Lab Assignment 5

Controlling the Robotic Arm by Software & Object-Oriented Programming

EECE 2160 – Embedded Design: Enabling Robotics

Section: 13196

Instructor: John Kimani

Author: Sean Vumbaco

vumbaco.s@northeastern.edu

Partner: Jiwoo (Ben) Yoon

yoon.ji@northeastern.edu

Submit Date: 10/20/2022

Due Date: 10/20/2022

Abstract

Lab 5 introduced the robotic arm, a system of 5 servo motors that work in conjunction to act and perform the function of a rudimentary arm. Throughout the course of the lab, a class was developed to be able to simply and robustly interface with each motor by building off the software used throughout previous labs. This experiment utilized the DE1SoCfpga class to interface with the DE1SoCfpga board while the newly developed RoboticArm class was written to write to the 5 servo motors of the robotic arm. By the end of the lab, students were able to write to each servo of the device individually, controlling both the direction of movement, precise extent of movement, and the speed at which the motors operate.

Introduction

While the De1-SoC has several configurable hardware systems on its own board, the processor may be used as an intermediary interface between a Linux-based operating system and other electronic devices. In this lab, students employed the De1-SoC board to write to a robotic arm. This robotic arm was simply a configuration of 5 servo motors. Although these motors may still be controlled using the same RegisterWrite() function used for the LED lights and 7-segment displays on the De1-SoC, they operate using a PWM signal, meaning that an alternating signal of 1's and 0's must be sent using specific intervals. This alternating signal controls the positioning of each servo motor in the robotic arm system, allowing the user to control specific directions of movement and locations within a 180° spectrum of angles.

Throughout the experiment, students utilized this PWM concept to control each individual servo on the robotic arm. This lab demanded a generalized knowledge of how PWM signals work and how they may be written to devices by writing 1's and 0's to the register in specified segments of time. Otherwise, the general hardware concepts that go behind the robotic arm design and advanced servo motor processes were not expected of the programmer for this lab. In general, students simply adapted the procedure used to write to the LEDs and 7-segment displays and modified it using a simple alternation of 1's and 0's in specified intervals to control the servo motors used in this lab.

Lab Discussion

Equipment Used in Lab			
Hardware		**Software**	
Name	**Description**	**Name**	**Description**
Altera DE1-SoC FPGA Board	FPGA device with several embedded systems	MobaXterm	Secure Shell (ssh) client
ARM Processor	DE1-SoC processor used to run C++ scripts	Linux	Operating System (OS)

Host computer	Student workstation	Vi	Linux command line text editor tool
USB-to-Ethernet adaptor cable	Cable connecting DE1-SoC to host computer	G++	C/C++ script compiler under Linux
Robotic Arm	5-servo motor system (base, bicep, elbow, wrist, gripper)	C++	Common programming language
		iostream	C++ library to simplify basic formatting
		stdlib.h	C++ library for memory allocation
		stdio.h	C++ library for input/output functions
		unistd.h	C++ library for misc functions
		fcntl.h	C++ library for working with files
		sys/mman.h	C++ library for memory management declarations

