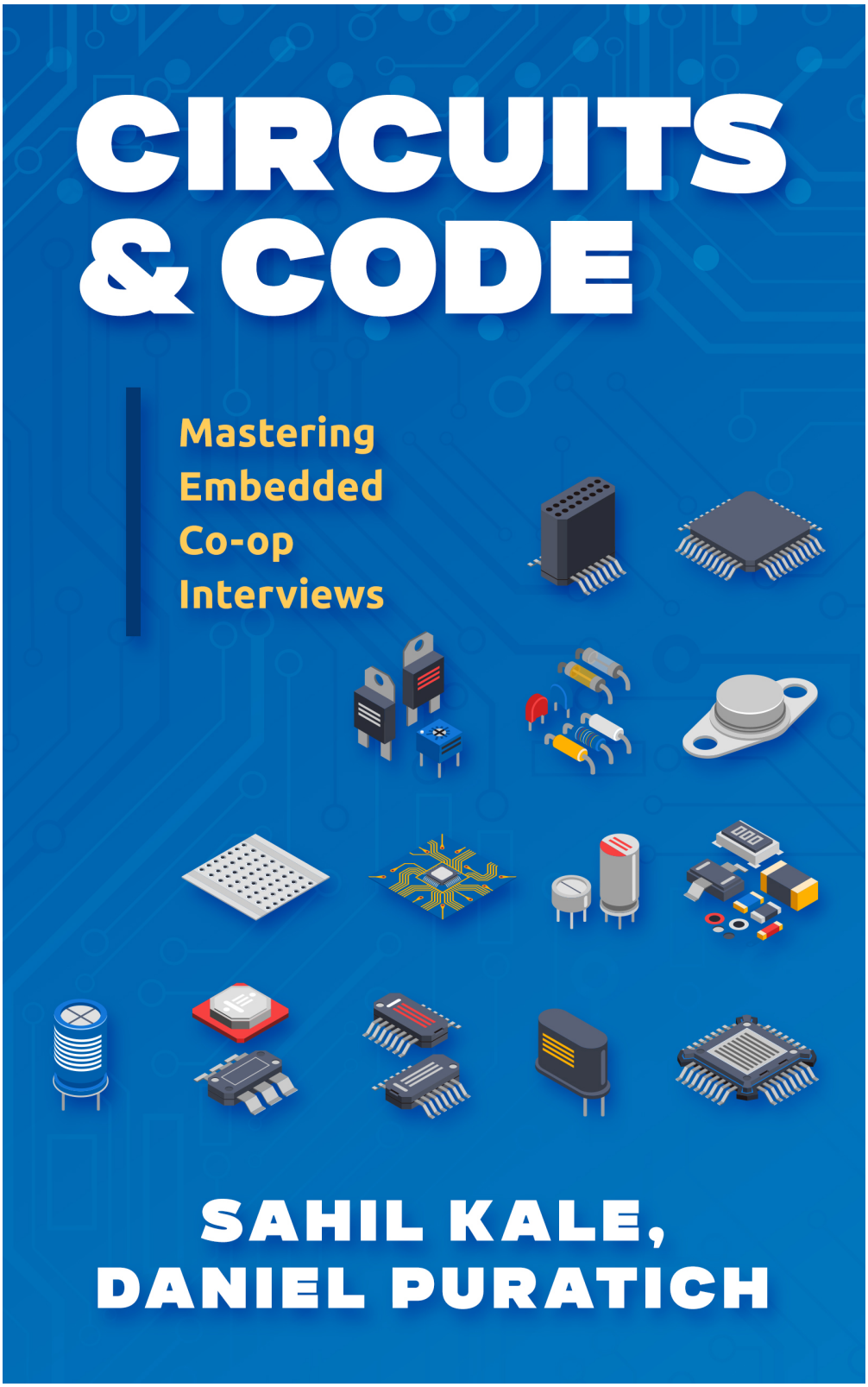
CIRCUITS
& CODE
Mastering
Embedded
Co-op
Interviews
SAHIL KALE,
DANIEL PURATICH

Circuits & Code: Mastering Embedded Co-op Interviews
Book Teaser

February 2, 2025

Contents

1 Book Introduction

Circuits and Code offers over 20 interview-style questions and answers designed for embedded software and electrical engineering interns. Written by two authors with experience hiring and mentoring interns for embedded software and electrical engineering roles, it draws on their academic background and industry expertise to focus on the concepts that matter most to interviewers.

