Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

151-0593-00 **Embedded Control Systems** (Fall 2017) **Lab 4**

Topic: **Pulse Width Modulation (PWM) and Virtual Worlds without Time**

Pre-Lab due: 15. 09. 17 at 1 p.m. Post-Lab due: 15. 09. 17 at 5 p.m.

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

marianne.schmid@idsc.mavt.ethz.ch, 15. September 2017

1 Overview

The purpose of this lab is to use the Pulse Width Modulation (PWM) functions of the enhanced Modular Input/Output Subsystem (eMIOS). The PWM Submodule will be used to control the torque applied to the haptic wheel. In this lab, you will design and implement two simple haptic virtual worlds: the virtual spring and the virtual wall. In Figure 1 an overview of the PWM unit is depicted. Comparator 1 compares the counter with the threshold value C_{Th}. The PWM signal is high until C_{Th} is reached and then switches to low. When the counter value matches C_{max} the signal goes high again and the counter is reset to zero. Therefore, C_{Th} determines the pulse width and C_{max} the PWM frequency, see also Figure 2. Please note that this is a very high-level description of PWM generation, and that the actual operating details of the PWM modes available on the MPC5553 eMIOS are more complex.

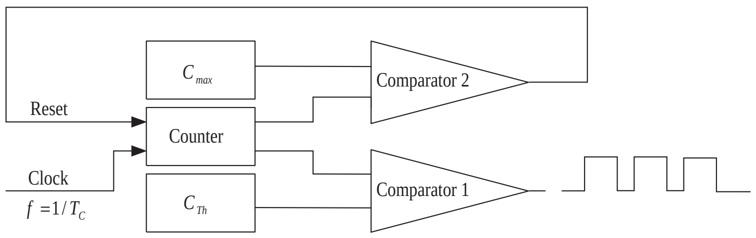


Figure 1: Overview of PWM Unit

1.1 Enhanced Modular Input/Output Subsystem (eMIOS)

The eMIOS module described in Chapter 17 of the MPC5553 Manual provides 24 channels with a versatile range of input and output functions. These functions range from simple digital I/O, to signal measurement abilities like pulse counting, edge accumulation, and versions of quadrature decoding. Many of these functionalities may also be accomplished with the eTPU module – a major difference between the eMIOS and the eTPU is that the eTPU is programmable, whereas the eMIOS functionality is all realized in hardware. In this lab we will be using the output PWM functionality of the eMIOS system to control the motor of our haptic wheel.

The eMIOS module supports several pulse-width modulation modes, including pulse width modulation (OPWM), pulse width and frequency modulation (OPWFM), center aligned output pulse width modulation (OPWMC), and a number of corresponding *buffered* modes, OPWMB, OPWFMB, and OPWMCB. The latter modes use *double buffering* to change the values of C_{th} and C_{max}, and the corresponding PWM function only obtains the new values after the completion of the current counting cycle. To see the problems that might ensue if the values of C_{th} or C_{max} are changed during a counting cycle, suppose that C_{th} is changed to a value that is smaller than the current counter value. Then the desired match between the counter and C_{th} cannot occur, and the PWM output will not change until the counter matches the new value of C_{th} in the next counting cycle. If both C_{th} and C_{max} are changed to values smaller than the current counter value, then the PWM signal will stay high until the counter rolls over and matches the new value of C_{th}. **As a consequence, the unbuffered modes should never be used.**

We shall use the buffered Output Pulse Width and Frequency Modulation (OPWFMB) mode of the eMIOS. In this mode, the values of C_{th} and C_{max} are determined by the data registers CADR and CBDR, respectively. Each of these registers has two subregisters used for double buffering, specifically, A1 and A2 for CADR, and B1 and B2 for CBDR. It is important to note that the value of C_{th} that is *read from* CADR, and used to set the PWM duty cycle, is that in subregister A1. A new value of C_{th} that is *written to* CADR changes subregister A2. Similarly, the value of C_{max} used to set the PWM period is that in subregister B1, and a new value of C_{max} written to CBDR changes subregister B2. The new values of C_{th} and C_{max} are transferred from A2 and B2 to A1 and B1 only after the eMIOS counter matches the old value of C_{max}, and thus only after a new counting cycle has started.
The internal counter used for PWM generation increments at a prescaled eMIOS clock rate. Its initial value is set to one, and the PWM channel output is initially set to the negation of the edge polarity (EDPOL) bit. When the counter value matches C_{th}, the PWM output is set to the EDPOL bit. When the counter matches C_{max}, its value is reinitialized to one, and the PWM output is again set to the negation of the EDPOL bit. As an example, if EDPOL is set to zero, the PWM signal will be high between counter values one and C_{th}, and low between counter values C_{th} and C_{max}.

