ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for
Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme
und Regelungstechnik

151-0593-00 **Embedded Control Systems** (Fall 2017) **Lab 7**

Topic: **Code Generation with Simulink**

Pre-Lab due: see NOTE below Post-Lab due: 21. 09. 17 at 5 p.m.

Name: ..... Forename: ..... Initials: .....

marianne.schmid@idsc.mavt.ethz.ch, 21. September 2017

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 autocode generation, and enables model based software development and rapid prototyping. The latter techniques can significantly reduce the amount of time required to develop a complex embedded control system. The software tools we use are:

- From The Mathworks:
  - MATLAB
  - Simulink Coder
  - Embedded Coder
- From Freescale Semiconductor:
  - RAppID Toolbox, a Simulink blockset

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 multi-tasking 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.

For each of the three lab tasks, hand in the Pre-Lab, In-Lab, and Post-Lab for that task at the same time, after completion of the task, and before you move to the next task.

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: Change the display resolution of the virtual machine to 1680x1050 before you open Matlab in order to make sure, that the RAPID tool box works properly.

2.1 Pre-Lab Assignment

1. 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 a simulink model called 'FourByteConvert' and place in it the 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 Figure 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 Figure 5 of the notes "Simulink Models for Autocode Generation".

   NOTE:
   For the first subsystem, show that it correctly translates the 32 bit HEX number **FEDCBA98** 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. For the output of the second subsystem, it will be helpful to configure the Display block so that the result is in HEX format.

   You don't have to hand in a printout of the Simulink blocks. Just show the running code to the assistants.

2. The handout "Analyzing Generated Code" shows how to configure the first subsystem to create a traceable model report when code is generated. Use the same procedure to generate a traceable model report for the second subsystem. For that, copy your second subsystem to a new simulink model file. Be sure to replace the Constant and Display blocks in Figure 4 of the handout "Simulink Models for Autocode Generation" by input port and output port blocks. Set the input port block to specify the correct data type and dimension. Follow the procedure of the handout to generate a traceable model report. You may wish to reuse the subsystem you created above.

   Use the traceable model report to find the lines of code that implement the shifting and summing blocks in Figure 5 of the handout "Simulink Models for Autocode Generation", and submit these lines of code.

3. Create a new simulink model to add two 4-bit numbers. As shown in Figure 1 add the three subsystems and implement them like shown in the Figure 2. By changing the values in the `Constant` blocks in the Read Bits subsystem, we can specify two 4-bit numbers. Together those 8 constant blocks are the 8 binary input values. Notice that the output `Display` blocks do not change value until you run the model.

   NOTE: You don't have to hand in a printout of the Simulink blocks. Just test your system, that it correctly adds the two numbers, e.g. **7** and **5**.