With clear, concise explanations, coding snippets, and sample circuit diagrams, this guide provides a practical resource for motivated students preparing for co-op interviews. The complete, thoughtfully designed questions are ideal for interview practice, helping you build confidence in areas like firmware development, control systems, and hardware design. Whether you're tackling your first technical interview or refining your skills, this book offers you the tools to succeed.

1.1 Interested in purchasing the full book?

This PDF is just a teaser - the full book contains 20+ questions and answers, as well as additional content. Head on over to the book's website!

1.2 About the Authors

Sahil interned at Tesla, Skydio, and BETA Technologies, focusing on real-time embedded software for safety-critical control systems.

Daniel interned at Tesla, Anduril Industries, and Pure Watercraft, focusing on power electronics and board design.

We met at (and both led) the Waterloo Aerial Robotics Group (WARG), a student team that designs and builds autonomous drones. Uniquely, we have experience in hiring and interviewing multiple co-op students, giving us insight into the interview process from both sides, as well as an understanding of what responses are expected from candidates. We value mentorship, enjoy sharing our knowledge, and take pride in helping others succeed in their co-op journeys.

1.3 Disclaimer

This book is designed to be an educational resource, drawing from the authors' experiences and research. While we've done our best to ensure accuracy, readers are encouraged to use their own judgment and explore additional resources as needed. The authors and publisher are not responsible for any errors or omissions. Please note, the content is for informational purposes only and is not intended as professional advice.

2 Draw a circuit to control a LED from a microcontroller GPIO pin.

The LED is to operate at 10mA and has a 2V forward voltage. The microcontroller GPIO, with a 3.3V logic level, can source and sink up to 5mA. Describe a method to control the LED's brightness without altering the circuit.

——— Answers Ahead ———

Remainder of page intentionally left blank. Solution begins on next page.

2.1 Context

Discrete LEDs (*Light Emitting Diodes*) are often used on circuit boards indicators to end users and firmware developers about the state of an embedded system[1]. A common first program executed during board bring-up by firmware developers is to blink the onboard LEDs as an indicator the microcontroller is alive and functional. For end users, it is very common to use LEDs to indicate that the embedded system is powered on and operating nominally. Often, in electronic circuits, an LED is connected in some fashion to a microcontroller GPIO (*General Purpose Input / Output*) pin in order to turn it on and off via firmware.

2.2 Controlling Current to an LED

The circuit given in Figure 1 shows a schematic of an LED powered from a constant voltage source, V_s, with a fixed resistance, R_l in series with the LED. Note this circuit cannot be controlled by a microcontroller yet. The LED has a forward voltage drop, V_f, and a forward current, I_f. The goal is to determine the value of R_l, limiting the current through the LED (I_f).
The circuit can be solved by modelling the forward voltage drop of the LED, V_f, as a fixed voltage and applying Ohm's Law to solve for solve for R_l, as shown in equation (1):

$$\begin{aligned} V_s - V_f &= R_l \cdot I_f \\ R_l &= \frac{V_s - V_f}{I_f} \end{aligned} \tag{1}$$

Note that for LEDs, the brightness is roughly proportional to the current flowing through the LED. Consequently, the brightness of the LED can be varied by changing the resistance value or the voltage to the LED.[2] In practice, an LED's forward voltage is somewhat dependent on I_f and device temperature. "I-V curves" across temperature are usually given by LED manufacturers in the LED's datasheets, however, an assumption of a constant V_f is enough for approximate solutions.

