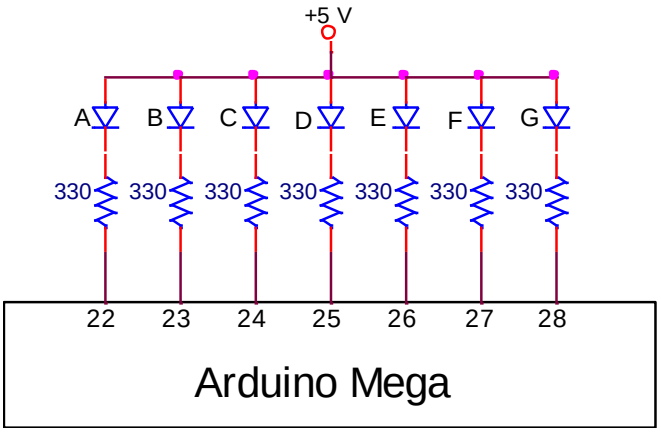


[9]

We can use either the common cathode or the common anode configuration. Each LED will require a series resistor, and the Arduino digital output can directly drive the LED. See section I.C.2, page 24 for example calculations. You can choose 7 digital outputs for this application. An example wiring diagram is given below for the common cathode configuration:

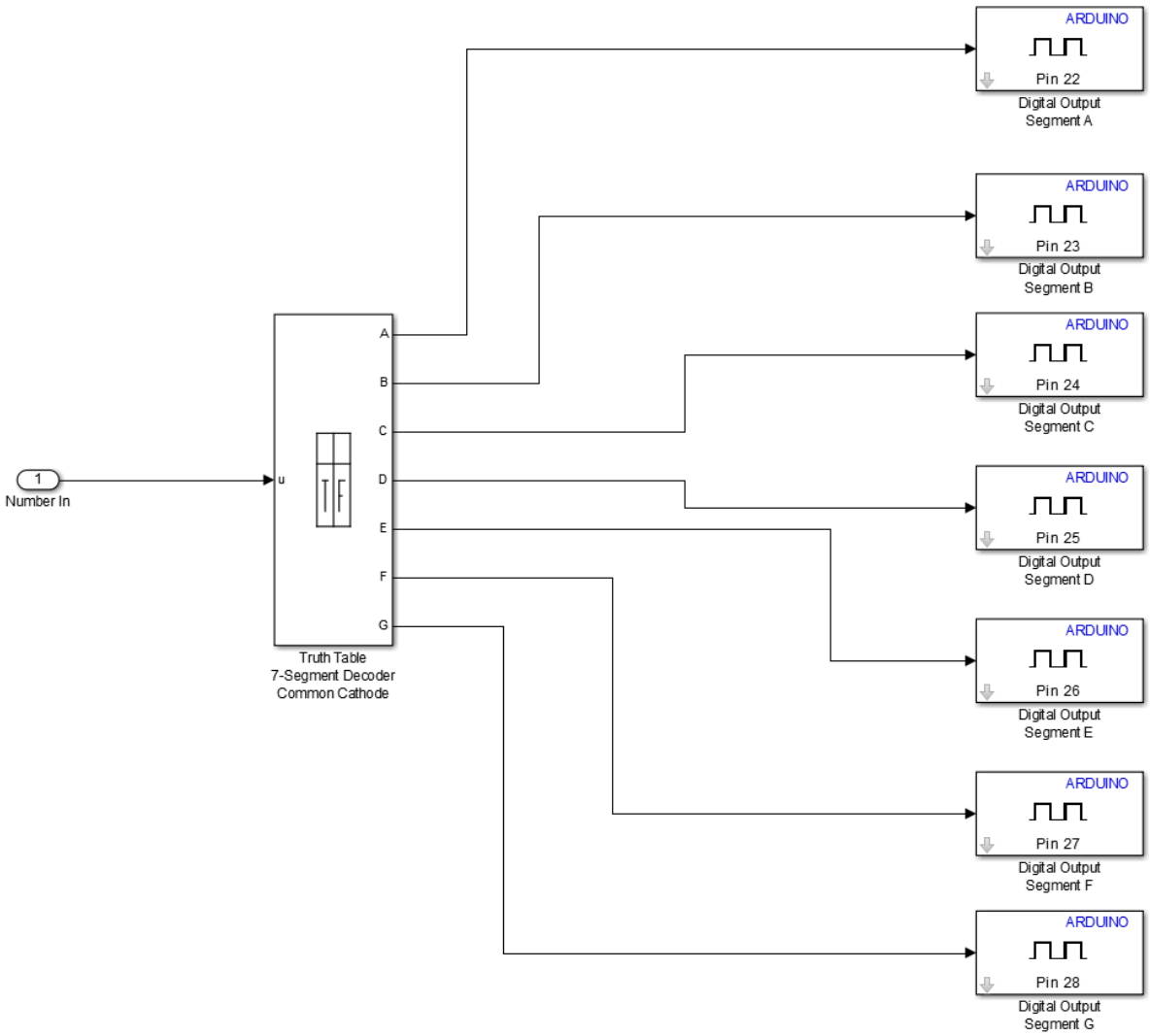


Note that in this configuration, an LED turns on when an Arduino Mega digital output goes high. The common anode wiring diagram is shown below:



In this configuration, an LED turn on when the Arduino Mega output is low.

The last thing we need to do is turn on the appropriate LED segments for the number we would like to display. We will do this with a truth table"



Labeling the truth table outputs as A through G and the **Digital Output** blocks as A through G makes it easy to do the logic and wire up the display. The condition portion of the truth table logic is shown below:

Block: Seven_Segment_Display/7-Segment Decoder/Truth Table 7-Segment Decoder*

File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D...
1	Input is 0	u==0	T	F	F	F	F	F	F	F	F	F	-
2	Input is 1	u==1	F	T	F	F	F	F	F	F	F	F	-
3	Input is 2	u==2	F	F	T	F	F	F	F	F	F	F	-
4	Input is 3	u==3	F	F	F	T	F	F	F	F	F	F	-
5	Input is 4	u==4	F	F	F	F	T	F	F	F	F	F	-
6	Input is 5	u==5	F	F	F	F	F	T	F	F	F	F	-
7	Input is 6	u==6	F	F	F	F	F	F	T	F	F	F	-
8	Input is 7	u==7	F	F	F	F	F	F	F	T	F	F	-
9	Input is 8	u==8	F	F	F	F	F	F	F	F	T	F	-
10	Input is 9	u==9	F	F	F	F	F	F	F	F	F	T	-
		Actions: Specify a row from the Action Table	1	2	3	4	5	6	7	8	9	10	11

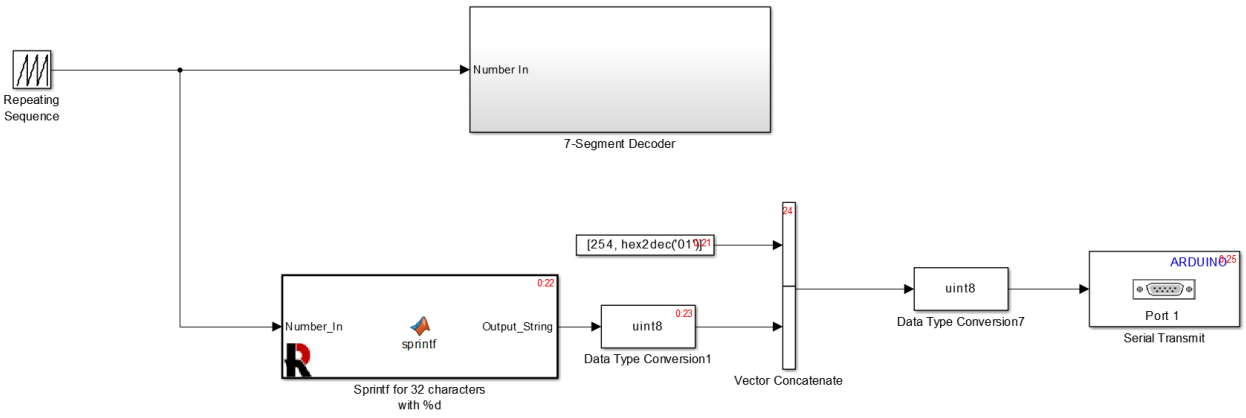
We are developing the table to display 0 through 9 in case we use this block for something other than this dice game, which only requires the digits 1 through 6. Note that action 11 is meant to indicate an invalid input, which will be displayed as an E on the 7-segment display.

The action portion of the truth table for a common cathode 7-segment display is shown below. Note that when an output is set to one, the specified segment will turn on.

Action Table

#	Description	Action
1	Display 0	A=1;B=1;C=1;D=1;E=1;F=1;G=0;
2	Display 1	A=0;B=1;C=1;D=0;E=0;F=0;G=0;
3	Display 2	A=1;B=1;C=0;D=1;E=1;F=0;G=1;
4	Display 3	A=1;B=1;C=1;D=1;E=0;F=0;G=1;
5	Display 4	A=0;B=1;C=1;D=0;E=0;F=1;G=1;
6	Display 5	A=1;B=0;C=1;D=1;E=0;F=1;G=1;
7	Display 6	A=0;B=0;C=1;D=1;E=1;F=1;G=1;
8	Display 7	A=1;B=1;C=1;D=0;E=0;F=0;G=0;
9	Display 8	A=1;B=1;C=1;D=1;E=1;F=1;G=1;
10	Display 9	A=1;B=1;C=1;D=0;E=0;F=1;G=1;
11	Display E	A=1;B=0;C=0;D=1;E=1;F=1;G=1;

Verify the logic of this truth table by testing your circuit with the model below:

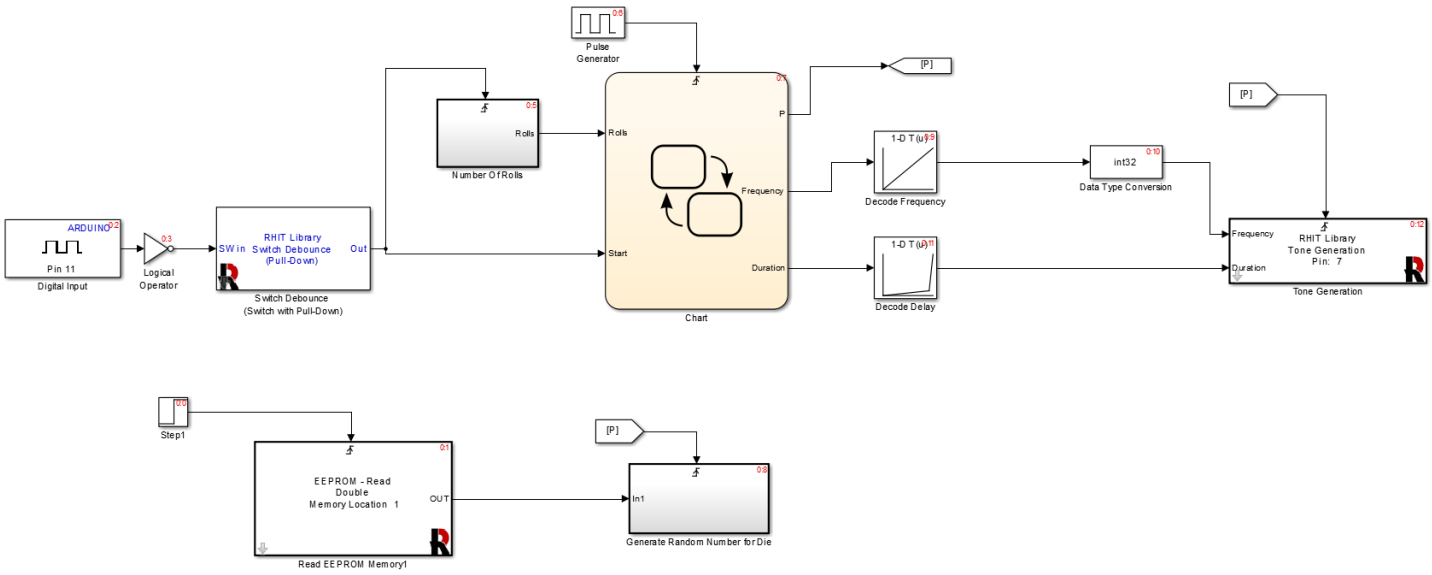


The repeating sequence generates numbers between 0 and 12. The 7-segment display should properly display the numbers 0 through 9, and E. The LCD display is used to verify that the values displayed by the 7-segment display correspond to the signal values.

Demo XII.2: Verify the operation of the 7-segment display.

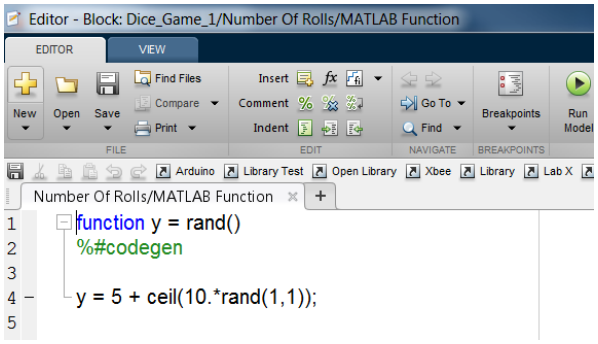
C. Random Length Roll Generator

We will now create our dice game. The idea is to simulate throwing a single die. There are actually two random events in this task. The first is how long the die rolls. The second is the value the die displays each time a new face ci shown as the die rolls. Thus, we will pick a random number between 5 and 15 for how many times the die changes its value as it rolls, and for each one of those value changes, we will pick a random number between 0 and 6. The complete model is shown below:



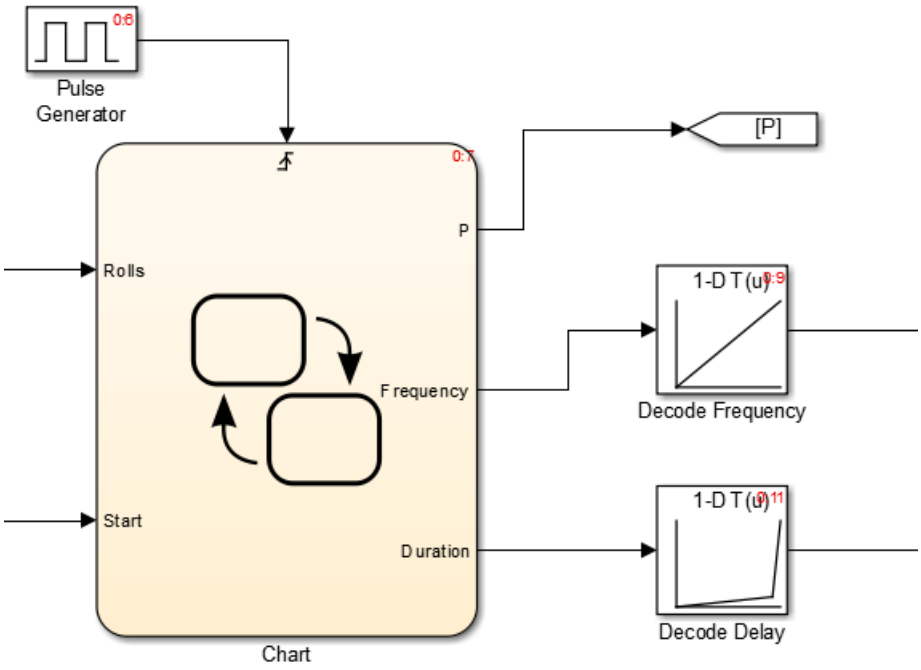
We will use a pushbutton to trigger our die roll. A positive edge will start things off. Each time we press the pushbutton, we will trigger the Number of Rolls subsystem. This subsystem contains the following:

The MATLAB function generates a random number between 5 and 15:



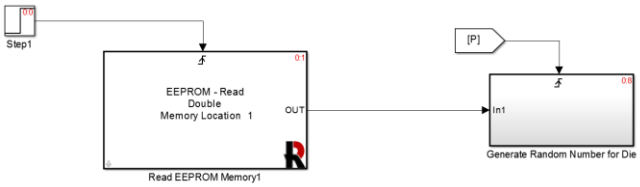
I chose these values because it just felt right when using the game. You can changes the values to your liking. (Called tuning…)

The Stateflow chart generates three outputs:

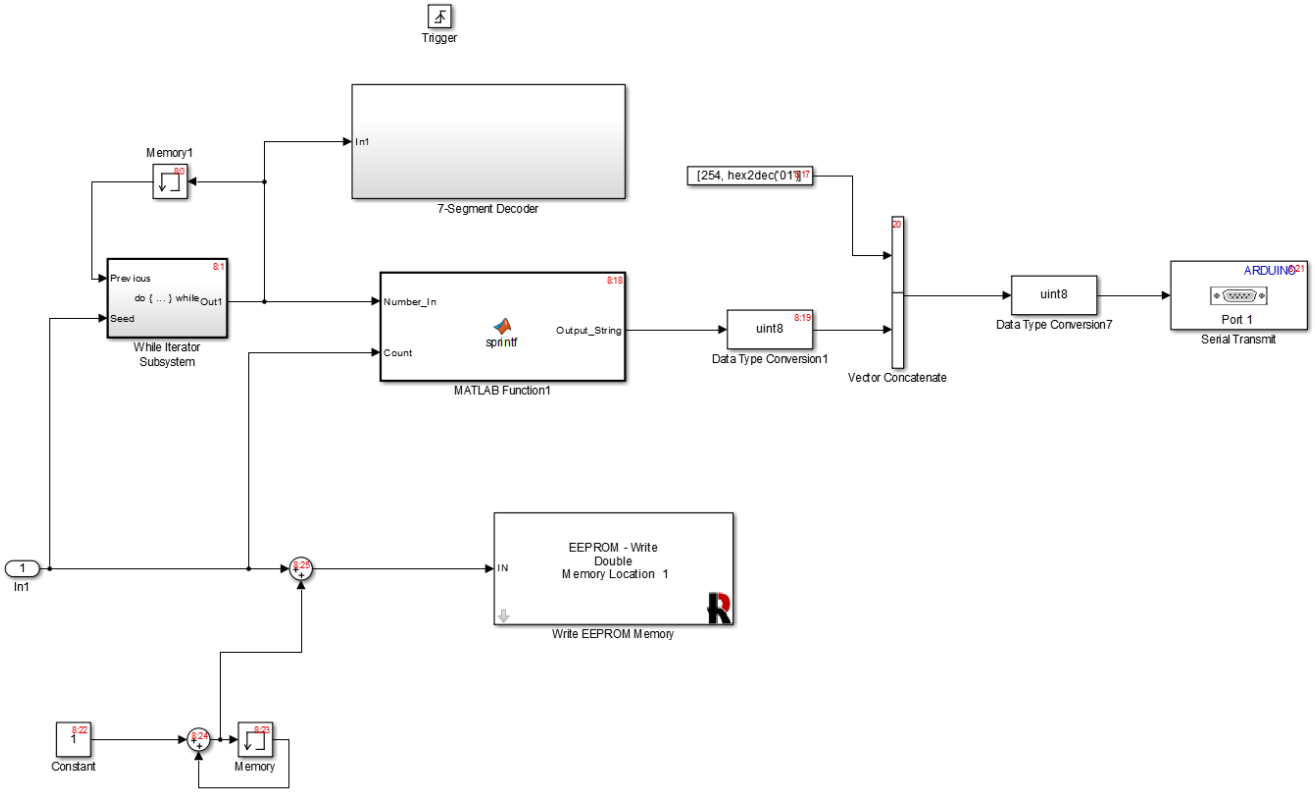


The **P** output triggers the Tone Generator so that it emits a tone and it also triggers subsystem “Generate Random Numbers for Die” to cause it to pick a random number between 1 and 6. Note that if the length of a roll is n, output **P** will trigger these subsystems a total of n times. The **Frequency** output is a number that starts at the number of rolls and counts down to 1. This value then goes to a lookup table (labelled as Decode Frequency) that converts a number between 1 and the number of rolls to a frequency. Using a lookup table allows us to tune the frequency to our liking; high frequencies for when the die is rolling fast and lower frequencies as the die slows down. The Duration output starts small and decreases to simulate a die slowing down as it rolls. Once again, this output is converted to duration in milliseconds for the tone generator using a lookup table. The lookup table was setup so that the speed of the beeping just sounded right.

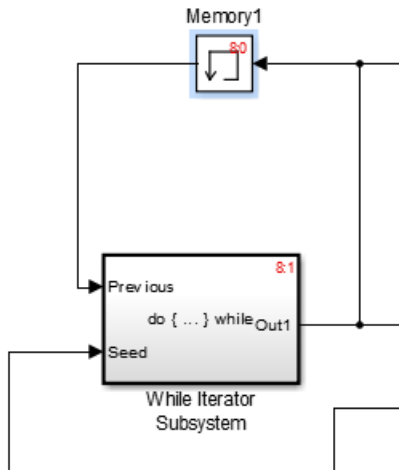
The remainder of the model is similar to what we did in Section XII.A:



Inside subsystem Generate Random Number for Die is a similar to the model we developed for picking a random number on page 207:

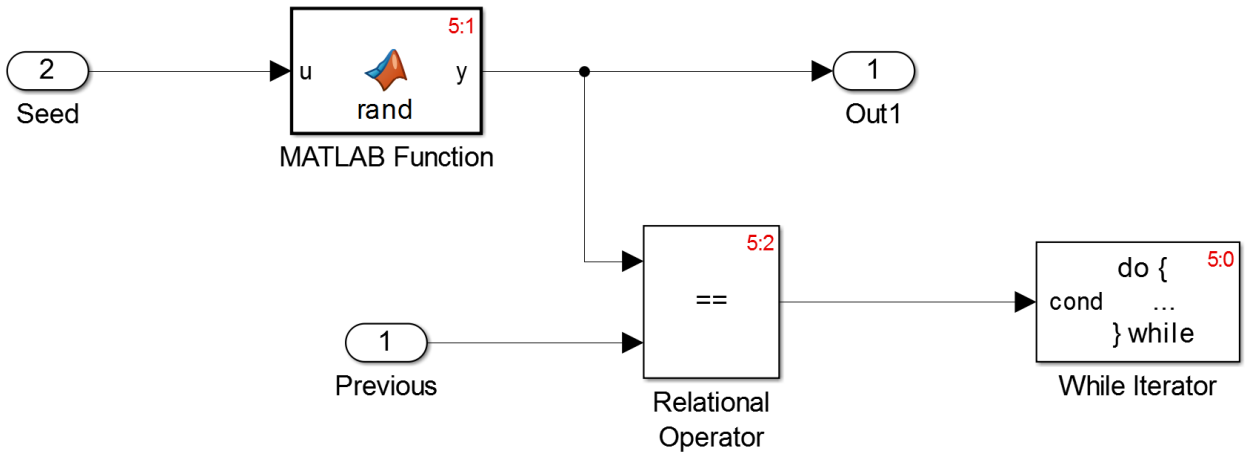


The difference is that the MATLAB Function block for generating a random number is replaced with a Do-While block and we added the 7-Segment display:

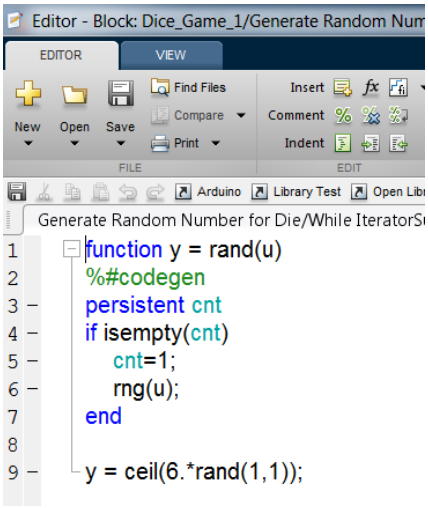


When you roll a die, as the die rolls from one side to another, a different number is displayed, and the die cannot display the same number twice. When it comes to rest, it can display the same number as it did the last time it came to rest. However, while it is rolling, two numbers cannot appear in a row because each face has a different number. Thus, when we pick a random number, we have a requirement that it must be different that the previous random number. Unfortunately, the random number generator in MATLAB does not care about this. Since the process is uniformly random, the chance of the next number being the same as the previous number is 1 out of 6. In our game, having it pick the same number 2 or three times in a row would look weird. Thus, we will use a do-while loop to keep choosing a random number until that random number is different

than the previous number. Note that a **DO-While** block is actually a While Iterator Subsystem which is located in the **Simulink** / **Ports & Subsystems** library. The contents of my **Do-While** block is shown below:



This block will be repeatedly executed until the condition becomes false. Thus, the MATLAB Function block will be called repeatedly until it generates a random number that is different than the previous value. The MATLAB function is the same as on page 208, and it is repeated below:



Demo XII.3: Demonstrate the Random Die game using a pushbutton to start the die rolling. A video of an example solution is shown at: http://wiki.ece.rose-hulman.edu/herniter/images/e/e1/Random_Die_Video.mp4.

Exercise XII.1: Modify the Die game to use an accelerometer. Every time the user shakes the board, the process of rolling will begin. An example video is shown at: http://wiki.ece.rose-hulman.edu/herniter/images/8/88/Random_Die_Video_2.mp4.

Lab XIII
XBee Wireless Communication

We will now look at using two XBee modules to create wireless link between two Arduino Mega microcontrollers. In this lab we will show how to send a limited amount of data between two Arduino computers, enabling a plethora of new applications to which we can apply our skills and the Ardunio Mega. This lab requires the following hardware available from www.sparkfun.com:

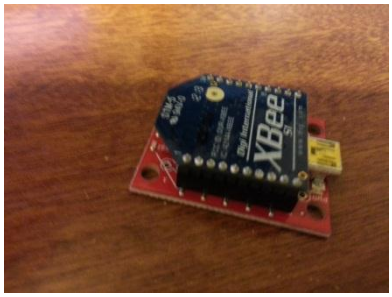
1. Two XBee Series 1 Modules, You can use either of the following
    a. XBee Pro 60mW Wire Antenna - Series 1 (WRL-08742)
    b. XBee 1mW Trace Antenna - Series 1 (WRL-11215)
2. One XBee Explorer, You can use either of the following:
    a. XBee Explorer Dongle (WRL-09819)
    b. XBee Explorer USB (WRL-08687)
3. Two XBee Shields (WRL-10854)

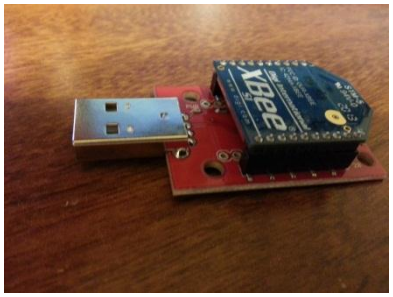
The difference between the two XBee modules is the transmitting power. For lab work, the 1 mW versions will work ok. For long transmission distances (say from my barn to my house), the higher power 60 mW version would be needed. The XBee explorer allows us to use an XBee receiver with our Windows laptop. This will allow us to configure the XBee module as well as monitor communications. It might not hurt to get three XBee modules, two for the two Arduino microcontrollers and one for your windows computer so that you can monitor the communication.

You will also need to download some software to configure the XBee modules. Go to http://www.digi.com/support/productdetail?pid=3352 and download and install the appropriate version of the Next Generation XCTU. Install the XCTU software.

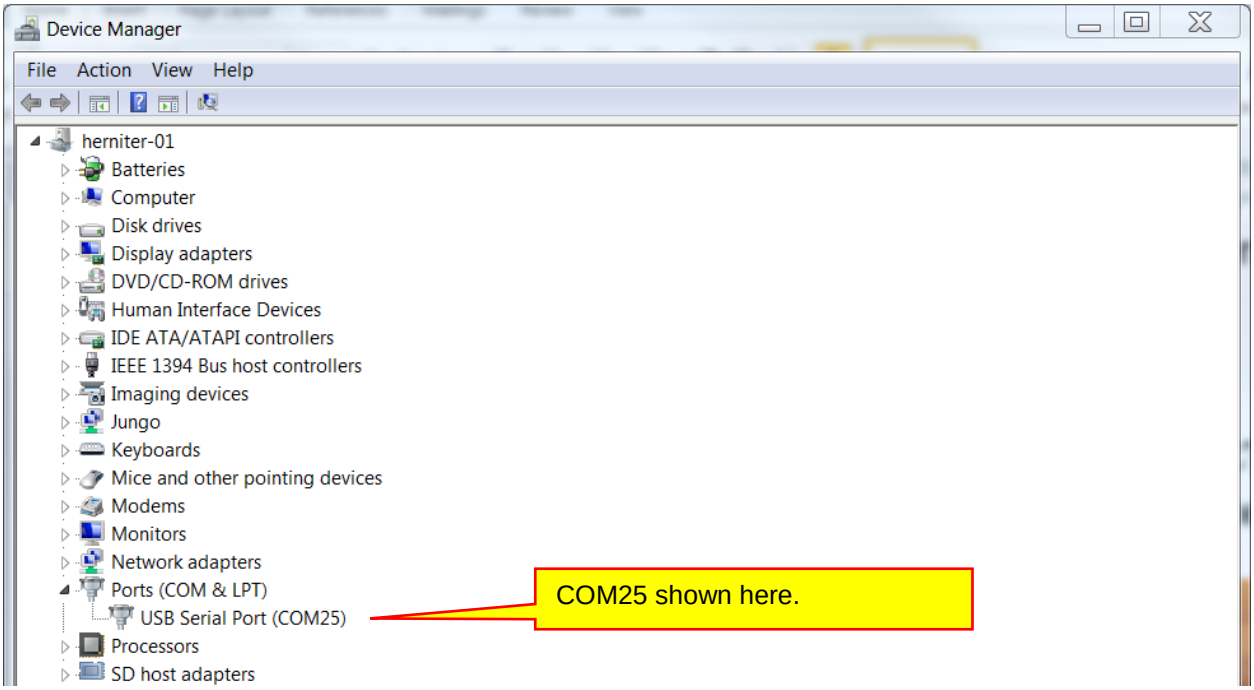
A. Configuring an XBee Module

Next, we will use the XBee explorer board and the XCTU software to configure the XBee module. Mount the XBee module in the explorer board as shown in one of the two pictures below:

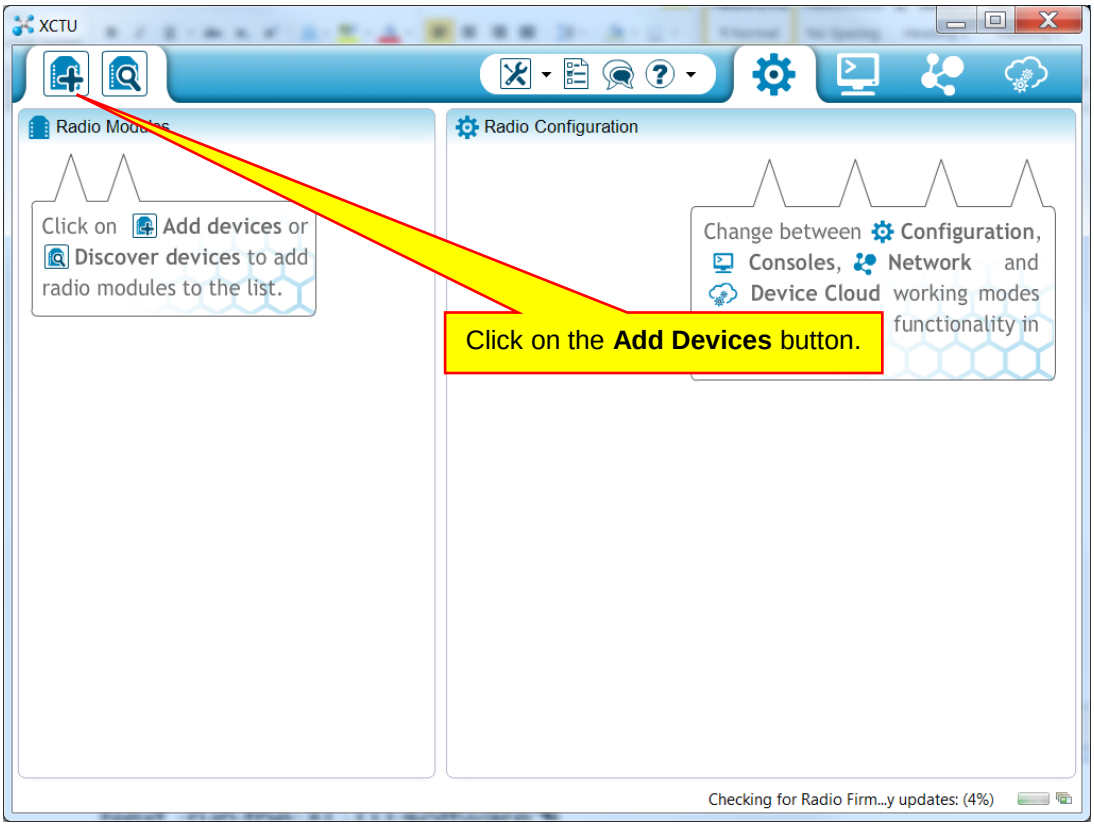




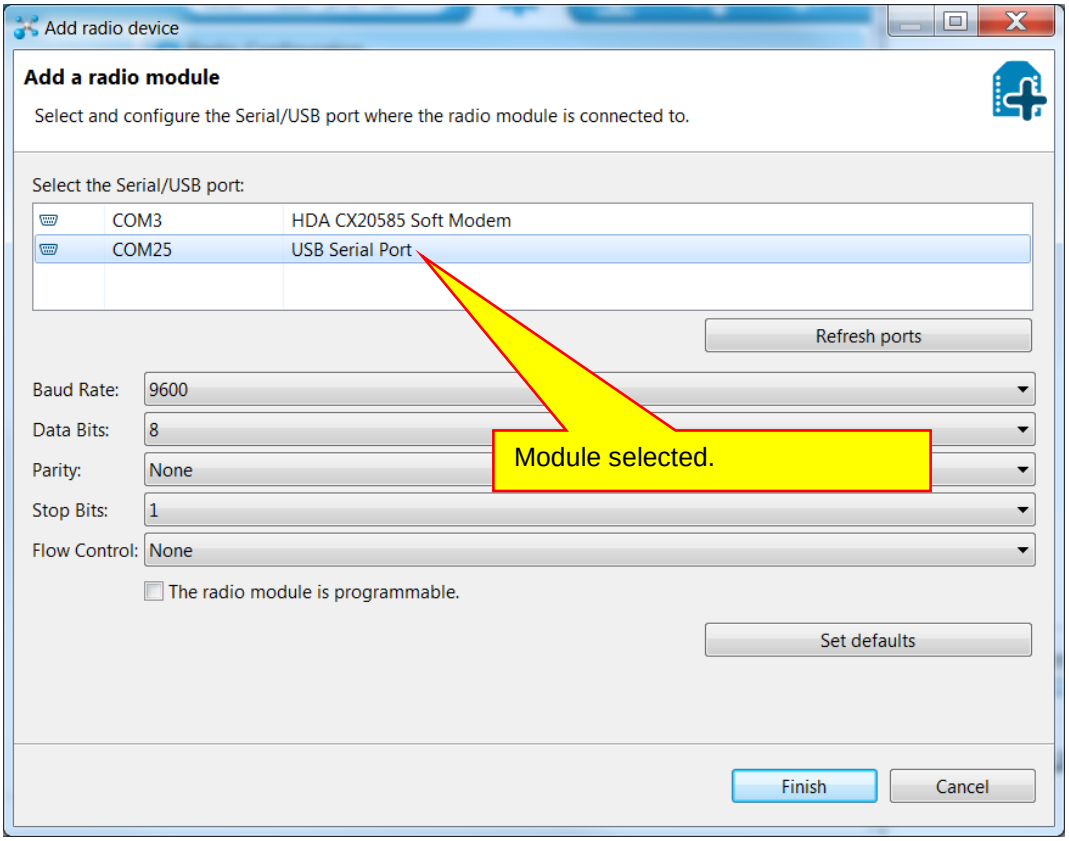
Plug your explorer into one of the USB ports in your computer. It will be configured as a serial port. Use the Windows Device Manager to determine the number of the COMM port that the XBee Explorer uses. Mine is configured as COM25:



