CMPE-240
Final Project

Henry Zimmerman

December 12, 2017

1 Abstract

The project as proposed set the objective of being able to unlock a door remotely using a Raspberry Pi, a servo, and an Android Wear SmartWatch. This objective was met using the specified items, while also utilizing technologies such as PWM to control the servo, speech-to-text to listen for commands, and broader networking technology to transmit messages to the server.

2 Design Methodology

Implementation of the project began with the webserver. Because it was known that controlling the servo was a time sensitive affair, Rust was chosen as the language for the server. Rust is new language developed within the last decade with the goal of supplanting C with something just as fast, but prone to fewer runtime errors. The design called for the webserver to talk to the servo controller, so the choice was made to keep both of those components in the same language and as part of the same program. Rust's lack of garbage collection, instead opting for a compile-time lifetime tracking system, inspired confidence that servo control would not be interrupted by GC pauses. Initially, Rust's promise of being able to run on bare metal was appealing, as being able to write the project without an underlying OS would be consistent with other labs in CMPE 240. The goal of running without an operating system did not come to fruition for reasons that are explained later.
Rocket, a webserver framework, was chosen to be used to write the server, primarily because of existing familiarity with the framework. It uses Rust's advanced macro and type systems to take very plainly written functions that represent API endpoints, and automatically converts them into other functions that will perform input validation, match expected request routes, and set up error handlers, manage server state, and other actions that usually require excessive boilerplate code. Because of how terse the webserver code was, it took very little time to get a simple webserver that would respond with HTTP 200 for a given request. In fact, it took longer to set up port forwarding and a domain then it did to get the initial server running. The webserver was tested by using Firefox's "Edit and Resend" feature in the developer tools to send POST requests to the domain under which the Pi was situated. HTTP 200 responses were observed in Firefox and `stdout` logs were observed over SSH on the Pi.
Unfortunately, in order for Rust to be able to run without an OS, every dependency in a binary or library needs to be marked with the `no_std` attribute, and Rocket, being a large library with dependencies of its own, did not have this marker. It was unlikely from the start that running on bare metal would be practical (getting networking working without an OS likely would have proved too difficult anyway), but this dependency issue made it clear that Raspbian would be required for this project.
With an initial webserver running, it was time to implement the servo controller. The first thing that had to be done was to wire up the servo to the Pi. For this project, the following items were purchased in addition to the existing Pi, SmartWatch, and breadboard:

1. Servo

2. Battery
3. 40 Pin Breakout Cable
4. Wires

The breakout cable was used to connect all 40 pins on the Raspberry Pi to the breadboard. Because the servo requires a full 5V to operate, and a pin on the Pi can only provide about 3.5V, an external battery pack was required to provide power. The servo has three wires to connect: Power, Ground and Signal. The battery pack was connected to the vertical power rails of the breadboard. The servo's power and ground were attached to the respective rails on the breadboard. The provided guide for the servo suggested connecting the Signal to pin 18 of the Raspberry Pi, as it suggested that only this pin could be controlled with enough precision to send meaningful signals to the servo. The guide was old and had been written for use with the Raspberry Pi 1 - Model B variant, which had fewer pins overall and it was unclear if later generations had pins that could send precise signals on all pins. Pin 16 of the Raspberry Pi 3 was chosen with the intension of trying other pins if pin 16 couldn't be driven with enough precision to control the servo.

With the servo wired to the Pi, software to control the servo was required next. The original proposal indicated that a library (called a 'crate' in Rust terminology) called `cupi` could be used to control the GPIO pins of the Pi. Unfortunately, it was found that the crate had not been maintained in 2 years and did not compile; likely due to changes to Raspbian in that time. Instead, another crate, `sysfs_gpio` was found to compile and control the GPIO pins.

`sysfs_gpio` works by passing a closure to a `Pin` instance that will spawn a separate thread to control the GPIO pin. The closure in question would contain code that would control the pin's state. This setup has the advantage of allowing code inside the closure to `panic!()`, a safer alternative to segfaulting, that would cause the thread in question to die without bringing down the rest of the server program.

The graceful failing was useful in tracking down a bug. When the closure executes, it would need to configure the pin before setting it high or low. If it couldn't, it would `panic!()`, causing the thread to crash. Apparently, the `Pin`'s function that takes the closure first tries to set the pin to allow configuration before executing the closure, but the `udev` system, which is responsible for device permissions on Linux, would take too long to set. This caused the closure to try to configure the pin before the OS had granted access to do so, causing the error. Because `expect()` could be called on a function that may return an error, an error message was able to be included that would indicate which function was responsible for this error. This problem was resolved by adding a 100 ms delay before the pin was configured, allowing udev to do its thing.

