# Simulink Models for Autocode Generation*

J. S. Freudenberg

EECS 461, Fall 2008

## 1 Simulink Models

Suppose that you have developed a Simulink model of a virtual world, such as a wall or spring-mass system. We have seen how to choose the parameters of the virtual world so that it has desired properties. For example, we have seen how to choose the spring constant and inertia of the virtual spring-mass system so that it has a desired frequency of oscillation and satisfies a maximum torque limit. We also learned how to add damping to such a model to counteract the destabilizing effect of forward Euler integration. Once we develop a model of the virtual world that behaves correctly in simulation, it remains to implement this world in `C` code that can be executed on the MPC5553 microprocessor. Until now we have simply written the `C` code by hand, and have debugged any resulting errors as necessary. Such errors may arise from simple mistakes in implementing the force feedback algorithm, such as using incorrect parameter values or sign errors. They may arise in converting from physical units to units that the processor understands, such as duty cycle and encoder counts. Other errors arise from type conversions, such as those from signed to unsigned integers of different lengths. Furthermore, changes to the virtual world that are relatively easy to model in Simulink by adding additional blocks may require substantial work to code in `C`.

The potential difficulties with hand coding control algorithms have not proven too burdensome in our lab exercises. However, many real world applications are much more complex, and the time taken to hand code an algorithm, with all the necessary debugging, may take months. Hence, if we already have an algorithm that works well in simulation, it would be advantageous to be able to generate `C` code directly from the Simulink model. Even if this code is not used in production, it may be used for testing on hardware, thus enabling the rapid prototyping paradigm for embedded software design. In this approach, control algorithms are first tested on a model of the system to be controlled. If the algorithms work correctly on the model, then autocode generation is used to obtain `C` code that can be tested on the mechanical hardware, thus enabling an additional level of testing and debugging to take place. The idea is that the algorithms will be known to work before they are coded into `C`, and thus any errors that arise must be in the coding, not in the original algorithm specification.

Consider the Simulink diagram in Figure 1. As we have seen, with appropriate values of $k$, $J_w$, $b$, and $T$, we may successfully implement a virtual spring mass system that is a harmonic oscillator with specified period that satisfies the limit imposed on the reaction torque. The `C` code required to implement this system on the microprocessor must perform several tasks in addition to computing the reaction torque for a given wheel position, as shown in Figure 1. Wheel position must be obtained from the QD function of the eTPU. The duty cycle must be updated and sent to the PWM function of the eMIOS subsystem. Because wheel position comes from the eTPU in encoder counts, it must be converted into degrees. The reaction torque generated by the Simulink model is in N-mm, and must be translated into duty cycle. Variable type conversions must be performed. The eTPU and eMIOS peripherals on the MPC5553 must be initialized, just as we initialized them when hand coding in `C`.

The various initialization and unit conversion tasks are tedious and error prone. We shall see that the best way to deal with these is to write Simulink subsystems that perform these tasks correctly. It will take some effort to do so, but once we are done, we will have a library of these subsystems that can be reused so that

*Revised October 30, 2008.

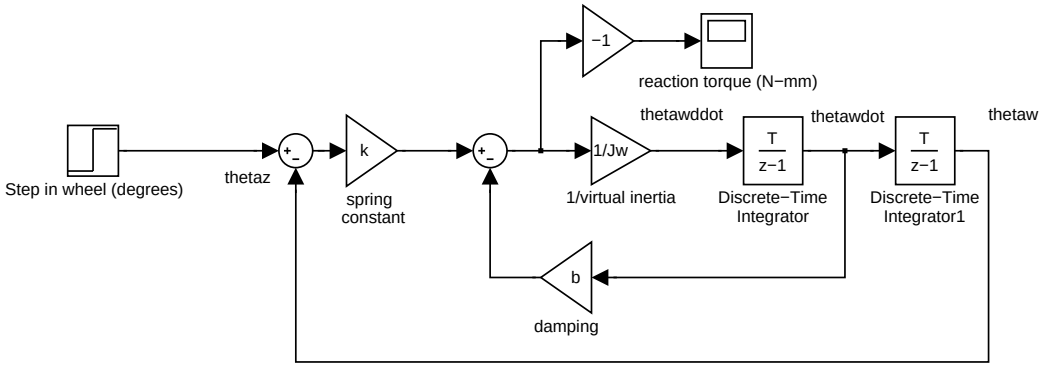


Figure 1: Discrete Simulation of Virtual Wheel and Torsional Spring with Damping (`virtual_wheel_discrete.mdl`).

we never have to do these low level operations again. This will free us to spend time designing virtual worlds using the Simulink model, and then automatically creating the `C` code that runs on the microprocessor.

To generate `C` code from a Simulink model, we shall need several additional software tools. These include Real Time Workshop [5], a Mathworks product that generate `C` code from a Simulink model and Embedded Coder [6], another Mathworks product that, when used in conjunction with Real Time Workshop, ensures that the generated code is compact and efficient. In addition, we shall need Simulink blocks that initialize the MPC5553 microprocessor and that supply device drivers for its peripherals, such as the eTPU and eMIOS. These latter blocks are available from the RAppID Toolbox [2], a product of Freescale Semiconductor Corporation.

# 2 Bit Manipulations

Recall our use of the "union" command in `C` to access various bit-fields in a register. One can also perform bit manipulations using Simulink blocks. This is sometimes necessary when developing a Simulink model to generate code that must interface with hardware (think of the dip switches and LEDS in the lab). For example, Figures 2-3 illustrate a subsystem that converts a single 32-bit unsigned integer into four 8-bit unsigned integers. The blocks used to build these figures are found in the Simulink Library Browser Menu:

- Simulink/Sources/Constant
- Simulink/Signal Attributes/Data Type Conversion
- Simulink/Ports & Subsystems/Subsystem
- Simulink/Sinks/Display
- Simulink/Logic and Bit Operations/Bitwise Operator
- Simulink/Logic and Bit Operations/Shift Arithmetic
- Simulink/Signal Routing/Mux

The same blocks may be used to build a subsystem that performs the reverse conversion, from four 8-bit unsigned integers to a single 32-bit unsigned integer. Such a subsystem is illustrated in Figures 4-5. A more elegant way to perform bit manipulations is through the use of Matlab S-functions to insert `C` code directly into a Simulink block. We shall learn about S-functions in a subsequent handout.

## Port Data Types

It is often convenient to have the data types of all signals displayed on the Simulink diagram. To do so, enable the option `Format/Port/Signal Displays/Port Data Types`. The results are illustrated in Figures 2-4.

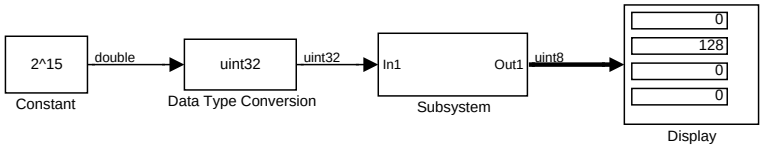


Figure 2: A subsystem to convert a 32 bit unsigned integer into four 8 bit unsigned integers (`thirtytwobit_foureightbits.mdl`).