2.2 In-Lab Assignment

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

2.2.1 4 Bit Adder

1. Make sure, that the motor is turned off.

2. Open the model you created to add two 4-bit numbers in the Pre-Lab.

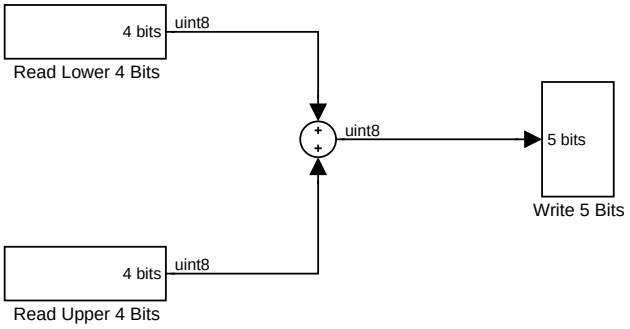
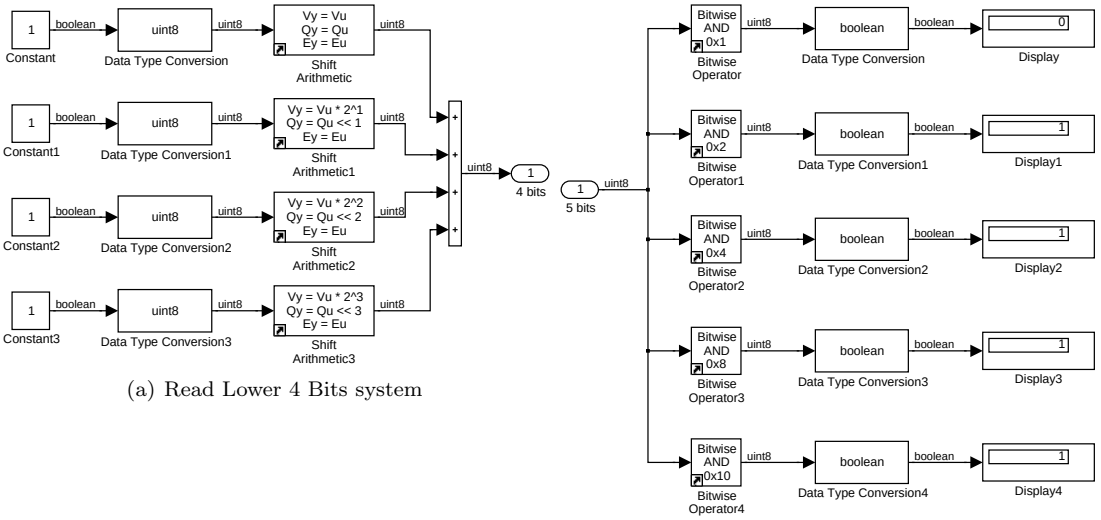


Figure 1: Top-level system



(a) Read Lower 4 Bits system

(b) Write 5 Bits system

Figure 2: Block diagrams of a 4-bit adder.

3. In order to tell Simulink Coder for which platform we want to generate code click on Simulation → Configuration Parameters (ctrl+E). In the following configuration go to the Code Generation setup, choose rappid.tlc as System target file. Go to the solver settings and make sure that you are using the discrete solver with fixed-step size.

4. In order to prepare the model, that it can be run on the MPC5553, place a block titled `RAppID-EC` from the Simulink library `RAppID` into the top level (not a subsystem) of your model.

   NOTE: Make sure that your working directory within MATLAB is set to a folder on your local C: drive.

   Open the configuration menu by clicking on the `RAppID-EC` block and configure it as follows:

   Set the following options in the menu bar:

   (a) Configuration → Compiler Config: MetroWerks

   (b) Configuration → Target Type: Internal RAM

   (c) Configuration → Real-Time Operating System: Simple Scheduler

   No further configuration is needed because no special peripheral functions will be used in this lab.

5. 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 and remove the automatically created input and output blocks. 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'.

6. 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.

7. 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 from the Simulink library `RAppID-Toolbox`. In order for that, perform the following steps:

   (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.

   (b) Open each GPI block, and set the pin number, see Table 1. 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, see Table 2.

   Since the current RappID-Toolbox installed provides support only for a limited number of GPIO pins we use the DIP switches and LEDs of the **Freescale evaluation board** itself for this exercise.

BIT	lower 1	lower 2	lower 3	lower 4	upper 1	upper 2	upper 3	upper 4
GPIO	93	94	95	96	97	98	99	100
PIN-Location	R26	R25	R24	T24	T23	T25	P23	U23

Table 1: Pin assignment for DIP switches

LED	1	2	3	4	5
GPIO	101	102	103	104	105
PIN-Location	U25	P25	M26	N23	N25

Table 2: Pin assignment for LEDs

8. 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.)

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

10. Build the model by typing **ctrl-b** or selecting 'Build Model' in 'Tools/Code Generation. After a successful build, Simulink Coder will echo:

```
### Successful completion of build procedure for model: Adder
```

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

12. 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).