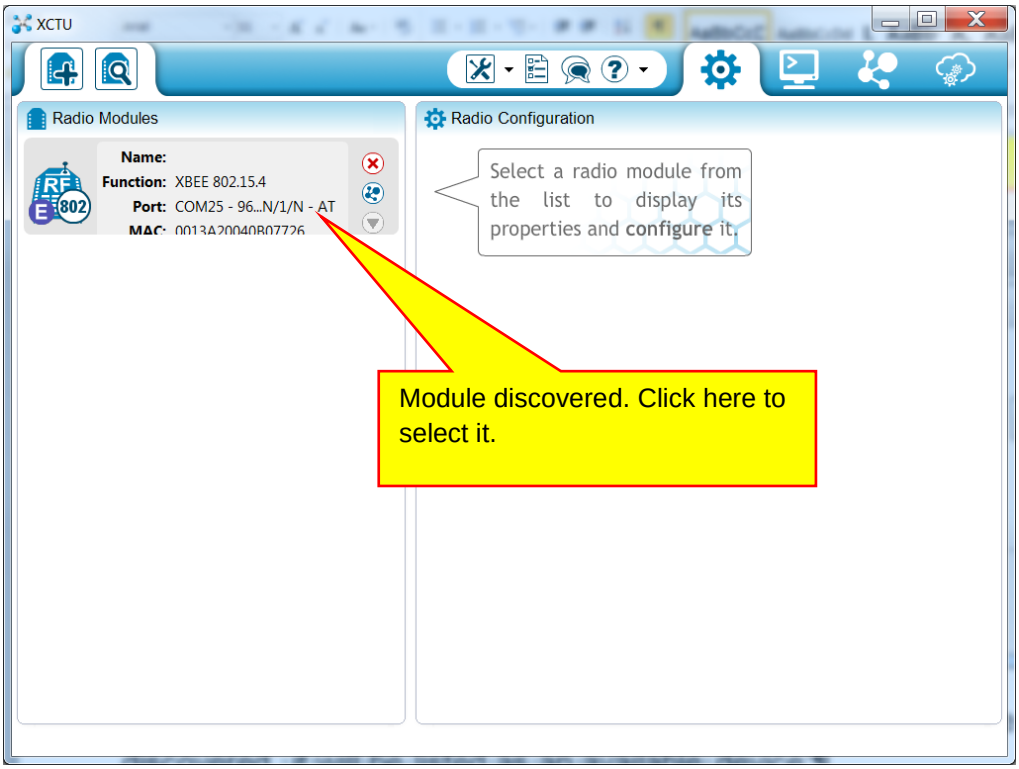
Next, run the XCTU software:



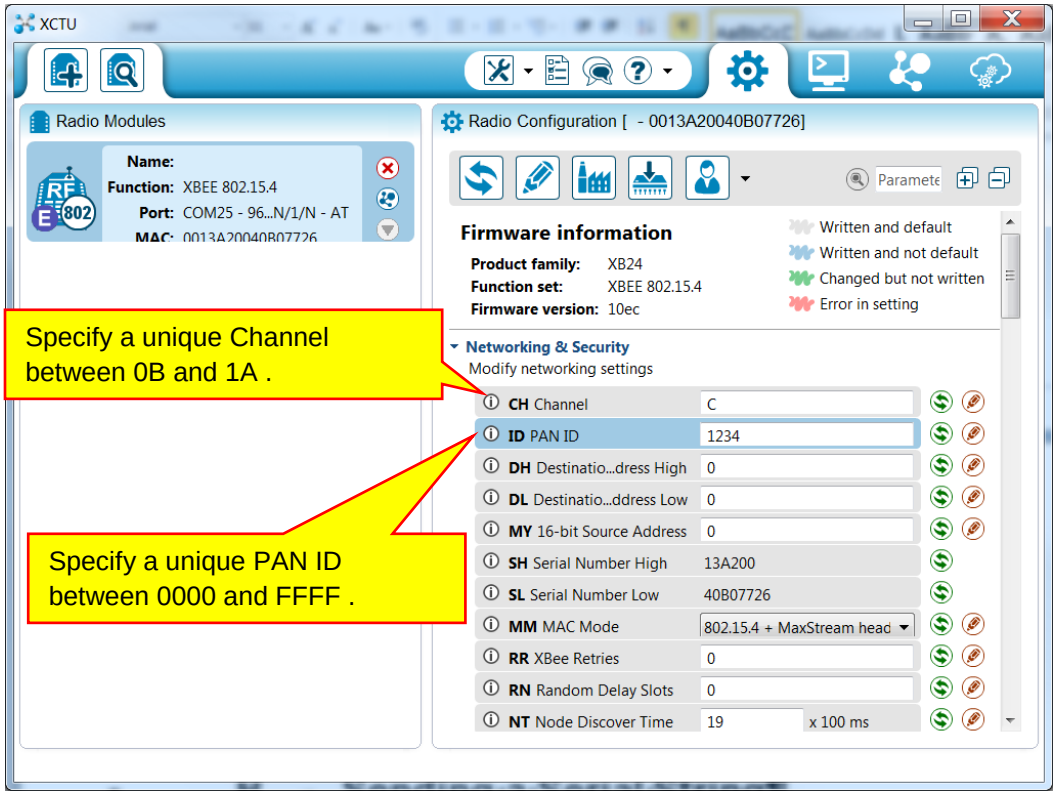
After clicking on the Add Devices button, select the device on the port for your XBee module. My XBee module is on COM25:



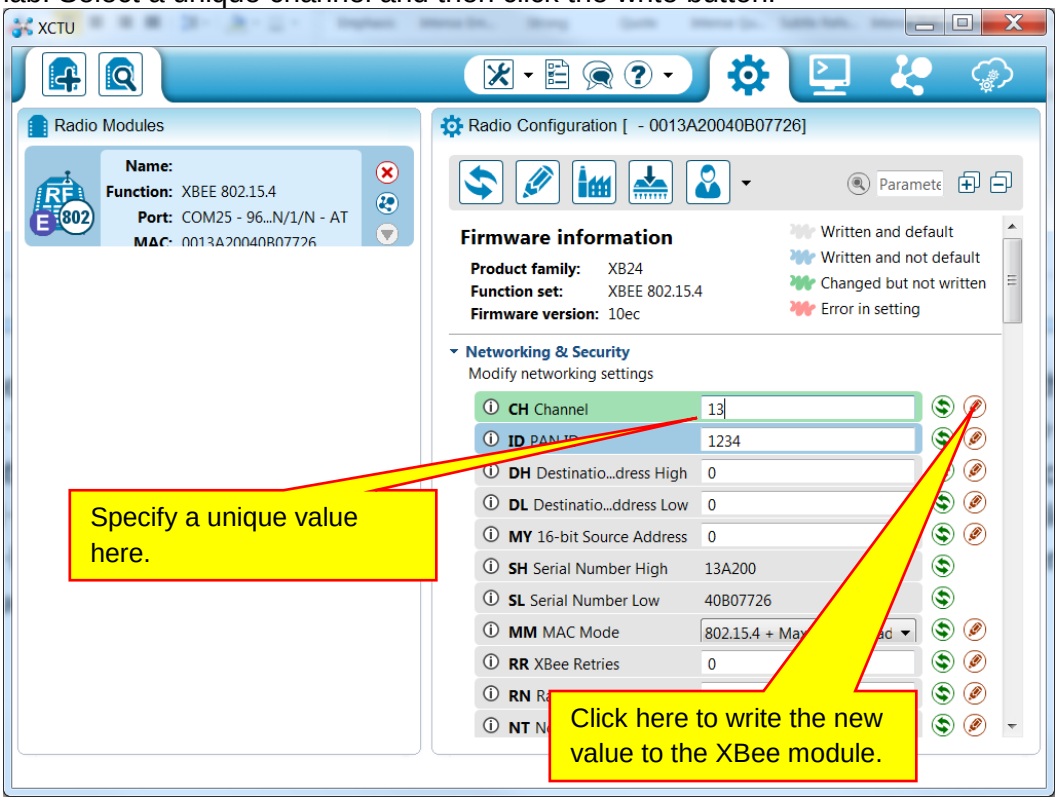
Note the communication speed. We will need these for later use. Click the **Finish** button. When your module is discovered, it will be listed as an available device:



After you select your module, its parameters will be listed:



We need to change two parameters. XBee modules that use the same channel and have the same PAN ID can communicate. The problem is, with several XBee receivers all in the same lab, how do we guarantee that messages are sent between the appropriate Modules? The answer is that you should select a unique channel and a unique PAN ID for your XBee modules. Thus, make sure that all of your XBee modules have the same Channel and PAN ID, but make sure that your Channel and PAN ID are different from everyone else's in the lab. Select a unique channel and then click the write button:

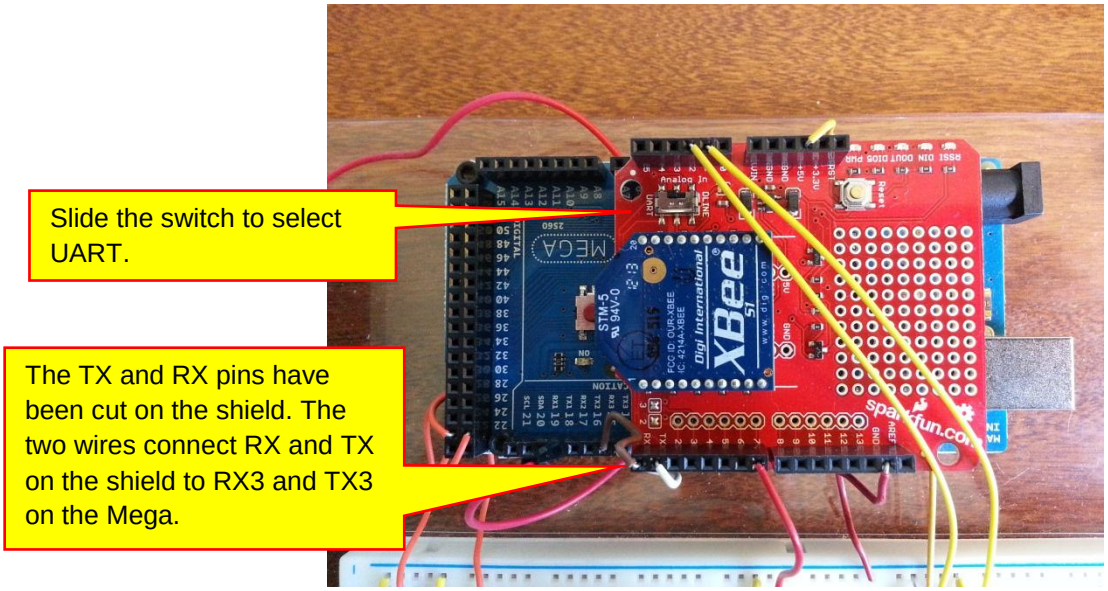


Select a new PAN ID and then click the write button to write that parameter to the XBee module. You will need to repeat this process for all of your XBee modules.

This is all that we need to do for our application. You should leave one XBee module in your XBee explorer board. In the next part, we will send text from the Arduino Mega to your Windows comnputer using the serial ports and the XBee modules.

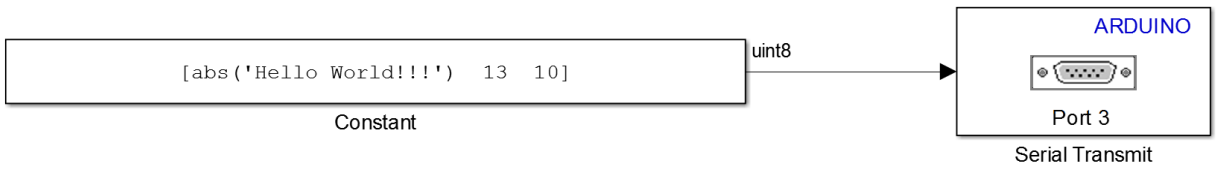
B. Sending a Serial String

As a first test of the XBee module, we will use the Arduino Mega to send a text string and use the XBee Explorer to receive the text on a Windows computer. The XBee shield has the pins labelled the same as the pins on the Arduino Mega. Thus, you could plug the shield directly into the Mega. This does cause one problem. The shield uses RX0 and TX0 on the Mega, which by itself is not a problem. The problem arises when we use External mode and when we program the Arduino. This communication goes through the USB port which also uses RX0 and TX0. Thus, if we use serial port zero for our XBee communication, we might have trouble reprogramming the Mega. To avoid this problem, cut the pins on the shield that connect the shield to RX0 and TX0. Then manually wire the RX and TX outputs of the shield to RX3 and TX3 of the Mega.



Also note in the picture above that the switch on the XBee shield should be set to UART. This will rout all communication to the XBee module through the RX and TX. Also note the orientation of how the XBee is mounted on the shield.

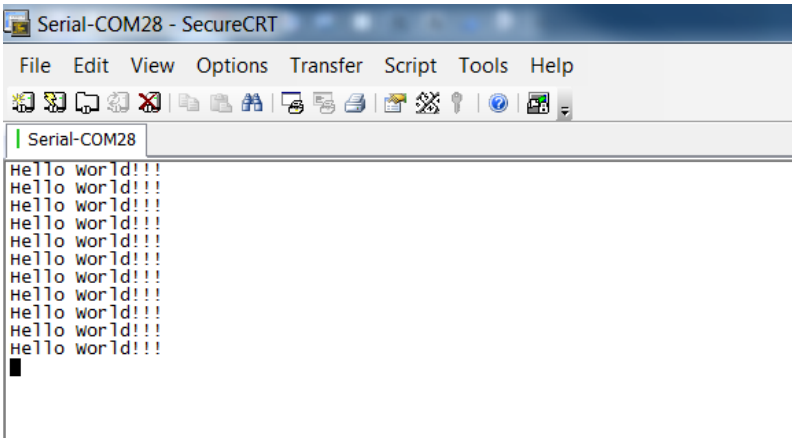
Now that we have the XBee hardware mounted correctly and the shield RX and TX connected to RX3 and TX3 of the Mega, we will create a model that send out a text string:



The constant is specified as data type uint8. The value of the constant is the ASCII codes for the characters in the text string ‘Hello World!!!’ followed by the ASCII codes for a carriage return and a line feed (13 and10). This

text string is sent to Serial Port 3 on the MEGA which is connected to the XBee serial input. The text string will be transmitted by the XBee Module and received by any XBee module on the same channel and PAN ID. This model runs with a fixed time step of 1 second. Thus, the text 'Hello World!!!' will be transmitted one every second.

Next, plug in your XBee explorer into your computer. Use the Windows Device Manager to determine the COM port that it uses. Then run a communication program such as Secure CRT to view the data received by the XBee mounted on the Explorer board:

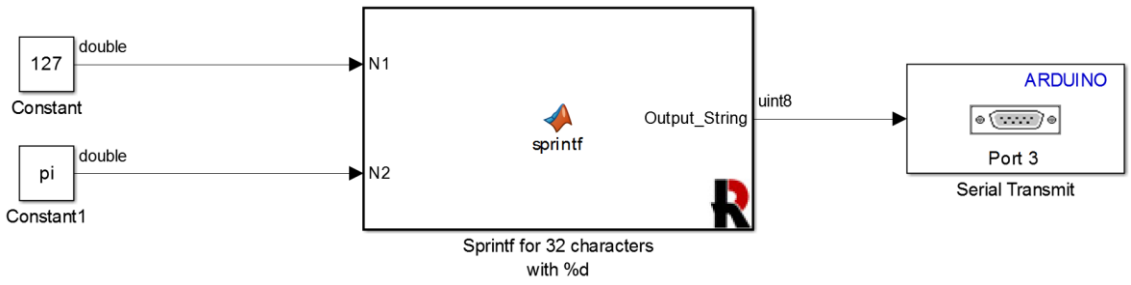


Demo XIII.1: Demonstrate the 'Hello World!!!' model by transmitting the text string with your Arduino Mega and receiving the text string using the XBee explorer and a serial communication program.

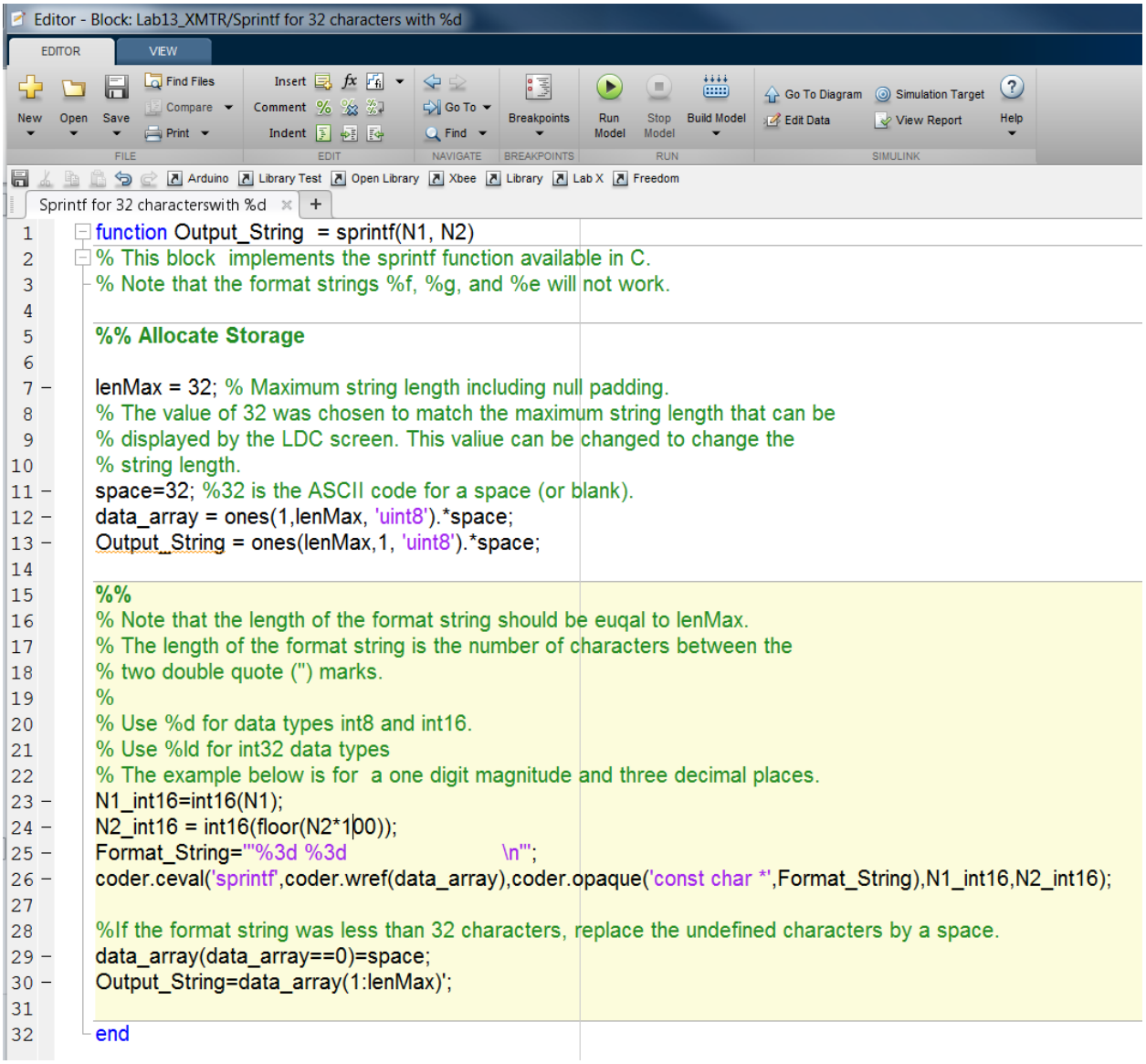
C. Sending and Receiving Numerical Data

Sending numerical data using the XBee module is quite easy. All we need to do is use the sprintf function to send data to the serial port used by the XBee module. Our only limitation will be that numerical data needs to be sent as integers, and we need to send a special character that marks the end of a transmission. Thus, if we need to send a value like 3.14, we will multiply it by 100 and send it as 314. When we receive it, we will receive it as 314 and then divide by 100 to obtain the original value. Receiving data will require us to develop a new function (which is in the libraries). This function will need to determine is data is incoming, when the data stream ends, parse the data into separate values, and then output those values as Simulink signals.

For the first example, we will transmit a numerical value from one Arduino Mega to another. One Arduino will be the transmitter and send two numbers, 127 and 3.14. The other Arduino will be the receiver and receive the two numbers and display them on the LCD screen. The transmitter is quite simple. We will just just convert the numerical values to a texts string and then send them to the serial port connected to the XBee module. The transmitter model is shown below:



We will send double precision values to our sprintf function. We will use this function to convert the values to decimal values in a text string, and then send the text string to port 3 which is connected to the XBee module. Most of the work is done inside the sprintf function. This function is located in the RHIT Arduino library. You will need to place this block in your model and then modify it. Remember that in order to modify the block, you must disable the library link and the break the library link. (If you do not remember how to unlink a block, the steps are listed in the block description when you double-click on the block.) Modify the function as shown:

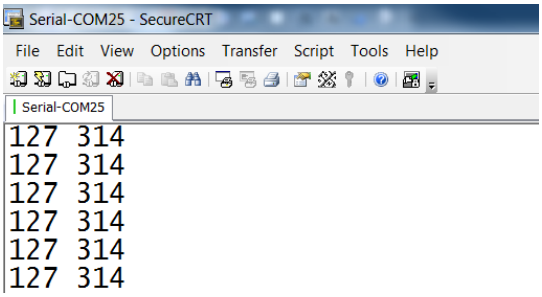


Some highlights of the changes:

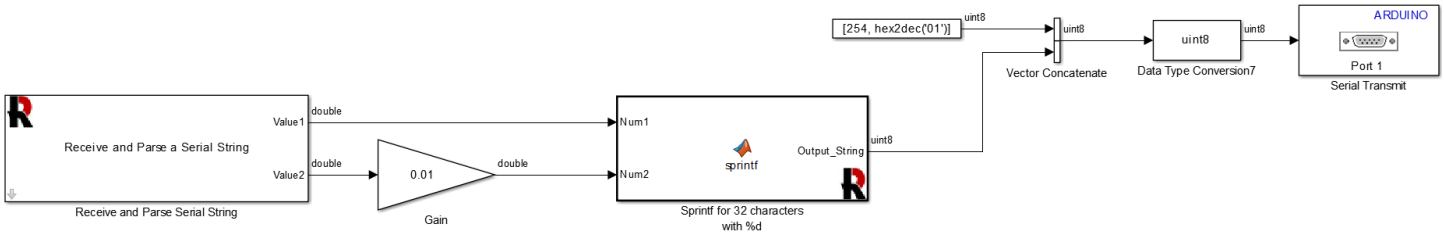
1. There are two inputs now N1 and N2.
2. We must convert both inputs to an integer data type.
    a. **`N1_int16=int16(N1);`** Since the first input is already an integer, we do not need to round off the value. All we need to do id convert the data type.
    b. **`N2_int16 = int16(floor(N2*100))`**: Remember that N2 is equal to π. We multiply the value by 100 so that the value is equal to 314.159… Then we floor the result to chop off the fractional part of the number. We are left with 314, which we convert to the int16 data type.

3. The format string must be 32 characters long. This is only required because we are reusing the sprintf function we developed for displaying text on the LCD screen.
4. The %d format string displays a number as an integer.
5. Numerical values are separated by a space. (There is a space between the %3d and the next %3d.)
6. The character \n is used to mark the end of a line and counts as one character. The receiver will look for this character to determine the end of a transmitted line of data.

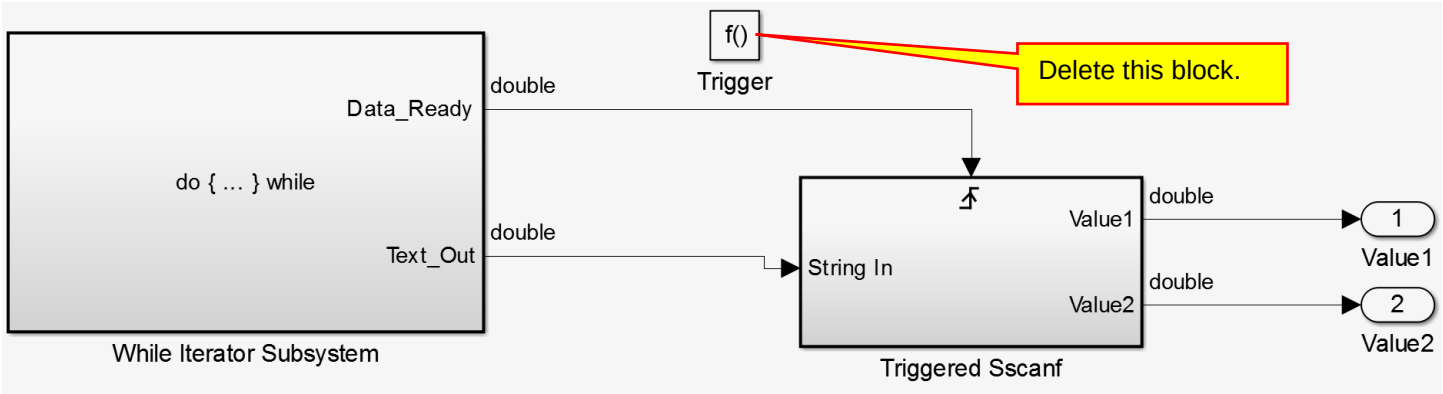
Run this model on your Arduino transmitter. You can use the XBee explorer board and a serial communication program on your windows computer to verify that the Arduino/XBee is transmitting data. The window from my serial communication program is shown below:



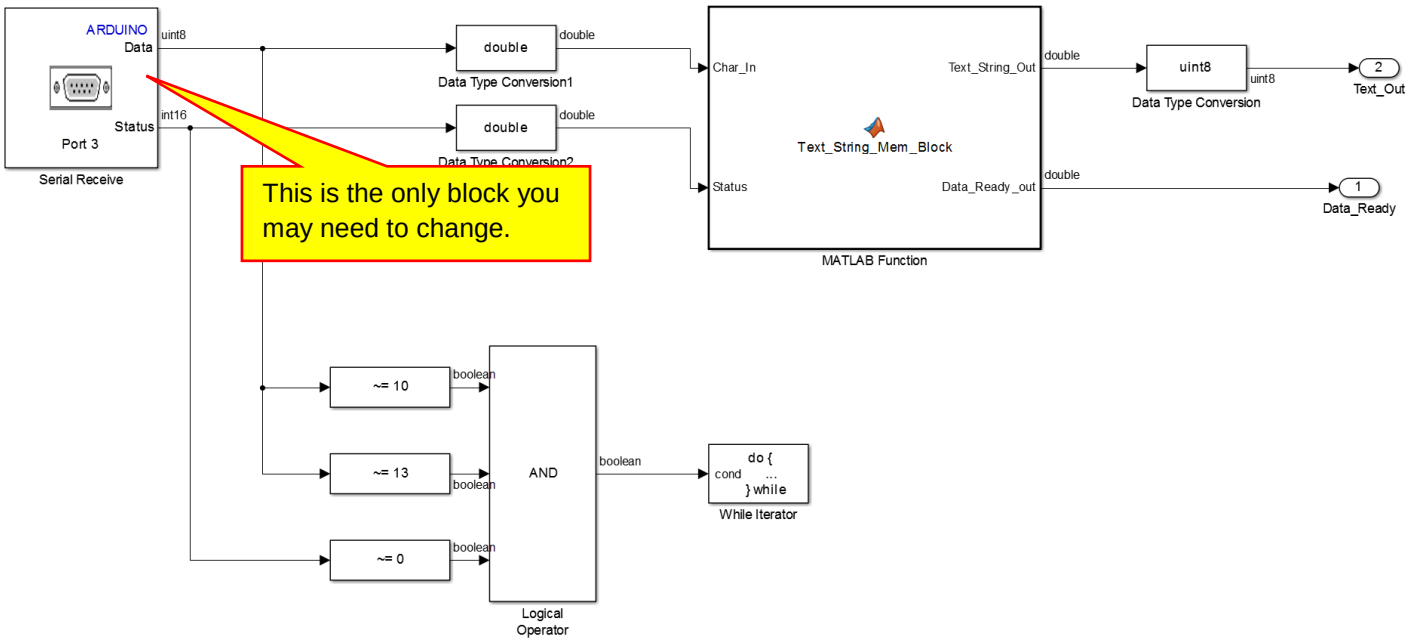
We will now create a model that receives the text string and displays the values. The complete model is shown below:



Most of the work is done for you in block **Receive and Parse Serial String** which is located in the RHIT Arduino library. Normally you will need to make a few modifications to the block to tailor it to your needs. For this example, all we need to do is to delete the trigger. However, we will look at the block to see where else we would make changes if needed for future projects. Remember to disable the library link and then delete the library link before proceeding. After disabling and breaking the link, look under the make of the **Receive and Parse Serial String** block:



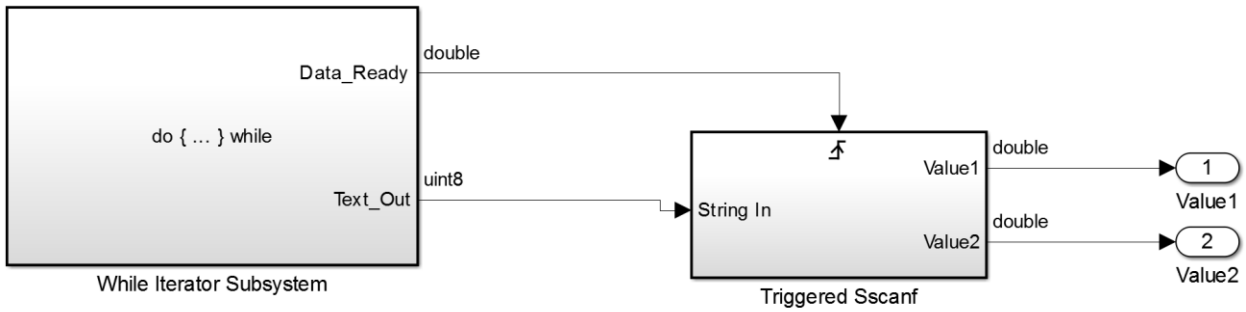
For this application, delete the **Trigger** block. Next, double-click on the While Iterator subsystem to view its contents:



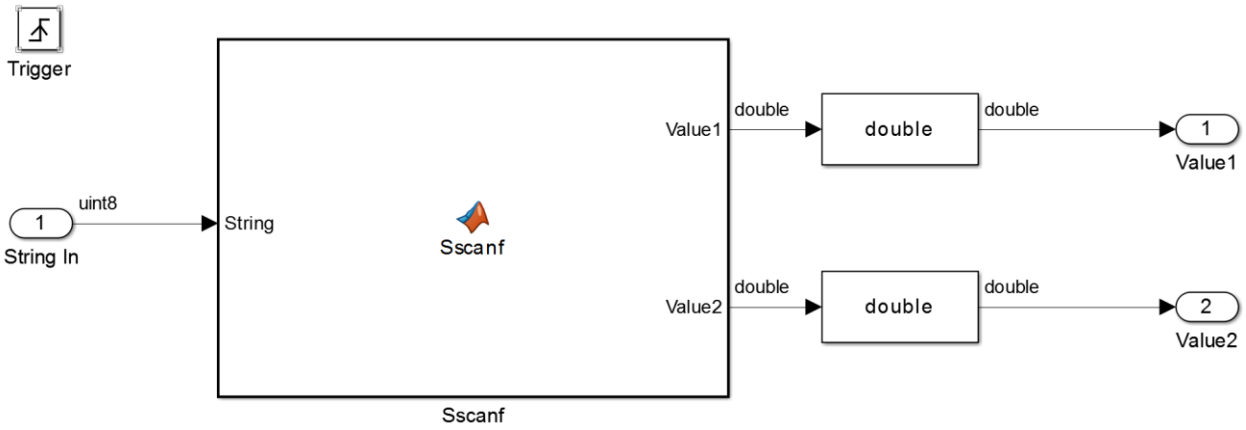
Change the serial port to match the port that you use for your XBee module.

Except for the port, you will usually not make any changes to this portion of the block. This While Iterator subsystem reads the serial string and fills the output text string with data until it receives an end of line (EOL) character (which we send using the \n of the sprintf function). When an EOL is received, the Data_Ready_out signal goes high signaling that a text string of 32 characters has been received with the proper terminating EOL character. This text string can now be used by the rest of the model. The data Ready_Line only goes high when a complete text string with the proper termination has been received. If these conditions are not met, the data is read but not passed to the output. Erroneous data is discarded up to the EOL character. After an EOL is received, the block will again look for a valid string.

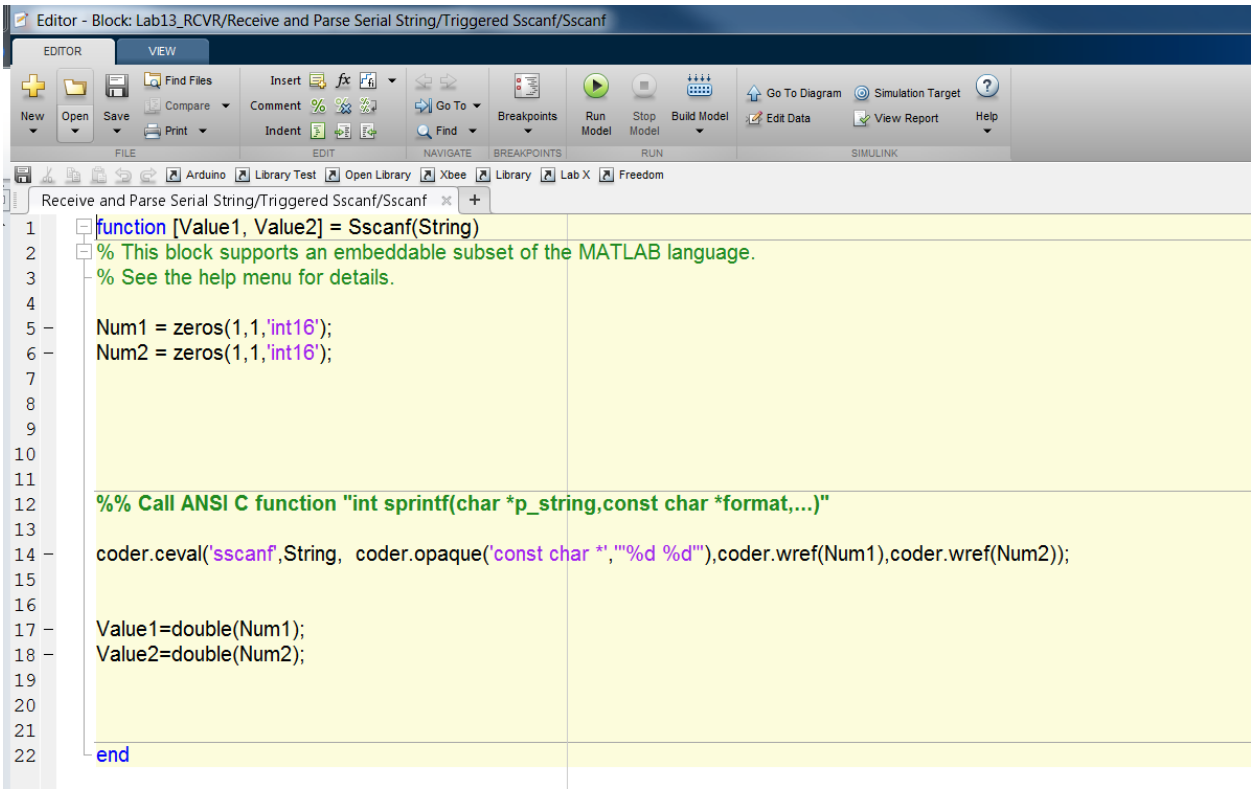
After changing the serial port and deleting the trigger, go up one level:



Open the Triggered **Sscanf** block:



You will typically need to modify the Sscanf block (but not for this example). Open the Sscanf block:



This block reads two values from the text string using the %d format. It is easy to see how to modify this function to read a single value or to read more than two values. We are reading two values, so we do not need to make any changes.

