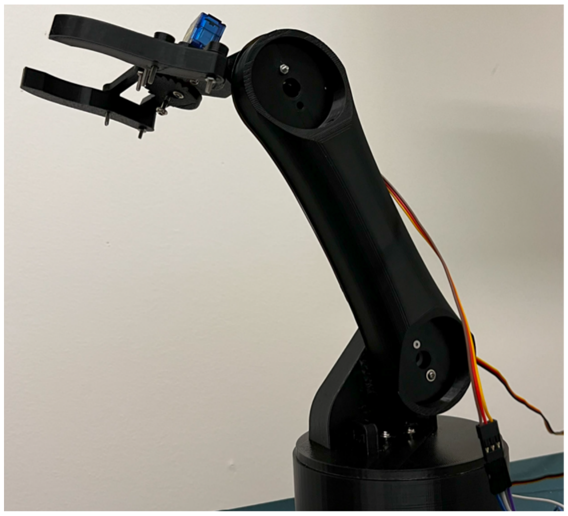
The Arm of the Future

ENG EC 535 Final Project
SPRING 2023
Professor Coskun
Boston University



Justin Sadler (jsad)
Abin Binoy George (abg309)
GitHub

2 May 2023

Abstract

This project aims to develop a system for controlling robotic arm servos directly from the Linux kernel. The primary focus will be on providing a high level of control over individual servo motors, allowing for precise movements of the robotic arm. The system will make use of a notification chain mechanism to receive input from a USB keyboard and control PWM signals. In addition to the kernel-level control, a user-space program was used for graphics on the LCD display. The resulting system will provide a powerful and flexible platform for controlling robotic arms from within the Linux operating system, with potential applications in areas such as industrial automation, research, and education.

1 Introduction

The goal of this project was to create and control a robot arm that can move anywhere in reach of the arm's workspace. Today, many laboratories run multiple experiments and tasks that are not only time-consuming but also extremely tedious. A robot arm can be used to automate this experimental procedure which in turn allows researchers to focus more of their attention on more trivial tasks. Generally, purchasing a robotic arm can be extremely expensive and difficult to set up. Our solution provides a much cheaper alternative and is also easier to set up.

Our solution includes a 2-degrees-of-freedom robot arm along with a gripper attachment at the end of the arm, all of which is 3D printed [3]. In order to control the arm, three servos are attached to the joints of the arm which can rotate allowing the robot arm to move to a different position and to close/open the gripper. To control the servos, a Pulse-Width Modulation (PWM) signal is used that can rotate the servo to the desired position.

Typically, robot arms are controlled either using code that the user will have to write themselves or on complicated software that is provided by the manufacturer. Our solution aims to alleviate this hardship on the user by developing a keyboard-based user input allowing the user to simply move the robot arm with literally a click of a button.

However, simply moving the robot arm using keyboard buttons will not help users automate a process. For this reason, we added additional features to our implementation that allows the user to add a set of specific locations as stages in their sequence and run the sequence. This will in turn make the robot arm move between each stage and loop back to the first one. This feature will ultimately allow the user to automate a certain process.

Finally, in order to easily tell the users the instructions or keyboard commands that can be used to control the robot arm, a simple GUI script was implemented that displays instructions on how to use and control the robot arm.

In addition to this, it was decided to run the controls side of this project, i.e. keyboard interrupts and servo control, purely from the kernel space. This way we can ensure that the robot arm will be able to move and react to keyboard commands as fast as possible.

While there was some chatter from the servo motors and some faults in the mechanical design, we were successfully able to control the robot arm using keyboard commands. In addition to that, we were also able to start a sequence of movements that are looped to simulate the arm automating a certain process.

2 Design Flow

In the kernel space, our project includes a keyboard notifier, a PWM controller, and our callback functions for sending PWM signals to the servos. On a key press, the user triggers an interrupt in the kernel module. Depending on which key was pressed, the callback function for the keyboard notifier would change the values of the duty cycles for the servos or store the current duty cycles in an array. When the user hits the ENTER key, the PWM controller will go through the cycle through stored duty cycles for each of the 3 servos.

Additionally, we used a Qt userspace program to display instructions on the LCD display. The graphics are static and unaffected by what happens in the kernel space.

The keyboard notifier was written by Justin Sadler. The PWM functions and QT code were written by Abin Geroge. Both team members collaborated on the PWM controller.

