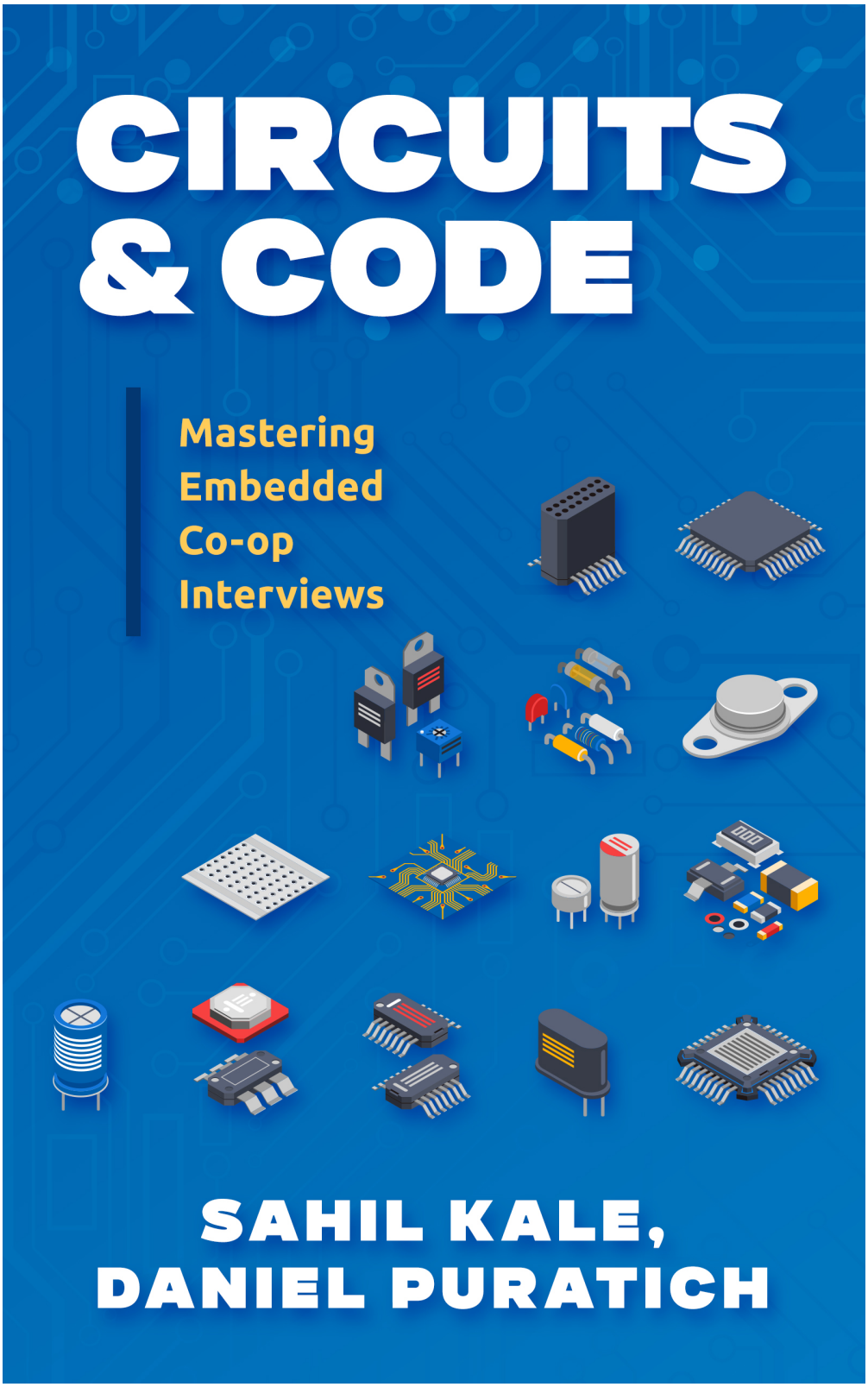

CIRCUITS
& CODE
Mastering
Embedded
Co-op
Interviews
SAHIL KALE,
DANIEL PURATICH

Circuits & Code: Mastering Embedded Co-op Interviews

February 15, 2025

Contents

1 Introduction

This guide was created by two Waterloo Engineering students, Daniel Puratich and Sahil Kale. Wanting to support our peers in their co-op journeys, we noticed that we were often answering the same questions when asked about co-op interview prep. Noticing that many firmware and hardware co-op interviews focus on fundamental concepts, we decided to compile a comprehensive list of commonly asked questions and answers to help students better prepare for their interviews. The book also has an accompanying website, which includes additional content and resources. You can find it at circuits-and-code.github.io.

1.1 Target Audience

Our guide is designed for engineering undergraduates seeking technical internships in embedded software, firmware, and electrical engineering across the US and Canada. The questions and answers aim to provide a foundational understanding, making them ideal for those just starting in the field. The content is also relevant for more experienced students looking to refine their skills and prepare for technical interviews.

1.2 How to Use This Guide

This guide is structured to help you understand and answer common technical questions asked in firmware and hardware co-op interviews. Each section starts by introducing a question and relevant informational content, followed by a detailed answer. We recommend reading through the content and attempting to answer the questions yourself before reviewing the provided answers. Answers begin after this line:

——— Answers Ahead ———

It's important to note that many questions in this guide have multiple correct answers. We've provided detailed explanations for each question to help you understand the concepts and reasoning behind our answers, and pointed out context-specific considerations where applicable. The 'best' answer will depend on the context of the question and the interviewer's expectations - remember that engineering is about problem-solving and tradeoff decision-making, and there are often multiple ways to approach a problem.

1.3 Extra Practice

The guide also features *extra practice*, which are extensions on top of the questions already discussed in this book. However, these extra practice questions do not have full explanations and are provided for intuition building as they are often common follow-up questions.

1.4 Breadth of Topics

Embedded systems cover a broad range of topics that intersect hardware and software. While most questions in this guide are broadly applicable to both firmware and hardware roles, some are tailored to specific industries. For example, topics like mutexes versus semaphores are more likely to appear in firmware interviews, whereas buck converter vs low dropout regulator is more likely to appear in EE interviews. **We recommend focusing on the questions most relevant to your target role while gaining a general understanding of both fields.**

1.4.1 Code

Code snippets are included to illustrate various embedded software concepts. The code is written in C and can be copied and compiled at your discretion, though some snippets may require the inclusion of additional header files and minor restructuring. Note that while snippets can be compiled, they are not intended to be standalone programs or run unless a `main` function is defined.

1.5 Acronyms

This guide features numerous acronyms and phrases that are commonly accepted in industry. We will provide definitions for these acronyms the first time we use them, but will assume you understand them in subsequent questions as understanding these terms will be integral to understanding common interview questions.

1.6 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 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.

1.6.1 Other Work

We've published other (free!) guides to help engineering students land firmware and hardware co-op roles. Check them out:

- The Sahil and Daniel Co-op Resume Guide - focuses on resume writing and tailoring for hardware and firmware roles.
- The Sahil and Daniel Co-op Process Guide - offers our tips and tricks to landing a co-op role in hardware and firmware.

1.7 Acknowledgements

We'd like to thank the several individuals for taking the time to review and provide feedback on this guide. Feel free to reach out to us with feedback as we are looking to improve the guide over time. A full list of acknowledgements can be found on the book's website.

1.8 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 Implement a function to validate the parity of a bitstream.

Assume the following header in Listing1 is available for use in the bitstream parity implementation.

```c
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

typedef enum {
  BITSTREAM_PARITY_EVEN, // Parity bit = 0 if number of ones in bitstream is
                         // even, 1 otherwise
  BITSTREAM_PARITY_ODD,  // Parity bit = 0 if number of ones in bitstream is
                         // odd, 1 otherwise
} bitstream_parity_E;

/**
 * @brief Check if the parity of the bitstream is valid
 * @param bitstream The byte array to check
 * @param length The length of the byte array
 * @param parity_bit The parity bit to check against
 * @param scheme The parity to use for the check
 * @return true if the bitstream has the correct parity, false otherwise
 */
bool bitstream_parity_valid(uint8_t *bitstream, uint32_t length,
                            bool parity_bit, bitstream_parity_E scheme);
```

Listing 1: Bitstream Parity Header

——— Answers Ahead ———

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

2.1 Concept of Parity

In digital communication, a parity bit is an extra bit appended to a binary data stream to assist in error detection. The goal is to keep the number of 1's in the data stream (including the parity bit) either even (*even parity*) or odd (*odd parity*), depending on the parity scheme. The *parity* of a binary number refers to whether the number of 1's in the binary representation of the number is even or odd. Figure 1 shows waveforms of digital signals with no parity, odd, and even parity, as well as even parity with an error.

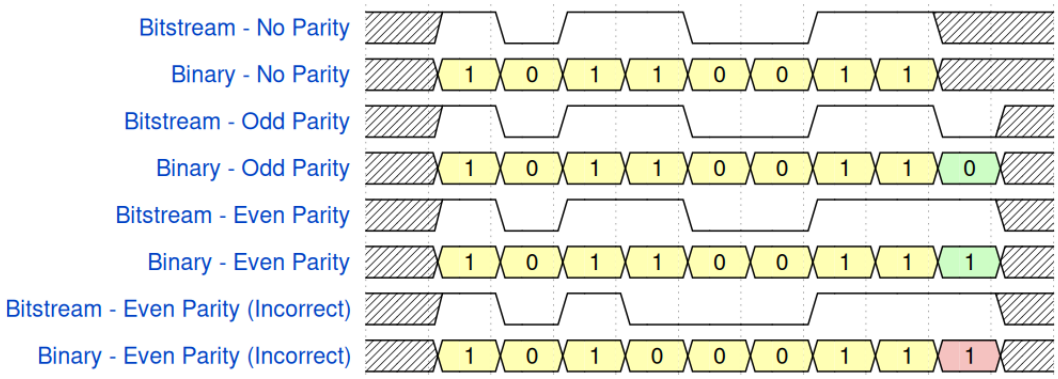


Figure 1: Parity Waveform Diagram

2.2 Counting number of ones in a Byte

To implement a parity bit, we need to count the number of 1's in the data stream[1].

2.2.1 Bitwise Operations

Before counting the number of `1`s in a byte, it's important to understand bitwise operations. If you're not familiar with these (ex: &, |, », etc.), consider reviewing Geeks for Geek's explanation on the topic. Bitwise operations are common topics in embedded software interviews, and it's essential to understand them.

2.2.2 Simple Approach

A simple approach to count the number of 1's in a byte is to iterate through each bit and check if it's set. Listing 2 shows a simple implementation of this approach.

```c
#include "bitstream_parity.h"

uint8_t count_ones_simple(uint8_t byte) {
  uint8_t count = 0;
  for (size_t i = 0; i < 8; i++) {
    const uint8_t mask = 1 << i;
    if (byte & mask) {
      count++;
    }
  }
  return count;
}
```

Listing 2: Simple Approach to Count Number of 1's in a Byte

[1] It's not uncommon to be asked this as a standalone question.

While effective, this approach is not efficient. It requires 8 iterations to count the number of 1's in a byte. We can improve this by using a technique called *Brian Kernighan's Algorithm*.

2.2.3 Brian Kernighan's Algorithm

Brian Kernighan's Algorithm is a technique to count the number of set bits in a byte. The algorithm works by repeatedly 'removing' the least significant 1 bit and counting the number of iterations required to reach 0. Listing 3 shows the implementation of this algorithm. It is more efficient than the simple approach as it only requires the number of set bits to be counted [1].

```c
#include "count_ones_bk.h"

uint8_t count_ones_bk(uint8_t byte) {
  uint8_t count = 0;
  while (byte != 0) {
    byte &= (byte - 1);
    count++;
  }
  return count;
}
```

Listing 3: Brian Kernighan's Algorithm to Count Number of 1's in a Byte

2.2.4 Faster Ways

There are faster ways to count the number of 1's in a byte, such as a lookup table to count the number of 1's for every byte. This approach is faster but requires more memory. Hamming Weight is another technique that counts the number of 1's in a byte in a more efficient manner, but it's more complex to implement (and usually out of scope for interviews). There's also usually a dedicated instruction, called `popcount`, that can count the number of set bits in one step, if supported by HW/compiler. More often than not, embedded interviews focus on the fundamentals and emphasize understanding over complex solutions.

2.3 Implementing Parity Bit Check

Putting it together, the steps to implement a parity bit are as follows:

1. For every byte, count the number of 1's (for this example, we'll use Brian Kernighan's Algorithm).
2. Determine the expected parity bit based on the parity scheme (even or odd).
3. Compare the parity bit to the expected parity bit.

Listing 4 shows the implementation of the `check_parity` function that implements the steps mentioned above.

```c
#include "bitstream_parity.h"
#include "count_ones_bk.h"

bool bitstream_parity_valid(uint8_t *bitstream, uint32_t length,
                            bool parity_bit, bitstream_parity_E scheme) {

  uint32_t ones = 0U;
```

```
  // Count the number of ones in the bitstream
  for (uint32_t i = 0U; i < length; i++) {
    ones += count_ones_bk(bitstream[i]);
  }
  bool expected_parity = false;
  const bool number_of_ones_is_odd = (ones & 1U); // & 1U is equivalent to % 2
  switch (scheme) {
  case BITSTREAM_PARITY_EVEN:
    expected_parity = (number_of_ones_is_odd) == true;
    break;
  case BITSTREAM_PARITY_ODD:
    expected_parity = (number_of_ones_is_odd) == false;
    break;
  default:
    return false;
    break;
  }

  return expected_parity == parity_bit;
}
```

Listing 4: Bitstream Parity Implementation

3 How would you sense the voltage of a battery with a microcontroller?

The nominal voltage of the battery is 24 volts. Consider how your solution behaves when the battery's voltage varies.

——— Answers Ahead ———

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

3.1 Implementation

At first glance, an analog-to-digital converter (*ADC*) could be used to directly sample the battery voltage. While this is a valid solution, it is not a standard practice as high voltage ADCs are less commonly included in microcontrollers and are more expensive to implement discretely. For this reason, the use of a low voltage ADC commonly found in microcontrollers is preferred.

To do this, a voltage divider circuit can be used to scale the voltage of the battery by a fixed ratio, specified by the two resistors, such that the ADC can sample the signal without exceeding the maximum input voltage threshold.

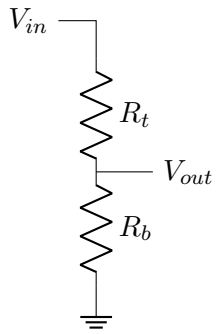


Figure 2: Voltage Divider Circuit

3.2 Motivation

Electronic systems often require high power actuators and batteries to operate - relatively high battery bus voltages are often selected by engineers to minimize current and conduction losses. Microcontrollers are often designed with low input voltages (i.e. 1.8V, 3.3V, 5V) in order to reduce power consumption and transistor size during operation. As chemical batteries charge and discharge the battery, the battery's voltage varies due to numerous factors (notably, state of charge). As a result, many battery-based embedded systems feature a battery-voltage sensing circuit hooked into a microcontroller to monitor the battery's voltage, and to take action when the battery voltage is too low or too high.

3.3 Voltage Divider Circuit

Ohm's Law relates the voltage across a resistor, V, to current flowing through the resistor, I, by $V = I \cdot R$. Applying Ohm's Law to each of the two resistors shown results in Equations (1) and (2). Because the resistors are in series, the current, I, flowing through them is the same ($I = I_{R_t} = I_{R_b}$).

$$\begin{aligned} V_{in} - V_{out} &= I \cdot R_t \\ I &= \frac{V_{in} - V_{out}}{R_t} \end{aligned} \tag{1}$$

$$\begin{aligned} V_{out} - 0 &= I \cdot R_b \\ I &= \frac{V_{out}}{R_b} \end{aligned} \tag{2}$$

Solving the system of equations to eliminate I gives Equation (3) as follows.

$$\begin{aligned}
\frac{V_{out}}{R_b} &= \frac{V_{in} - V_{out}}{R_t} \\
V_{out} \cdot R_t &= R_b \cdot (V_{in} - V_{out}) \\
V_{out} \cdot R_t &= V_{in} \cdot R_b - V_{out} \cdot R_b \\
V_{out} \cdot R_t + V_{out} \cdot R_b &= V_{in} \cdot R_b \\
V_{out} \cdot (R_t + R_b) &= V_{in} \cdot R_b \\
\frac{V_{out}}{V_{in}} &= \frac{R_b}{R_t + R_b}
\end{aligned} \tag{3}$$

This indicates, based on the values chosen for R_t and R_b, V_{out} is scaled down from V_{in} by a ratio determined by R_t and R_b. Given a maximum ADC voltage rating, V_{out}, and a maximum battery voltage, V_{in}, a desired ratio between R_t and R_b can be determined.

However, there are additional considerations in selecting R_t and R_b, as the voltage divider circuit draws current proportional to the sum of its resistance ($R_{sum} = R_t + R_b$). Electrical power dissipation is given by $P = V \cdot I$. Substituting in Ohm's Law to describe the power dissipation of a resistor gives $P = \frac{V^2}{R}$ and $P = I^2 \cdot R$. Note that if the input voltage of the battery is roughly fixed at some nominal V_{in}, then the circuit's power dissipation increases as R_{sum} decreases. This power dissipation (*referred to as quiescent current*) occurs constantly as voltage is always supplied to the circuit and can be significantly wasteful.

If the R_{sum} is too large, then the current flowing in the resistor divider is so small that noise coupling into the signal or input bias current into the ADC can result in significant measuring error.[2]

3.4 Voltage Divider Intuition

Consider a few specific cases of this circuit to help build intuition to approach problems featuring the voltage divider.

- A simple case of the voltage divider circuit is when both resistors have the same value. $R = R_t = R_b$. In this case, $\frac{V_{out}}{V_{in}} = \frac{R}{R+R} = \frac{R}{2 \cdot R} = \frac{1}{2}$ which means $V_{in} = 2 \cdot V_{out}$ or $V_{out} = \frac{1}{2} \cdot V_{in}$.
- Because $R_t > 0\,\Omega$ and $R_b > 0\,\Omega$ are required (as negative resistors do not exist), in all cases of the circuit being employed we observe that $0 < \frac{V_{out}}{V_{in}} < 1$. This indicates that $V_{out} < V_{in}$ always holds for the voltage divider so the circuit always scales a voltage down from its input to its output.
- This circuit assumes no source impedance from V_{in} and no loading connected to V_{out}. However, this assumption is not always valid in practical circuits (and explored later in this answer).

3.5 Input Bias Current

When an ADC is operating nominally, it suffers from a non-ideality known as input bias current.[3] This input bias current, I_{bias}, is in the micro-amp range, is used to feed the internal analog circuit, and is present to varying degrees of severity in all forms of ADCs. This is a problem because it adds loading to our resistor divider and results in Equation (3) being incorrect.

[2] A common value selected for I is roughly 1mA of current to flow in a sensing resistor divider, though different values may be seen based on the application.

[3] Note that input bias current may also be represented by input impedance in which the load current is instead modelled by a loading resistance to ground, however, the concept remains similar.

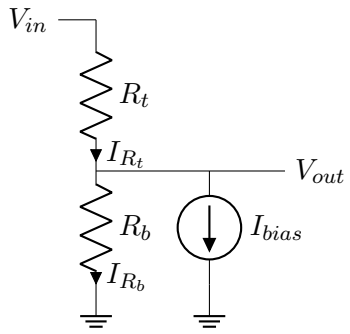


Figure 3: Voltage Divider Circuit with Loading

$$\frac{V_{out}}{V_{in}} = \frac{R_b}{R_t + R_b} - \frac{I_{load}}{V_{in}} \cdot \frac{R_t \cdot R_b}{R_t + R_b} \tag{4}$$

Using the circuit in Figure 3 results in an adjusted transfer function shown in Figure 4. Input bias current is also hard to model and can vary significantly so the simplest method of compensating for this by reducing $R_{sum} = R_t + R_b$ which increases $R_t||R_b = \frac{R_t \cdot R_b}{R_t + R_b}$ and decreases the entire second term of Figure 4.

Another method to compensate for a large input bias current is to use an external voltage buffer, aka a unity gain op-amp (*Operational Amplifier*) to repeat the voltage, but buffer the current.

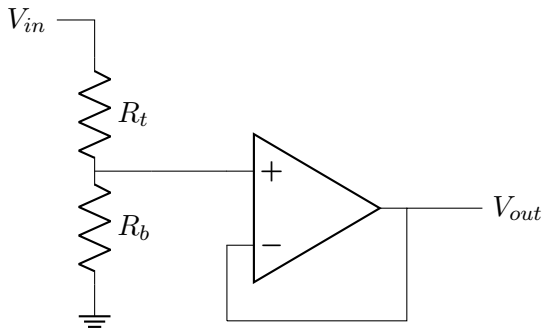


Figure 4: Op-amp Buffered Resistor Divider

Op-amps also have input bias current, however, this can be compensated for it using the circuit in Figure 5 where $R_c = R_t||R_b$.