Figure 2 illustrates the PWM signal and the corresponding counter for the OPWFMB mode. Note that the duty cycle D satisfies the relation $C_{th} = D \times C_{max}$. (Other eMIOS PWM generation modes, such as OPWFM, use different conventions, and thus have different expressions for the duty cycle.)

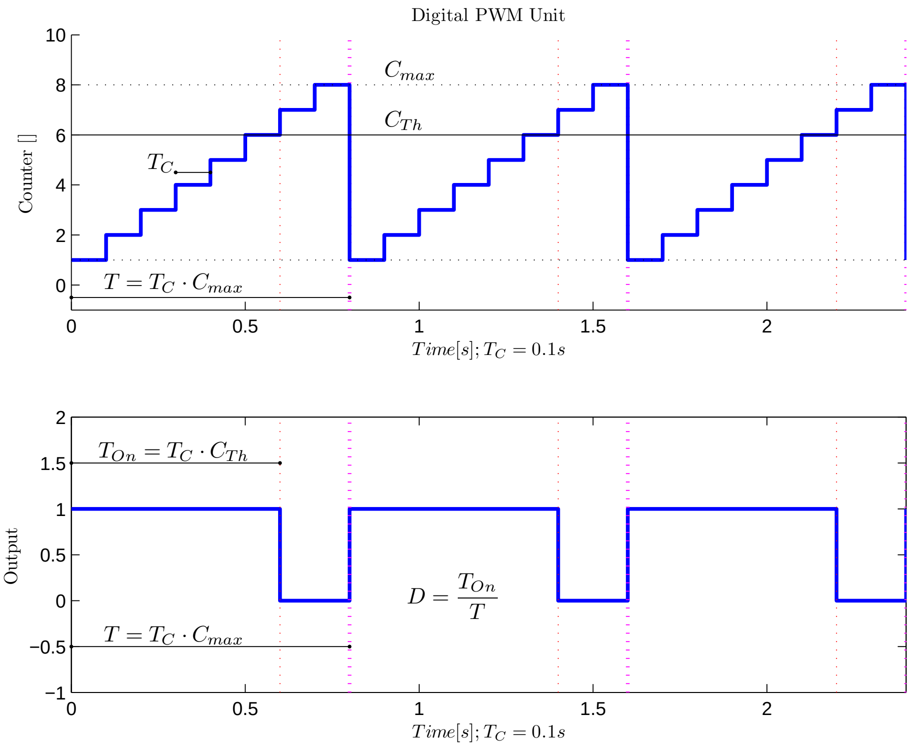


Figure 2: Output and Counter of a PWM Signal using the OPWFMB mode with $C_{th} = 6$, $C_{max} = 8$ and duty cycle $D = 6/8$

The registers CADR and CBDR do not store times but counter values. To calculate the on-time T_{on} and the period T of the PWM signal in seconds, we must know T_C, the period of the counter, in seconds. The eMIOS counter periode T_C depends on the CPU clock frequency and the prescalers. The two prescalers are (GPRE + 1) and (UCPRE + 1) where the values GPRE and UCPRE represent the configuration register value. See Figure 17-54 of the MPC5553 manual for further reference. The times can be calculated as follows:

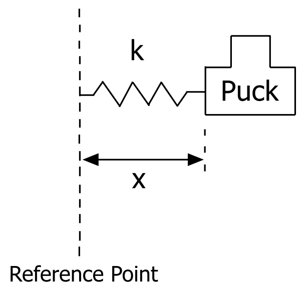
$$\begin{aligned} T_C &= \frac{(GPRE+1)\times(UCPRE+1)}{systemclockfrequency} \\ T_{on} &= T_C \times C_{Th} \\ T &= T_C \times C_{max}. \end{aligned}$$

