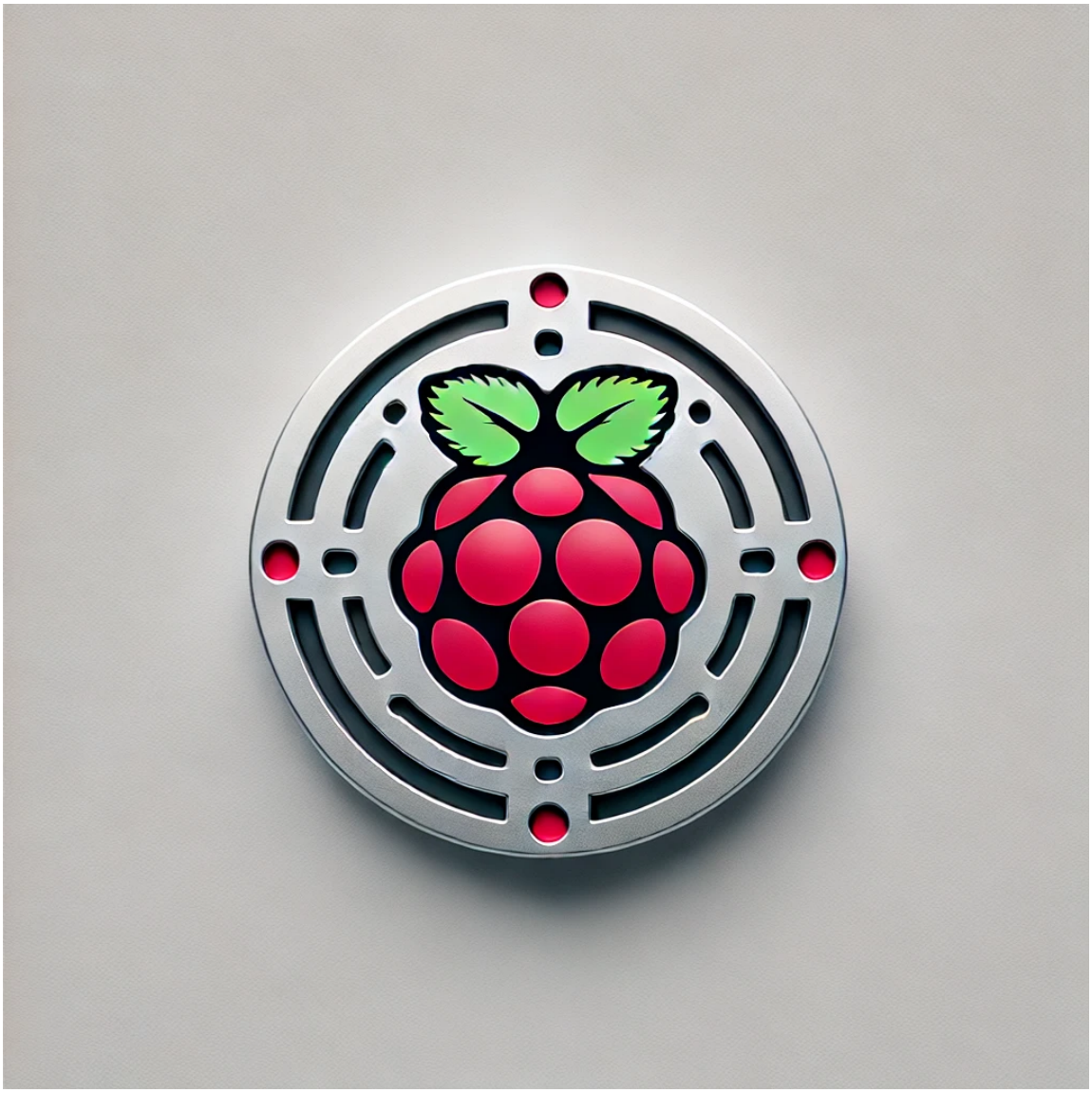
Arduino and Raspberry Pi Pico: Hands-On Approach



Navid Mohaghegh

Arduino and Raspberry Pi Pico: Hands-On Approach

Navid Mohaghegh

Contents

Appendices

Preface

The rapid evolution of embedded systems and hardware programming has brought once-specialized tools and concepts into the hands of hobbyists, students, and professionals alike. With the advent of versatile and affordable development platforms such as Arduino and the Raspberry Pi Pico, the barriers to entry for exploring hardware design and programming have been significantly lowered. This book aims to provide a clear and concise guide to help students and enthusiasts quickly orient themselves in this exciting field.

Why This Book?