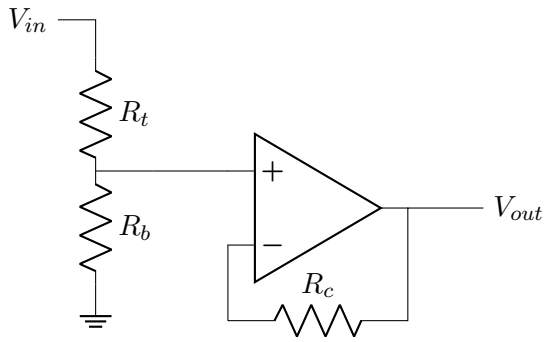


Figure 5: Compensated Op-amp Buffered Resistor Divider

3.6 Pin Overvoltage

Microcontroller pins often feature clamping diodes to protect the device from some transient voltages outside of the permissible operating range. ESD (*Electro-static discharge*) is an example of a potentially destructive transient event. The use of external clamping diodes is common to protect for higher power transients in addition to internal clamping.

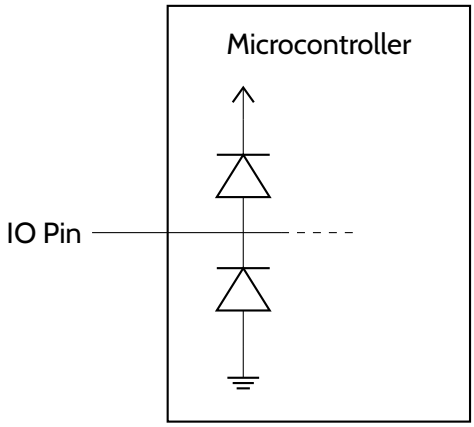


Figure 6: Microcontroller Pin with Clamping Diodes

These clamping diodes, shown in Figure 6, will conduct current when a voltage is applied that exceeds the microcontroller's supply voltage, V_{cc}, and when a voltage is applied that is below the microcontroller's ground reference, V_{ss}. They will conduct current unless if the current becomes excessive resulting in damage to the diodes and consequently damage to the device. Allowing the microcontroller clamping diodes to sink some current during an overvoltage event is permissible. Note that when an ADC pin is overvolted, accurate ADC readings cannot be expected.

External TVS diodes are also used when faster response times are required. A disadvantage of external protection diodes is that they consume some leakage current which will result in less accurate ADC measurements. This leakage current is often difficult to model (non-linear) and can be dependent on numerous factors.

3.7 Analog to Digital

An ADC (*Analog to Digital Converter*) is a component that, as the name implies, converts an analog value into a digital value. The frequency components of the sampled signal and sample rate are critical design decisions to avoid aliasing. Aliasing is a large concept in sampling theory that will not be explored in this guide, however, Tim Wescott's article titled Sampling: What Nyquist Didn't Say, and What to Do About It provides a deeper understanding of this sampling theory.

To avoid aliasing, it is common to see a low pass filter (LPF) added to voltage divider circuits (a capacitor added in parallel to the V_{out} signal) to filter out higher frequency noise. The cutoff frequency of a LPF is the frequency in which the circuit will attenuate to half its input power or $\frac{1}{\sqrt{2}}$ of its input voltage. The cutoff frequency of this low pass filter is usually selected to be roughly five times lower than the sampling frequency to avoid aliasing[4].

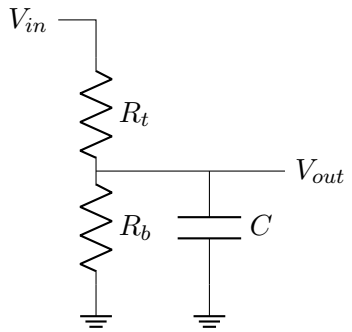


Figure 7: Voltage Divider Circuit with Anti-Aliasing Capacitor

3.8 Layout Considerations

When placing a voltage divider circuit on a PCB, consider:

- The V_{out} trace should be as short as possible to avoid noise from coupling into the signal.
- The low pass capacitor should be placed near the ADC pin so it can filter out noise that couples into V_{out} before the ADC samples it.

3.9 Follow-ups

- Reducing the quiescent current of a voltage divider can be done by increasing R_{sum}, however, this only gets you so far. Propose a simple circuit to disable current consumption when the voltage divider is not needed.[5]
- What if all analog input pins of the microcontroller are already in use?

[4] Refer to LPF theory as to why the cutoff frequency is chosen to be higher - the need arises due to the cutoff being -20dB/decade

[5] A solution is given in extra practice question 23

4 What is an interrupt service routine (ISR), and how do they differ from regular functions in implementation?

——— Answers Ahead ———

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

4.1 Interrupts

An interrupt is a signal that *interrupts* the currently-executing process on the CPU to handle a specific event. Interrupts are often used to handle time-sensitive tasks, such as input/output (I/O) operations, and are essential for real-time systems. This section summarizes general principles of interrupt service routines; more information can be found in embedded systems literature (e.g., [2], [3], and [4]). Some examples are as follows:

- **Timer Interrupts:** Used to keep track of time and schedule tasks.
- **I/O Interrupts:** Used to handle input/output operations.
- **Hardware Interrupts:** Used to signal hardware events, such as a button press.

An interrupt is a powerful construct as it allows the CPU to handle events without necessarily *polling* for whether an event has occurred. This allows the CPU to perform other tasks while waiting for an event to occur.

Elaborating on the button press interrupt example: the CPU can continue executing other tasks until the button is pressed, at which point the interrupt is triggered and the CPU can handle the button press. This is in contrast to polling, where the CPU would have to continuously check if the button is pressed, which is inefficient and wastes CPU cycles.

4.1.1 Theory of Operation Review

When an interrupt occurs, the CPU saves the current state of the program and pushes it on the stack, executes the *interrupt service routine* (ISR), and then restores the program's state. [2]

Broadly speaking, there are 2 types of interrupt handling mechanisms. Depending on the system architecture and interrupt source type, one or both may be used [3]:

- **Non-Vectored/Polled Interrupts:** Interrupts are handled by a common ISR, which then determines the source of the interrupt. An example would be when 3 unique button interrupts are handled by a single ISR, which then determines which button was pressed.
- **Vectored Interrupts:** The interrupting device directly specifies the ISR to be executed; the address of the ISR to call is usually stored in a table of function pointers called the *vector table*. An example would be when 3 button interrupts are handled by 3 separate ISRs, avoiding the need for the ISR to explicitly determine the button issuing the interrupt.

Several other concepts are important to understand when working with interrupts, but are not directly related to the question. These include:

- **Interrupt Priority:** Determines which interrupt is serviced first when multiple interrupts occur simultaneously.
- **Interrupt Nesting:** The ability to handle interrupts while another interrupt is being serviced.
- **Interrupt Masking:** Disabling interrupts to prevent them from being serviced.

4.2 Interrupt Service Routines

An *Interrupt Service Routine* (ISR) is a special type of function that is called when an interrupt occurs. Because they are called asynchronously and through hardware, ISRs differ from regular functions in several key ways.

- **Execution Time:** ISRs should be kept as short as possible, as they can block other interrupts from being serviced and stall the main program from running. This is especially important in real-time systems, where missing an interrupt can have serious consequences. In an RTOS, a key assumption made by deadline scheduling algorithms is that ISRs are extremely fast when compared to task execution time [4]. As a corollary, ISR's should avoid blocking operations.
- **Concurrency:** ISR's, by their very nature, are concurrent with the main program. This means that they can interrupt the main program at any time, and the main program must be written with this in mind.
  - In particular, attention should be paid to shared variables and resources between the ISR and the main program, and may require the use of queues, semaphores, or other synchronization mechanisms.
  - In addition, ISR's should avoid non-reentrant (*reentrant functions are functions that can be called again, or re-entered, before a previous invocation completes*) functions, as they can be interrupted and cause unexpected behavior. Examples of non-reentrant functions include those that use global variables or static variables, as well as `malloc()` and `printf()`.
- **No Return Value:** ISRs do not return a value, as they are not called by the program but by the hardware.

5 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.

5.1 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[6]. 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.

5.2 Controlling Current to an LED

The circuit given in Figure 8 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 (6):

$$\begin{aligned} V_s - V_f &= R_l \cdot I_f \\ R_l &= \frac{V_s - V_f}{I_f} \end{aligned} \tag{5}$$

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.[7] 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.

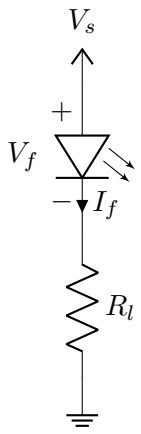


Figure 8: Voltage Source Powering an LED

5.3 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:

[6] 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.

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

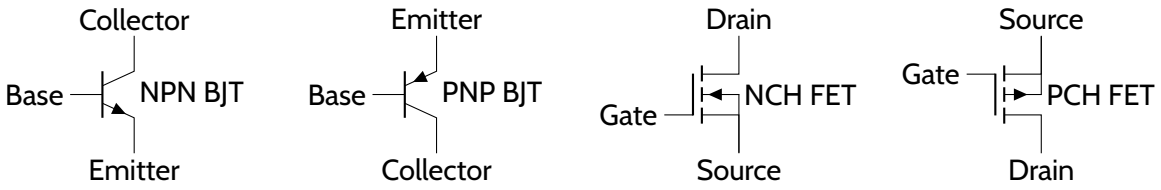


Figure 9: Common Transistors

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.[8]

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 10 demonstrates low side switching. There are numerous implications of this design decision that are explored further in Section **??**.

5.4 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 10 demonstrates a simple cost optimized solution to this question.

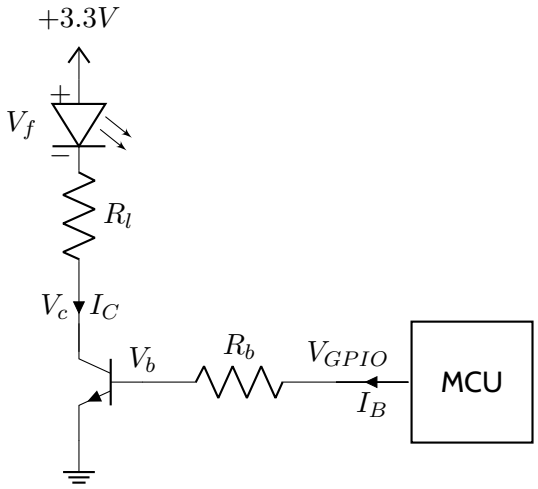


Figure 10: GPIO Driving an LED

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

[8] A solid understanding of both types of transistors is important for common interview questions!

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[9]. 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:

$$R_b = \frac{V_{GPIO} - V_B}{I_b} = \frac{3.3V - 0.7V}{2.5\text{mA}} = 1040\Omega \tag{6}$$

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 $R_l = \frac{V_s - V_f}{I_f} = \frac{3.3V - 2V}{10m} = 130\Omega$. This resistor can be found in the E24 resistor series as a common resistor value, so no rounding is needed.

5.5 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 waveforms are usually created by hardware via dedicated timers, where the frequency is set to a constant, high value and the timer's duty cycle is adjusted (in this case, to control the brightness of an LED). An example of a PWM waveform with a duty cycle of 80% is shown in Figure 11.

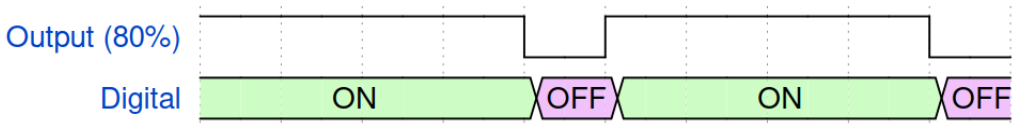


Figure 11: PWM Waveform

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

[9] Technically V_{ce} will not decrease entirely to zero, instead plateauing around $V_{ce} \approx 0.2$V. As the question does not call for very precise control over the LED current the aproximation made here is permissible. In some cases using a smaller value for β is done to compensate for the non-zero V_{ce}.

5.6 Follow-ups

- Prpose a solution using a MOSFET instead of a BJT.
- Propose a solution without using any transistors.

6 Explain the following C keywords: `volatile`, `const`, and `static`.

For each keyword, explain its purpose, as well as use-cases where it would be beneficial to use it.

——— Answers Ahead ———

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

6.1 Volatile

The **volatile** keyword is used to tell the compiler that a variable's value can be *changed unexpectedly* (the value can be changed by something outside the current scope, such as an interrupt service routine (ISR) or by hardware via memory-mapped input/outputs, or I/O). The compiler should not optimize the variable's access, as it may change at any time. Note that using volatile where not necessary can lead to worse performance, as the compiler may not optimize the variable's access.

6.1.1 Example Volatile Use Case

Consider a short program below in Listing5 where a timer interrupt service routine (ISR) increments a variable `timer_counter` every millisecond.

```c
#include <stdbool.h>
#include <stdint.h>

uint32_t shared_counter = 0; // Shared variable

// Interrupt Service Routine (ISR) updates the counter
void ISR_Timer(void) {
  // called every 1ms
  shared_counter++; // Increment counter
}

// Main loop checks the counter
int main(void) {
  while (1) {
    if (shared_counter % 1000 == 0) {
      // Do something every 1 second
    }
  }
  return 0;
}
```

Listing 5: Volatile Use Case

The main loop checks if the `timer_counter`'s modulo (remainder) has reached a certain value and then performs an action. If the `timer_counter` variable is not declared as `volatile`, the compiler may optimize the loop and cache the value of `timer_counter` after reading it only once as the ISR_Timer function does not appear to be called, causing the if statement to never be true. Effectively, the compiler assumes that you, the human, has written dead code (*term for code that is unreachable*) and thinks it can optimize it away. By declaring `timer_counter` as `volatile`, the compiler will always read the variable from memory, ensuring the loop works as expected.

Note: A major use-case of volatile is to access memory-mapped I/O registers in embedded systems. These registers can change due to external events by the hardware (*HW*) rather than code, and the compiler should not optimize the reads/writes to these registers. Memory-mapped I/O registers should be declared as such (ex: `volatile uint32_t * const UART_DR = (uint32_t *)0x40000000;`) to avoid the aforementioned issues.

6.2 Const

The **const** keyword refers to a variable that is **read-only** [5]. The compiler will throw an error if an attempt is made to modify a variable labeled as `const`. **Importantly, `const` does not mean that the value is constant, but rather that the variable cannot be modified by the program**. The distinction is important, as it's possible to have values that are declared as `const`, but are not constant.

Const's primary use is to make code more readable and maintainable. By declaring a variable as `const`, the programmer can signal to others that the variable should not be modified and have the compiler enforce this behaviour. This can help prevent bugs and make the code easier to understand. As a general rule, it's good practice to declare variables as `const` whenever possible.

6.2.1 Example Const Use Case

Consider the code snippet in Listing6. The variable `UART_RECEIVE_REGISTER_READ_ONLY` is declared as `const`, meaning its value cannot be modified through the program (a compiler error will be thrown). However, the value at the memory address `0x12345678` can still change due to external events, such as hardware (e.g., UART) writing to it[10], which is why `const` does not mean *constant*.

```
#include <stdint.h>

volatile const uint8_t* const UART_RECEIVE_REGISTER_READ_ONLY = (uint32_t*)0
    x12345678;

int main() {
    uint8_t uart_receive_char = *UART_RECEIVE_REGISTER_READ_ONLY; // Valid C
        code!
    *UART_RECEIVE_REGISTER_READ_ONLY = 0xEF; // Compiler error: assignment of
        read-only location
    return 0;
}
```

Listing 6: Example use case of Const

6.3 Static

The **static** keyword has different meanings depending on the context in which it is used.

6.3.1 Static Functions

When used with functions, the `static` keyword limits the function's scope to the file in which it is defined (as all functions are implicitly declared as extern without the `static` qualifier [6]). This means that the function cannot be accessed by other files through linking. This is useful for helper functions that are only used within a single file and should not be exposed to other files, in effect creating a private function. Listing 7 shows an example of a static function.

```
static void helper_function() {
    // Function implementation. This function can only be accessed within the
        file it is defined in.
}
```

[10] It's also possible to bypass the protection of const, both intentionally and inadvertantly.

Listing 7: Example of a Static Function

6.3.2 Static Declarations

When used with global variables, the `static` keyword limits the variable's scope to the file in which it is defined [6]. This means that the variable cannot be accessed by other files through linking. This is useful for creating private global variables that are only accessible within a single file. Ex: declaring `static uint8_t counter = 0U;` in a file means `counter` is accessible only within that file, and no where else. These variables are stored in the data segment, or Block Starting Symbol (BSS) segment if uninitialized, and retain their value throughout the program's execution.[11]

6.3.3 Static Local Variables

When used with local variables within functions, the `static` keyword changes the variable's storage class to static. This means that the variable is stored in the data segment rather than the stack (where temporary data and variables are stored [7]), and its value is retained between function calls. This is useful when you want a variable to retain its value between function calls. Listing 8 shows an example of a static local variable.

```c
#include <stdint.h>
#include <stdio.h>

void some_function(void) {
  static uint32_t counter = 0; // Static local variable
  printf("%u ", counter);
  counter++;
}

int main(void) {
  for (uint32_t i = 0; i < 5; i++) {
    some_function();
  }
  // prints: 0 1 2 3 4
  return 0;
}
```

Listing 8: Static Local Variable

[11] For more information, see this FAQ entry on static variables.

7 What type of signal would be best for transferring data from a sensor located 1 meter away to a microcontroller?

The sensor is a temperature sensor and readings are needed at a rate of 100 Hz, with a precision of 0.1C and a temperature range of -40C to 125C.

——— Answers Ahead ———

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

7.1 Digital Signalling

Digital signals represent data using discrete values. Digital protocols describe the rules in which digital signals can be used to transfer data. Important attributes of these protocols include:

- **Differential vs. Single Ended:** Single ended signals use a single wire and a reference ground to transmit data, while differential signals use two complementary wires (and optionally also use a reference ground). Differential signalling does offers increased noise immunity for the same logic level voltages.
- **Network Topology:** Different network topologies connect devices in different ways. Point-to-point connections are the simplest, but more complex topologies, such as a *Bus* allow for multiple devices to communicate over the same signal lines.
- **Full vs. Half Duplex:** *Duplex* means that both devices are capable of transmitting data. Half-duplex means that only one device can transmit to another at a time, whereas full-duplex means both devices can simultaneously transmit at the same time.
- **Push Pull vs. Open Drain (Drive Type):** Push-Pull drive means that the devices are capable of pushing the signal lines high and pulling the signal lines low. This is in contrast to an open drain protocol, where the devices are only capable of pulling the signal lines low and resistors are used to pull the line high when no drivers are asserting them. Typically, a protocol with an open drain drive type will have slower speeds than a push-pull protocol[12].
- **Synchronous vs. Asynchronous**: A synchronous protocol makes use of a clock signal to ensure the sender and receiver are synchronized together. Clock signals can be on their own dedicated wire or be encoded as a part of the data. Asynchronous protocols assume that the sender's and receiver's individual clocks are sufficiently synchronized to ensure successful data reception. This assumption holds for protocols with lower data rates or those that include transmission pauses for re-alignment; however, it can introduce errors in high data rate applications.

The some common digital protocols in embedded systems are:

	Abbreviation	Name	Type	Bus	Duplex	Driver	Synchronicity	Typical Data Rate	Maximum Data Rate
	PWM	Pulse Width Modulation	Single Ended	Point to Point	Half-Duplex / Uni-directional	Push Pull	Asynchronous	50 Hz	200 Hz
	UART	Universal Asynchronous Receiver Transmitter	Single Ended	Point to Point	Full-Duplex	Push Pull	Asynchronous	115.2 kHz	921.6 kHz
	I2C	Inter-Interconnected Controller	Single Ended	Bus	Half-Duplex	Open Drain	Synchronous	400 kHz	1 MHz
	SPI	Serial Peripheral Interface	Single Ended	Bus	Full-Duplex	Push Pull	Synchronous	24 MHz	60 MHz
	CAN	Controller Area Network	Differential Pair	Bus	Half-Duplex	Open Drain	Asynchronous	1 mbps	8 mbps

Table 1: Digital Protocol Definitions

Digital signals have discrete states defined by voltage thresholds. Using smaller voltage differences between states results in lower power consumption, but offers less noise immunity. For this reason, some devices may natively support different voltage thresholds. Consequently, for them to communicate properly, logic level shifting the signal between devices may required. For low speed, single-ended signals, this can be implemented with a single transistor as shown in Figure 12 - for more complex cases, there are often level-shifting ICs available.