13. 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. Note these values for the post lab.

14. **Show the working adder to the assistants.**

2.2.2 4 Bit Adder using the S-Function Builder

In this part we will create the same 4 Bit Adder as in the previous section. But this time we implement the bit shifting using a S-Function instead of Simulink blocks.

1. Shut off the motor.
2. Create a copy of your Simulink model from the previous section and open it.
3. Delete the contents of the Adder Task except for the Trigger block.
4. Create the model shown in Figure 3 in the Adder Task submodel. The `S Function Builder` block can be found in the Simulink library `Simulink / User-Defined Functions`. Save your model before continuing.
5. Open the S-Function Builder.
6. Set the S-Function name to *AdderFunction*.
7. Next we need to configure the inputs and outputs of the S-Function. Select the *Data Properties* tab and there the *Input Ports* tab. Set the *Rows* count to 8 for the input `u0`. This allows us to use the Simulink inputs in the C code as an array named `u0`. Switch to the *Output Ports* tab and set the *Rows* count to 5. In the tab *Data type attributes* set the *Data Type* for `u0` and `y0` to `boolean`.
8. Now we can enter the C code in the *Outputs* tab. The following code from Lab1 can be used:

```
unsigned char sumA;
unsigned char sumB;
unsigned char result;
int i;

/* convert the inputs to two unsigned char numbers*/
sumA =  (unsigned char)u0[0]
      | (unsigned char)u0[1] << 1
      | (unsigned char)u0[2] << 2
      | (unsigned char)u0[3] << 3;

sumB =  (unsigned char)u0[4]
      | (unsigned char)u0[5] << 1
      | (unsigned char)u0[6] << 2
      | (unsigned char)u0[7] << 3;

/* get the sum of the two numbers */
result = sumA + sumB;

/* display the two numbers*/
for(i = 0; i < 5; i++)
{
y0[i] = (result & (1<<i)) ? 1 : 0;
}
```

   Note that we need to cast the inputs to `unsigned char` before we apply the shift operations because our inputs are of `boolean` data type.

9. You are now ready to build your function. Click on the *Build Info* tab. Check the boxes named *Show compile steps*, *Create a debuggable MEX-file* and *Generate wrapper TLC*. Now click *Build* and close the dialog.
10. 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.)