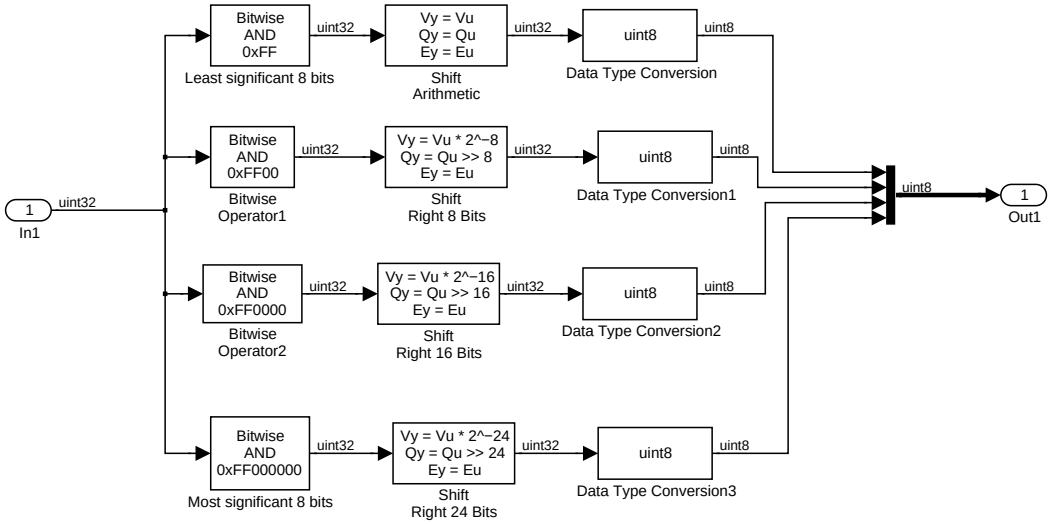


Figure 3: Inside the subsystem block that performs the conversion in Figure 2.

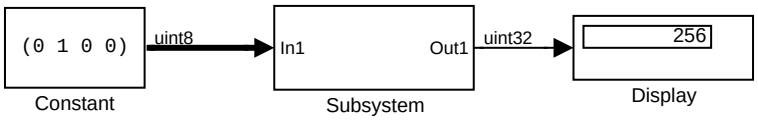


Figure 4: A subsystem to convert four 8 bit unsigned integers into a 32 bit unsigned integer (`foureightbits_thirtytwobit.mdl`).

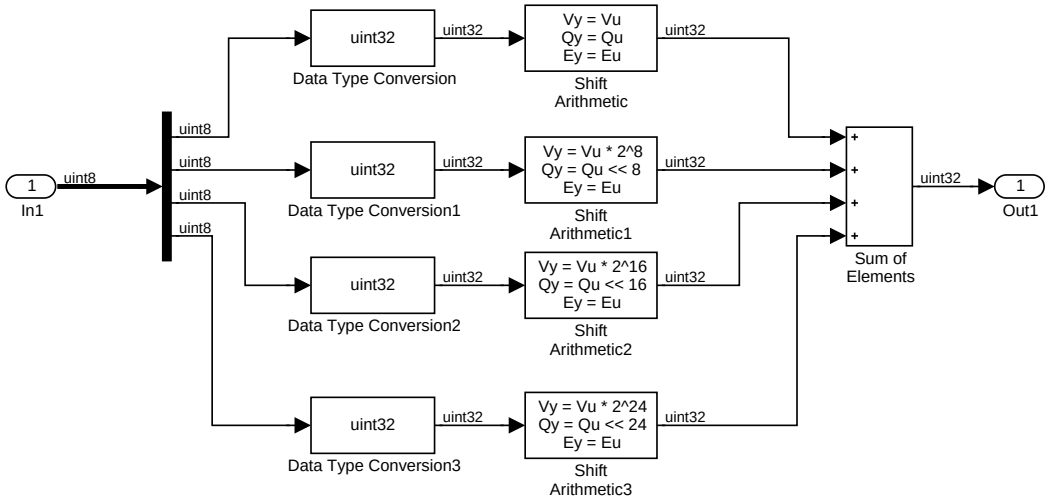


Figure 5: Inside the subsystem block that performs the conversion in Figure 4.

# 3 Device Driver Blocks

In order that `C` code generated from a Simulink model interface with the peripheral devices on the MPC5553, it is necessary to use device driver blocks that configure these peripherals. For example, in Figure 6 is a driver block for the QD function of the eTPU. This block can be configured to specify which channels of the eTPU are to be used for quadrature decoding. The outputs from the block include a 32-bit number that holds the current value of the 24-bit counter used by the eTPU to keep track of wheel position.[1]

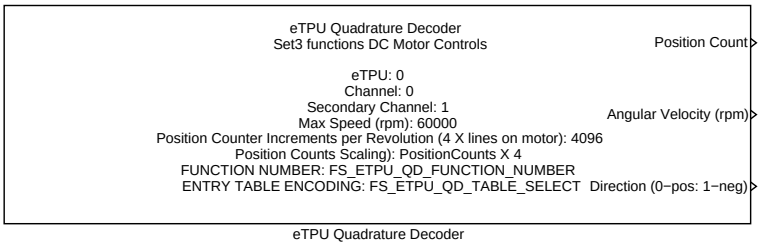


Figure 6: A device driver block for the QD function of the eTPU (`eTPU_QD.mdl`).

The driver block in Figure 6 may be used to develop a subsystem to convert encoder counts into wheel angle in degrees. Such a subsystem is shown in Figures 7-8. Similarly, a subsystem may be created that converts reaction torque from N-mm to PWM duty cycle (Figures 9-10).

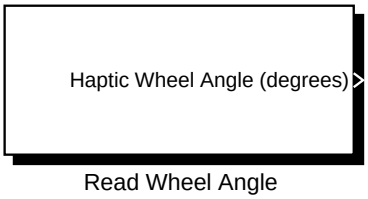


Figure 7: A subsystem to read wheel angle in encoder counts from the eTPU and output wheel angle in degrees (`read_wheel.mdl`).

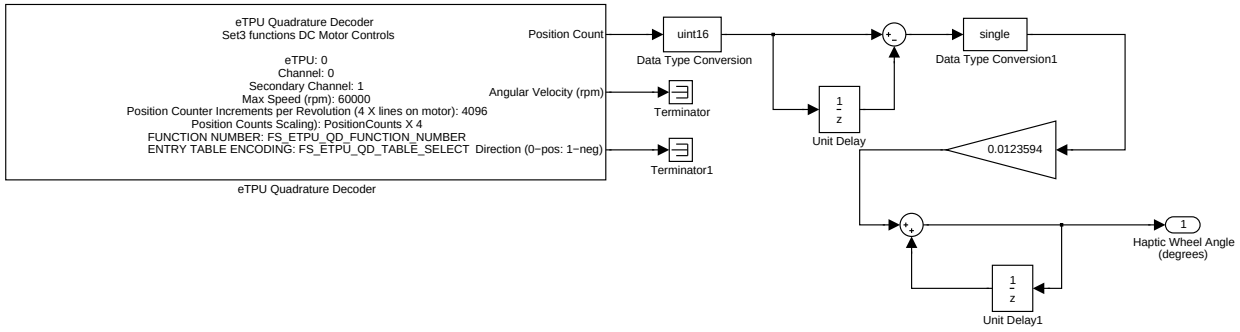


Figure 8: Converting wheel position in encoder counts to wheel position in degrees.