Project Contributions:

- Justin Sadler: 50%
- Abin George: 50%

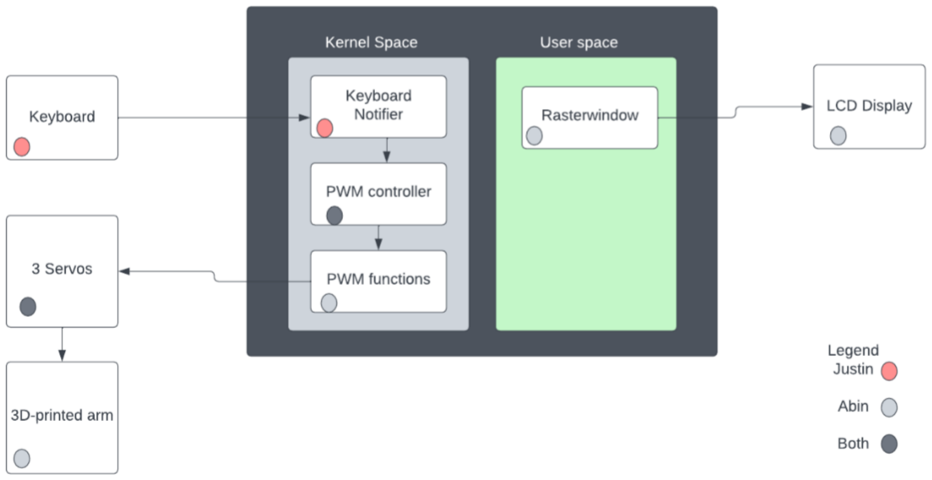


Figure 1: Design Flow

3 Project Details

3.1 Modules

3.1.1 Keyboard Interrupts

For our project, we wanted to use a keyboard device as the main interface between the user and the robotic arm. We decided on this because the keyboard is an intuitive interface that allows room for complex commands.

We implemented the keyboard interrupts with a notification chain, a mechanism to implement event-driven programming between subsystems. A notification chain is composed of a list of functions to execute when a given event occurs. Subsystems on the Linux kernel can register themselves in any notification chain created by any other subsystem1. This allows a single subsystem to observe an event and trigger callback functions in many other subsystems. In technical terms, the notifier is the subsystem that observes an event and calls the callback functions. The notified are the subsystems that ask to be notified about an event and provide the callback functions to invoke.

Several subsystems may be interested in what is pressed on the USB keyboard. To allow the subsystems to remain independent of each other, a notification chain is used so that key press notifications are received by every relevant subsystem.

The notification chain consists of a linked list of a struct of notifier blocks. Linux provides the struct definition in notifier.h.

```
#include <linux/notifier.h>
 struct notifier_block {
   notifier_fn_t notifier_call; // Pointer to the callback function
   struct notifier_block __rcu *next; // The next notifier block
   int priority; // The priority of the callback function
 };
```

The Linux kernel API provides two functions for easily registering and unregistering keyboard notifier blocks to the notification chain.

```
#include <linux/keyboard.h>

 int register_keyboard_notifier(struct notifier_block *nb);
 int unregister_keyboard_notifier(struct notifier_block *nb);
```

The entire code snippet for the keyboard notifier is as follows.

```
#include <linux/notifier.h>
#include <linux/module.h>
#include <linux/keyboard.h>

 static struct notifier_block nb {
   .notifer_call = keys_pressed
 };

 static int __init arm_init(void){
   register_keyboard_notifer(&nb);
 }

 static int arm_exit(void){
   unregister_keyboard_notifer(&nb);
 }
```

3.1.2 PWM

In order to control the servo via a PWM signal, two methods were used. One included the PWM driver library and the other with GPIOs and timers.

PWM Driver

To use the PWM driver the following library was used in the kernel module.

```c
  #include <linux/pwm.h>
```

With this module, we could configure a certain PWM port by setting the period of the PWM signal and the duty cycle of the signal. From the datasheet of the servo [2], we could determine the actual parameters required.

Period: 20 ms
Duty Cycle Range: 1 ms - 2 ms
Angle Range: 0-180 degrees

Using these system parameters, the PWM port was configured using the following code snippets.

```c
request_module("BB-PWM1-00A1");
pwm0 = pwm_request(0,"pwm-wrist");
pwm_config(pwm0, dutyCycle_time, pwm_period);
pwm_enable(pwm0);
```

This way we were able to change the duty cycle of the PWM signal, which ultimately moved the servo to a certain position. However, from our testing, it was determined that this module was not very consistent with setting the position of the servo. For this reason, we decided to try a different solution involving GPIOs and timers.