Once accessing GPIO pins ceased causing the program to crash, determining if controlling the servo was possible was the next task. Based on figure FIGURE, it was apparent that the servo would expect a signal pulse every 20 ms, and the servo would rotate to an angle that corresponds to the pulse width. This technique for controlling hardware in this manner is known as pulse width modulation, or PWM. The initial implementation of the function that would send the pulses would set the pin low, wait for 20 ms, set it high, and then wait for 1 or 2 ms, and this would repeat for about a second. Surprisingly, this worked on the first try, but the servo experienced some jitter where it would reach its expected angle, but would move 10 degrees in either direction once there. This was solved by setting the pin low, waiting for 20 ms *minus* the pulse duration, then setting it high, and waiting out the pulse duration. This led the pulse period to always be 20 ms, instead of 21 or 22 ms depending on the desired angle. Because the pulse period was closer to what the servo expected, the jitter cleared up and the servo worked as expected, rotating about 90 degrees, enough to move a door lock from the unlocked to locked position. The OS's context switching did not seem to interfere with the servo's signal generator, which indicated that the use of pin 16 was adequate for the purposes of this project. The time it took for the servo to rotate into place was eventually ascertained to be less than a second, so the 20 ms pulse was repeated 40 times to lead to closure taking about 800 ms to execute.

The pulse sending function was included with a Struct representing the servo, that held the `Pin` instance and an Enum that indicated which orientation the servo was in. A toggle function was provided that would move the servo between the locked and unlocked positions every time it was called. The servo struct was then initialized in the main body of the server, wrapped in a Mutex

to enforce synchronous access, and then managed by the Rocket server, so that it could be given to the endpoint function so it could be toggled when a request was received that matched the endpoint.
Once the server and servo were working as expected, the only remaining component was the watch client. The proposal assumed that a companion service was required to run on the phone to provide networking, as that scheme was required for Android Wear 1.X. It was discovered in the process of creating the app that Android Wear 2.0 supports sending network messages directly from the watch, which would then either use the watch's own WiFi connection, or proxy through the phone via bluetooth automatically. It was simple to set up an activity (Android term for a UI thread) that would send a request to the Pi when the app started. At this time, the servo was controllable via the watch, but it lacked the voice command specified in the proposal.
Setting up the voice command was obtuse. The app was originally called "Sesame" because Android Wear supports opening installed applications by saying "Ok Google. . . Open APPNAME". The chance at clever wordplay was the inspiration behind this project. Despite hours of experimentation, it was impossible convince Google's speech to text engine and internal app name lookup system to open an app named "Sesame" with voice commands. The name of the app used for lookup via Google's STT system was changed to "Lock", after which, the command "Open Lock" would reliably open the Sesame app, causing the POST request to be sent to the Pi.

3 Results and Analysis

With the required components completed: the webserver, servo wiring, servo controller logic, and Android Wear app, it was now possible to rotate a servo by remotely speaking a voice command. The server would toggle the servo, and log this information. A sample server log of Rocket starting and handling requests for toggling the servo is provided in figure FIGURE.
The initial observed shaking of the servo indicated something was wrong. There were concerns about trying to control the servo from a process running in an OS. It might not have worked because context switching performed by the OS might have caused the thread controlling the servo to pulse or rest for too long. This would have led the servo to jitter, as the pulses would instruct the servo to rotate to a slightly different angle. There were also concerns about Rust taking too long to execute causing inconsistencies in the servo signal, but this was disproved when no observable difference was detected when controlling the servo with an unoptimized build and a release build. The release build for Rust is typically much faster than the unoptimized build, so if the language was at fault, a difference in servo behavior would have been noticeable between builds. Ultimately, it was found that the delay used to keep the pin low was leading to the full period of the pulse being too long (20 ms + pulse width), causing the servo receive a signal it couldn't adequately translate to a rotation angle.
While Android Wear 2.0 has the ability to send network requests, it couldn't send requests to the local network, likely because it routes the requests through Google before sending them to the intended destination. This required setting up a domain for the domicile in which the Pi was located and port forwarding requests received on port 8001 to the Pi at the local address 192.168.1.74. Once this was done, the SmartWatch was able to send messages to the Pi.
Because the server had to be publicly accessible, and I couldn't reasonably do that on RIT's network, so it was not feasible to demonstrate this project in a class presentation. Instead, a video showing the Android Wear app being opened with voice commands, and the servo subsequently moving in response served as evidence that the project's acceptance criteria was met. This video will be included as a separate file submitted with this project.