By replacing the step input and scope output in Figure 1 with subsystems that interface to the MPC5553 (see Figure 11) we begin to build a Simulink model that can be used for autocode generation of a virtual world.

[1] As in earlier labs, we only use 16 bits of this counter.

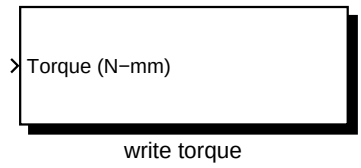


Figure 9: A subsystem to input reaction torque in N-mm and update the duty cycle of the MIOS PWM module (`write_torque.mdl`).

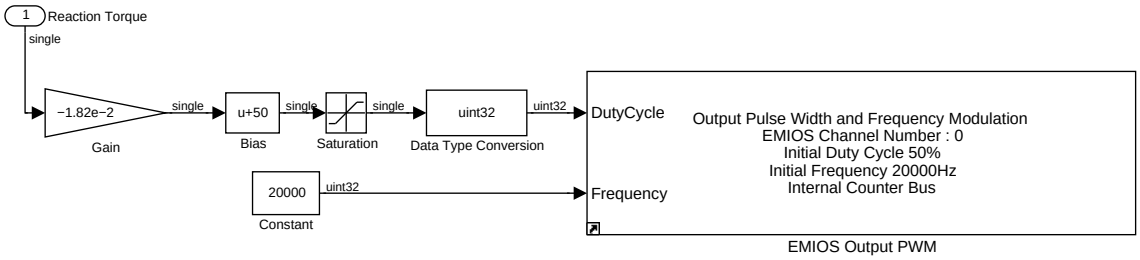


Figure 10: Convert torque in N-mm to torque in duty cycle.

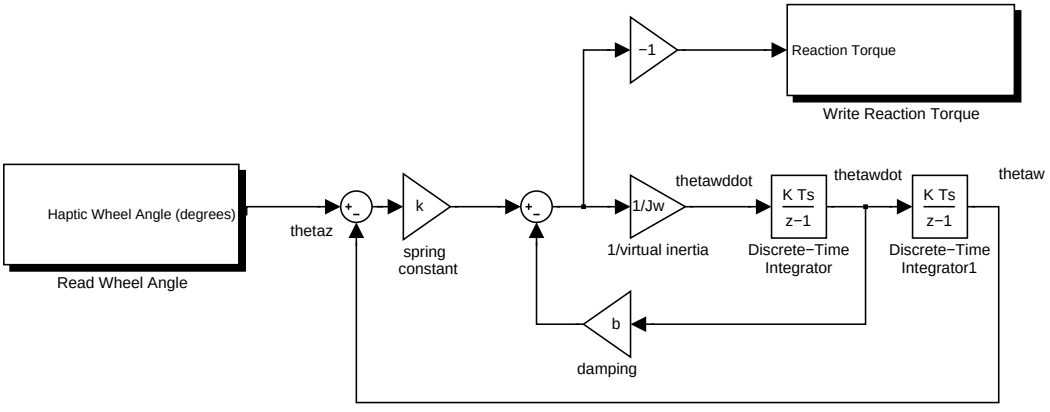


Figure 11: Simulink model from Figure 1 modified to interface with the MPC5553 (`virtual_wheel_drivers.mdl`)

# 4 Processor and Peripheral Initialization

Although the basic functionality of the virtual world and its interfacing are captured in Figure 11, several items remain before it is possible to use the model for autocode generation. We need to initialize the microprocessor we are using, as well as the peripherals such as the eTPU and the eMIOS. We also need to control the timing with which the virtual world is updated; we have done this previously by using the decrement counter to generate an interrupt at a specified rate. Finally, we may need to structure the embedded software into several tasks that execute at different rates, and to address the resulting shared data issues.

To accomplish the first item listed in the previous paragraph, we shall use an additional Simulink block, depicted in Figure 12. This block identifies the microprocessor target, the system clock speed, the `C` compiler used, and whether the generated code is in RAM or flash memory. It allows the user to specify whether a real time operating system (RTOS) is present, in which case we use OSEKturbo [1], an OSEK/VDX compliant RTOS available from Freescale. If an RTOS is not available, the "simpletarget" option is selected. Menus for initializing the peripherals are available by opening the block in Figure 12.

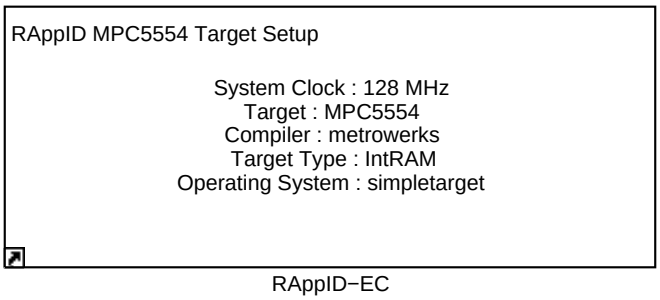


Figure 12: Initializaton block from the RAppID library (`RAppIDinit.mdl`).

# 5 A Virtual Spring Inertia Damper System

We can now build a complete virtual world, based on the spring damper system of Figure 1, that can be used to generate `C` code for the MPC5553. The highest level of the Simulink model is shown in Figure 13. The purpose of the processor initialization block in this model has already been explained. The model we wish to simulate must have device driver blocks added, as in Figure 11, and must be executed with a fixed time period. To accomplish the latter, we place the model from Figure 11 in a Triggered Subsystem block and add a Trigger block, as shown in Figure 14. The trigger block causes the simulation to be updated periodically based on the Function-Call Generator block in Figure 13, which itself executes every $T$ seconds.

There is one additional difference between Figures 14 and 11. This is the presence of the Environment Controller block, which allows the block diagram to be used either for code generation or for simulation, in which case the input is obtained from the Step input block in Figure 14.

## Initializing Parameter Values with Callbacks

It is possible to configure a Simulink model so that it initializes parameter values, such as sample time $T$ and the spring and inertia constants, every time it starts. To do so, assign these values in the window `File/Model Properties/Callbacks/InitFcn`. Alternately, assign these values using an m-file, and place the name of the m-file (without the .m suffix) inside this window. The latter option requires that the working Matlab directory be the directory containing the Simulink model and m-file.