Table 1. Hardware and Software Tools used in Lab 5

Results and Analysis

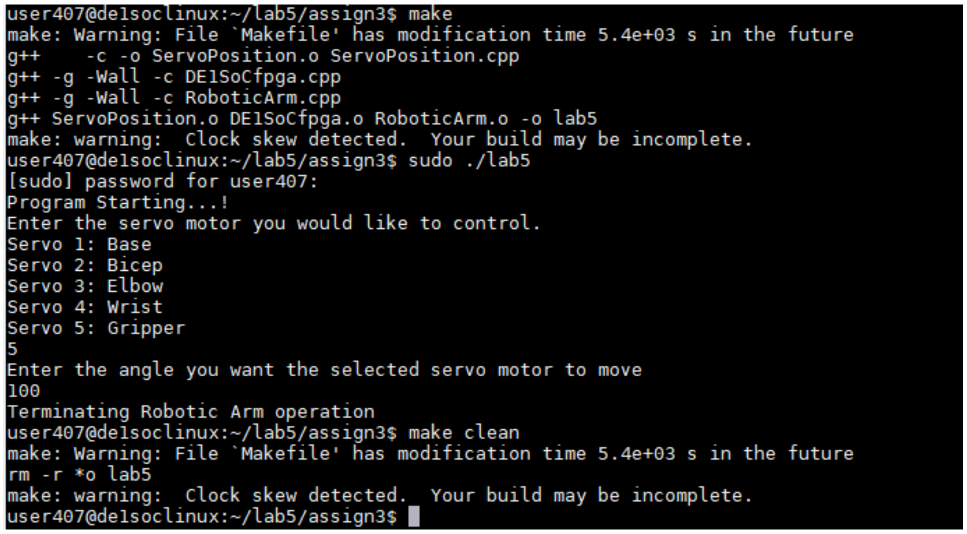
The first part of the lab simply consisted of setting up the robotic arm. For safety purposes, the contraption was clamped to the table. It was also connected to power and the De1SoC board to receive signals. Servo 1 (the base), servo 2 (the bicep), servo 3 (the elbow) servo 4 (the wrist) and servo 5 (the gripper) were each controlled through pins 29, 27, 25, 23, and 28 on the connector, respectively.

In completing the software demanded by this experiment, students simply generated a RoboticArm class (through the implementation of RoboticArm.h and RoboticArm.cpp files) to interface with the robotic arm used throughout the lab. The RoboticArm class inherited all the functions from the DE1SoCfpga class, which ultimately amounted to the streamlined use of the RegisterRead() and RegisterWrite() functions without creating a separate DE1SoCfpga object.

The software encompassing the DE1SoCfpga class is included in Appendices J and K (DE1SoCfpga.h and DE1SoCfpga.cpp, respectively). This code was written and developed before starting this lab procedure. The RoboticArm class, however, was developed throughout the duration of this lab and may be found in Appendices C, D and G.

As the very first task assigned by the user to the robotic arm, every servo was programmed to orient themselves in the 90° position at the same time. The software used for this phase of the experiment is included in Appendix A.

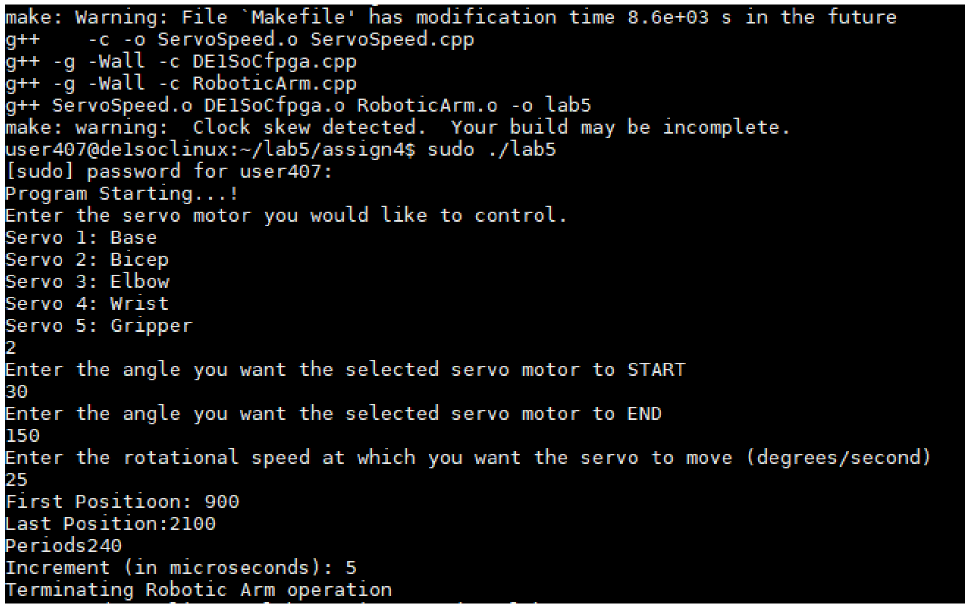
Next, the entire system including the robotic arm, the De1SoC board, and the Linux-based operating system was configured to accept input from the user and orient the arm accordingly. In this case, the user was prompted to enter a servo number between 1 and 5 as well as the position of the specified servo. Boundaries on the user input were put in place to ensure that the entered servo was valid and the entered position was not extreme or out of range. The output presented on the screen is included in Figure 1.