4 Conclusion

While the primary goal of speaking a command to a watch would cause a servo to rotate in a manner that could unlock a door was met, a number of optional details laid out in the proposal were not met.
Due to time and material constraints, as well as a desire to get my security deposit back, it was decided that the servo would not be mounted to a door. This would likely have required some

glue, to attach the servo's head to the lock, something to affix the servo's body statically to the door, as well as something to hold the Raspberry Pi and the other related circuitry. It was too much effort for a requirement that was made optional in the proposal.
The Android Wear app was originally planned to be written in Kotlin, a JVM language that works on Android platforms. The decision to use Java instead was motivated by the simplicity of modifying some provided example code in Java instead of learning a whole new language just to send a message over the network using minimal available documentation.
The server was originally intended to run on bare metal, but it quickly became obvious that it would not work due to dependency issues within Rust's packaging ecosystem. The decision to use an OS instead thankfully did not impact the performance of the servo as it related to this project.
The library used for controlling GPIO pins that was mentioned in the proposal was found to be unsupported when development began. An alternative library was used and thus did not

5 Source

The relevant parts of source are made available here. On top of that, Rust has a wonderful documentation tool that generates a static website from provided source code. A zipped archive of the generated documentation is included in the submission. Additionally, the project itself will be hosted on Github LINK HERERRERERERERR for at least the duration of the grading period and will not be modified after the due date.

5.1 Server

Listing 1: src/main.rs

```rust
#![feature(plugin)]
#![plugin(rocket_codegen)]
#![feature(const_fn)]
#![feature(duration_from_micros)]

// Bring in external crates (libraries).
// Only the main.rs file is responsible for bringing crates into scope.
extern crate sysfs_gpio;
#[macro_use]
extern crate log;
extern crate simplelog;
extern crate rocket;
extern crate rocket_contrib;

// Import elements from the crates.
use simplelog::{Config, TermLogger, WriteLogger, CombinedLogger,
    LogLevelFilter};
use std::sync::Mutex;
use std::fs::File;
use rocket::State;

// Use the servo file
mod servo;
use servo::Servo;

/// API endpoint that Rocket will route POST requests with empty bodies to.
/// This will toggle the servo's position.
#[post("/")]
fn toggle_servo_endpoint(servo: State<Mutex<Servo>>) {
    info!("Got message");
    // Lock the mutex and toggle the servo's state
    servo.lock().unwrap().toggle();
    // The lock will be dropped at the end of this function by a RAII destructor.
}

/// The pin used to drive the signal to the physical servo motor.
const SERVO_PIN_NUMBER: u64 = 16;

/// Main function.
fn main() {
    // Initialize the Servo, and protect it from synchronous access with a Mutex.
    let servo_position = Mutex::new( Servo::new(SERVO_PIN_NUMBER) );

    // Set up logging
    const LOGFILE_NAME: &'static str = "servo.log";
    CombinedLogger::init(
        vec![
            TermLogger::new(LogLevelFilter::Info, Config::default()).unwrap(),
            WriteLogger::new(LogLevelFilter::Trace, Config::default(), File::create(LOGFILE_NAME).unwrap()),
        ]
    ).unwrap();

    // Start the server
    rocket::ignite()
        .manage(servo_position)
        .mount("/", routes![toggle_servo_endpoint])
        .launch();
}
```

Listing 2: src/servo.rs

```rust
use sysfs_gpio::{Direction, Pin};
use std::time::Duration;
use std::thread::sleep;

const UNLOCK_PULSE_WIDTH_MICROS: u64 = 2000; // keep pin high for 2 ms
const LOCK_PULSE_WIDTH_MICROS: u64 = 1000; // keep pin high for 1 ms

/// Holds information on the current rotational state of the servo.
#[derive(Clone, Debug)]
enum ServoState {
    Locked,
    Unlocked
}

/// Wrapper around the Servo's state and the pin used to send the signal to the servo.
#[derive(Clone, Debug)]
pub struct Servo {
    state: ServoState, // The current rotational state of the servo
    signal_pin: Pin // Pin controlling GPIO for servo
}

impl Servo {

    /// Constructs a new Servo instance with a given pin number used to drive the
    /// signal to the physical servo.
    ///
    /// Assume that the servo starts in a locked position.
    /// If it isn't, the first action to toggle the servo will have no effect,
    /// but after that, the servo state will reflect the state of the real servo.
    pub fn new(pin_number: u64) -> Servo {
        Servo {
            state: ServoState::Locked,
            signal_pin: Pin::new(pin_number)
        }
    }

    /// Depending on the current rotational state of the servo, move the servo into the other state.
    pub fn toggle(&mut self) {
        info!("Toggling servo state:");
        // Set the state to the new servo state.
        self.state = match self.state {
            ServoState::Locked => {
                self.unlock();
                ServoState::Unlocked
            },
            ServoState::Unlocked => {
                self.lock();
                ServoState::Locked
            }
        }
    }

    /// Move the servo into the "locked" position.
    fn lock(&self) {
        info!("Locking");
        self.send_pulses(Duration::from_micros(LOCK_PULSE_WIDTH_MICROS));
        info!("Servo now in locked position");
    }

    /// Move the servo into the "unlocked" position.
    fn unlock(&self) {
        info!("Unlocking");
        self.send_pulses( Duration::from_micros(UNLOCK_PULSE_WIDTH_MICROS));
        info!("Servo now in unlocked position");
    }


    /// The Servo expects a signal every 20 ms.
    /// The signal shall go high for the duration of pulse_width parameter.
    /// Depending on how long the pulse width is (usually between 1-2 ms),
    /// the servo will rotate to a given angle.
    ///
    /// Once signals stop, the servo will remain in its last position.
    fn send_pulses(&self, pulse_width: Duration) {
        let pulse_pin = self.signal_pin.clone();
        pulse_pin.with_exported(|| {
            // udev is apparently awful, and takes a while to set the permissions of the pin.
            // If this delay isn't present, there is the possibility that the pulse pin will fail to
            // be enabled, and will crash the thread responsible for sending the signals.
            sleep(Duration::from_millis(100));
            pulse_pin.set_direction(Direction::Low).expect("Couldn't set the
                    direction of the pin");

            // loop until the servo has had a chance to get into position
            for _ in 0..40 {
                pulse_pin.set_value(0).expect("Couldn't set pin to low");
                // stay low for 20 ms - the width of the pulse
                sleep(Duration::from_millis(20) - pulse_width);
                pulse_pin.set_value(1).expect("Couldn't set pin to high");
                sleep(pulse_width); // stay high for the provided pulse
                    width
            }
            Ok(())
        }).unwrap();
    }

}
```