Before we run this model, a few notes about the **Receive and Parse Serial String** block are in order. If no data is available on the serial port, the output of the block stays at the same value it was from the last successful read. Thus if no data is present, the block output will not change. Because of this property, we can read data faster than it is transmitted. If no new data is present, the block holds the last valid value received. Since we can read faster than it is transmitted, we can receive data immediately as we can set this model to have a smaller fixed time step than the transmitter.

Realizing the above property of this block, set the fixed time step of the receiver model to 0.5 seconds. Remember that we set the fixed time step of the transmitter to 1 second. Thus, we send new data every second and we attempt to read the data every 0.5 seconds. Because the block holds the last value and only changes it value when new data is received, this timing allows us to receive the most recent data at the next read. Also note that it is possible to read data slower than it is transmitted. Logically this has a problem as the buffer in the XBee module will eventually fill up and we will lose data.

Demo XIII.2: Show the transmitter / receiver pair sending out data and receiving data.

Exercise XIII.1: Create a model with a counter that counts in a loop from 0 to six. The count changes once a second. The Arduino and XBee module will transmit the values of the count, the value of the count times pi, and two times the count. The receiver will receive all three values and display all three values on the LCD screen.

Exercise XIII.2: Create a counter that uses two Arduino Mega's linked by XBee modules. When a Mega receives an integer, it will add one to the integer, wait one second, and then transmit the updated value through the XBee module Each model will display the count on the LCD screen. With two linked computers, the counters should count up at a rate of one count per second. (Each mega will actually change its value once every two seconds.) A pushbutton should be used to reset the count to zero. The counters can automatically start counting after you press the reset button or cycle the power.

Exercise XIII.3: Create the singing cricket's demo. In this demo, we have three crickets. When cricket 1 hears a "chirp" from cricket 3, it will wait one second and then chirp at a frequency of 2000 Hz. When cricket 2 hears cricket 1, it will wait one second and then chirp at 2500 Hz. When cricket 3 hears cricket 2, it will wait one second and then chirp at 3000 Hz. The crickets don't listen for sound. Instead, each cricket sends out an ID number wirelessly using the XBee module. When the cricket receives the appropriate ID, it emits a chirp and then sends out a new ID that tells the next cricket in the series that it is that cricket's time to chirp.

Exercise XIII.4: Combine the functions of Exercise XIII.2 and Exercise XIII.3 to create a series of three crickets that count and chirp.

Exercise XIII.5: A possible problem with the model of Exercise XIII.3 is that if one cricket fails to hear its ID, it will never chirp and the ring will be broken. Create error-detecting logic that will notice that the chirping has stopped and restart the sequence. This can be difficult as only one of the crickets should restart the sequence or you may get several crickets chirping at the same time.

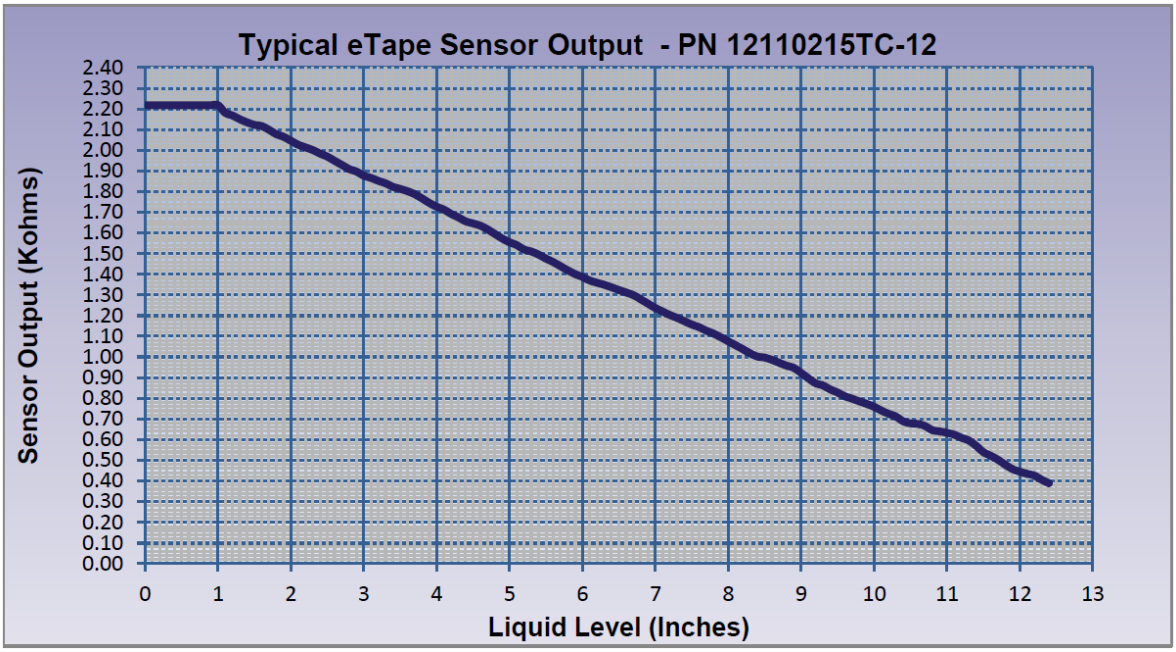
Exercise XIII.6: Add a pushbutton to each of your crickets. If the crickets are currently emitting tones, pressing the pushbutton on any cricket will immediately disable sound for all of the crickets. If the crickets are currently silenced, pressing the pushbutton on any cricket will immediately enable sound for all of the crickets. When silenced, the crickets will continue to count.

Lab XIV
Mapping Sensors

In this lab we will measure the level of water using a MILONE eTape Continuous Fluid Level Sensor. The main part of this lab is to read the sensor accurately. Once we have this information, we can use it for a number of different applications such as monitoring the later level remotely using an XBee wireless link (see Lab XIII), a basement sump alarm if the water level becomes too high, or a dry water alarm for when the water level in a bucket out in a pasture becomes too low.

A. MILONE eTape Continuous Fluid Level Sensor

The sensor we will use is the MILONE eTape Continuous Fluid Level Sensor [9]. A data sheet is provided in the Appendix XVI.F on page 282. This is a resistive sensor where the value of resistance varies linearly with the level of the fluid. A plot of the resistance versus level for the 12-inch sensor is shown below [10]:



The graph shows that the resistance decreases linearly with increasing fluid level. We also note that the sensor comes in different lengths (8 to 32 inches):

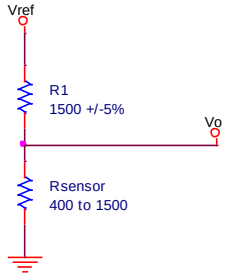
- **Specifications**

Part Number	PN-12110215TC-8	PN-12110215TC-12	PN-12110215TC-24	PN-12110215TC-32
Nominal Length	**8-inch**	**12-inch**	**24-inch**	**32-inch**
Sensor Length	10.2" (259 mm)	14.2" (361 mm)	26.0" (660 mm)	34.2" (869 mm)
Active Length	8.4" (213 mm)	12.4" (315 mm)	24.34" (618 mm)	32.4" (823 mm)
Sensor Output	400-1500Ω ±20%	400-2000Ω ±20%	400-3000Ω ±20%	400-5000Ω ±20%
Ref Resistance	1500Ω ±20%	2000Ω ±20%	3000Ω ±20%	5000Ω ±20%

[9] Information about this sensor is available at http://www.milonetech.com/About_eTape.php.

We are using the 8-inch sensor, so the resistance of this sensor varies from 400 Ω to 1500 Ω as the fluid level changes. Note that the tolerance of the sensor resistance is ±20%, which is a rather large tolerance. Not show here, but the sensor resistance is also a function of temperature. All of this tolerance van be eliminated by using the **Ref**erence **Resistance** that is part of the sensor. This is a constant 1500 Ω resistor that is part of the sensor. It has the same tolerance and temperature dependence as the Sensor Output resistance. For the sake of this example, we will assume that the two dependencies track one another. (They have the same temperature dependence, they are both at the same temperature, and they both are off from their nominal value due to tolerance variation by the same amount.)

Now that we have this information we can use several different circuits to measure the water level. We will start with the simple circuit below:



In this example, R1 is a standard 5% resistor (not the one that comes built-in to the sensor) and the reference voltage is a DC voltage that we will provide from the Arduino. We remember that the sensor resistance includes tolerance and temperature dependence which we will include as unknown functions:

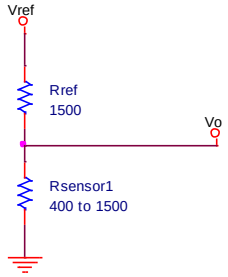
$$Sensor\ Resistance = Rsensor[1 + f(Tol)][1 + f(T)]$$

Where f(Tol) is the resistance variation due to tolerance. We don't know by how much a specific sensor is off from the nominal value due to tolerance, and thus we specify the tolerance as an unknown function f(Tol). We do know that it has an upper limit of 0.2 and a lower limit of -0.2. Similarly, the sensor resistance varies due to temperature, and this is included as the unknown function f(T). Note that reference resistor is a separate 5% resistor that we found in a bin somewhere, so its tolerance and temperature dependence are completely unrelated to the fluid sensor resistance. For sake of argument, we will just assume that R1 is constant at 1500 Ω.

Now that we know the sensor resistance, we can come up with an equation for the output voltage of our sensor circuit:

$$Vo = \frac{Vref \cdot Rsensor[1 + f(Tol)][1 + f(T)]}{R1 + Rsensor[1 + f(Tol)][1 + f(T)]}$$

We see that the sensor output is completely dependent on temperature and tolerance. This is not a good method. We can fix these two problems by using the reference resistor that is built-in to the sensor. This is a constant resistance, but it has the same tolerance and temperature dependence as the sensor resistance



The reference resistance is:

$$Reference\ Resistance = Rref[1 + f(Tol)][1 + f(T)]$$

Note that the tolerance function and temperature function are the same as for the senors. The output voltage equation now becomes:

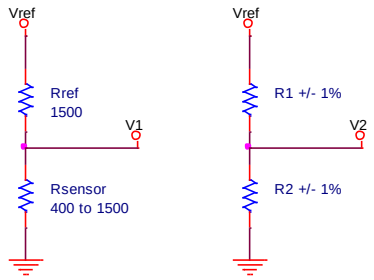
$$Vo = \frac{Vref \cdot Rsensor[1 + f(Tol)][1 + f(T)]}{Rref[1 + f(Tol)][1 + f(T)] + Rsensor[1 + f(Tol)][1 + f(T)]}$$

The tolerance and temperature dependence cancels out and the equation reduces to:

$$Vo = \frac{Vref \cdot Rsensor}{Rref + Rsensor}$$

Thus, making use of the reference eliminates the effects of device tolerance and temperature dependence.

We have neglected one other tolerance. In our implementation, Vref will be provided by the 5 V supply of the Arduino. This is never really 5 V and it can have a large tolerance and may vary with time and temperature. We can eliminate the effect of variations in Vref by measuring Vref. We cannot measure Vref directly because the A/D inputs of the Arduino read a maximum of 5 V, and might not accurately measure small variations at this extreme. Thus, we will use the circuit below:



We will assume that R1 and R2 are very accurate resistors, are the same temperature, and have the same temperature variation. Thus, we will not include these factors in our analysis. This is not strictly true and is a second order effect. You can research this problem further if you need more accuracy.

The output voltage V1 is the same as Vo above and free of temperature and tolerance dependence:

$$V1 = \frac{Vref \cdot Rsensor}{Rref + Rsensor}$$

The voltage V2 is:

$$V2 = \frac{Vref \cdot R2}{R1 + R2}$$

If we divide V1 by V2 we can eliminate the dependence on Vref:

$$\frac{V1}{V2} = \frac{\frac{Vref \cdot Rsensor}{Rref + Rsensor}}{\frac{Vref \cdot R2}{R1 + R2}} = \frac{Rsensor}{Rref + Rsensor} \cdot \frac{R1 + R2}{R2}$$

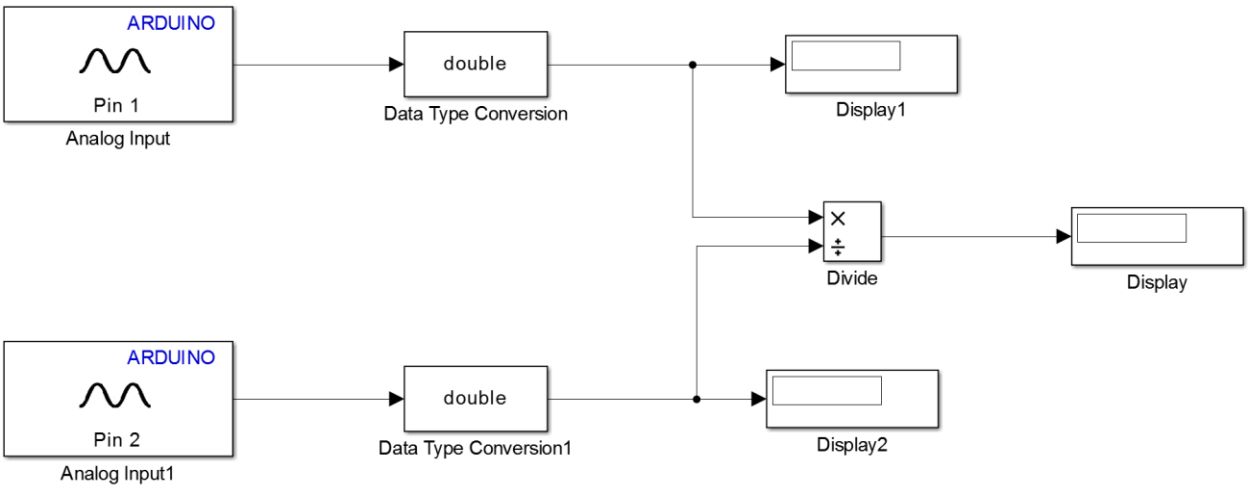
If we assume that R1 ≈ R2 (they have the same resistance value, temperature dependence and are at the same temperature, and have the same tolerance and are off by the same amount), the equation reduces to

$$\frac{V1}{V2} = 2 \cdot \frac{Rsensor}{Rref + Rsensor}$$

We have thus eliminated temperature and tolerance variations[10]. For our 8-inch sensor, Rref is 1500 Ω and Rsensor varies from 400 Ω to 1500 Ω. Thus, we expect the ratio to vary from 0.21 to 1, corresponding to a fluid level of 8 inches down to 0 inches.

For numerical values of R1 and R2, use a value of 10 K or so. Although we only care about the ratio of R1 and R2, using larger values will reduce power consumption.

We will measure the ratio of V1/V2 using the Simulink model shown below:



Important Note: To get accurate reading from the fluid sensor, or for any sensor, you must use the sensor properly. For this fluid sensor, you must make sure that you mount it properly. The data sheet specifies the following mounting instructions: ***Suspend the eTape sensor in the fluid to be measured. To work properly the sensor must remain straight and must not be bent vertically or longitudinally. For best results install the sensor inside a section of 1-inch diameter PVC pipe. Double sided adhesive tape may be applied to the upper back portion of the sensor to suspend the sensor in the container to be measured. However, the liquid must be allowed to interact freely with both sides of the sensor. The vent hole located above the max line allows the eTape to equilibrate with atmospheric pressure. The vent hole is fitted with a hydrophobic filter membrane to prevent the eTape from being swamped if inadvertently submerged.*** [10]

Demo XIV.1: Show that the ratio V1/V2 varies between approximately 0.2 and 1 as the level varies between the maximum and minimum.

Exercise XIV.1: Obtain a table of measured values for the water level and ratio V1/V2 for your sensor. Fill out the table below:

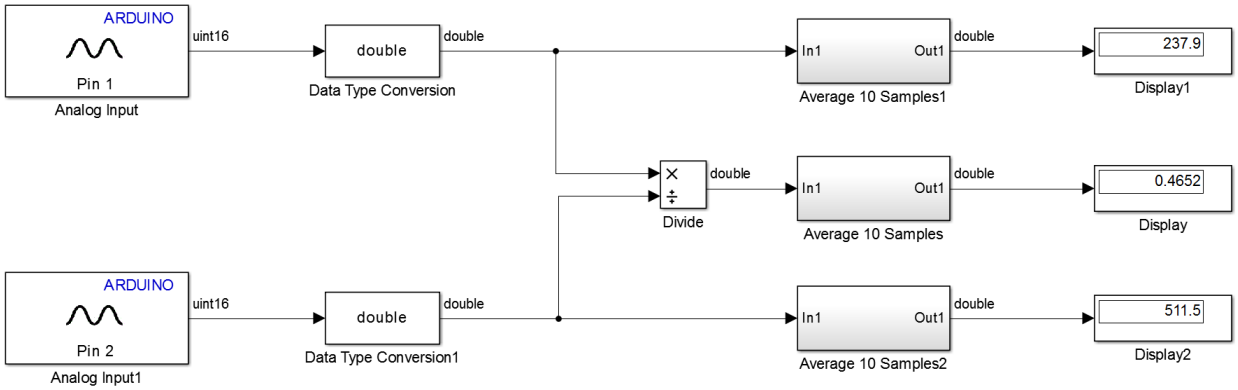
Water Level (Inches)	V1/V2	Water Level (Inches)	V1/V2	Water Level (Inches)	V1/V2	Water Level (Inches)	V1/V2
0		2		4		6	
0.5		2.5		4.5		6.5	

[10] We know that this is not quite true because R1 and R2 have tolerance and temperature dependence. However, we are using 1% resistors and placing them physically close together.

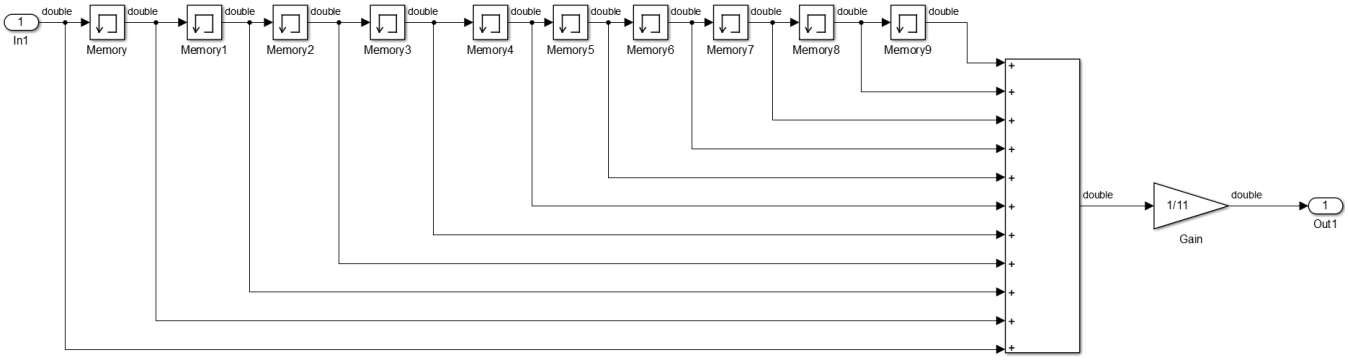
1		3		5		7	
1.5		3.5		5.5		7.5	
						8	

Generate a MATLAB plot of your results.

While taking the data in the exercise above, you may have noticed two things. First, it would be nice to know the raw values of V1 and V2 for our data because it is hard to see when the ration begins to change significantly. We can fix that by recording those values and generating appropriate plots. Second, the values displayed by the **Display** blocks jump around too much to get an accurate reading. We will fix this by averaging several samples together. We will use the modified model shown below:



Notice that the model contains three copies of a new subsystem called Average 10 Samples. (It should be Average 11 Samples, as you will soon see.) This subsystem is shown below:



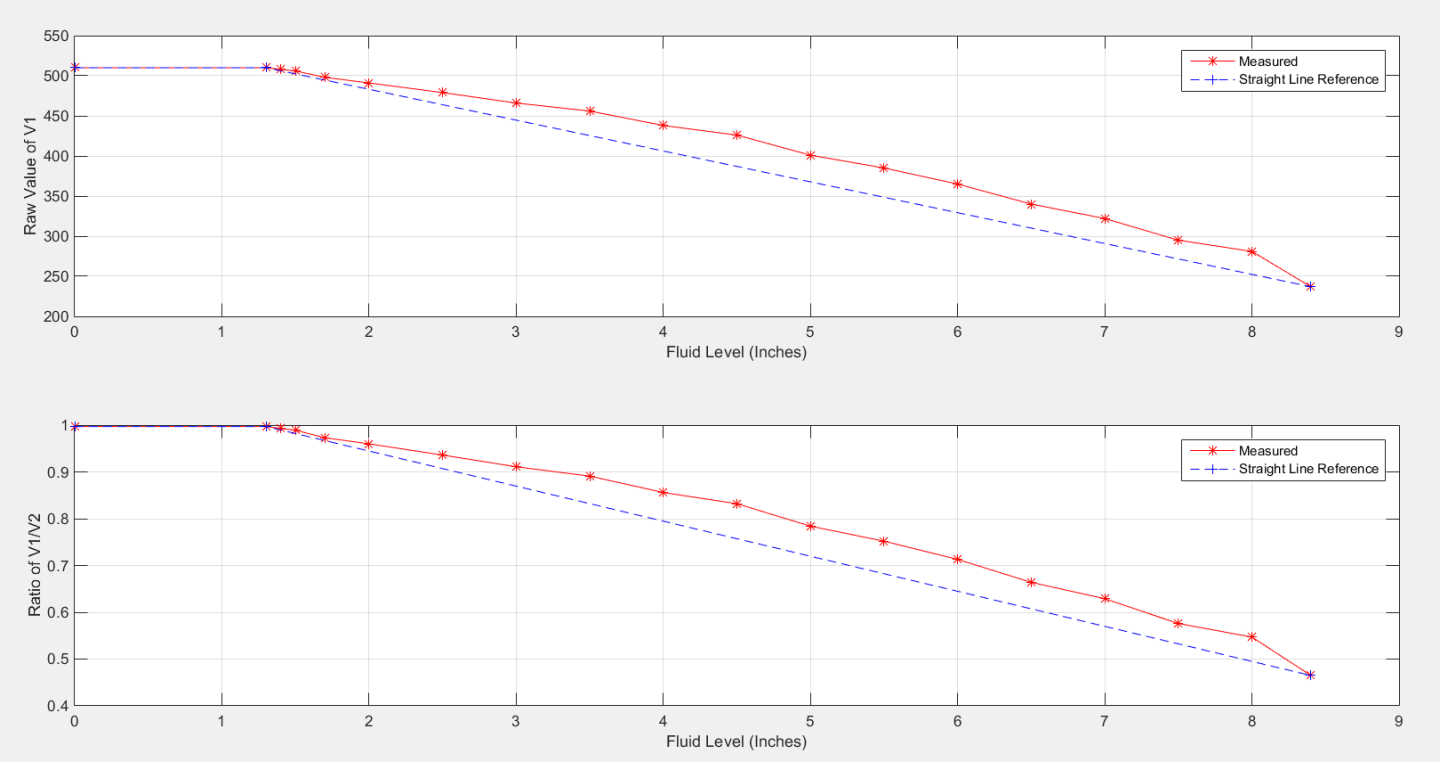
This subsystem contains 10 Memory blocks. Remember that the output of a memory block is its input from the previous time step. You can also think of this block as a one-time step delay. For in the input to reach the output of the last memory block takes ten time steps. Also, the present value of the input and the last 10 inputs are stored in the chain of memory blocks. To take a time average, all we need to do is sum the input (In1) and outputs of the memory blocks and then divide by the number of samples. In this sum, we are including the present input and the past 10 samples. Thus, we will add these 11 values together with a **Sum** block and then divide by 11 to get a time average of the input over the most recent 11 input values. This will remove most of the jumping around we saw on the display. Note that for 10 memory blocks you will need to wait for 10 fixed time steps before the average is valid after a change in the fluid level. With a fixed time step of 0.1 seconds, as I used in my model, this works out to one second, not too long for a human observer collecting data. You can use more than ten memory blocks if you want a slower average.

Exercise XIV.2: Use the improved model to obtain a table of measured values for the water level, and ratio V1/V2, the raw value for V1, and the raw value of V2 for your sensor. Fill out the table below:

	V2 =							
Water Level (Inches)	V1	V1/V2	Water Level (Inches)	V1	V1/V2	Water Level (Inches)	V1	V1/V2
0			3			6		
0.5			3.5			6.5		
1			4			7		
1.5			4.5			7.5		
2			5.0			8		
2.5			5.5			8.4		

B. Sensor Decoding – Lookup Table

The measured data from the previous exercise is shown below. The two plots are shown, one for the raw value of V1 and one for the ratio V1/V2. Also shown are the straight lines for the sensor output that we had hoped for.



The MATLAB code used to create this figure is shown below:

```
% Measured Data for Lab 14
Level = [0 1.3 1.4 1.5 1.7 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.4];
Level_Line = [0 1.3 8.4];
V1_Line = [510 510 237];
V2 = 511.5;
V1 = [510 510 508 506 498 491 479 466 456 438 426 401 385 365 340 322 295 281
237];
```

```
V1rV2 = 0.001*[997 997 993 989 973 960 936 911 891 856 832 784 752 713 664 629
576 547 465];
V1rV2_Line = 0.001*[997 997 465];
subplot(2,1,1)
plot(Level, V1, '-r*', Level_Line, V1_Line, '--b+');
xlabel('Fluid Level (Inches)');
ylabel('Raw Value of V1')
legend('Measured', 'Straight Line Reference');
grid on
subplot(2,1,2);
plot(Level, V1rV2, '-r*', Level_Line, V1rV2_Line, '--b+');
xlabel('Fluid Level (Inches)');
ylabel('Ratio of V1/V2')
legend('Measured', 'Straight Line Reference');
grid on
```

Important note: Your data may be different than that show in the plots above. The difference may be due to different mounting methods or other environmental conditions that are not controlled.

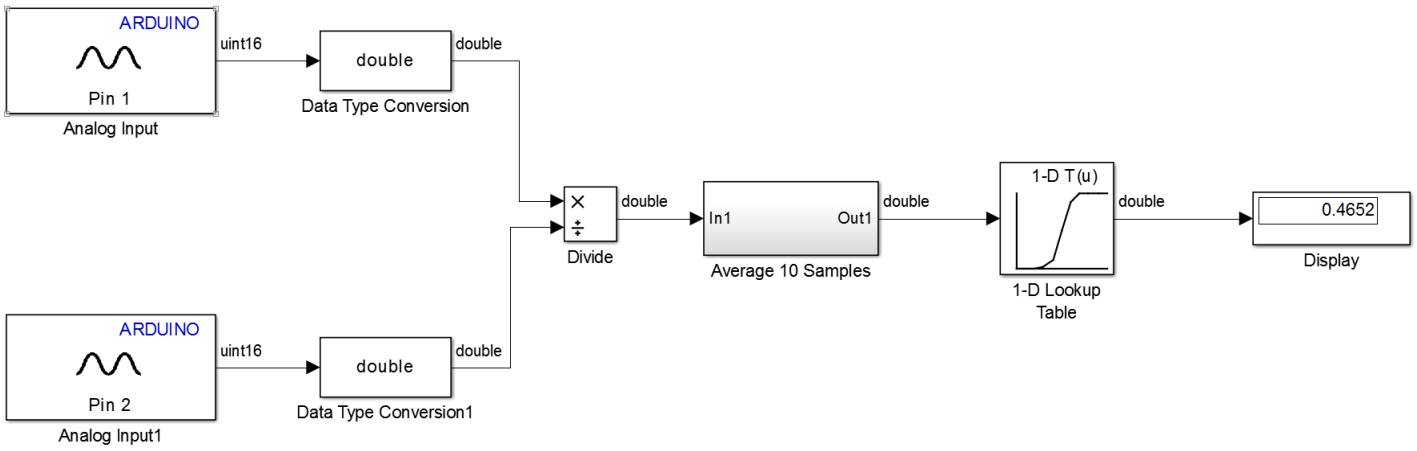
We see that the sensor output is not quite linear. It is not that non-linear and we could fit a straight line to it if needed or desired. However, we shall use a more powerful idea. We have measured data that directly relates sensor output to fluid level. We can place this data in a one-dimensional lookup table with the sensor values as the input to the table. The output of the lookup table will be the fluid level data. In the script above, we have already created the arrays we can use in the lookup table:

```
V1rV2 = 0.001*[997 997 993 989 973 960 936 911 891 856 832 784 752 713 664 629
576 547 465];
```

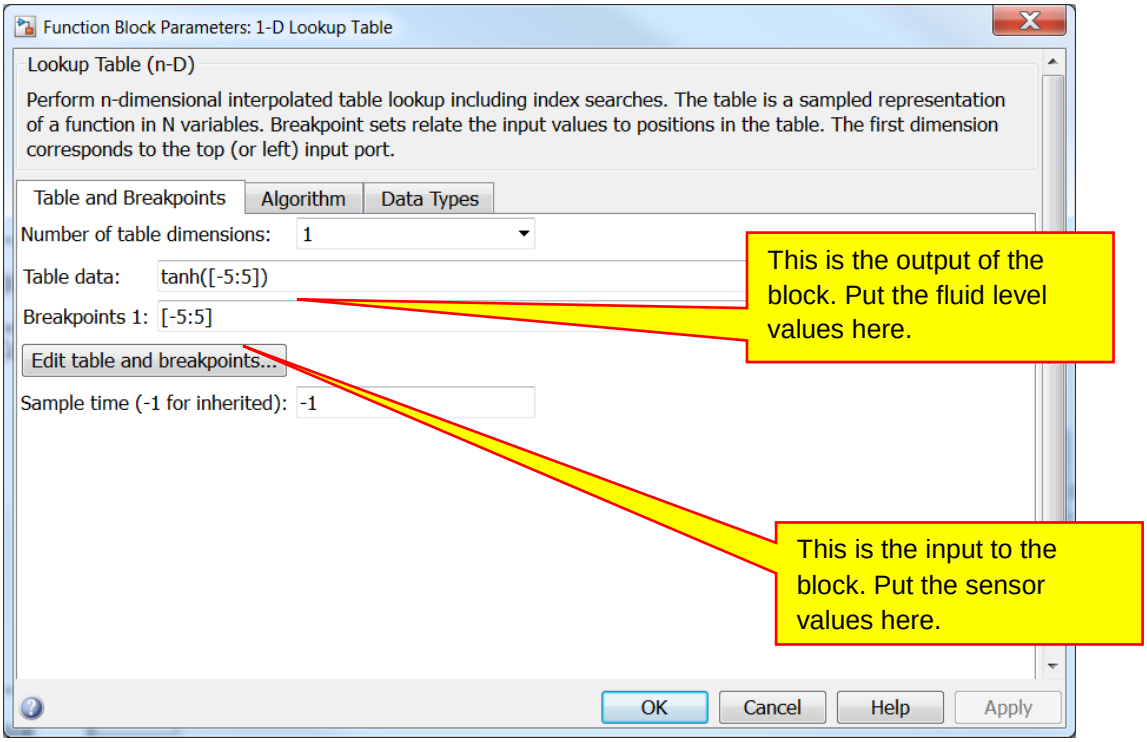
These two values, which are the same, will be a problem.

```
Level = [0 1.3 1.4 1.5 1.7 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.4];
```

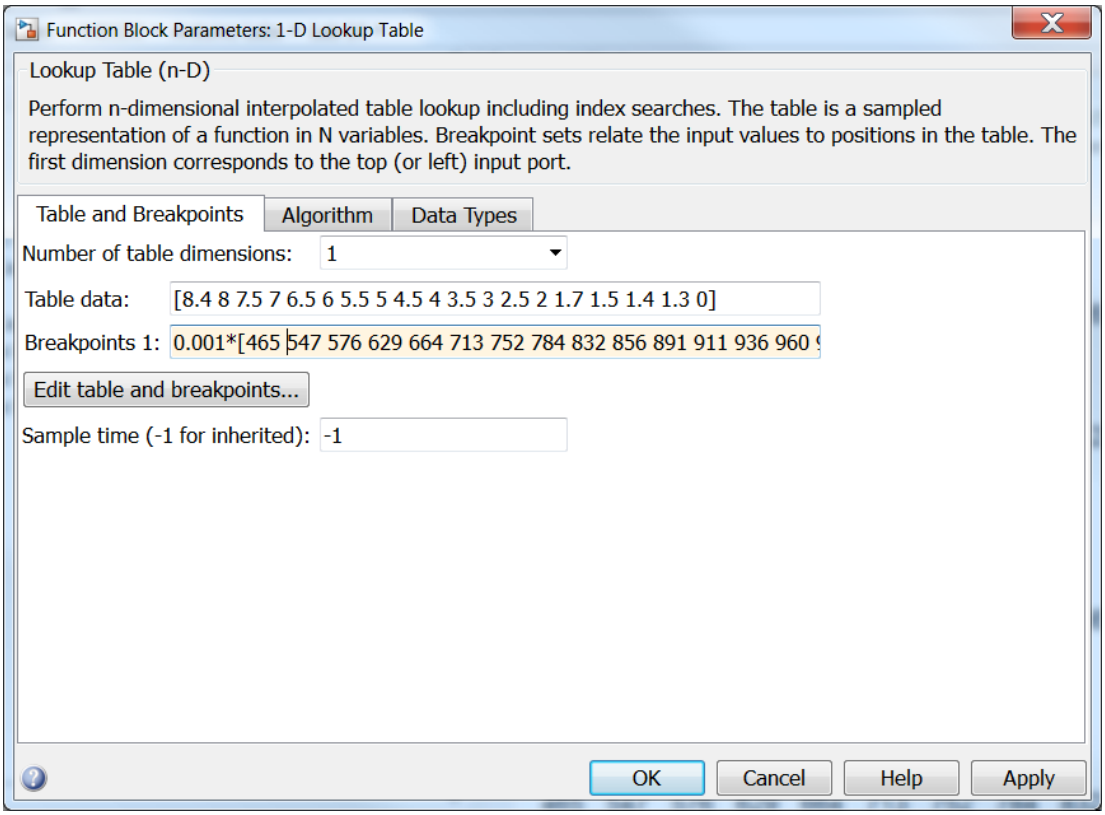
Place a **1-D Lookup Table** in you model as shown:



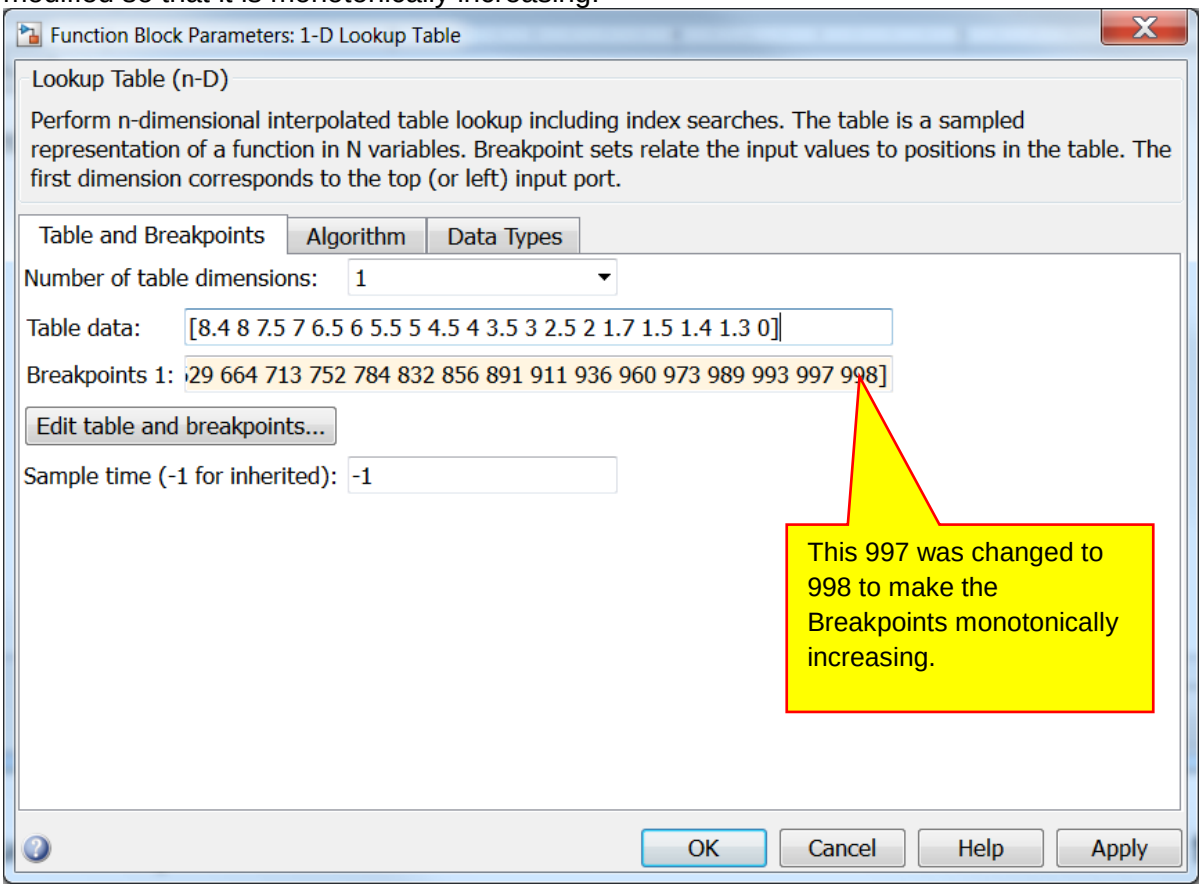
Double-click on the 1-D Lookup Table block to open it:



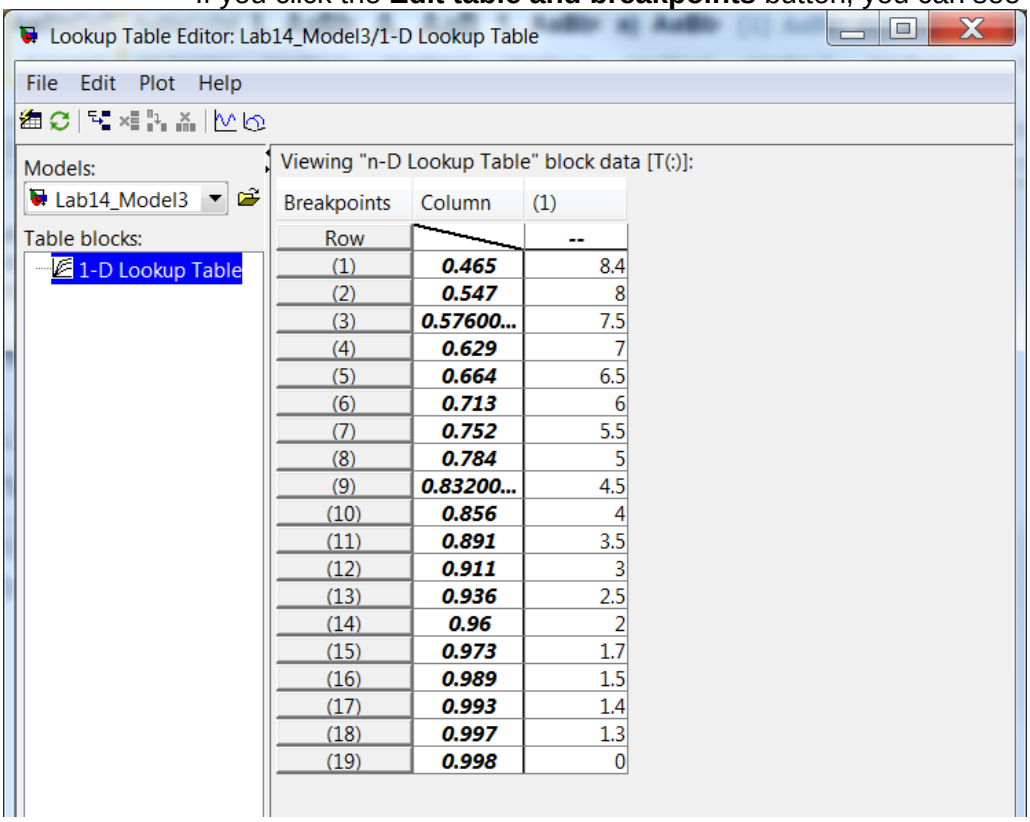
The **Breakpoints** is the array of data for the input to the table which, for our case, is the sensor data. An important note is that the Breakpoints must go from lowest to highest and must not repeat (monotonically increasing). Thus, we need to change the order of the data we used in creating the plots shown previously. The **Table Data** is the output of the block, which is the fluid level for this example. Fill in the fields with you measured data as shown:



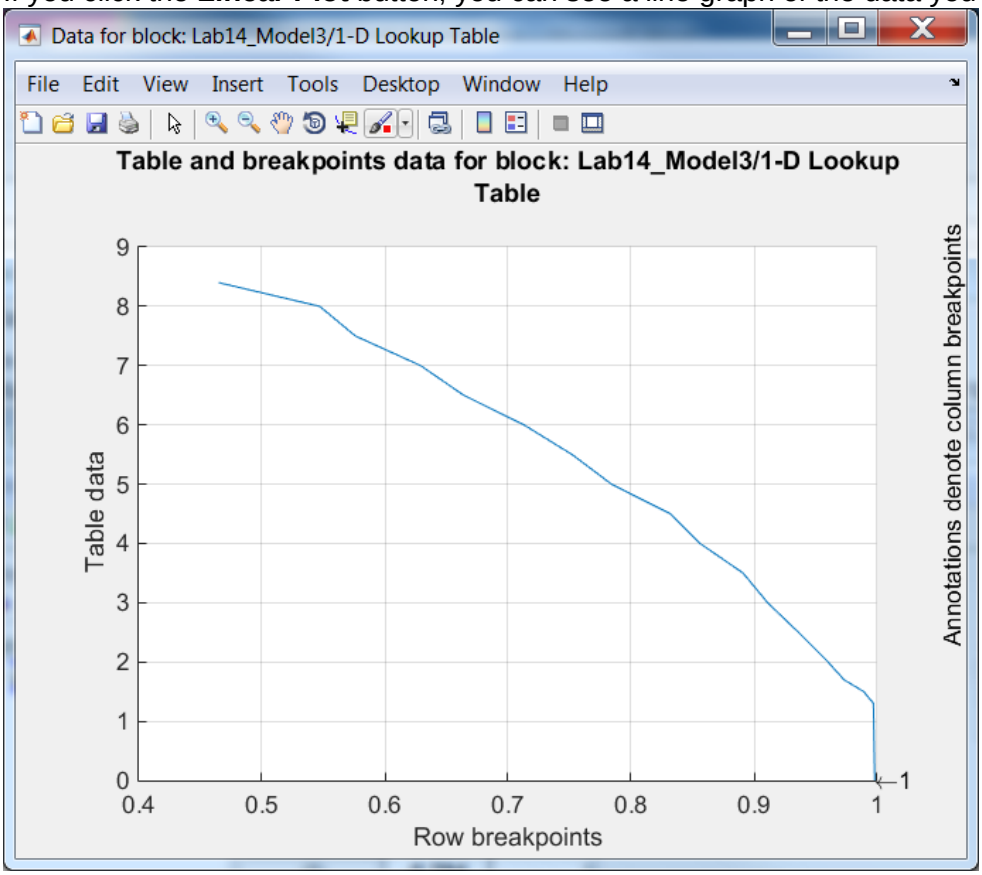
The data must be enclosed in square brackets. Also note that the data in the Breakpoints array must be modified so that it is monotonically increasing:



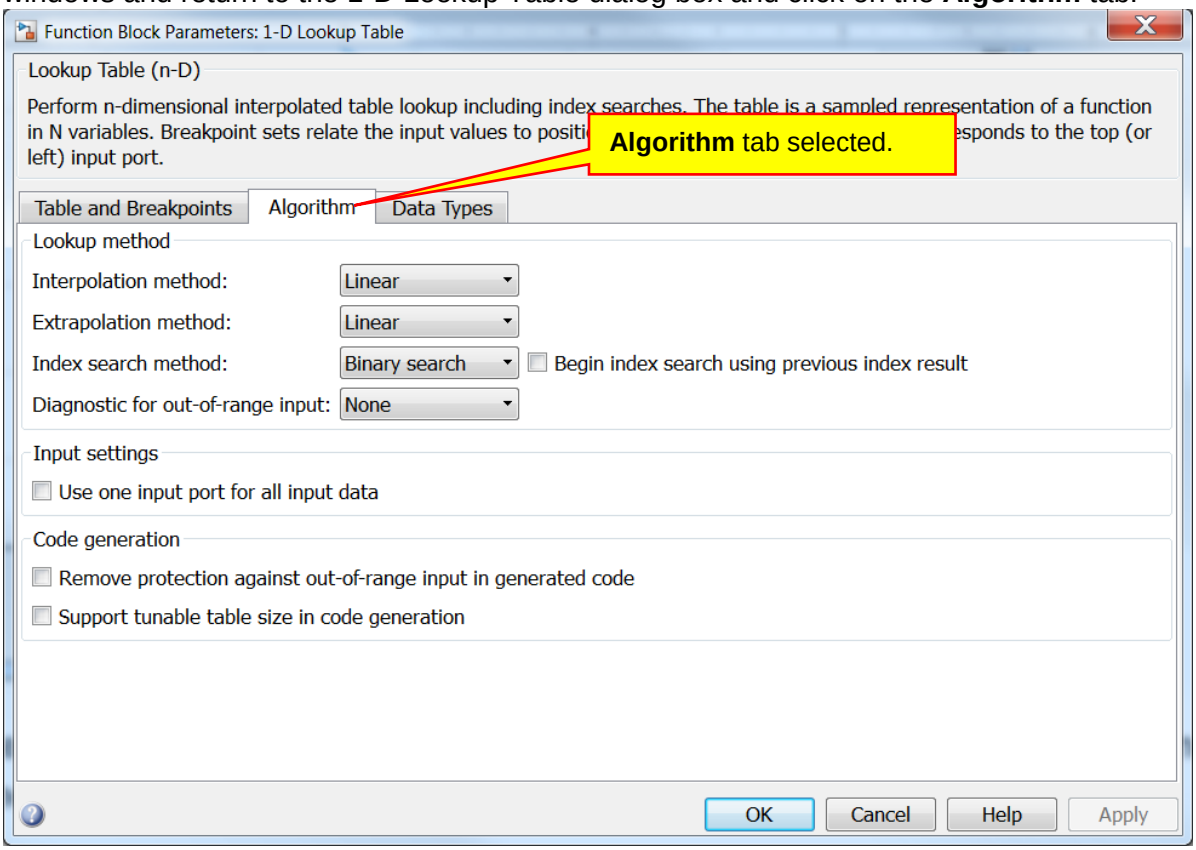
If you click the **Edit table and breakpoints** button, you can see the data in a tabular form:



If you click the **Linear Plot** button, you can see a line graph of the data you entered:

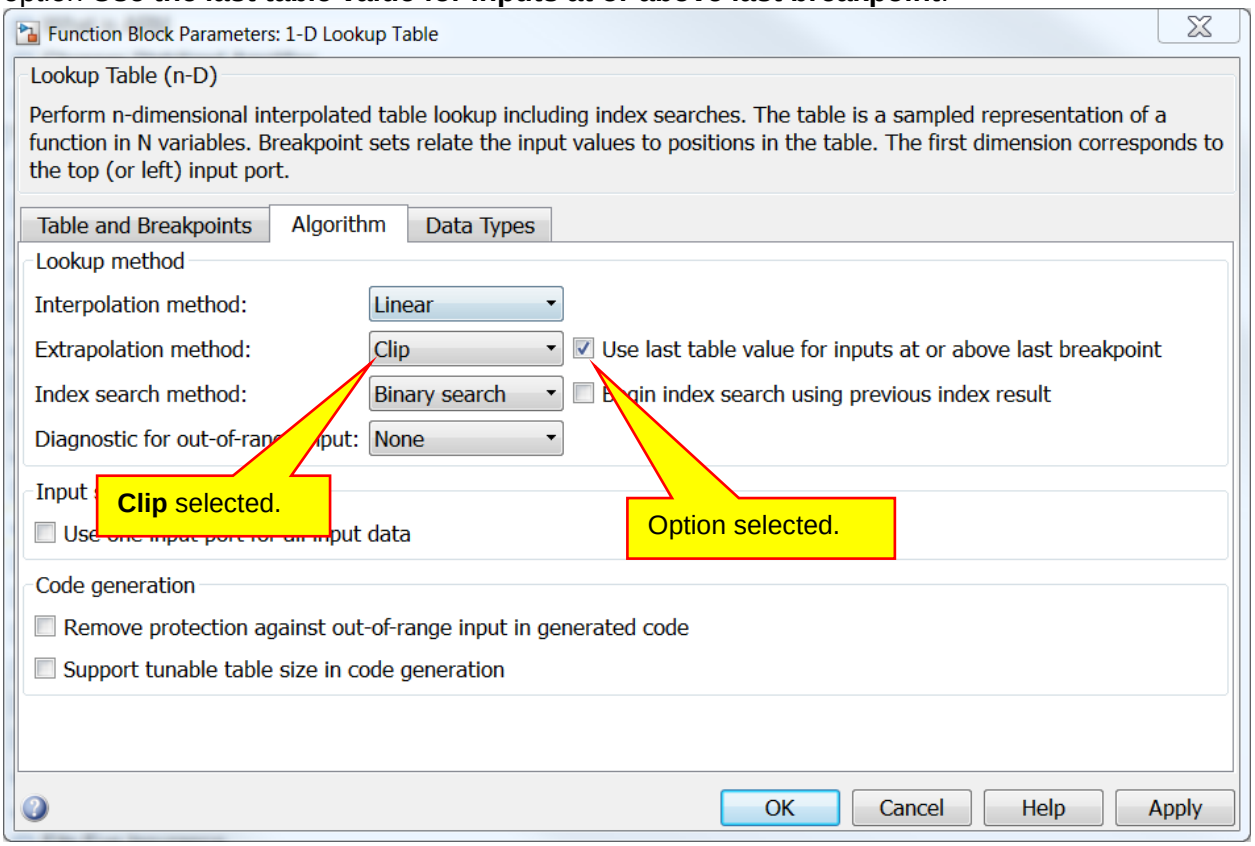


Use these two tools to verify that you entered your data correctly. When done, close the plot and the table windows and return to the 1-D Lookup Table dialog box and click on the **Algorithm** tab:



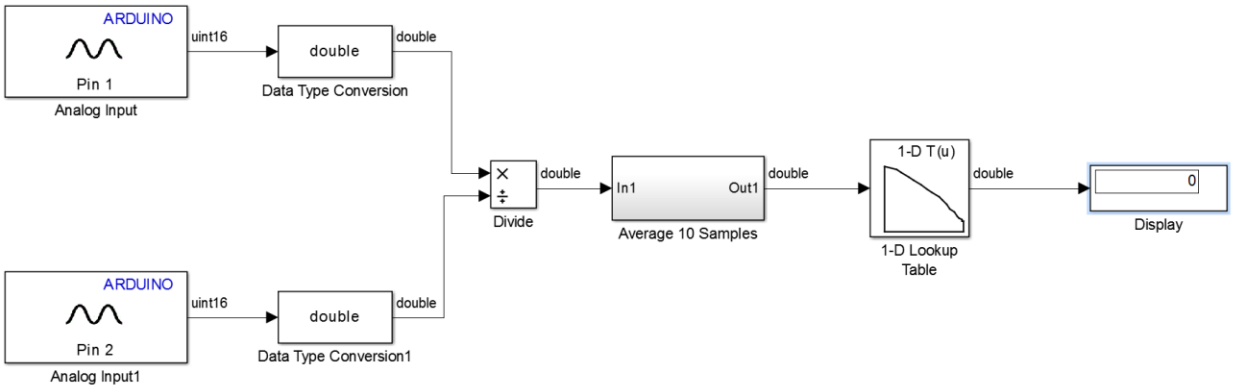
The interpolation method is how Simulink calculates the output value when the input is between two of the points in the table. The selected method is **Linear** which means it assumes a straight line between two points, and the output value is calculated from the line. You can also use a cubic spline if you want Simulink to fit a curve between the two points. There are other methods from which to choose. However, since we have somewhat sparse data, make sure that you select either Cubic spline or Linear.

The **Extrapolation method** is how Simulink will deal with data outside the limits of the table. If the sensor was a well know function, we could use this method to extend the table beyond the values specified in the table. In our case, the sensor no longer functions outside the range specified in the table. Thus we want to clip the data. If the V1/V2 ration is greater than 0.998 we will use the last valid value in the table, which is a level of 0. If the value of V1/V2 is less than 0.465 we will use the last valid value in the table, which is 8.4 inches. This makes sense as the sensor is only 8.4 inches and we cannot measure less than 0 and more than 8.4 inches. To implement this method, we need to specify the **Extrapolation method** as **Clip** and select the option **Use the last table value for inputs at or above last breakpoint**:



Click the OK button to save the changes.

We are now ready to demonstrate the complete model shown below:



Demo XIV.2: Show that the output varies from 0 to 8.4 as the water level varies between 0 and 8.4 inches.

C. Fluid Level Projects

Now that we have a way to measure fluid level, we can use the sensor to facilitate some useful projects. These projects are listed as exercises below. You should be able to solve the logic needed in these exercises by using the techniques shown in previous labs.

Exercise XIV.3: Basement Sump Alarm. Create a model that monitors the water level of a basement sump with the following functions:

- There is one setting, which is the overflow level. For our 8.4 inch sensor, this level should be between 1.5 and 8.4 inches. The overflow level is different than the sensor maximum level. It should be less than the sensor max reading as this sensor should not be submerged.
- The water level is displayed on the LCD. If the water is below the minimum level, the LCD will display the level as “LOW.” If the water level is above the overflow level, the LCD will display the level as “High.” If the water level is between the overflow and minimum levels, the LCD will display the number of inches remaining until the overflow level is reached.
- Different alarms will sounds under various conditions:
    - The alarm will chirp (much like a smoke detector chirps when the battery is low) when the water level is between one and two inches less than the overflow level. This slower chirping will continue to sound until the water drops 0.5 inches below the slow chirp level, in which case the sound will stop.
    - The alarm will chirp at twice the frequency of the slow chirp alarm when the water level is within one inch of the overflow level. This faster chirping will continue to sound until the water drops 0.5 inches below the fast chirp level level, in which case the slower chirping alarm will start again.
    - When the water level is at or above the overflow level, a continuous alarm will sound. This alarm will continue to sound until the water drops 0.5 inches below the overflow level, in which case the faster chirping alarm will start again.
- The audio alarm will be silenced for 15 minutes if the user presses a pushpbutton.

Exercise XIV.4: Modify the Basement Sump Alarm of Exercise XIV.3 so that the user can specify the overflow level using a pushbutton. The level should be adjustable in 0.5 inch steps. The overflow level should be displayed on the LCD screen.

Exercise XIV.5: Implement the Basement Sump Alarm of Exercise XIV.3 using an XBee wireless link. This will allow the sensor to be in the basement and the audible alarm to be located elsewhere so that the user can hear it.

Exercise XIV.6: Basement Sump Pump Controller – Create a controller that measures the water level in a sump and then turns on and off the pump appropriately.

- There are two user setpoints: High_Level and Low_Level.
- When the water level is above the High_Level, a relay is turned on which will turn on the sump pump.
- When the pump turns on, it will remain on until the level goes below the Low_Level.

Lab XV
Creating S-Functions

In previous sections, we used a number of hardware blocks to read sensors, write to the SD Card, and store memory in the EEPROM. All of these functions did not come with the standard Arduino Support Package and were created using the S-Function Builder block available in Simulink. The RHIT library provides a number of sensor blocks that were created using the method presented here. We assume, however, that a function or sensor that you need may not be in the library and that you will need to develop a Simulink block to use that specific function. The method presented here will cover the procedure used to create s-functions and to add it to the library. The Arduino is very popular and forums exist where people post code for using sensors, using hardware aspects of the Arduino not available yet in Simulink, or have developed code that solves a problem you wish to solve. It is assumed that you have found or developed the c or c++ code for your application. Here we will just show how to use that code in a Simulink model for the Arduino Mega. We will demonstrate this example for a TC74 temperature sensor[11].

A. TC74 Datasheet

The first thing we will look at is the datasheet for the TC74. This device uses an I2C bus, so we need to find the address for the device:

MICROCHIP

TC74

Tiny Serial Digital Thermal Sensor

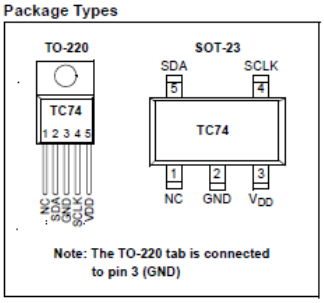
Features

- Digital Temperature Sensing in SOT-23-5 or TO-220 Packages
- Outputs Temperature as an 8-Bit Digital Word
- Simple SMBus/I²C™ Serial Port Interface
- Solid-State Temperature Sensing:
  - ±2°C (max.) Accuracy from +25°C to +85°C
  - ±3°C (max.) Accuracy from 0°C to +125°C
- Supply Voltage of 2.7V to 5.5V
- Low Power:
  - 200 µA (typ.) Operating Current
  - 5 µA (typ.) Standby Mode Current

Applications

- Thermal Protection for Hard Disk Drives and other PC Peripherals
- PC Card Devices for Notebook Computers
- Low Cost Thermostat Controls
- Power Supplies
- Thermistor Replacement

Package Types



Note: The TO-220 tab is connected to pin 3 (GND)

General Description

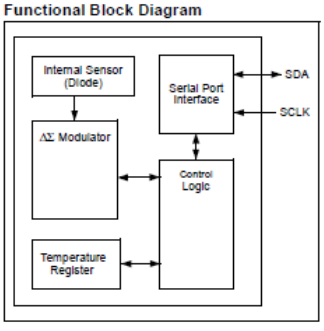
The TC74 is a serially accessible, digital temperature sensor particularly suited for low cost and small form-factor applications. Temperature data is converted from the onboard thermal sensing element and made available as an 8-bit digital word.

Communication with the TC74 is accomplished via a 2-wire SMBus/I²C compatible serial port. This bus also can be used to implement multi-drop/multi-zone monitoring. The SHDN bit in the CONFIG register can be used to activate the low power Standby mode.

Temperature resolution is 1°C. Conversion rate is a nominal 8 samples/sec. During normal operation, the quiescent current is 200 µA (typ). During standby operation, the quiescent current is 5 µA (typ).

Small size, low installed cost and ease of use make the TC74 an ideal choice for implementing thermal management in a variety of systems.

Functional Block Diagram



[11] Many thanks to JM Modisette of the MathWorks for helping develop the files and procedures presented in this section.

Datasheets for the TC74 are readily available on-line. For this lab we will be using the TO-220 package as it is large enough for us to easily use and test on a breadboard.

Note that the TC74 has eight possible addresses. Multiple addresses are available so that we can use several TC74 temperature sensors on the same I2C bus. We will make the address an input to our S-Function so that we can make the block more flexible. The addresses are specified as part of the part number:

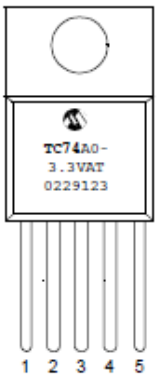
SOT-23 Package Marking Codes

SOT-23 (V)	Address	Code	SOT-23 (V)	Address	Code
TC74A0-3.3VCT	1001 000	V0	TC74A0-5.0VCT	1001 000	U0
TC74A1-3.3VCT	1001 001	V1	TC74A1-5.0VCT	1001 001	U1
TC74A2-3.3VCT	1001 010	V2	TC74A2-5.0VCT	1001 010	U2
TC74A3-3.3VCT	1001 011	V3	TC74A3-5.0VCT	1001 011	U3
TC74A4-3.3VCT	1001 100	V4	TC74A4-5.0VCT	1001 100	U4
TC74A5-3.3VCT	1001 101*	V5	TC74A5-5.0VCT	1001 101*	U5
TC74A6-3.3VCT	1001 110	V6	TC74A6-5.0VCT	1001 110	U6
TC74A7-3.3VCT	1001 111	V7	TC74A7-5.0VCT	1001 111	U7

Note: * Default Address

TO-220 Package Marking Information

TO-220



Legend:	XX...X	Customer specific information*
	YY	Year code (last 2 digits of calendar year)
	WW	Week code (week of January 1 is week '01')
	NNN	Alphanumeric traceability code

Note: In the event the full Microchip part number cannot be marked on one line, it will be carried over to the next line thus limiting the number of available characters for customer specific information.

* Standard marking consists of Microchip part number, year code, week code, and traceability code.

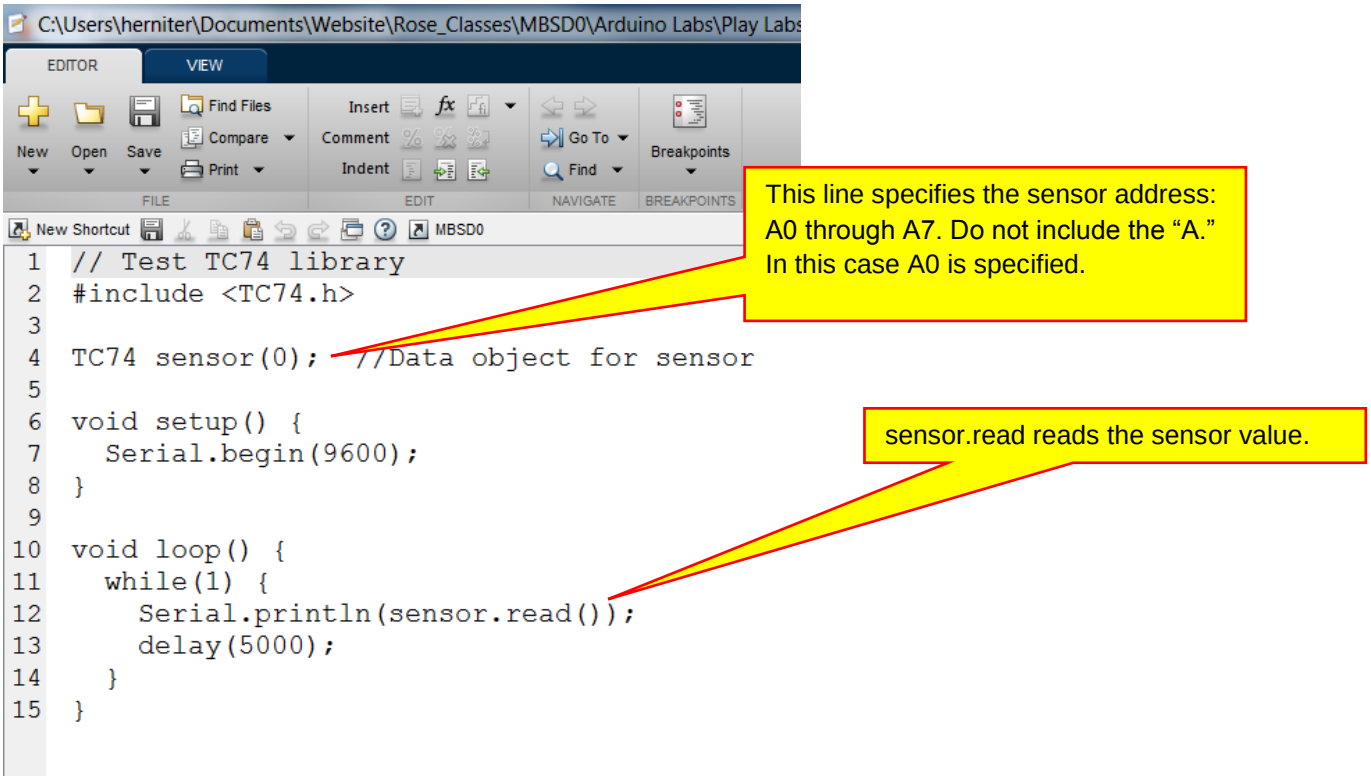
Note that the addresses range from 48 (hex) to 4F (hex), or 72 (dec) to 79 (dec).

B. Examining Previously Developed Code

Next, we need to find some code for the TC74. Search the web for "Arduino TC74." It is best to find ones in the Arduino forum as they contain code examples and libraries developed for the Arduino. I found a file named TC74_V1_00.zip located at http://code.google.com/p/tc74-library-for-arduino/source/browse/TC74.cpp. This file contains examples and a library that we can use.[12]

The readme file, shown below, tells us how to use the library and obtain data from the sensor:

[12] The downloaded files were created by Paul Jenkins, December 16, 2011 and released into the public domain.



We will not use the TC74.h or the TC74.cpp libraries. However, we will study the code and copy the parts that read the sensor. For a single sensor with one address, you could just as easily use the code here without modification. However, we want to make a general block where we can specify the address, so we use the portions of the code that read the sensor.

The line

```
TC74 sensor(0);  //Data object for sensor
```

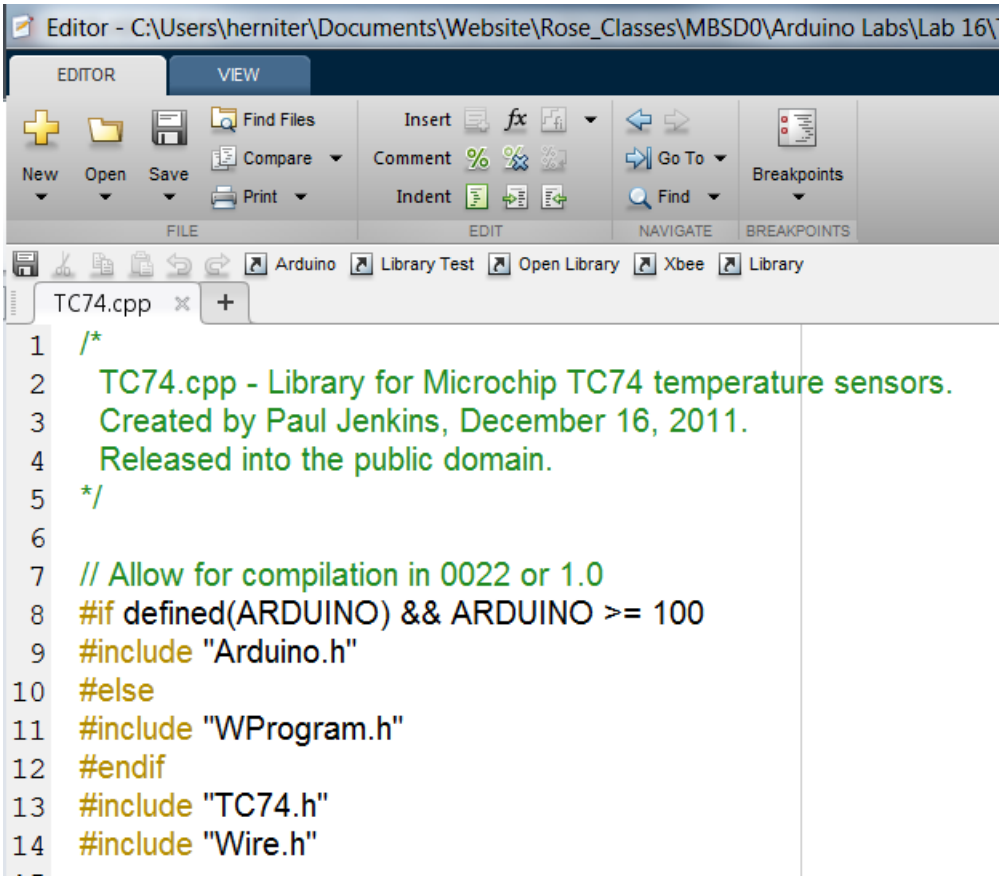
specifies the address of the sensor. 0 means A0, 1 means A1, 2 means A2, and so on. This line also specifies **`sensor`** as a TC74 object. The TC74 object is defined in the TC74.cpp library.

The other command we garner from the readme file is

```
sensor.read()
```

This line reads a numerical value from the sensor. If we use this code, these two commands would be all that we need. We will be generating a more general block, so we will look a little further.

If you look in the file named TC74.cpp, you will see the code below:



This tells us that we need to include the files Arduino.h, TC74.h and Wire.h in our s-function. We will not include the TC74.h as we will be copying the code and not using the functions defined in the TC74.cpp file. Note that if any includes are listed here, we need to place them in our s-function. **Important note:** If the files Wire.cpp and twi.c were listed here (they are not) you would **not** include them. This is because so many sensors use the I2C bus that Wire.cpp and twi.c would be listed in each sensor block and cause an error because they were defined more than once. To avoid this problem, we included these files in the RHIT_Library_Definition block. If your function includes the Wire.cpp and twi.c files, do not include them. Instead, place the RHIT_Library_Definition block in your model.

The next portion of the code calculates the address of the sensor:

```
//First, the constructor where an address is supplied
TC74::TC74(byte addr) {
  if (addr < 8)
    _addr = 0x48 + (addr & 0x07);  //Do the & in case parameter is malformed
  else
    _addr = 0x4d;  // Manufacturers default
}
```

This block calculates the address based on the zero (0) in the line `TC74 sensor(0)`. We will not need this calculation. However, we do need to know that the address (variable addr) is specified as a byte data type.

If you look up the byte data type at the Arduino website, you will find that it is an unsigned 8-bit data type. Thus, when we specify the address in Simulink, we will use type uint8.

The next portion of the code tells us how to read the sensor:

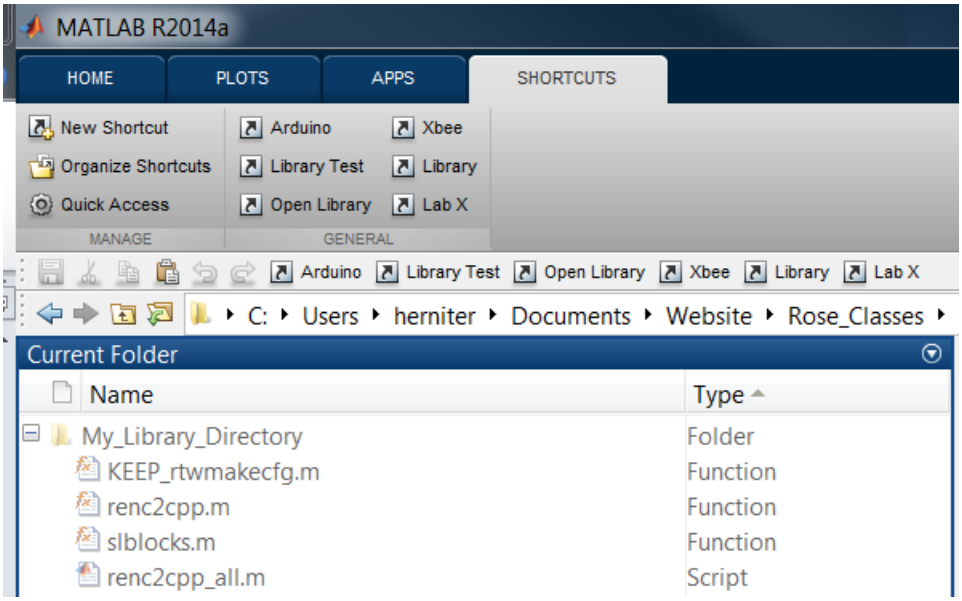
```
int TC74::read(void)
{
   int _temp = 255;
   // Get a reading from the temperature sensor
     Wire.beginTransmission(_addr);
     Wire.write((uint8_t) 0x01);
     Wire.write((uint8_t) 0x00);
     Wire.endTransmission();

     Wire.beginTransmission(_addr);
     Wire.write((uint8_t) 0);
     Wire.endTransmission();

     // request 1 byte from sensor
     Wire.beginTransmission(_addr);
     Wire.requestFrom((int) _addr, (int) 1);

     _temp = Wire.read();
           Wire.endTransmission();

     if (_temp > 127) {
       _temp = 255 - _temp + 1;
     }

     return _temp;
}
```

We will use this portion of the code.

