Controlling the Robotic Arm by Software

Introduction

In this lab, you will move the robotic arm using programs written in C++ running on the ARM processor in the ZedBoard. Your program will communicate with the robotic arm servos through the Peripheral Module (Pmod) connectors using Pulse Width Modulation (PWM) signals. First, you will only control the arm position, and later you will control its speed as well.

Input/output redirection and pipes

The two major means of interaction between a Unix application and the user are the keyboard as an input device, and the terminal as an output device. Applications can read input strings from the keyboard using function `scanf` in C, or global object `std::cin` in C++. Likewise, applications can dump strings into the terminal using function `printf` in C, or global object `std::cout` in C++. In Unix operating systems, the keyboard and terminal are represented by two virtual device files, named *standard input* and *standard output*, respectively.

Unix allows us to redirect the standard input and output of a program into other files. If we choose to do so, a program that expects to read its input from the keyboard will instead read the content of an existing file. Or the output that is displayed on the terminal under normal circumstances can instead be dumped directly into a file. The shell provides a dedicated syntax for input/output redirection through the < and > operators. Try and type the following command in the shell:

```
$ echo “Hello”
```

As you learned in the previous lab assignment, `echo` is a command that prints its arguments into the standard output. In this case, string `Hello` is dumped into the terminal. Using operator `>`, we can redirect this output into a file named `message.txt` as follows:

```
$ echo “Hello” > message.txt
```

If file `output.txt` existed before running the command above, it is overwritten and its content is fully replaced with string `Hello`. If you run the command above multiple times, you will notice that the content of file `output.txt` does not change as compared to the first time. If, on the contrary, you want to keep the previous content of file `message.txt` and append string `Hello` to it, you can use operator >> instead, as follows:

```
echo “Hello” >> message.txt
```

To illustrate the redirection of the standard input, let us use command `wc`. When you run this command without arguments, it reads lines of text from the standard input, and then shows a message counting the number of lines, words, and characters entered. When you enter text from the keyboard you need to notify `wc` about the end of the input with an `EOF` (end-of-file) character, which you can provide by pressing Control+D. Example:

```
$ wc
Hello
How are you?
       2       4      19
```

If, instead of reading the input from the keyboard, you want to read it directly from file `message.txt`, you can use operator < as follows:

```
$ wc < message.txt
       1       1       6
```

Pre-Lab Assignment
The following reading list will help you to understand how PWM Signals work and how servo motors in the robotic arm work

[1] *Require Reading:* PWM Tutorial on Blackboad
- https://blackboard.neu.edu/

Pre 9.1) Use the information in document [1], write a C++ function (`degreeToOnDelay()`) to calculate the on period of a PWM signal for controlling a servomotor. The C++ function receives the servo position (0 to 180 degrees) and returns the time in micro seconds that PWM signal should be on during each period so that the RC servo moves to the specified servo position.

Pre 9.2) This assignment involves the creation of two shell scripts on the COE machines. Attach the scripts to your pre-lab submission with the file names specified below.
a) Write a shell script named `CombineFiles.sh` that takes three file names as arguments: `<dest>`, `<src1>`, and `<src2>`. The script should read the content of files `<src1>` and `<src2>`, and combine them into a new file called `<dest>`. For example, running

```
$ ./CombineFiles.sh new.txt /etc/passwd message.txt
```

will combine the content of files `/etc/passwd` and `message.txt` —file message.txt must exist— into a new file named `new.txt`. Attach script `CombineFiles.sh` in your lab submission on Blackboard.
b) Write a shell script named `wc2.sh` that takes two file names as arguments. The script should count the number of lines, words, and characters adding up both files. You can invoke script `CombineFiles.sh` from here, or you can find an alternative strategy. Attach script `wc2.sh` in your pre-lab submission on Blackboard.

Lab Assignment

Controlling the servos

The robotic arm has five servo motors named *base*, *bicep*, *elbow*, *wrist*, and *gripper*. Each servo can rotate through 180°, and its exact position is determined by a Pulse Width Modulation (PWM) signal sent by the ZedBoard, generated in our C++ programs. A PWM signal is a periodic signal constantly transitioning between digital values equal to 0 or 1.