This book serves as a practical starting point for those new to embedded programming or seeking a structured approach to understanding essential concepts. By addressing foundational topics such as C programming, GPIO handling, and advanced peripherals, it offers readers a solid base to build upon. With a hands-on approach, we bridge the gap between theory and application, ensuring that learners can directly implement what they read.

How to Use This Book

The book is organized into chapters that gradually introduce concepts in a structured manner:

- **Chapter 1: C Primer** – Covers the fundamentals of the C programming language, including syntax, data types, control structures, and memory management.
- **Chapter 2: Arduino and Raspberry Pi Pico** – Introduces the hardware features and capabilities of these platforms, along with tools like the Arduino IDE.

- **Chapter 3: Basic GPIOs** – Explains essential GPIO operations, such as reading input from buttons and controlling LEDs.
- **Chapter 4: Advanced GPIOs, Timers, and DMA** – Dives into advanced topics, including interrupts, timers, and direct memory access for efficient hardware interfacing.
- **Chapter 5: Advanced Peripherals** – Covers advanced features such as UART, SPI, I2C, and PWM, along with examples of real-world applications.

Each chapter includes practical examples and exercises to reinforce the material. Detailed explanations and codes ensure clarity, while modular content allows readers to explore topics in any order.

We also cover a hands-on introduction to serial protocol development, focusing on the foundational aspects of low-level communication systems. Using the NAMO Serial Protocol as a practical framework, it guides readers through key concepts such as message framing, serialization, and deserialization. Designed specifically for educational purposes, the protocol demonstrates how both text-based and binary communication can be structured and processed effectively. Readers will learn to implement real-world solutions for data exchange between microcontrollers and other systems, starting from the ground up.

The NAMO Serial Protocol emphasizes bi-directional communication with a customizable message structure and callback-driven design. Through this, readers can experiment with dynamic command handling and flexible messaging formats, ranging from human-readable text to efficient binary data. By addressing practical challenges such as message parsing, error handling, and acknowledgment responses, this book bridges the gap between theoretical understanding and practical application. Whether it's toggling LEDs, controlling motors, or managing sensors, the examples provided make hardware communication accessible and engaging.

With a focus on modularity and scalability, this book provides a practical resource for students, educators, and hobbyists. Each chapter builds on the last, enabling readers to progress from basic concepts to advanced topics such as multi-device communication and networked serial-to-TCP bridges. By the end, readers will not only grasp the technical details of protocol design but also appreciate the versatility and importance of low-level communication in modern embedded systems.

Who Should Read This Book?

This book is designed for:

- Students and educators looking for an introductory yet comprehensive resource on hardware programming.
- Hobbyists and makers eager to explore the capabilities of platforms like Arduino and Raspberry Pi Pico.
- Professionals seeking to brush up on the basics or explore advanced peripherals and techniques.

Acknowledgments

This work would not have been possible without the support of the embedded systems community and the incredible resources available online. Special thanks go to the developers of open-source tools and libraries that make embedded programming accessible to all.

We hope this book inspires you to dive into the world of hardware programming, build amazing projects, and push the boundaries of what's possible with these powerful yet affordable platforms.

Navid Mohaghegh
December 2024

Chapter 1

C Primer

1.1 Introduction to C Language

1.1.1 Structure of a C Program

A C program typically includes the following components:

- Header files inclusion using `#include`.
- Main function `int main()` as the program's entry point.
- Declarations and definitions of variables, constants, and functions.
- Execution statements within functions.

Here is an example of a simple C program:

```c
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

1.1.2 Compiling and Running a Program

To compile and run a C program:

1. Save the program as a `.c` file (e.g., `hello.c`).
2. Use a C compiler like `gcc` or an IDE such as Arduino IDE.

3. Compile using: `gcc hello.c -o hello`.

4. Run using: `./hello`.

1.2 Using `#include` for Libraries and Headers

The `#include` directive is used to include standard libraries or user-defined headers. A **header file** is a file with a `.h` extension that contains declarations for functions, macros, constants, and data types. Header files serve as an interface between different source files in a program, enabling modular development and code reuse. Instead of redefining common functionality in every source file, a header file centralizes the declarations, promoting better organization and maintainability of code.

Header files are used to declare the prototypes of functions that are implemented in other source files, enabling the compiler to verify function calls. For example, the standard library header `<stdio.h>` declares functions like `()` and `scanf()`, which are implemented in the standard library. By including the header file, the programmer informs the compiler of the function prototypes, avoiding compilation errors due to missing declarations.

Another significant use of header files is to define macros and constants using `#define` or `const`. This allows the programmer to reuse common values or code snippets across multiple files. Header files are also used to declare structures, enumerations, and global variables, ensuring consistency in their usage across different parts of a program.

To include a header file in a source file, the `#include` directive is used. Angle brackets (`< >`) are typically used for standard library headers, while double quotes (`" "`) are used for user-defined headers. For instance, `#include <stdio.h>` includes a standard library header, while `#include "myheader.h"` includes a custom header file created by the programmer.

Example:

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

int main() {
  double result = sqrt(16.0);
  printf("Square root of 16 is: %f\n", result);
  return 0;
}
```