Since the switching period and the on-time are both multiples of the eMIOS clock period T_C, there is a tradeoff between switching period and duty cycle resolution for a given T_C. A high frequency PWM signal will require a relatively small value of C_{max} to obtain a small switching period T. Since T_{on} is also a multiple of T_C, it follows that the number of possible values of the duty cycle is limited, and we may not be able to implement a small change in duty cycle. To illustrate, suppose that $C_{max} = 10$. Then there are only 10 possible nonzero values of the duty cycle, $D = 0.1, 0.2, \ldots 1$, and our resolution is 10%. If we are required to control small torque changes in the motor, then this resolution may not be sufficient, and we will need to use a longer switching period and thus a lower switching frequency.

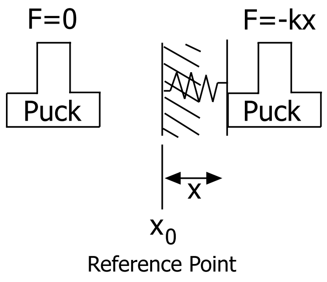
1.2 Virtual Spring

Figure 3a shows the puck attached to a reference point by a virtual spring with spring constant *k*. If the puck is moved to either side of the reference point, the spring will exert a restoring force F_s = -kx that will accelerate the puck back toward the zero position (because the spring is 'virtual', we assume that its relaxed length is equal to zero, and thus that the puck can move to either side of the reference point).

In our application, we use a motor and encoder to create a virtual torsional spring to rotate the wheel back to a nominal position.



(a) Virtual Spring



(b) Virtual Wall

Figure 3: Example of Virtual Worlds

1.3 Virtual Wall

There are two major differences between a virtual spring and virtual wall. On one side of a virtual wall ($x < x_o$), the disc should be freely spinning and the motor should apply no torque to the wheel. Once the wheel rotates into the other region ($x > x_o$), the motor should apply a torque to oppose further penetration into the wall. In order to allow the motor to spin freely on one side of the wall and apply torque on the other side of the wall, a simple if-then statement can be used. In order to apply a large torque opposing penetration into the wall, the spring constant, *k*, can be increased. Figure 3b illustrates the virtual wall for linear motion.

2 Design Specifications

2.1 Hardware

The haptic interface available in-lab contains a power supply, an amplifier, an encoder, a motor, and two ribbon cables for connecting to the interface board. The eMIOS channel 0 produces a digital signal, which is filtered to produce an analog signal. As the MPC5553 can only output a limited amount of energy the analog signal has to be amplified to drive the motor. The amplification of the complete chain is as follows.

The PWM signal switches between zero and 5 V. The filter unit that demodulates the digital signal to an analog one has a gain of 2.5 V/V. This module not only amplifies the signal, it also provides an offset adjustment. The amplifier gain is 1 A/V and the motor gain 257 Nmm/A, see Figure 4. The motor must be enabled to work, to do this the GPIO 205 pin is enabled by the `init_ECS()` function.

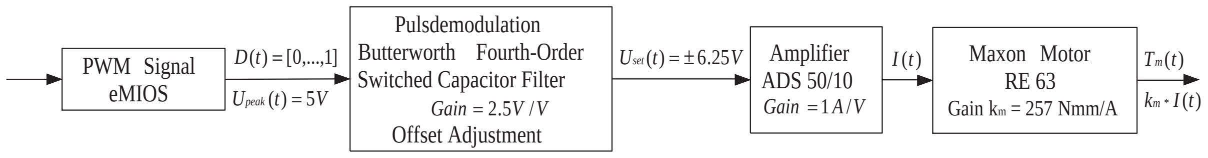


Figure 4: Total gain of the setup: $dM/dD = 5\ V * 2.5\ V/V * 1\ A/V * 257\ Nmm/A = 3212.5\ Nmm$

The interface board has connections to 13 eMIOS channels [0, ..., 12]. eMIOS outputs [0, ..., 3] are located at the top right hand side of the interface board, see Figure 5. The properly configured eMIOS output will provide a square wave output with a configured frequency and a variable pulse width controlled by the MPC5553. At the analog output pins (ANout) filtered PWM signals can be measured with an oscilloscope. The digital filter (demodulator) is a fourth-order butterworth switched capacitor filter that will be explained in more detail in Lab 5. The sc-filter clock of 1 MHz is provided by eMIOS12.

