Secure Software Project: Safe Programming in Rust

Stijn Volckaert
imec-DistriNet, KU Leuven
stijn.volckaert@cs.kuleuven.be

1 Introduction

The goal of this project is to write a simple driver and a client application in Rust. The application should parse an image file (encoded in P6-PPM format) from the file system and then use the driver to show the image as a scrolling banner on an adafruit LED matrix. You can base your Rust implementation on the C sample I provided. I will also provide a (small) part of the Rust code. Note, however, that the C implementation is incomplete. It only implements the base functionality. For this project, there are some additional requirements beyond implementing this base functionality. Please refer to the course slides for additional information on the precise requirements, and on how this assignment will be graded.

1.1 Getting Started

To get started, we will need to set up some hardware. I will provide one full set of hardware components per project group. However, if you want to purchase an additional set yourself, these are the items you will need:

- A Raspberry PI 3 development board
- A 16Gb microSD card with a fresh installation of NOOBS. NOOBS is a Linux distribution for the Raspberry PI. It can be downloaded at `https://www.raspberrypi.org/downloads/noobs`
- An adafruit RGB Matrix Bonnet
- An adafruit 16x32 RGB LED Matrix
- A 5V power supply
- An RGB 2x8 IDC data cable. This is a flat gray cable with one pink wire.
- An RGB power cable. This is a cable with two or four thick black and red wires.

1.2 Hardware Setup

To set up your board, please refer to the `adafruit.pdf` document, pages 11-18. Note that you are **not** supposed to solder any connections. You might, however, have to strip or recut the RGB power cable to get it to fit in the bonnet's terminal block.

1.3 Software Setup

After setting up the board, you will have to configure the OS and install some software. Start by connecting your raspberry pi to an HDMI screen and a keyboard/mouse. Depending on the operation system version that was preinstalled on your microSD card, the default user credentials are either `root/root` or `pi/raspberry`. After logging in, you should connect the board to a local network (either through an ethernet cable or through WiFi) and install an OpenSSH server. To install the OpenSSH server, open a terminal and type: `sudo apt update; sudo apt install openssh-server`

Next, you should download adafruit's demo software to check if your LED matrix works. Follow the Step 6 instructions in `adafruit.pdf`, pages 18-21. Be sure to compile and try out the demos. The demo programs may come in handy later because they reset the state of the LED matrix when you shut them down. Once you're done with the tests, you're ready to install Rust. You can do this by entering the following command into a terminal: `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`

2 System Overview

The LED matrix should now be connected to the Raspberry PI board via its GPIO header. The GPIO header has 40 pins in total, of which 28 are data pins. Some of these data pins are input pins, while the others are output pins. The programmer can control the peripheral devices connected to the GPIO header by communicating with the GPIO controller. The GPIO controller can read the voltage levels on the input pins, or set the voltage level on output pins according to the programmer's instructions. As with any piece of digital circuitry, there are only two meaningful voltage levels. The voltage level on a pin can either be set to 0V (low) or 5V (high). 1 bit of data therefore suffices to store the current voltage level on any given pin.

2.1 Controlling GPIO Peripherals

Several mechanisms exist to communicate with the GPIO controller. In this project, we will use a mechanism called "memory-mapped I/O" (or MMIO for short). Although the internal workings of MMIO are quite complex, its usage is pretty straightforward as the operating system handles almost everything for us. When we connect an MMIO-capable device to our system, the OS will set up the memory controller such that certain memory regions are shared between the main CPU (i.e., the Broadcom BCM2709 CPU on our Raspberry PI 3) and the MMIO-capable device (i.e., the GPIO controller which is actually baked on to the same die as our CPU). Whenever we read from or write to an MMIO memory region, the memory controller will forward all of our reads and writes to the associated device.