1.3 Input and Output

1.3.1 Using `printf` for Output

The `printf` function outputs formatted data.

```
#include <stdio.h>

int main() {
  int age = 25;
  printf("Age: %d\n", age);
  return 0;
}
```

`printf` supports multiple format specifiers.

```
#include <stdio.h>

int main() {
  int num = 255;
  float pi = 3.14159;

  printf("Decimal: %d, Hex: %x, Float: %.2f\n", num, num, pi);
  return 0;
}
```

1.4 Variables and Data Types

In C programming, a **variable** is a named storage location in memory that holds a value that can be modified during the program's execution. Variables are fundamental to programming as they allow data to be stored, manipulated, and retrieved. Each variable in C must be declared with a specific **data type** that defines the kind of value the variable can hold, such as integers, floating-point numbers, characters, or more complex types like arrays and structures. The syntax for declaring a variable includes the data type followed by the variable name, for example: `int count;`. Proper use of variables ensures efficient memory usage and prevents type-related errors.

Data types in C are categorized into *primitive types*, *derived types*, and *user-defined types*. Primitive types include `int`, `float`, `char`, and `double`, which are used to store fundamental data like whole numbers, decimal numbers, and single characters. Derived types such as arrays, pointers, and functions are constructed from primitive types to manage more complex data. User-defined types like `struct` and `enum` allow programmers to create custom data representations tailored to specific requirements. Each data type occupies a specific amount of memory and has a range of values it can hold, which is determined by the system's architecture and compiler settings.

1.4.1 Primitive Data Types

C supports various primitive data types such as:

- `int`: Integer values.
- `float`: Single precision floating-point numbers (IEEE 754 standard).
- `double`: Double precision floating-point numbers (IEEE 754 standard).
- `char`: Single characters.

Example:

```c
#include <stdio.h>

int main() {
  int age = 20;
  float pi = 3.14;
  char initial = 'R';

  printf("Age: %d, Pi: %f, Initial: %c\n", age, pi, initial);
  return 0;
}
```

1.4.2 Constants and Enumerations

Constants are defined using `#define` or `const` keyword, and enumerations using `enum`.

```c
#include <stdio.h>

#define MAX_AGE 100
enum Days { MON, TUE, WED, THU, FRI, SAT, SUN };

int main() {
  const float PI = 3.14159;
  enum Days today = FRI;
  printf("Today is day number: %d\n", today);
  printf("Max Age: %d, Pi: %f\n", MAX_AGE, PI);
  return 0;
}
```

1.5 Conditions and Branching

1.5.1 `if`, `else if`, and `else`

Conditional statements control program flow based on conditions.


```c
#include <stdio.h>

int main() {
  int num = 10;

  if (num > 0)
  {
    printf("Yes!, We have a positive number\n");
  }
  else if (num == 0)
  {
    printf("Ah, we habe zero\n");
  }
  else
  {
    printf("Ah, negative number\n");
  }
  return 0;
}
```

1.5.2 `switch` Statements

The `switch` statement simplifies multi-condition scenarios.

```c
#include <stdio.h>

int main() {
  int day = 3;

  switch (day) {
    case 1: printf("Monday\n"); break;
    case 2: printf("Tuesday\n"); break;
    case 3: printf("Wednesday\n"); break;
    default: printf("Other day\n");
  }
  return 0;
}
```

1.6 Loops and Iterations

1.6.1 `for`, `while`, and `do-while` Loops

`for` loop example:

```c
#include <stdio.h>

// loop 5 times (index=0,1,2,3,4)
int main() {
  for (int index = 0; index < 5; index++) {
    printf("Our inex is:  %d\n", index);
  }
  return 0;
}
```


`while` loop example:

```c
#include <stdio.h>

int main() {
  int count = 0;
  while (count < 5) {
    printf("Count: %d\n", count);
    count++; //means count = count + 1
  }
  return 0;
}
```

Note that the **`while`** loop is an entry-controlled loop, meaning the condition is evaluated before the loop's body is executed. If the condition evaluates to `true`, the loop body runs; otherwise, the loop terminates immediately. This implies that if the condition is initially `false`, the body of the loop may never execute. The `while` loop is typically used when the number of iterations depends on a condition being met, and it's unclear whether the loop body should execute even once.