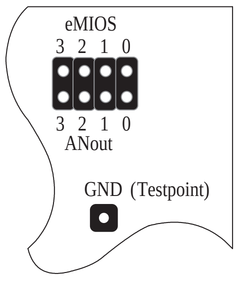


Figure 5: eMIOS and AN output (ANout) pins on the interface board

Using a parallel ribbon cable Channel 0 (ANout0) analog signal, GPIO 205 and ground (GND) are delivered to the amplifier at the haptic interface. The amplifier powers the motor of the haptic device. A quadrature encoder is also connected to the haptic interface, as you have learned in Lab 2. With these connections to the motor and encoder, virtual worlds can be constructed and controlled with the MPC5553.

The amplifier that drives the motor contains safety circuitry that will limit the power to the motor. Despite these safety functions, **be ready to switch off power to the haptic box at all times**, using the black main power switch on the rear of the box.

2.2 Software

mios.h and mios.c

The *mios.h* contains function prototypes for `init_MIOS_clock`, `init_PWM`, `set_PWMPeriod_ns`, `set_PWMDutyCycleLimits` and `set_PWMDutyCycle`. These functions will deal with configuring the counter prescalar for the eMIOS module, initializing an individual PWM channel, and setting its duty cycle and period. You will be responsible for writing the definitions for these functions in the *mios.c* file (rename *mios_template.c*).

You will need to use your FQD and PWM software to control the haptic interface. Note that the PWM output needs to be **limited to 24 to 76 percent duty cycle** to prevent damage to the haptic devices. These values are set in the function `set_PWMDutyCycleLimits`.

The *mios.h* file should be placed in the `include/` directory and the *mios.c* file should be placed in your `lib/` directory.

worlds.h and worlds.c

The *worlds.h* file contains the prototypes for the different virtual worlds you will implement in this course. You will be responsible for writing the definitions for these functions in the *worlds.c* file (rename *worlds_template.c*).

The *worlds.h* file should be placed in the `include/` directory and the *worlds.c* file should be placed in your `lib/` directory.

motor.h and motor.c

You need to make a C file called *motor.c* from the *motor_template.c* file that contains the function `outputTorque`. Place this file in your `lib/` directory. The prototypes and details of these functions can be found in the *motor.h* file. It is very important that you follow these prototypes and the comments in the *motor_template.c* file closely. Since you will now also be using the MIOS functions, be sure to add the necessary `#include` lines.

Extreme torque values will cause the motor to overheat and harm the system. **Limit the actual applied torque to a range from -835 Nmm to 835 Nmm** within *motor.c*.

lab4.c

Create the main program for your lab from the *lab4_template.c*. Use eMIOS0 for the motor and eMIOS12 to set the 1 MHz clock for the switched capacitor (sc) filter. Most of the initializations and the necessary `#include` lines are done for you.

Makefile

Use the provided makefile to compile your code.

3 Pre-Lab Assignment

Pre-lab questions must be done **individually** and handed in at the start of the in-lab section. You must also, **with your partners**, design an initial version of your software before the in-lab section.

In the pre-lab you need to complete the following C files: *mios.c*, *motor.c* and *worlds.c*. For each file a template is provided, which needs to be renamed and placed in your `lib/` directory. The ***mios.c*** file contains five functions, named:

- `init_MIOS_clock`
- `init_PWM`
- `set_PWMPeriod_ns`
- `set_PWMDutyCycleLimits`
- `set_PWMDutyCycle`

Each of the PWM functions will accept an integer, `miosChannel`, whose value ranges from 0 to 3 corresponding to the eMIOS channels available on the interface board. The prototypes of these functions can be found in the *mios.h* header file. In ***motor.c*** you need to calculate the duty cycle according to the torque and the total gain of the haptic device. The ***worlds.c*** file contains the code to calculate the resulting torque of the different virtual worlds.