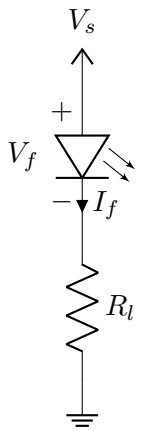

Figure 1: Voltage Source Powering an LED

2.3 Transistors

Transistors are three terminal, electronically-controlled switches in which one terminal is used to control the switching between the other two terminals. The two most commonly used transistors are MOSFETs (*Metal Oxide Semiconductor Field Effect Transistors*) and BJTs (*Bipolar Junction Transistors*), though there are other types. For a BJT, a small current to the base allows a large current to flow between emitter and collector terminals. For a FET (*Field Effect Transistor*), a voltage potential difference between the gate and the source allows current to flow between drain and source. These devices can be drawn with a variety of schematic symbols, but are most commonly seen as:

[1] LEDs can also be a primary feature of a device - an example case is high power LEDs, such as automobile headlights, which require more complex circuitry to drive. This question will address only the simpler case of lower power LEDs.

[2] To give a reference, a small, surface-mounted LED are usually rated for 20mA max (so 20mA * 2V = 40mW), and are visible indoors at just 1mA. For a firmware debugging LED, 2.5mA, is very common.

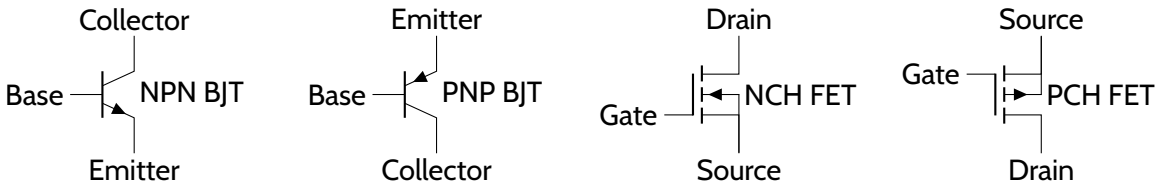

Figure 2: Common Transistors

FETs, ideally, do not require any power consumption to keep them enabled, whereas BJTs require current to be supplied constantly. This means FETs are typically preferred when power consumption is a critical consideration; this is primarily for higher power circuits in which excessive power consumption directly results in a need for expensive cooling systems. BJTs are a much older technology and are easier to fabricate, making them far preferred when optimizing for cost.

Another consideration is low vs. high side switching when using a transistor to enable and disable (aka switch) a load. The solution given in Figure 2.4 demonstrates low side switching. There are numerous implications of this design decision that are out of the scope of this guide.

2.4 Controlling the LED from a Microcontroller

The microcontroller GPIO (*General Purpose Input / Output*) pin is not capable of providing enough current to drive the LED (*Light Emitting Diode*) as desired so an external transistor is required to buffer the signal from the microcontroller. The following circuit in Figure 2.4 demonstrates a simple cost optimized solution to this question.

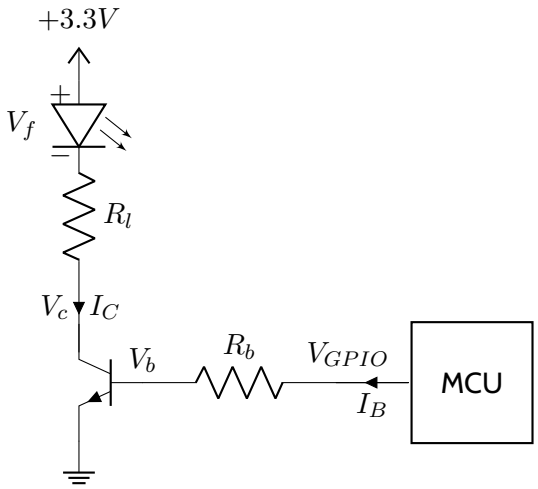

Figure 3: GPIO Driving an LED

An NPN BJT is used to switch the LED on and off. This type of transistor has the governing equation: $I_C = I_B \cdot \beta$. I_C represents current into the collector pin, I_B represents current into the base pin, current out of the emitter, I_E, is given by $I_E = I_B + I_C$. Common parameters for this BJT are $V_{BE} \approx 0.7V$, $\beta \approx 100$, where V_{BE} is the forward voltage drop from the base to the emitter, and β is the current gain

of the transistor. From this circuit drawing, the emitter voltage (V_E) is connected to ground so $V_E = 0V$ meaning $V_{BE} = V_B$. Note that these are approximations and vary based on the part number selected.