Taking our Raspberry PI's GPIO controller as an example, we find that the OS automatically configures the board's memory controller such that the GPIO's "register file" is mapped at physical memory address 0x3F200000. Whenever we read from or write to a register in this register file (by reading from or writing to memory address 0x3F200000 or one of the subsequent addresses), the memory controller forwards our read/write request to the GPIO controller. The GPIO controller has a lot of registers in its register file. Some of the registers we will use are:

- At offsets 0x0, 0x4, 0x8, 0xC, 0x10, and 0x14, we find the six 32-bit **Function Select** registers. We can write to these registers to configure the GPIO header's pins. This is necessary because the GPIO controller has to know which of its pins should be used as data input pins, data output pins, power pins, etc. If we configure the pins correctly, the GPIO controller will also ensure that we cannot perform dangerous actions such as raising the voltage on an input pin.

  The configuration parameters for every data pin consist of three bits. Setting these bits to 000 instructs the GPIO controller that the data pin is an input pin. Setting the bits to 001 instructs the controller that the data pin is an output pin. The upper two bits can be used to configure "alternative functions", which we will not be using in this project.

  To configure a data pin, we must first find the function select register that controls it, and then find the three configuration bits within that register. We can calculate the mapping using the following formulas (n is the data pin number):

  $register\_number(n) = \frac{n}{10}$
  $bit\_range(n) = [(n\%10) * 3...(n\%10) * 3 + 2]$

Example: By applying these formulas, we find that bits [0...2] within function select register 0 control the configuration for data pin 0. Bits [21...23] in function select register 1 control the configuration for data pin 17.

Since the register file is mapped at address 0x3F200000, and function select register 0 is located at offset 0 within the register file, this means that we can configure GPIO pin 0 as an input pin by writing the three bits 000 to memory address 0x3F200000.

- At offsets 0x1C and 0x20, we find two 32-bit **Pin Output** registers. By setting a bit in a pin output register to 1, we can instruct the GPIO controller to raise the associated output data pin's voltage level to high. To calculate the pin output register number and bit number for a given data pin, we apply the following formulas:

  $register\_number(n) = \frac{n}{32}$
  $bit\_number(n) = n\%32$

  Note that, because the highest pin number we will use is 27, we will only need to access register 0.

  Example: By applying these formulas, we find that bit 2 in pin output register 0 controls the voltage level for output data pin 2, whereas pin 3 in register 1 controls the voltage level for output data pin 35.

- At offsets 0x28 and 0x2C, we find two 32-bit **Pin Output Clear** registers. By setting a bit in a pin output clear register to 1, we instruct the GPIO controller to clear the currently active output on the associated output data pin (i.e., by lowering the voltage level on that pin to low). To calculate the pin output clear register and bit number for a given data pin, we apply the following formulas:

  $register\_number(n) = \frac{n}{32}$
  $bit\_number(n) = n\%32$

- At offsets 0x34 and 0x38, we find two 32-bit **Pin Level** registers. We can read these registers to determine the current voltage level of a pin. To calculate the pin output clear register and bit number for a given data pin, we apply the following formulas:

  $register\_number(n) = \frac{n}{32}$
  $bit\_number(n) = n\%32$

  Example: To read the current voltage level of data pin 6, we should read the value of bit 6 in pin level register 0. We can do so as follows (in pseudo C code):

```c
#define GPIO_REGISTER_FILE 0x3F200000
uint32_t pin_level_reg = *(uint32_t*)(GPIO_REGISTER_FILE + 0x34);
// use bitwise AND operation to select only bit 6
uint32_t pin_6_value = pin_level_reg & (1 << 6);
// now we shift this bit to the right so we can store it in a char
char pin_6_value_shifted = pin_6_value >> 6;
// pin_6_value_shifted now has value 0 if data pin 6's value is 0,
// or value 1 if the pin's value is 1
```

To gain access to these registers, we need to map the register file into our driver's address space. We can do this by using the `mmap` function to map the `/dev/mem` file into our address space. The Rust skeleton code contains a sample implementation of a function that maps this file.

2.2 Controlling the LED Matrix

Now that we know how to configure the GPIO controller, and how to set, clear, and read the voltage values of individual GPIO pins, we can take a closer look at the Adafruit RGB LED matrix, and how it can be controlled using GPIO pins. The LED matrices we will be using have 16 rows and 32 columns of LEDs. This means there are 512 RGB LEDs in total. Every LED has three sub-LEDs (an R LED, a G LED, and a B LED). We can switch each of these sub-LEDs on or off by writing to the GPIO pins.