For our servo motors, the frequency of the PWM signal is set to a fixed value of 50Hz. In other words, the signal period is equal to 20ms. Within that period, however, we can vary the amount of time that the signal is set to 1 —this time is referred to as the signal pulse— and the time that it is set to 0. The ratio between the time the signal is set to 1 and the time it is set to 0 is is referred to as the *duty cycle*.

In our case, acceptable values for the duty cycle range between 3% (a 0.6ms pulse) and 12% (a 2.4ms pulse). Different duty cycles in the PWM signal are used to bring the servos to different positions, as shown here:

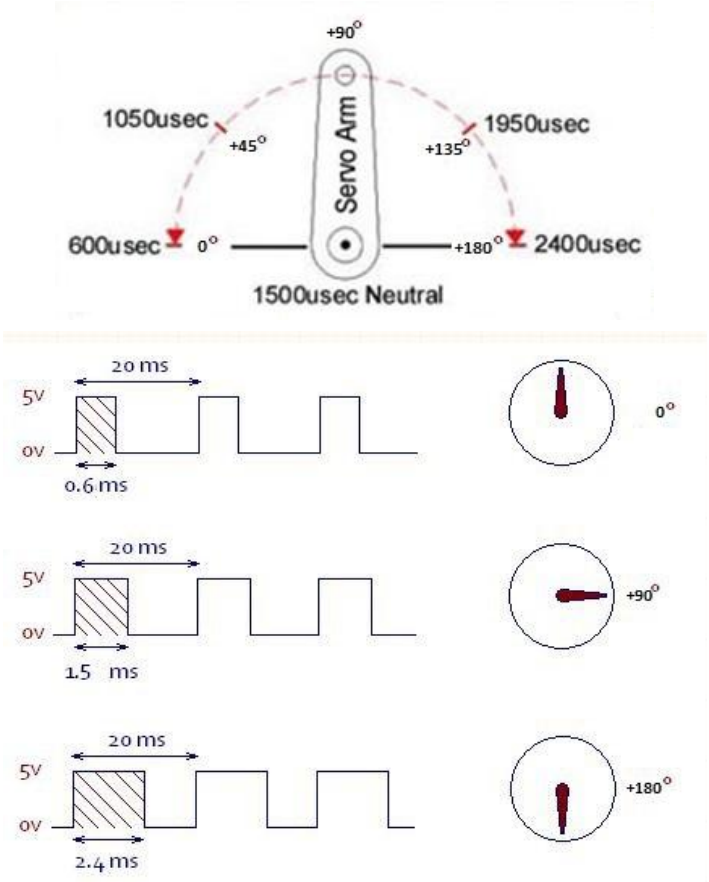
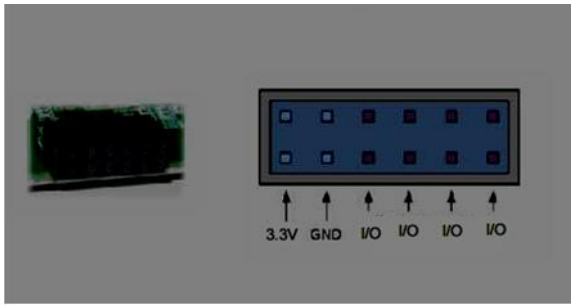


Figure 1. Servo motor operation for different PWM signal duty

Warning: Some servos in our robot don't have their full range of motion available, as other robot components may stand in their way. It is thus safer to reduce the range of valid pulse widths from [0.6ms, 2.4ms] to [0.7ms, 2.3ms]. Alternatively, you can avoid entering angles that are right toward the edges of this range, reducing the range [0°, 180°] to [20°, 160°], for exmaple.

The Peripheral Module (Pmod) connectors

In order to send the generated PWM signals to the Robotic arm, you will use a Pmod connector. The ZedBoard has 5 Pmod connectors, labelled *JA*, *JB*, *JC*, *JD*, and *JE*. Each Pmod connector has 12 pins organized in two rows:



The first four Pmod connectors (*JA* through *JD*) interface with the programmable logic on the ZedBoard, while the fifth

connector (*JE*) interfaces both with the programmable logic and the ARM microprocessor. Since we're now interested in controlling the robotic arm by software, connector *JE* will be used.