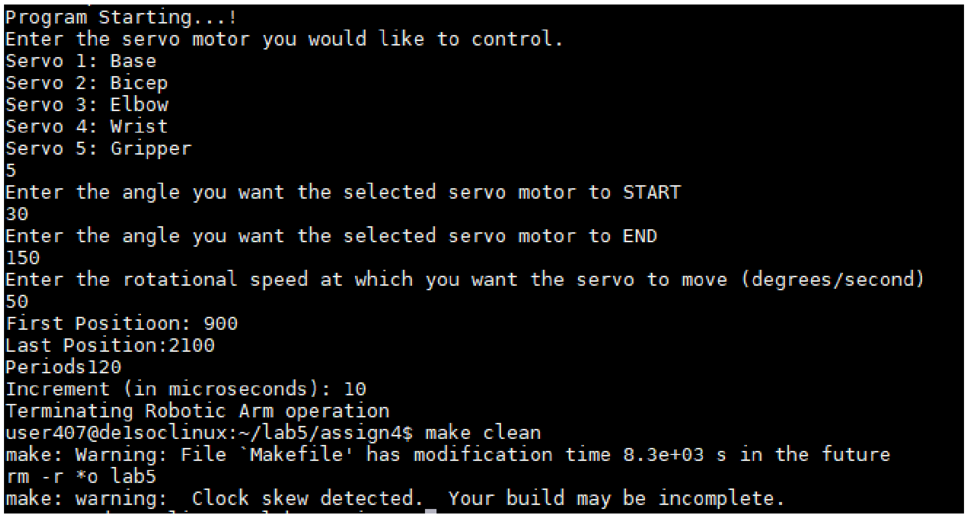
```
user407@de1soclinux:~/lab5/assign3$ make
make: Warning: File `Makefile' has modification time 5.4e+03 s in the future
g++    -c -o ServoPosition.o ServoPosition.cpp
g++ -g -Wall -c DE1SoCfpga.cpp
g++ -g -Wall -c RoboticArm.cpp
g++ ServoPosition.o DE1SoCfpga.o RoboticArm.o -o lab5
make: warning:  Clock skew detected.  Your build may be incomplete.
user407@de1soclinux:~/lab5/assign3$ sudo ./lab5
[sudo] password for user407:
Program Starting...!
Enter the servo motor you would like to control.
Servo 1: Base
Servo 2: Bicep
Servo 3: Elbow
Servo 4: Wrist
Servo 5: Gripper
5
Enter the angle you want the selected servo motor to move
100
Terminating Robotic Arm operation
user407@de1soclinux:~/lab5/assign3$ make clean
make: Warning: File `Makefile' has modification time 5.4e+03 s in the future
rm -r *o lab5
make: warning:  Clock skew detected.  Your build may be incomplete.
user407@de1soclinux:~/lab5/assign3$ 
```

Figure 1. User Interface for Assignment 3 – Controlling Individual Servo Positions