In contrast, the **`do-while`** loop is an exit-controlled loop, where the condition is evaluated after the loop's body executes. This guarantees that the loop body will execute at least once, regardless of the condition's value. This makes `do-while` loops suitable for scenarios where the body must run initially to determine whether further iterations are necessary. The key syntactical difference is that `do-while` requires a semicolon (`;`) after the condition, whereas `while` does not. Here is a `do-while` loop example:

```c
#include <stdio.h>

int main() {
  int count = 0;
  do {
    printf("Count: %d\n", count);
    count++;
  } while (count < 5);
  return 0;
}
```

1.7 Functions and Procedures

1.7.1 Defining and Calling Functions

In C programming, **functions** and **procedures** are fundamental building blocks for organizing code into reusable units. A **function** is a block of code designed to perform a specific task and can optionally return a value to the caller. A **procedure** is often used interchangeably with functions in C but typically refers to a function

that does not return a value (i.e., a function with a `void` return type). Functions and procedures promote modularity, making programs easier to read, maintain, and debug.

The syntax of a function in C consists of its **signature**, which includes the return type, function name, and a parameter list enclosed in parentheses. This is followed by the function body enclosed in curly braces {}. The general syntax is:

```
return_type function_name(parameter_list) {
  // function body
}
```

For example, `int add(int a, int b)` is a function that takes two integers as parameters and returns their sum.

A function's **signature** plays a critical role in defining its behavior and interactions. The return type specifies the type of value the function returns (e.g., `int`, `float`, `void`), while the parameter list defines the input values the function accepts. Functions without input parameters use an empty parameter list (`void`) or omit parameters entirely. For example, `void printNewLine()` is a function that takes no arguments and returns nothing (simply prints a new line on the console terminal), whereas `float calculateArea(float radius)` calculates and returns the area of a circle based on the given radius.

The **key keywords** used in functions include:

- `return`: Used to return a value from the function to the caller.
- `void`: Specifies that a function does not return any value.
- Data types such as `int`, `float`, `char`, etc., to specify the return type or parameter types.

For instance, a procedure can be implemented as `void printNewLine()` to print a a new line without returning any value.

Functions and procedures are invoked by their name, followed by parentheses containing any required arguments. For example, `int sum = add(5, 10);` calls the `add` function with arguments `5` and `10`. The program flow transfers to the function body, executes its statements, and optionally returns a value to the calling code. Procedures, being `void`, do not return values but may still modify external variables or produce side effects.

The signatures of functions and procedures define their behavior, while keywords like `return` and `void` determine their output. Here is an example:


```c
#include <stdio.h>

int add(int a, int b) {
  return a + b;
}

int main() {
  int result = add(5, 3);
  printf("Sum: %d\n", result);
  return 0;
}
```

1.8 Pointers and Memory

In C programming, a **pointer** is a variable that stores the memory address of another variable. Pointers are a powerful feature in C, enabling direct memory access and manipulation. They are declared using the `*` operator, and their value represents the address of a specific memory location. For instance, a pointer to an integer can be declared as:

```c
int *ptr;
```

A pointer to an integer allows the program to store the address of an integer variable and access or modify its value indirectly using the `*` operator. This is particularly useful in scenarios such as dynamic memory allocation, pass-by-reference in functions, and efficient manipulation of arrays.

= A **pointer to an integer**, declared as `int *ptr`. Please note `ptr` is a variable name that stores the memory address of another variable whose type is `int`. The pointer itself does not store an integer value directly but rather the location in memory where an integer is stored. This allows the program to indirectly access or modify the value of the integer through the pointer.

Let's do another example: We declare and set `int *p` consists of two parts:

- `int`: Specifies that the pointer will point to a memory location that stores an integer.
- `*`: Indicates that the variable `p` is a pointer, not a regular integer.

```c
int x = 10;    // Declare an integer variable
int *p = &x;   // Declare a pointer to integer and assign it the address of x
```

Here:

- `p` stores the memory address of the variable `x`.

- The value of `x` can be accessed or modified using the dereference operator `*`, as in `*p`.