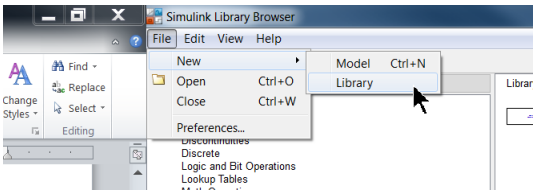
We will copy most of this code unchanged, except we will change _addr to our variable that specifies the address, and change _temp to our variable we use for the output. Note that the output (variable _temp) is of type int. Different Arduino boards define data type int differently, so you should verify the data type for the board you are using. In Simulink, we will use a variable of type int32 as it worked correctly for this application. Note that we will only use the lines enclosed in the dashed red box shown above. We will copy them when needed.

C. Creating a Simulink Library and Block

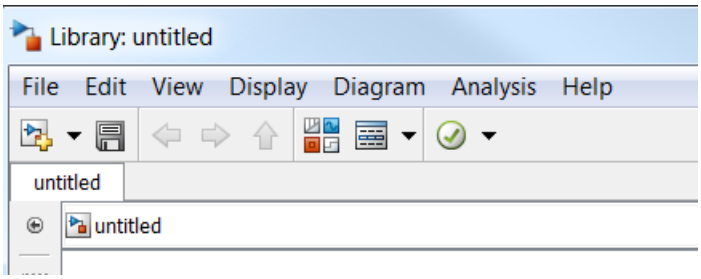
Next, we will show how to create a Simulink library. We will only create a single block for this library. However, you can add more blocks when needed. We will need to use several files as part of our library, so create a new directory called “My_Library_Directory.” From the RHIT library directory, copy the following files: KEEP_rtwmakecfg.m, renc2cpp.m, slblocks.m, renc2cpp_all.m. Your directory should look as shown:



Next, we will create a new library. From the Simulink library browser menus, select **File**, **New**, and then **Library**:



Notice that the window states that the new model is a Library:

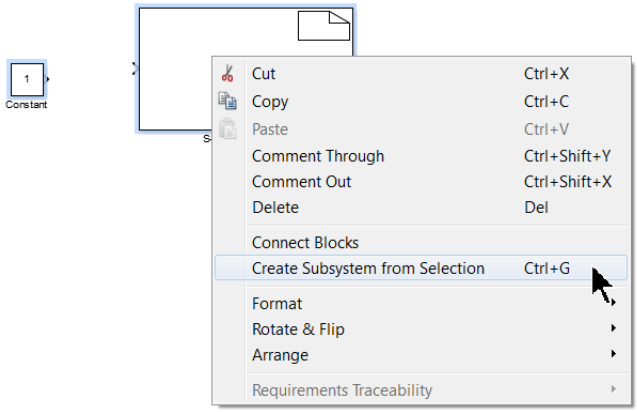


Save the model as “My_Library” in directory “My_Library_Directory.”

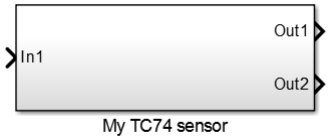
We will now create the model for the sensor. In the library, place the following parts:

- S-Function Builder – From Library Simulink / User Defined Functions
- Constant - From Library Simulink / Commonly Use Blocks

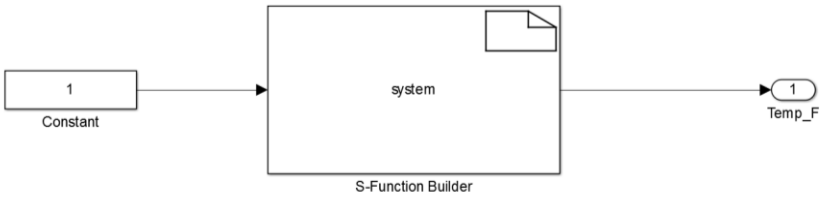
Next, select the two parts, then right click and select **Create Subsystem from Selection** to group the two parts into a subsystem:



Rename the newly created subsystem, “My TC74 Sensor”



Open the block. The constant is going to be an input to the S-Function_Builder. Plus, the sensor will only have a single output named Temp_F. Configure the subsystem as shown:



We will add more Simulink blocks to the subsystem later. For now, this is enough to get us started.

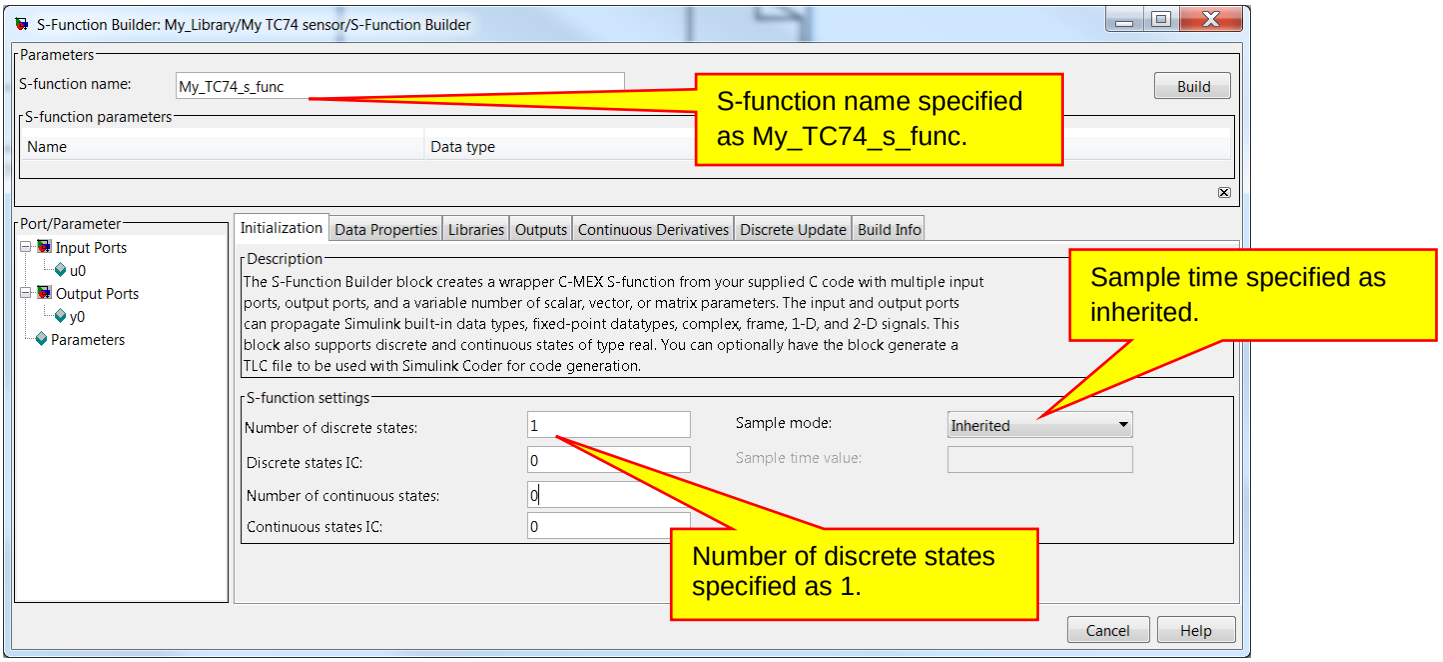
D. Creating the S-Function

We will now use the S-Function Builder to import the c-code we downloaded. Before we do this, we need to verify that we have a compiler setup and installed. In MATLAB, type the command **mex –setup**:

```
C:\Users\herniter\Documents\Website\Rose_Classes\MBSD0\Arduino Labs\Lab 16

>> cd 'C:\Users\herniter\Documents\Website\Rose_Classes\MBSD0\Arduino Labs\Lab 16'
>> mex -setup
MEX configured to use 'Microsoft Windows SDK 7.1 (C)' for C language compilation.
Warning: The MATLAB C and Fortran API has changed to support MATLAB
         variables with more than 2^32-1 elements. In the near future
         you will be required to update your code to utilize the
         new API. You can find more information about this at:
         http://www.mathworks.com/help/matlab/matlab_external/upgrading-mex-files-to-use-64-bit-api.ht

To choose a different language, select one from the following:
 mex -setup C++
 mex -setup FORTRAN
fx >>
```

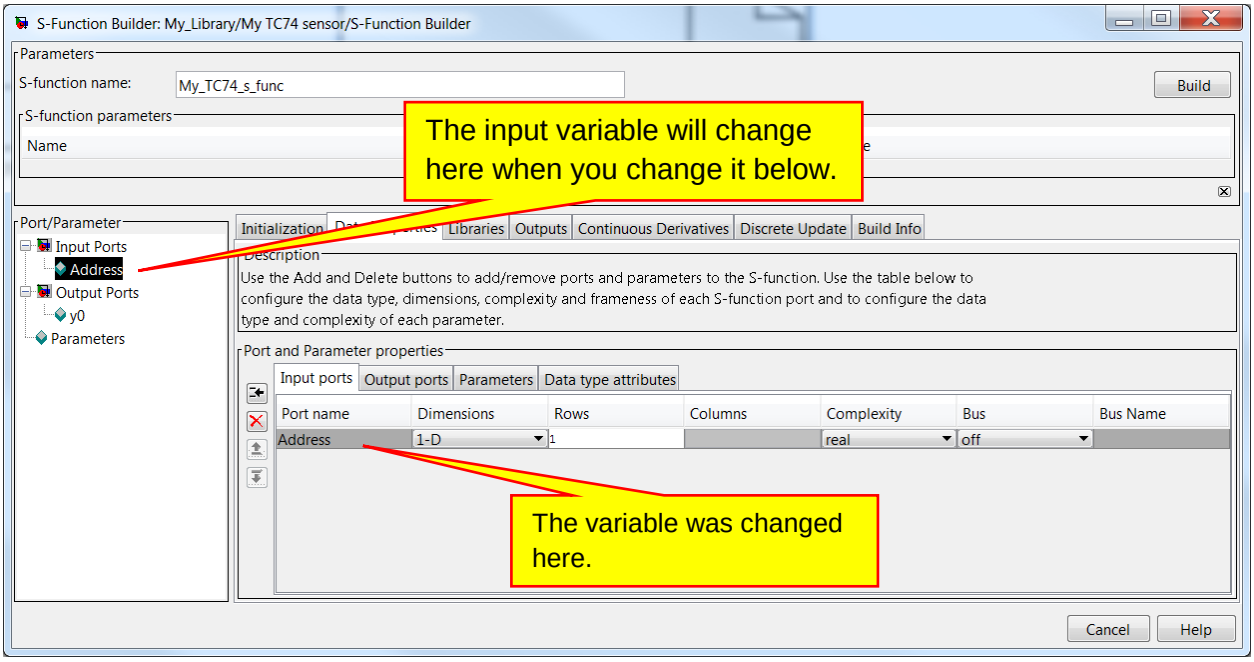
If you do not have a compiler setup, visit http://www.mathworks.com/support/compilers/R2014a/index.html and select and install the appropriate compiler. Note that this link points to the compilers for R2014a and you may need to navigate to the page for your version.

Double click on the S-Function Builder block to open it. Fill in the **S-function name** as **My_TC74_s_func**, specify the number of discrete states as 1, and specify the sample mode as inherited:

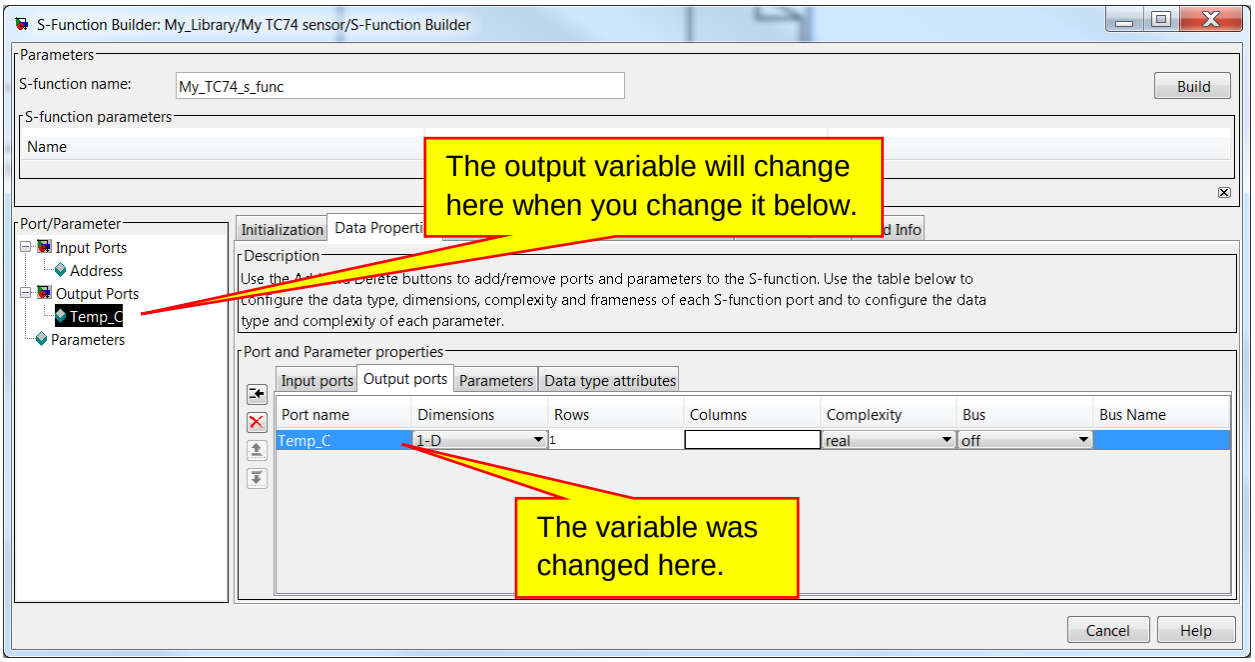


The number of discrete states specifies that there will be two parts of the function that execute at different times. The code segment in the **Discrete Update** tab will execute the first time the block is called. This code is used for initialization code that only needs to run once. The second part is the code in **Outputs** tab. This code executes every time the block is run except the first time. Thus the code in this block is the code you want executed each time the model runs through a loop (except the first time).

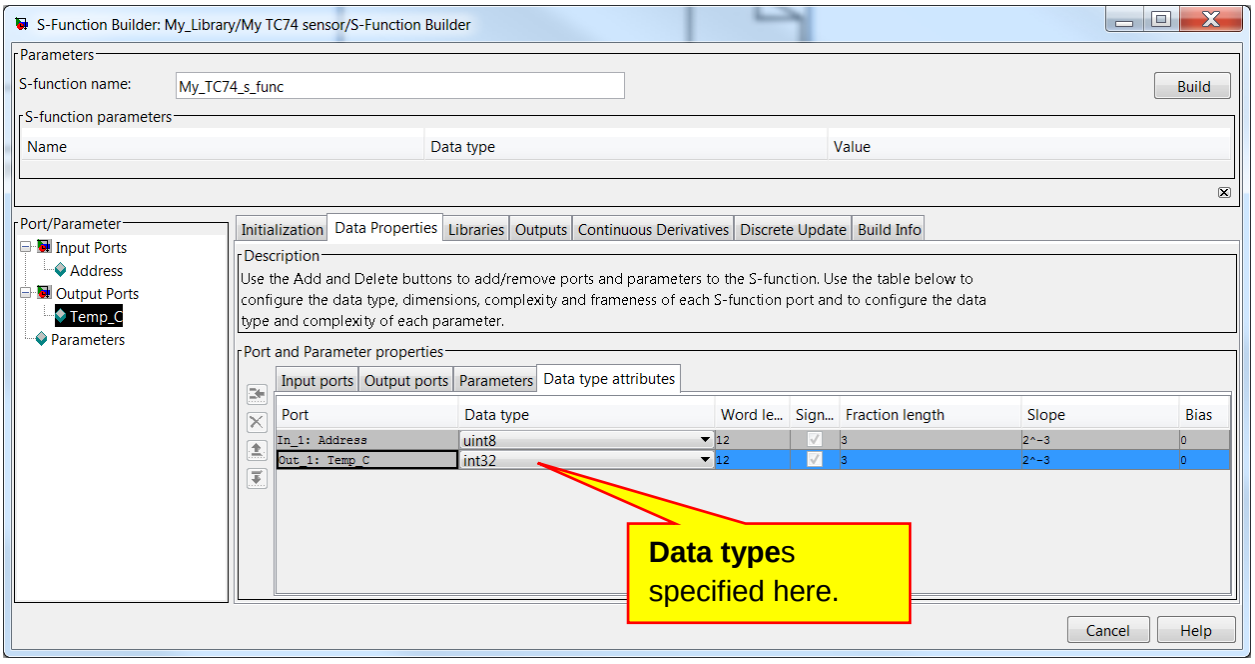
Next we need to specify the input and output parameters. Click on the **Data Properties** tab and then select the **Input ports** tab. Change the name **u0** to **Address** as shown:



Next, select the **Output ports** tab and change y0 to **Temp_C** as shown below:

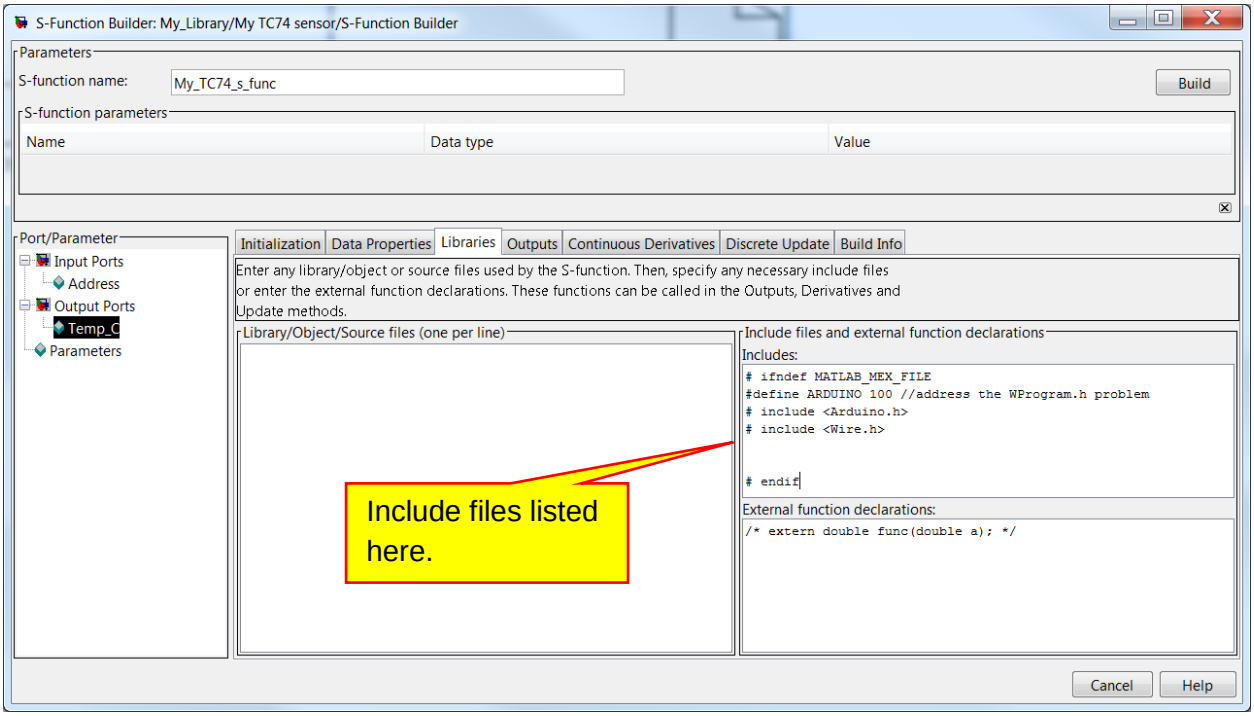


The last thing to do in this screen is to change the data types for the two variables. Remember that we found from the C-code that variable Address should be a uint8 type and Temp_C should be an int32 type:

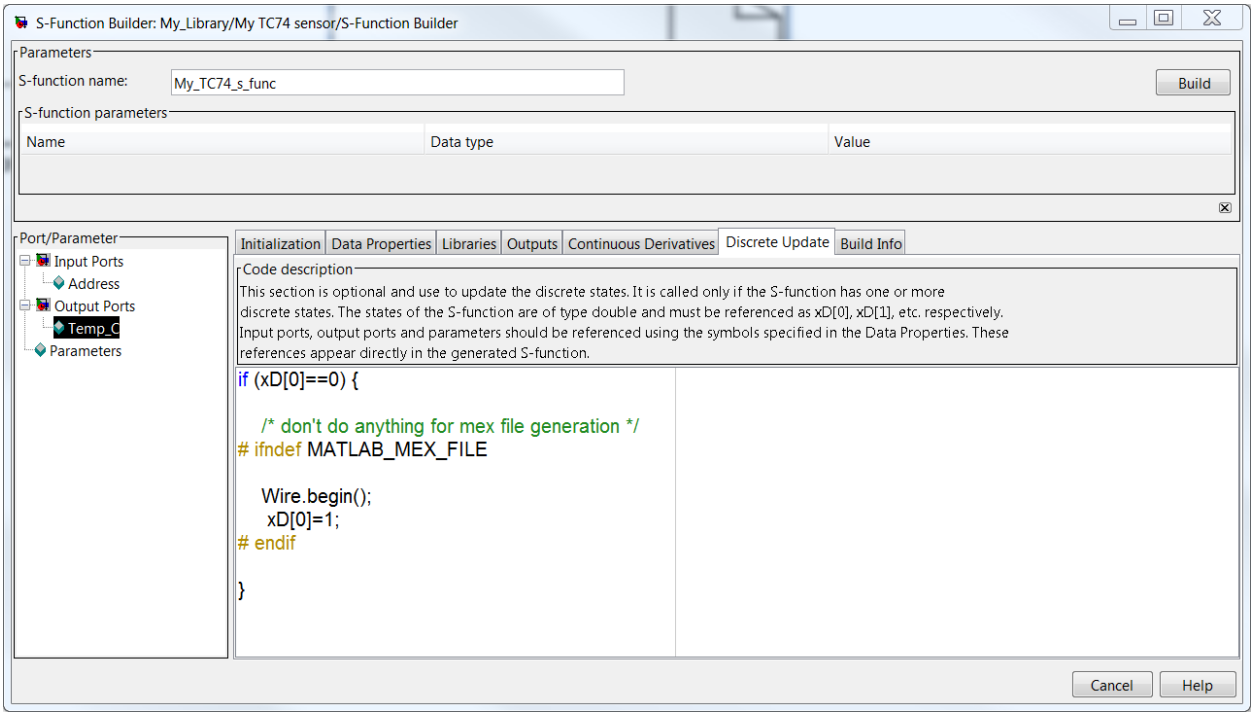


We are done with the input and output variables.

Next, we will specify the include files needed by the function. We will include all of the specified .h files and all .c and .cpp files listed in the copied C-code except Wire.cpp and twi.cpp (if they were listed). Select the libraries tab and fill it in as shown:



Next we will modify the **Discrete Update** tab. This code is executed the first time the block is called, and is used to run any initialization commands that are needed. In this case we need to start the I2C communication on the Arduino:

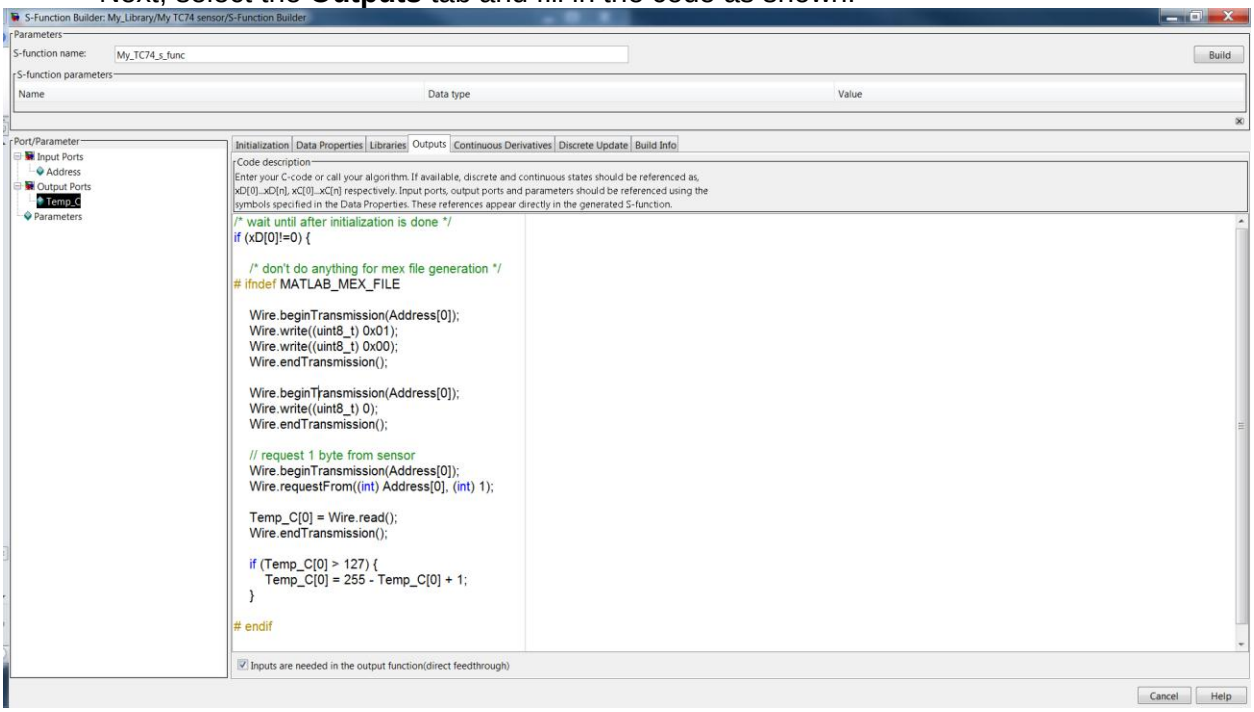


Since we specified that this S-Function has one discrete state, the code in this section will be executed. Since we have one discrete state, information for that state is stored in variable xD[0]. The initial value of xD[0] is zero, which means that the first time this code segment executes, statements inside the **if** statement will execute. The statement **`Wire.begin();`** starts and enables I^2C communication on the Arduino Mega board. The line **`xD[0]=1;`** will make the **if** statement false the next time this code segment executed. The result is

that the code segment **`Wire.begin();`** executes the first time the S-Function runs, initializing I^2C communication, and then the code segment is never executed again until the Mega is reset. Thus, this portion of the S-Function setup is used to run any initialization routines required by your driver that only need to be run once. In this case, all we need to do is initialize I^2C communication.

As a final note on this portion of the setup, the lines **`# ifndef MATLAB_MEX_FILE`** and **`# endif`** serve the same function as they did in the **Libraries** tab. If we are running a simulation, **`MATLAB_MEX_FILE`** will be defined and code for the statements between the **`ifndef`** and **`endif`** will not be generated. (There is no need for code generation if we are running a simulation.) If we are generating code, **`MATLAB_MEX_FILE`** will not be defined and code for our statements will be generated. This technique is also used in the **Outputs** tab, which will be discussed next.

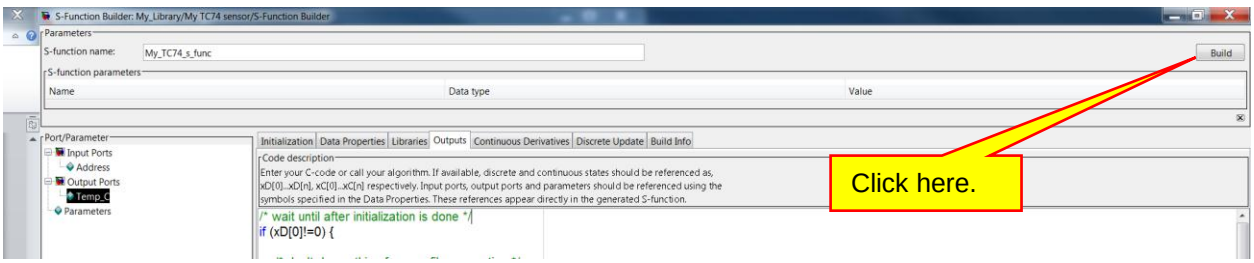
Next, select the **Outputs** tab and fill in the code as shown:



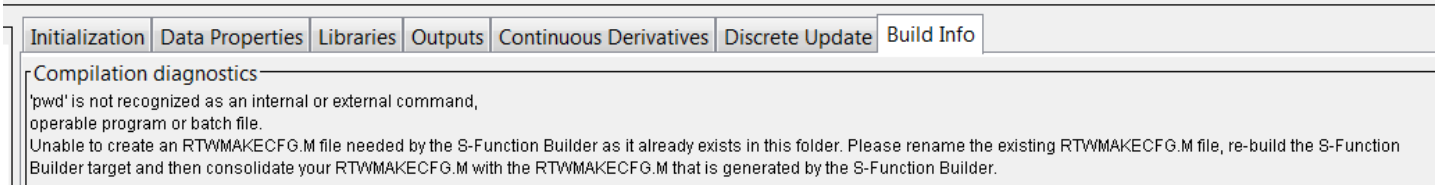
Code in this tab is repeated each time the block is executed (except the first time) and in this case is used to read the temperature sensor. The code shown is the c-code from the example we downloaded except we replaced **`_addr`** with **`Address[0]`** and **`_temp`** with **`Temp_C[0]`**. Note that **`Address[0]`** and **`Temp_C[0]`** are pointers to the variables we defined. With these few changes, we have ported the code to Simulink.

We have enclosed the copied code within an **if** statement. The first time we call the S-Function, xD[0] is 0, so the code segment in this **if** statement does not execute. This allows the initialization routines specified in the **Discrete Update** tab to run before we try to read the sensor. After the initialization code runs, xD[0] is 1, and the code within this **if** statement runs. This function will be called every time the S-function runs. It does not run the first time, but runs every time after that. The code reads the sensor and sets the block output named **Temp_C** to that value.

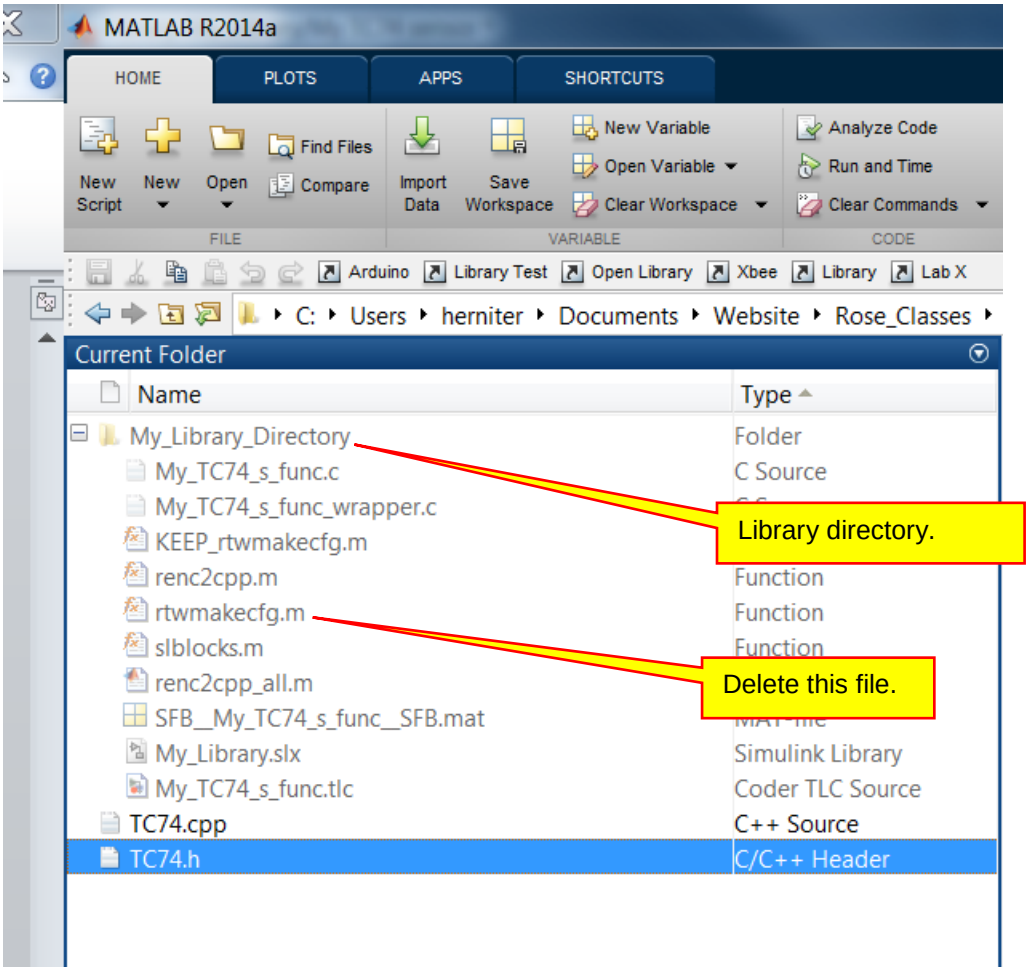
We are now ready to build our C++ files from the S-Function Builder block. Click on the **Build** button:



If you receive the message below:



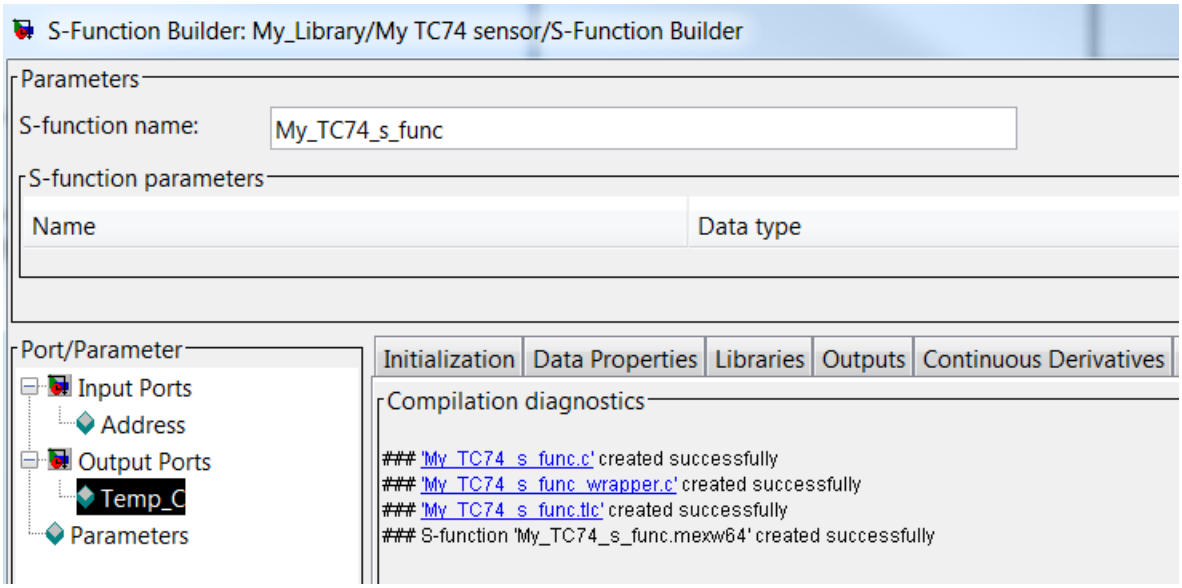
Go back to the MATLAB window, go to the directory where your library is stored, and find the file named rtwmakecfg.m:



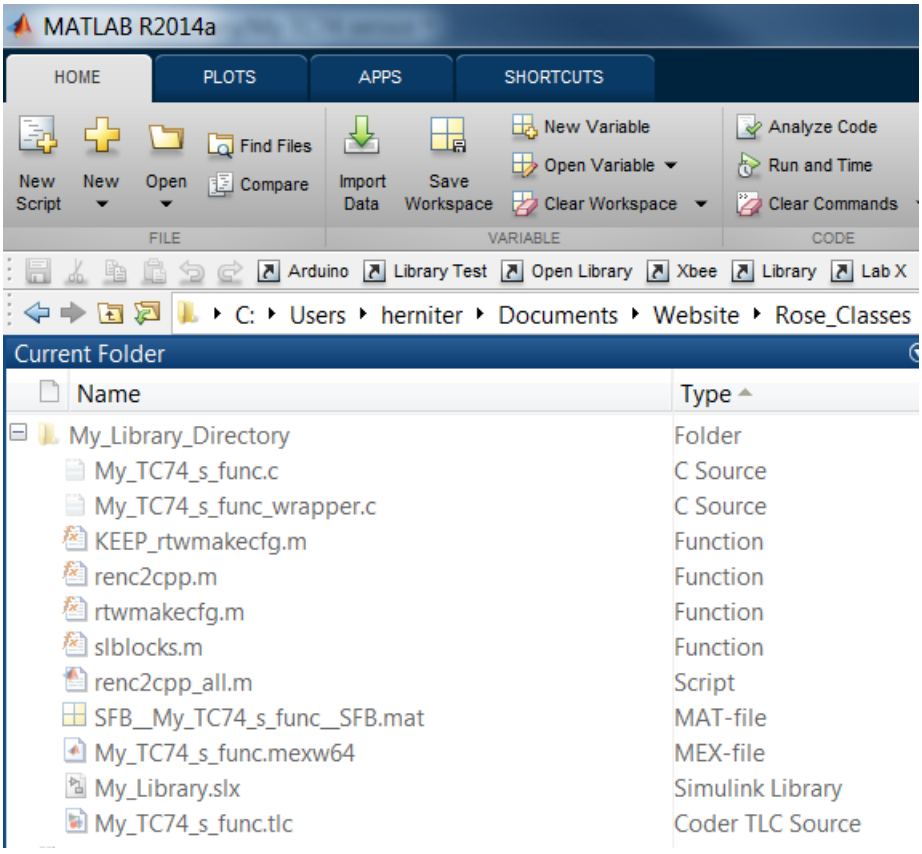
Delete the file named rtwmakecfg.m.