[12] This topic is further explained in Compare and contrast I2C *(Inter-Integrated Circuit)* and SPI *(Serial Peripheral Interface)*

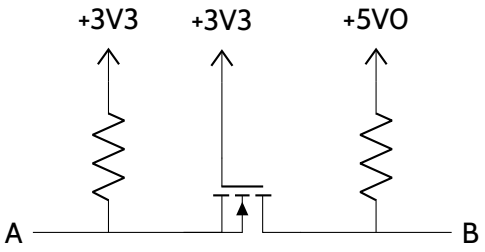


Figure 12: Single N-Channel MOSFET Logic Level Shifter

7.2 Analog Signalling

Analog signals are continuous in both time and value (as compared to digital signals, which typically only contain logical values, such as 0 or 1). While microcontrollers operate digitally, there are many reasons to rely on analog signals.

- All signals from sensors begin as "analog" values. Any IC (*integrated circuit*) that outputs a digital value is doing so because it features an on-board ADC (*Analog to Digital Converter*). Because digital signals from sensors begin as analog signals, working only with analog signals can simplify the design process.
- Analog signalling circuitry is often cheap to implement in a variety of embedded system contexts. Microcontrollers often feature internal ADC's, and may only require a few additional components to directly interface with an analog sensor.
- Simple signal processing can be performed in hardware before digital computation is required. An example of analog signal processing in hardware is first order low pass filter.
- More complex analog circuits are used in applications where higher bandwidth is needed in control loops. Analog feedback loops are very common in power electronics, but the flexibility and easy modification of digital control loops is becoming more common on highly integrated embedded systems.

A drawback of analog signalling techniques is that they are quite often less resilient when faced with noise compared to digital signalling techniques. While a digital signal is tolerant to small amounts of noise without affecting the signal transmission at all, analog signals directly realize the effects of noise. For this reason, virtually all long distance data transmission systems employ some form of digital transmission.

Differential signalling is also an option in the analog world to compensate for common-mode noise for cost optimized analog circuitry. This is uncommon however as conversion to a digital protocol is often preferred when optimizing for noise immunity.

7.3 Conclusion

For a sensor located 1 meter away from a microcontroller, a digital signal would likely be the best candidate for transferring data as it is less susceptible to noise over long distances, while likely offering a simpler implementation on the microcontroller due to the availability of on-board digital protocol peripherals. The high data-rate requirement of 100 Hz is easily achievable with most digital protocols, and the precision requirement of 0.1C is usually achievable with most digital sensors. A protocol like UART

or I2C would likely be suitable for this application.

It's important to keep in mind that when asked about the best type of (something) for a given application, the best answer is often "it depends". The interviewer is looking for you to weigh the pros and cons of different solutions and make a recommendation based on the information given. For this problem, many solutions could be valid depending on the specific circumstances of the application.

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

——— Answers Ahead ———

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

8.1 Introduction

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.

8.2 PID Controller Implementation

The form of a PID controller is given by Equation (7) [8]. 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[13]. The equation can appear daunting, but the implementation is quite straightforward.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{7}$$

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 9.

```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 9: 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 [8], 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 13 for a visual representation of the Backward Euler method.
  `pid->integral += error * dt;.`
  `const float i_term = pid->K_i * pid->integral;.`

[13] 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.

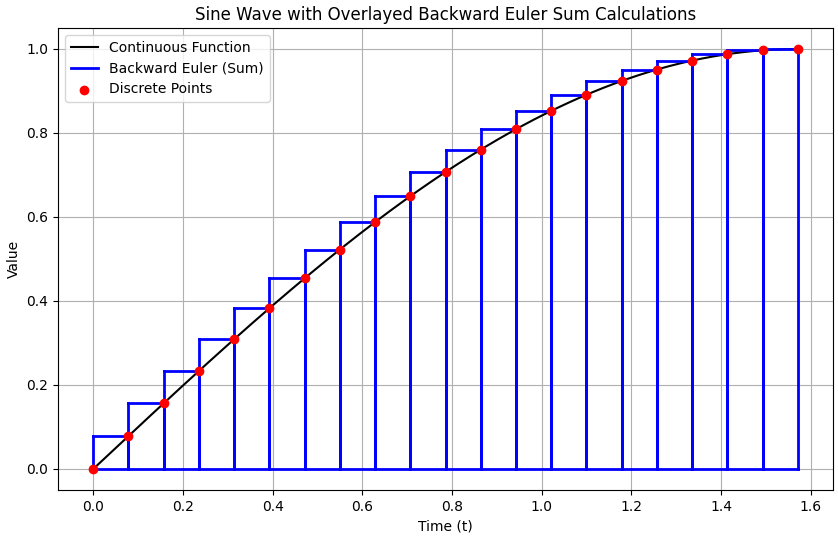


Figure 13: Backward Euler Discretization of the Integral Term

- 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 ($\text{slope} = \frac{\text{rise}}{\text{run}}$). See Figure 14 for a visual representation of the Backward Euler method.
  `const float d_term = pid->K_d * ((error - pid->prev_error) / dt);`.

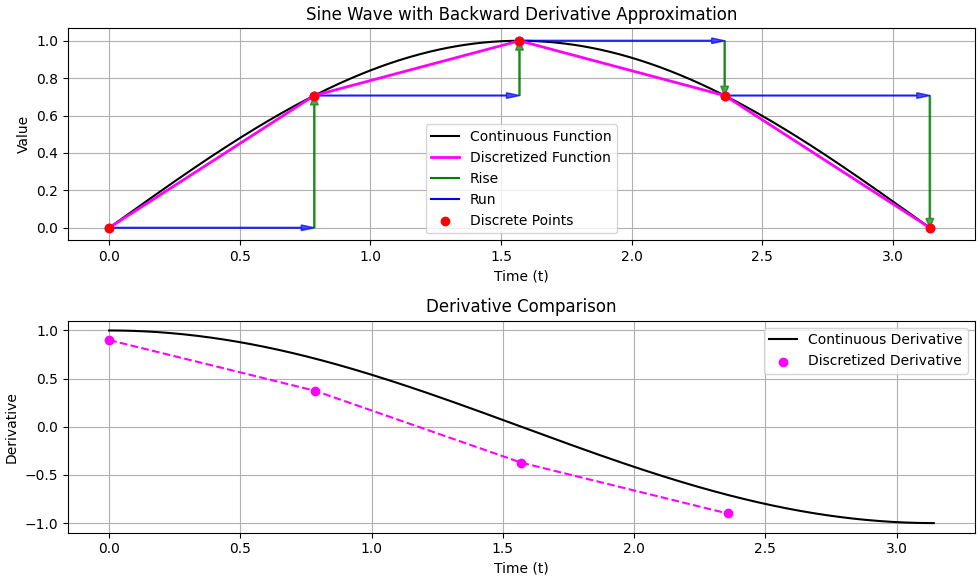


Figure 14: Backward Euler Discretization of the Derivative Term

Putting it all together, the PID controller step function is shown in Listing 10. Note the added check for a `NULL` pointer and positive timestep as either can cause the function to not run correctly.

```
#include "pid_typedef.h"
```

```c
#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 10: PID Controller Step

8.3 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?

9 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.

9.1 I2C

I2C is a *half-duplex* (can either transmit or receive, but not both simultaneously) digital protocol developed by Phillips in 1982 [9]. It enables a host device[14] (referred to as a *master*[15]) to communicate with multiple peripheral devices (referred to as *slaves*) over a two-wire serial bus.

9.1.1 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 15.

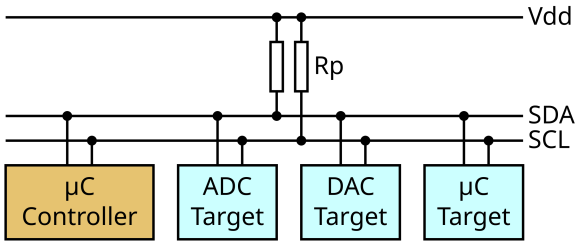


Figure 15: I2C Bus Diagram, Source: Wikipedia [10]

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.

9.1.2 Data Format

The data format of I2C features the following components, and is shown graphically in Figure 16:

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.
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.

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

[15] 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.

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.

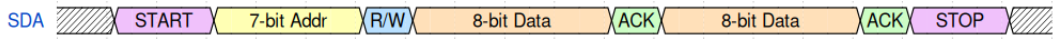


Figure 16: 2 Byte I2C Data Frame Format

9.2 SPI

SPI was developed by Motorola [11] 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 17.

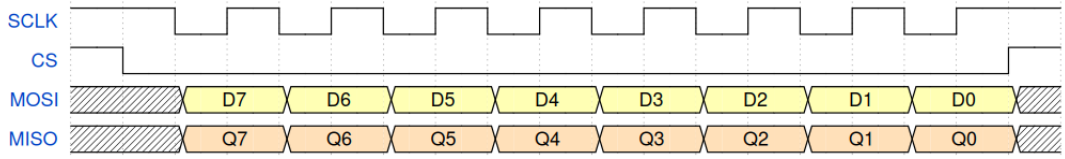


Figure 17: 1-byte SPI Timing Diagram

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 18 (Note the diagram in Figure 18 uses *SS* for *Slave Select*, which is synonymous with *CS*).

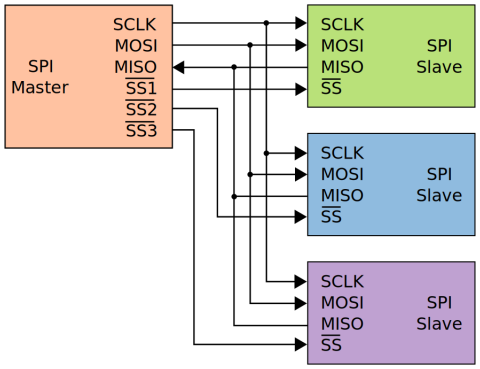


Figure 18: SPI Bus Diagram, Source: Wikipedia [12]

9.3 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.

9.4 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?

10 Implementing a bit-bang'ed SPI master.

10.1 Given Code

Assume the header shown in Listing11 is available for use in the bit-banging implementation of SPI. The transceive function should use CPOL = 1, CPHA = 1.

```c
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

typedef enum { PIN_NUM_CS, PIN_NUM_SCLK, PIN_NUM_MISO, PIN_NUM_MOSI }
    spi_pin_E;

/** Externally Provided Functions - these are assumed to be error free **/
void HAL_GPIO_write(spi_pin_E pin, bool value);
bool HAL_GPIO_read(spi_pin_E pin); // Returns the logical value of the pin
void delay_nanoseconds(uint32_t ns);

/** USER IMPLEMENTED FUNCTIONS **/
/**
 * @brief Transmits and receives data over SPI using bit-banging
 * @param clk_freq_hz The frequency of the SPI clock in Hz
 * @param tx_data Pointer to the data to be transmitted
 * @param rx_data Pointer to the buffer to store the received data
 * @param len_bytes The number of bytes to transmit and receive. The length of
 * tx_data and rx_data must be at least len
 * @return true if the transaction was successful, false otherwise
 */
bool bitbang_spi_transceive(float clk_freq_hz, uint8_t const *const tx_data,
                            uint8_t *const rx_data, size_t len_bytes);
```

Listing 11: Bit Bang HAL Header

——— Answers Ahead ———

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

10.2 Bit-Banging Basics

In embedded systems, communication protocols like SPI are typically handled by dedicated hardware peripherals. These peripherals manage the precise timing and fast data transmission required to communicate over physical wires. However, it is also possible to implement these protocols purely in software, a technique commonly known as *bit-banging*. Bit-banging can be useful in the following scenarios:

- When a hardware peripheral (ex: SPI/I2C/UART) is unavailable on the microcontroller, usually because it's not supported, or the EE (*Electrical Engineer*) gave the pins a 'creative reassignment' (accidentally routed incorrect pins during schematic capture).
- When a custom or non-standard protocol is required, which cannot be supported by existing hardware peripherals.

Bit-banging is usually implemented by manually controlling the individual pins of a microcontroller in software to emulate a hardware peripheral [13]. While this approach is slower and uses more CPU resources, it provides flexibility for situations where hardware support is limited. Listing 12 shows an example of a square wave generator implemented using bit-banging. Note the use of `delayMicroseconds()` to control the timing of the square wave, which blocks the CPU until the desired time has passed, contributing to the inefficiency of bit-banging.

```
#include "bitbang_ex.h"

void square_wave(uint8_t pin, uint8_t cycles, uint8_t period_us) {
  for (uint8_t i = 0; i < cycles; i++) {
    HAL_GPIO_WritePin(pin, HIGH);
    delayMicroseconds(period_us / 2);
    HAL_GPIO_WritePin(pin, LOW);
    delayMicroseconds(period_us / 2);
  }
}
```

Listing 12: Bit-Banged Square Wave Implementation

10.3 SPI Review

Consider the SPI timing diagram in Figure 19. The SPI protocol consists of 4 signals: MOSI (Master Out Slave In), MISO (Master In Slave Out), SCLK (Serial Clock), and CS (Chip Select). The master device controls the clock signal and selects the device using the CS signal; data is transmitted on MOSI and received on MISO simultaneously on predefined edges of the clock signal.

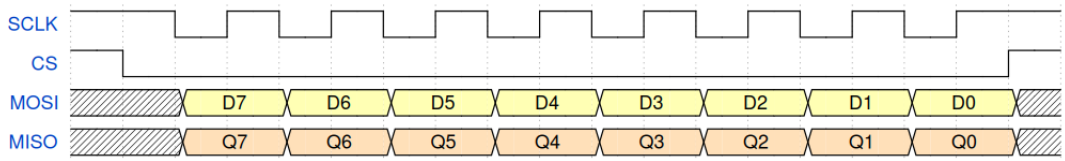


Figure 19: 1-byte SPI Timing Diagram

Note that the SPI protocol can be configured in different clock sampling modes, which define the clock polarity (CPOL) and phase (CPHA). Figure 17 shows CPOL = 1, CPHA = 1. A SPI slave device's datasheet will usually specify the required CPOL and CPHA settings for proper communication real-world implementation.

- **CPOL (Clock Polarity)**: Determines the idle state of the clock signal. CPOL = 0 means the clock is low when idle, while CPOL = 1 means the clock is high when idle.
- **CPHA (Clock Phase)**: Determines when data is sampled and changed. CPHA = 0 means data is sampled on the leading edge (first transition) of the clock, while CPHA = 1 means data is sampled on the trailing edge (second transition) of the clock.

10.4 Bit-banging SPI Implementation

Going through the timing diagram in Figure 17 piece by piece, we can implement the SPI protocol in software. The key parts are as follows:

- Pull the CS line low to select the slave device. Wait for a brief period (referred to as *setup time*).
- Generate a square wave on the SCLK line, ensuring the correct CPOL and CPHA settings.
- Loop through each bit of the tx byte starting from the most significant bit (MSB), and transmit the logical value of the bit on the MOSI line on every SCLK falling edge.
- In parallel, read the MISO line on the rising edge of SCLK line to receive the slave's response, and write it to the corresponding bit in the received byte buffer.
- Repeat for the specified number of bytes.
- Terminate the SPI transaction by pulling the CS and SCLK line high to deselect the slave device.

Listing 13 shows a basic implementation of a bit-banged SPI master transceive.

```
#include "bitbang_spi_header.h"

#define NUM_BITS_IN_BYTE 8
#define NS_PER_SECOND 1000000000

bool bitbang_spi_transceive(float clk_freq_hz, uint8_t const *const tx_data,
                            uint8_t *const rx_data, size_t len_bytes) {
  const bool ret = (tx_data != NULL) && (rx_data != NULL) && (len_bytes > 0);

  if (ret) {
    // clock calcs - integer division is used, but error is acceptable
    const uint32_t period_ns = NS_PER_SECOND / clk_freq_hz;
    const uint32_t half_period_ns = period_ns / 2;

    HAL_GPIO_write(PIN_NUM_SCLK, true);
    HAL_GPIO_write(PIN_NUM_CS, false);
    delay_nanoseconds(period_ns); // CS setup time, arbitrary value

    for (uint32_t byte = 0; byte < len_bytes; byte++) {
      const uint8_t tx_byte = tx_data[byte];
      for (uint32_t bit = 0; bit < NUM_BITS_IN_BYTE; bit++) {
        // falling edge - write MOSI output here
        HAL_GPIO_write(PIN_NUM_SCLK, false);
        const uint8_t tx_bit = (tx_byte >> (7 - bit)) & 0x01;
        HAL_GPIO_write(PIN_NUM_MOSI, (bool)tx_bit);
        delay_nanoseconds(half_period_ns);
        // rising edge - read MISO input here
```

```
28          HAL_GPIO_write(PIN_NUM_SCLK, true);
29          delay_nanoseconds(half_period_ns);
30          rx_data[byte] |= HAL_GPIO_read(PIN_NUM_MISO) << (7 - bit);
31        }
32      }
33
34      HAL_GPIO_write(PIN_NUM_CS, true);
35    }
36
37    return ret;
38  }
```

Listing 13: Bit Bang SPI Implementation

11 Describe how to use an oscilloscope to measure a signal.

Include the following in your response:

- How does the trigger function?
- Describe bandwidth considerations.
- When the use of probe multiplication is warranted?

——— Answers Ahead ———

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

11.1 Oscilloscopes

An oscilloscope (in shorthand, just *scope*) is a device that plots the voltage of a probed signal with respect to time. It does so by utilizing an ADC (*Analog to Digital Converter*) to discretize a the signal into quantized samples that can be plotted on a monitor. As a result, the oscilloscope can perform many digital logic operations on the probed signal and offers a variety of features for engineers (such as logic analysis, protocol decryption, and frequency domain analysis).

11.2 Probes

The most basic oscilloscope probes have a ground clip and probe hook connected through a coaxial cable to a BNC connector that connects to the oscilloscope. These probes are designed to have a relatively high impedance and affect the circuit under test minimally[16].

11.2.1 Probe Multipliers

Often, it's desired to measure voltages outside of the input voltage range of the oscilloscope. As a result, probes often have a "probe multiplication" setting (commonly, a 1x or 10x switch) that scales the voltage being probed to a safe level before it enters the oscilloscope. For example, a 10x probe will scale the voltage by a factor of 10 before it enters the oscilloscope. Another advantage of using a higher probe multiplication is that the bandwidth of the probe is increased and the capacitance the probe adds to the circuit is decreased[17].

11.2.2 Other Types of Probes

Oscilloscope probes typically share a common ground (which is often attached by a clip), which is tied to earth ground for safety by the oscilloscope. This complicates differential voltage measurements, such as across a resistor not connected to ground. Using single-ended probes, two probes are needed—one on each side of the resistor—with the oscilloscope calculating the difference. Alternatively, a differential probe simplifies this by directly outputting the voltage difference between 2 arbitrary probed signals, saving an oscilloscope channel.

A current clamp is another type of probe that can be used to measure the current going through a wire - it works by outputting a voltage that is proportional to the current going through the wire. When a current clamp is connected to an oscilloscope, the oscilloscope can mathematically convert the voltage output by the current clamp to a current measurement. Current clamps work on the principle of magnetic current sensing and have an active amplifier in them.

11.3 Triggering

Since the signals being analyzed typically operate at frequencies much higher than humans can process (e.g., in the MHz range), oscilloscopes provide a 'trigger' function that establishes a $t = 0$ s reference point based on a specific signal event.

[16] In reality, probes often specify the capacitance the probe will add when it is connected to a circuit. This capacitance may have a noticeable impact on the circuit, especially at higher frequencies.

[17] This is because the step down is performed in the probe tip so a lower voltage (and therefore less energy) signal propagates in the coaxial cable and into the scope. This phenomenon is often described as the probe having less effective/apparent capacitance as it pulls less current from the circuit under test.