Finally, the system was programmed to be receptive to not only the servo to move, but also starting and ending orientations (both between 20° and 160°) as well as the speed to travel from one position to another. The screen presented to the user to enter this information is included in Figure 2, and the software used to generate this interface is included in Appendix H. This phase of the experiment was when the GenerateVariablePWM() method was included in the RoboticArm.cpp file (Appendix G).



```
make: Warning: File `Makefile' has modification time 8.6e+03 s in the future
g++    -c -o ServoSpeed.o ServoSpeed.cpp
g++ -g -Wall -c DE1SoCfpga.cpp
g++ -g -Wall -c RoboticArm.cpp
g++ ServoSpeed.o DE1SoCfpga.o RoboticArm.o -o lab5
make: warning:  Clock skew detected.  Your build may be incomplete.
user407@de1soclinux:~/lab5/assign4$ sudo ./lab5
[sudo] password for user407:
Program Starting...!
Enter the servo motor you would like to control.
Servo 1: Base
Servo 2: Bicep
Servo 3: Elbow
Servo 4: Wrist
Servo 5: Gripper
2
Enter the angle you want the selected servo motor to START
30
Enter the angle you want the selected servo motor to END
150
Enter the rotational speed at which you want the servo to move (degrees/second)
25
First Positioon: 900
Last Position:2100
Periods240
Increment (in microseconds): 5
Terminating Robotic Arm operation
```



```
Program Starting...!
Enter the servo motor you would like to control.
Servo 1: Base
Servo 2: Bicep
Servo 3: Elbow
Servo 4: Wrist
Servo 5: Gripper
5
Enter the angle you want the selected servo motor to START
30
Enter the angle you want the selected servo motor to END
150
Enter the rotational speed at which you want the servo to move (degrees/second)
50
First Positioon: 900
Last Position:2100
Periods120
Increment (in microseconds): 10
Terminating Robotic Arm operation
user407@de1soclinux:~/lab5/assign4$ make clean
make: Warning: File `Makefile' has modification time 8.3e+03 s in the future
rm -r *o lab5
make: warning:  Clock skew detected.  Your build may be incomplete.
```

Figure 2. User Interface for Assignment 4 – Controlling Servo Motor Speed

Conclusion

As shown in the results and analysis, we can conclude that we were successful in controlling the robotic arm using C++ and object-oriented programming. The lab displayed fundamental connections between hardware and software by demonstrating the uses of Pulse-Width Modulation and manipulating it through C++. In addition, we were able to utilize techniques such as inheritance, header files, and Makefiles in Linux, which could prove to be extremely useful in industry.

Appendices

Appendix A: main.cpp (Assignment 2)

```
#include <iostream>
#include "RoboticArm.h"
#include "string"

using namespace std;


int main()
{
  cout << "Program Starting...!" << endl;

// Declare a RoboticArm dynamic object
  RoboticArm *servo = new RoboticArm;

// Set pin direction to output for all servos (1 = output, 0 = input)
// Your code here...
  for(int i=1; i<6; i++){
      servo->PinDirectionSetup(i, 1);
      servo->setServoPosition(90, i);
      servo->GeneratePWM(50, i);
```

```
   }

// Done
  delete servo;
}
```

Appendix B: Makefile (Assignment 2)

```
lab5: main.o DE1SoCfpga.o RoboticArm.o

      g++ main.o DE1SoCfpga.o RoboticArm.o -o lab5

main.o: main.cpp DE1SoCfpga.h RoboticArm.h

      g++ -g -Wall -c main.cpp

DE1SoCfpga.o: DE1SoCfpga.h DE1SoCfpga.cpp

      g++ -g -Wall -c DE1SoCfpga.cpp

RoboticArm.o: RoboticArm.h RoboticArm.cpp

      g++ -g -Wall -c RoboticArm.cpp

clean:

      rm -r *o lab5
```

Appendix C: RoboticArm.h (Assignment 2, 3 and 4)

```
#ifndef ROBOTICARM_H
#define ROBOTICARM_H
#include "DE1SoCfpga.h"
#include <iostream>

class RoboticArm : public DE1SoCfpga {
private:
int servo_pin[5]; // An array of pin numbers corresponding to each servo
int servo_position[5]; // An array of positions the servo is set to be moved
int SetRegisterBit(unsigned int reg_offset, int bitNumber, int state); // Sets specified
bit in the given register

// Function prototypes
public:
RoboticArm();
~RoboticArm();
void PinDirectionSetup(int servo_num, int direction);
int setServoPosition(int degrees, int servo_num);
void GeneratePWM(int num_periods, int servo_num);

};
```

```
#endif
```