When the GPIO pin is at logic low ($V_{GPIO} = 0$ V), the base voltage (V_B) is approximately 0 V. Consequently, the base current (I_B) and collector current (I_C) are both 0 A, and the LED remains off. When the GPIO pin is at logic high ($V_{GPIO} = 3.3$ V), the goal is to fully enable the transistor and allow more than 10 mA of current through the collector (I_C). To achieve this, the base current is selected as $I_B \approx \frac{I_{B_{max}}}{2} = 5\,\text{mA}/2 = 2.5\,\text{mA}$, which allows a maximum collector current of $I_C = 2.5\,\text{mA} \cdot 100 = 250\,\text{mA}$. Since $250\,\text{mA} \gg 10\,\text{mA}$, the LED will turn on, and the collector voltage (V_C) will approach 0 V. Note that the current flowing through the LED can be adjusted by setting R_l to an appropriate value.

The value of R_b can be solved by using Ohm's law where $V_{GPIO} - V_B = I_b \cdot R_b$. For this circuit, $V_B = V_{BE}$ - therefore, the equation becomes $R_b = \frac{V_{GPIO} - V_B}{I_b} = \frac{3.3\text{V} - 0.7\text{V}}{2.5\text{mA}} = 1040\Omega$. Rounding to commonly available resistor values gives $R_b = 1\text{k}\Omega$ as a potential solution.

The forward voltage drop, V_f, is given as 2V, so Ohm's law can be used to solve for the value of R_l. Ohm's Law gives $V_s - V_f = I_f \cdot R_l$ which can be rearranged into $R_l = \dfrac{V_s - V_f}{I_f} = \dfrac{3.3V - 2V}{10m} = 130\Omega$. This resistor can be found in the E24 resistor series as a common resistor value, so no rounding is needed.

2.5 Pulse Width Modulation

When controlling an LED from a microcontroller, the brightness of the LED can be modulated using PWM (*Pulse Width Modulation*). Adjusting the *duty cycle* (amount of 'on' or logic high time) of pulse width modulation, provided the frequency f is much greater than perceivable by the human eye, results in the appearance that the LED brightness is changing. If f is too low then it will be apparent to a viewer that the LED is turning on and off. PWM implementations are usually done in hardware via dedicated timers, where the frequency is set to a constant, high value and the timer's duty cycle is adjusted to control the brightness. An example of a PWM waveform with a duty cycle of 80% is shown in Figure 4.

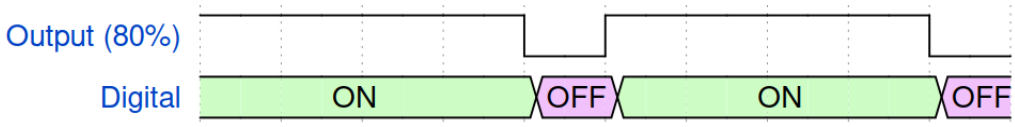

Figure 4: PWM Waveform

Note that PWM's application is not limited to LEDs - in general, PWM can be thought of as a way to control the average voltage across a load, and is used in motor control, power supplies, and more.

3 Implement a PID controller in C and discuss its typical applications.

——— Answers Ahead ———

Remainder of page intentionally left blank. Solution begins on next page.