## Sorted Execution Order

When Simulink is preparing to simulate a system that contains several blocks, it must order the execution of these blocks to account for functional dependencies between them and the times at which they need to be

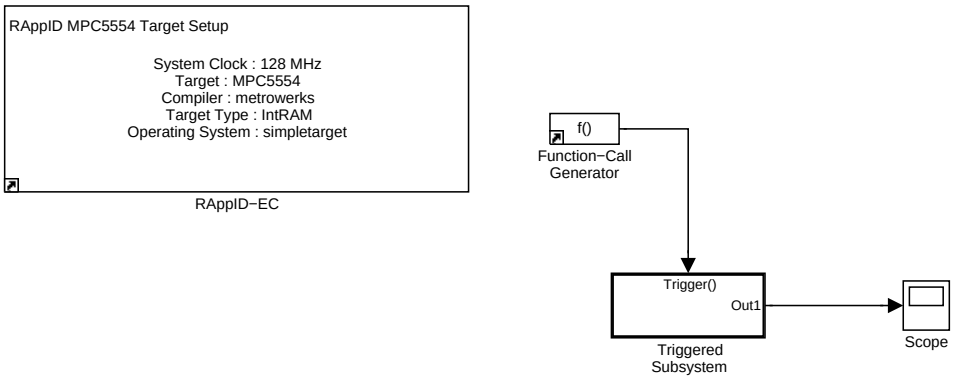


Figure 13: Highest Level of the Virtual Spring Mass Damper System (one_virtual_wheel_autocode.mdl)

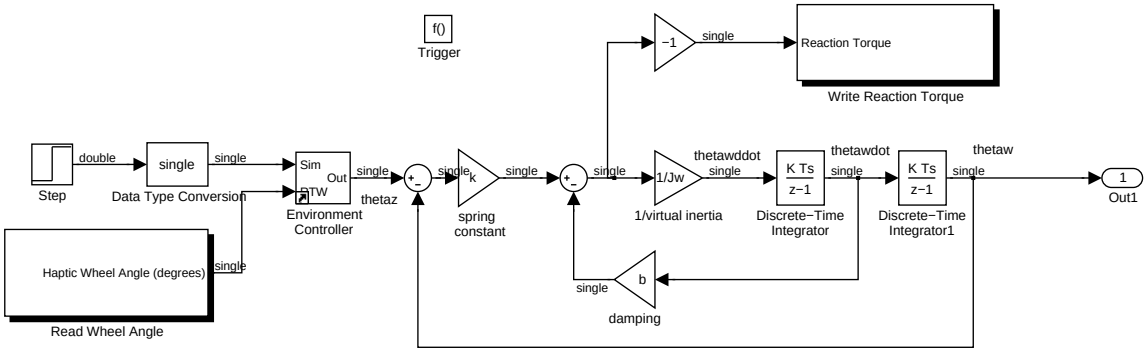


Figure 14: Inside the Triggered Subsystem block from Figure 13

updated. Similarly, the `C` code generated by Real-Time Workshop from a Simulink diagram must also take the flow of execution into account. To see the order in which Simulink will update each block in a diagram, enable the option `Format/Block Displays/Sorted Order`. The result of doing so for the simple diagram in Figure 1 is displayed in Figure 15. Note there are two numbers on each block: the first indicates subsystem number (in this case there is only one subsystem), the second refers to the order that the block is executed inside that subsystem.

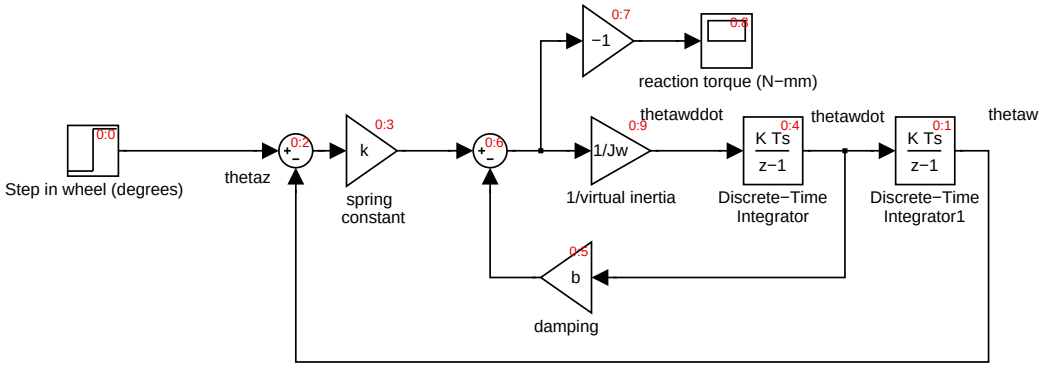


Figure 15: Discrete Simulation of Virtual Wheel with Sorted Blocks. (`virtual_wheel_discrete_sort.mdl`).

It is instructive to work around the diagram in Figure 15 to determine the reasons for the specified block sorting. Keep in mind that Simulink sorts blocks according to the following two principles, paraphrased from pp. 30-32 from Chapter 4 of the Simulink user's guide [4]:

- During each time step of the simulation, a given block must be evaluated before any other block whose output at that time step depends upon the output of the given block at the same time step.
- Blocks whose outputs at a given time step depend only upon past inputs and initial conditions can be evaluated in any order consistent with the previous principle.

For example, in Figure 15, the output of the step block and the rightmost discrete integrator block must be evaluated before the output of the leftmost summing block can be computed.

# 6 Simulation vs. Real Time Execution

There are important differences between general Simulink simulations that we have been doing throughout the semester, and embedded software that must execute in real time. These differences are important to consider when setting up a Simulink model for code generation.

A Simulink simulation does not need to evolve in real time. For example, it does not take precisely 10 seconds to simulate 10 seconds of the behavior of the system in Figure 1. The reason is that, between each two simulation time steps, all the computations required to update the model must be completed. If these computations are relatively simple, they will not take the entire time interval to complete, and the simulation can run faster than real time. On the other hand, if the computations are complex and time consuming, they may take longer than one time step to complete, in which case the simulation runs slower than real time. If all we care about is the correct result at the end of the simulation, then the difference between real time and time taken to perform the simulation computations does not matter.

Simulations that must interact with the physical world, such as the virtual worlds we implement on the MPC5553 which must interact with a human user through the haptic interface, must evolve precisely in real time. This means that if the computations required to update the simulation are completed relatively quickly, then the processor is idle for the rest of the time interval between simulation updates. On the other hand, lengthy computations may take longer than the simulation time step, thus preventing the simulation from executing in real time. To minimize the risk of this happening, it is sometimes necessary to break a simulation down into multiple subsystems that contain dynamics that evolve at different rates. A subsystem

with fast dynamics must be updated at a faster rate than a subsystem with much slower dynamics. As we shall see in Section 7, it is possible to perform such a multi-rate simulation in Simulink. There are some subtleties that arise when performing multirate simulations, and we shall also discuss these in Section 7.

When code is generated for a multirate simulation, Real-Time Workshop does either one of two things. If a real time operating system (RTOS) is available, such as OSEKturbo, then each subsystem is implemented as a separate task in the RTOS, with faster tasks given higher priority. If an RTOS is not available, then multi-tasking is simulated using nested interrupts in a procedure call "pseudo-multitasking". Although the generated code is different in each case, the simulations will yield equivalent results. Multitasking and pseudo-multitasking are discussed in Chapter 8 of the User Guide for Real-Time Workshop [3].

One issue that arises when a simulation is broken into multiple tasks is that of guarding the integrity of any data that must be shared between the tasks. This is done through the use of rate transition blocks, which we shall illustrate in Section 7.