After deleting the file, go back to the S-Function Builder window and click the **Build** button again. If you have no mistakes[13] in the code you entered, you will see the text below, and the Builder will create several files for you:

[13] By mistakes, we do not include logic errors. The builder will detect a minimal amount of errors in your code.

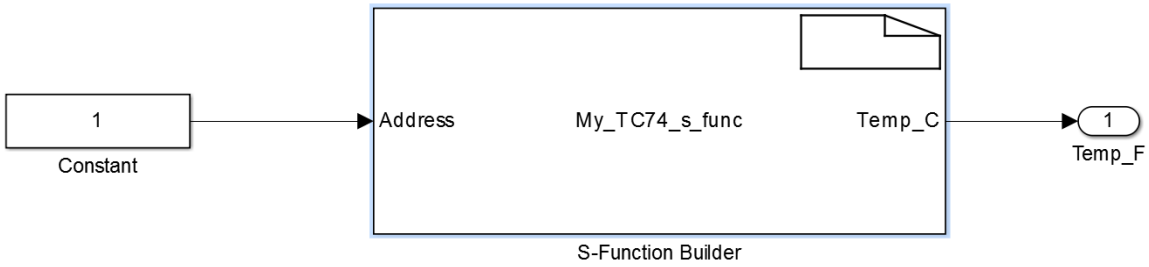


The files listed will be saved in your library directory:



You can now close the S-Function Builder window.

If you look back at the Simulink model, your will see that the blocks have been updated:



The input has been changed to **Address**, the output to **Temp_C**, and the name of the block to **My_TC74_s_func**.

The code that we entered is contained in the file named My_TC74_s_func_wrapper.c. Double-click on this file to view it with the MathWorks text editor. We see several portions that were created from code that we entered with the S-Function Builder block:

```
1
2
3   /*
4    * Include Files
5    *
6    */
7   #if defined(MATLAB_MEX_FILE)
8   #include "tmwtypes.h"
9   #include "simstruc_types.h"
10  #else
11  #include "rtwtypes.h"
12  #endif
13
14  /* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
15  # ifndef MATLAB_MEX_FILE
16  #define ARDUINO 100 //address the WProgram.h problem
17  # include <Arduino.h>
18  # include <Wire.h>
19
20
21  # endif
22  /* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
23  #define u_width 1
24  #define y_width 1
25  /*
26   * Create external references here.
27   *
28   */
29  /* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
30  /* extern double func(double a); */
31  /* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */
32
```

This was created from code we placed in the **Library** tab.

We will need to modify this line to 'extern "C" My_TC74…' Don't make the changes manually. We have a MATALB script that will make the changes.

```
 * Output functions
 *
 */
void My_TC74_s_func_Outputs_wrapper(const uint8_T *Address,
          int32_T *Temp_C,
          const real_T *xD)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
/* wait until after initialization is done */
if (xD[0]!=0) {

   /* don't do anything for mex file generation */
# ifndef MATLAB_MEX_FILE

   Wire.beginTransmission(Address[0]);
   Wire.write((uint8_t) 0x01);
   Wire.write((uint8_t) 0x00);
   Wire.endTransmission();

   Wire.beginTransmission(Address[0]);
   Wire.write((uint8_t) 0);
   Wire.endTransmission();

   // request 1 byte from sensor
   Wire.beginTransmission(Address[0]);
   Wire.requestFrom((int) Address[0], (int) 1);

   Temp_C[0] = Wire.read();
   Wire.endTransmission();

   if (Temp_C[0] > 127) {
      Temp_C[0] = 255 - Temp_C[0] + 1;
   }

# endif
```

This was created from code we placed in the **Output** tab.

And finally,

We will need to modify this line to 'extern "C" My_TC74…' Don't make the changes manually. We have a MATALB script that will make the changes.

```
78  void My_TC74_s_func_Update_wrapper(const uint8_T *Address,
79            const int32_T *Temp_C,
80            real_T *xD)
81  {
82    /* %%%-SFUNWIZ_wrapper_Update_Changes_BEGIN --- EDIT HERE TO _END */
83  if (xD[0]==0) {
84
85     /* don't do anything for mex file generation */
86  # ifndef MATLAB_MEX_FILE
87
88     Wire.begin();
89      xD[0]=1;
90  # endif
91
92  }
```

This was created from code we placed in the **Discrete Updates** tab.

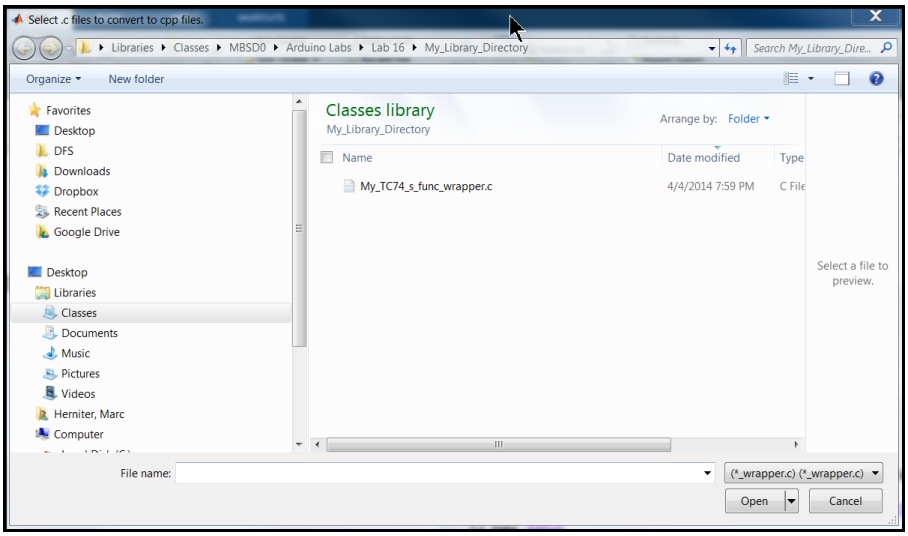
Close the file without making any changes.

There are three changes we need to make and two more steps we need to apply this file before we can use it. You do not have to make these changes because we[14] have created a MATLAB script to make the changes for you:

1. Change **`void My_TC74_s_func_Outputs_wrapper(const uint8_T *Address`**, to **`extern "C" void My_TC74_s_func_Outputs_wrapper(const uint8_T *Address`**,
2. Change **`void My_TC74_s_func_Update_wrapper(const uint8_T *Address`**, to **`extern "C" void My_TC74_s_func_Update_wrapper(const uint8_T *Address`**,
3. Rename the file My_TC74_s_func_wrapper.**c** to My_TC74_s_func_wrapper.**cpp**, and
4. Delete the file named rtwmakecfg.m.
5. Copy the **KEEP_rtwmakecfg.m** to **rtwmakecfg.m**.

Steps 1 through 3 are necessary because the Arduino IDE uses C++. Steps 4 and 5 set up the MATLAB path so that when we run a model with this block, the compiler knows where to find the include files.

To do all of these steps, make sure that you are in your library directory and then run the file named **ren2cpp_all.m**[15]. This file will find all of the wrapper.c functions in the current directory and ask you which ones you would like to modify. (We only have one, but you can select multiple files if you have more than one):

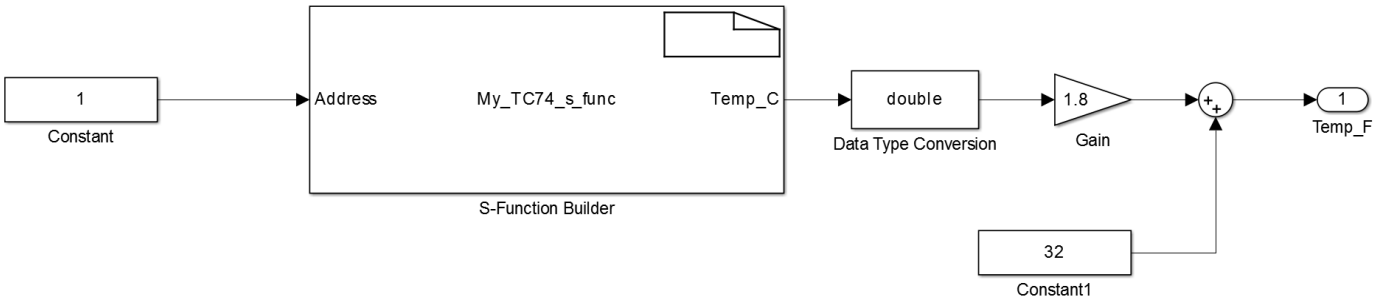


Select the file named **My_TC74_s_func_wrapper.c** and then click the **Open** button. Steps 1 through 5 above will be completed. You can tell that it worked by verifying that files rtwmakecfg.m and KEEP_rtwmakecfg.m contain the same commands, the file My_TC74_s_func_wrapper.c has been renamed to My_TC74_s_func_wrapper.cpp, and that the contents of file My_TC74_s_func_wrapper.cpp have been modified to add the extern “C” text.

E. Modifying the Model

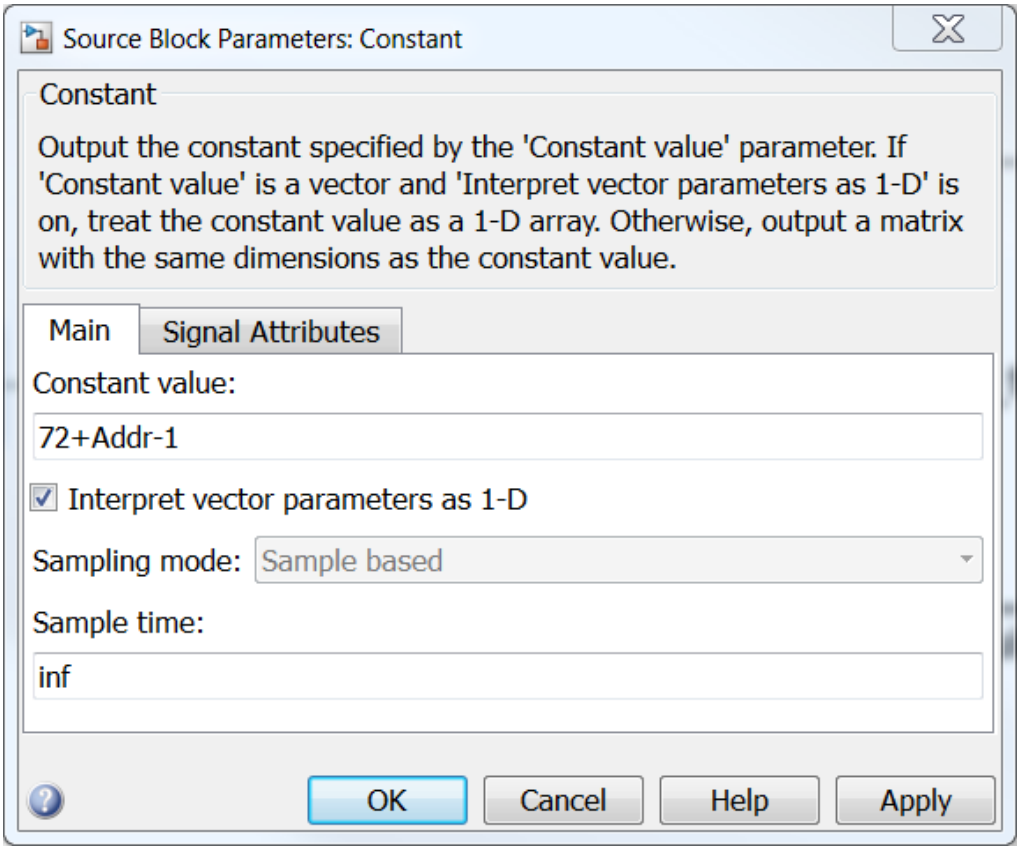
We now need to make some modifications to the Simulink model for the temperature sensor. First, add the blocks as shown to change the output data type to double and convert the temperature to Fahrenheit:

[14] “We” being The MathWorks. I did not write this script and disavow all knowledge of how it works or what is contained within.
[15] I actually did write this script. But this script calls the script I did not write…..

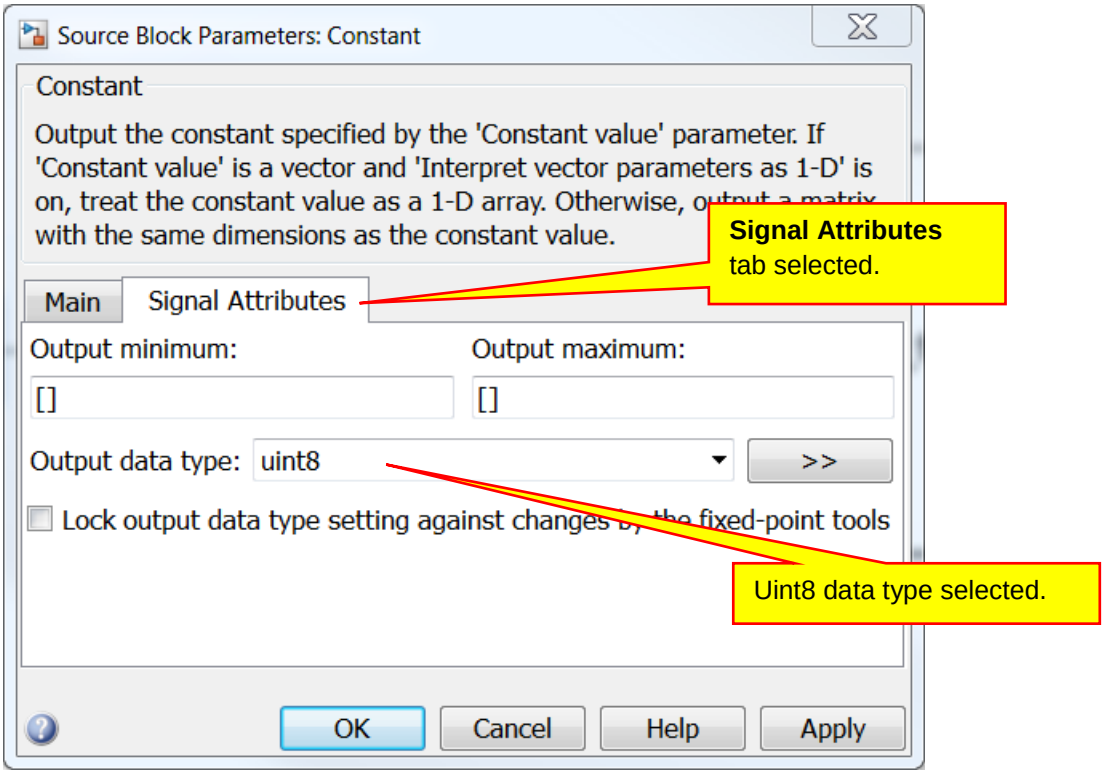


The address is going to come from the mask, which we will create in the next step. The mask will give us a value from 1 to 8, which we need to conve

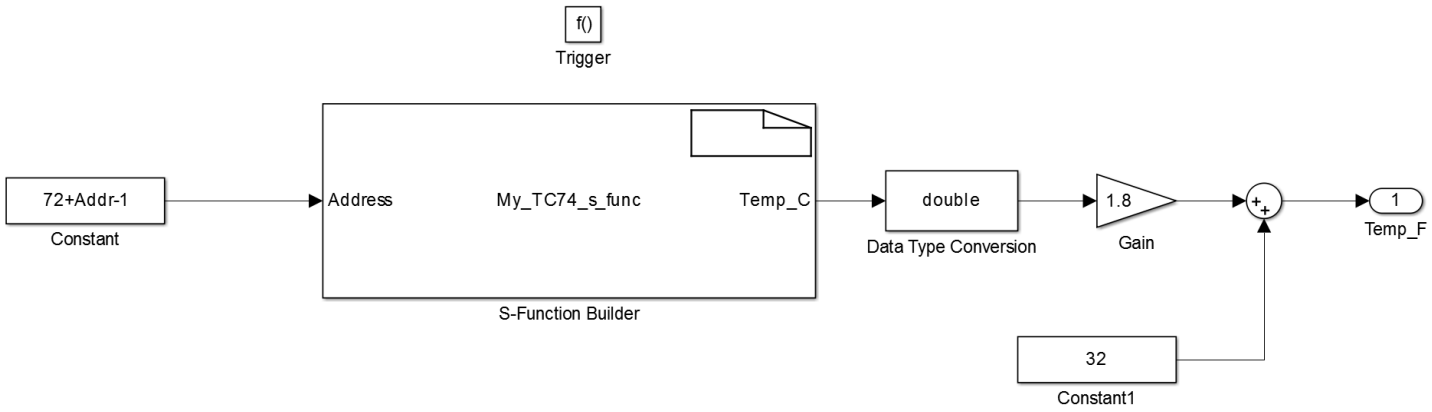
rt to 72 to 79. Double-click on the constant block and change the value to 72+Addr-1:



Variable Addr will be defined in the mask. The C code that we created requires a data type of uint8, so we need to change the data type of the constant to uint8. Click on the **Signal Attributes** tab and specify the type as uint8:

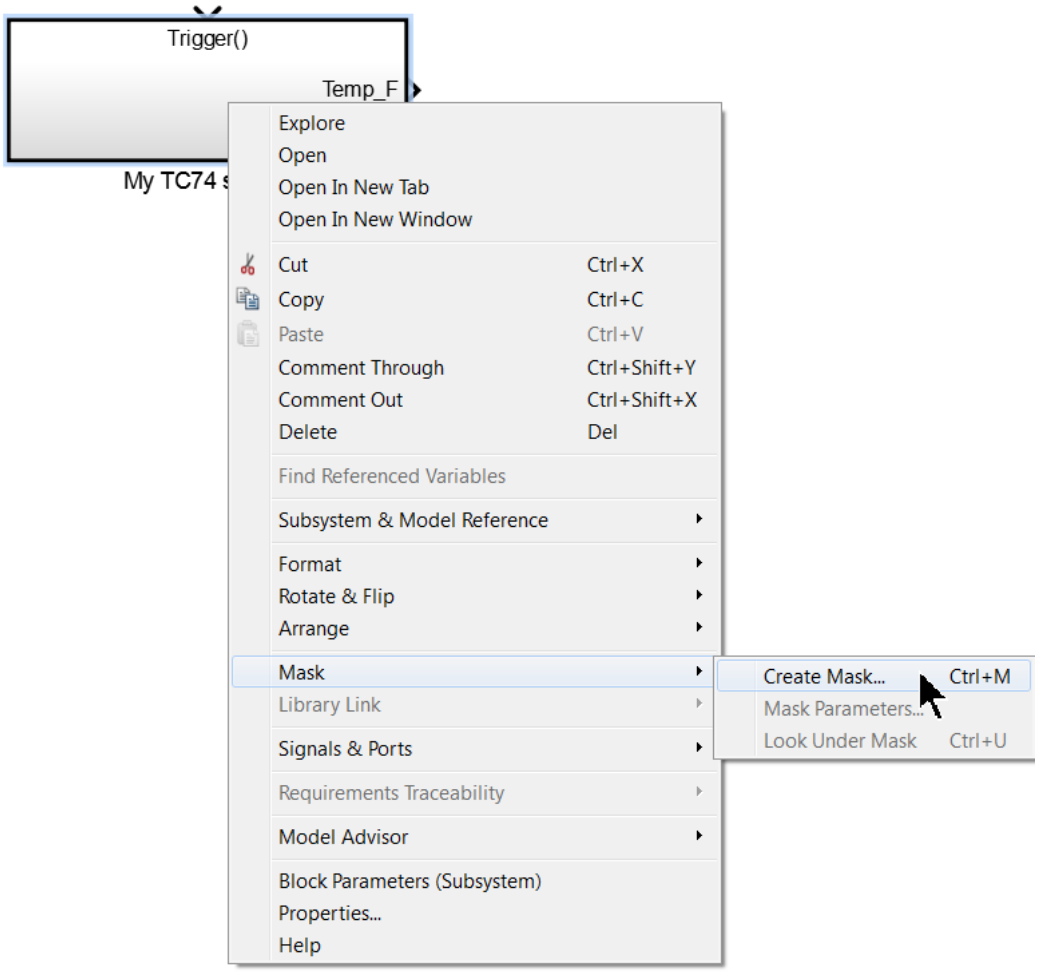


Click the **OK** button to accept the changes. The last thing we need to do is add a trigger. I have chosen a function trigger, but you can select a different type if you like. The trigger block is located in the **Simulink** / **Ports & Subsystems** library:

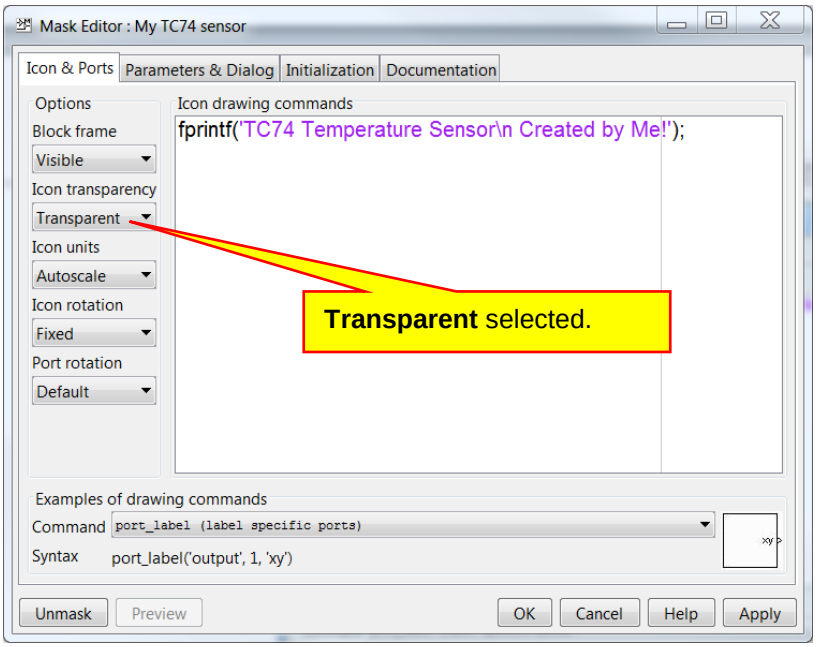


F. Creating a Subsystem Mask

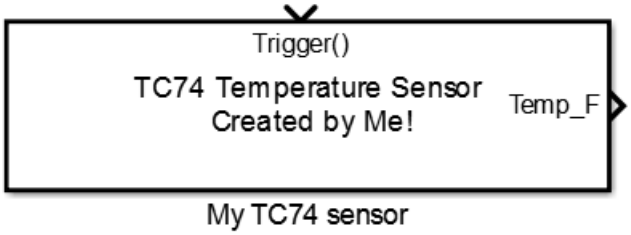
The last step we need to do is to create the mask for our library block. Navigate to the top level of the library and right click on the sensor block and select **Mask** and then **Create Mask**:



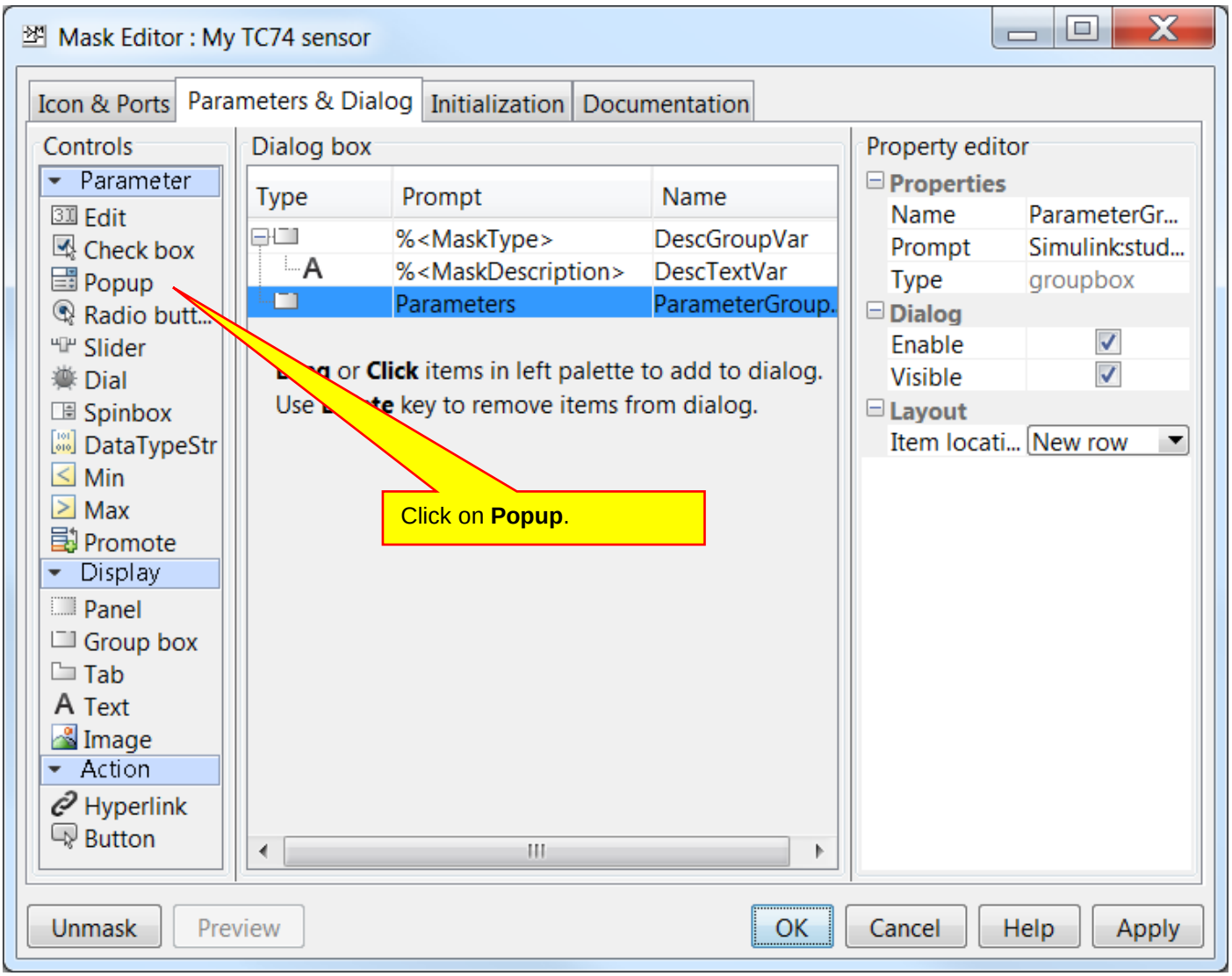
The **Mask Editor** will open. The icons and ports window allows us to place text and graphics on the subsystem block. For now, we will just add text. Fill in the dialog box as shown:



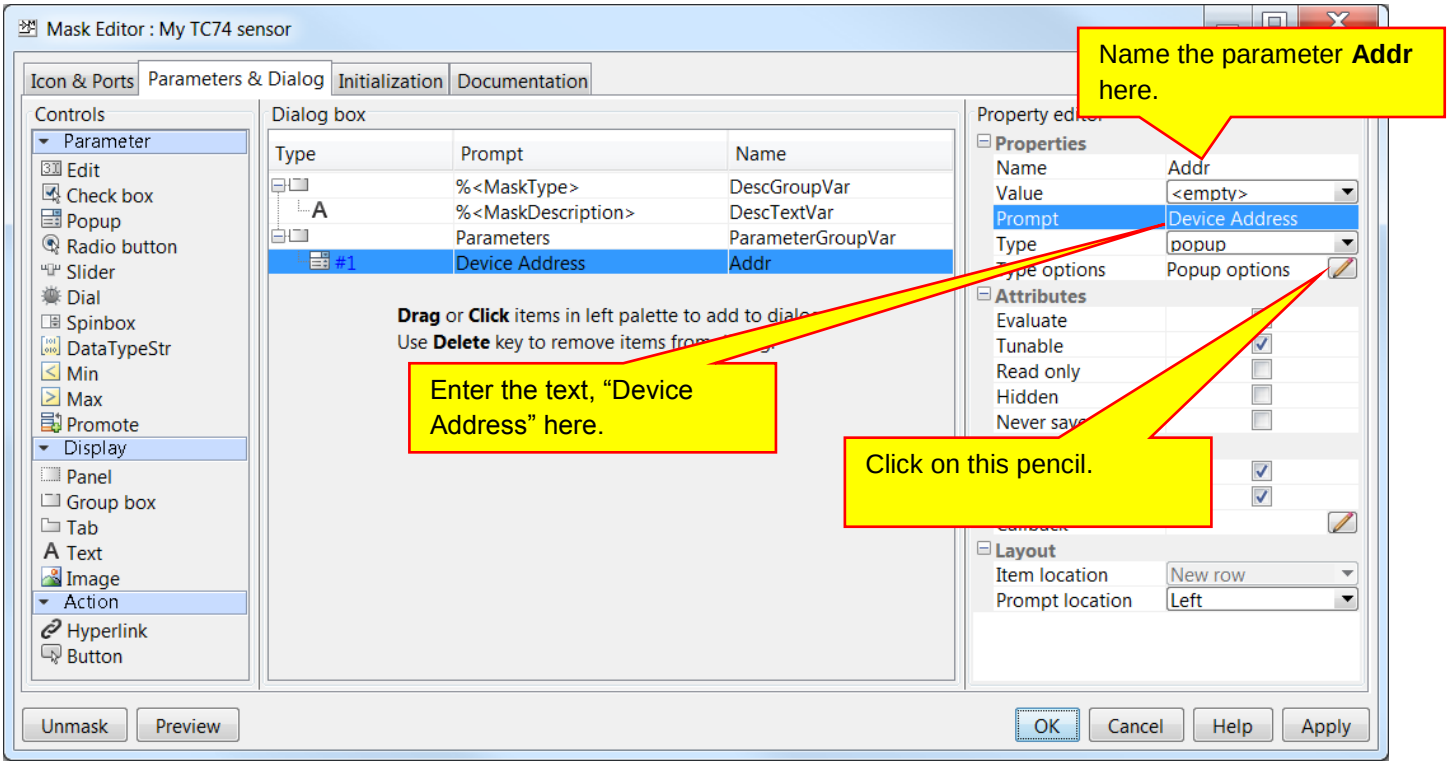
Make sure that you select **Transparent** as shown above, or your port labels will be covered up and no longer visable. If you click the **Apply** button, you will see this text appear on your subsystem:



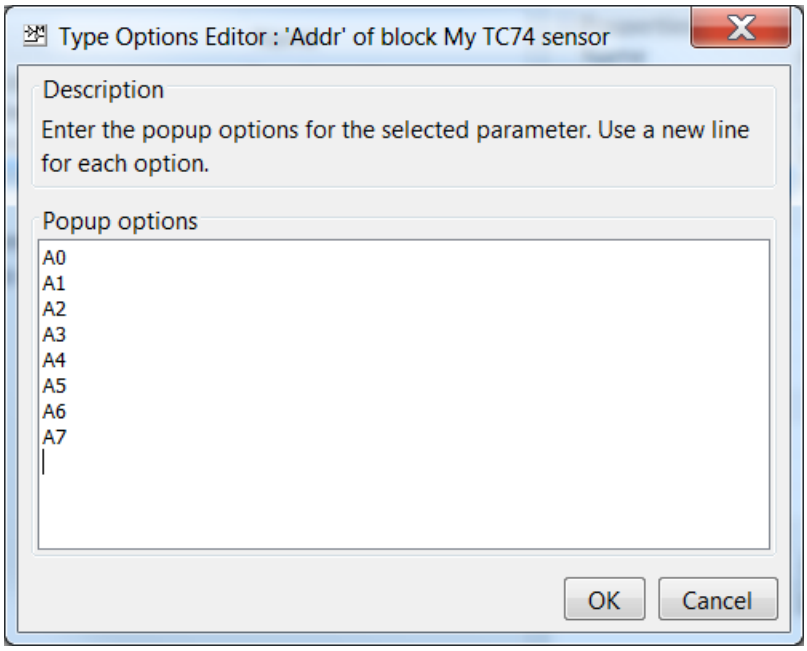
The **Parameters & Dialog** tab allows us to set up the Addr parameter used inside the subsystem:



There are several ways to create parameters. We will show just one here. Click on the **Popup** menu selection to create a new parameter and then fill in the portions listed below:

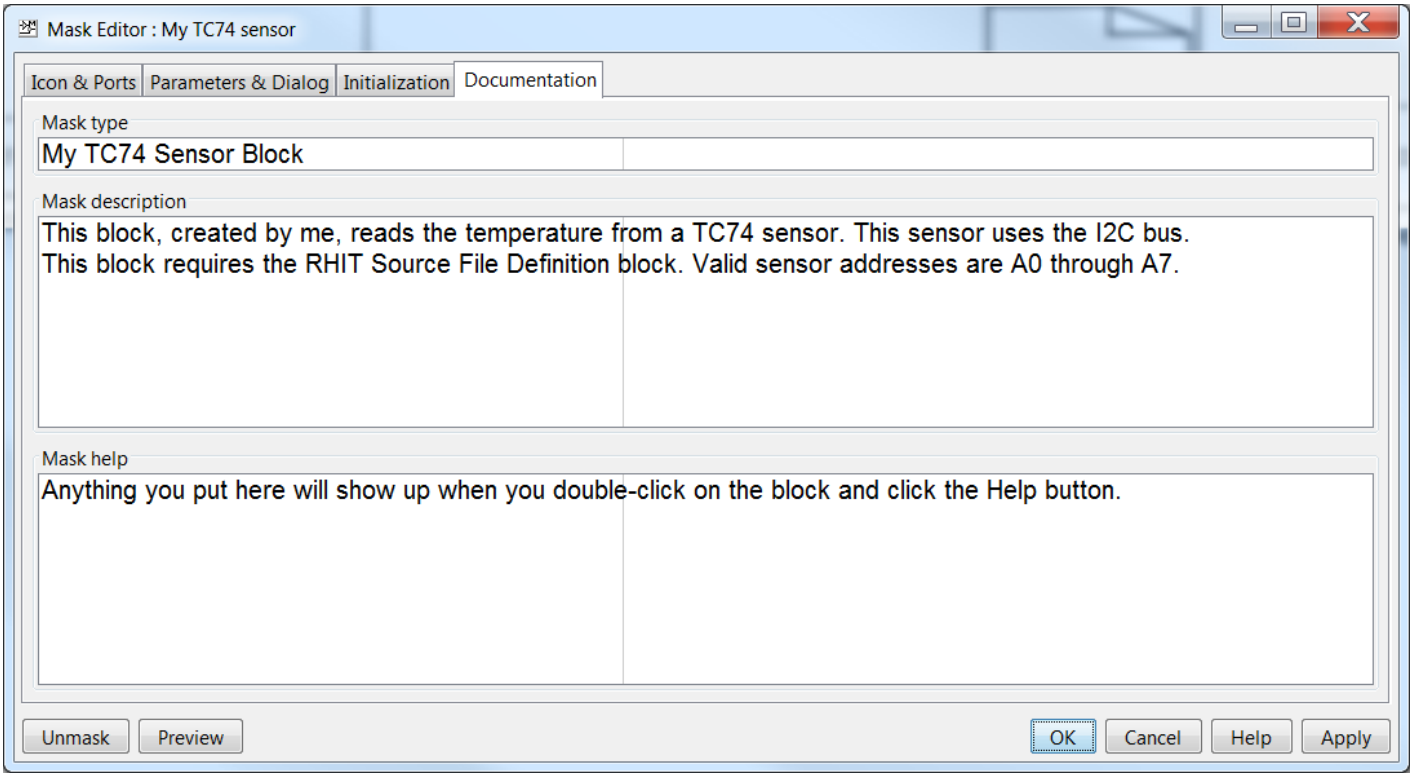


A Popup will give us an enumerated list of text based choices. Click on the pencil as shown above to create the list. The list should contain the values A0, A1, …, A7 as shown:



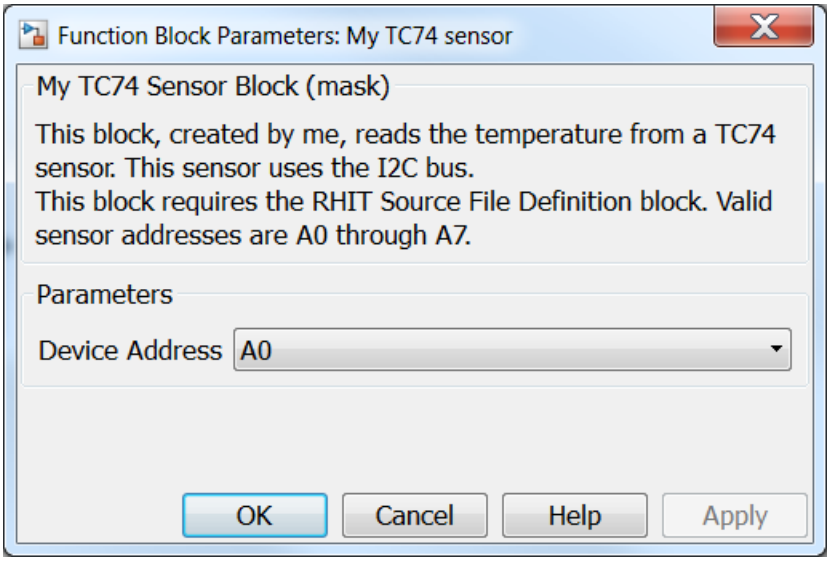
Click the **OK** button.