Pmod connectors are seen by the operating system as general-purpose input/output (GPIO) ports. We will refer to them as Pmod or GPIO ports, interchangeably.

Setting up the GPIO ports

GPIO ports offer great configuration flexibility, allowing the processor to use them as either input or output ports. This means that, before starting our experiments, we need to set up the hardware interface by first allocating the ports to be used, and specifying their direction. When we finish using them, they need to be freed. All these actions require you to be logged in as *root*.

A GPIO port can be allocated by writing a string containing a two-digit number into virtual file /sys/class/gpio/export. We start by reserving pin *JE1*, identified with value 13 by the operating system, with the following shell command (remember, log in as *root*):

```
$ echo 13 > /sys/class/gpio/export
```

After running this command, a new virtual directory named /sys/class/gpio/gpio13/ is created, containing a set of virtual files that can be used to configure port 13. If you're curious, you can run command ls to inspect the content of both directory /sys/class/gpio/ and /sys/class/gpio/gpio13/.

In order to set the direction of a reserved pin, you can write string in or out into virtual file /sys/class/gpio/gpio<NR>/direction, where <NR> is the number of the reserved pin. Set up pin 13 as an output port with the following command (still, logged in as *root*):

```
$ echo out > /sys/class/gpio/gpio13/direction
```

If a port is configured as an output port, you can configure its current state by writing the ASCII character corresponding to number 0 or 1 into virtual file /sys/class/gpio/gpio<NR>/value, where <NR> is the number of the reserved pin. For example, you can set output port 13 to take a value of 1 with command

```
$ echo 1 > /sys/class/gpio/gpio13/value
```

In order to read the value provided by an external device into a port configured as an input, you can run command cat on the virtual file representing the port value. For example, if port 13 was configured as an input port, you could display its value by running

```
$ cat /sys/class/gpio/gpio13/value
```

The following table shows the connections between the pins of the *JE* connector and the servos. The middle column shows the port number used in Linux to access each pin using a GPIO virtual file. This number determines the name of the virtual file names to be written to in order to control the servos.

PIN name	**Port number in Linux**	**Servo**
JE1	13	Base
JE2	10	Bicep

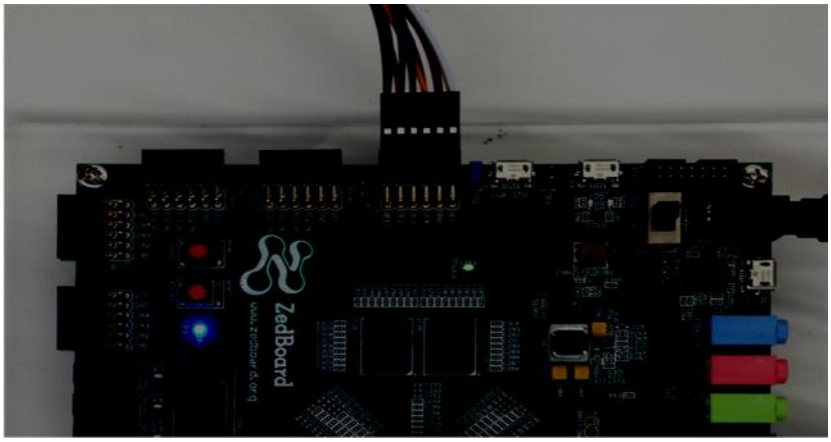
JE3	11	Elbow
JE4	12	Wrist
JE7	0	Gripper
JE8	9	–
JE9	14	–
JE10	15	–

Connecting the robotic arm to the ZedBoard

IMPORTANT

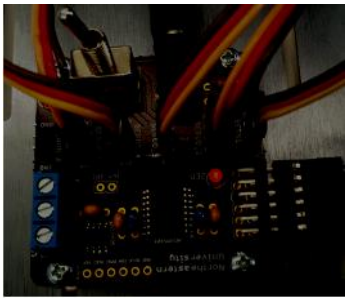
- Secure the robot to the table using clamps.
- Turn off the robot while writing your program.
- Be careful when the robot moves, it can hurt you.
- Keep the ZedBoards and other equipment far from the robots.
- Review connections carefully, the robot can be damaged.
- If a servo gets hot, stop your program right away with Control+C.