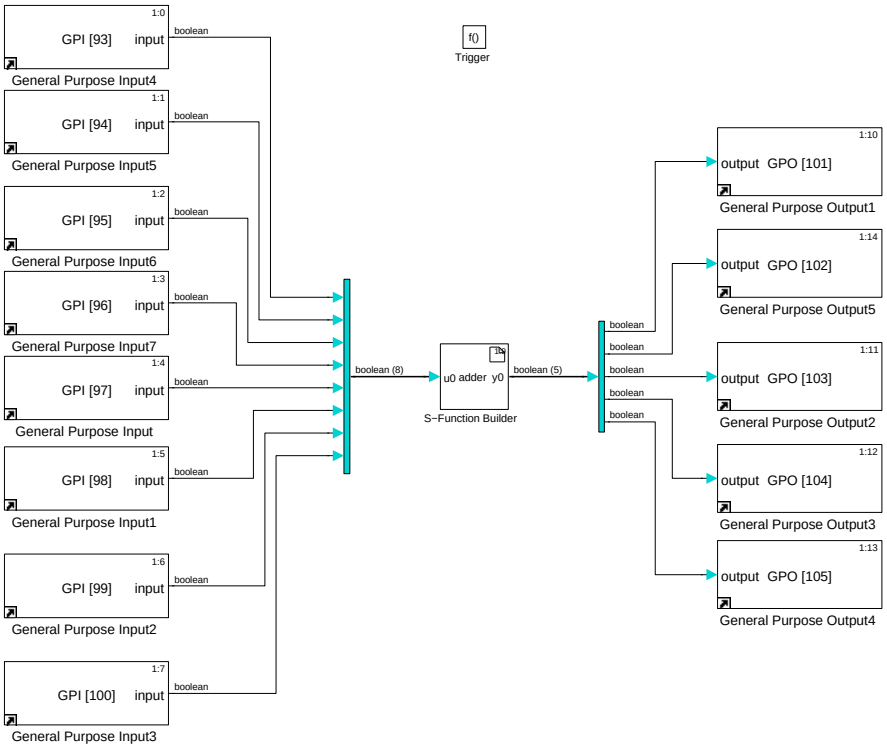


Figure 3: 4 Bit Adder with C-Code

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

12. Build the model by typing **ctrl-b**. After a successful build, Simulink Coder will echo:

```
### Successful completion of build procedure for model: Adder
```

13. The ELF binary file is located in <`model_name`>`_rappid_rtw/`. Load and test your program. Note that we use the DIP switches and LEDs of the **Freescale evaluation board** itself for this exercise.

2.3 Post-Lab Assignment

1. Explain the benefits and downsides of automatic code generation from a Simulink model.
   (a) Write down the ELF sizes resulting in the two automatically generated cases (in-lab 2.2.1 and in-lab 2.2.2) and in the handwritten case from lab 1.
   (b) Why is the automatically generated code so much larger in this case? Will this always be true?
   (c) Name two important benefits of automatic code generation.

2. What is the name of the function in the automatically generated code of in-lab 2.2.1, where the 4 bit adder calculations are carried out?

3. How is this function called in the main function? (Hint: Why is the for loop in the main function empty?)

4. What frequency does the system clock have?

3 The Virtual Wall

3.1 Pre-Lab Assignment

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. Create a m-file `VirtualWallParam` which sets variables that appear in the workspace.

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

   (a) Create a new, blank Simulink model, name it 'VirtualWall' and open it.

   (b) In order to make the m-file with the parameter run automatically, 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.

   (c) Create a subsystem, and call it 'Simulate Virtual Wall' and open it. In this block we will translate the position of the virtual wheel in degrees into a torque in Nmm.

   (d) 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.

   (e) Name the input port 'Wheel Position (deg)'. Name the output port 'Motor Torque (Nmm)'.

   (f) 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 degree.

   If you are not familiar with Stateflow you find an introductory in the file "Stateflow Modelling" on the course web site.

   (g) Connect the output of your Stateflow chart to the input of a `Saturation` block. Set the upper limit in the `Saturation` block to 835 and the lower limit to -835. Connect the input of the `Saturation` block to the output of the Simulate Virtual Wall subsystem.

   (h) Change back to the top level of the model. 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 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.

   (i) 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. Simulate the system for 5 s. You can change the simulation time in the 'Configuration Parameters' dialog, found under the 'Simulation' menu of your model. Attach the output of the `Step` block to the same `Scope` block as the output of the Simulate Virtual Wall subsystem for comparison. You can do this by changing the number of axes in the scope parameter settings. Inspect the resulting graph. 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.

HAND IN: Answers to the above questions.

3. Figure 4 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 eTPU/FQD device-driver block to produce wheel position in degrees.

   HAND IN: A printout of Figure 4 with the functionality of each block correctly identified.

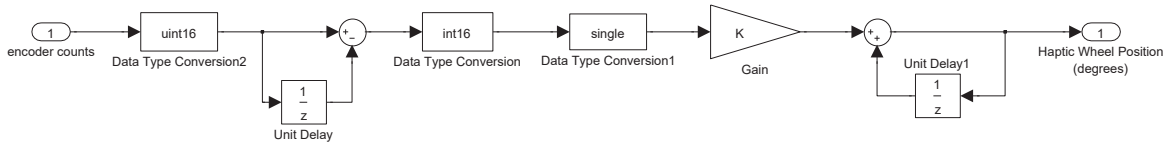


Figure 4: Block diagram for converting encoder counts to haptic wheel degrees

3.2 In-Lab Assignment

1. Open your Virtual Wall model and navigate to the top level of the model.
2. Enable the haptics motor by setting `GPO[205]=0`. Therefore, you place a `GPO` block on the top level and feed it with a constant block. The General Purpose Output block can be found in `RAppID-Toolbox/Peripheral Driver Blocks/GPIO Blocks`.
3. Place a block titled `RAppID-EC` from the Simulink library into the top level (not a subsystem) of your model. Open the configuration menu by clicking on the block and configure it as follows:

   Set the following options in the menu bar:

   (a) Configuration → Compiler Config: MetroWerks
   (b) Configuration → Target Type: Internal RAM
   (c) Configuration → Real-Time Operating System: Simple Scheduler

   The module specific configuration menu can be opened by clicking on the according image part. Configure the following modules with the specified settings:

   (a) `CPU: System Clock`
   Check that the system clock is set to 128 MHz, which should be the default setting.
   (b) `eTPU: Quadrature Decoding`
   To setup the Quadrature Decoding code on the eTPU do the following steps:
      i. In the tab `Time Base` set `TCR1 Clk Source` to `(System Clock)/2` and `TCR1 prescalar` to `1`.
      ii. Switch to the tab `ETPU A Channel Setup` and click on the `Register Function` button to add the Quadrature Decoding code.
      iii. Click `Add` to create a new function and fill in the following settings:

```
Function Name: Quadrature Decoding
Function Number: FS_ETPU_QD_FUNCTION_NUMBER
Number of Ch Params: FS_ETPU_QD_NUM_PARMS
Entry Table Macro: ETPU_ENTRY_TABLE_A
Entry Table Encoding: ENTRY_TABLE_ENC
Function API Header: etpu_qd_auto.h
Function Prototype Hdr: etpu_qd.h
```

Click on `Apply` and `Ok` to register the function.

iv. Rightclick on the default item in the `Translation Channel Config` list and select `Insert TPU Function: Quadrature Decoding`. Check and set the following values in the `Translation Channel Config Entry` menu below:

```
Func. Frame Address Dynamic Allocation: AUTO_FUNC_FRAME
Channel Priority (CPR): medium
Channel Type: input
Channel Number: 3
Channel Name: QD1
CIE, DTRE ODIS,OPOL: disabled
Function Mode: 0
```

v. Switch to the tab `Software Attributes` and set the field `ETPU Engine Code header file (*.h):` to `etpu_set3.h`.

vi. Close the eTPU Configuration menu by clicking `Ok`.

(c) Save the RAppID-EC block and close it.

4. Place a block titled `EMIOS Output PWM` from the Simulink library into the top level (not a subsystem) of your model. Double click on the block and set the initial duty cycle to 50%, the initial frequency to 1 MHz and the channel to 12. Close the dialogbox afterwards. This block takes two inputs: a duty cycle as an integer value between 0 and 100 which represents a duty cycle between 0-100% and a frequency value in Hz. For the frequency input, add a `Constant` block with value 1e6, for the duty cycle 50 and input these values to the block. You will need to cast both inputs to type 'uint32' before connecting them to the `EMIOS Output PWM` block. This PWM block sets the 1 MHz external clock for the switched capacitor (sc) filter.

5. Create a new subsystem and name it 'VirtualWallTask'. Open the subsystem and remove the automatic added input and output block. Place the subsystem 'Simulate Virtual Wall' without the step and scope block in it and remove the constant and scope block. As before, add a `Trigger` block and set its 'Trigger type' parameter to 'function-call'. Outside of the 'VirtualWallTask' subsystem, add a `Function-Call Generator` block, set its sample time to 0.001 s and connect it to the subsystem.

6. In the 'VirtualWallTask' subsystem, add a subsystem to represent the conversion between encoder ticks and haptic degrees. Into this subsystem, add blocks to build a system as shown in Figure 4. Watch carefully the remarks to Port Data Types and sample times for certain blocks given in Figure 4.