The most common form of trigger is known as edge triggering. When rising edge trigger mode is used, the oscilloscope begins a displayed image at $t = 0s$ every time the sampled signal rises across a voltage threshold. Other trigger types occur, specifically for digital protocols, to trigger on more complex conditions such as a specific series of bytes in a digital protocol. Untriggered waveforms can appear inconsistent, as illustrated in Figure 20, where the sampled waveform appears highly distorted. [18]

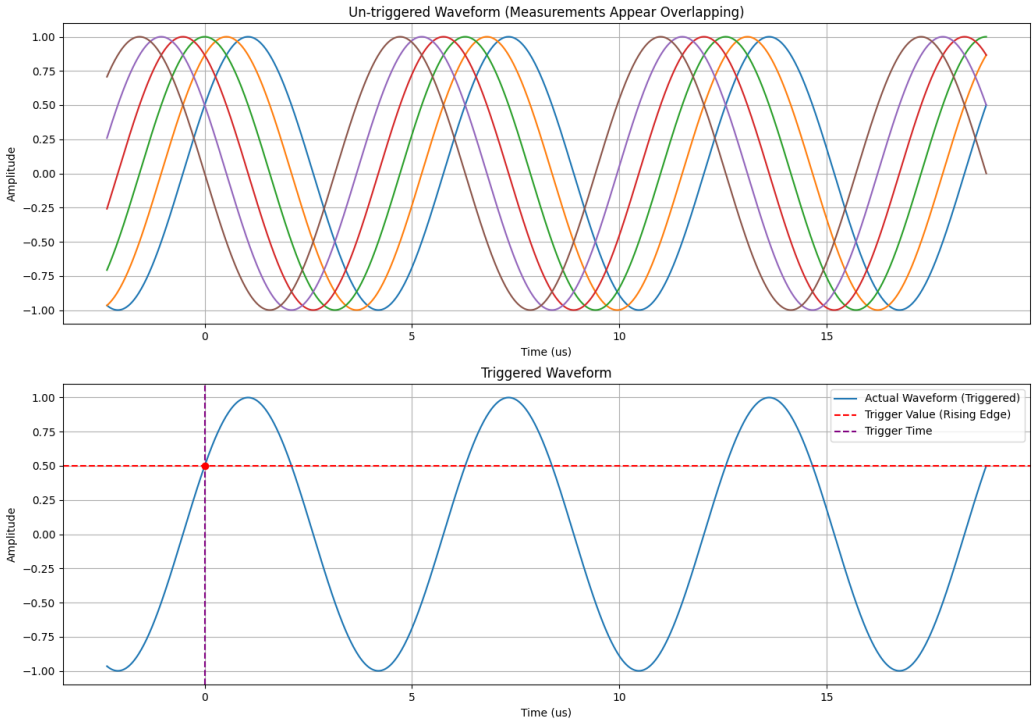


Figure 20: Impact of Triggering on an Oscilloscope Display

11.4 Bandwidth

The bandwidth of a low pass filter is defined by the frequency in which half the power, or $\frac{1}{\sqrt{2}}$ of the voltage, of an input signal passes to the output signal. [19] This means if you probe a 10 MHz signal with an oscilloscope with 10 MHz bandwidth, you will be seeing $\frac{1}{\sqrt{2}}$ of the voltage! A common rule of thumb used to ensure accurate measurements is that the oscilloscope bandwidth should a few times higher than the highest frequency needing to be measured. Oscilloscopes and measurement probes each have their own bandwidth specifications - the lower of these values should be used as the bandwidth for comparison.

[18] Another parameter of the trigger is the mode. In normal mode the oscilloscope will only trigger when a trigger event occurs. In auto mode the oscilloscope will automatically trigger if a trigger event has not occurred for some period of time. The oscilloscope can also be stopped which means no trigger events are allowed meaning the display does not update.

[19] In the context of wide-band measurements and oscilloscopes, the bandwidth of the circuits is always discussed in this regard though broader definitions of bandwidth are used in analyzing more complex circuitry.

11.5 DMM

Digital multi-meters (*DMMs*) are also capable of sampling voltage with respect to time, however, their sample rate and bandwidth is significantly lower than that of an oscilloscope. However, DMMs feature circuitry to determine the root-mean square of a signal which allows the measurement of AC signal amplitude, DC signals, and PWM duty cycle.

11.6 Follow-ups

- What are the implications of oscilloscope sample rate?
- How would you verify a scope probe is functioning correctly?
- Why do simple scope probes need to be tuned?

12 Briefly describe *Controller Area Network* (CAN)?

——— Answers Ahead ———

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

12.1 Summary

Controller Area Network (CAN) is a popular communication protocol developed by Bosch in the 1980's [14]. It's widely used in the automotive and robotics industries due to its robustness, reliability, and tailor-made features for automotive-type environments.

Note that CAN is a fairly complex protocol, and depending on the interviewee's knowledge level about the protocol, can cause an answer to this question to get quite deep and involved. However, this answer describes some of the primary features of CAN that are integral to its operation, as many of those features are asked as standalone questions or elaborated upon in an interview setting.

12.2 CAN Physical Layer

The CAN standard defines the physical and data link layers of the OSI model. The physical layer uses a **differential pair** of wires (*CAN_H and CAN_L*) to transmit data, as opposed to a single wire. It is an open-drain bus, meaning that the bus is pulled to a dominant state (logical 0) by a node asserting a dominant state (pulling `CAN_H` to 2V above `CAN_L`), while a recessive state (logical 1) is achieved by no node asserting a dominant state (`CAN_H` and `CAN_L` at the same potential voltage) - this is shown in Figure 21. Note that CAN requires a transceiver to output the differential signal. Commonly, microcontrollers with a CAN peripheral communicate over a serial interface (an interface where transmission occurs sequentially, byte-by-byte) with two single ended signals, *CAN_TX* and *CAN_RX*, to a transceiver IC (*Integrated Circuit*) which drives and receives the differential signals on the CAN bus.

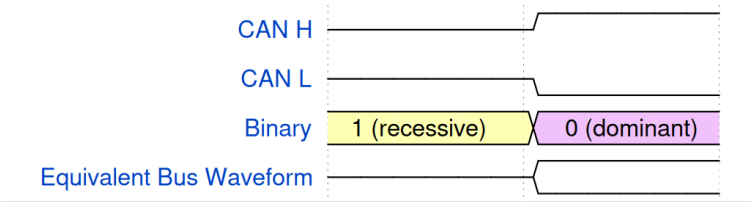


Figure 21: Recessive and dominant bits waveform [15]

12.2.1 Differential Pair

A differential pair works by creating two signals: a positive signal, and a negative signal (which is the logical inverse of the positive signal). Both signals are transmitted simultaneously to the receiver, which can then subtract the two signals to recover the original data. A schematic implementation of a differential receiver is shown in Figure 22, with a sample digital message shown in Figure 23. [20].

[20] In the case of CAN, the dominant state (0) is when the CAN_H signal is higher than the CAN_L signal, and the recessive state is when the CAN_H and CAN_L signals are at the same voltage level. However, differential signals are often implemented with a different convention, where a logical 1 is signaled when $V_{positive} > V_{negative}$, and a logical 0 is signaled when $V_{negative} > V_{positive}$

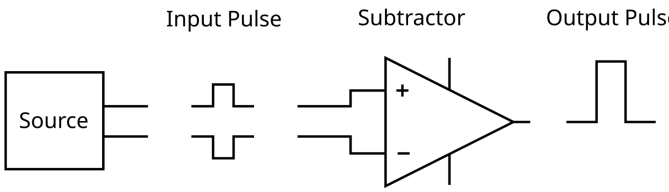


Figure 22: Differential Pair Physical Layer, Source: Wikipedia [16]

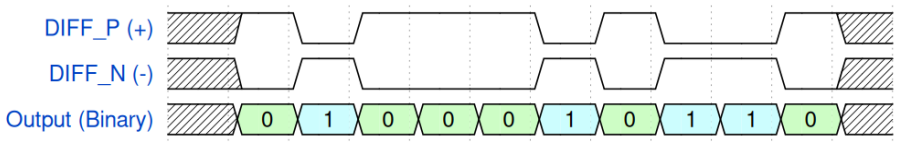


Figure 23: Digital Differential Signal Sample

An advantage of using a differential pair is that it reduces the impact of EMI (*Electromagnetic Interference*) on the signal, as any interference will affect both wires equally. This allows the receiver to subtract the two signals, effectively removing the interference. The schematic in Figure 24 shows how the receiver can subtract the two signals to remove noise.

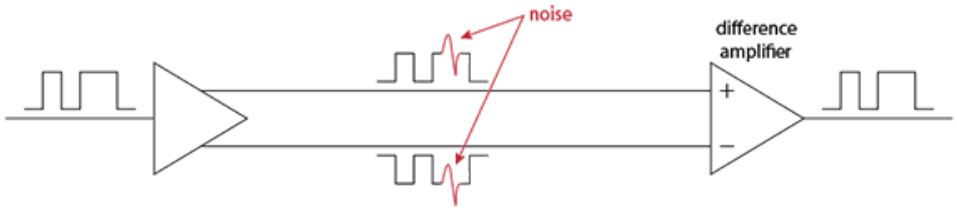


Figure 24: Differential Pair Physical Layer with Noise, Source: Stack Overflow [17]

Another advantage of this configuration is immunity to common mode voltage offsets. Ideally, CAN transceivers are designed to output a dominant state with `CAN_H` at 3.5V and `CAN_L` at 1.5V, and a recessive state with `CAN_H` and `CAN_L` both at 2.5V. However, an advantage of differential signalling is that these values can be offset (referred to as *voltage shifting*) if two transceivers have a voltage potential difference between their ground planes. More accurately, we can define a dominant state as `CAN_H` at $3.5 + N$ V and `CAN_L` at $1.5 + N$ V, and a recessive state as `CAN_H` and `CAN_L` both at $2.5 + N$ V, where N is a common mode voltage offset.[21]

12.2.2 Causes of Noise in a CAN Network

Voltage differences in ground potential between devices are common in embedded systems with high current power transmission (ex: applications involving motor control - a potential difference between the ground of the controller and the ground of a motor controller board can arise due to the high current going to the motor, as explained by Ohm's law $V = I \cdot R$).

[21] Note that there is a limit to how large N can be, depending on the specific CAN transceivers being used.

12.2.3 Twisted Pair Wiring

CAN wires are often twisted as the current flow in the signal lines opposes each other when changing state. The rationale for wanting current flow to oppose each other is to make the magnetic fields induced by current flowing in the signal lines cancel each other out, as the magnetic field will be in opposite directions. Twisting wires also allows some control over the characteristic impedance. [22]

12.3 Bus Topology

CAN uses a **multi-master** communication scheme, meaning that any node on the bus can initiate a message transmission. This is in contrast to a **master-slave** communication scheme, where only the master can initiate communication. They are connected in a **bus topology**, where all nodes are electrically [23] connected in parallel. Essentially, every node on the bus can see every message transmitted on the bus, and every node has the ability to initiate message transmission. Note that the CAN bus is *asynchronous*, meaning every node must know the bus *bitrate* beforehand to understand transmissions from other nodes.

The bus is terminated with 120 ohm resistance at both ends to prevent signal reflections in the transmission line and to ensure that a recessive bus state can be caused when no device is asserting a dominant state. A schematic of a simple bus topology is shown in Figure 25 with a termination at each end of the bus and a small branch length. CAN is a very resilient protocol, especially for lower data-rates, so optimizing electrically is often unnecessary for simple/small busses as long as at least one termination is present. For production applications guidelines for bus topologies, terminations, and termination values are given in standards, but are often optimized by testing from EMC (*Electro-Magnetic Compliance*) engineers in the lab.

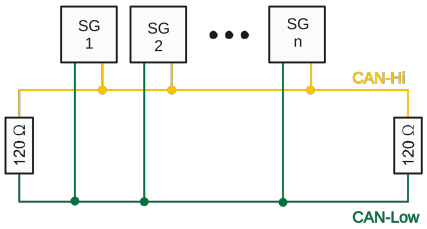


Figure 25: CAN Bus Topology, Source: Wikipedia [18]

12.4 Message Structure

A CAN message consists of a variety of fields, of which the most important are:

- **Arbitration Field/Message identifier:** This field contains the message identifier, which is used to determine message priority. The lower the value of the identifier, the higher the priority of the message.
- **Control Field:** This field contains information about the message, such as the message length.
- **Data Field:** This field contains the actual data being transmitted. In CAN 2.0, this field can contain up to 8 bytes of data.

[22] Note that CAN is very forgiving and outside of production applications and more complex bus topologies, twisting CAN wires is unnecessary for successful data transmission.

[23] Physically, they may be connected differently, however, this answer only focuses on the electrical connection.

- **CRC Field:** This field contains a cyclic redundancy check (CRC) to ensure the message's data integrity.
- **Acknowledgement Field:** This field is used to acknowledge the receipt of a message.

Typically, a microcontroller will have a CAN peripheral that handles the low-level details of the CAN protocol, such as message transmission, reception, error detection, etc. The microcontroller's firmware will interact with the CAN peripheral to send and receive messages, as well as configure key parameters such as the bitrate, message filters (only receive certain message ID's), etc.

12.4.1 CAN Arbitration

CAN features a unique, *non-destructive* (without data loss), *bit-wise* arbitration mechanism, based on the message identifier. Due to the open-drain nature of the bus, a dominant bit (0) will always override a recessive bit (1). When two nodes start transmitting a message identifier, they constantly monitor the bus to see if the message identifier bit they are transmitting is the same as the message identifier bit on the bus. If a node sees that the message identifier it is transmitting is different from the message identifier on the bus, it will stop transmitting and wait for the bus to become idle before trying to transmit again. Because of this, the node with the lowest message identifier will always win the arbitration and be able to transmit its message (and thus, lower message ID's have higher priority). An example of CAN arbitration in action is shown in Figure 26.

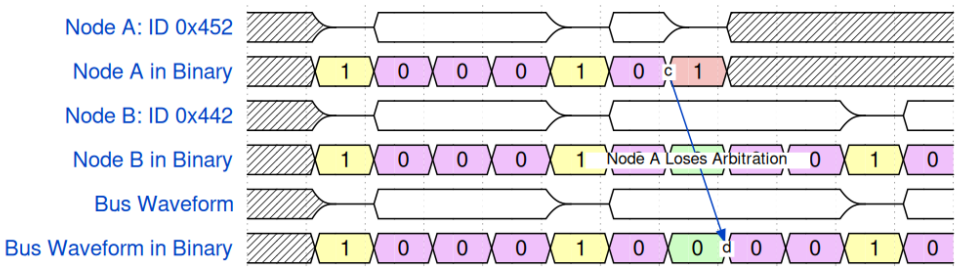


Figure 26: CAN Arbitration Sequence

The CAN arbitration mechanism is one of the key features that make CAN a robust and reliable communication protocol. It ensures that the node with the highest priority message will always be able to transmit its message, even in the presence of multiple nodes trying to transmit messages simultaneously. However, care should be used with low message identifiers, as they can starve (or *step on*) higher message identifiers from transmitting.

12.5 Follow-ups

- Explain how CAN can detect TX errors using TX/RX loopback?
- How would you debug a CAN bus that is not working?
- What is 'Time Quanta' in CAN?

13 Determine the step response of the following circuits.

State any assumptions regarding component values.

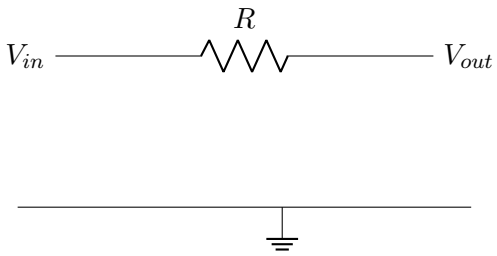


Figure 27: Circuit A

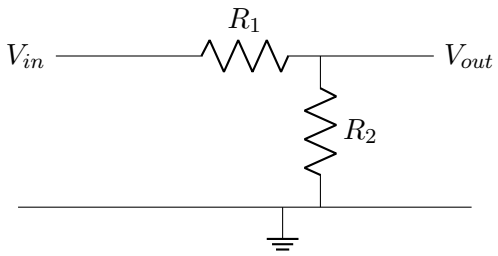


Figure 28: Circuit B

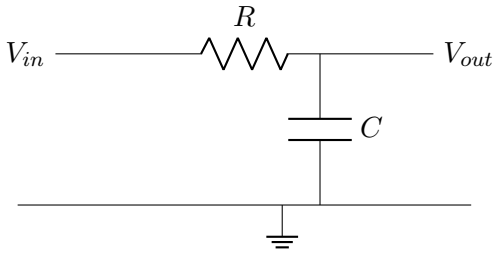


Figure 29: Circuit C

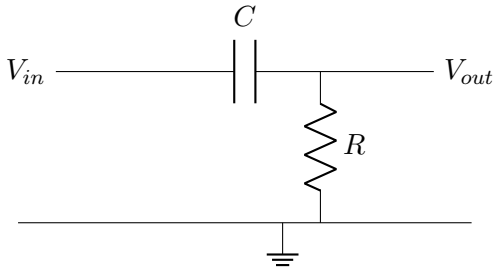


Figure 30: Circuit D

——— Answers Ahead ———

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

13.1 Passives

For a question like this, assuming ideal components is expected as further information has not been provided to aid in a more complex analysis. In a real interview, it's likely you would be given just one of these circuits. Table 2 provides a summary of the key properties of ideal resistors, capacitors, and inductors.

	Resistor	Capacitor	Inductor
Time Domain	$V = I \cdot R$	$I = C \cdot \frac{dv}{dt}$	$V = L \cdot \frac{di}{dt}$
Frequency Domain	$Z = R$	$Z = \frac{1}{j \cdot \omega \cdot C}$	$Z = j \cdot \omega \cdot L$
As $f \to \infty$	$Z = R$	$Z \to 0$	$Z \to \infty$
As $f \to 0$	$Z = R$	$Z \to \infty$	$Z \to 0$
Energy Stored	$E = 0$	$E = \frac{1}{2} \cdot C \cdot V^2$	$E = \frac{1}{2} \cdot L \cdot I^2$
Real Power Dissipation	$P = I \cdot V$	$P = 0$	$P = 0$

Table 2: Passive Element Definitions

Keep in mind when interpreting table 2 that:

- The relationship between angular frequency and frequency is given by: $\omega = 2\pi f$.
- Impedance, Z, is frequency dependent for circuits with inductors and capacitors.
- $Z = \frac{V}{I}$ represents the impedance of a circuit.
- The imaginary component of impedance, Z, represents reactance X which defines a phase shift between V and I.
- Resistors have no frequency dependence and do not store any energy though they do dissipate power, $P = I \cdot V = I^2 \cdot R = \frac{V^2}{R}$.
- An open circuit (nets that are completely isolated from each other) is represented as connected by an impedance of value $Z = \infty$.
- A short circuit (nets that are at the same voltage potential) is represented as connected by an impedance of value $Z = 0$.
- For a series connection of impedances Z_1, Z_2, ..., Z_n, the equivalent impedance is given by $Z = Z_1 + Z_2 + \cdots + Z_n$.
- For a parallel connection of impedances Z_1, Z_2, ..., Z_n, the equivalent impedance is calculated as $\frac{1}{Z} = \frac{1}{Z_1} + \frac{1}{Z_2} + \cdots + \frac{1}{Z_n}$.
- For two impedances in parallel, Z_1 and Z_2, the formula $\frac{1}{Z} = \frac{1}{Z_1} + \frac{1}{Z_2}$ can be simplified algebraically to $Z = \frac{Z_1 \cdot Z_2}{Z_1 + Z_2}$.

13.2 Step Response

The step response of a circuit is what the output waveform with respect to time looks like given a unit step is applied to the input. Traditionally, this means plotting V_{out} waveform given

$$V_{in}(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0 \end{cases}$$

13.3 Circuit A: Series Resistor

In this circuit we note Ohm's law, $V = I \cdot R$, however, as there is no load, $I = 0$ on the resistor in this circuit we note there is no voltage drop, $V = I \cdot R = 0 \cdot R = 0$, across the resistor, consequently $V_{in} = V_{out}$ for this circuit.

Note for this circuit, and for all subsequent circuits as well, an assumption is made that there is no extra loading on V_{out} and there is no source impedance on V_{in}. This is reasonable as the question shows no extra loading drawn and the question does not state otherwise.

13.4 Circuit B: Voltage Divider

This circuit is a voltage divider depicted slightly differently than in a previous question[24], but the same equations hold: $\frac{V_{out}}{V_{in}} = \frac{R_2}{R_1 + R_2}$. In this case, to produce a step response plot an assumption is required about the values of R_1 and R_2. For simplicity, it is assumed that $R_1 = R_2$, resulting in the following transfer function: $\frac{V_{out}}{V_{in}} = 1/2$.