A PID (*Proportional-Integral-Derivative*) controller is common type of feedback controller. It's ubiquity comes from its relatively easy implementation and effectiveness in a wide range of control systems. Its typical applications include **motor speed/position control, temperature control, lighting regulation**, and many more. The theory behind PID controllers is considered to be out of scope for this guide, but Tim Wescott's article titled PID Without a PhD provides a great introduction to PID controllers without delving into linear control theory.

3.1 PID Controller Implementation

The form of a PID controller is given by equation (2) [1]. K_p is the proportional gain, K_i is the integral gain, and K_d is the derivative gain, with $u(t)$ being the desired controller output. The error term $e(t)$ is the difference between the desired setpoint (i.e. reference) and the system output (i.e. process variable). The integral term is the sum of all past errors, and the derivative term is the rate of change of the error[3]. The equation can appear daunting, but the implementation is quite straightforward.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{2}$$

First, a C structure definition is created to hold the PID gains, as well as temporary variables for calculating the integral and derivative terms. This structure is shown in Listing 1.

```c
typedef struct {
  float kp; // Proportional gain constant
  float ki; // Integral gain constant
  float kd; // Derivative gain constant

  float integral;   // Stored integral value
  float prev_error; // Stored last input value
} pid_t;
```

Listing 1: PID Control Structure

Breaking apart the problem into smaller functions, we can implement the terms of the equation as follows in C. Note that `dt` is the timestep (period) between control loop iterations.

- The proportional term is simply the product of the proportional gain and the error. It is responsible for increasing system responsiveness, but can cause overshoot.
  `const float p_term = pid->K_p * error;.`
- The integral term accumulates the error over time, summing up all past errors. Applying a control signal proportional to the integral-error helps reduce steady-state error. To approximate this integral, we use the commonly-chosen Backward Euler method [1], which updates the integral (sum) by adding the product of the current error and the timestep. Note that the computation of the `i_term` comes after the addition of the integral in this approximation. See Figure 5 for a visual representation of the Backward Euler method.
  `pid->integral += error * dt;.`
  `const float i_term = pid->K_i * pid->integral;.`

[3] If the terms integral and derivative are unfamiliar, an excellent resource is Khan Academy's calculus courses here. These concepts are typically covered in an introductory calculus course.

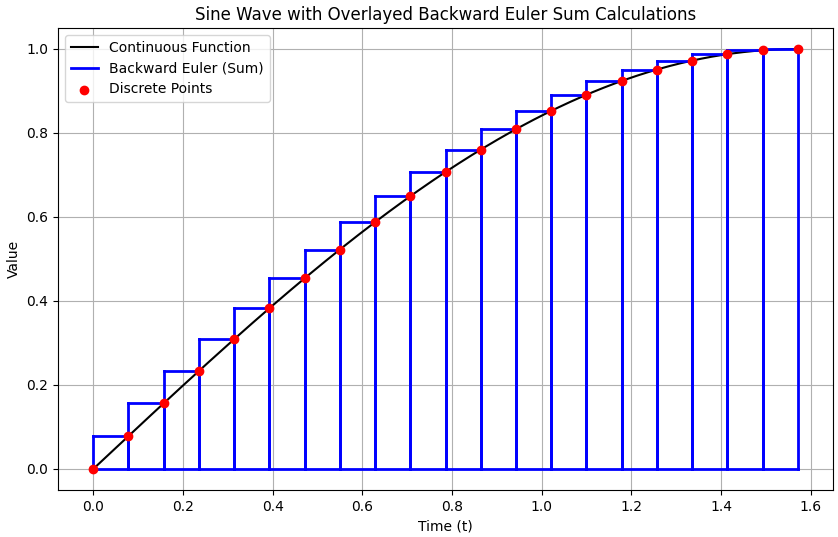

Figure 5: Backward Euler Discretization of the Integral Term

- The derivative term is the rate of change of the error, and can help reduce overshoot. To approximate this derivative, we use the Backward Euler method, which calculates the derivative (slope) as the difference between the current error and the previous error, divided by the timestep (slope $= \frac{\text{rise}}{\text{run}}$). See Figure 6 for a visual representation of the Backward Euler method.
`const float d_term = pid->K_d * ((error - pid->prev_error) / dt);`.