Note, however, that there is no direct mechanism to control the intensity of an LED. This has two consequences. First, this means that we cannot set the brightness of the LEDs. If we take a snapshot of the LED board at any given time, we will see that every LED is either fully bright, or it is fully dimmed. Second, the matrix only supports eight colors out of the box (RGB colors #000000, #FF0000, #00FF00, #0000FF, #FFFF00, #FF00FF, #00FFFF, and #FFFFFF). To show color #FF0000, for example, we set the R led to 1, and the G and B LEDs to 0. To display other colors, or to control the brightness of the LEDs, we will have to use a technique called "pulse width modulation" (see Section 2.3), which you have probably seen in digital electronics courses.

Another peculiar property of the Adafruit RGB matrix is that the LEDs do not have individual addresses. Instead, the RGB matrix only assigns addresses to rows. To set the RGB values of a pixel, we need to "activate" a row, and then set the RGB values of that **entire row**. Moreover, this particular LED matrix is actually organized into two sub-panels. The top eight rows of the matrix belong to sub-panel 0 and they have row addresses 0 to 7. The bottom eight rows belong to sub-panel 1 and they **also have row addresses** 0 **to** 7. Because of this design decision, we always have to set the RGB values of **an entire double-row** at a time (a double-row is a set of two rows with the same row address). Luckily, this does not mean that the top and bottom rows in a double-row show the same colors, as the matrix has separate sets of pins to control the R, G, and B sub-LEDs of both rows in the double-row.

The RGB matrix uses the following GPIO pins. An explanation of how to use these pins follows in Section 2.2.1. The mapping of the RGB pins to GPIO pin numbers can be found in Appendix A.

- The **output enable (OE)** pin activates or deactivates the LEDs of the entire panel. This pin is somewhat unusual since writing a 1 to the memory location that controls this pin will actually make the LED matrix **disable** the LED pins, while writing a 0 **enables** the LED pins.

- The **clock (CLK)** pin is used to signal the matrix controller when now data has arrived on the color pins (see below). The CLK pin is set to 0 by default, but can be set to 1 right after pushing new color data through the color pins. Once we have pushed new color data, and we have "raised" the clock by setting the CLK pin to 1, we should immediately set it back to 0.

- The **strobe/latch (LAT)** pin is used to signal the matrix controller when we have finished pushing color data for a row. After pushing color data for an entire double-row, we should set the LAT pin to 1, and then immediately back to 0.

- The **address** pins are used to push the address of a double-row. The RGB matrix has five address pins (A,B,C,D,E) in total, but only uses the first three (A,B,C). The A pin represents the least significant bit of the double-row address. The B pin represents the second least significant bit, and the C pin represents the third least significant bit.

  Therefore, to push the address of double-row 5 (which can be written as 101 in binary) to the matrix, we should activate A and C pins. Similarly, to push the address of double-row 6, we should activate the B and C pins.

- The **color** pins are used to push the color bits of the current column in the current double-row. There are six color pins in total. Color pins R1, G1, and B1 control the R, B, and B sub-LEDs of the current LED in the top half of the current double-row. Similarly, pins R2, G2, and B2 control the sub-LEDs of the current LED in the bottom half of the double-row.

2.2.1 Pushing RGB Data for a Single Double-row

Before we can activate the LEDs in a double-row, we need to push the color values of the that row. This means we need to push 6 bits (i.e., the R, G, and B bits of the top and double-row) of color data for every column of the double-row. In other words, we need to push $6 * 32 = 192$ bits of data at a time. Since we can only push 6 bits of color per clock cycle, we are going to need 32 clock cycles for every double-row. The LED matrix' controller expects us to push this color data by "clocking in" color data using pins R1, G1, B1, R2, G2, B2. We also need to generate a clock signal on the CLK pin.