<u>Appendix D: RoboticArm.cpp</u> (Assignment 2 and 3)

```
#include "RoboticArm.h"
#include <iostream>
#include "string"
// Imported header file and modules

using namespace std;

/**
* RoboticArm() sets array values that were created in the header file
*
* @param  none
* @return none
*/
RoboticArm::RoboticArm(){
   servo_pin[0] = 29;
   servo_pin[1] = 27;
   servo_pin[2] = 25;
   servo_pin[3] = 23;
   servo_pin[4] = 28;
   servo_position[0] = 1500;
   servo_position[1] = 1500;
   servo_position[2] = 1500;
   servo_position[3] = 1500;
   servo_position[4] = 1500;

}

// Destructor that outputs a message
RoboticArm::~RoboticArm(){
   cout << "Terminating Robotic Arm operation" << endl;
}

/**
* SetRegisterBit() sets the bit to a value depending on state in a specified memory
location
*
* @param  dataValue binary number
* @param  index of the binary number
* @param  state 0 or 1
* @return dataValue
*/
int RoboticArm::SetRegisterBit(unsigned int dataValue, int index, int state){
    int mask = (1 << index);
    if (state == 1) {
        dataValue = dataValue | mask;
    }
    else {
        dataValue = dataValue & ~mask;
    }
    return dataValue;
}

/**
* PinDirectionSetup() sets the servo pin to be either an input or output
*
* @param  servo_num positive integer [1,5]
* @param  direction 0 or 1
* @return none
*/
void RoboticArm::PinDirectionSetup(int servo_num, int direction){
   int in_out_state = RegisterRead(JP2_BASE);
   in_out_state = SetRegisterBit(in_out_state, servo_pin[servo_num-1], direction);
   RegisterWrite(JP2_BASE, in_out_state);
}

/**
* setServoPosition() Converts the degrees [0 to 180] to position in µsec [600 to 2400]
for the
* specified servo and updates the servo_position array for that servo
*
* @param  degrees positive integer
* @param  servo_num positive integer [1,5]
* @return dCycle a pulse in µsec
*/
int RoboticArm::setServoPosition(int degrees, int servo_num){
   int dCycle;
    // Converts degrees to µsec
   dCycle = (degrees*10) + 600;
   servo_position[servo_num-1] = dCycle;
   return dCycle;
}

/**
* GeneratePWM() generates PWM signal
*
* @param  num_periods positive integer
* @param  servo_num positive integer [1,5]
* @return none
*/
void RoboticArm::GeneratePWM(int num_periods, int servo_num){
   int DirBit;
   for(int i=0; i <= num_periods; i++){
      DirBit = RegisterRead(JP2_DDR);
      DirBit = SetRegisterBit(DirBit, servo_pin[servo_num -1], 1);
      RegisterWrite(JP2_DDR, DirBit);
        // The duty cycle is determined by usleep
      usleep(servo_position[servo_num-1]);
      DirBit = SetRegisterBit(DirBit, servo_pin[servo_num -1], 0);
                RegisterWrite(JP2_DDR, DirBit);
      usleep(20000 - servo_position[servo_num-1]);
   }
}
```

Appendix E: ServoPosition.cpp (Assignment 3)

```
#include <iostream>
#include "RoboticArm.h"
#include "string"

using namespace std;

/**
* getServoNum() receives servo number inputs from user
*
* @param  none
* @return getServoNum() recursively
*/
int getServoNum() {
   int servo_num;

   cout << "Enter the servo motor you would like to control." << endl;
   cout << "Servo 1: Base" << endl;
   cout << "Servo 2: Bicep" << endl;
   cout << "Servo 3: Elbow" << endl;
   cout << "Servo 4: Wrist" << endl;
   cout << "Servo 5: Gripper" << endl;

  cin >> servo_num;

   if (servo_num >= 1 && servo_num <= 5) {
      return servo_num;
   }
  else {
      cout << "Not a valid servo number" << endl;
      return getServoNum();
   }
}

/**
* getServoPosition() receives angle (in degrees) inputs from user
*
* @param  none
* @return getServoPosition() recursively
*/
int getServoPosition() {
   int angle;
   cout << "Enter the angle you want the selected servo motor to move" << endl;
   cin >> angle;

    // In order to reduce servo damage risk
   if (angle >= 20 && angle <= 160){
      return angle;
   }

   else{
      cout << "Not a valid angle" << endl;
      return getServoPosition();
   }

}

int main()
{
  cout << "Program Starting...!" << endl;

// Declare a RoboticArm dynamic object
  RoboticArm *servo = new RoboticArm;


  int servo_num = getServoNum();
        int servo_position = getServoPosition();

   servo->PinDirectionSetup(servo_num, 1);
   servo->setServoPosition(servo_position, servo_num);
   servo->GeneratePWM(50, servo_num);

// Done
   delete servo;
}
```