We will now establish a connection between the ZedBoard and the board on the base of the robotic arm, hereafter referred to as the *robotic arm board*. Connect one end of a Pmod cable into the *JE* Pmod port on the ZedBoard, as shown in the figure below. Notice that the connector should be placed in such a way that the white cable is located on the top left of the port.

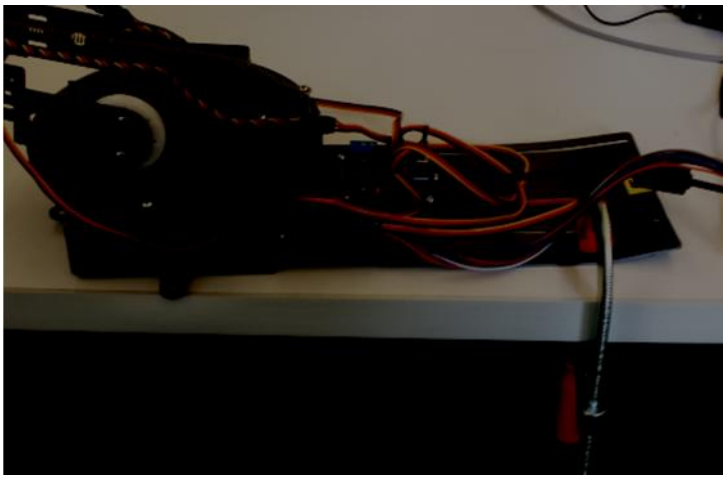


Validate the connections on the robotic arm board, making sure that the following associations between servos and ports apply:

Port	**Servo**
Servo 1	Base
Servo 2	Bicep
Servo 3	Elbow
Servo 4	Wrist
Servo 5	Gripper



Before moving on, make sure that your robotic arm is moved as far away from the ZedBoard and other lab equipment to prevent it from crashing. Still, it should be capable of moving without unplugging from the ZedBoard. Keep your hands and face away from the robot while you run your programs; it can move pretty violently sometimes. Secure the robot's base using a clamp available in the cabinet, as shown in the figure below. Finally, make sure that you turn off the robotic arm base through its switch while you're writing programs or manipulating connections.



Connect the power supply to the robotic arm board, using the power adapter shown on the left. You can get this adapter from the lab equipment boxes available near the robotic arms. Turn on the robotic arm board by flipping the switch on it, making sure that the adjacent power indicator LED is switched on. Careful: the robot may move now!

Automating the port setup

The ZedBoard loses its GPIO configuration on each reboot, going back to the default configuration regarding the direction of the ports. To quickly reconfigure the GPIO, we will create a shell script that automates this process. Remember that a shell script is a set of shell commands encapsulated in a plain-text file with execution permissions, which can be invoked directly from the shell, as if it was a regular binary executable file.

Create a directory named lab6 in your home folder, and place the following content in a file named gpio-init.sh within that directory:

```
FILEPATH='/sys/class/gpio'
echo 13 > $FILEPATH/unexport 2>/dev/null
echo 10 > $FILEPATH/unexport 2>/dev/null
echo 11 > $FILEPATH/unexport 2>/dev/null
echo 12 > $FILEPATH/unexport 2>/dev/null
echo 0 > $FILEPATH/unexport 2>/dev/null
echo 13 > $FILEPATH/export
echo 10 > $FILEPATH/export
echo 11 > $FILEPATH/export
echo 12 > $FILEPATH/export
echo 0 > $FILEPATH/export
echo out > $FILEPATH/gpio13/direction
echo out > $FILEPATH/gpio10/direction
echo out > $FILEPATH/gpio11/direction
echo out > $FILEPATH/gpio12/direction
echo out > $FILEPATH/gpio0/direction
```

You can assign execution permissions to the shell script with the following command:

```
$ chmod +x gpio-init.sh
```

And finally, you can invoke the script every time the ZedBoard reboots with the following command:

```
$ sudo ./gpio-init.sh
```

Assignment 1 (1 pt.)