7. In the 'VirtualWallTask' subsystem, add the `Quadrature Decoder` block, found in `RAppID-Toolbox / Peripheral Driver Blocks / eTPU Blocks`. Open the block parameters, and choose eTPU 0, Channel 3, Max Speed = 1000, set the encoder resolution properly, and choose a Position Count Scaling of 4.

8. 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`.

9. In the 'VirtualWallTask' subsystem, 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 Nmm) to a duty cycle ranging from 0 to 1. Recall from Lab 4 and 6 that the equation relating torque at the wheel to duty cycle is:

```
Duty Cycle = 0.5 + 3.11e-4 * torque;
```

Once the duty cycle is calculated, use a saturation block to limit its value to the range 24% to 76%.

10. Finally, we need to output the desired duty cycle to the motor. Within the 'VirtualWallTask' subsystem, add an `eMIOS Output PWM` block. Choose Channel 0, an initial Duty Cycle of 50% and an initial Frequency of 40 kHz. This block takes two inputs: an integer value between 0 and 100 representing a duty cycle between 0 and 100% and a frequency value in Hz. For the frequency input, add a `Constant` block with value 40000. For the duty cycle input, scale the duty cycle in the appropriate range, and input this to the block. You will need to cast both inputs to type 'uint32' before connecting them to the `PWM Output` block.

11. Configure your model to generate a traceable model report as described in the handout "Analyzing Generated Code".

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

13. Using the traceable model report you generated, find the part of the generated `C` code that implements the Stateflow chart used to model the virtual wall, and copy these lines for use in the Post-Lab.

14. Load and test your program. **Proceed as follows**:

    (a) Switch on the boards, the motor not connected to it.

    (b) Load and run your program.

    (c) Check with the oscilloscope the digital output of eMIOS 0, the analogue output of AN 0 and the trigger input of the switched capacitor filter eMIOS 12. i) What signal, frequency and voltage at every signal do you expect? When you are happy with the measurements of the outputs connect the encoder cable to the encoder socket Enc_2 on the interface board. Turn the hand wheel without having the haptic box powered. ii) What signal changes at the oscilloscope, what is the value and why?

    (d) Reset the debugger, switch on the haptic box, connect the motor/amplifier wire to the interface board, **hold the wheel tight** and run the program. Turn the wheel and experience the virtual wall. **Show the assistants the running system.**

3.3 Post-lab Assignment

1. Hand in the lines of generated `C` code that implement the Stateflow chart used to model the virtual wall; you found these using traceability during the In-Lab.

HAND IN: The answers to the questions in section 3.2 14c, and the printed C code from the post-lab assignment.

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

1. The files *two_virtual_wheels.m*, *two_virtual_wheel_analog.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. Create a copy of the file *two_virtual_wheel_discrete.mdl* and name it *two_virtual_wheel_subsystems.mdl*. Remove all `Input`, `Output` and `To Workspace` blocks from the diagram but do not remove the `scope` and `step` blocks. The subsystem containing the fast dynamics will contain the input (at the moment a `Step` block) and output block (at the moment the `scope` block), because we want the system to respond at the fast rate.
   Separate the fast (10 Hz) and slow (1 Hz) dynamics into two subsystems. In order for that, select the components corresponding to a system and select 'create subsystem' from the context menu. Make sure, that the `Rate Transition` blocks stay at the top level of the model outside of the subsystems. Name the subsystems "FastDynamics" and "SlowDynamics" and name their inputs and outputs according to the signal.

3. 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. Recall that the sample times in the `step` and `discrete integrator` blocks should be set to "-1" so that the sample time is determined by the trigger blocks.

4. 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.