Using `&x`, we obtain the address of `x`, which is stored in the pointer `p`. Using `*p`, we dereference the pointer to access the value stored at the memory address:

```
printf("Value of x: %d\n", *p);  // Output: 10
*p = 20;                         // Modify the value of x through the pointer
printf("Updated value of x: %d\n", x);  // Output: 20
```

To recap the above example, pointers allow indirect access to the value stored at the referenced memory address. The `&` operator is used to obtain the address of a variable, while the `*` operator (dereference operator) is used to access or modify the value at the memory address. For example:

```
int x = 10;
int *ptr = &x;  // Pointer stores the address of x
printf("Value of x: %d\n", *ptr); // Dereference to access value of x
*ptr = 20;  // Modify the value of x through the pointer
```

In this example, `&x` gets the address of `x`, and `*ptr` accesses or modifies the value stored at that address.

The **arrow operator (`->`)** is used with pointers to access members of a structure. When a pointer points to a structure, the `->` operator simplifies access to its members, as shown below:

```
struct Point {
  int x, y;
};
struct Point p = {10, 20};
struct Point *ptr = &p;
printf("x: %d, y: %d\n", ptr->x, ptr->y);
```

Here, `ptr->x` is equivalent to `(*ptr).x`, but the arrow operator is more concise and commonly used.

One critical application of pointers is in **pass-by-reference**. In C, functions typically use **pass-by-value**, where a **copy** of the argument is passed to the function. Changes made to the parameter inside the function **do not affect** the original argument. For example:

```
void modifyValue(int x) {
  x = 20;  // This change won't affect the original variable
}
int main() {
  int num = 10;
  modifyValue(num);
  printf("Value of num: %d\n", num);  // Output will be still 10
}
```


In contrast, **pass-by-reference** uses pointers to allow a function to modify the original variable. Instead of passing a copy, the address of the variable is passed, and the function operates on the memory location directly. For example:

```
void modifyValue(int *x) {
  *x = 20;  // Modify the value at the referenced address
}
int main() {
  int num = 10;
  modifyValue(&num);  // Pass the address of num
  printf("Value of num: %d\n", num);  // Output: 20
}
```

Pointers are also crucial for dynamic memory allocation, where memory is allocated at runtime using Memory Allocation functions like `malloc` and `calloc`. These functions return a pointer to the allocated memory, which can be used to access and manipulate the data. For example:

```
int *arr = (int *)malloc(5 * sizeof(int));  // Allocate memory for an array
for (int i = 0; i < 5; i++) {
  arr[i] = i + 1;  // Populate the array
}
free(arr);  // Release the allocated memory
```

Here, pointers enable efficient memory management by dynamically allocating and freeing resources.

Another common use case for pointers is working with arrays. In C, the name of an array acts as a pointer to its first element. This allows for pointer arithmetic to traverse the array. For example:

```
int arr[] = {10, 20, 30};
int *ptr = arr;  // Points to the first element
printf("First element: %d\n", *ptr);
printf("Second element: %d\n", *(ptr + 1));  // Pointer arithmetic
```

While pointers are powerful, they require careful handling to avoid common pitfalls such as **dangling pointers** (pointers referencing deallocated memory), **null pointers** (pointers that do not reference any memory), and **buffer overflows**. Proper use of pointers ensures efficient and robust programs.

1.8.1 Declaring and Using Pointers

As we discussed pointers are variables that store the memory address of another variable. You can print that memory address with printf and `%p`:

```
#include <stdio.h>

int main() {
```



```
  int number = 42;
  int *ptr = &number;

  printf("Value: %d, Address: %p\n", *ptr, ptr);
  return 0;
}
```

1.8.2 Pointer Arithmetic

Pointer arithmetic allows navigating through memory locations.

```
#include <stdio.h>

int main() {
  int arr[] = {10, 20, 30};
  int *ptr = arr;

  for (int i = 0; i < 3; i++) {
    printf("Value: %d, Address: %p\n", *(ptr + i), ptr + i);
  }
  return 0;
}
```

1.8.3 Passing by Reference

Passing by reference allows a function to modify the original variable.

```
#include <stdio.h>

void increment(int *value) {
  (*value)++;
}

int main() {
  int num = 5;
  increment(&num);

  printf("Incremented value: %d\n", num);
  return 0;
}
```

1.9 Structures in C

1.9.1 Declaring and Using `struct`

Structures group multiple variables under a single name.

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



```c
struct Point {
  int x;
  int y;
};

int main() {
  struct Point p1 = {3, 4};

  printf("Point: (%d, %d)\n", p1.x, p1.y);
  return 0;
}
```

1.9.2 Nested Structures and Accessing Members

Structures can be nested and accessed accordingly.

```c
#include <stdio.h>

struct Point {
  int x, y;
};

struct Rectangle {
  struct Point topLeft;
  struct Point bottomRight;
};

int main() {
  struct Rectangle rect = {{0, 0}, {10, 10}};
  printf("Top-left: (%d, %d), Bottom-right: (%d, %d)\n",
  rect.topLeft.x, rect.topLeft.y,
  rect.bottomRight.x, rect.bottomRight.y);
  return 0;
}
```

1.10 Using Libraries in C

1.10.1 Standard C Libraries (`stdio.h`, `math.h`)

The standard library provides reusable functionality.