Listing 3: Cargo.toml

```
[package]
name = "sesame"
version = "0.1.0"
authors = ["Henry Zimmerman <zimhen7@gmail.com>"]

[dependencies]
# Webserver
rocket = "0.3.3"
rocket_codegen = "0.3.3"
rocket_contrib = "0.3.3"

# Json serialization, not used currently.
serde = "1.0.14"
serde_json = "1.0.3"
serde_derive = "1.0.14"

# GPIO control
sysfs_gpio = "0.5"
# Logging
log = "0.3.8"
simplelog = "^0.4.2"
```

Listing 4: Rocket.toml

```
[development]
address = "localhost"
port = 8001
workers = 2
log = "normal"
limits = { forms = 32768 }

[staging]
address = "0.0.0.0"
port = 8001
workers = 2
log = "critical"
limits = { forms = 32768 }

[production]
address = "0.0.0.0"
port = 8001
workers = 2
```

```
log = "normal"
limits = { forms = 32768 }
```

5.2 Android Wear

Listing 5: android/Sesame2/app/src/main/java/com/example/hzimmerman/sesame/ToggleServoActivity.java

```
package com.example.hzimmerman.sesame;

import android.net.ConnectivityManager;
import android.net.Network;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.support.wearable.activity.WearableActivity;
import android.widget.TextView;

import com.android.volley.Request;
import com.android.volley.RequestQueue;
import com.android.volley.Response;
import com.android.volley.VolleyError;
import com.android.volley.toolbox.StringRequest;
import com.android.volley.toolbox.Volley;


public class ToggleServoActivity extends WearableActivity {

    // Textview to show what is going on
    private TextView mTextView;

    // When the Activity is created, send a POST to the raspberry pi.
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mTextView = (TextView) findViewById(R.id.text); // set up the text
            view

        // Instantiate the RequestQueue.
        RequestQueue queue = Volley.newRequestQueue(this);
        // armbar-abode.mooo.com is the domain name of my domicile.
        // The request is port forwarded to the raspberry pi.
        String url ="http://armbar-abode.mooo.com:8001/";

        // Request a response from the provided URL.
        StringRequest stringRequest = new StringRequest(Request.Method.POST,
             url,
                new Response.Listener<String>() {
                    @Override
                    public void onResponse(String response) {
                        // Indicate with the text on screen that the lock
                            was toggled
                        mTextView.setText("Toggled the lock.");
                    }
                }, new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                // Indicate that the watch failed to tell the server to
                    toggle
                mTextView.setText("Failed to send message" + error);
            }
        });
        // Add the request to the RequestQueue.
        queue.add(stringRequest);

    }

}
```

Listing 6: android/Sesame2/app/src/main/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.hzimmerman.sesame">

    <uses-feature android:name="android.hardware.type.watch" />

    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@android:style/Theme.DeviceDefault">
        <uses-library
            android:name="com.google.android.wearable"
            android:required="true" />

        <meta-data
            android:name="com.google.android.wearable.standalone"
            android:value="true" />

        <!--
            This android:label="Lock" makes the app open when a user says
            "Ok Google ... Open Lock"
        -->
        <activity
            android:name="ToggleServoActivity"
            android:label="Lock">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```