Appendix F: Makefile (Assignment 3)

```
lab5: ServoPosition.o DE1SoCfpga.o RoboticArm.o

      g++ ServoPosition.o DE1SoCfpga.o RoboticArm.o -o lab5

main.o: ServoPosition.cpp DE1SoCfpga.h RoboticArm.h

      g++ -g -Wall -c ServoPosition.cpp

DE1SoCfpga.o: DE1SoCfpga.h DE1SoCfpga.cpp

      g++ -g -Wall -c DE1SoCfpga.cpp

RoboticArm.o: RoboticArm.h RoboticArm.cpp

      g++ -g -Wall -c RoboticArm.cpp
```

```
clean:

	rm -r *o lab5
```

Appendix G: RoboticArm.cpp (Assignment 4)

```
#include "RoboticArm.h"
#include <iostream>
#include "string"
// Imported header file and modules

using namespace std;
/**
* RoboticArm() sets array values that were created in the header file
*
* @param  none
* @return none
*/
RoboticArm::RoboticArm(){
    servo_pin[0] = 29;
    servo_pin[1] = 27;
    servo_pin[2] = 25;
    servo_pin[3] = 23;
    servo_pin[4] = 28;
    servo_position[0] = 1500;
    servo_position[1] = 1500;
    servo_position[2] = 1500;
    servo_position[3] = 1500;
    servo_position[4] = 1500;

}

// Destructor that outputs a message
RoboticArm::~RoboticArm(){
    cout << "Terminating Robotic Arm operation" << endl;
}

/**
* SetRegisterBit() sets the bit to a value depending on state in a specified memory
location
*
* @param  dataValue binary number
* @param  index of the binary number
* @param  state 0 or 1
* @return dataValue
*/
int RoboticArm::SetRegisterBit(unsigned int dataValue, int index, int state){
    int mask = (1 << index);
    if (state == 1) {
        dataValue = dataValue | mask;
    }
    else {
        dataValue = dataValue & ~mask;
    }
    return dataValue;
}

/**
* PinDirectionSetup() sets the servo pin to be either an input or output
*
* @param  servo_num positive integer [1,5]
* @param  direction 0 or 1
* @return none
*/
void RoboticArm::PinDirectionSetup(int servo_num, int direction){
    int in_out_state = RegisterRead(JP2_BASE);
    in_out_state = SetRegisterBit(in_out_state, servo_pin[servo_num-1], direction);
    RegisterWrite(JP2_BASE, in_out_state);
}

/**
* setServoPosition() Converts the degrees [0 to 180] to position in μsec [600 to 2400]
for the
* specified servo and updates the servo_position array for that servo
*
* @param  degrees positive integer
* @param  servo_num positive integer [1,5]
* @return dCycle a pulse in μsec
*/
int RoboticArm::setServoPosition(int degrees, int servo_num){
    int dCycle;
    // Converts degrees to μsec
    dCycle = (degrees*10) + 600;
    servo_position[servo_num-1] = dCycle;
    return dCycle;
}

/**
* GeneratePWM() generates PWM signal
*
* @param  num_periods positive integer
* @param  servo_num positive integer [1,5]
* @return none
*/
void RoboticArm::GeneratePWM(int num_periods, int servo_num){
    int DirBit;
    for(int i=0; i <= num_periods; i++){
        DirBit = RegisterRead(JP2_DDR);
        DirBit = SetRegisterBit(DirBit, servo_pin[servo_num -1], 1);
        RegisterWrite(JP2_DDR, DirBit);
        // The duty cycle is determined by usleep
        usleep(servo_position[servo_num-1]);
        DirBit = SetRegisterBit(DirBit, servo_pin[servo_num -1], 0);
        RegisterWrite(JP2_DDR, DirBit);
        usleep(20000 - servo_position[servo_num-1]);
    }
}

/**
```

```
* GenerateVariablePWM() evolves the pulse linearly
*
* @param  first_pulse posInt in µsec
* @param  last_pulse posInt in µsec
* @param  num_periods positive integer
* @param  servo_num positive integer
* @return none
*/
void RoboticArm::GenerateVariablePWM(int first_pulse, int last_pulse, int num_periods,
int servo_num){
   int degree;
   float increment;
    // If the start position degree is larger than the end
   if (first_pulse > last_pulse) {
      increment = -1 * (first_pulse - last_pulse) / ((float) num_periods);
   }
    // If the end position degree is larger than the start
    else {
      increment = (last_pulse - first_pulse) / ((float) num_periods);
   }
   float dCycle = first_pulse;
    // Changes the dCycle every period by the increment
   for (int i = 0; i < num_periods; i++) {
      degree = (int) ((dCycle - 600) / 10);
      setServoPosition(degree, servo_num);
      GeneratePWM(1, servo_num);
      dCycle += increment;
   }
}
```