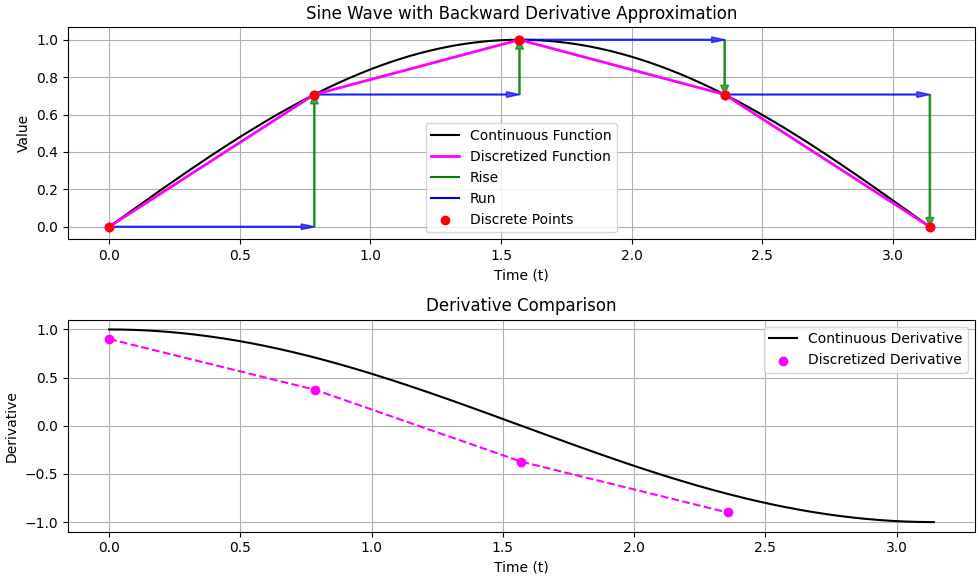

Figure 6: Backward Euler Discretization of the Derivative Term

Putting it all together, the PID controller step function is shown in Listing 2. Note the added check for a `NULL` pointer and positive timestep as either can cause the function to not run correctly.

```c
#include "pid_typedef.h"
#include <stddef.h>

void pid_init(pid_t *pid) {
  if (pid != NULL) {
    pid->integral = 0.0F;
    pid->prev_error = 0.0F;
  }
}

float pid_step(pid_t *pid, float setpoint, float measured_output, float dt) {
  float ret = 0.0F;
  // Check for NULL pointer and positive time step
  if ((pid != NULL) && (dt > 0.0F)) {
    const float error = setpoint - measured_output;
    pid->integral += error * dt;

    const float p_term = pid->kp * error;
    const float i_term = pid->ki * pid->integral;
    const float d_term = pid->kd * (error - pid->prev_error) / dt;

    pid->prev_error = error;
    ret = p_term + i_term + d_term;
  }
  return ret;
}
```

Listing 2: PID Controller Step

3.2 Follow-ups

- **Anti-windup**: What is integral windup, and how can it be prevented in a PID controller?
- **Tuning**: What effect does adjusting the gains K_p, K_i and K_d have on the system's response?
- **Filtering**: What are possible implications of using a poorly filtered signal with a PID controller?

4 Compare and contrast I2C *(Inter-Integrated Circuit)* and SPI *(Serial Peripheral Interface)*.

——— Answers Ahead ———

Remainder of page intentionally left blank. Solution begins on next page.

4.1 I2C

I2C is a *half-duplex* (can either transmit or receive, but not both simultaneously) digital protocol developed by Phillips in 1982 [2]. It enables a host device[4] (referred to as a *master*[5]) to communicate with multiple peripheral devices (referred to as *slaves*) over a two-wire serial bus.

