EECS 461 Fall 2009
Lab 8: Introduction to Rapid Prototyping

1 Overview

The purpose of this exercise is to create programs for the MPC5553 by generating code from a Simulink model. This method of creating programs from high-level modeling languages, such as Simulink and Stateflow, is commonly referred to as rapid-prototyping. It can significantly reduce the amount of time required to develop a complex embedded control system. The software tools we use:

- From The Mathworks:
  - MATLAB
  - Real Time Worskhop
  - Real Time Workshop Embedded Coder
- From Freescale Semiconductor:
  - RAppID Toolbox, a Simulink blockset
  - OSEK Turbo, a real time operating system
- From the University of Michigan
  - Resource Allocation for OSEK Turbo

In this lab, you will first replicate the adding program from Lab 1 and the virtual wall from Lab 4 as Simulink models, and then generate C code for the MPC5553 directly from these models. You will then create a virtual world that has two virtual spring/inertia/damper systems coupled to the haptic wheel. These systems have significantly different time constants that suggest they be implemented in a multitasking environment.

NOTE: This lab document is structured differently than the previous ones – there are three separate lab tasks, each with a Pre-Lab, In-Lab, and Post-Lab. In the Pre-Lab, you will create a model to represent the desired system behavior. You will test this model using virtual inputs and outputs available in Simulink. In the In-Lab exercise, you will use the RAppID Toolbox blockset, available on the lab computers, to connect physical inputs and outputs to the model you generated. You will then use Simulink to generate C code and a binary. The Post-Lab will ask you to look at some of the code generated.

NOTE: Simulink models and simulations are to be completed individually. As a reminder, each Pre-Lab and Post-Lab question is to be completed individually.

2 Adding Two Numbers

In this section, you will use Simulink to do some simple arithmetic. In lab, you will use one of these models to implement a 4-bit adder, similar to the one you wrote in C for Lab 1, generate code for it, and run it on the MPC5553.

NOTE: The autocode generation blocks are not available outside of the lab. You will **not** need them to complete the Pre-Lab. You may, of course, still use the lab computers to work on your Pre-Lab during open lab hours.

2.1 Pre-Lab Assignment

All of these pre-lab questions must be done individually and handed in at the start of your lab section.

1. Before coming to lab, you need to create several Simulink subsystems that encapsulate commonly used bit operations. Just like functions in C, a Simulink subsystem forms a high-level abstraction of a complicated and frequently used calculation. The input ports of a subsystem bring data into the subsystem, like the input parameters do for a C function. The output ports perform the same function as the return values of a C function.

   Once you have thoroughly tested a particular subsystem, you will be able to build more complicated models quickly and spend less time debugging, by reusing the tested subsystems.

   Create subsystems that implement the following functions:

   (a) Given a 32-bit unsigned integer as an input signal, extract the four bytes that make up the signal and form one output signal with four 8-bit unsigned integers muxed together. A diagram of this system is found in Figures 2-3 of the notes "Simulink Models for Autocode Generation."

   (b) Reconstruct a 32-bit unsigned integer from four 8-bit unsigned integers muxed into one input signal. This subsystem should implement the inverse operation of the first subsystem. A diagram of this system is also found in Figures 4-5 of the notes. Simulink Models for Autocode Generation.

   [5] HAND IN: Printouts of the two subsystems you constructed. For the first subsystem, show that it correctly translates the 32 bit binary representation of $\mathbf{2^{32} - 1}$ into four 8-bit pieces. For the second subsystem, show that it correctly reverses this operation. Use "Display" blocks in Simulink to check and show the validity of your model.

2. Create a model to add two 4-bit numbers, as shown in Figure 1 (Page 4). By changing the values in the `Constant` blocks, we can specify two 4-bit numbers. These 8 blocks are the 8 binary input values. Notice that the output `Display` blocks do not change value until you run the model.

   [5] HAND IN: A printout of your model, showing it correctly adding the two numbers **7** and **5**.

2.2 In-Lab Assignment

The RAppID Toolbox blockset is found in the Simulink library browser under `RAppID` and `RAppID-Toolbox`.

1. Open the model you created to add two 4-bit numbers in the Pre-Lab. You will add device-driver and initialization blocks to this model so that it can be run on the MPC5553.