The last thing we need to do is set up some documentation. Click in the **Documentation Tab** and fill in it as shown:



Click the **OK** button to select the settings.

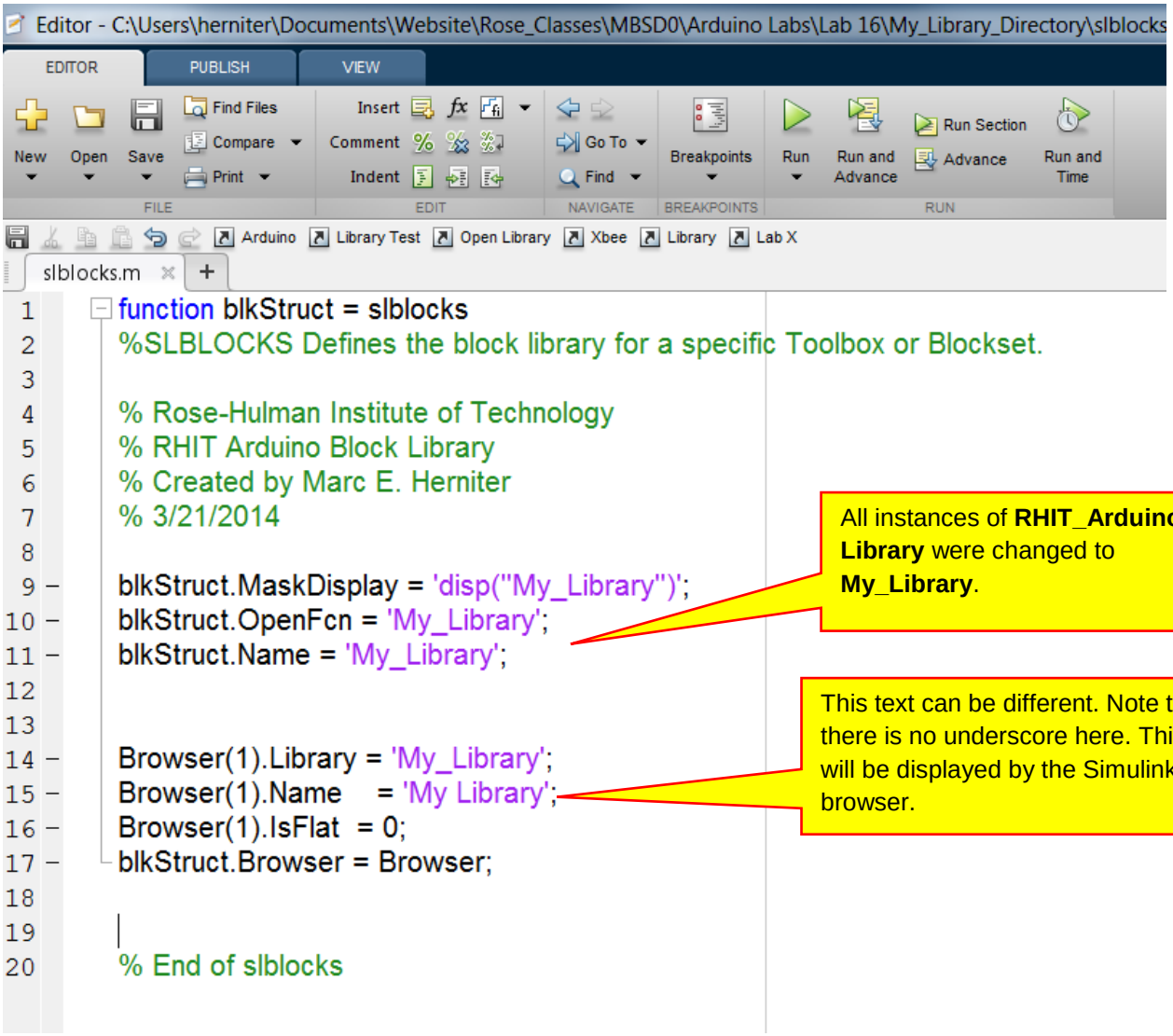
You can now double-click on your TC74 block and see the mask that you just created:



We are done creating the library, so you should save the file.

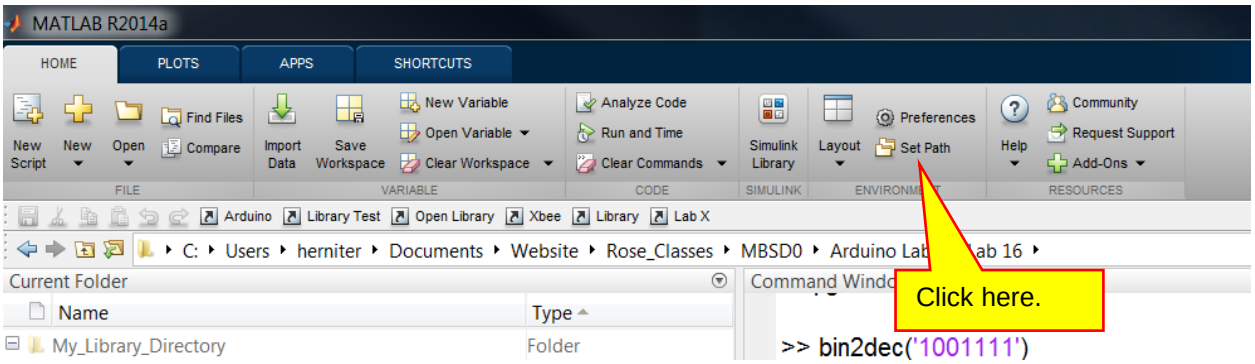
G. Adding the Library to Your Simulink Browser

We will now go through the steps that you will need to add the library you just created to the Simulink library browser. First, edit the file named sblocks.m. We copied this file to the library directory previously. Make the changes shown below:



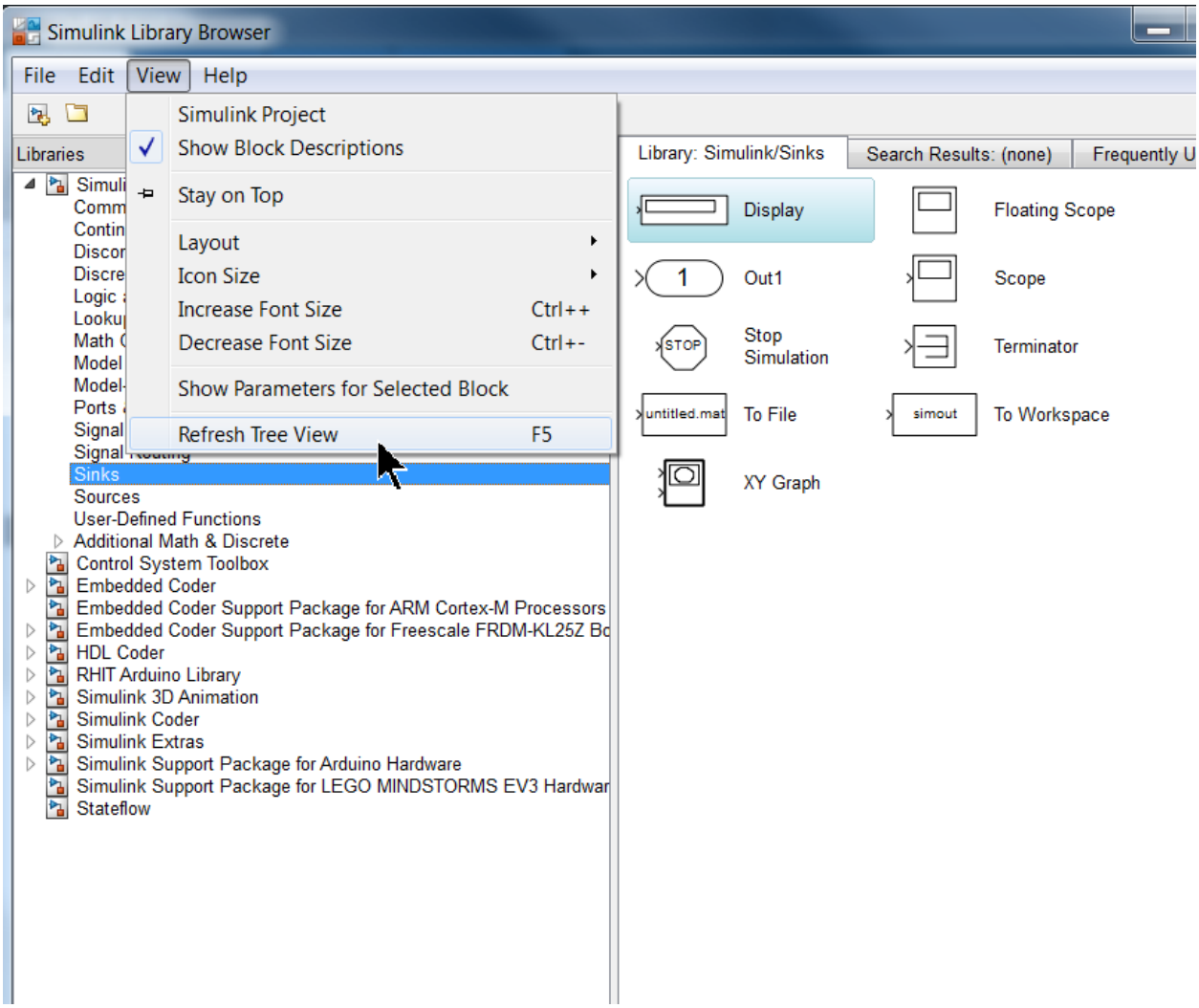
Save and close the file. Note that any Simulink library that is specified in a sblocks.m file that is in a directory that is on the MATLAB path will be listed as a library in the Simulink Library browser.

Next, we need to modify the path to include the directory where our library resides. Click on the **Set Path** icon in the MATLAB window:

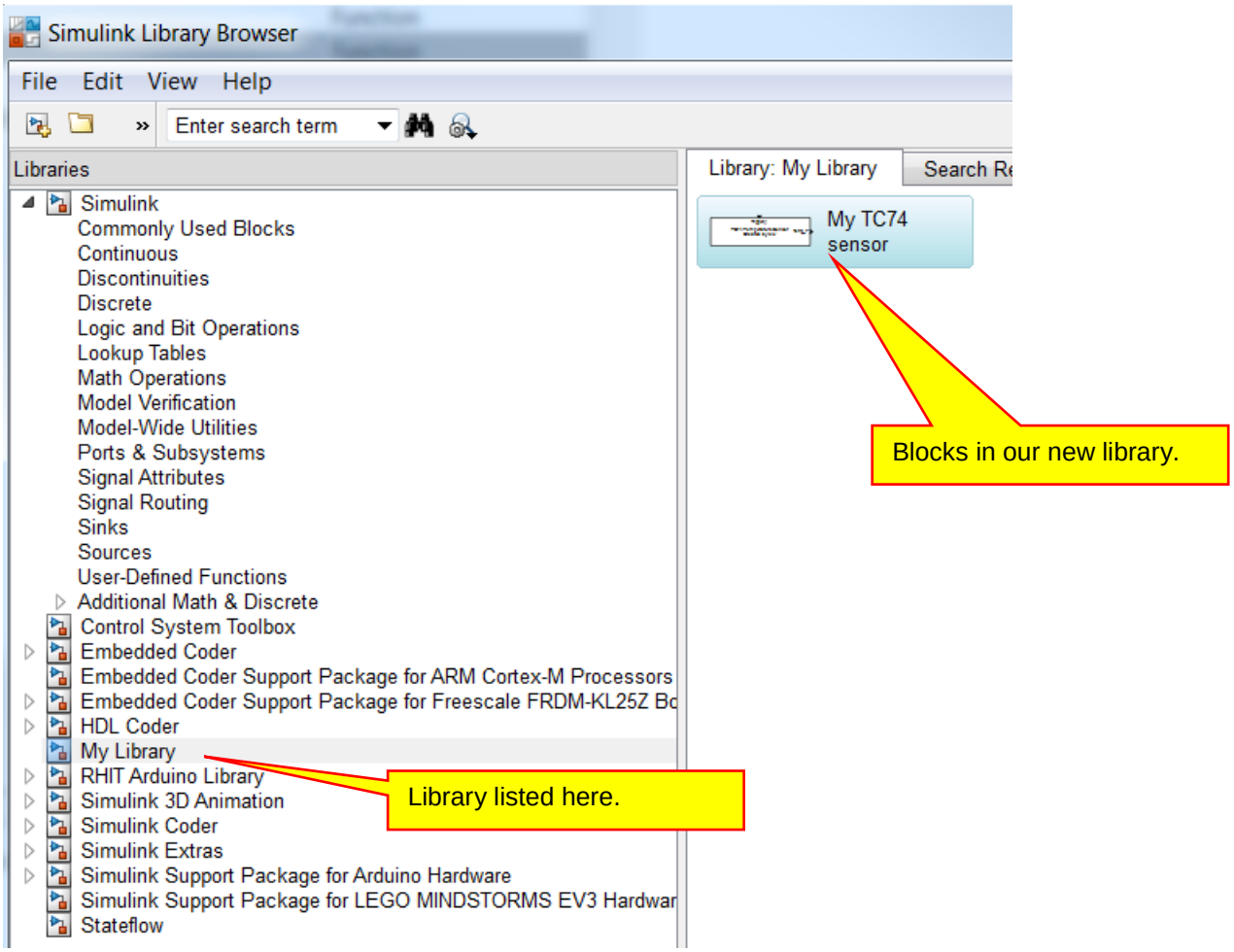


Add directory **My_Library_Directory** to the path, click the **Save** button and then the **Close** Button.

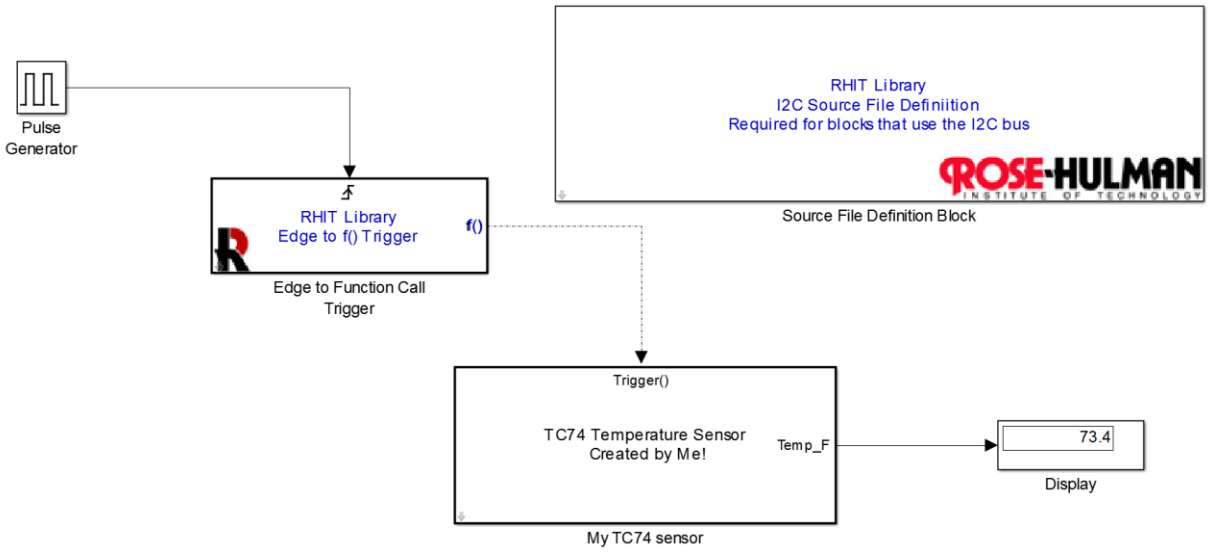
Lastly, run Simulink and then select **View** and then **Refresh Tree** from the Simulink menus:



The library should now appear in your Simulink library browser:



We can now use the block to create models. Create the model below to test the sensor!



Run the model in **External** mode to verify its operation.

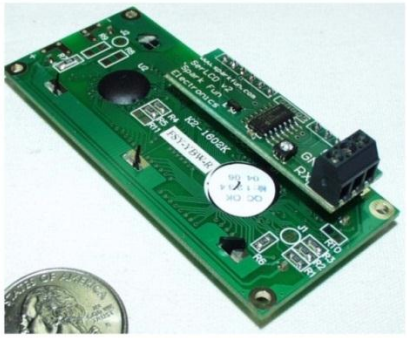
Appendix

A. SparkFun Serial LCD Module Datasheet

SerLCD v2.5
Serial Enabled LCD
7/5/2006

1 Overview

The SerLCD v2.5 is a simple and cost effective solution for interfacing to Liquid Crystal Displays (LCDs) based on the HD44780 controller. The SerLCD module takes incoming 9600bps TTL level signals and displays those characters on the LCD screen. Only three wires - 5V, GND, and Signal - are needed to interface to the LCD.



Please report typos, inaccuracies, and especially unclear explanations to us at spark@sparkfun.com. Suggestions for improvements are welcome and greatly valued.

1.1 Features in Version 2.5

SerLCD v2.5 has some new features that make the SerLCD even more powerful and economical:

- New PIC 16F688 utilizes onboard UART for greater communication accuracy
- Adjustable baud rates of 2400, 4800, 9600 (default), 14400, 19200 and 38400
- Operational backspace
- Greater processing speed at 8MHz
- Incoming buffer stores up to 80 characters
- Backlight transistor can handle up to 1A
- Pulse width modulation of backlight allows direct control of backlight brightness and current consumption
- All surface mount design allows a backpack that is half the size of the original
- Faster boot-up time
- Boot-up display can be turned on/off via firmware
- User definable Splash Screen

2 Interface Specifications

Default communications occur at 9600bps with 8 bits of data, 1 start bit, 1 stop bit, and no parity.

The SerLCD is controlled using actual ASCII characters. This means that if you pass the ASCII character 'r' to the module, an 'r' will be displayed on the LCD at the next cursor position. There are only two exceptions to this. These are the command characters decimal 254 (**0xFE**) and 124 (**0x7C**).

3 Configuration

All settings are stored on onboard EEPROM and loaded during power up.

SerLCD

Backlight Brightness	
Value	**Brightness**
128	**Off**
140	**40% On**
150	**73% On**
157	**Fully On**
158	**Not Valid**

3.1 Backlight

The SerLCD v2.5 Pulse Width Modulates the backlight via a 1A BJT transistor. This allows the user to set 1 of 30 different brightness settings.

By sending the special command character **0x7C** (decimal 124) followed by a number 128-157, the backlight PWM value will be set. This is handy when power consumption of the unit must be minimized. By reducing the brightness, the overall backlight current consumption is reduced.

3.2 LCD Type Setup

The SerLCD v2.5 firmware includes settings to interface to the following types of LCDs : 2x16, 2x20, 4x16, and 4x20.

If you purchased the SerLCD soldered to an LCD, it has already been configured to work with that specific LCD. You should not have to configure anything.

LCD Type	
20 Characters Wide	3
16 Characters Wide	4
4 Lines	5
2 Lines	6

If you purchased the SerLCD module by itself, you will have to tell the module what type of LCD it is going to be, or is currently, attached to.

To control what type of LCD the SerLCD module is attached to, transmit the special command - 124 (0x7C). Follow this command with either 3, 4, 5, or 6. These commands set the LCD character width and number of lines. These settings are used to correctly wrap the cursor to keep it within the viewable screen. The type of LCD is saved to EEPROM after each change.

3.3 Extended LCD Commands

HD44780 Commands	
Clear Display	**0x01**
Move cursor right one	**0x14**
Move cursor left one	**0x10**
Scroll right	**0x1C**
Scroll left	**0x18**
Turn visual display on	**0x0C**
Turn visual display off	**0x08**
Underline cursor on	**0x0E**
Underline cursor off	**0x0C**
Blinking box cursor on	**0x0D**
Blinking box cursor off	**0x0C**
Set cursor position	**0x80 +**

The HD44780 LCD controller is very common. The extended commands for this chip include but are not limited to:

Please refer to the HD44780 datasheet for more information.

Clear display and set cursor position are the two commands that affect the SerLCD the most. By sending these commands to the SerLCD the cursor position gets changed. This change is tracked by the firmware and cursor wrapping is performed normally. A cursor move to outside the viewable area is possible and the cursor position variable will be updated accurately.

16 Character Displays	
Line Number	**Viewable Cursor Positions**
1	0-15
2	64-79
3	16-31
4	80-95

20 Character Displays	
Line Number	**Viewable Cursor Position**
1	0-19
2	64-83
3	20-39
4	84-103

The viewable area cursor positions are as follows for almost all HD44780 based LCDs:

To perform a cursor move, a series of steps must occur:

1. You will need to determine the correct decimal position to move to. For example, the viewable position three on the second line of a 16 character display is 66.
2. Set bit 7 (the highest bit) of that decimal number to '1'. Position 66 + 128 = 194.
3. Now transmit the special character 254 to tell the SerLCD you want to send a command.
4. Finally, transmit the number 194. The cursor is now sitting in the third position of the second line.

3.4 Splash Screen

The SerLCD v2.5 displays a splash screen by default. This splash screen ('Sparkfun.com SerLCD v2') verifies that the unit is powered, working correctly, and that the connection to the LCD is correct.

The splash screen is displayed for 500ms during boot-up and may be turned off if desired.

A new addition to the V2.5 firmware is the ability for the user to set their own splash screen (2 lines). To do this, just set up the top 2 lines as you would like them to appear, then send special character 124 followed by "<control>j" to save it to memory. To test, just cycle power.

To disable the splash screen, send the 'special command' **0x7C** (decimal 124) to the unit followed by decimal 9. Every time this command is sent to the unit, the Splash Screen Display option will toggle. That is, if the splash screen is currently being displayed, sending the **0x7C 0x09** command will disable the splash screen during the next boot. Sending the **0x7C 0x09** command again will enable the splash screen.

3.4 Changing the Baud Rate

The Serial LCD V2.5 defaults to 9600 baud, but can be set to a variety of baud rates. To change the baud rate, first enter the special command character 124, then:

- 2400 baud, enter "<control>k"
- 4800 baud, enter "<control>l"
- 9600 baud, enter "<control>m"
- 14400 baud, enter "<control>n"
- 19200 baud, enter "<control>o"
- 38400 baud, enter "<control>p"

If the Serial LCD gets into an unknown state, or you otherwise can't communicate with it at the baud rate to which it is set, enter "<control>r" at 9600 baud while the splash screen is active and the unit will reset to 9600 baud.

4 Hardware

4.1 Vcc and Current Draw

The SerLCD v2.5 should only be powered by 5V DC. Higher than 5.5V will cause damage to the PIC, LCD, and backlight (if attached).

The SerLCD uses 3mA with the backlight turned off and ~60mA with the backlight activated.

4.2 Contrast Control

The SerLCD v2.5 comes equipped with a 10k potentiometer to control the contrast of the LCD. This is set by during assembly and testing but may need correcting for your specific LCD module. Temperature and supply voltage can effect the contrast of the LCD.

4.3 Hi-Current Control Pin

The SerLCD v2.5 uses a general purpose, 1000mA NPN transistor to control the Backlight. If you purchased the SerLCD module, you may use this pin as a general purpose, high power control pin. If you issue the backlight on/off command to the SerLCD, pin 15 (next to the last pin) will turn on/off. Pin 16 (last pin above 'www.sparkfun.com') is connected to ground.

B. SUNON DC Cooling Fan

SUNON

80x80x25 mm

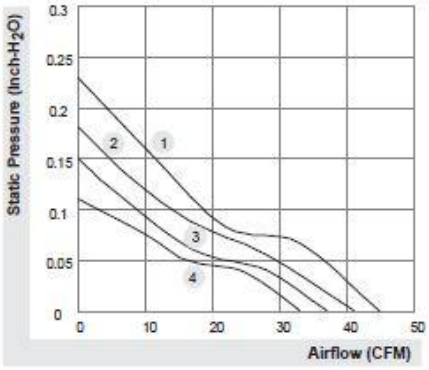
NEW

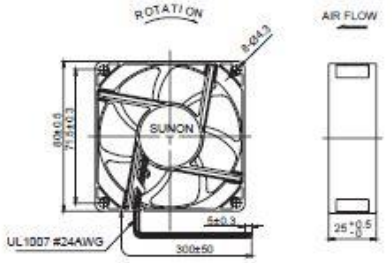
33~45 CFM

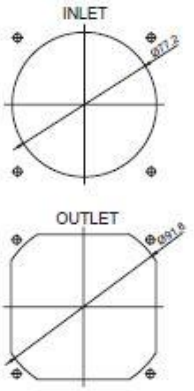
MAGLev by SUNON	Bearing ● VAPO	Rating Voltage (VDC)	Power Current (AMP)	Power Consumption (WATTS)	Speed (RPM)	Air Flow (CFM)	Static Pressure (Inch-H_2O)	Noise (dBA)	Weight (g)	Curve
ME80251VX-0000-A99	●	12	0.165	1.98	3600	45	0.23	37.0	81	1
ME80251V1-0000-A99	●	12	0.145	1.7	3200	41	0.18	33.0	77	2
ME80251V2-0000-A99	●	12	0.12	1.4	2900	37	0.15	30.0	77	3
ME80251V3-0000-A99	●	12	0.09	1.1	2600	33	0.11	28.0	77	4
ME80252VX-0000-A99	●	24	0.08	1.92	3600	45	0.23	37.0	81	1
ME80252V1-0000-A99	●	24	0.073	1.8	3200	41	0.18	33.0	75	2
ME80252V2-0000-A99	●	24	0.059	1.4	2900	37	0.15	30.0	75	3
ME80252V3-0000-A99	●	24	0.049	1.2	2600	33	0.11	28.0	75	4

Model	◎ 2BALL ○ Sleeve	(VDC)	(AMP)	(WATTS)	(RPM)	(CFM)	(Inch-H_2O)	(dBA)	(g)	Curve
EE80251BX-0000-A99	◎	12	0.172	2.06	3600	45	0.23	38.0	89	1
EE80251B1-0000-A99	◎	12	0.145	1.74	3200	41	0.18	33.0	74	2
EE80251B2-0000-999	◎	12	0.12	1.44	2900	37	0.15	30.0	74	3
EE80251B3-0000-999	◎	12	0.09	1.08	2600	33	0.11	28.0	74	4
EE80252BX-0000-A99	◎	24	0.08	1.92	3600	45	0.23	38.0	89	1
EE80252B1-0000-A99	◎	24	0.073	1.8	3200	41	0.18	33.0	75	2
EE80252B2-0000-999	◎	24	0.059	1.41	2900	37	0.15	30.0	75	3
EE80252B3-0000-999	◎	24	0.046	1.1	2600	33	0.11	28.0	75	4
EE80251S1-0000-A99	○	12	0.145	1.7	3200	41	0.18	33.0	75	2
EE80251S2-0000-999	○	12	0.12	1.4	2900	37	0.15	30.0	75	3
EE80251S3-0000-999	○	12	0.09	1.1	2600	33	0.11	28.0	75	4
EE80252S1-0000-A99	○	24	0.073	1.8	3200	41	0.18	33.0	75	2
EE80252S2-0000-999	○	24	0.059	1.4	2900	37	0.15	30.0	75	3
EE80252S3-0000-999	○	24	0.049	1.2	2600	33	0.11	28.0	75	4

* Preliminary specification for reference.







*All model could be customized. Please contact with Sunon Sales.
*Specifications subject to change without notice. Please Visit SUNON web site at http://www.sunon.com for update information.

C. PN2222A datasheet

ON Semiconductor™

Amplifier Transistors

NPN Silicon

P2N2222A

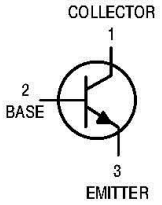
CASE 29–11, STYLE 17
TO–92 (TO–226AA)

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Collector–Emitter Voltage	V_{CEO}	40	Vdc
Collector–Base Voltage	V_{CBO}	75	Vdc
Emitter–Base Voltage	V_{EBO}	6.0	Vdc
Collector Current — Continuous	I_C	600	mAdc
Total Device Dissipation @ T_A = 25°C Derate above 25°C	P_D	625 5.0	mW mW/°C
Total Device Dissipation @ T_C = 25°C Derate above 25°C	P_D	1.5 12	Watts mW/°C
Operating and Storage Junction Temperature Range	T_J, T_{stg}	–55 to +150	°C

THERMAL CHARACTERISTICS

Characteristic	Symbol	Max	Unit
Thermal Resistance, Junction to Ambient	$R_{\theta JA}$	200	°C/W
Thermal Resistance, Junction to Case	$R_{\theta JC}$	83.3	°C/W



ELECTRICAL CHARACTERISTICS (T_A = 25°C unless otherwise noted)

Characteristic	Symbol	Min	Max	Unit
OFF CHARACTERISTICS				
Collector–Emitter Breakdown Voltage (I_C = 10 mAdc, I_B = 0)	$V_{(BR)CEO}$	40	—	Vdc
Collector–Base Breakdown Voltage (I_C = 10 μAdc, I_E = 0)	$V_{(BR)CBO}$	75	—	Vdc
Emitter–Base Breakdown Voltage (I_E = 10 μAdc, I_C = 0)	$V_{(BR)EBO}$	6.0	—	Vdc
Collector Cutoff Current (V_{CE} = 60 Vdc, $V_{EB(off)}$ = 3.0 Vdc)	I_{CEX}	—	10	nAdc
Collector Cutoff Current (V_{CB} = 60 Vdc, I_E = 0) (V_{CB} = 60 Vdc, I_E = 0, T_A = 150°C)	I_{CBO}	 — —	 0.01 10	μAdc
Emitter Cutoff Current (V_{EB} = 3.0 Vdc, I_C = 0)	I_{EBO}	—	10	nAdc
Collector Cutoff Current (V_{CE} = 10 V)	I_{CEO}	—	10	nAdc
Base Cutoff Current (V_{CE} = 60 Vdc, $V_{EB(off)}$ = 3.0 Vdc)	I_{BEX}	—	20	nAdc

March, 2001 – Rev. 1
Publication Order Number:
P2N2222A/D

P2N2222A

ELECTRICAL CHARACTERISTICS (T_A = 25°C unless otherwise noted) (Continued)

Characteristic	Symbol	Min	Max	Unit
ON CHARACTERISTICS				
DC Current Gain	h_{FE}			—
(I_C = 0.1 mAdc, V_{CE} = 10 Vdc)		35	—	
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc)		50	—	
(I_C = 10 mAdc, V_{CE} = 10 Vdc)		75	—	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, T_A = −55°C)		35	—	
(I_C = 150 mAdc, V_{CE} = 10 Vdc)(1)		100	300	
(I_C = 150 mAdc, V_{CE} = 1.0 Vdc)(1)		50	—	
(I_C = 500 mAdc, V_{CE} = 10 Vdc)(1)		40	—	
Collector–Emitter Saturation Voltage(1)	$V_{CE(sat)}$			Vdc
(I_C = 150 mAdc, I_B = 15 mAdc)		—	0.3	
(I_C = 500 mAdc, I_B = 50 mAdc)		—	1.0	
Base–Emitter Saturation Voltage(1)	$V_{BE(sat)}$			Vdc
(I_C = 150 mAdc, I_B = 15 mAdc)		0.6	1.2	
(I_C = 500 mAdc, I_B = 50 mAdc)		—	2.0	
SMALL–SIGNAL CHARACTERISTICS				
Current–Gain — Bandwidth Product(2) (I_C = 20 mAdc, V_{CE} = 20 Vdc, f = 100 MHz)	f_T	300	—	MHz
Output Capacitance (V_{CB} = 10 Vdc, I_E = 0, f = 1.0 MHz)	C_{obo}	—	8.0	pF
Input Capacitance (V_{EB} = 0.5 Vdc, I_C = 0, f = 1.0 MHz)	C_{ibo}	—	25	pF
Input Impedance	h_{ie}			kΩ
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		2.0	8.0	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		0.25	1.25	
Voltage Feedback Ratio	h_{re}			X 10^{-4}
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		—	8.0	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		—	4.0	
Small–Signal Current Gain	h_{fe}			—
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		50	300	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		75	375	
Output Admittance	h_{oe}			μmhos
(I_C = 1.0 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		5.0	35	
(I_C = 10 mAdc, V_{CE} = 10 Vdc, f = 1.0 kHz)		25	200	
Collector Base Time Constant (I_E = 20 mAdc, V_{CB} = 20 Vdc, f = 31.8 MHz)	$rb'C_c$	—	150	ps
Noise Figure (I_C = 100 μAdc, V_{CE} = 10 Vdc, R_S = 1.0 kΩ, f = 1.0 kHz)	N_F	—	4.0	dB
SWITCHING CHARACTERISTICS				
Delay Time	(V_{CC} = 30 Vdc, $V_{BE(off)}$ = −2.0 Vdc, I_C = 150 mAdc, I_{B1} = 15 mAdc) (Figure 1)	t_d	—	10	ns
Rise Time		t_r	—	25	ns
Storage Time	(V_{CC} = 30 Vdc, I_C = 150 mAdc, I_{B1} = I_{B2} = 15 mAdc) (Figure 2)	t_s	—	225	ns
Fall Time		t_f	—	60	ns

1. Pulse Test: Pulse Width ≤ 300 μs, Duty Cycle ≤ 2.0%.
2. f_T is defined as the frequency at which $|h_{fe}|$ extrapolates to unity.