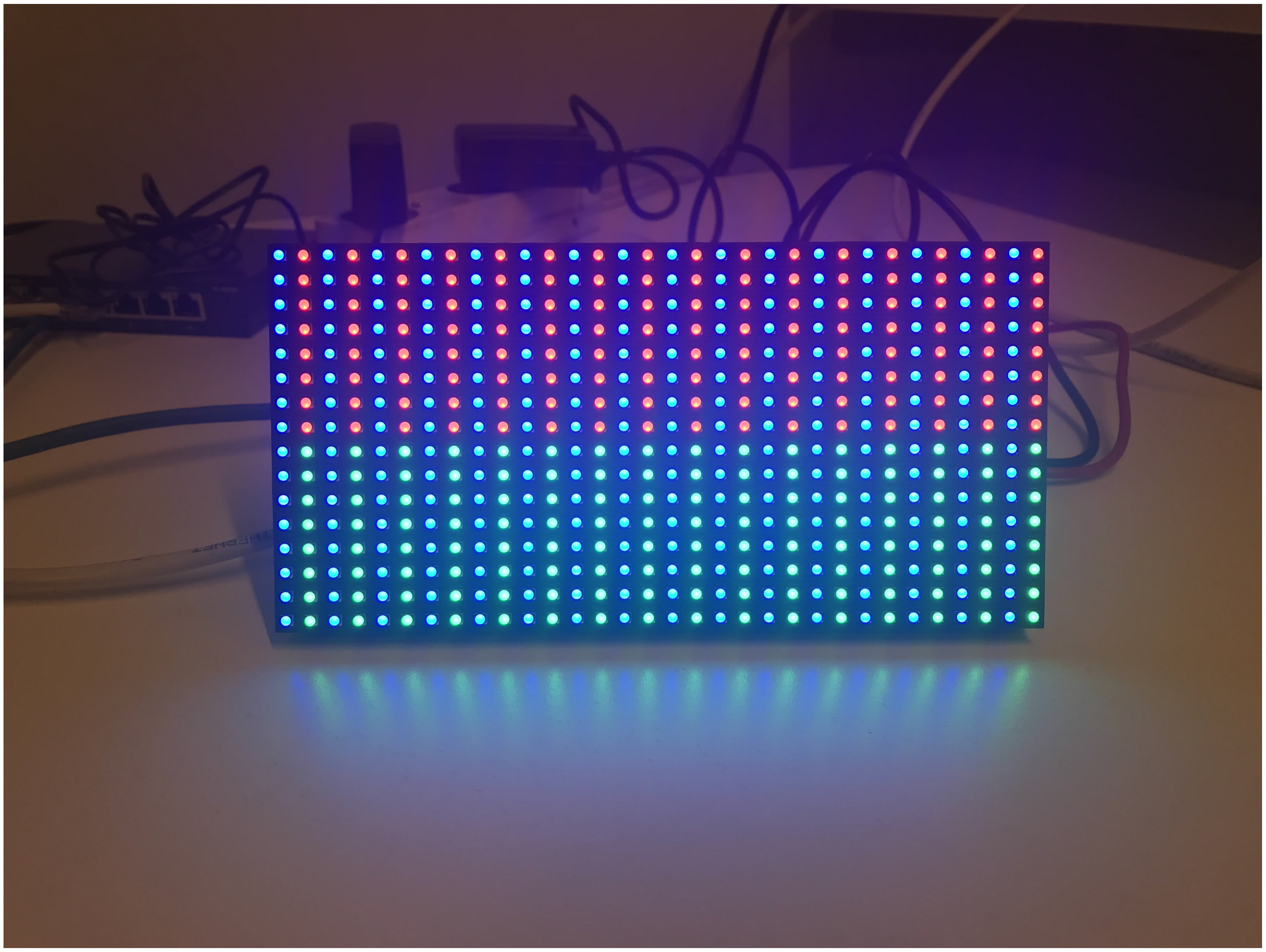

Figure 1: A simple color pattern on the adafruit LED matrix

Suppose, for example, that we want to show the pattern from Figure 1 on our LED board. In this pattern, we have eight identical double-rows. In every double-row, the LEDs in the odd-numbered columns are bright red in the top half of the double-row, and bright green in the bottom half. The LEDs in the even-numbered columns are bright blue in all rows. To push the color data for a double-row, we have to make sure that the voltage levels for the R1 pin (which controls the red sub-LEDs in the top half of the double-row) and the G2 pin (which controls the green sub-LEDs in the bottom half of the double-row) are set to high in all odd clock cycles, and that the voltage levels for the B1 and B2 pins are set to high in all even clock cycles. The resulting signal will look like the one in Figure 2.