# 7 Two Virtual Spring Inertia Damper Systems

The Simulink model developed in Section 5 is relatively straightforward and makes little use of the flexibility afforded by a real time operating system. Let us now consider a more complex virtual world that naturally suggests a multi-tasking software architecture. Specifically, we consider a virtual world consisting of dynamical subsystems with very different time constants. It is natural to simulate these subsystems with separate tasks that execute at different rates. Compare with the discussion of hardware-in-the-loop testing in [7].

Consider a virtual world consisting of *two* virtual wheels connected to the haptic wheel with virtual torsional springs and dampers. A continuous time Simulink model of such a system is shown in Figure 16. The parameter values for the two virtual inertias and springs have been chosen so that the frequency of oscillation of one subsystem is ten times that of the other. Suppose that we move the haptic wheel 45° and hold it in this position. Then we will experience the restoring torque plotted in Figure 17.

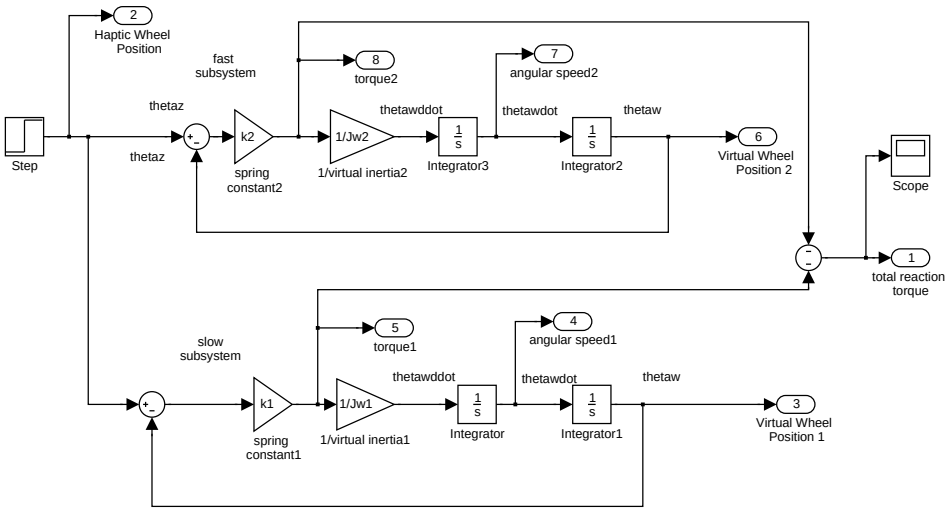


Figure 16: Continuous time model of two virtual wheels (`two_virtual_wheels_analog.mdl`)

Although it should not matter for this relatively simple model, for more complex models there is an advantage to separate the slower and faster dynamics before implementing this model on the microprocessor. The faster portion of the model must be implemented with a smaller time step for numerical integration than that used for the slower portion of the model. There is no need to numerically integrate the slower dynamics at the fast rate, and doing so has the disadvantages of increasing computation time needlessly and perhaps causing numerical roundoff errors to accumulate.

Motivated by the preceding discussion, we consider a discrete time simulation of this system using multiple sample rates: a slow sample rate is used for the slow dynamics and a faster rate for the fast dynamics. The

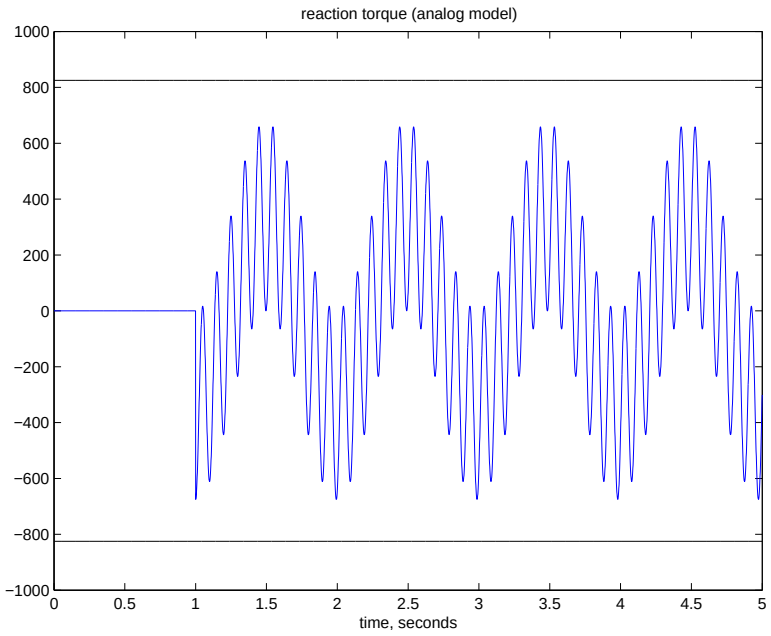


Figure 17: Restoring torque in response to a step change in haptic wheel position (`two_virtual_wheel_plots.m`)

model also includes damping to counteract the destabilizing effects of the forward Euler integration. The resulting model is shown in Figure 18.

## Displaying Sample Time Colors

It is often convenient to display subsystems executing at different rates in different colors. To do so, select the option `Format/Port/Signal Displays/Sample Time Colors`.

## Rate Transition Blocks

An important feature of the Simulink diagram in Figure 18 is the presence of rate transition blocks connecting the fast and slow subsystems of the simulation. The purpose of these blocks differs depending on whether they are being used in a Simulink simulation, which need not execute in real time, or for autocode generation, which does have real time constraints. When used for simulation, the blocks insure that different parts of the simulation are evaluated in the proper order (recall our discussion of sorted ordering at the close of Section 5). When used for autocode generation, the rate transition blocks are used to achieve integrity and determinism of data transfers between those parts of the generated code that must execute at different rates.

To explain further, consider what happens when a fast subsystem produces data that is used by a slow subsystem. If the computations required to update the slow subsystem cannot be completed before the fast subsystem must execute again, the output of the fast subsystem will have changed when the slow subsystem resumes execution, and these different values of data may result in an incorrect update to the slow subsystem. To prevent this from happening, the rate transition block acts like a sample and zero order hold operating at the slow period, thus insuring that all calculations required to update the slow subsystem are performed using the value of the fast subsystem output at the beginning of the slow subsystem update. (It is assumed that the update times for each subsystem are sufficiently long that all calculations can be completed required to update a given subsystem can be completed before the next update of that subsystem is required.)