1. The eMIOS module should be configured so that the highest frequency possible is maintained, meaning that one counter increment should be as fast as possible. The system clock frequency (f_{SYS}) is 120 MHz. Which registers would you use to set the two prescalers, what values would you use and what would the resulting T_c be?

2. Using the prescalers chosen in the previous question, what is the highest-frequency PWM output we can establish and still control duty cycle to a resolution of 0.5% ? What is the maximum attainable resolution of a 40 kHz PWM output?

3. The **init_MIOS_clock** function configures the eMIOS module. Complete this function where indicated.

4. The **init_PWM** function configures the input eMIOS channel for OPWFMB. Complete this function where indicated.

5. The **set_PWMPeriod_ns** function will accept a 24-bit number (passed as a 32-bit integer), `newPeriod`, indicating the new period in ns to be used for the PWM output. Complete this function.

6. The **set_PWMDutyCycleLimits** function will set the safety limits of the duty cycle. This function has already been completed for you.

7. The **set_PWMDutyCycle** function will accept a floating-point number, `dutyCycle`, corresponding to the new duty cycle of the OPWFMB. The value of `dutyCycle` will represent a fraction of the total 'on-time' of the output and range from zero to 1. Complete this function.

8. Set the torque limits and create the function **outputTorque** within *motor.c.* Convert between a virtual torque value and a PWM duty cycle. Refer to Figure 4 and the data sheet of the motor (inside red or black folder on your desk) to answer the questions below. We use the motor with the part number 353301.

   a. What is the equation for torque at the haptic wheel as a function of duty cycle?

   b. What is the conversion ratio between encoder ticks and the angle at the haptic wheel?

   c. What is the maximum speed of the hand wheel in rpm with a maximum of 24 VDC provided by the power supply?

9. Write a file *worlds.c* in which you implement the functions **virtualWall** and **virtualSpring** as specified in the *worlds.h* file. Place this file in the `lib/` directory.

   - For the *virtual spring*, choose the spring constant k_{spring} to yield a restoring torque of 50-Nmm per degree of wheel displacement from the reference position.
   - For the *virtual wall*, choose the spring constant k_{wall} to yield 400 Nmm torque per degree of wheel movement into the wall.

10. Assume that the haptic wheel/motor combination is a pure inertia with $J = 4.5 \times 10^{-4}$ kg-m^2/radian, and ignore the digital implementation. (That is, assume the wheel/motor combination has transfer function $G(s) = 1/Js^2$.) Using the value of k_{spring} specified in part 8. above, what frequency of oscillations f_{spring} do you expect to see? Express your answer in Hz. **Be careful with units when you work on this problem!**

11. Because the of limit imposed on the maximum amount of torque that may be applied to the motor, turning the wheel too far before releasing it will result in initial torque saturation. Because of the damping inherent in the motor, the oscillations will decay with time. Hence, even if the initial displacement is large enough to cause torque saturation, eventually the saturation limits will be satisfied. What is the largest value of initial displacement $\theta_z(0)$ that will avoid saturation?

4 In-Lab Assignment

After each part of the In-Lab you implemented, check with the hardware oscilloscope if the PWM and analog output behave like expected, before turning on the motor!

1. Write a program called *lab4.c* from the *lab4_template.c*. The `main` function calls the `init_ECS(4)` function for Lab 4 to initialize the floating point unit, to set the processor speed to 120 MHz and to enable the motor. The *lab4.c* file then calls `init_MIOS_clock` and `init_PWM` to initialize the module prescalers and initial PWM period for the output on eMIOS0 (motor) and to set the clock frequency of eMIOS12 (sc-filter). Scale the input values such that the duty cycle you output is between 24 and 76% (`set_PWMDutyCycleLimits` [0.24, 0.76]). The quadrature decoding function is also initialized for later use in the lab. Set the duty cycle to 0.5 using the `set_PWMDutyCycle` function.

2. Compile and debug your *lab4.c* program using the makefile and the `gmake` command.

3. Power up the boards and use the P&E Debugger to download the *lab4.elf* file to the processor. Run your program and verify that the PWM output operates at 40 kHz and has an initial duty cycle of 50%. Measure the outputs of eMIOS0 and ANout0. Verify that ANout0 is 0 for 50% duty cycle.