P2N2222A

SWITCHING TIME EQUIVALENT TEST CIRCUITS

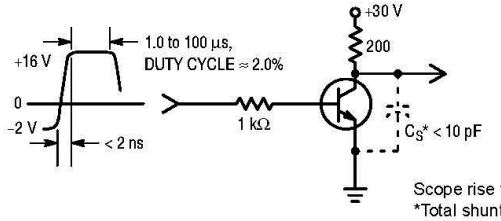
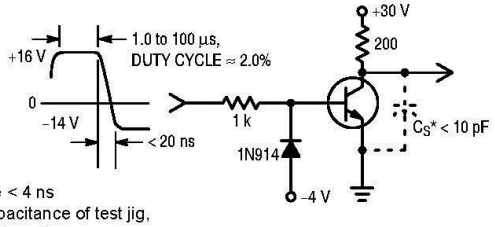


Figure 1. Turn–On Time



Scope rise time < 4 ns
*Total shunt capacitance of test jig, connectors, and oscilloscope.

Figure 2. Turn–Off Time

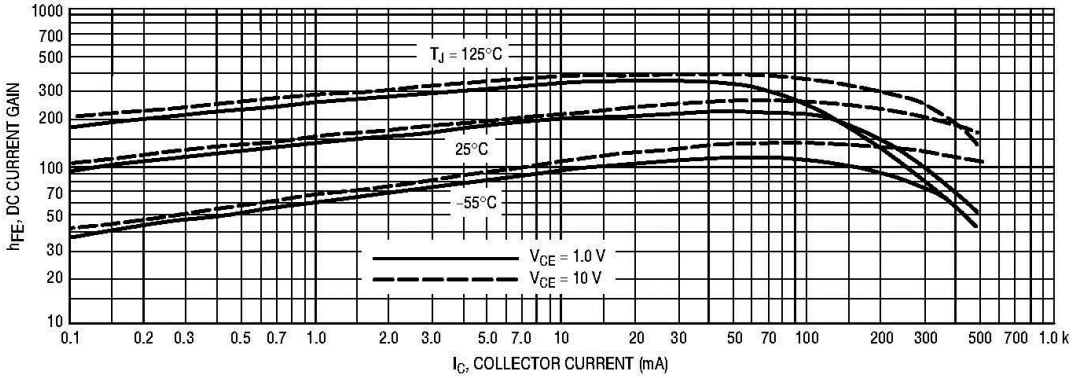


Figure 3. DC Current Gain

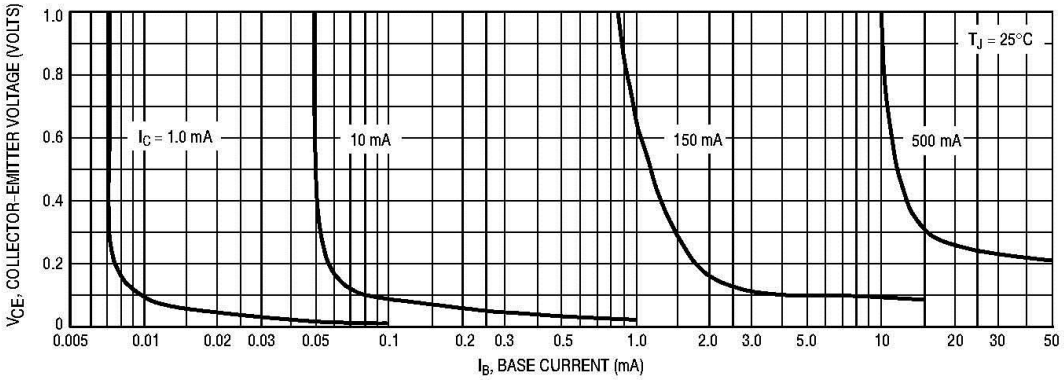


Figure 4. Collector Saturation Region

P2N2222A

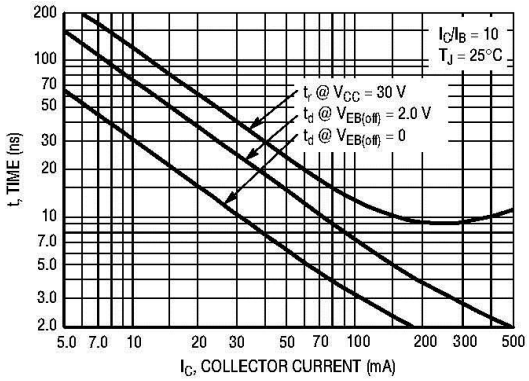


Figure 5. Turn–On Time

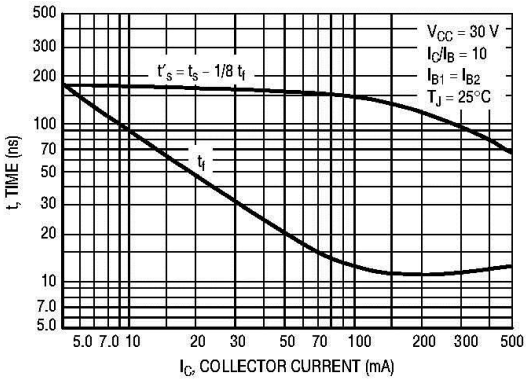


Figure 6. Turn–Off Time

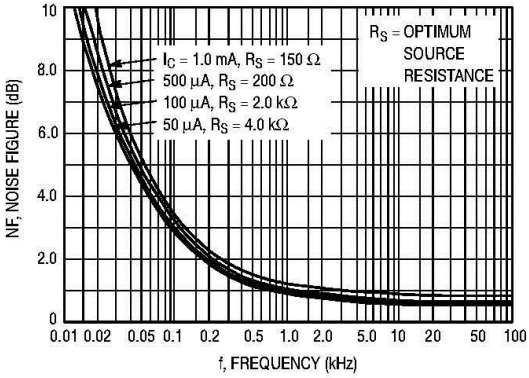


Figure 7. Frequency Effects

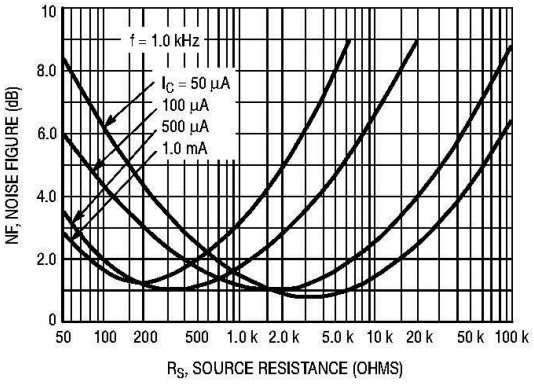


Figure 8. Source Resistance Effects

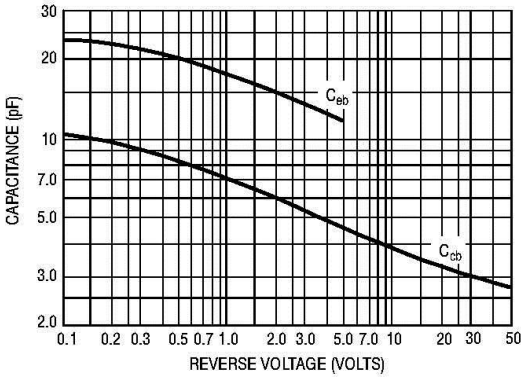


Figure 9. Capacitances

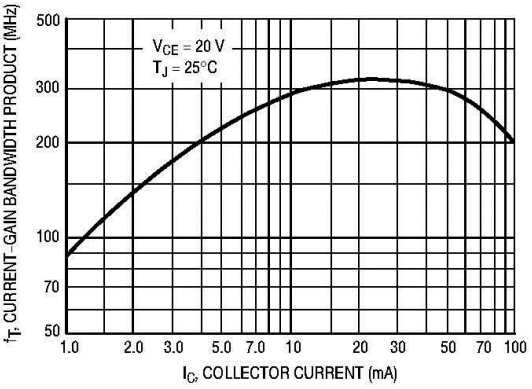


Figure 10. Current–Gain Bandwidth Product

D. Datasheet 3

CAN STACK STEP MOTORS

20M SERIES

SMALL BUT POWERFUL

GENERAL SPECIFICATIONS

Step Angle	18°
Step Accuracy	± 1.5°
Operating Temperature	100°C Max
Ambient Temperature Range	-20°C ~ +70°C
Insulation Resistance at 500Vdc	100MΩ
Dielectric Withstanding Voltage	450 ± 50 VRMS, 2 sec

CLICK HERE TO PRINT THIS SERIES
OPENS PDF IN NEW WINDOW

The specifications in this publication are believed to be accurate and reliable. However, it is the responsibility of the product user to determine the suitability of Portescap products for a specific application. While defective products will be replaced without charge if promptly returned, no liability is assumed beyond such replacement.

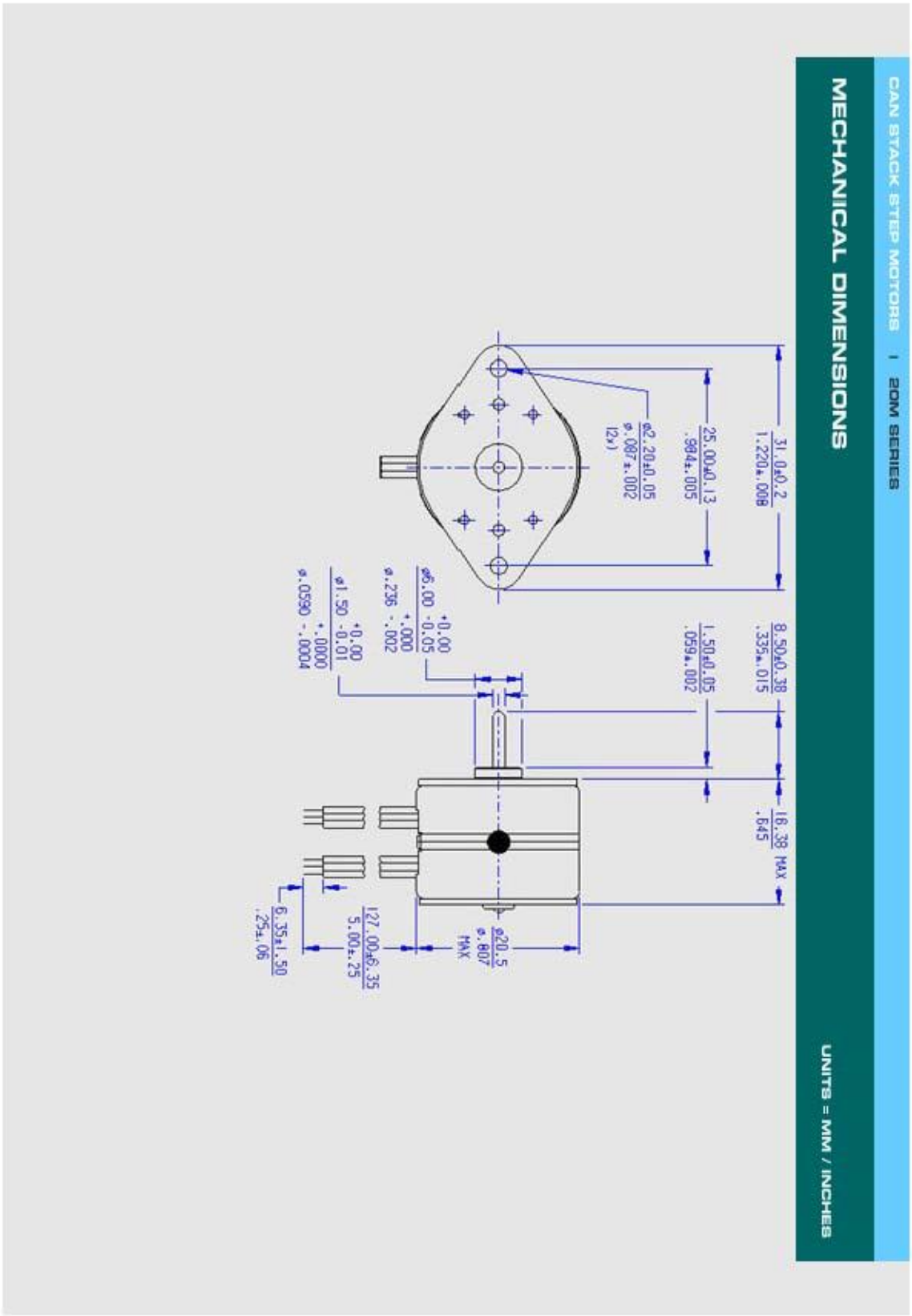
Portescap Danaher Motion motors will not be CE marked where the Low Voltage Directive, the Electro-Magnetic Compatibility or other appropriate EU directives are not applicable - this is an EU legal requirement.

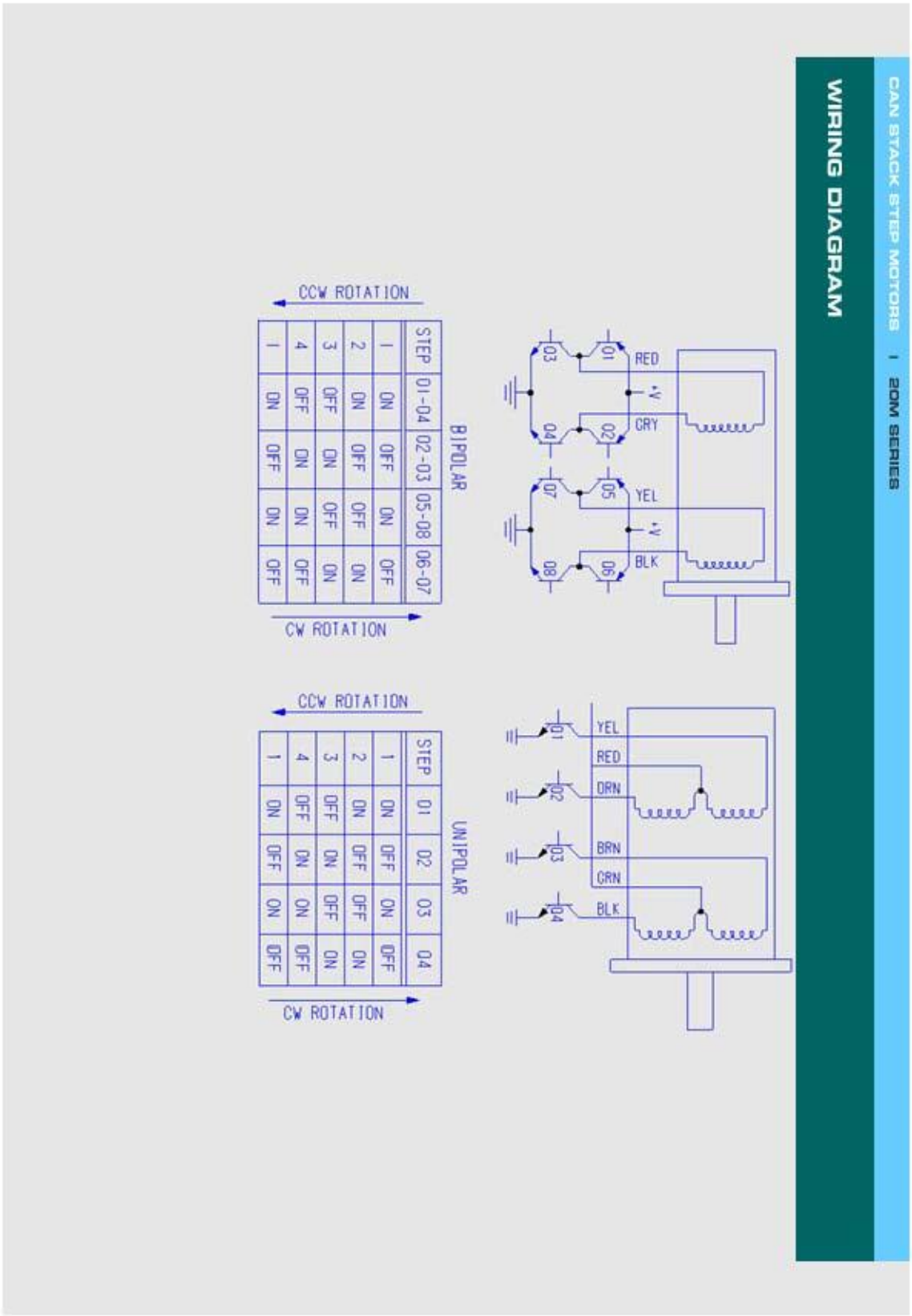
CAN STACK STEP MOTORS | 20M SERIES

TECHNICAL SPECIFICATIONS

	UNIPOLAR		BIPOLAR	
Part Number	20M020D1U	20M020D2U	20M020D1B	20M020D2B
DC Operating Voltage	5	12	5	12
Resistance per Winding (ohms)	20	115.2	20	115.2
Inductance per Winding (mH)	3.9	20.3	7.8	52.8
Holding Torque* (mNm/oz-in)	7.77 / 1.10	7.77 / 1.10	11.30 / 1.60	11.30 / 1.60
Rotor Moment of Inertia ($g.m^2$)	4.1×10^{-5}	4.1×10^{-5}	4.1×10^{-5}	4.1×10^{-5}
Detent Torque (mNm/oz-in)	3.53 / 0.50	3.53 / 0.50	3.53 / 0.50	3.53 / 0.50
Step Angle	18°	18°	18°	18°
Step Angle Tolerance*	± 1.5°	± 1.5°	± 1.5°	± 1.5°
Steps per Revolution*	20	20	20	20
Max. Operating Temperature	100°C	100°C	100°C	100°C
Ambient Temperature Range				
Operating	-20°C to 70°C	-20°C to 70°C	-20°C to 70°C	-20°C to 70°C
Storage	-40°C to 85°C	-40°C to 85°C	-40°C to 85°C	-40°C to 85°C
Bearing Type	Sintered bronze sleeve	Sintered bronze sleeve	Sintered bronze sleeve	Sintered bronze sleeve
Insulation Resistance at 500Vdc	100 megohms	100 megohms	100 megohms	100 megohms
Dielectric Withstanding Voltage	450 ± 50 VRMS, 2 sec	450 ± 50 VRMS, 2 sec	450 ± 50 VRMS, 2 sec	450 ± 50 VRMS, 2 sec
Weight (g/oz)	23.5 / 0.83	23.5 / 0.83	23.5 / 0.83	23.5 / 0.83
Leadwires	28 AWG, UL Style 1429	28 AWG, UL Style 1429	28 AWG, UL Style 1429	28 AWG, UL Style 1429

* Measured with 2 phases energized





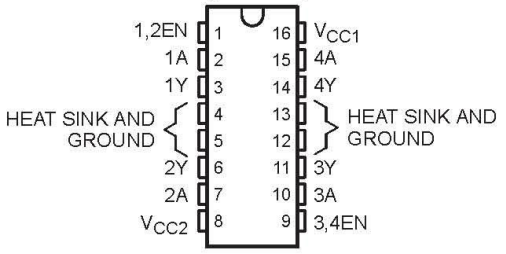
E. Texas Instruments L293 H-Bridge

L293, L293D
QUADRUPLE HALF-H DRIVERS

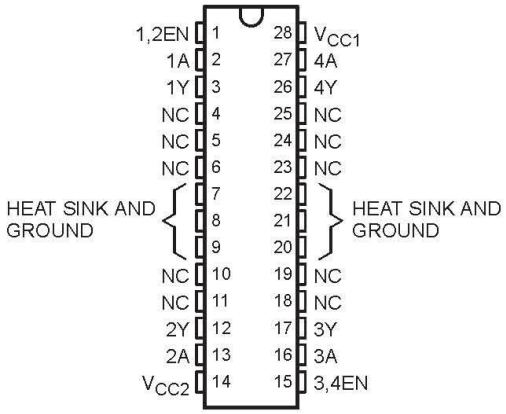
SLRS008C – SEPTEMBER 1986 – REVISED NOVEMBER 2004

- **Featuring Unitrode L293 and L293D Products Now From Texas Instruments**
- **Wide Supply-Voltage Range: 4.5 V to 36 V**
- **Separate Input-Logic Supply**
- **Internal ESD Protection**
- **Thermal Shutdown**
- **High-Noise-Immunity Inputs**
- **Functionally Similar to SGS L293 and SGS L293D**
- **Output Current 1 A Per Channel (600 mA for L293D)**
- **Peak Output Current 2 A Per Channel (1.2 A for L293D)**
- **Output Clamp Diodes for Inductive Transient Suppression (L293D)**

L293 . . . N OR NE PACKAGE
L293D . . . NE PACKAGE
(TOP VIEW)



L293 . . . DWP PACKAGE
(TOP VIEW)



description/ordering information

The L293 and L293D are quadruple high-current half-H drivers. The L293 is designed to provide bidirectional drive currents of up to 1 A at voltages from 4.5 V to 36 V. The L293D is designed to provide bidirectional drive currents of up to 600-mA at voltages from 4.5 V to 36 V. Both devices are designed to drive inductive loads such as relays, solenoids, dc and bipolar stepping motors, as well as other high-current/high-voltage loads in positive-supply applications.

All inputs are TTL compatible. Each output is a complete totem-pole drive circuit, with a Darlington transistor sink and a pseudo-Darlington source. Drivers are enabled in pairs, with drivers 1 and 2 enabled by 1,2EN and drivers 3 and 4 enabled by 3,4EN. When an enable input is high, the associated drivers are enabled, and their outputs are active and in phase with their inputs. When the enable input is low, those drivers are disabled, and their outputs are off and in the high-impedance state. With the proper data inputs, each pair of drivers forms a full-H (or bridge) reversible drive suitable for solenoid or motor applications.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PART NUMBER	TOP-SIDE MARKING
0°C to 70°C	HSOP (DWP)	Tube of 20	L293DWP	L293DWP
	PDIP (N)	Tube of 25	L293N	L293N
	PDIP (NE)	Tube of 25	L293NE	L293NE
		Tube of 25	L293DNE	L293DNE

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

L293, L293D
QUADRUPLE HALF-H DRIVERS

SLRS008C – SEPTEMBER 1986 – REVISED NOVEMBER 2004

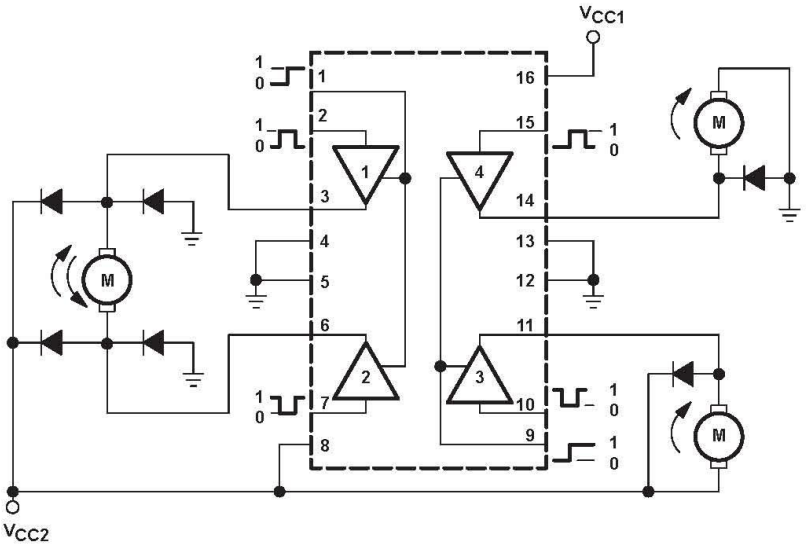
description/ordering information (continued)

On the L293, external high-speed output clamp diodes should be used for inductive transient suppression.

A V_{CC1} terminal, separate from V_{CC2}, is provided for the logic inputs to minimize device power dissipation.

The L293and L293D are characterized for operation from 0°C to 70°C.

block diagram



NOTE: Output diodes are internal in L293D.

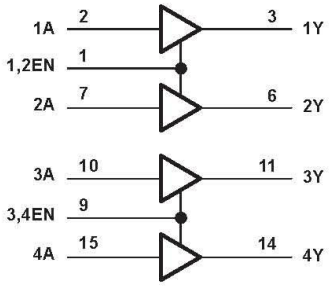
FUNCTION TABLE
(each driver)

INPUTS†		OUTPUT
A	EN	Y
H	H	H
L	H	L
X	L	Z

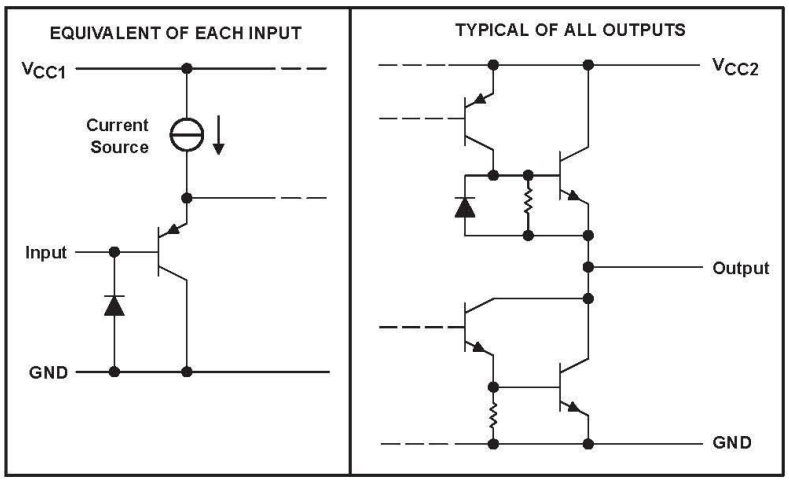
H = high level, L = low level, X = irrelevant,
Z = high impedance (off)

† In the thermal shutdown mode, the output is in the high-impedance state, regardless of the input levels.

logic diagram



schematics of inputs and outputs (L293)



F. MILONE eTape Continuous Fluid Level Sensor

eTape™ Continuous Fluid Level Sensor PN-12110215TC

Description

The eTape sensor is a solid state, continuous (multi-level) fluid level sensor for measuring levels in water, non-corrosive water based liquids and dry fluids (powders). The eTape sensor is manufactured using printed electronic technologies which employ additive direct printing processes to produce functional circuits.

Theory of Operation

The eTape sensor's envelope is compressed by hydrostatic pressure of the fluid in which it is immersed resulting in a change in resistance which corresponds to the distance from the top of the sensor to the fluid surface. The eTape sensor provides a resistive output that is inversely proportional to the level of the liquid: the lower the liquid level, the higher the output resistance; the higher the liquid level, the lower the output resistance.

Specifications

Part Number	PN-12110215TC-8	PN-12110215TC-12	PN-12110215TC-24	PN-12110215TC-32
Nominal Length	8-inch	12-inch	24-inch	32-inch
Sensor Length	10.2" (259 mm)	14.2" (361 mm)	26.0" (660 mm)	34.2" (869 mm)
Active Length	8.4" (213 mm)	12.4" (315 mm)	24.34" (618 mm)	32.4" (823 mm)
Sensor Output	400-1500Ω ±20%	400-2000Ω ±20%	400-3000Ω ±20%	400-5000Ω ±20%
Ref Resistance	1500Ω ±20%	2000Ω ±20%	3000Ω ±20%	5000Ω ±20%

Thickness: 0.015" (0.381mm)

Width: 1.0" (25.4 mm)

Actuation Depth: Nominal 1" (25.4 mm)

Resolution: < 0.01" (0.25 mm)

Resistance Gradient: 150Ω /inch (60Ω/cm)

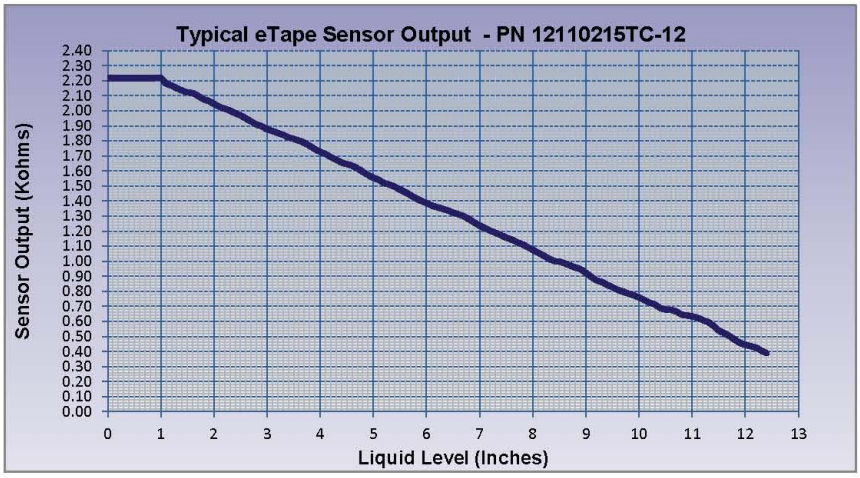
Connector: Male Crimpflex Pins

Power Rating: 0.5 Watts (VMax = 10V)

Temperature Range: 15°F - 150°F (-9°C - 65°C)

Sensor Output

The eTape can be modeled as a variable resistor. The typical output characteristics of the eTape sensor are show in the figure below:

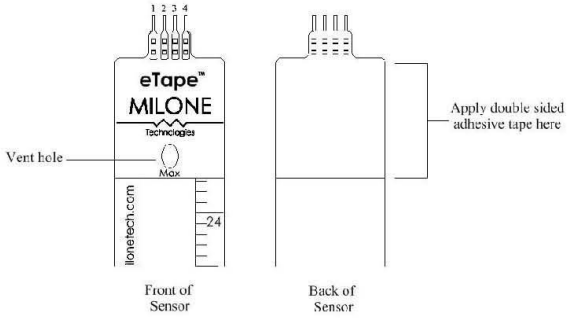


Milone Technologies, Inc - 17 Ravenswood Way - Sewell, New Jersey 08080 - Phone: (856) 270-2688
Email: info@milonetech.com Web: www.milonetech.com

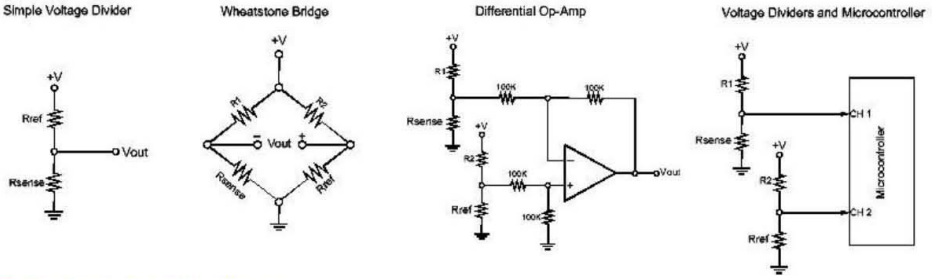
eTape™ Continuous Fluid Level Sensor PN-12110215TC-12

Connection and Installation

Connect to the eTape by attaching a 4 pin connector with pre-soldered wires to the Crimpflex pins. Do not solder directly to the Crimpflex pins. The inner two pins (pins 2 and 3) are the sensor output (Rsense). The outer pins (pins 1 and 4) are the reference resistor (Rref) which can be used for temperature compensation. Suspend the eTape sensor in the fluid to be measured. To work properly the sensor must remain straight and must not be bent vertically or longitudinally. For best results install the sensor inside a section of 1-inch diameter PVC pipe. Double sided adhesive tape may be applied to the upper back portion of the sensor to suspend the sensor in the container to be measured. However, the liquid must be allowed to interact freely with both sides of the sensor. The vent hole located above the max line allows the eTape to equilibrate with atmospheric pressure. The vent hole is fitted with a hydrophobic filter membrane to prevent the eTape from being swamped if inadvertently submerged.



Sample Circuits



Custom Applications

The eTape sensor can be manufactured in custom lengths to fit any application. Contact Milone Technologies if you have an application that requires specific length, configuration or output characteristics.

Technical Support

If you require technical support for the eTape liquid level sensor, please contact our technical support department by email at: techsupport@milonetech.com.

Innovative Fluid Sensing

MILONE Technologies

Milone Technologies, Inc - 17 Ravenswood Way - Sewell, New Jersey 08080 - Phone: (856) 270-2688
Email: info@milonetech.com Web: www.milonetech.com

Bibliography

[1] Wikipedia, "ASCII," 9 March 2013. [Online]. Available: http://en.wikipedia.org/wiki/ASCII. [Accessed 4 4 2013].

[2] Sparkfun Electronics, "Barometric Pressure Sensor - BMP085 Breakout," Sparkfun Electronics, [Online]. Available: https://www.sparkfun.com/products/11282. [Accessed 13 April 2013].

[3] A. Industries, "MicroSD card breakout board+ -," [Online]. Available: http://www.adafruit.com/products/254. [Accessed 16 August 2013].

[4] Wikipedia, "Serial Peripheral Interface Bus," [Online]. Available: http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus. [Accessed 16 August 2013].

[5] A. Mega, "Arduino Mega 2560," [Online]. Available: Arduino Mega 2560. [Accessed 16 August 2013].

[6] Wikipedia, Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Piano_key_frequencies. [Accessed 2 5 2014].

[7] S. Pictures, "This is Jeopardy," [Online]. Available: http://www.jeopardy.com/. [Accessed 2 5 2014].

[8] Lumex, *LDS-SMHTA302RISUGT Data Sheet,* 290 E. Helen Rooad, Palatine, IL 60067-6976: Lumex, 2012.

[9] Lumex, *LDS-C324RI Datasheet,* Lumex, 1993.

[10] I. Milone Technologies, *eTape Continuous Fluid Level Sensor PN-12110215TC,* Sewell, New Jersey: MILONE.

[11] [. t. H. Ref]. [Online].

Index