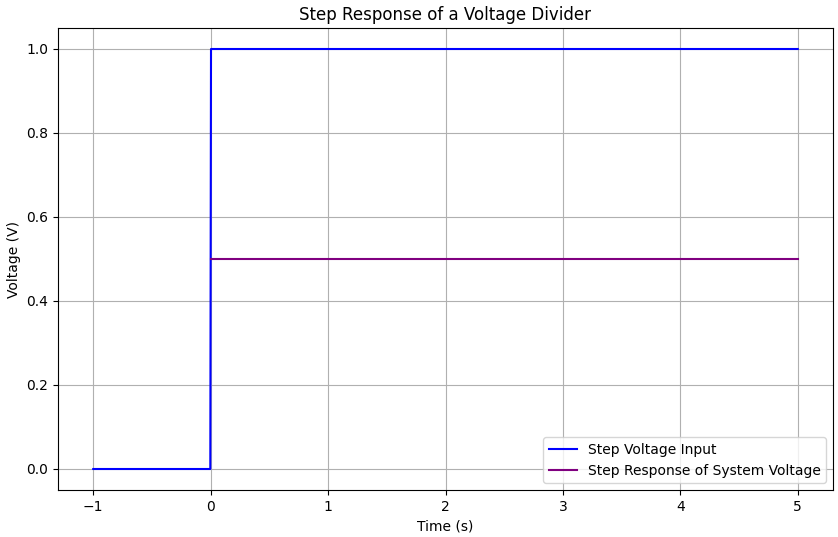


Figure 31: Step Response of a Voltage Divider Circuit

[24] Question: 3

13.5 Circuit C: RC Low Pass Filter

Circuits with inductors and capacitors are dependent on time and frequency and can be analyzed in either domain. For the circuit in Figure 29 analysis will be performed in the time domain.

Analyzing the two elements in the circuit in time domain shows $I = C \cdot \frac{dV_{out}}{dt}$ and $V_{in} - V_{out} = I \cdot R$. As I is equivalent in both elements, it can be cancelled out when solving the system of equations, resulting in $V_{in} - V_{out} = R \cdot C \cdot \frac{dV_{out}}{dt}$. This expression can be algebraically manipulated into $-\frac{dV_{out}}{V_{out} - V_{in}} = -\frac{dt}{R \cdot C}$. Integrating both sides, the expression becomes $\ln \frac{V_{out} - V_{in}}{V_{out}} = -\frac{t}{R \cdot C}$.

After re-arranging the above expression, the solution to the differential equation is given by Equation 8.

$$\frac{V_{out}}{V_{in}} = 1 - e^{\left(\frac{-t}{R \cdot C}\right)} \tag{8}$$

Note when analyzing this circuit, a constant $\tau = R \cdot C$ is defined as the time constant of the RC circuit which represents the time it takes $1 - e^{-1}$ or 63 percent of the step size. As the question allows assumptions to be made for component values, a logical assumption is to select R and C that $\tau = R \cdot C = 1$ for simplicity as the transfer function becomes $\frac{V_{out}}{V_{in}} = 1 - e^{-t}$.

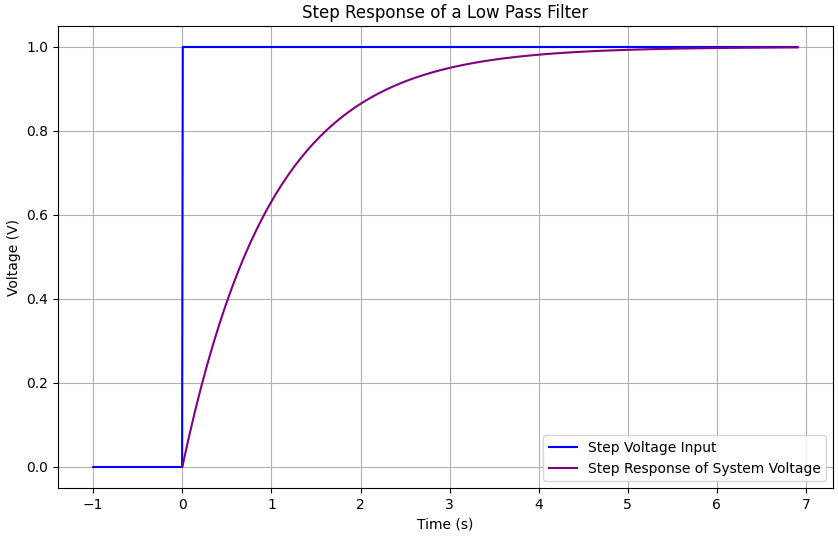


Figure 32: Step Response of a Low Pass Filter Circuit

13.5.1 Connection to I2C

The low-pass filter circuit is extremely fundamental and shows up in a variety of situations. Consider an I2C digital signal in which all drivers are not asserting the line low - in this case, the signal trace has some parasitic capacitance to ground and a pull-up resistor is responsible for pulling the line to a logic high state. This can be modelled as an RC low-pass filter circuit - consequently, an I2C signal (such as the one shown in Figure 33, SCL line) will have a similar step response to the low-pass filter circuit.

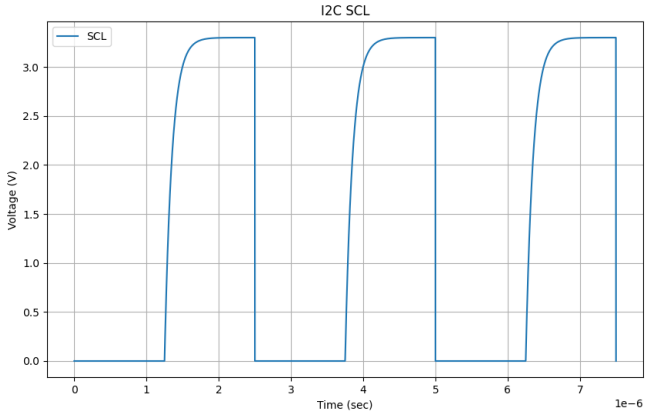


Figure 33: I2C SCL Rise/Fall

Increasing the value of R_{pullup} results in less power consumption, but slower rise times as $\tau = R \cdot C$. There is often a lower limit to the value of R_{pullup} as drivers must be rated for the current to pull the line low. Selecting the value of R_{pullup} is a careful consideration for design engineers with values commonly ranging from $1\text{k}\Omega$ to $10\text{k}\Omega$.

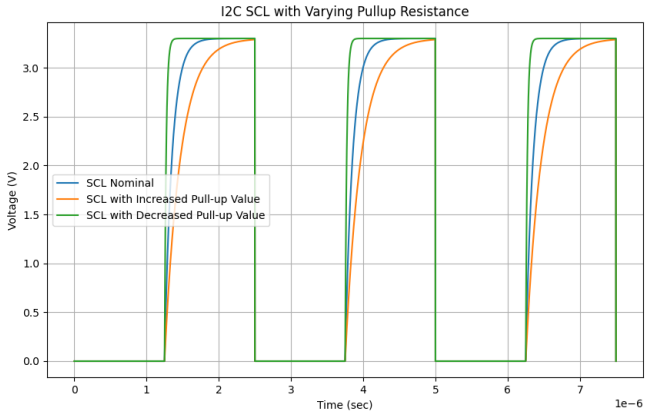


Figure 34: I2C SCL Rise/Fall with Varying Pullup Resistances

When tackling a question like this, performing a conceptual sanity check can be invaluable! Since a capacitor acts as an open circuit under DC conditions, it follows logically that as time approaches infinity, Circuit C will resemble Circuit A.

13.6 Circuit D: RC High Pass Filter

For this question, a complex impedance approach will be used rather than time domain analysis (though both will produce the correct solution). The transfer function of a voltage divider, $\frac{V_{out}}{V_{in}} = \frac{R_b}{R_t + R_b}$, can be applied to impedance values, resulting in $\frac{V_{out}}{V_{in}} = \frac{Z_b}{Z_t + Z_b}$. For this circuit $Z_b = Z_r = R$ and $Z_t = Z_c = \frac{1}{j \cdot \omega \cdot C}$ can be substituted in to get $\frac{V_{out}}{V_{in}} = \frac{R}{\frac{1}{j \cdot \omega \cdot C} + R}$.

When in the time domain $t \approx \infty$ in the frequency $\omega \approx 0$, this the transfer function becomes $\frac{V_{out}}{V_{in}} = \frac{R}{\frac{1}{j \cdot 0 \cdot C} + R} = \frac{R}{\infty + R} = \frac{1}{\infty} = 0$. Additionally, at $t = 0$, the frequency domain representation of the edge contains very high frequencies where $\omega \approx \infty$, so the transfer function becomes $\frac{V_{out}}{V_{in}} = \frac{R}{\frac{1}{j \cdot \infty \cdot C} + R} = \frac{R}{0 + R} = 1$.

Interpreting these results, we see that this circuit exhibits no gain at low or zero frequency and a gain of 1 at high frequencies. Consequently, it allows high-frequency signals to pass, which is why it is called a *high-pass filter*. A step input contains a wide range of frequency components, including high frequencies, which the filter initially allows to pass. This results in a sharp initial response, which then tapers off over time as lower-frequency components dominate. This matches the behavior observed in Figure 35.

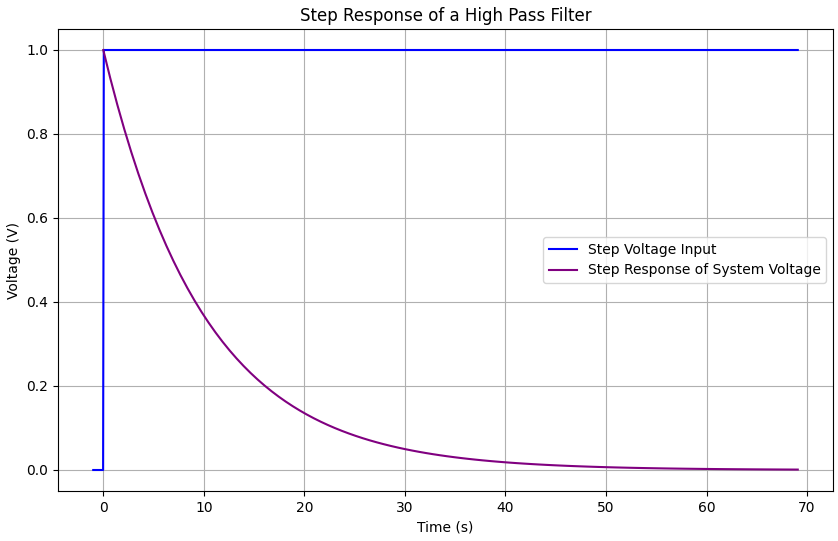


Figure 35: Step Response of a High Pass Filter Circuit

13.7 Follow-ups

- Given an unknown discrete capacitor find the capacitance? How would you determine the capacitance at a given DC bias voltage?
- For circuit C, does changing the value of R (assuming all else remains constant) change the total energy dissipated in the resistor?
- Consider the circuits given in extra practice question 24.

14 Implement a C bytestream parsing function for a weather data sensor.

Use the header in Listing14. Note that within the `weather_data_t` structure, the temperature is stored as a float with units of degrees Celsius, while the pressure is stored as a float with units of **kilopascals**. The function will only be called on a complete packet (i.e. the entire packet is received before the function is called).

```c
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

typedef struct {
  float temperature_degC;
  float pressure_kPa;
} weather_data_t;

/**
 * @brief Parse a packet of data and extract the weather data
 * @param packet The packet of data to parse
 * @param len The length of the received packet
 * @param weather_data The weather data to populate
 * @return true if the packet was parsed successfully, false otherwise
 */
bool parse_packet(const uint8_t *packet, size_t len,
                  weather_data_t *weather_data);
```

Listing 14: Bytestream Parsing Header

14.1 Sample Datasheet

The weather sensor outputs a UART packet with the following format. A graphical version of the packet format is shown in Figure 36.

Byte	Description
0	Start of Frame (0x55)
1	Temperature data (Celsius) - Integer part + 32 ($T_{integer} = Byte_1 - 32$)
2	Temperature data (Celsius) - Fractional part (0-100)
3	Pressure data (Pa - Integer Part) - MSB
4	Pressure data (Pa - Integer Part)
5	Pressure data (Pa - Integer Part)
6	Pressure data (Pa - Integer Part) - LSB
7	Checksum - Lowest byte of the sum of bytes 0-6

Table 3: Byte Descriptions for Weather Sensor UART Packet

Figure 36: Weather Sensor Packet Format

——— Answers Ahead ———

14.2 Packet Parsing Algorithm

A packet parsing algorithm is a staple of embedded systems programming. Generally, an implementation of such an algorithm involves receiving a buffer (packet) of bytes (also referred to as a *bytestream*), followed by iterating through the buffer and extracting relevant data. If present in the packet, a checksum is used to verify the integrity of the data.

14.2.1 Checksum

A checksum is a simple error-detection method[25] that involves summing the bytes of a packet and comparing the result to a predefined value. If the checksum is incorrect, the packet is considered corrupt. In this case, the checksum is the sum of bytes 0-6. A simple algorithm for checksum calculation is shown in Listing15.

```
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

bool verify_checksum(const uint8_t *buf, size_t len, uint16_t expected) {
  uint16_t checksum = 0; // note: overflow possibility if len > 255
  for (size_t i = 0; i < len; i++) {
    checksum += buf[i];
  }

  return checksum == expected;
}
```

Listing 15: Checksum Calculation Example

For this example, the checksum is calculated by summing the bytes from 1 to 6. However, the checksum that is transmitted is the lowest byte of the sum. This is done to save bandwidth and reduce the number of bytes transmitted. It is equivalent to saying that `checksum_transmitted = checksum_calculated & 0xFF`.

14.3 Parsing Function Implementation

When implementing a parsing function, it's important to consider the failure cases that require attention in our implementation (and usually make up most of what interviewers are expecting). In the scope of this question, the edge cases include:

- The packet length is incorrect (the length is fixed and the packet is assumed to be completely received, therefore, any buffer length that is not equal to the expected packet length is considered an error).
- The buffer or data packet arguments are `NULL`.
- The start of frame byte (0x55) is not found at the beginning of the packet.
- The checksum is incorrect.

A sample implementation of the parsing function is shown in Listing16. Note that type-punning is explicitly not used in this implementation to avoid dealing with endianness, alignment, and platform portability issues, although it is a common practice in embedded systems programming.

[25] In real-world implementations, a more robust method, like a cyclic redundancy check (CRC) is usually used.

```c
#include "packet_parsing_header.h"
#define NUM_BITS_IN_BYTE (8U)

bool parse_packet(const uint8_t *packet, size_t len,
                  weather_data_t *weather_data) {
  bool success = false;
  // Let the compiler do the work of calculating len
  const size_t expected_len = (1U + 2U + 4U + 1U);
  if ((len == expected_len) && (packet != NULL) && (weather_data != NULL)) {
    const bool SOF_match = (packet[0] == 0x55);

    uint16_t sum_of_bytes = 0;
    for (size_t i = 0; (i < 7) && SOF_match; i++) {
      sum_of_bytes += packet[i];
    }
    const uint8_t received_checksum = packet[7];
    // Only compare the least significant byte of the sum
    const bool checksum_match = ((sum_of_bytes & 0xFF) == received_checksum);
    success = checksum_match && SOF_match;

    if (success) {
      float temperature_degC = 0.0f;
      temperature_degC += (float)packet[1] - 32.0f; // integer part;
      temperature_degC += (float)packet[2] / 10.0f; // fractional part;
      float pressure_Pa = 0.0f;
      pressure_Pa += (float)((uint32_t)packet[3] << (NUM_BITS_IN_BYTE * 3));
      pressure_Pa += (float)((uint32_t)packet[4] << (NUM_BITS_IN_BYTE * 2));
      pressure_Pa += (float)((uint32_t)packet[5] << (NUM_BITS_IN_BYTE * 1));
      pressure_Pa += (float)(packet[6]);
      weather_data->temperature_degC = temperature_degC;
      weather_data->pressure_kPa = pressure_Pa / 1000.0f; // Convert Pa to kPa
    }
  }
  return success;
}
```

Listing 16: Packet Parsing Function

14.3.1 Testing

In some interviews, an interviewer may ask you to write a test function to verify that the parsing function works correctly. The list of edge cases mentioned above can be used to write test cases - this is further elaborated upon in Extra Practice: Write a unit test for a packet parsing function.

14.4 Follow-ups

- How would you modify the parsing function to handle a packet with a different checksum algorithm?
- How would you modify the parsing function to handle asynchronous data transmission (i.e. fragmented packets)?

15 How would you sense how much current is flowing through a PCB to a load?

——— Answers Ahead ———

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

15.1 Motivation

Embedded systems often need to track current consumption of loads for a variety of reasons, including:

- Detecting if a load is drawing an excessive amount of power, indicating it has failed or an anomaly is occurring.
- Ensuring a system does not overload a source power supply as it could cause the entire system to lose power.
- Determine battery state of charge by integrating the power ($P = I \cdot V$) the battery has been charged and/or discharged with over time.

Note that power sensing is a related concept that can be done by multiplying the current and voltage sensed across a load (as $P = I \cdot V$). Previous questions have covered voltage sensing, so this question will focus on current sensing - combining both techniques allow for power sensing.

15.2 Resistive Current Sensing

The simplest method of current sensing in embedded systems is the *resistive current sense* technique. By placing a resistor in series with the load as shown in Figure 37, the voltage drop across the resistor can be measured to determine the current flowing through the load. This is done by leveraging Ohm's Law, $V = I \cdot R$, where I is the current flowing through the load, V is the voltage drop across the resistor, and R is the resistance of the resistor. Since the resulting voltage drop is proportional to the current flowing through the resistor, the amount of current flowing through the load can be calculated as $I = \frac{V}{R}$.

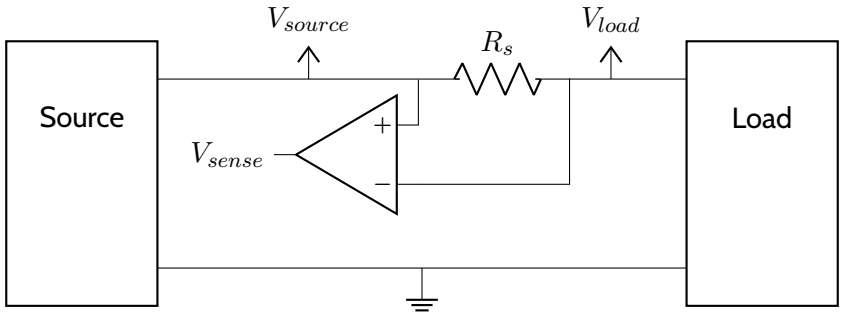


Figure 37: High Side Current Sensing

This technique can be used for both DC (*Direct Current*) and AC (*Alternating Current*) loads as the impedance of the resistor is not dependent on frequency or time.

15.2.1 Resistor Selection

An important design decision for engineers is selecting the resistor to use as R when employing resistive current-sensing. If the value of R is too large, then the resistor will dissipate excessive power - this is shown when Ohm's Law, $V = I \cdot R$, is substituted into $P = I \cdot V$ to get $P = I^2 \cdot R$. Another issue that arises with a large R value is that loads are often designed to operate with a fixed input voltage - the larger R is, the larger the voltage drop induced by the series sense resistor due Ohm's Law, and by doing so, reduces the voltage across the load proportionally to the load current.

On the other hand, if R is too small, then the induced voltage drop across the resistor will be so small that it becomes difficult to measure. Common resistances employed in this technique include values ranging from 0.1 mΩ to 100 mΩ, depending on the current range being measured and allowable power dissipation.

15.2.2 Current Sense Amplifiers

Since the voltage drop across a sense resistor is typically in the low millivolt range, amplifiers are often used to increase this small voltage difference, making it large enough for a microcontroller's ADC to sample accurately, improving the precision of current measurement. The amplifier circuits are oftentimes handled by ICs (*Integrated Circuits*)[26] that implement op-amp based circuits.[27]
When selecting an amplifier, ensure it is rated for the common mode offset that it will be used at. Additionally, amplifiers usually come with a fixed voltage gain that must be selected based on the maximum input voltage difference, $V_{ID_{max}}$, and maximum ADC voltage, $V_{ADC_{max}}$.

15.2.3 Low Side vs. High Side Sensing

Figure 37 shows a high side current sensing circuit, however, 'low side' current sensing is also possible, as shown in Figure 38.[28]

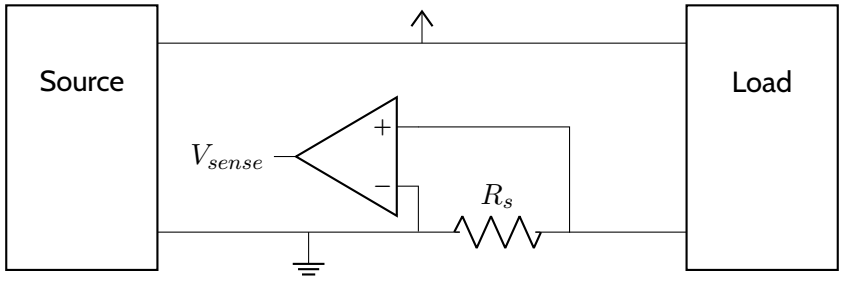


Figure 38: Low Side Current Sensing

Low side[29] current sensing is advantageous as, in theory, only a single-ended amplifier is needed as one of the resistor ends is connected to ground. In contrast, high side sensing needs to amplify the voltage difference across the sense resistor as the voltage drop across the resistor is not referenced to ground, but rather between the input voltage and the load's high side. This means the CSA's input pins have a common-mode offset which the device must be rated to handle. Often, in high precision current-sensing applications, differential amplifiers are used even for low side applications, though they are not theoretically required.[30].