4. Using your QADC functions developed in Lab 3, change the *lab4.c* file so that it reads a value from a QADC input and translates it to a duty cycle on the PWM output. Scale the input values such that the duty cycle you output is between 24 and 76%. Connect the potentiometer to 'ANin3' input pin. Verify that the PWM and the AN output are what they should be by connecting them to the oscilloscope. **Show the working code to the TAs.**

5. Switch on the haptic devices at the rear of the box. Connect the encoder of the motor to the Enc_2 socket and the amplifier cable to the socket on the bottom right corner of the interface board. Hold onto the wheel while turning the potentiometer and observe how the torque supplied by the motor to the wheel changes while you do so.

6. Change the PWM frequency to 2 kHz and recompile your program. Output a constant 0.5 duty cycle and note how the motor reacts to the PWM signal. **Do not run this code for a long time.** Change the PWM frequency back to 40 kHz before continuing to the next step.

7. Use the `outputTorque` function to apply a desired amount of torque to the haptic wheel. Compile and run your code to determine what 200 Nmm of torque feels like. Be sure to hold on to the wheel before you start your code. If you want, you can try some other values as well.

8. Test and debug your virtual spring, first by monitoring the PWM and analog signal with the oscilloscope and then by feeling the behavior of the wheel as you turn it. Give the wheel an initial displacement $\theta_z(0)$ large enough to cause initial torque saturation. Record for later analysis: (1) the (approximate) value of displacement $\theta_z(0)$, (2) the frequency of oscillations f_{spring}^{sat} when torque is saturated, and (3) the frequency of oscillations f_{spring}^{exp} after torque comes off saturation. (To determine the frequency of oscillations, it will help to measure several oscillation periods, and average the result. You may find it useful to observe the analog output with the oscilloscope.) Recall that your code is executing in an untimed loop, and try to minimize the amount of time this loop takes to execute. By toggling a bit and displaying the resulting waveform on the oscilloscope, determine the average sample time T of your code. Note this value for later analysis. **Show your measurement on the oscilloscope to the TAs.**

9. Test and debug your virtual wall. Try different values of the spring constant k. On the oscilloscope, you should notice that the constant 50% duty cycle observed when outside the wall changes only when inside the wall.

10. Choose a very large value for the virtual wall's spring constant. Note the reaction of the system.

11. **Show your running virtual wall and spring to the TAs.**

5 Post-Lab Assignment

1. How do the oscillation frequencies f^{sat}_{spring} and f^{exp}_{spring} recorded in part 7 of the In-Lab compare with the theoretical value f^{spring} computed in part 9 of the Pre-Lab? Explain. Use the undamped double integrator model of the wheel and the sample time T you measured to compute the magnitude of the expected closed loop pole locations (see Figure 6), assuming that T is a constant.

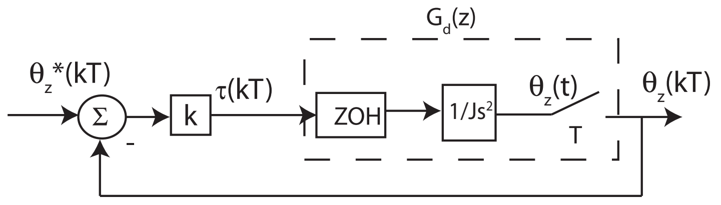


Figure 6: Discretized undamped double integrator model of the wheel/motor system.

2. What effect does the value of the EDPOL bit in the eMIOS Channel Control Register have on the PWM output? Using the continuous time model of the wheel/motor system as an undamped double integrator (see Figure 7), find the locations of the closed loop poles if the EDPOL bit is set incorrectly.

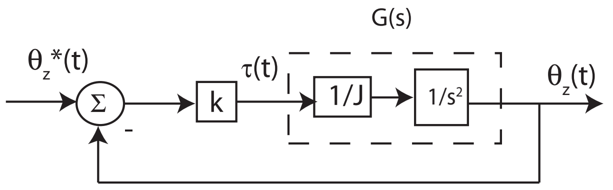


Figure 7: Continuous model of the wheel/motor system.

If you have comments that you feel would help improve the clarity or direction of this assignment, please include them following your post-lab solutions.