Consider next what happens when a slow subsystem produces data that is used by a fast subsystem. If the slow subsystem takes longer to update than the time interval between fast subsystem updates, then the fast subsystem may read data in the process of being changed, leading to incorrect results. Moreover, even

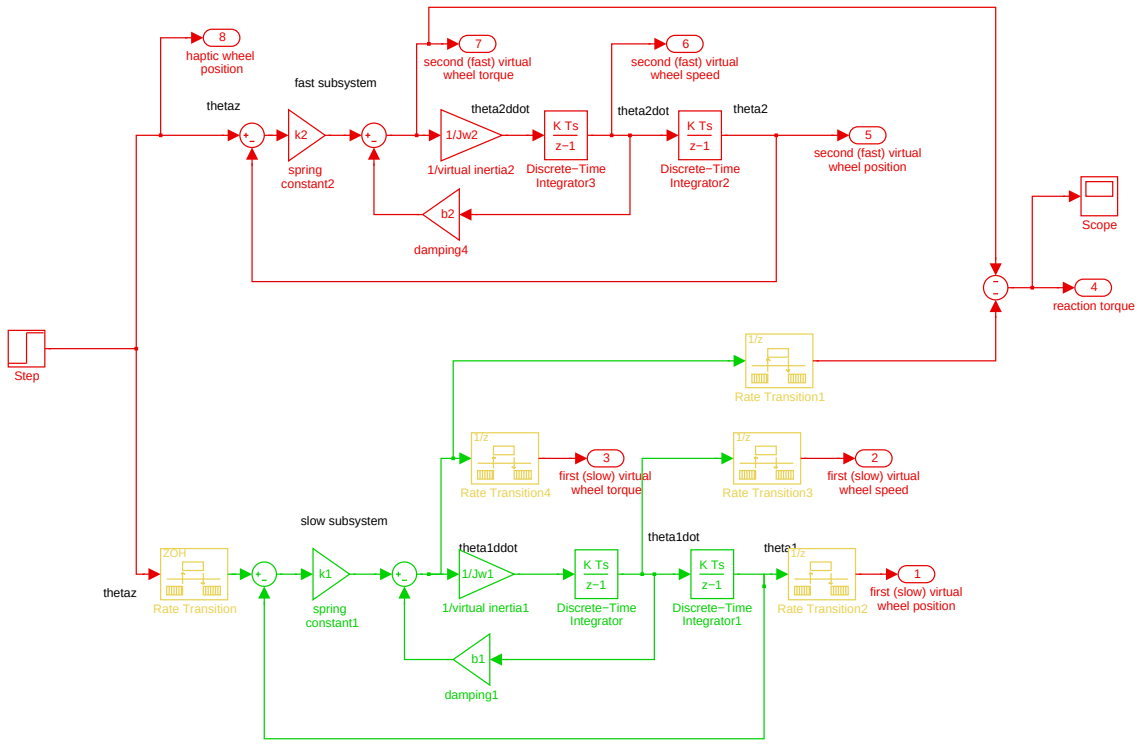


Figure 18: Discrete time model of two virtual wheels (`two_virtual_wheels_discrete.mdl`).

if the data is transferred correctly, the timing of the transfer may vary, with the result that it is not possible to know exactly when the fast subsystem will begin to use the new data from the slow subsystem. To resolve these issues, a rate transition block will effectively act like a delay equal to one slow update period. Hence the fast subsystem will always work with a value of the slow subsystem that is delayed by one slow update period.

The latency introduced in slow to fast data transfers described in the preceding paragraph is the price paid to insure deterministic transfer timing. It is possible to configure a rate transition block so that date protection and/or deterministic data transfer are turned off. In these cases, the generated `C` code will require less memory and execute more quickly, with the downside that unpredictable results may occur. See Chapter 6 of the Real Time Workshop documentation [3] for more information.

# 8 Code Generation for the Two Inertia System

It is necessary to modify the Simulink model in Figure 18 if we are to use it for autocode generation. The reason is that different parts of the model are simulated at different rates, and the `C` code generated from this model must execute at corresponding rates. There are two approaches to autocode generation with different execution rates, depending on whether or not an RTOS is present. We now illustrate both these approaches.

Consider the Simulink diagram in Figure 19. In this diagram, each of the virtual wheel subsystems is implemented in a "Triggered Subsystem" block that responds to a periodic function call generator, thus providing the block with a sample period. Inside each triggered subsystem block is one of the virtual wheel subsystems: the "slow" block contains the subsystem shown in Figure 21, and the "fast" block contains the subsystem shown in Figure 20.

The Simulink diagram in Figure 19 is used only for simulation, and is functionally equivalent to that in Figure 18. In Sections 8.1 and 8.2 we show how to modify these diagrams for the purpose of code generation, both without an RTOS and with an RTOS, respectively.

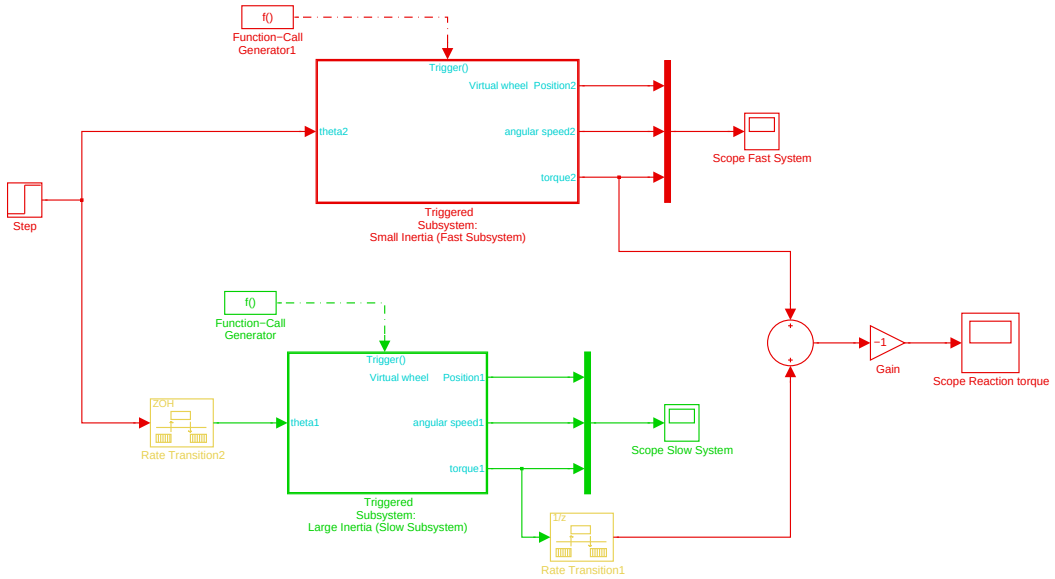


Figure 19: Two virtual wheels implemented as triggered subsystems (`two_virtual_wheel_subsystems.mdl`).

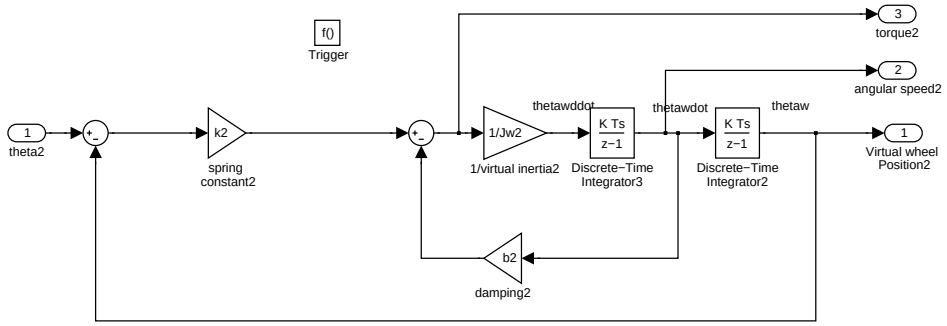


Figure 20: Fast triggered subsystem from Figure 19.