15.3 Magnetic Sensing

When current flows through a wire, it creates a magnetic field around it that is proportional to the current flowing through the wire. This phenomenon is known as *Ampere's Law* and gives rise to numerous

[26] Common examples of simple amplifiers are INA180 and INA240 though numerous options exist for varying applications.
[27] Some ICs have current sense amplifiers (*CSAs*) implemented along with other functionality, such as the INA226.
[28] Low side and high side current sensing is heavily related to low side and high side circuit switching which is explored in Question **??**.
[29] The terms "low" and "high" side come from describing the location of the resistor relative to the load.
[30] More information can be found in Texas Instruments "System Trade-offs for High- and Low-side Current Measurements"

methods of current sensing. An application of this law is used by current clamps, where a core is placed around the current-carrying conductor under measurement. A current clamp core has sense windings on it, forming the secondary winding of a transformer that can then measure AC amplitudes.

This method also works with DC currents if an extra winding around the core is used to actively cancel the magnetic field generated by the sensed DC current. Based on the turns ratio and the current using to cancel the induced magnetic field, the sensed current can be calculated by a meter.

15.4 Follow-ups

- What is kelvin sense?
- What are four-wire digital multi-meter probes and why would you use them?
- What is a hall effect sensor?
- How would you sense the inductor current in a buck converter?
- What is common mode rejection ratio for a differential amplifier?

16 Given the following datasheet and code snippet, initialize the ADC and write a polling function to read the ADC voltage.

16.1 Supporting Problem Information

Assume that the ADC is to be run with an ADC conversion clock frequency of 1 MHz. Implement the functions shown in Listing17 to initialize the ADC and read the ADC voltage.

```c
#include <stdbool.h>
#include <stdint.h>
/**
 * @brief Initialize the ADC.
 * @note Blocking function
 */
void adc_init(void);
/**
 * @brief Read the ADC value
 * @param channel The channel to read from
 * @param voltage The voltage at the specified channel
 * @return true if the read was successful, false otherwise
 */
bool adc_read_channel_V(uint8_t channel, float *voltage);
```

Listing 17: User Implemented Functions

16.1.1 Sample Datasheet

The MythicalMicrocontroller is a microcontroller with a clock frequency of 8 MHz. It features a single-shot 12-bit analog-to-digital converter (ADC) with a reference voltage of 3.3V, with a multiplexer to select between 4 input channels.

ADC Configuration Register (WRITE) - ADDR 0x4000007C The ADC configuration register is used to configure the ADC module with the key system parameters. The ADC must be enabled and ready before any conversions can be performed.

Name	Bits	Description
`ADC_EN`	7	ADC Enable Bit - Write `1` to enable the ADC for conversion.
`ADC_CONVERSION_BEGIN`	6	Set bit to begin an ADC conversion. The `ADC_INITIALIZED` bit in the `ADC_STATUS_REGISTER` must be asserted before this bit is set. This bit clears automatically once a conversion finishes.
`ADC_MUX_SEL 1:0`	5:4	ADC multiplexer channel selection bits.
`ADC_CLK_DIV 3:0`	3:0	ADC Clock prescaler. The ADC clock frequency is determined by the following equation: $f_{ADC\_CLK} = f_{MCU}/(ADC\_CLK\_DIV + 1)$

Table 4: ADC Register Description

ADC Status Register (READ) - ADDR 0x4000007D The ADC status register is used to check the status of the ADC module. The ADC is ready to perform a conversion when the `ADC_INITIALIZED`

bit is set. The `ADC_BUSY` bit is set when the ADC is performing a conversion, and cleared when the conversion is complete.

Name	Bits	Description
`ADC_INITIALIZED`	7	ADC is enabled and ready for use when bit is `1`
`RESERVED`	6:1	Reserved
`ADC_BUSY`	0	ADC is busy with processing the conversion when the bit is asserted. A transition from the bit being asserted to being deasserted indicates the completion of a conversion.

Table 5: ADC Register Description

ADC Data HIGH Register (READ) - ADDR 0x4000007E The ADC Data H register contains the upper 4 bits of the 12-bit ADC conversion result.

Name	Bits	Description
`RESERVED`	7:4	Reserved
`ADC_DATA_H 3:0`	3:0	Upper 4 bits of the 12-bit ADC conversion result

Table 6: ADC Data H Register Description

ADC Data LOW Register (READ) - ADDR 0x4000007F The ADC Data L register contains the lower 8 bits of the 12-bit ADC conversion result.

Name	Bits	Description
`ADC_DATA_L 7:0`	7:0	Lower 8 bits of the 12-bit ADC conversion result

Table 7: ADC Data L Register Description

——— Answers Ahead ———

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

16.2 Setting Up the Problem

When coming across a peripheral initialization problem, it is essential to break down the problem into smaller parts and to understand the requirements - they often have a lot of bark, but not much bite given the limited scope of an interview. The problem can be broken down into two main parts: initialization and conversion. As a first step, we can go ahead and define the register addresses and bit positions for the ADC configuration and status registers, shown in Listing18.

```
#include <stdint.h>
const float V_REF = 3.3;

volatile uint8_t *const ADC_CONFIG_REG = (uint8_t *)0x4000007C;
const uint8_t BITPOS_ADC_EN = 7;
const uint8_t BITPOS_ADC_CONVERSION_BEGIN = 6;
const uint8_t BITPOS_ADC_MUX_SEL = 4;

const uint8_t MAX_ADC_CHANNEL = 3;
const uint8_t MUX_SEL_MASK = MAX_ADC_CHANNEL
                           << BITPOS_ADC_MUX_SEL; // 0b00110000
volatile uint8_t *const ADC_STATUS_REG = (uint8_t *)0x4000007D;
const uint8_t BITPOS_ADC_CONVERSION_INITIALIZED = 7;
const uint8_t BITPOS_ADC_BUSY = 0;

volatile uint8_t *const ADC_DATA_HIGH_REG = (uint8_t *)0x4000007E;
const uint8_t ADC_DATA_HIGH_REG_MASK = 0x0F; // 0b00001111
volatile uint8_t *const ADC_DATA_LOW_REG = (uint8_t *)0x4000007F;
```

Listing 18: ADC Registers and Bit Positions

Note the use of `volatile` in the register definitions. This keyword tells the compiler that the value of the variable can change at any time, which is essential for memory-mapped registers.

16.3 Initialization

Initializing the ADC can be broken down into the following steps. The code is shown in Listing19.

1. **Set the ADC Clock Frequency**: Calculate the ADC clock prescaler value to achieve a 1 MHz ADC conversion clock frequency given the 8 MHz MCU clock frequency. We can use the formula $f_{ADC\_CLK} = f_{MCU}/(ADC\_CLK\_DIV + 1)$ to calculate the prescaler (ADC_CLK_DIV) value (7).

2. **Enable the ADC**: Write to the ADC configuration register to enable the ADC.

3. **Poll the ADC Status Register**: Check if the ADC is ready for a conversion.

```
#include "adc_init_registers.h"
#include <stdbool.h>

void adc_init(void) {
  // Set the prescaler to 3
  const uint8_t PRESCALER_1MHZ_ADC_CLK = 7;
  *ADC_CONFIG_REG = PRESCALER_1MHZ_ADC_CLK;

  // Enable the ADC
```

```
  *ADC_CONFIG_REG |= (1 << BITPOS_ADC_EN);

  // Wait for the ADC to be ready
  while ((*ADC_STATUS_REG & (1 << BITPOS_ADC_CONVERSION_INITIALIZED)) == false
      )
    ;
}
```

Listing 19: ADC Initialization

16.4 Conversion

The conversion problem can be addressed as follows. The code is shown in Listing20.

1. **Set the ADC Multiplexer Channel**: Write to the ADC configuration register to select the desired ADC multiplexer channel.
2. **Begin the ADC Conversion**: Write to the ADC configuration register to begin an ADC conversion.
3. **Poll the ADC Status Register**: Wait for the ADC to finish the conversion by polling the ADC status register.
4. **Read the ADC Data Registers**: Read the ADC data registers to get the 12-bit ADC conversion result. We need to read the ADC data registers in two separate reads (high and low) and combine the results by shifting the high bits left by 8 and ORing with the low bits.
5. **Calculate the ADC Voltage**: Calculate the ADC voltage from the 12-bit ADC conversion result by using the formula $V_{ADC} = \frac{ADC\_RESULT \cdot V_{REF}}{2^{12}-1}$. Note that the $2^{12} - 1$ term is the maximum value of a 12-bit number, and corrosponds to the ADC reference voltage.

```
#include "adc_init_registers.h"
#include <stdbool.h>
#include <stddef.h>

bool adc_read_channel_V(uint8_t channel, float *voltage) {
  bool ret = false;
  const bool channel_valid = (channel <= MAX_ADC_CHANNEL);
  const bool adc_ready =
      (*ADC_STATUS_REG & (1 << BITPOS_ADC_CONVERSION_INITIALIZED));
  if (channel_valid && adc_ready && (voltage != NULL)) {
    // reset the MUX_SEL bits - & with the inverse of the mask, which sets the
    // masked bits to 0
    *ADC_CONFIG_REG &= ~(MUX_SEL_MASK);
    // Set the MUX_SEL bits to the channel
    *ADC_CONFIG_REG |= (channel << BITPOS_ADC_MUX_SEL);
    // Start the conversion
    *ADC_CONFIG_REG |= (1 << BITPOS_ADC_CONVERSION_BEGIN);
    // Wait for the conversion to complete
    while ((*ADC_STATUS_REG & (1 << BITPOS_ADC_BUSY)))
      ;
    const uint16_t ADC_HIGH_VALUE = *ADC_DATA_HIGH_REG &
        ADC_DATA_HIGH_REG_MASK;
    // Read the ADC value.
```

```
    const uint16_t adc_value = (ADC_HIGH_VALUE << 8) | *ADC_DATA_LOW_REG;

    const uint16_t ADC_MAX = 4095; // 2^12 - 1 for 12-bit ADC
    *voltage = (adc_value * V_REF) / (float)ADC_MAX;
    ret = true;
  }
  return ret;
}
```

Listing 20: ADC Conversion

16.5 Follow-ups

- **Strategies for avoiding polling in the ADC conversion function (ex: DMA, interrupts, etc).**
- **What is the impact of the ADC clock frequency on the ADC conversion time and resolution?**
- **What are sources of error in an ADC and how are they mitigated?**

17 Given a 64-bit timer consisting of two 32-bit count registers, implement a function to get the 64 bit count.

Assume that the timer is initialized, counts up, and updates itself atomically. The following registers are available, with the header for the question available in Listing 21:

- `CNT_LOW` is the lower 32 bits of the timer count.
- `CNT_HIGH` is the upper 32 bits of the timer.

```c
#include <stdint.h>

volatile uint32_t *CNT_LOW = (uint32_t *)0x40000000;
volatile uint32_t *CNT_HIGH = (uint32_t *)0x40000004;

/**
 * @brief Get the 64 bit time from the timer
 * @return uint64_t The 64 bit time
 */
uint64_t get_64_bit_time(void);
```

Listing 21: Timer Registers

——— Answers Ahead ———

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

17.1 Timer Background

As a review, a timer is a hardware peripheral that can be configured in firmware to perform specific operations. Generally, timers have a counter that increments or decrements at a fixed rate, and the current count of the timer is accessed by means of a dedicated `COUNT` register. In the context of this question, it's mentioned that the timer counts up, meaning the count value increases with time at some unspecified rate. As an example, suppose we want to keep track of the number of milliseconds that have passed since the timer was started. If the timer is configured to increment every millisecond, the count value will increase by 1 every millisecond, and the count value can be read to determine the time elapsed in milliseconds.

17.2 Simple Implementation

On first glance, it's tempting to claim the answer is as simple as combining the two 32-bit registers into a 64-bit value, much like what is presented in Listing 22.

```
#include "timer_registers.h"

uint64_t get_64_bit_time(void) {
  uint64_t time = 0;
  time |= *CNT_LOW;
  time |= ((uint64_t)*CNT_HIGH) << 32;
  return time;
}
```

Listing 22: Naive 64-bit Timer Read

However, this implementation fails to consider that the 32-bit registers are updated by timer hardware asynchronously. If the timer updates the `CNT_HIGH` register between reading `CNT_LOW` and `CNT_HIGH`, the resulting 64-bit value will be incorrect. Consider the following scenario:

1. Assume that at the start of the function, the timer count register values are as follows: `CNT_LOW = 0xFFFFFFFF`, while `CNT_HIGH = 0x00000000`.
2. The function reads the `CNT_LOW` value as `0xFFFFFFFF`[31].
3. In between reading `CNT_LOW` and `CNT_HIGH`, the timer increments. This causes `CNT_LOW` to wrap around to `0x00000000` and consequently `CNT_HIGH` to increment to `0x00000001`.
4. The function now reads the `CNT_HIGH` value as `0x00000001`.
5. The resulting 64-bit value is `0x00000001 FFFFFFFF`, which is incorrect. The correct value should be `0x00000001 00000000`.

The implications of this is that the 64 bit time value is no longer *monotonic* (constantly increasing), aside from being straight up incorrect. Consumers of this function may experience unexpected behavior if the timer is used for timekeeping or scheduling, as the time value may appear to jump backwards if called near a timer update/roll-over. Note that this issue would also occur if the read order of the high and low COUNT registers were swapped in the implementation.

[31] This is $2^{32} - 1$ in hex - `0x` means hexadecimal notation in C and is commonly used in datasheets and embedded software manuals as a result of the industry's adoption of C and C++.

17.3 Correct Implementation

To correctly read the 64-bit timer value, we need to ensure that the `CNT_HIGH` register and `CNT_LOW` register are 'synchronized' in time. This can be achieved by reading the `CNT_HIGH` register first, then reading the `CNT_LOW` register, and finally reading the `CNT_HIGH` register again. If the two `CNT_HIGH` reads are equal, we can be confident that the 64-bit value is correct as a roll-over did not occur. If a roll-over did occur, we re-read the low and high values and use that for the computation of the 64 bit time, as we are no longer close to a roll-over event. The correct implementation is shown in Listing 23.

```
#include "timer_registers.h"

uint64_t get_64_bit_time(void) {
  uint64_t high = *CNT_HIGH;
  uint64_t low = *CNT_LOW;
  // check if the high word has changed since reading low
  if (high != *CNT_HIGH) {
    // roll-over occurred between reading low and high, read low again
    low = *CNT_LOW;
    high = *CNT_HIGH;
  }
  const uint64_t time = ((uint64_t)high << 32) | low;
  return time;
}
```

Listing 23: Correct 64-bit Timer Read

18 Write a C function to determine the direction of stack growth on a system.

——— Answers Ahead ———

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

18.1 Understanding the Stack

The stack is a region of memory that is used to store local variables and function call information. The stack uses a *stack* data structure, meaning that data is organized with by last-in, first-out (LIFO) organization. This means that the last item placed on the stack is the first item to be removed. The stack is used to store local variables, function arguments, and the return address of the function[32]. This is opposed to *heap memory*, which is allocated *dynamically* and is used to store data that persists beyond the scope of a function call and is managed by the user using `malloc` and `free`.

18.1.1 Stack Frame Example

When a function is called, a new *stack frame* is pushed onto the call stack. This stack frame typically includes the return address of the calling function, local variables, and sometimes arguments for the function. When the function returns, its stack frame is popped off the stack, and execution resumes from the stored return address. The term stack trace refers to examining the stack's contents to understand the sequence of function calls that led to the current state, along with associated information like return addresses and (in some cases) function arguments.

Consider the following C snippet in this example, on a system where the stack grows downwards (from higher addresses to lower addresses).

```c
#include <stdint.h>
#include <stdio.h>

uint8_t multiply(uint8_t c, uint8_t d) { return c * d; }

uint8_t add_and_multiply(uint8_t a, uint8_t b) {
  return a + b + multiply(a, b);
}

int main() {
  uint8_t a = 10;
  uint8_t b = 20;
  uint8_t c = add_and_multiply(a, b);
  printf("Result: %u\n", c);
  return 0;
}
```

Listing 24: Stack Example

Generally speaking, the stack might look something like what is found in Table 8 if we paused inside the `multiply` function. Note the stack grows downwards in this example system, hence the decrease in address values. Also note that a new stack frame is created for each function call.

Address	Function	Called From	Arguments
0x5000	`multiply`	`add_and_multiply`	`c = 10, d = 20`
0x6000	`add_and_multiply`	`main`	`a = 10, b = 20`
0x7000	`main`	`_start`	(N/A)

Table 8: High-level overview of the stack during program execution in `multiply`.

[32] It's worth noting that the direction of stack growth is usually a strictly academic question - there is usually little use in only knowing the direction of stack growth.

18.2 Determining Stack Growth Direction

Since we know that a new stack frame is created for each function call, we can use this property to determine the direction of stack growth. The steps to determine the direction of stack growth are as follows:

1. Create a main function that calls a dummy function. Pass in a pointer to a stack-allocated variable to the dummy function.
2. In the dummy function, compare the address of the stack-allocated variable to the address of a local variable in the dummy function.
3. If the address of the stack-allocated variable is less than the address of the local variable, the stack grows downwards. If the address of the stack-allocated variable is greater than the address of the local variable, the stack grows upwards.

The code snippet in Listing25 demonstrates this concept.

```c
#include <stdint.h>
#include <stdio.h>

void check_stack_dir(uint8_t *comparison) {
  uint8_t local = 0;
  if (&local < comparison) {
    // Stack grows down
    printf("Stack grows down\n");
  } else {
    // Stack grows up
    printf("Stack grows up\n");
  }
}

int main() {
  uint8_t comparison = 0;
  check_stack_dir(&comparison);
  return 0;
}
```

Listing 25: Determining Stack Growth Direction

19 Solve the transfer function for the following circuit.

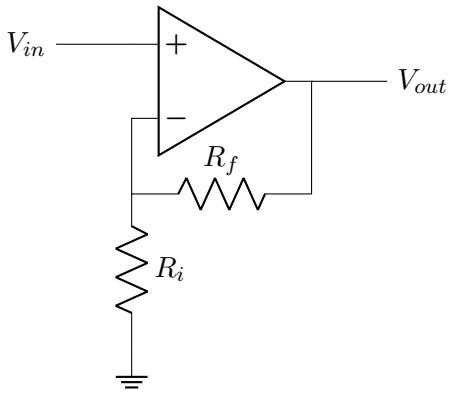


Figure 39: Given Circuit

——— Answers Ahead ———

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

19.1 Operational Amplifiers

Ideal operational amplifiers (*op-amps*) are active devices characterized by the following equations:

$$\begin{aligned} V_{out} &= A \cdot (V_+ - V_-) \quad \text{where} \quad A \to \infty, \\ I_+ &= I_- = 0. \end{aligned} \tag{9}$$

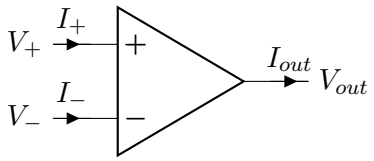


Figure 40: Labelled Op-amp

19.1.1 Virtual Short

When an op-amp is connected in a negative feedback configuration, the inputs (V_+ and V_-) are considered "virtually shorted", meaning $V_+ = V_-$. Negative feedback in an op-amp circuit means there is a current path from it's output, V_{out}, to it's inverting terminal, V_-.

19.2 Solving

When analyzing circuits, begin by defining equations for simple elements and build from there. In this circuit, Ohms' law can be applied to the R_f and R_i resistors, which results in the following relationships: $V_{out} - V_- = I \cdot R_f$ and $V_- = I \cdot R_i$. Note in this case, $I_- = 0$ as the current (I) through resistor R_f is the same as the current through R_i.

Next, because R_f connects a current path from V_{out} to V_-, the virtual short assumption holds for the op-amp - therefore, $V_+ = V_-$. From the circuit, it's seen that $V_{in} = V_+$, so $V_{in} = V_+ = V_-$. This conclusion can be substituted into the above-derived resistor equations to get $V_{out} - V_{in} = I \cdot R_f$ and $V_{in} = I \cdot R_i$.