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

int main() {
  double result = sqrt(49.0);
  printf("Square root: %.2f\n", result);
  return 0;
}
```


1.10.2 Using `stdlib.h` for Utility Functions

Utility functions like random number generation are available in `stdlib.h`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
  srand(time(NULL));
  printf("Random number: %d\n", rand());
  return 0;
}
```

1.10.3 Using `string.h` for String Manipulation

String functions such as `strcpy` and `strlen` are provided by `string.h`.

```
#include <stdio.h>
#include <string.h>

int main() {
  char str1[20] = "Hello";
  char str2[20];

  strcpy(str2, str1);
  printf("Copied string: %s\n", str2);
  printf("Length: %lu\n", strlen(str1));
  return 0;
}
```

1.11 Example: A Very Basic Calculator

A simple calculator that uses user input, conditions, and functions.

```
#include <stdio.h>

float calculate(float a, float b, char op) {
  switch (op) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/': return a / b;
    default: return 0;
  }
}

int main() {
  float num1, num2;
  char op;

  printf("Enter expression (e.g., 5 + 3): ");
```



```
  scanf("%f %c %f", &num1, &op, &num2);

  printf("Result: %.2f\n", calculate(num1, num2, op));
  return 0;
}
```

1.12 Function Pointers

A **function pointer** is a pointer variable that stores the address of a function. This allows functions to be passed as arguments to other functions, enabling flexible and reusable code structures at runtime. Function pointers are particularly useful in scenarios where different behaviors are needed at runtime, such as callback functions, event handling, or sorting with custom comparison logic and runtime selection.

A function pointer is declared using the syntax `return_type (*function_pointer_name)(paramet` where `function_pointer_name` stores the address of a function that matches the specified signature. For example:

```
int (*func_ptr)(int, int);
```

This declares a pointer `func_ptr` to a function that takes two `int` arguments and returns an `int`. The function pointer can then be assigned the address of a function, and the function can be invoked using the pointer.

1.12.1 Example: Sorting Array with `qsort`

Using the standard library's quick sort `qsort` function to sort a series of numbers:

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
  return (*(int *)a - *(int *)b);
}

int main() {
  int arr[] = {20, 5, 12, 8, 1};
  int n = sizeof(arr) / sizeof(arr[0]);

  qsort(arr, n, sizeof(int), compare);

  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  return 0;
}
```


Explanation: Above code demonstrates the use of a function pointer in the `qsort` function. The `compare` function, which implements custom comparison logic, is passed as a function pointer to `qsort`. The signature of `compare` matches the requirements of `qsort`, taking two generic arbitrary pointers (e.g., `const void *` as arguments) and returning if the given numbers in arguments are smaller, equal or biger than each other (e.g., an `int` as an indicator). The function dereferences the arguments, casts them to `int *`, and performs subtraction to determine the ordering:

```
int compare(const void *a, const void *b) {
  return (*(int *)a - *(int *)b);
}
```

In the `main` function, `qsort` is called to sort an array of integers. The `qsort` function uses the `compare` function pointer to determine the relative order of elements:

```
qsort(arr, n, sizeof(int), compare);
```

Qsort takes `arr` which is the array to be sorted, `n` is the number of elements, `sizeof(int)` specifies the size of each element, and `compare` is the function pointer for the custom comparison logic. After sorting, the array elements are printed in ascending order.

1.13 Advanced Pointers and Memory Management

1.13.1 Dynamic Memory Allocation

Dynamic memory allocation allows you to request memory during runtime using `malloc`, `calloc`, and `realloc`. The `malloc` function allocates a specified amount of memory but does not initialize it, meaning the allocated memory contains garbage or stale values left in the memory from the past operations. In contrast, `calloc` allocates memory for an array of elements, initializes all bytes to zero, and requires two arguments: the number of elements and the size of each element. The `realloc` function is used to resize previously allocated memory, either expanding or shrinking the block, and it preserves the existing data up to the new size. While `malloc` and `calloc` are used for fresh memory allocation, `realloc` modifies existing allocations. Additionally, `malloc` and `calloc` return `NULL` if memory allocation fails, and the use of `realloc` on `NULL` behaves like `malloc`.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *arr = malloc(5 * sizeof(int));
```



```
  if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  for (int i = 0; i < 5; i++) {
    arr[i] = i + 1;
    printf("%d ", arr[i]);
  }
  free(arr);  // Free allocated memory
  return 0;
}
```