5. Inside the fast task subsystem, make sure, that at the `step` block the step time of `step_time` and the final value of `Theta_z_0` is set. We will use this to simulate a sudden turn of the haptic wheel. The reaction torque of the slow and fast system will be displayed at the `scope` block inside the fast subsystem.

6. Simulate the system.

7. Compare the plots generated by *two_virtual_wheel_analog.mdl* and *two_virtual_wheel_subsystems.mdl* by comparing the results on the two `Scope` blocks in the respective Simulink models. The discrete time simulation should match the continuous time simulation quite well. However, by zooming in on the plots, you should eventually see differences between the simulations. **Record these differences for use in the Post-Lab**.

HAND IN: A brief description of how the analog and discrete simulations differ.

4.2 In-Lab Assignment

1. Open model *two_virtual_wheel_subsystems.mdl*, and rename it to *two_virtual_wheel_autocode.mdl*. Copy the `RAppID-EC`, `GPO` and `EMIOS Output PWM` blocks from the top level of your *VirtualWall.mdl* model to the top level of *two_virtual_wheel_subsystems.mdl*.

2. Inside the fast 10 Hz 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. Configure your model to generate a traceable model report. In order for that, open 'Tools/Code Generation/Options...' menu and add 'Traceability' by clicking the 'Set objectives...' button. Select 'Code Generation/Report' and activate all checkboxes. Close afterwards the options window.

5. Save, update, and build the model.

6. Using the traceable model report, find the lines of generated `C` code that transfer data between tasks (the `Rate Transition` blocks). Copy these lines for subsequent analysis in the Post-Lab.

7. Run the generated ELF binary using the debugger. Now you should have the model working on the processor.

8. Set the two virtual wheels in motion by turning the haptic wheel and holding it steady. You should feel reaction torques generated by both the slow and fast virtual wheels. If you have trouble doing so, read the handout "A More Realistic Human Model for the Virtual Spring Inertia System" available on the course website for possible explanations. **Show the working system to your assistants.**

9. In order to compile your models faster, you can remove the previously added 'Traceability' objective and deactivate the report generation by deselecting the corresponding box. Save a copy of your system at the current state for later use.

10. One of the advantages of autocode generation is that it enables rapid prototyping because it is relatively quick and easy to test out new ideas, and thus to iterate on a design. Hence, it is easy to try out the various suggestions in Section 4 of the handout "A More Realistic Human Model for the Virtual Spring Inertia System" mentioned above. Do this for the suggestion in Section 4.2, and remove the human from the feedback loop. You may find the `Switch` block found in the `Commonly Used Blocks` library useful. As an option, you may also wish to try out the suggestions in Sections 4.1 and 4.3. Record your results for future reference.

11. Reuse your previously saved system without the modifications of the last step to proceed.

12. We motivated the use of two tasks by the fact that we are implementing virtual wheels with very different frequencies. However, this software architecture may also be used for wheels with similar frequencies. Try two wheels, one with frequency $f_{n1} = 1$ Hz, and one with frequency $f_{n2} = 1.14$ Hz. Change the virtual mass value to achieve the given frequency. You may find it useful to simulate the system with the provided simulink models to find the correct value.
    What frequencies do you feel? Be sure to hold the wheel for sufficient time (30 seconds should suffice), and record the periods of oscillation that you observe. **Show your running code on the microprocessor to your assistants.**

4.3 Post-Lab Assignment

1. Refer to the lines of generated `C` code that transfer data between tasks (the `Rate Transition` blocks), and explain what these lines of code are doing.

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

2. Describe the response that you feel to 1 Hz and 10 Hz virtual wheels both with (Part 8) and without (Part 10) the human in the feedback loop. If you tried the other suggestions from the handout "Further Analysis of the Virtual Spring Inertia System" describe what happened with them also.

3. What period or periods of oscillation did you feel when you implemented both 1Hz and 1.14 Hz virtual wheels? Give a formula that will predict this oscillation period. (Hint: recall Problem 4 (c) of Problem Set 3.)