2. Drag the block titled `RAppID-EC` into the top level (not a subsystem) of your model. It can be found under `RAppID/RAppID Initialization Blocks`. This block performs two functions: it configures the build system (by locating the compiler, choosing the hardware, etc.), and it generates processor and peripheral initialization code.

3. Create a subsystem to encapsulate the single task for this model. Rename this subsystem to something appropriate, like "Adder Task." Place the adder blocks into this subsystem. Place a `Trigger` block into this subsystem. It can be found under `Simulink/Ports and Subsystems`. Open the Trigger block, and change the "Trigger type" parameter to "function-call."

4. In the top-level of your model, place a `Function-Call Generator` block. Connect its output to the top (trigger) port of the subsystem containing the adder blocks. Open the Function-Call Generator block, and change the "Sample time" parameter to 0.1. This will cause the adder task to be executed 10 times per second.

5. Now, we have a model which can be run on the MPC5553. However, we want the model to interact with the DIP switches and LEDs. So, we will now replace each Constant block with a General Purpose Input block, and Display block with a General Purpose Output block.

   (a) Open the adder task, and then open one of the subsystems that obtains an input number. Delete each Constant block, and replace it with a General Purpose Input block. The block can be found in `RAppID-Toolbox/Peripheral Driver Blocks/GPIO Blocks`.

   (b) Open each GPI block, and set the pin number. Repeat for the other input subsystem.

   (c) Open the subsystem that displays the final sum. Delete each Display block, and replace it with a General Purpose Output block. Open each GPO block, and set the pin number.

6. Type **ctrl-d** to update the diagram. This checks the consistency of the model. If it succeeds without errors, your model is ready for code generation. (You may need to update the model twice initially to update without errors.)

7. Verify that the current working directory of MATLAB is set to a directory where you would like the model code to be generated.

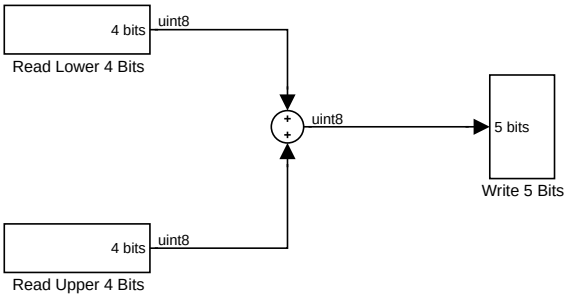
8. Build the model by typing **ctrl-b** or selecting "Build Model" in "Tools->Real-Time Workshop." After a successful build, Real Time Workshop will echo:

   ```
   ### Successful completion of Real-Time Workshop build procedure for model: Adder
   ```

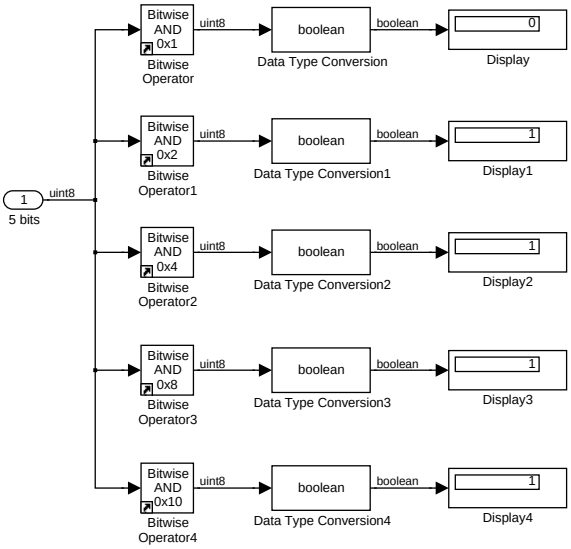
9. The ELF binary file is located in `<model_name>_rappid_rtw/`. Load and test your program.

10. Try using different task rates, by changing the rate of the `Function-Call Generator`. Rebuild the model and observe the effect for a few different rates (faster and slower).