These equations can be substituted into each other to cancel out I and algebraically rearranged to determine $\frac{V_{out}}{V_{in}} = 1 + \frac{R_f}{R_i}$. This solution for $\frac{V_{out}}{V_{in}}$ is known as the transfer function of the circuit. The concept of a transfer function is used to analyze numerous circuits.

19.2.1 Intuition

To understand this circuit better, consider how this circuit operates.

- Because negative resistors don't exist, $R_f > 0$ and $R_i > 0$, $\frac{V_{out}}{V_{in}} > 1$ always. As the gain of the circuit is always positive, this circuit is referred to as a *Non-inverting Amplifier*.
- When $R_f = R_i = R$ the transfer function simplifies into $\frac{V_{out}}{V_{in}} = 2$.
- When $R_f >> R_i$ the gain becomes very large, $\frac{V_{out}}{V_{in}} \approx \infty$.
- When $R_i >> R_f$ the gain approaches unity, $\frac{V_{out}}{V_{in}} \approx 1$.

19.3 Unity Gain Amplifier

A special case of this circuit occurs when R_f is replaced with a short circuit, $R_f = 0$, and R_i is replaced with an open circuit, $R_i = \infty$ as drawn in Figure 56.

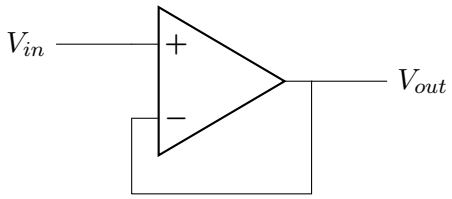


Figure 41: Unity Gain Amplifier Circuit

In this case the transfer function becomes unity, $\frac{V_{out}}{V_{in}} = 1$ which can be simplified into $V_{out} = V_{in}$.

This circuit acts as a current buffer. Because $I_{V_+} = 0$ and $V_{in} = V_+$ we see that $I_{in} = 0$ so the circuit doesn't load the input at all. Instead, any current required by the load is provided by the op-amp!

19.4 Follow-ups

- What is an inverting amplifier?
- What is a difference amplifier?
- Describe the non-idealities of op-amps and how to compensate for them?
- Why are non-inverting amplifiers preferred over inverting amplifiers?
- Solve the circuits in extra practice question 25.

20 What are the differences between a mutex and a semaphore, and in what use cases are each typically employed?

——— Answers Ahead ———

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

20.1 Introduction

A mutex and a semaphore are both synchronization primitives that can be used to signal and/or synchronize access to shared resources in a multi-threaded environment. This section summarizes general principles of interrupt service routines, as commonly described in embedded systems literature - DigiKey has excellent articles on the implementation of real-time operating systems (RTOS) and synchronization primitives [19].

20.2 Semaphore

A *semaphore* (also known as a *counting semaphore*) is a signalling mechanism that allows a thread to signal one or more waiting threads that a particular event has occurred. It can be thought of as a shared non-negative integer counter.

Using a semaphore involves two main operations:

- `signal_semaphore()` - increments the semaphore counter
- `wait_for_semaphore_signal(timeout)` - decrements the semaphore count, and if the count is less than zero, blocks the calling thread for the duration of the timeout until the count is greater than zero.

A semaphore is often used in the following scenarios:

- Controlling access to a pool of resources
  - Ex: A server thread with a limited number of connections - the semaphore count would represent the number of available connections, with the semaphore being decremented when a connection is acquired and incremented when a connection is released. A client thread would wait for the semaphore to be incremented before attempting to acquire a connection.
- Signalling to thread(s) that a particular event has occurred. In the context of an RTOS, where there is only one core available, a semaphore can be used to signal an event to a higher priority task from a lower priority task, and have the higher priority task run instantly. This is because the higher priority task will preempt the lower priority task once it is no longer 'blocked' by the semaphore as it will be available for the higher priority task to acquire.
  - Ex: An interrupt service routine (ISR) can signal to a worker thread that an event has occurred. *Note: Event flags are also commonly used for this purpose, if provided by the OS.*

A short example of using a semaphore is shown in Listing26.

```
semaphore_wait(conn_semaphore); // Wait for an available connection
use_connection();
semaphore_signal(conn_semaphore); // Release the connection
```

Listing 26: Example of Semaphore Usage

20.3 Mutex

A *mutex* (short for *mutual exclusion*) is a signalling mechanism that is used to prevent multiple threads from accessing a shared resource simultaneously. It can be thought of as a lock that is either locked or unlocked.

Using a mutex involves two main operations:

- `lock_mutex(timeout)` - attempts to lock the mutex. If the mutex is already locked, the function blocks the calling thread for the duration of the timeout until the mutex is unlocked.
- `unlock_mutex()` - unlocks the mutex, allowing other threads to lock it.

Consider a UART driver that is being accessed by multiple threads. If the UART driver is not thread-safe, it is possible that two threads could attempt to write to the UART at the same time, causing garbled output. In this case, a mutex could be used to ensure that only one thread can access the UART at a time. Pseudocode for this scenario is shown in Listing27.

```
mutex_lock(uart_mutex, DELAY_INFINITE);
uart_write("Hello");
mutex_unlock(uart_mutex);
```

Listing 27: Example of Mutex Usage

20.4 Putting it together

The key differences between a mutex and a semaphore are that

- A mutex only has 2 states, locked and unlocked, while a semaphore can count.
- A mutex has a concept of ownership, while a semaphore does not. This means that the thread that locks a mutex must be the one to unlock it, while any thread can signal or wait on a semaphore.
- A mutex is typically used to protect access to a shared resource, while a semaphore is typically used to signal an event or control access to a pool of resources.

20.5 Priority Inversion and Inheritance

A common follow-up question is to ask about the potential consequences of using a mutex in a real-time system. One of the most common issues is *priority inversion*, which occurs when a high-priority task is blocked by a lower-priority task that holds a mutex. An example of this situation is shown in Figure 42. In this scenario, the high-priority task is blocked by the mutex held by the low-priority task. Since the medium priority task is of higher priority than the low-priority task, it will run before the low-priority task, causing the high-priority task to be blocked for an extended period of time. This can lead to missed deadlines and degraded system performance.

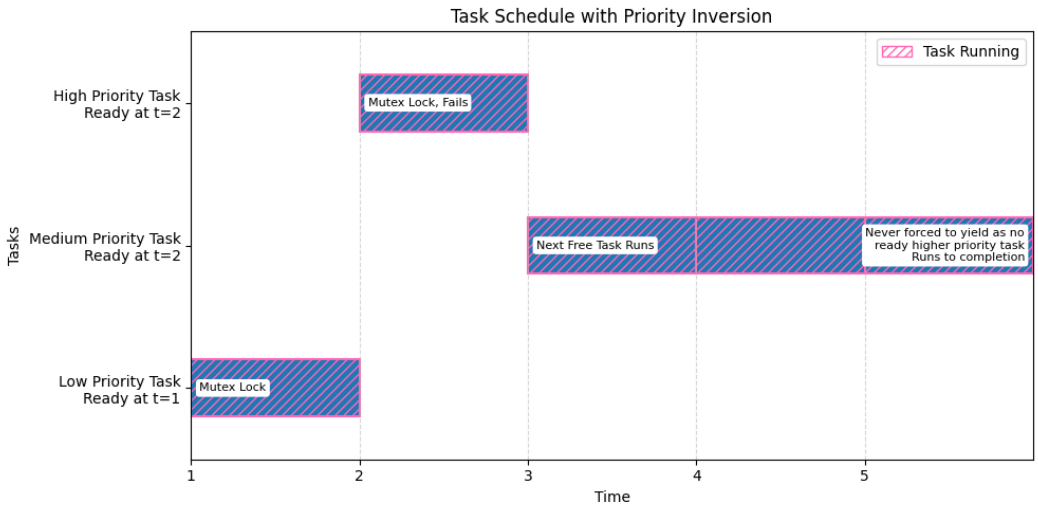


Figure 42: Priority inversion diagram.

20.5.1 Priority Inheritance

A solution to fix this is called *priority inheritance* - the OS will temporarily boost the priority of the low-priority task to the priority of the high-priority task while the low-priority task holds the mutex [20]. This ensures that the high-priority task can run as soon as the low-priority task releases the mutex. An example of this situation is shown in Figure 43.

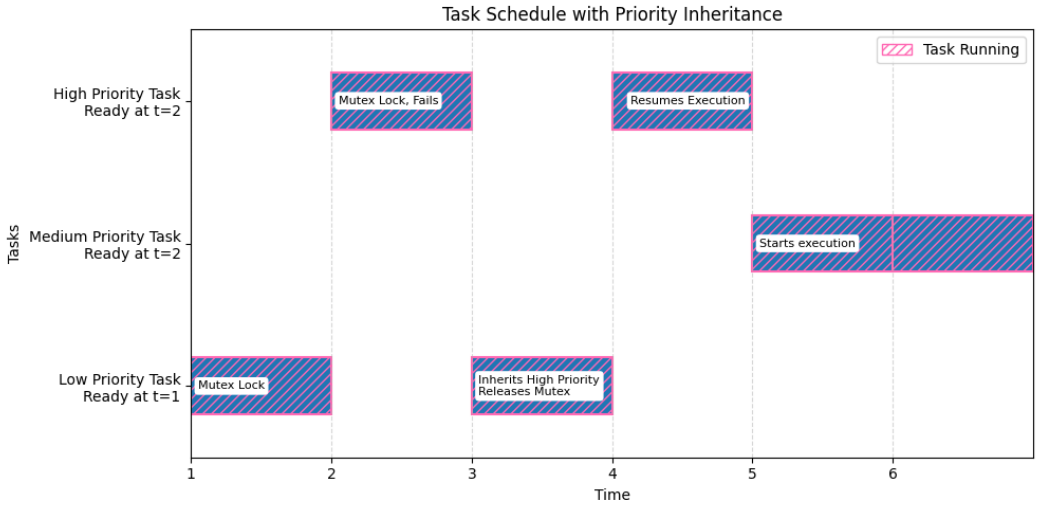


Figure 43: Priority inheritance diagram.

20.6 Follow-ups

- Describe other RTOS features that can be used for inter-task communication.
- Explain how deadlock can occur in an RTOS and how it can be prevented.

21 When would you use a buck converter or a low dropout regulator?

——— Answers Ahead ———

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

21.1 Low Dropout Regulator

LDOs (*Low Dropout Regulators*) are commonly available as ICs (*Integrated Circuits*). LDOs are most commonly used as a voltage source for circuits that are characterized by requiring low voltage low current, and high stability, such as microcontrollers and analog circuitry. They are also ideal in space constrained applications as they can be extremely small. A schematic representation of an LDO is shown in Figure 44.

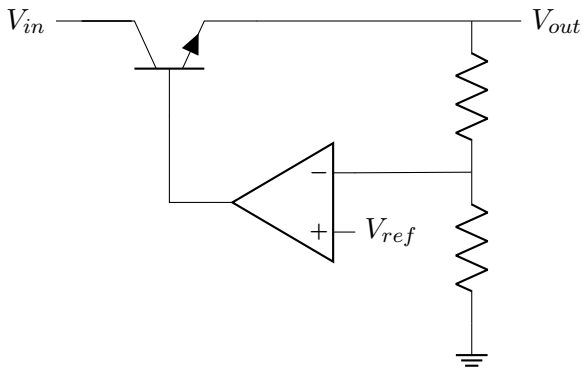


Figure 44: Low Dropout Regulator Conceptual Schematic

21.1.1 Selection

When selecting an LDO for a given application, consider optimizing the following parameters in component selection to save board space and cost while ensuring functionality.

- **Dropout Voltage:** To maintain a stable output voltage, the LDO requires sufficient input voltage. Mathematically, this is expressed as $V_{in} > V_{out} + V_{dropout}$. The $V_{dropout}$ varies between components but is usually around 0.5 V.
- **Maximum Input Voltage:** The semiconductor pass element inside the transistor is rated to handle a specific maximum input voltage.
- **Maximum Output Current:** The pass element semiconductor has a specific limit to the maximum output current it can provide. Exceeding this limit can lead to overheating or failure. This relates to the maximum power dissipation of the package as there is a limit to the amount of heat that can be dissipated into the environment.
- **Output Voltage:** Some LDOs feature adjustable output voltages with a feedback pin, while others have fixed output voltages.

21.1.2 Losses

In an LDO, the pass element transistor (usually implemented as a PMOS FET or NPN BJT) acting similarly to a variable resistor is actively controlled to maintain a fixed output voltage, regardless of changes to input voltage, load current, and temperature. This variable resistance gives rise to *conduction losses* within this type of regulator. Assuming a fixed input voltage and load current, the power losses in an LDO can be calculated as $P = V_{drop} \cdot I_{out} = (V_{in} - V_{out}) \cdot I_{out}$. Note that the power is dissipated as heat in the pass element - this is why LDOs are often equipped with a heat sink, and attention may be

required to the thermal limitations of the chosen LDO.

Another source of losses in an LDO is quiescent losses - the LDO requires a small amount of current to operate, even when the load current $I_{out} = 0$. This is known as the quiescent current, and is usually negligible compared to conduction losses.

21.2 Buck Converter

The buck converter is a simple yet fundamental circuit in power electronics. It is a *Switched-Mode Power Supply* (SMPS) that converts a higher input voltage to a lower output voltage. Note that buck converters and associated topics are frequently covered in electrical engineering interviews. An example of a buck converter is shown in Figure 45.

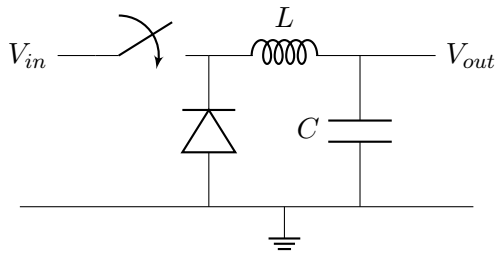
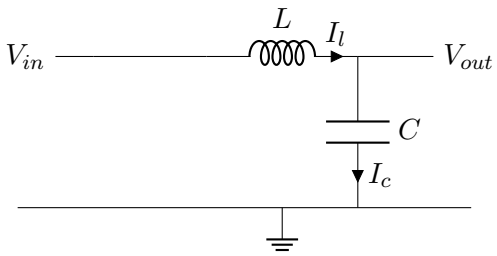


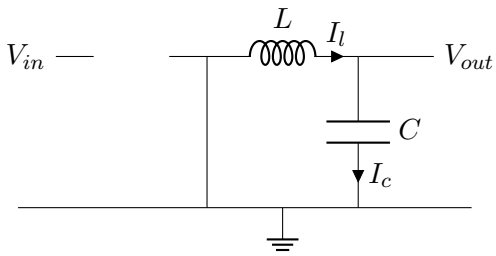
Figure 45: Asynchronous Buck Converter Conceptual Schematic

21.2.1 Operation

When a buck converter is operational, the high side switch turns on during T_{on} and off during T_{off}. The duty cycle of a buck converter is defined as $D_{on} = \frac{T_{on}}{T_{on}+T_{off}}$. The switching frequency of a buck converter is given by $f_{SW} = \frac{1}{T_{on}+T_{off}}$. Buck converters generally operate under a roughly constant switching frequency, but modulate their duty cycle with an active control loop to produce a stable output voltage. The current paths during T_{on} and T_{off} are depicted by Figure 46a and Figure 46b respectively.



(a) Buck Converter with High Side Switch ON



(b) Buck Converter with Low Side Switch ON

Figure 46: Buck Converter Operation in Different Switching States

Buck converters usually operate in CCM (*Continuous Conduction Mode*) meaning there is constantly current flowing forward in the inductor. In CCM, $D_{on} = \frac{V_{out}}{V_{in}} = \frac{T_{on}}{T_{on}+T_{off}}$, describes the duty cycles of an ideal buck converter as shown in the following figure.

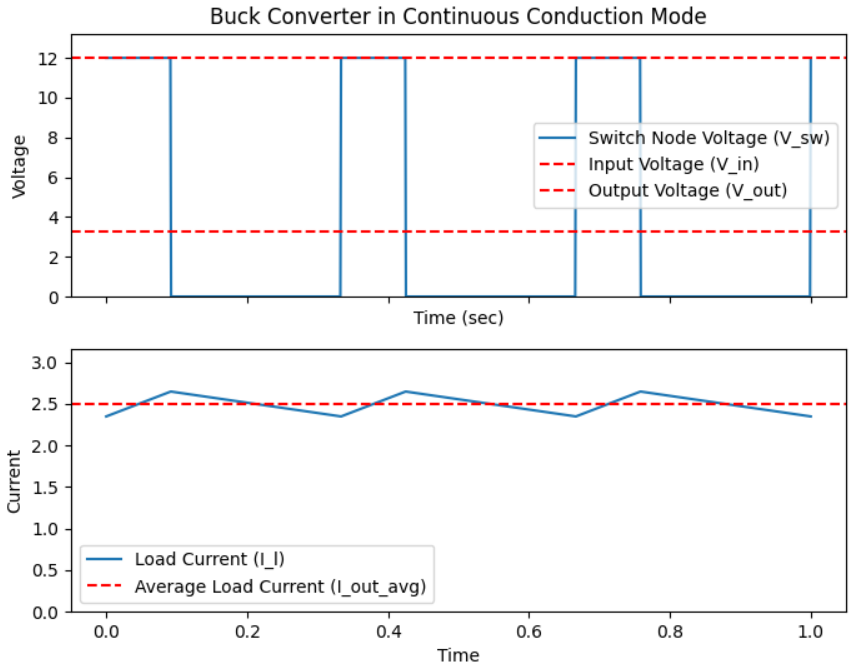


Figure 47: Buck Converter in Continuous Conduction Mode

When $I_{L_{avg}}$ decreases below a threshold, the buck converter circuit enters DCM (*Discontinuous Conduction Mode*) in which I_L reaches $I_L = 0$ at the end of the t_{off} period as depicted in Figure 48.

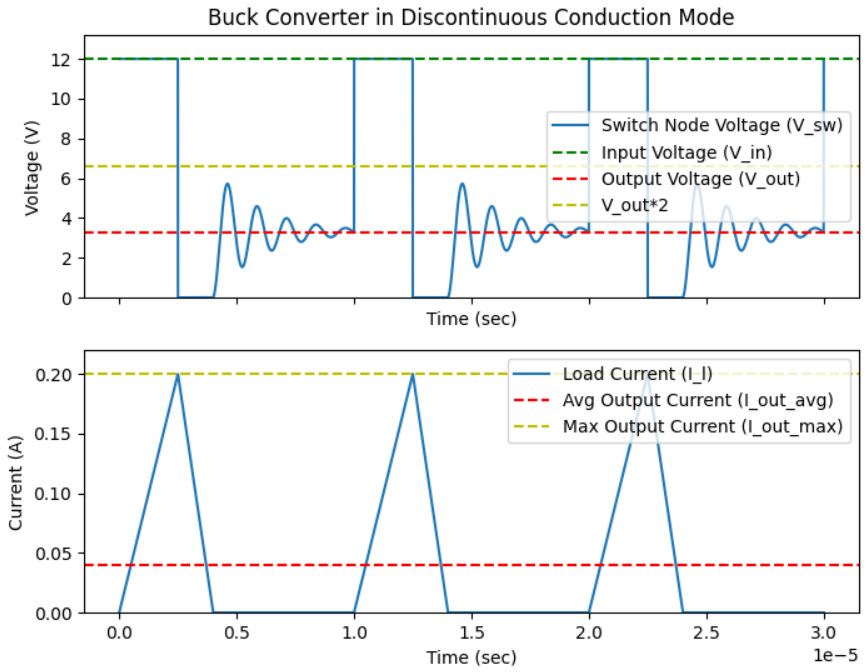


Figure 48: Buck Converter in Discontinuous Conduction Mode

When $I_L \approx 0$ the switch node voltage, V_{sw} begins to oscillate with a peak of $\approx V_{out} \cdot 2$ converging towards V_{out}.[33] This is because when $I_L \approx 0$, $V_{sw} = 0$, however, the output node is not at zero, $V_{out} \neq 0$ which means current will begin to flow through the inductor into the parasitic capacitance, $C_{parasitic}$ that exists between V_{sw} and ground. This small resonant current, I_L, results in a noticable voltage fluctuation on V_{sw}, but not on V_{out} because $C_{out} >> C_{parasitic}$ (where C_{out} is the output capacitance of the buck converter).

21.2.2 Switching

The switch depicted in Figure 45 is commonly referred to as a high side switch. In practice, the high side switch is usually implemented with a MOSFET (*Metal-oxide semiconductor field effect transistor*) that features fast switching speeds, low switching losses, low conduction losses, low cost, and low leakage. The gate of this MOSFET requires active control from a feedback loop to maintain a stable output voltage.