Explain the code in script gpio-init.sh. You can provide a common description for each group of commands that share the same structure.

Controlling the servos

Now that the setup is complete, we are ready to start controlling the robotic arm. The following C++ program sends a control signal to Pmod port *JE1* during 8 seconds. The program is composed of three files: GPIO.h, GPIO.cc, and main.cc. The first two files implement a class named GPIO, whose constructor takes the identifier of a GPIO port as an argument. Function GPIO::GeneratePWM() allows the caller to generate a periodic PWM signal. The behavior of this function is documented with Doxygen comments in the header file below.

File GPIO.h

```
#ifndef GPIO_H
#define GPIO_H

class GPIO
{
	// File descriptor
	int fd;

public:

	/**
	 * Class constructor.
	 *
	 * @param number
	 *	Port number for GPIO.
	 */
	GPIO(int number);

	/**
	 * Class destructor.
	 */
	~GPIO();

	/**
	 * Generate a PWM signal, blocking the caller while the signal is being
	 * generated.
	 *
	 * @param period
	 *	PWM period in microseconds.
	 *
	 * @param pulse
	 *	Duration of the pulse in microseconds.
	 *
	 * @param num_periods
	 *	Number of periods to generate.
	 */
	void GeneratePWM(int period, int pulse, int num_periods);
};

#endif
```

File GPIO.cc

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream>
#include <stdlib.h>
#include "GPIO.h"


GPIO::GPIO(int number)
{
	// GPIO device files will follow the format
	//	/sys/class/gpio/gpio<NR>/value
	// <NR> has to be replaced with the actual device number passed as an
	// argument to the class constructor.
	char device_name[128];
	sprintf(device_name, "/sys/class/gpio/gpio%d/value", number);

	// Open special device file and store file descriptor in class member.
	fd = open(device_name, O_WRONLY);
	if (fd < 0)
	{
		std::cerr << "Cannot open " << device_name <<
				" - forgot sudo?  \n";
		exit(1);
	}
}

GPIO::~GPIO()
{
	// Close open file descriptor
	close(fd);
}

void GPIO::GeneratePWM(int period, int pulse, int num_periods)
{
	// Generate num_perios of the PWM signal
	for (int i = 0; i < num_periods; i++)
	{
		// Write ASCII character "1" to raise pin to 1, starting the
		// ON cycle, then wait duration of pulse.
		write(fd, "1", 1);
		usleep(pulse);

		// Write ASCII character "0" to lower pin to 0, starting the
		// OFF cycle, then wait the rest of the period time.
		write(fd, "0", 1);
		usleep(period - pulse);
```

```
        }
}
```

File main.cc

```
#include "GPIO.h"

int main()
{
	// Open device file 13 on Linux file system
	GPIO gpio(13);

	// Generate PWM signal with 20ms period and 1.5ms on time.
	// Generate 400 periods, this will take 20ms * 400 iterations = 8s
	gpio.GeneratePWM(20000, 1500, 400);