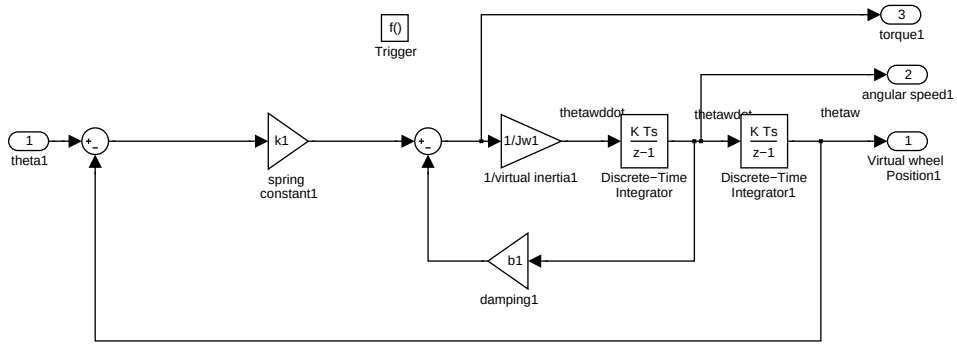


Figure 21: Slow triggered subsystem from Figure 19.

## 8.1 Without an RTOS

Now that the two virtual wheel system has been separated into separate subsystems, we must add initialization and device driver blocks. The resulting model is shown in Figures 22-24. The processor and peripheral initialization block has been placed at the highest level of the model. The device driver blocks are placed inside the fast subsystem, so that the encoder is read and the duty cycle is updated at the fast rate.

Note that the torque computed by the slow subsystem must be passed to the fast subsystem, so that the latter may compute the total reaction torque used to update the duty cycle. Similarly, the angle of the haptic wheel, which is obtained from the eTPU driver block in the fast subsystem, must be passed to the slow task so that the latter may use this information to compute the reaction torque for the virtual wheel with the larger inertia.

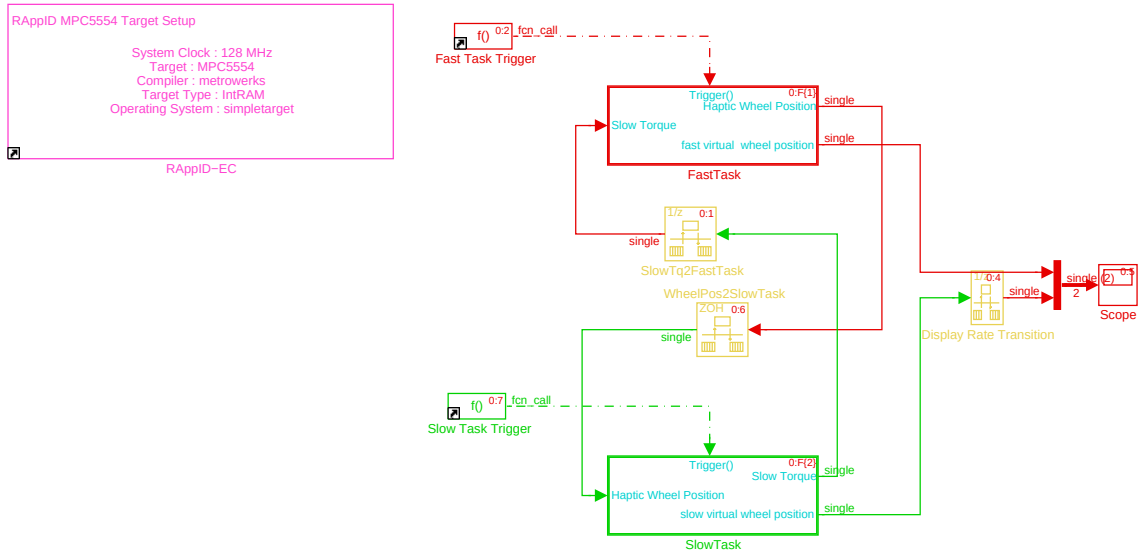


Figure 22: Highest Level of the Two Virtual Spring Inertia Damper System (`two_virtual_wheels.mdl`)

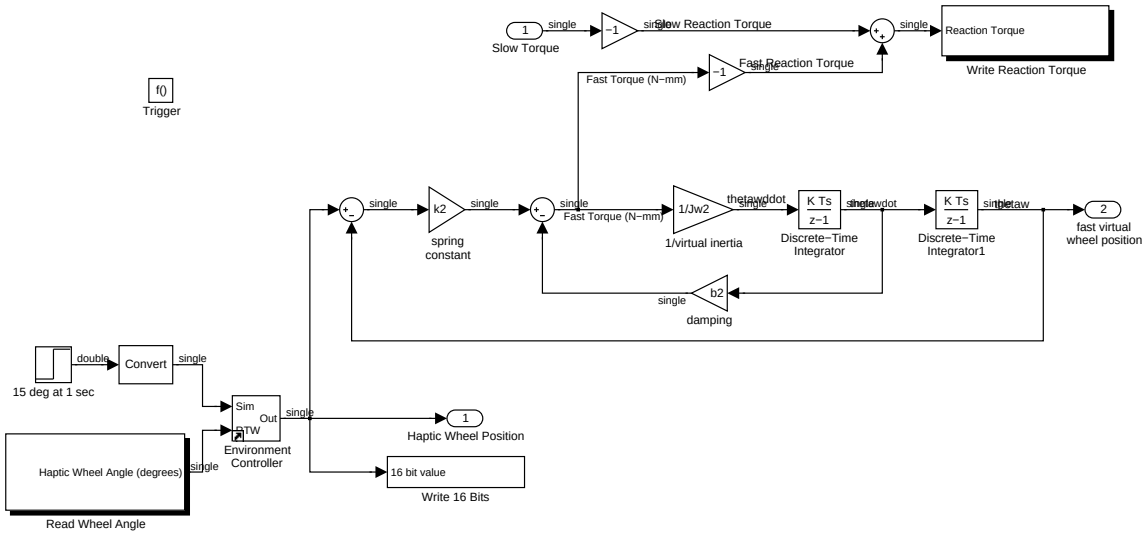


Figure 23: Fast triggered subsystem for autocode generation from Figure 22.

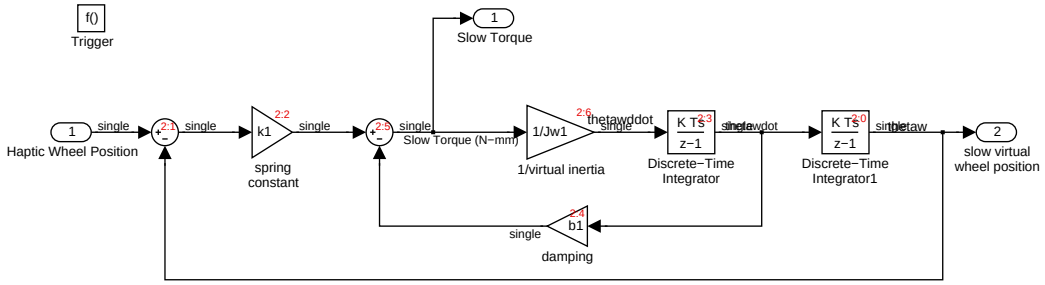


Figure 24: Slow triggered subsystem for autocode generation from Figure 22.