4.1.1 Physical Layer

The physical layer of I2C consists of two wires: SDA (Serial Data) and SCL (Serial Clock). By definition, it is a *synchronous* protocol, meaning that the clock signal is shared between the master and slave devices. The SDA line carries the data, while the SCL line carries the clock signal. The SDA line is bidirectional, allowing both the master and slave to transmit and receive data. The SCL line is unidirectional, controlled by the master device (though it can be asserted by a slave to pause communications to give time for processing, known as *clock stretching*). A hardware bus diagram is shown in Figure 7.

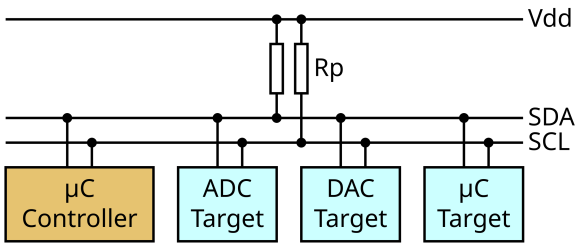

Figure 7: I2C Bus Diagram, Source: Wikipedia [3]

The bidirectional nature of the SDA and SCL lines is achieved by using *open-drain* drivers. Open-drain drivers can pull the line low, but not drive it high, instead relying on a *pull-up* resistor to "pull the voltage up". This is contrast to a *push-pull* driver, which can drive the line both high and low. Open-drain drivers, while advantageous in ensuring that the bus can never be shorted by two devices driving the line with different voltages, suffers from slower rise/fall times due to the pull-up resistor forming an RC circuit with the parasitic bus capacitance, and limits the maximum bus speed to 400kHz traditionally, though higher speeds just above 1 MHz are permissible in newer versions of the specification.

4.1.2 Data Format

The data format of I2C features the following components, and is shown graphically in Figure 8:

1. **Start Condition**: The master device initiates communication by pulling the SDA line low while the SCL line is high, signalling the beginning of a transfer.

2. **Address (7 bits)**: A 7-bit address is transmitted on the SDA line to indicate which slave device the master wishes to communicate with.

[4] I2C does support multiple master devices, however, this article focuses on the significantly more prevalent single-master implementation.

[5] The phrases 'master' and 'slave' are slowly being phased out due to their origins. However, at time of writing, 'master' and 'slave' are the most commonly used terms and are used in this guide. Alternative verbiage includes 'controller' for master and 'peripheral' or 'target' for slave.

3. **Read/Write Bit**: The 8th bit of the address byte is used to indicate whether the master wishes to read from or write to the slave device. A `0` indicates a write operation, while a `1` indicates a read operation. If a read is requested, control of the SDA line is transferred to the slave device.

4. **Acknowledge Bit**: After each byte is transmitted, the receiving device (master or slave) sends an acknowledge bit. If the receiving device pulls the SDA line low, it indicates that it has received the byte and is ready for the next byte. If the SDA line remains high, it indicates that the receiving device is not ready, or that an error occurred.

5. **Data Byte(s)**: Data bytes are transmitted in 8-bit chunks, with each byte followed by an acknowledge bit.

6. **Stop Condition**: The master device signals the end of the transfer by, in the case of a write, releasing the SDA line while the SCL line is high, or in the case of a read, sending a `NACK` (Not Acknowledge) bit followed by a stop condition.

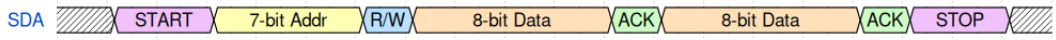

Figure 8: 2 Byte I2C Data Frame Format

4.2 SPI

SPI was developed by Motorola [4] and is a *full-duplex* (simultaneous transmit and receive) synchronous serial communication protocol. It is commonly used in embedded systems to communicate between a master device and one or more slave devices. SPI is a *four-wire* protocol, consisting of the following signals: MISO (Master In Slave Out), MOSI (Master Out Slave In), SCLK, and CS (Chip Select). A timing diagram of a 1-byte SPI transaction is shown in Figure 9.