Appendix H: ServoSpeed.cpp (Assignment 4)

```
#include <iostream>
#include "RoboticArm.h"
#include "string"

using namespace std;

/**
* getServoNum() receives servo number inputs from user
*
* @param  none
* @return getServoNum() recursively
*/
int getServoNum() {
   int servo_num;

   cout << "Enter the servo motor you would like to control." << endl;
   cout << "Servo 1: Base" << endl;
   cout << "Servo 2: Bicep" << endl;
   cout << "Servo 3: Elbow" << endl;
   cout << "Servo 4: Wrist" << endl;
   cout << "Servo 5: Gripper" << endl;

  cin >> servo_num;
```

```
   if (servo_num >= 1 && servo_num <= 5) {
      return servo_num;
   }
  else {
      cout << "Not a valid servo number" << endl;
      return getServoNum();
   }
}

/**
* getStartPosition() receives angle (in degrees) input from user
*
* @param  none
* @return getStartPosition() recursively
*/
int getStartPosition() {
   int angle;

   cout << "Enter the angle you want the selected servo motor to START" << endl;
   cin >> angle;

  if (angle >= 20 && angle <= 160){
      return angle;
   }

   else{
      cout << "Not a valid angle" << endl;
      return getStartPosition();
   }

}

/**
* getEndPosition() receives angle (in degrees) input from user
*
* @param  none
* @return getEndPosition() recursively
*/
int getEndPosition() {
   int angle;

        cout << "Enter the angle you want the selected servo motor to END" << endl;
        cin >> angle;

        if (angle >= 20 && angle <= 160){
                return angle;
        }

        else{
                cout << "Not a valid angle" << endl;
                return getEndPosition();
        }
```

```
}

/**
* getRotationalSpeed() receives rotational speed from user
*
* @param  none
* @return speed degrees/second
*/
int getRotationalSpeed() {
   int speed;

   cout << "Enter the rotational speed at which you want the servo to move
(degrees/second)" << endl;
   cin >> speed;

   // No defined restraints for entered speed
   return speed;
}


int main()
{
  cout << "Program Starting...!" << endl;

// Declare a RoboticArm dynamic object
  RoboticArm *servo = new RoboticArm;

// Set pin direction to output for all servos (1 = output, 0 = input)
// Your code here...

  // Get information from user
   int servo_num = getServoNum();
    int start_position = getStartPosition();
   int end_position = getEndPosition();
   int speed = getRotationalSpeed();

    // Converts degree to µsec
   int start_pulse = start_position * 10 + 600;
   int end_pulse = end_position * 10 + 600;
   float num_cycles;
   if (start_pulse > end_pulse) {
        // number of periods is frequency multiplied by the difference between start and
end position
        // divided by rotational speed
      num_cycles = 50 * ((float) start_position - end_position) / speed;
   } else {
      num_cycles = 50 * ((float) end_position - start_position) / speed;
   }
        int num_periods = (int) num_cycles;

   // Set to initial position and hold for 1 second
   servo->PinDirectionSetup(servo_num, 1);
   servo->setServoPosition(start_position, servo_num);
   servo->GeneratePWM(50, servo_num);
   servo->GenerateVariablePWM(start_pulse, end_pulse, num_periods, servo_num);

// Done
  delete servo;
}
```