1.13.2 Using `calloc` for Zero-Initialized Memory

The `calloc` function allocates and initializes memory to zero.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *arr = calloc(5, sizeof(int));
  if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);  // All elements initialized to 0
  }
  free(arr);
  return 0;
}
```

1.13.3 Resizing Memory with `realloc`

The `realloc` function adjusts the size of previously allocated memory.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *arr = malloc(5 * sizeof(int));
  if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  arr = realloc(arr, 10 * sizeof(int));
  if (arr == NULL) {
    printf("Memory reallocation failed!\n");
    return 1;
  }
```



```
  for (int i = 0; i < 10; i++) {
    arr[i] = i + 1;
    printf("%d ", arr[i]);
  }
  free(arr);
  return 0;
}
```

1.14 Working with Strings in C

1.14.1 String Copying and Concatenation

C provides `strcpy` and `strcat` for manipulating strings. These string manipulation functions are operating on an assumption that the given string is null-terminated character arrays. `strcpy` copies a source string into a destination buffer, while `strcat` appends a source string to the end of a destination string. However, both functions assume that the destination buffer is large enough to hold the resulting string, and they do not perform bounds checking or null termination checks. If the destination buffer is too small, these functions can cause **buffer overflows**, leading to undefined behavior, memory corruption, or security vulnerabilities.

To address these issues, safer alternatives like `strncpy` and `strncat` are provided. These functions allow the programmer to specify the maximum number of characters to copy or append, preventing buffer overflows. For example, `strncpy(dest, src, n)` copies at most `n` characters from the source to the destination, ensuring that the buffer is not overwritten. Similarly, `strncat(dest, src, n)` appends up to `n` characters from the source to the destination string. While these safer functions mitigate some risks, they require careful handling of null terminators and may not automatically append them in some cases. This necessitates additional checks to ensure proper string termination.

```
#include <stdio.h>
#include <string.h>

int main() {
  char str1[50] = "Hello, ";
  char str2[] = "World!";

  strcat(str1, str2);
  printf("%s\n", str1);  // Output: Hello, World!
  return 0;
}
```


1.14.2 String Comparison

String comparison is achieved using `strcmp`.

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

int main() {
  char str1[] = "apple";
  char str2[] = "banana";

  int result = strcmp(str1, str2);
  if (result < 0) {
    printf("%s comes before %s\n", str1, str2);
  } else if (result > 0) {
    printf("%s comes after %s\n", str1, str2);
  } else {
    printf("Strings are equal.\n");
  }
  return 0;
}
```

1.15 File Handling in C

1.15.1 Reading and Writing Files

C supports file operations using the `FILE` structure and associated functions like `fopen`, `fprintf`, and `fscanf`.

```c
#include <stdio.h>

int main() {
  FILE *file = fopen("data.txt", "w");
  if (file == NULL) {
    printf("Error opening file.\n");
    return 1;
  }

  fprintf(file, "Hello, File!\n");
  fclose(file);

  file = fopen("data.txt", "r");
  char line[100];
  if (fgets(line, sizeof(line), file)) {
    printf("Read from file: %s", line);
  }
  fclose(file);
  return 0;
}
```


1.15.2 Binary File Operations

Binary file operations use `fread` and `fwrite`.

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

int main() {
  int data[] = {1, 2, 3, 4, 5};
  FILE *file = fopen("binary.dat", "wb");
  if (file == NULL) {
    printf("Error opening file.\n");
    return 1;
  }

  fwrite(data, sizeof(int), 5, file);
  fclose(file);

  int buffer[5];
  file = fopen("binary.dat", "rb");
  fread(buffer, sizeof(int), 5, file);
  fclose(file);

  for (int i = 0; i < 5; i++) {
    printf("%d ", buffer[i]);
  }
  return 0;
}
```

1.16 Bit Manipulation in C

1.16.1 Setting and Clearing Bits

Bitwise operations like AND, OR, and XOR are used for manipulating individual bits.

```c
#include <stdio.h>

int main() {
  unsigned int num = 0b00001111;

  // Set the 5th bit
  num |= (1 << 4);

  // Clear the 2nd bit
  num &= ~(1 << 1);

  printf("Result: 0b%08b\n", num);
  return 0;
}
```


1.16.2 Toggling and Checking Bits

You can toggle bits using XOR and check bit states using AND.

```c
#include <stdio.h>

int main() {
  unsigned int num = 0b01010101;

  // Toggle the 3rd bit
  num ^= (1 << 2);

  // Check if the 4th bit is set
  if (num & (1 << 3)) {
    printf("4th bit is set.\n");
  } else {
    printf("4th bit is not set.\n");
  }

  printf("Result: 0b%08b\n", num);
  return 0;
}
```

1.17 Debugging and Error Handling

1.17.1 Error Codes and Return Values

Functions often return error codes to indicate success or failure.