## 8.2 With an RTOS

If the OSEKturbo RTOS is present and selected on the initialization block, the generated code is structured as two tasks in the operating system, with the task corresponding to the fast subsystem having higher priority. The data shared between these tasks is protected by the priority ceiling protocol used in OSEK/VDX compliant operating systems. If there is no RTOS available, then the code for each subsystem is executed in interrupt routines, with the fast subsystem having higher priority, and data integrity is achieved by transferring the data at appropriate times.

Figure 25 shows the top level of the block diagram for code generation with an RTOS present. As in Figure 22, the fast and slow subsystems are separated into triggered subsystems. However, the flow if execution in the generated code is now controlled by the OSEKturbo task scheduler, with the faster subsystem receiving the higher priority. Data shared between tasks is passed back and forth through Resource Allocation blocks. These blocks implement the priority ceiling protocol discussed in class.

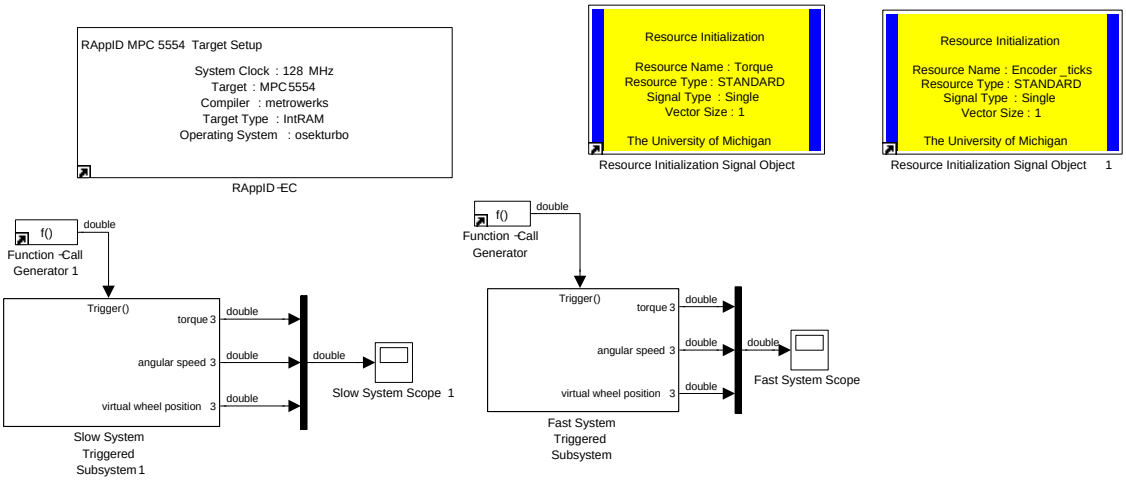


Figure 25: Highest Level of the Two Virtual Spring Inertia Damper System for Code Generation with an RTOS

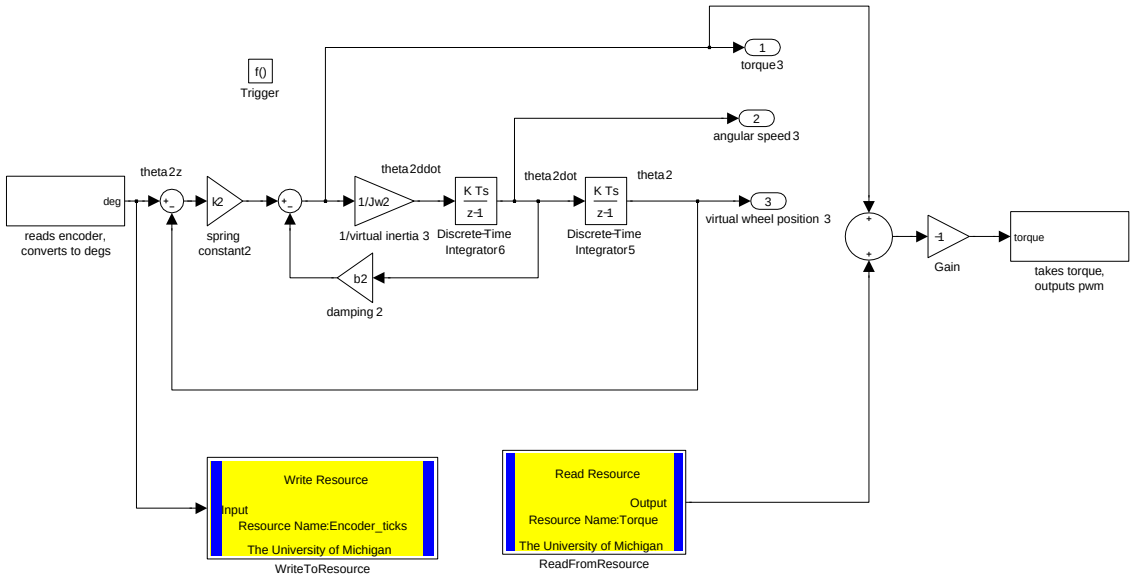


Figure 26: Fast triggered subsystem for autocode generation from Figure 25.

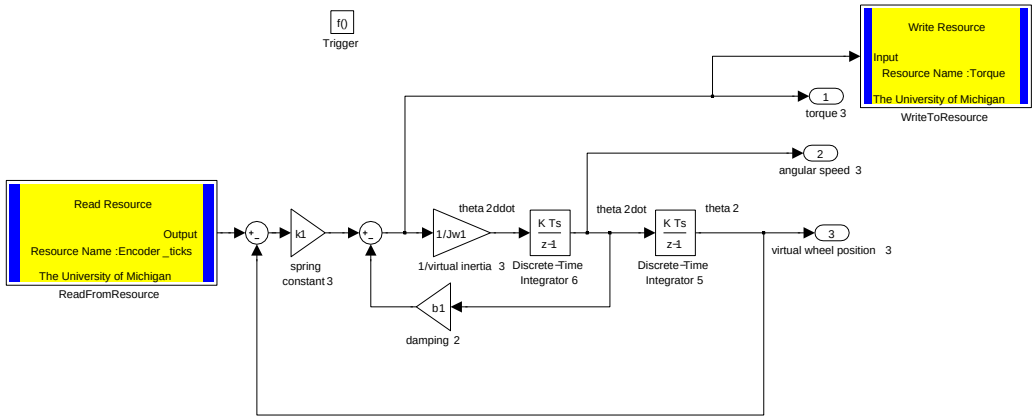


Figure 27: Slow triggered subsystem for autocode generation from Figure 25.

# References


[1] www.freescale.com/webapp/sps/site/overview.jsp?code=CW_OSEK.

[2] www.freescale.com/webapp/sps/site/prod_summary.jsp?code=RAPPIDTOOLBOX.

[3] www.mathworks.com/access/helpdesk/help/pdf_doc/rtw/rtw_ug.pdf.

[4] www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf.

[5] www.mathworks.com/products/rtw/.

[6] www.mathworks.com/products/rtwembedded/.

[7] J. A. Ledin. Hardware-in-the-loop simulation. *Embedded Systems Programming*, pages 42–60, February 1999.