Appendix I: Makefile (Assignment 4)

```
lab5: ServoSpeed.o DE1SoCfpga.o RoboticArm.o

      g++ ServoSpeed.o DE1SoCfpga.o RoboticArm.o -o lab5

main.o: ServoSpeed.cpp DE1SoCfpga.h RoboticArm.h

      g++ -g -Wall -c ServoSpeed.cpp

DE1SoCfpga.o: DE1SoCfpga.h DE1SoCfpga.cpp

      g++ -g -Wall -c DE1SoCfpga.cpp

RoboticArm.o: RoboticArm.h RoboticArm.cpp

      g++ -g -Wall -c RoboticArm.cpp

clean:

      rm -r *o lab5
```

Appendix J: DE1SoCfpga.h

```
#ifndef DE1SOCFPGA_H
#define DE1SOCFPGA_H

// Physical base address of FPGA Devices
const unsigned int LW_BRIDGE_BASE   = 0xFF200000;  // Base offset
// Length of memory-mapped IO window
const unsigned int LW_BRIDGE_SPAN   = 0x00005000;  // Address map size
const unsigned int LEDR_BASE      = 0x00000000;  // Leds offset
const unsigned int SW_BASE       = 0x00000040;  // Switches offset
const unsigned int KEY_BASE       = 0x00000050;  // Push buttons offset

const unsigned int JP2_BASE = 0x00000070;  // J2 Data register
const unsigned int JP2_DDR = 0x00000074; // Data direction


const unsigned int HEX3_HEX0_BASE = 0x00000020; // HEX Reg1 Offset
const unsigned int HEX5_HEX4_BASE = 0x00000030; // HEX Reg2 offset

class DE1SoCfpga {

    char *pBase;
    int fd;
public:
// Function prototypes
    DE1SoCfpga();

    ~DE1SoCfpga();

    void RegisterWrite(unsigned int offset, int value);

    int RegisterRead(unsigned int offset);

};
#endif
```

Appendix K: DE1SoCfpga.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>
#include <bitset>
#include <math.h>
#include "DE1SoCfpga.h"
using namespace std;
// Imported modules and header file

// Class constructor
DE1SoCfpga::DE1SoCfpga()
{
    int *fdp = &fd;
    // Open /dev/mem to give access to physical addresses
    *fdp = open( "/dev/mem", (O_RDWR | O_SYNC));
    if (*fdp == -1)    //  check for errors in opening /dev/mem
    {
        cout << "ERROR: could not open /dev/mem..." << endl;
        exit(1);
    }

    // Get a mapping from physical addresses to virtual addresses
    char *virtual_base = (char *)mmap (NULL, LW_BRIDGE_SPAN, (PROT_READ |
                                                             PROT_WRITE), MAP_SHARED,
*fdp, LW_BRIDGE_BASE);
    if (virtual_base == MAP_FAILED)    // check for errors
    {
        cout << "ERROR: mmap() failed..." << endl;
        close (fd);   // close memory before exiting
        exit(1);        // Returns 1 to the operating system;
    }
    pBase = virtual_base;
}

// Class destructor
```

```
DE1SoCfpga::~DE1SoCfpga() {
    // Finalize() function goes here
    if (munmap (pBase, LW_BRIDGE_SPAN) != 0)
    {
        cout << "ERROR: munmap() failed..." << endl;
        exit(1);
    }
    close (fd);   // close memory
}

// Writes a value to a specified place in memory
void DE1SoCfpga::RegisterWrite(unsigned int offset, int value) {
    * (volatile unsigned int *)(pBase + offset) = value;
}

// Reads a value from a specified place in memory
int DE1SoCfpga::RegisterRead(unsigned int offset) {
    return * (volatile unsigned int *)(pBase + offset);
}
```