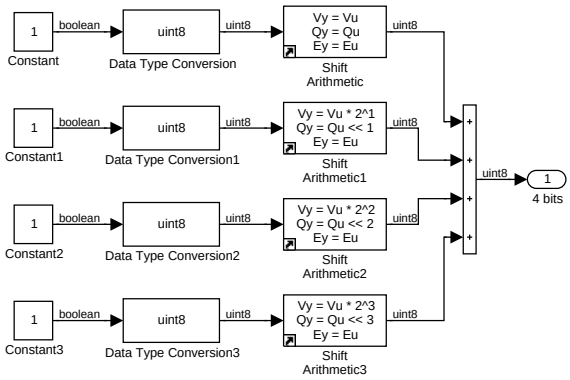
11. Before you leave the lab, compare the sizes of the ELF files generated for your adder in this lab and the one you generated in lab 1. Compare, also, the amount of C code that went into each one.



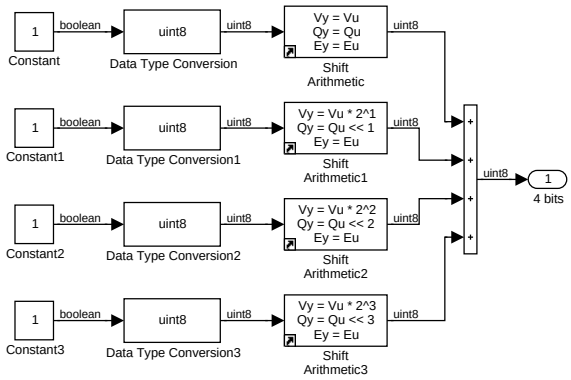
(a) Top-level system.



(b) Write 5 Bits system.



(c) Read Lower 4 Bits system.



(d) Read Upper 4 Bits system.

Figure 1: Block diagrams of a 4-bit adder.

2.3 Post-Lab Assignment

1. **[3]** Explain the benefits and downsides of automatic code generation from a Simulink model, considering a) the generated ELF size, b) the generated C code size, and c) the development time taken. Why is the automatically generated code so much larger in this case? Will this always be true?

3 The Virtual Wall

3.1 Pre-Lab Assignment

All of these pre-lab questions must be done individually and handed in at the start of your lab section.

Next you will implement a virtual wall. You will model the wall in this Pre-Lab, add device-driver blocks and test the wall in lab, and then inspect the code for the Post-Lab.

Instead of encoding the values that represent unit conversions and system properties directly into your Simulink model, use variables to these quantities. Make a *.m* file which sets variables that appear in the workspace, and make the m-file run automatically. To do this, enter the name of the m-script (without the *.m*) under the "File/Model Properties/Callbacks/InitFcn" menu from window of the Simulink diagram. This script will be run each time the model is updated.

1. Figure 2 on Page 9 is a Simulink model of a subsystem that converts haptic motor position, in encoder counts, to haptic wheel position, in degrees. What is the purpose of each block in this diagram? As shown in Figure 8 of the handout "Simulink Models for Autocode Generation," this subsystem will be used in conjunction with the TPU/FQD device-driver block to produce wheel position in degrees.

   [3] HAND IN: A printout of Figure 2 (Page 9) with the functionality of each block correctly identified.

2. Build the model we will use to simulate and generate code:

   (a) Create a new, blank Simulink model.

   (b) Create a subsystem, and call it "Simulate Virtual Wall." In this block we will translate the position of the virtual wheel in degrees into a torque in N-mm.

   (c) From the "Simulation" menu, select "Configuration Parameters..." and locate the "Max step size" edit box. Replace "auto" with "0.01" to represent at least 100 model steps per second.

   (d) Add or delete ports until you have one input and one output port. Name the input port "Wheel Position (deg)." Name the output port "Motor Torque (N-mm)."

   (e) Recall that the torque at the wheel is related to the position of the wheel by the following piecewise equation:

```
if( Current_Position - Wall_Position > 0 )        /* Inside the wall */
     torque = k * (Wall_Position - Current_Position)
else                                              /* Outside the wall */
     torque = 0
```

   Calculate this torque using a Stateflow chart. Use a wall position of 0.

   (f) As input, instead of the current wheel angle derived from the encoder, place a `Step` block, which is located under `Simulink/Sources`. Place this block at the top level of your model, and connect its output to the input of the Simulate Virtual Wall subsystem. Change the Final Value parameter to 10 (for 10 degrees). This will mimic a sudden turn of the wheel. Connect the output of your Stateflow chart to the input of a `Saturation` block. Set the upper limit in the `Saturation` block to 825 and the lower limit to -825. Connect the output of the `Saturation` block to the output of the Simulate Virtual Wall subsystem.