The following pseudo-code shows how we could generate the necessary signal:

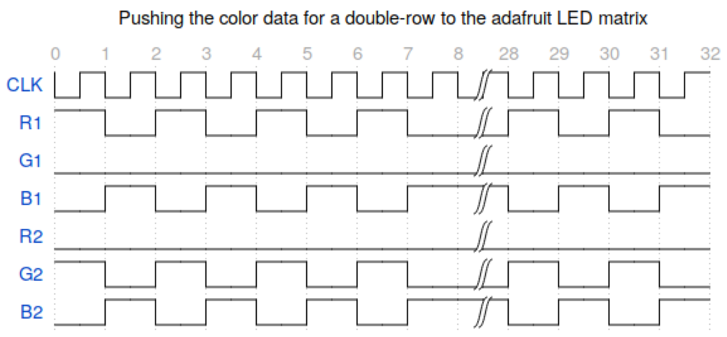
Pushing the color data for a double-row to the adafruit LED matrix
0
1
2
3
4
5
6
7
8
28
29
30
31
32
CLK
R1
G1
B1
R2
G2
B2

Figure 2: The digital signal we must generate to push the color data for the pattern shown in Figure 1

```
// the GPIO object is our interface to the GPIO controller
fn PushColorData(GPIO io) {

  for (int c = 0; c < columns; ++c) {
    if (c % 2 == 1) {
      // this is an odd-numbered column
      io.ClearAllPinsAndActivate(R1 | G2);
    } else {
      // this is an even-numbered column
      io.ClearAllPinsAndActivate(B1 | B2);
    }

    // With the voltage levels on the color pins still set to the
    // appropriate levels, we now raise the voltage level on the
    // CLK pin to finish this clock cycle.
    io.ActivatePins(CLK);
  }
}
```

Now that we've clocked in the necessary color data, we need to "latch in" the row want to activate by pushing the row address to the matrix, and then raise the voltage level on the **Output Enable (OE)** pin. The following pseudo-code shows how to do this:

```
// We're done pushing color data. We can clear the CLK pin and all of the
// color pins now
io.ClearAllPins(R1 | G1 | B1 | R2 | G2 | B2 | CLK);

// Next, we need to push the address of the double-row. Note that we have to
// do this AFTER pushing the color data for that row.
// Here we are activating address pins A and C, which means that the double-row
// we're setting the colors for has address 5
io.ClearAllPinsAndActivate(A | C);

// Now, we need to tell the matrix we've pushed a row address. We do this
// using the latch pin.
io.ActivatePins(LAT);
// We need to immediately disable the LAT pin after raising it
io.ClearPins(LAT);

// Now, we can activate the LEDs. Keep in mind that clearing the OE signal
// actually ACTIVATES the LEDs, while raising the OE signal DEACTIVATES them!
io.ClearPins(OE);
// Leave the LEDs on for 150 nanoseconds
nanosleep(150);
io.ActivatePins(OE);
```

2.3 Controlling Color Brightness and Intensity

The pseudo-code we have seen so far is sufficient to control the RGB values of every individual LED. However, we can only set the R, G, and B voltages to 0 or 5V. The 0V level corresponds with the lowest possible brightness. The 5V level corresponds with the fully bright color. If we want any color values in between, we need to use a technique called Pulse Width Modulation (PWM). One simple way to control brightness values would be to modulate the pulse width of the output enable signal.

The basic concept is pretty simple. We want to render images to the LED matrix at a fixed frame rate. We will refer to the duration of one frame as the **period (T)**. During each frame, we want to push fresh color and address data to the LED matrix, and then send pulses on the output enable pin. The brightness of the colors we will be rendering depends on the **duty cycle (D)** of the output enable signal. The duty cycle is the fraction of each period during which the output enable signal is enabled. The time during which the output enable signal is enabled is commonly referred to as the **pulse width (PW)**. The formula for the duty cycle D is therefore: $D = \frac{PW}{T} * 100\%$