	// Done
	return 0;
}
```

Assignment 2 (2 pt.)

Write a Makefile with rules to compile main.cc, compile GPIO.cc, and link executable main. Build your program using shell command make, and run it. Your Makefile should include a rule clean that deleted any binary file created during the compilation process. Test the behavior of this rule by running make clean.

Download files GPIO.cc, GPIO.h, main.cc, and Makefile and pack them into a ZIP file named q2.zip. When this file is unpacked, the code should compile without errors by just running make. Attach this file to your lab submission.

Controlling the robotic arm position

We will now extend the functionality of the previous program to control the position of different servos of the robotic arm. Create a new program named ServoPosition.cc, which asks the user to enter the following information:

- Servo number between 1 and 5. The program should display the meaning of each number before having the user enter one.
- The position for the servo, given as an angle in degrees.

The program should verify that the user entered valid values. Then, it should bring the servo to the given position. If your PWM signal pulses are in the range [0.6ms, 2.4ms], make sure that you don't enter angles too close to 0° or 180°, to avoid exceeding the free motion range of the robot servos.

Assignment 3 (2 pt.)

Create a ZIP package named q3.zip containing files ServoPosition.cc, GPIO.h, GPIO.cc, and Makefile, and attach it to the lab submission on Blackboard. When the ZIP file is unpacked, your program should compile without errors by just running make.

Controlling the robotic arm speed

Servos can only be controlled through PWM signals that encode a target position. Given this information, a servo moves to the specified position at its fastest speed, regardless of its initial state. A technique used to produce slower motions consists in providing intermediate positions for the servo, guiding it progressively through its way to a final state. If the intermediate steps are enforced with some delay in between, the overall servo motion will seem to have slowed down.

Create a new main program named ServoSpeed.cc that allows the user to control the speed of the movements by asking for the following input information from the user:

- Servo number, given as a number between 1 and 5.
- Start position, given as an angle between 0 and 180.
- End position, given as an angle between 0 and 180.
- Rotational speed, given in degrees per second.

The program should control the servo to carry out the following actions:

- Stay in the start position for 1 second, by generating a PWM signal with constant duty cycle.
- Move from the start to the end position with the specified speed.
- Stay in the end position for 1 second, then exit.

Extend class GPIO with a new public member function with the following prototype:

```
void GPIO::GenerateVariablePWM(int period,
                               int first_pulse,
                               int last_pulse,
                               int num_periods);
```

In a similar way to function GPIO::GeneratePWM(), this function takes a period for a PWM signal as its first argument, given in microseconds, and a number of periods in its last argument. The product of period and num_periods is equal to the total duration of the PWM signal. The function will only return after this time has elapsed.

But as opposed to GPIO::GeneratePWM(), function GPIO::GenerateVariablePWM() now takes two pulse arguments, also given in microseconds. Argument first_pulse indicates the width of the first pulse, while last_pulse refers to the width of pulse num_periods – 1 (the last pulse). The width of every pulse in between should evolve linearly, taking intermediate values between first_pulse and last_pulse. Notice that this progression could be positive or negative.

Once you complete function GPIO::GenerateVariablePWM(), you are ready to write your main program. You will find that calculating the values for the first three arguments of the function is not much different from how you calculated the values for GPIO::GeneratePWM(). You may find it more challenging to calculate the value of the last argument of the function. Think of how to calculate the number of PWM periods from the input provided by the user.

Assignment 4 (3 pt.)

Write the main program that supports speed control for the servos and show it in action.

a) Create a ZIP file named q4.zip containing files GPIO.cc, GPIO.h, ServoSpeed.cc, and a Makefile. When unpacked, the program should compile without errors by just running command make.

b) Record a video with a maximum duration of 20 seconds where you demonstrate the behavior of the program by moving one servo of the robotic arm at two different speeds. Upload your video as part of your lab submission in a file named q4.mov.

Controlling multiple servos

In this optional assignment, you can reason about the challenge imposed by controlling multiple servos at once. Just in order to move two servos to different positions, and for now disregarding the speed, you need to produce two different PWM signals. Your previous implementation relies on the use of system call sleep(), which suspends the program for the amount of time specified in the argument. This is a problem if, during the time one PWM signal stays in the same state, you have the need to change the value of a different PWM signal. Thus, you need to find an alternative way to change values of PWM signals at the right time without involving calls to sleep(). Once you have come up with the right strategy, add support for speed control as well.

Extra credit (2 pt.)

Write a program named MultiSpeed.cc that collects information from the user related with the motion of two servos, including two servo numbers, two initial positions, two final positions, and two speed values. Once all information is entered, the program should start moving the two servos at the same time with the corresponding speeds.

a) Create a ZIP file named extra.zip containing files MultiSpeed.cc, GPIO.h, GPIO.cc, and Makefile. When unpacked, your code should compile without errors by just running make. Attach this file to your lab submission.

b) Record a video with a maximum duration of 20 seconds where you demonstrate the capability of your program to bring to servos to different target positions at different speeds. Save it in a file named extra.mov and attach it to your lab submission.