   (g) Instead of applying the resulting torque to the haptic motor, graph the command torque using a Scope block. Place the Scope at the top level of your model, and connect the output of the Simulate Virtual Wall subsystem to the input of the Scope.

   [2] HAND IN: A printout of your model.

3. Simulate the system for 5 seconds. You can change the simulation time in the "Configuration Parameters" dialog, found under the "Simulation" menu of your model. Inspect the resulting graph. You may also attach a Scope block to the output of the Step block, for comparison. The torque should respond as the wheel angle is increased.

   (a) Explain the shape of the output torque plot.

   (b) What is the effect of the `Saturation` block? What would the resulting torque graph look like if we did not limit the torque applied to the haptic wheel?

   (c) Replace the `Step` with a `Ramp` block. Change the Slope parameter to 10 and Initial Output parameter to -20. This will mimic turning the wheel slowly. Simulate the system again, and answer the questions above for this input.

   [5] HAND IN: Printouts of the resulting torque. The answers to these questions.

3.2 In-Lab Assignment

1. Open your Virtual Wall model. Add the `RAppID Initialization` block.

2. As before, place the blocks that make up the wall into a Task. Name the subsystem "VirtualWallTask." Open the Task, and delete the simulated input and Scope blocks.

3. Add a subsystem to represent the conversion between encoder ticks and haptic degrees. Into this subsystem, add blocks to make Figure 2 on Page 9.

4. Add the `Quadrature Decoder` block, found in `RAppID-Toolbox/Peripheral Driver Blocks/eTPU Blocks`. Open the block parameters, and choose Channel 0, set the encoder resolution properly, and choose a Position Count Scaling of 4.

5. Connect the Position Count output to the input of the conversion subsystem. Connect the output of the conversion subsystem to the input of the virtual wall blocks. Add terminators to the Velocity and Direction outputs of the quadrature block. We will not use these outputs, and must inform Simulink. Terminators are found under `Simulink/Sinks`.

6. Add a subsystem to represent the conversion between commanded torque and duty cycle. The first step in this subtask is to convert your input (a torque in N-mm) to a duty cycle ranging from 0 to 1. Recall from Lab 6 that the equation relating torque at the wheel to duty cycle is:

   ```
   Duty Cycle = 0.50 + 5.15e-4 * torque
   ```

   Once the duty cycle is calculated, use a saturation block to limit its value to the range 35% to 65%.

7. Finally, we need to add a PWM block to output the desired duty cycle to the motor. Within the VirtualWallTask subsystem, add an `eMIOS Output PWM` block. Choose an initial Duty Cycle and Frequency of 50% and 20 kHz, respectively. This block takes two inputs: a duty cycle value (0-100) and a frequency value.

8. For the frequency input, add a Constant block with value 20000. For the duty cycle input, scale the duty cycle to be 0-100, and input this to the block. You will need to cast both inputs to type 'uint32' before connecting them to the PWM Output Block.

9. Save your model, update the diagram by pressing **ctrl-d**, then build it by pressing **ctrl-b**.

3.3 Post-lab Assignment

1. **[2]** Find and submit the part of the generated C code that implements your virtual wall logic.

4 Two Virtual Spring Inertia Damper Systems

You will now build and implement the system in Section 7 of the handout "Simulink Models for Autocode Generation."

4.1 Pre-lab Assignment

All of these pre-lab questions must be done individually and handed in at the start of your lab section.

1. The files *two_virtual_wheels.m*, *two_virtual_wheel.mdl*, *two_virtual_wheel_discrete.mdl*, and *two_virtual_wheel_params.m* may be found on the class website. Download these, inspect them, and run them to familiarize yourself with how they work.

2. Starting with *two_virtual_wheel_discrete.mdl*, create *two_virtual_wheel_discrete_triggered.mdl*. Separate the fast and slow dynamics into two subsystems. The subsystem containing the fast dynamics will contain the input and out blocks, because we want the system to respond at the fast rate. Add function-call `Triggers` and `Function-Call Generators` as before, to run the fast-dynamics task and slow-dynamics task at 0.002 sec and 0.02 sec respectively. Name the subsystems "FastDynamics" and "SlowDynamics."

3. The *two_virtual_wheel_params.m* script sets various workspace variables that are used as block parameters. We want this script to run each time the model is updated, so make sure it is in "File/Model Properties/Callbacks/InitFcn" dialog as before.