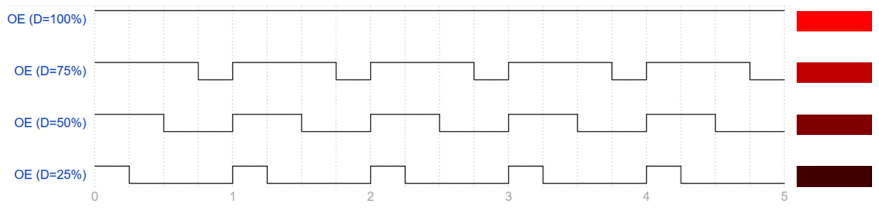

Figure 3: Pulse width modulation of the output enable (OE) signal. Every wave shows a different duty cycle (D) for the OE signal. The brightness of the color our LEDs show depends on the duty cycle. With higher duty cycles, the colors become brighter. To illustrate this effect, the shade of red corresponding with each duty cycle is shown next to the signal.

Figure 3 illustrates output enable signals with duty cycles of 50% and 75% respectively. Using this technique is pretty straightforward if we only have one color (e.g., red) whose brightness we want to control, and if in every row, all LEDs show the same color. If we keep the R1 pin voltage level at 5V at all times, then we can do PWM on the OE pin to control the brightness of the red color we want to show. With an output enable signal duty cycle of 50%, the color red would have a brightness level of 50%, whereas with a duty cycle of 75%, the color would have a brightness level of 75%, etc.

Our rendering algorithm now becomes:

```
fn RenderFrame() {
  for r in 0..doublerows {
    // Clock in the color data for all of the LED columns in this double-row
    io.PushColorData(r);

    // Send the address of this double row
    io.LatchAddress(r);

    // Do pulse width modulation on the OE pin
    io.PulsePin(OE);
  }
}
```

2.4 Rendering Bit Planes

Now we're getting closer to a full solution. There's just one problem left. The pseudo-code shown at the end of the previous section controls the brightness of an entire double-row at once. This is fine if all of the colors in the double-row are supposed to have the same intensity (brightness level), but this is seldom the case! The trick to controlling the intensity of each individual pin is to divide our image into **bit planes**, and to render the image bit plane by bit plane.

To understand how this works, we must look at what these bit planes are first. A bit plane of an image is the set of bits corresponding to a given bit position in each of the binary numbers representing an image pixel. If we represent our image by 1 R, 1 G, and 1 B byte for every pixel, then our image would have 8 bit planes. Each bit plane would contain 1 R, 1 G, and 1 B bit for each pixel. The 1st bit plane of our image would contain the most significant bit of the R,G, and B bytes of every pixel. The 2nd bit plane would contain the second most significant bit of the R,G,B bytes. An example of an image and its decomposition into bit planes is shown in Figure 4. A useful property of bit planes is that, the lower the number of the bit plane, the more that bit plane contributes to the overall look of the image. In fact, there is a formula to calculate exactly how much each bit plane contributes to the overall image (n is the number of the bit plane): $contribution(n) = \frac{1}{2^n} * 100\%$