GPIOs and Timers

Alternatively, we tried a different method to control the servos using GPIOs and timers essentially trying to mimic a classical PWM signal. To do this, the following libraries were used.

```c
  #include <linux/gpio/driver.h>
  #include <linux/timer.h>
  #include <linux/jiffies.h>
```

This method essentially turns on the GPIO pin for duty cycle time and turns off the GPIO pin for the remaining time of the period. The timers were used to simulate the periodicity of a PWM signal, i.e. output the necessary signal every period (20ms). The following code snippet shows how the timer was implemented and how the GPIO output was set for the necessary duty cycle.

```c
// setting up the timer
timer_setup(timer, moveServo, 0);

// setting duty cycle
gpio_set_value(GPIO_NUM, 1);
udelay(dutyCycle_time);
gpio_set_value(GPIO_NUM, 0);
mod_timer(timer, jiffies + usec_to_jiffies(period-dutyCycle_time));
```

Using this method, we were able to move the servo to the appropriate position. However, adding the display module to the BeagleBone black caused the servo to move a little jittery due to the additional traffic between the BeagleBone and the display.

3.1.3 PWM Controller

The PWM controller is responsible for translating the keyboard presses into controlling the PWM signals.

When a user presses the up arrow key, down arrow key, left arrow key, right arrow key, the “g” key, or the “h” key, it increases or decreases the duty cycle of one of the three servos. To create a sequence of moves, the user can press “1”, “2”, “3” or “4” on the keyboard to store the current duty cycles into an array. When the user presses the “ENTER” key, the servo begins cycling through the array of duty cycles to move the robotic arm through a sequence of moves. The user can press the “ESC” key to stop the sequence and continue using the keyboard.

In addition to this, we also ensured that the movements were smooth by progressively increasing the PWM signal to the servo, rather than abruptly changing the PWM signal. By doing this, the servo will take steps to slowly move to the desired position, rather than moving to the final position in one step.

3.1.4 GUI

We used the Qt library along with the Rasterwindow class provided to us to simply display a string on the display screen. Then, we compiled the code and ran it in userspace.

3.2 Circuitry

Figure 2 shows the circuitry of our setup, all three servos are powered using 5V from the BeagleBone Black. Moreover, three GPIO pins are used to send PWM signals to control the respective servos. Finally, the keyboard was connected to the BeagleBone Black via a USB cable.

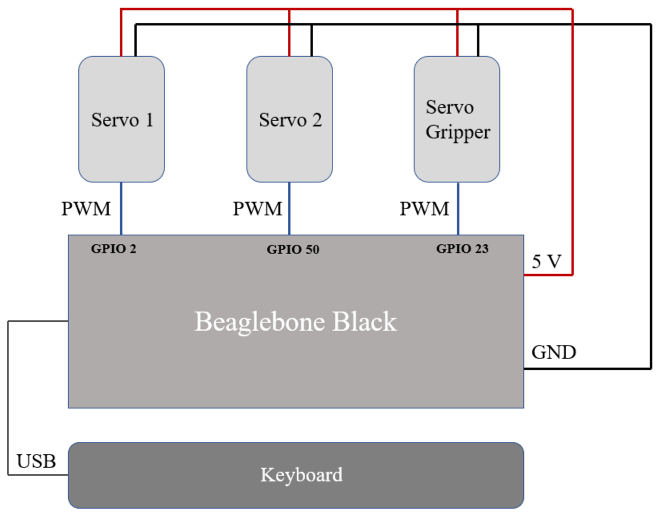


Figure 2: Circuitry

4 Summary

The main objective of this project was to control a robot arm and start a sequence of movements using a keyboard as an interface. In addition to this, we wanted to send PWM signals to the robotic arm servos from Linux kernel space. We were successfully able to accomplish this objective. We use the Linux notification chain to implement event-driven programming. Additionally, we used the Qt library in user space to develop a simple display.

The project used functions from the Qt library and the standard Linux Kernel library for version 4.19.

Given more time and resources, we would include the following features:

- A more interactive GUI to control the robot arm.
- Using the PWM library to control the servos from userspace. This was tested and worked flawlessly. However, it was a little late to incorporate it into our final code framework.
- Use better mechanical fasteners to keep the robot arm sturdy.

References

[1] Christian Benvenuti "Understanding Linux Network Internals, Chapter 4. Notification Chain", December 2005.

[2] Hitec, "Servo Datasheet".
https://media.digikey.com/pdf/Data%20Sheets/DFRobot%20PDFs/FIT0185_Web.pdf

[3] Kunal Bhagwat
https://thangs.com/designer/kunalbhagwat2202/3d-model/Robotic%20Arm%203D%20Model.stp-505558

[4] Linux Keylogger and Notification Chains.
https://0x00sec.org/t/linux-keylogger-and-notification-chains/4566/1