**CMPE 250 Laboratory Exercise Twelve**

**D/A Conversion, A/D Conversion, PWM, and Servos**

**with Mixed Assembly Language and C**

Ari Sanders

Performed: November 30th, 2015

Submitted: November 30th, 2015

Lab Section: L4

Instructor: Breton Minnehan

TA: Stephen Moore

Edward Maskelony

Lecture Section: 02

Professor: Roy Melton

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

|  |  |
| --- | --- |
| Your Signature: |  |

**Abstract:**

The purpose of this exercise was familiarization with digital-to-analog conversion, analog-to-digital conversion, and pulse-width-modulation, as well as familiarization with programing in a mix of C and Assembly. This was done using a Freescale Freedom KL46Z and its included DAC0, ADC0, and TPM0 modules. The objective within the exercise was to produce an analog signal, convert it to a digital signal, and use that to move a servo to one of five positions.

**Procedure:**

The exercise began with producing the C code to initialize the KL46Z’s DAC0, ADC0, and TPM0. The DAC was set for conversion of a 12-bit input to an analog value in the range of (0, 3.3] V. The ADC was set for polled conversion from channel 23, which is the DAC output, to an unsigned 10-bit right-justified value, then calibrated. The TPM was set to produce a waveform with a 20ms period and 10% duty period on channel 4.

Following writing the setup and calibration of the new hardware modules, this code was inserted into a C program. The C program utilized an Assembly program which provided it with subroutines from prior labs, including IO functions and lookup tables for DAT and TPM conversions. The C program works as follows: first, a value from 1-5 is requested from the user. This value is then converted to a 12-bit DAC step, then sent to the DAC for conversion to an analog value in the range of (0, 3.3] V. The new value is output to the terminal. The program then requests the ADC to convert the value back to a 10-bit digital one, which it also outputs to the terminal. This value is downconverted from its value from 0-1023 to a value from 0-4. The new value plus one is output to the terminal, and the value itself is used as the index for a lookup table containing the duty values for the PWM. Having found the requested value, it is then sent to the TPM, which sets the servo to the requested position.

The program was tested initially by checking the output values with no servo connected. Having determined they are correct, the board was connected to an oscilloscope. Seeing that the correct frequency was output, the board and a power supply were connected to the servo. The program was then tested again on the actual servo. Noticing the motor did not go all the way to the numbers 1 and 5, the duty period values for 5% and 10% were adjusted by guess-and-check until the motor went all the way. 5% was adjusted from 3000 to 2300, and 10% was adjusted from 6000 to 6500.

**Results:**

The result of this exercise was a program that took the users input, converted it to an analog signal, converted it back to a digital value, then used that to move a servo to the requested position. The values at each step for each valid input are shown in Figure 1.

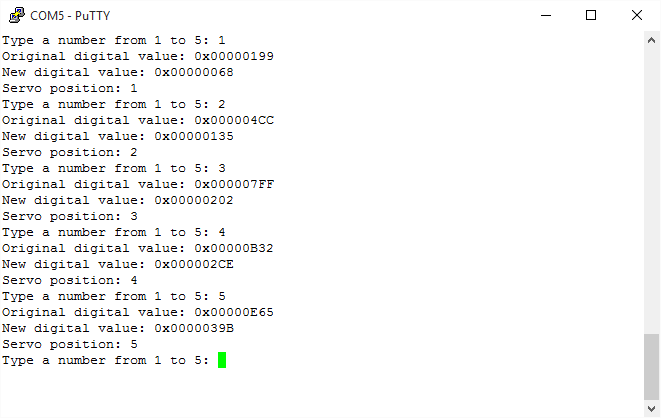


Figure 1: Terminal Output

**Conclusion:**

**Your conclusion should include a comparison and contrast of device programming in C versus assembly language.**

The exercise resulted in a program that properly converted from digital to analog and back, then moved a servo to the requested position. This was a success. One thing that was different in this lab from the previous ones was that it involved device programming in C. This was considerably easier than doing it in Assembly, as the entire control register can be set or updated in a single command, provided proper constants have been defined. In Assembly, that same thing could take five or six commands to do the same thing, again provided proper equates. Similarly, to set or get data from device registers, C can do it in only one command over the several in Assembly. However, this results in the possibility of it being slower. When programming in Assembly, the same register can be reused for several operations without changing. When programming in C, the compiler cannot make such an observation, and will instead reset the required value rather than knowing it is already there. This is just one example of a situation in which an Assembly programmer can optimize their performance over a C programmer. At the device level this is most important, as interfaces are expected to be quick, and devices themselves may not always be. This was overall a good exercise.