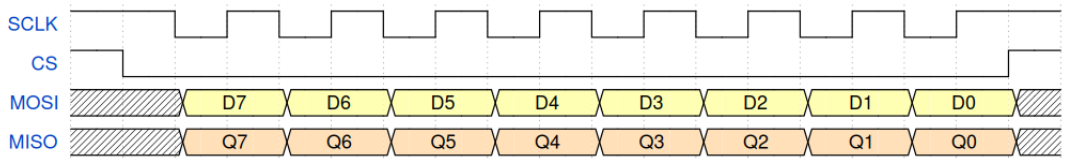

Figure 9: 1-byte SPI Timing Diagram

Unlike I2C, SPI does not have a standard addressing scheme, and the master device must *assert* (pull down) a CS line to connected to the slave device it wishes to communicate with - the requirement for each slave device to feature its own CS line increases SPI's wiring complexity. Owing to the push-pull nature of SPI drivers, the bus is faster than I2C (low MHz range). The bus diagram is shown in Figure 10 (Note the diagram in Figure 10 uses *SS* for *Slave Select*, which is synonymous with *CS*).

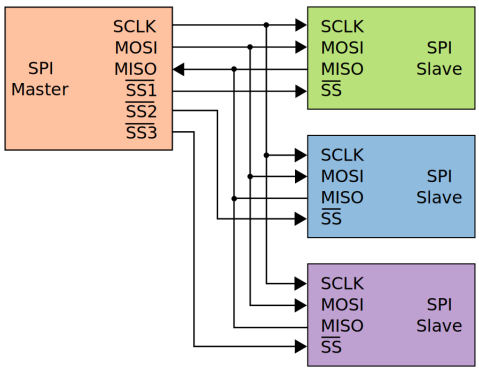

Figure 10: SPI Bus Diagram, Source: Wikipedia [5]

4.3 Comparison

- **Bus Speed**: SPI is faster than I2C, with speeds in the MHz range (ex: 1-40 MHz) compared to I2C's 400 kHz range due to the differences in drive type. As a result, SPI is often used in applications where the transfer speed is required to be high.
- **Wiring Complexity**: I2C requires, at most, two wires, regardless of the number of devices on the bus owing to its addressing scheme. A SPI master requires at least four wires, plus an additional wire for each slave device (each slave will require 4 wires).
- **Frame Format**: I2C has a more complex frame format than SPI, with start and stop conditions, address bytes, and acknowledge bits, which can assist in debugging unresponsive slave devices. However, SPI has a simpler frame format, with no addressing scheme and no acknowledge bits, which reduces the overhead of each transaction and affords more flexibility in the data format.

4.4 Follow-ups

- What are strategies for dealing with conflicting I2C addresses?
- What are strategies for dealing with the possibly-large number of CS lines in SPI?
- How are pull-up resistance values selected for I2C?

References

[1] D. Y. Abramovitch, "A unified framework for analog and digital pid controllers," pp. 1492–1497, 2015. **DOI:** 10.1109/CCA.2015.7320822.

[2] SparkFun Electronics. "I2c." Accessed: 2024-12-26. (n.d.), [Online]. Available: https://learn.sparkfun.com/tutorials/i2c/a-brief-history-of-i2c.

[3] Tim Mathias. "Example i2c schematic." Accessed: 2024-12-26, CC-BY-SA 4.0. (2021), [Online]. Available: https://en.wikipedia.org/wiki/I%C2%B2C#/media/File:I2C_controller-target.svg.

[4] Total Phase. "Spi background." Accessed: 2024-12-26. (n.d.), [Online]. Available: https://www.totalphase.com/support/articles/200349236-spi-background/?srsltid=AfmBOooNHnQYWKPxz4YWdKyz9Bp4tEq3lw87j9EO6zr8YzMyRQkNYJz1.

[5] User:Cburnett. "Spi three slaves." Accessed: 2024-12-26, CC-BY-SA 3.0. (2006), [Online]. Available: https://commons.wikimedia.org/wiki/File:SPI_three_slaves.svg.