4. Inside the fast task, add a step input. Use a step time of `step_time` and a final value of `Theta_z_0`. We will use this to simulate turning of the haptic wheel. Connect the output of this block to the input of the fast dynamics. The haptic angle will be the input to this system. To allow the slow dynamics also respond to the haptic wheel, we need to bring this value into the slow subsystem. Add a single output port, and connect the haptic angle value to it.

5. Inside the slow task, add an input port. In the next step, we will connect it to the signal representing the haptic angle. Connect the slow dynamics to this input port. Add an output port, and connect the response torque of the slow system to it.

6. Re-open the fast task, and add an input port. This input will represent the amount of torque the slow dynamics are generating. Sum the resulting net torque (from the fast dynamics and slow dynamics) and connect this to a `Scope` block.

7. Now, we must connect the two subsystems. The two tasks run at different rates, so we cannot connect them directly. Add two `Rate Transition` blocks to the top level of your model. Connect the slow-dynamics response torque to the input of the fast-dynamics block by first using one of the rate transition blocks. Do the same for the haptic angle ports.

8. Simulate this system.

9. Compare the plots generated by *two_virtual_wheel.mdl* and *two_virtual_wheel_discrete_triggered.mdl*. How well does the discrete implementation model the actual continuous dynamics?

[5] HAND IN: A printout of your Simulink models. A plot, in the discrete implementation, of the reaction torque obtained from the scope (click on the binoculars to set the axes properly). The answer to the question above.

4.2 In-Lab Assignment

1. Open model *two_virtual_wheel_discrete_triggered.mdl*, and rename it to *two_virtual_wheel_discrete_triggered_rp.mdl*. Place the `RAppID Initialization` block in the top level of the model.

2. Inside the fast task, add the blocks you used before to read, decode, and scale the encoder value. Replace the `Step` input with these blocks.

3. Copy the blocks you used to output torque in the previous part of the lab into the fast task. Replace the `Scope` block with these blocks.

4. Save, update, and build the model.

5. Run the generated ELF binary using the debugger.

6. Now you should have the Simple Target model working on the processor. Go back to your model. We will now set up the embedded Operating System OSEKturbo. To start double click the `RAppID Initialization` block. When the RAppID GUI opens click the Configuration tab. Click Real-Time Operating System and select: OSEK turbo. Save the GUI and exit.

7. Replace the Rate Transition Blocks with the **Resource Allocation blocks** provided by the University. Ask your GSI for clarification and instruction on how to do this.

8. Repeat step 4.

9. Run the generated ELF binary using the debugger. Make sure it behaves the same way as the previous one.

10. The Post-Lab will ask you to inspect some of the generated code. Inside the <`model_name`>`_rappid_rtw/` directory, find and print the files named *two virtual wheel discrete triggered.c*. You will need these for the Post-Lab.

4.3 Post-Lab Assignment

1. [**1**] Find the lines of generated C code that transfer data between tasks (the `Rate Transition` blocks). Copy these lines and explain what is happening.

   You do not need to hand in printouts of your Simulink systems from this section, **just the lines of code specified**.

[20 pre, 25 in, 5 post = 50 total]

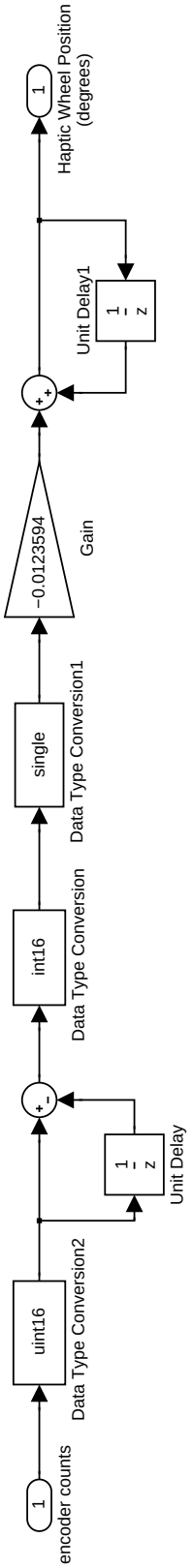


Figure 2: Block diagram for converting encoder counts to haptic wheel degrees.