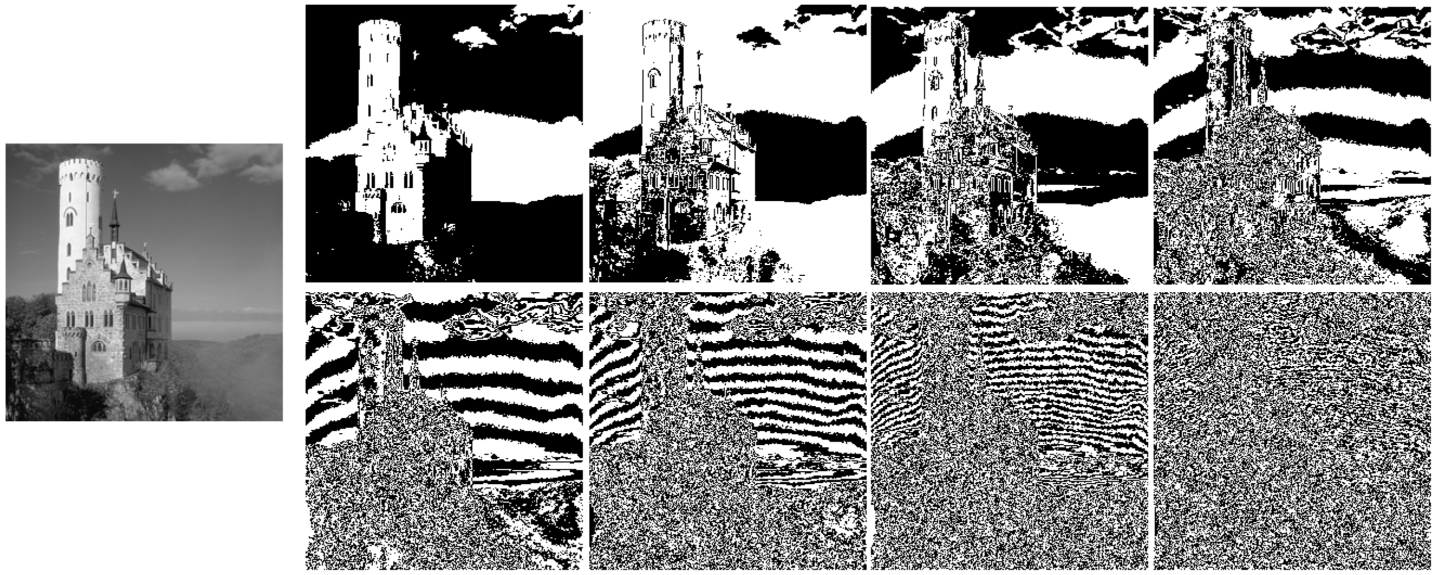
Figure 4: A grayscale image (shown on the left) and its decomposition into eight bitplanes. The 1st bitplane is shown in the top left position. The 8th bitplane is shown in the bottom right position. This image was created by Alessio Damato and is licensed under the CC BY-SA 3.0 license. See https://commons.wikimedia.org/w/index.php?curid=2201771 for details.

By applying this formula, we see that bit plane 1 contributes 50% of the overall look of the image. Bit plane 2 contributes 25%, etc. Now how do you translate this property into a working rendering algorithm? It is actually much simpler than you would expect! We just write our rendering algorithm such that bit plane 1 is shown for 50% of the time, bit plane 2 is shown for 25% of the time, etc.

We can do this by dividing each frame we render into sub-frames (one sub-frame for each bit plane) and by rendering the sub-frames in reverse order. We start with the 8th bit plane by rendering the colors for that frame and by activating the LEDs for a really short time (e.g., for 150 nanoseconds). For each subsequent sub-frame we render, we use the colors of the corresponding bit plane, and we double the time the LEDs are active. This means the LEDs would be active for 300 ns for the 7th bit plane, 600 ns for the 6th, etc. For the 1st bit plane, we would activate the LEDs for 19200 nanoseconds.

Now we finally have a full rendering algorithm:

```
fn RenderFrame() {
  for b in bitplanes..0 {
    for r in 0..doublerows {
      // Get the color bits for this bit plane and for this given doublerow
      // In our case, we have 6 color bits for each double row:
      // 1 R, 1 G, and 1 B bit for both the top and the bottom half
      // of the double row
      let colors = GetBitplaneColors(b, r);

      // Push the color data for this double row
      io.PushColorData(colors);

      // Latch in the double row's address
      io.LatchAddress(r);

      // Now we activate the LEDs. Keep in mind that we need to CLEAR the
      // voltage on the OE pin to activate the LEDs!
      io.ClearPin(OE);
      // Now wait for a little bit
      nanosleep(150 * (bitplanes-b));
      // And disable the LEDs
      io.ActivatePin(OE);
    }
  }
}
```

2.5 Rendering Colors Accurately

Once you have implemented the basic rendering mechanism, you will notice that the colors do not seem quite right. Most colors will seem too dark and some colors will not be visible at all. The underlying issue here is that the human perception of color brightness is non-linear. The relationship between the color brightness our eyes perceive and the output brightness of our LED matrix follows an approximate power function. To compensate for this non-linear curve, we need to apply a technique called **gamma correction**. There are many possible implementations of the technique. The sample C implementation of the renderer converts the input RGB pixels to the CIE 1931 color space while applying luminance correction. Another option is to just stick with the original color space and to convert the input pixels using a gamma curve. Finding the correction method that works best for you is part of the assignment.