An asynchronous buck converter is depicted in Figure 45 depicts the low side switch as a diode. The diode is nice to use because it does not require control and is cheaper than a transistor. Diodes, due to their forward voltage drop, have higher conduction losses than MOSFETs.

Synchronous buck converters replace this low side diode with a MOSFET. In order to avoid shorting out the input voltage source dead-time, a short period of time in which both high side and low side transistors are off, is inserted into the control loop. During the dead-time, current continues to flow through the body diode of the low side MOSFET.

21.2.3 Losses

There are two major forms of loss in a buck converter: *switching losses* and *conduction losses*. Switching losses is power dissipated every time the FETs are switched (transitioned from off to on, or vice-versa) and scale proportionally with switching frequency. Conduction losses are due to parasitic resistance of elements in the converter and consequently scale proportionally to load current. Buck converters have some quiescent current associated with their active control loop, however, these losses are negligible compared to switching and conduction losses and are not usually analyzed.

For a majority of applications, especially when I_{load} is large and/or $\frac{V_{out}}{V_{in}}$ is very small, buck converters are significantly more efficient than LDOs. Typical buck converter efficiency, $\frac{P_{out}}{P_{in}}$, exceeds 90 percent.

21.2.4 Switching Frequency

Selection of switching frequency is critical in the design of a buck converter. Increasing switching frequency increases switching losses in the converter, but allows the usage of a lower inductance inductors which are usually cheaper and come in smaller packages[34]. Switching frequencies of modern buck converters range from roughly 100 KHz to 5 MHz.

21.3 Comparison

Buck converters are more efficient than LDOs in a majority of applications though an LDO may be optimal in applications where $V_{in} \approx V_{out}$ and/or $I_{out} \approx 0$.

[33] If the oscillation was undamped it would have a peak of exactly $V_{out} \cdot 2$.

[34] The inductor is often the largest single component in a buck converter so increasing switching frequency to reduce buck converter overall circuit board area is a common practice.

Buck converters require more physical space than LDOs on a circuit board, mostly because they need an inductor to operate. The need for an inductor and switching circuitry also makes buck converters more expensive than LDOs.

LDOs produce a more stable output voltage and can have higher control loop bandwidth as they do not have a switching stage nor an output filter as compared to a buck converter. Both converter topologies require input and output capacitance to produce relatively stable output voltages.

21.3.1 Power Tree

Embedded system controllers often have relatively high voltage power sources, to minimize current and in turn conduction losses, and are tasked with distributing lower voltage power to endpoints. Usually a single controller board is responsible for driving numerous loads with each having their own power requirements, most notably a roughly fixed voltage.

Consider a hypothetical controller in an automobile tasked with powering a 24V motor and a 5V sensor from a 48V source. A solution could be to convert 48V to 24V for the motor and use another converter to convert 48V to 5V. However, a more efficient approach is to convert 48V to 24V and supplying the 24V to the motor and another converter from 48V to 5V intended solely for the sensor. There are numerous tradeoffs between operating cascaded versus parallel converters in a power tree including: failure modes, low power states, and conversion efficiency.

21.4 Follow-ups

- How would you measure the efficiency of a buck converter? What does the test setup look like?
- When would you use Diode Emulation Mode and Forced Pulse Width Modulation Mode for a Buck Converter?
- Why are input capacitors required? What happens if you do not have input capacitors?
- What are some considerations when selecting output capacitors?

22 Extra Practice: Write a unit test for a packet parsing function.

——— Answers Ahead ———

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

22.1 Solution

As mentioned in Question 14 (*Implement a C bytestream parsing function for a weather data sensor*), unit tests are a crucial part of software development. They help ensure that the code behaves as expected and catches bugs early in the development process. In this section, an example of a unit test framework and unit tests for the packet parsing function are presented. Note that the tests and framework are crude to demonstrate the type of code that would be written in an interview setting. In real-life, a dedicated test framework, like CppUTest or GoogleTest, is strongly recommended.

The unit tests in principle work by the expected output being the extracted values as well as the return value of the parsing function. Listing 28 shows example unit tests and crude test framework for testing the packet parsing function.

```c
#include "packet_parsing_header.h"
#include <math.h>
#include <stdio.h>

// Use a macro to compare floating point numbers since == is not reliable for
// floats
#define FLOAT_EQUALS(a, b) (fabsf(a - b) < 0.0001f)

bool test_invalid_args(void) {
  weather_data_t weather_data;
  uint8_t data[8U] = {0};
  bool success = parse_packet(NULL, 0, &weather_data);
  success |= parse_packet(data, 0, &weather_data);
  success |= parse_packet(data, 8, NULL);

  return !success;
}

bool test_incorrect_SOF(void) {
  uint8_t packet[] = {0x54, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x54};

  weather_data_t weather_data;
  bool success = parse_packet(packet, sizeof(packet), &weather_data);

  return !success;
}

bool test_incorrect_checksum(void) {
  uint8_t packet[] = {0x55, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x53};

  weather_data_t weather_data;
  bool success = parse_packet(packet, sizeof(packet), &weather_data);

  return !success;
}

bool test_correct_packet(void) {
  uint8_t packet[] = {0x55, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

  uint8_t fake_temperature_degC = 12U;
```

```
  packet[1] = fake_temperature_degC + 32U;

  uint8_t fake_temperature_fraction_degC = 5U;
  packet[2] = fake_temperature_fraction_degC;

  uint32_t fake_pressure_pA = 101325U;
  packet[3] = (fake_pressure_pA >> 24) & 0xFF;
  packet[4] = (fake_pressure_pA >> 16) & 0xFF;
  packet[5] = (fake_pressure_pA >> 8) & 0xFF;
  packet[6] = fake_pressure_pA & 0xFF;

  uint32_t checksum = 0U;
  for (size_t i = 0; i < 7; i++) {
    checksum += packet[i];
  }

  packet[7] = checksum & 0xFF;

  weather_data_t weather_data;
  bool success = parse_packet(packet, sizeof(packet), &weather_data);

  success &= (FLOAT_EQUALS(weather_data.temperature_degC, 12.5f));
  success &= (FLOAT_EQUALS(weather_data.pressure_kPa,
                           (float)fake_pressure_pA / 1000.0f));

  printf("Temperature: %.2f degC\n", weather_data.temperature_degC);
  printf("Pressure: %.2f kPa\n", weather_data.pressure_kPa);

  return success;
}

int main() {
  bool success = true;
  if (test_invalid_args() == false) {
    printf("test_invalid_args failed\n");
    success = false;
  }

  if (test_incorrect_SOF() == false) {
    printf("test_incorrect_SOF failed\n");
    success = false;
  }

  if (test_incorrect_checksum() == false) {
    printf("test_incorrect_checksum failed\n");
    success = false;
  }

  if (test_correct_packet() == false) {
    printf("test_correct_packet failed\n");
    success = false;
  }

  if (success) {
```

```
    printf("All tests passed\n");
  }

  return success ? 0 : 1;
}
```

Listing 28: Parsing Test Function

23 Extra Practice: Propose a simple circuit to disable current consumption when the voltage divider is not needed.

——— Answers Ahead ———

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

23.1 Quiescent Current

Quiescent current is current drawn when a circuit is inactive or not being used and is proportional to power consumption. Reducing the quiescent current of a voltage divider can be done by increasing R_{sum}, however, this only gets you so far. For low power devices, transistor based circuits can be used to disable current flow when sampling the voltage divider is not necessary. An example circuit is given in Figure 49.

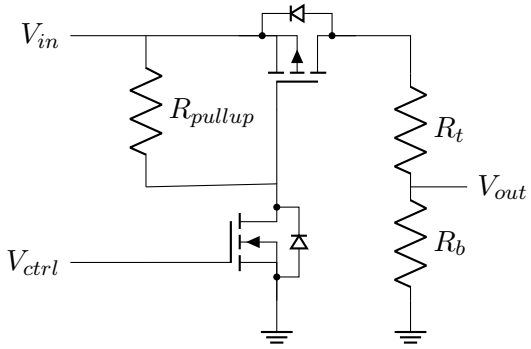


Figure 49: Switched Voltage Divider Circuit

24 Extra Practice: Determine the step response of the following circuits.

State any assumptions regarding component values. These questions are more complex then those given in Section 13, but are fun to consider.

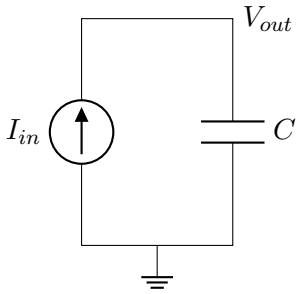


Figure 50: Circuit E

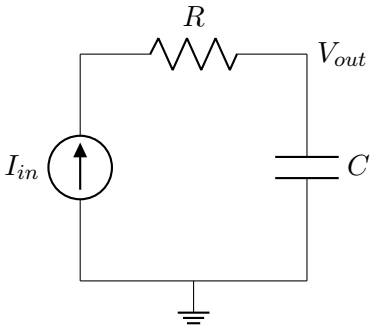


Figure 51: Circuit F

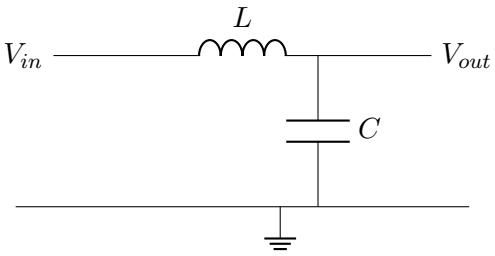


Figure 52: Circuit G

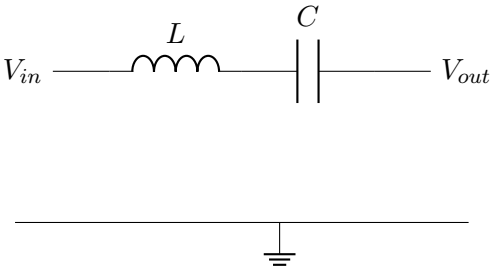


Figure 53: Circuit H

——— Answers Ahead ———

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

24.1 Circuit E: Series Capacitor with Current Source

Recall that for a capacitor, $I = C \cdot \frac{\mathrm{dV}}{\mathrm{dt}}$ and current through series elements is identical. This results in the solution given in Figure 54.

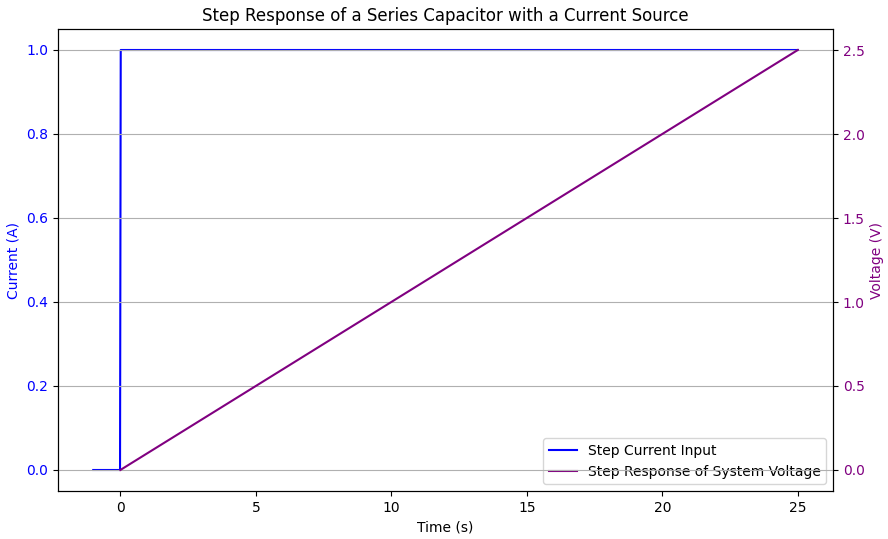


Figure 54: Step Response of a Circuit with a Series Capacitor and Current Source

24.2 Circuit F: Series RC with Current Source

As current through series elements is identical and for a resistor, $V = I \cdot R$, adding the series resistor does not affect the step response of this circuit. Circuit E and Circuit F have identical step response plots.

24.3 Circuit G: Series Inductor with Shunt Capacitor

This circuit results in LC resonance and can be found inside a buck converter circuit. The step response solution is given in Figure 55.

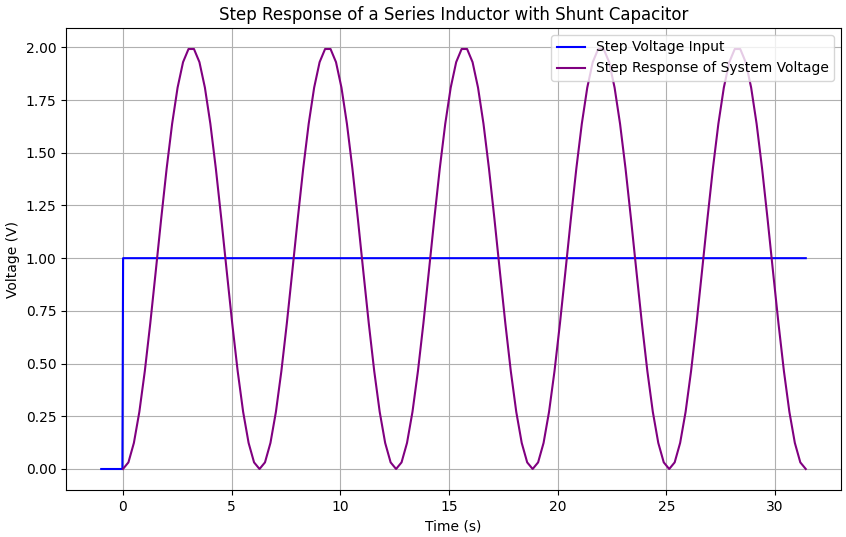


Figure 55: Step Response of a Circuit with a Series Inductor with Shunt Capacitor

24.4 Circuit H: Series Inductor and Capacitor

This circuit does not result in resonance as there is no current path for the two series elements. The step response solution is $V_{out} = V_{in}$.

25 Extra Practice: Solve the transfer function of the following circuits.

The following circuits are similar to those in Section 19 and are given as extra excersices with final answers only for checking.

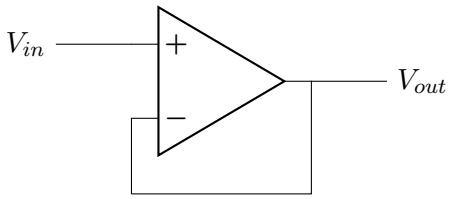


Figure 56: Circuit A

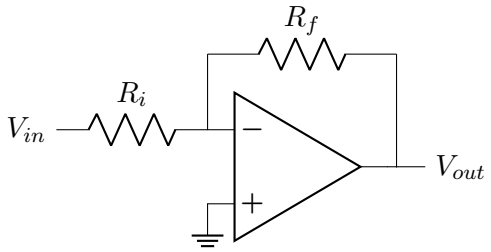


Figure 57: Circuit B

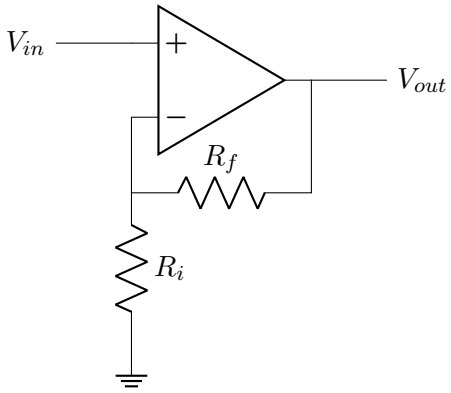


Figure 58: Circuit C

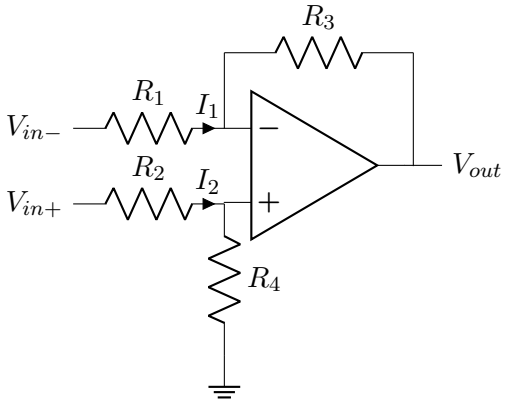


Figure 59: Circuit D

——— Answers Ahead ———

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

25.1 Solutions

- Circuit A is a unity gain amplifier where $\frac{V_{out}}{V_{in}} = 1$.
- Circuit B is an inverting amplifier where $\frac{V_{out}}{V_{in}} = -\frac{R_f}{R_i}$.
- Circuit C is a non-inverting amplifier where $\frac{V_{out}}{V_{in}} = 1 + \frac{R_f}{R_i}$.
- Circuit D is a differential amplifier where $V_{out} = -V_{in-} \cdot \frac{R_3}{R_1} + V_{in+} \cdot \frac{R_4}{R_2+R_4} \cdot \frac{R_1+R_3}{R_1}$. Consider a speacil case when $R_1 = R_2$ and $R_3 = R_4$ where the transfe function becomes $\frac{V_{out}}{V_{in+}-V_{in-}} = \frac{R_3}{R_1}$. Additionally, if $R_1 = R_2 = R_3 = R_4$ then $V_{out} = V_{in+} - V_{in-}$.

References

[1] S. E. Anderson, *Bit twiddling hacks*, Accessed: 2024-12-23, n.d. [Online]. Available: https://graphics.stanford.edu/~seander/bithacks.html.

[2] E. White, *Making Embedded Systems*, 2nd. O'Reilly Media, 2024.

[3] C. J. Myers, *Lecture 9: Interrupts in the 6812*, Lecture notes for ECE/CS 5780/6780: Embedded System Design, n.d. [Online]. Available: https://www.rose-hulman.edu/class/ee/hoover/ece331/old%20stuff/my%20csm12c32%20downloads/lec9-2x3.pdf.

[4] P. Koopman, *Better Embedded System Software, 1st Edition, Revised 2021*. 2021.

[5] J. Beningo, "Embedded basics: Peculiarities of the keyword `const`," 2015, Accessed: 2024-12-23. [Online]. Available: https://www.beningo.com/embedded-basics-peculiarities-of-the-keyword-const/.

[6] J. Beningo, *Using the `static` keyword in c*, Accessed: 2024-12-23, 2014. [Online]. Available: https://community.arm.com/arm-community-blogs/b/embedded-blog/posts/using-the-static-keyword-in-c.

[7] E. Staff, *The c keyword: `static`*, Accessed: 2024-12-23, 2014. [Online]. Available: https://www.embedded.com/the-c-keyword-static/.

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

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

[10] 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.

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

[12] 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.

[13] L. K. Seong, *Introduction to bit-banging*, Accessed: 2024-12-23, 2020. [Online]. Available: https://medium.com/@kslooi/introduction-to-bit-banging-46e114db3466.

[14] Cadence. "Can bus history at a glance." Accessed: 2024-12-25. (n.d.), [Online]. Available: https://resources.pcb.cadence.com/blog/2022-can-bus-history-at-a-glance.

[15] John Griffith - Texas Instruments. "What do can bus signals look like?" Accessed: 2024-12-25. (2023), [Online]. Available: https://www.ti.com/document-viewer/lit/html/SSZTCN3#:~:text=As%20you%20can%20see%2C%20in,potential%20(approximately%201.5V)..

[16] Gutten på Hemsen. "Differential signalling." Accessed: 2024-12-25, CC-BY-SA 4.0. (n.d.), [Online]. Available: https://en.wikipedia.org/wiki/Differential_signalling#/media/File:Differential_signal_fed_into_a_differential_amplifier.svg.

[17] A. aka (https://electronics.stackexchange.com/users/20218/andy-aka), *Noise can be differential or common!* Electrical Engineering Stack Exchange, URL:https://electronics.stackexchange.com/q/231301 (version: 2016-04-29), CC-BY-SA. eprint: https://electronics.stackexchange.com/q/231301. [Online]. Available: https://electronics.stackexchange.com/q/231301.

[18] Stefan-Xp. "Can bus — elektrische zweidrahtleitung." Accessed: 2024-12-25, CC-BY-SA 3.0. (n.d.), [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus#/media/File:CAN-Bus_Elektrische_Zweidrahtleitung.svg.

[19] S. Hymel, *What is a real-time operating system (rtos)?* Accessed: 2024-12-23, 2021. [Online]. Available: https://www.digikey.com/en/maker/projects/what-is-a-realtime-operating-system-rtos/28d8087f53844decafa5000d89608016.

[20] S. Hymel, *Introduction to rtos: Solution to part 11 - priority inversion*, Accessed: 2024-12-23, 2021. [Online]. Available: https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-11-priority-inversion/abf4b8f7cd4a4c70bece35678d1783