2.6 Timer Registers

One problem with our rendering algorithm is that it needs a very precise sleep function to work reliably. If our sleep function is not precise, we will see lots of rendering glitches. Linux has a sleep function (`clock_nanosleep`) that offers nanosecond precision on some systems, but not on our raspberry pi. On our raspberry pi, calling `clock_nanosleep(150)` usually makes the program sleep for a lot more than 150 nanoseconds. This virtually guarantees that we will see rendering glitches if we use the standard `clock_nanosleep` function.

There are several solutions to this problem and you will have to figure out which solution works best for you. One possible solution is to implement a custom `nanosleep` wrapper that only calls nanosleep if the duration of the sleep exceeds a certain threshold. For short sleeps, this function should just do a busy wait while reading the current time. In this hybrid sleep function, we might want to use our raspberry pi's 1 Mhz timer. This timer has a register that it increments by 1 in every timer cycle. With a 1 Mhz frequency, this means the timer register is incremented by 1 every microsecond.

Reading the timer register's value is pretty similar to interfacing with the GPIO. Just like the GPIO controller, the system timer has a register file that we can read/write using memory-mapped I/O. The timer register block is found at physical memory address 0x3F03000. At offset 4 within this register block, we find the 32-bit timer register. Note that the value in this register wraps over if it is incremented while it is at its maximum value!

Another possible solution is to slow down the rendering such that all nanosleep periods last longer than 1 ms.

3 Software Overview

Now that we know how to render to our RGB matrix, let's take a quick look at the code you have to write. In principle, you are free to structure the code in any way you like. However, I recommend that you implement the following structs:

The GPIO struct: This struct should be a low-level interface for communicating with the GPIO controller. I recommend adding functions to configure the output pins (see the Function Select item in Section 2.1), to raise and lower the voltage levels on specific pins (see the Pin Output item in Section 2.1), and to do sanity checks.

The Timer struct: This struct should implement methods to read the timer register (see Section 2.6) and to sleep with high precision.

The Image struct: This struct should store the raw pixel data for the image and should implement the image parsing routines.

The Frame struct: This struct should store the pixel data for the current frame, and should implement the logic to fetch color data and to determine which bits we have to set to push the row address and color data for a specific row.

4 Useful Rust Crates

Rust does not offer direct access to many of the POSIX functions we find in the C implementation of the driver. To access the necessary functionality in Rust, we will have to use some external crates. Here are some of the crates you might find useful when implementing your driver and client. You can look up the documentation and Cargo.toml entries for these crates at `https://crates.io`:

- The **mmap** crate provides access to the memory mapping API. You need this API to map the GPIO and Timer register files into your application's memory.
- The **shuteye** crate provides access to the system's nanosleep function. Note that, even with this crate, the nanosleep function will be imprecise on your raspberry pi.
- The **ctrlc** crate allows you to install a signal handler to catch the CTRL+C signal.
- The **time** crate provides various methods for reading the system time and calculating time durations.
- The **nix** crate provides methods you can use to check if the user that started the program has sudo privileges. Note that sudo privileges are required to map `/dev/mem` using `mmap`.

Appendices

A Mapping of RGB matrix pins to GPIO pin numbers

If we use the Adafruit RGB HAT bonnet, the matrix pins will be mapped to the following GPIO pin numbers:

0. Not mapped
1. Not mapped
2. Not mapped
3. Not mapped
4. **Output Enable (OE)** pin
5. **R1** color pin
6. **B1** color pin
7. Not mapped
8. Not mapped
9. Not mapped
10. Not mapped
11. Not mapped
12. **R2** color pin
13. **G1** color pin
14. Not mapped
15. Not mapped
16. **G2** color pin
17. **Clock (CLK)** pin
18. Not mapped
19. Not mapped
20. **D** address pin (not used)
21. **Strobe/Latch (LAT)** pin
22. **A** address pin
23. **B2** color pin
24. **E** address pin
25. Not mapped
26. **B** address pin
27. **C** address pin