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

int main() {
  FILE *file = fopen("nonexistent.txt", "r");
  if (file == NULL) {
    printf("Error: %s\n", strerror(errno));
    return 1;
  }

  fclose(file);
  return 0; // zero error happened in main function
}
```

1.18 Recursion and Tail Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem until a base condition is met, as seen in tasks like calculating factorials or traversing data structures. **Tail recursion**, a specific type of recursion, occurs when the recursive call is the last operation performed in the function, enabling optimization by reusing the current function's stack frame and improving efficiency.


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

int factorial(int n) {
  //Assertions are used to enforce assumptions
  assert(n >= 0);  // Ensure input is non-negative
  return (n <= 1) ? 1 : n * factorial(n - 1);
}

int main() {
  printf("Factorial of 5: %d\n", factorial(5));
  return 0;
}
```

1.19 Preprocessor Directives in C

1.19.1 `#define` for Macros

As we quickly discussed, the `#define` directive creates symbolic constants or macros that are resolved at compile time.

```c
#include <stdio.h>

#define PI 3.14159
#define AREA_OF_CIRCLE(radius) (PI * (radius) * (radius))

int main() {
  printf("Area: %.2f\n", AREA_OF_CIRCLE(5));
  return 0;
}
```

1.19.2 `#ifdef`, `#ifndef`, and `#endif`

Conditional compilation is useful for debugging and platform-specific code.

```c
#include <stdio.h>

#define DEBUG

int main() {
  #ifdef DEBUG
  printf("Debugging mode is ON\n");
  #endif

  #ifndef RELEASE
  printf("Release mode is OFF\n");
  #endif

  return 0;
}
```


1.19.3 Using `#include` for Modular Code

The `#include` directive allows splitting code into headers and source files.

```c
// mymath.h
#ifndef MYMATH_H
#define MYMATH_H

int add(int a, int b);
int subtract(int a, int b);

#endif


// mymath.c
#include "mymath.h"

int add(int a, int b) {
  return a + b;
}

int subtract(int a, int b) {
  return a - b;
}


// main.c
#include <stdio.h>
#include "mymath.h"

int main() {
  printf("Sum: %d\n", add(5, 3));
  printf("Difference: %d\n", subtract(5, 3));
  return 0;
}
```

1.20 Multi-Dimensional Arrays

Arrays store multiple elements of the same type.

```c
#include <stdio.h>

int main() {
  int arr[5] = {1, 2, 3, 4, 5};

  for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  return 0;
}
```

Two-dimensional arrays represent grids or matrices.


```c
#include <stdio.h>

int main() {
  int matrix[2][2] = {{1, 2}, {3, 4}};

  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

1.20.1 Passing Arrays to Functions

Arrays can be passed to functions as pointers.

```c
#include <stdio.h>

void printArray(int *arr, int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
}

int main() {
  int arr[5] = {10, 20, 30, 40, 50};
  printArray(arr, 5);
  return 0;
}
```

1.21 Debugging Techniques

1.21.1 Using Logs for Debugging

Print statements can be used for runtime debugging.

```c
#include <stdio.h>

int main() {
  int num = 5;
  printf("Debug: num = %d\n", num);
  return 0;
}
```

1.21.2 Debugger Tools

Use debugging tools like GNU Debugger (GDB) to step through code and inspect variables.


```
gcc -g program.c -o program
gdb ./program
```

Within GDB:

```
run           # Start program
break main    # Set breakpoint at main
next          # Step over to the next line
print var     # Print the value of a variable
```


Chapter 2

Arduino and Raspberry Pi Pico

2.1 Introduction to Raspberry Pi Pico (RP2040 and RP2350)

The Raspberry Pi Pico is a compact and powerful microcontroller board featuring the RP2040 chip. Designed for versatility and efficiency, the Pico is suitable for a wide range of embedded applications, including IoT, robotics, and real-time systems. The RP2040 chip, developed by Raspberry Pi, uses a dual-core Arm Cortex-M0+ processor running at 133 MHz, with 264 KB of SRAM and up to 16 MB of external flash memory. The board includes GPIO pins, I2C, SPI, and UART interfaces, making it ideal for interfacing with sensors, actuators, and peripherals.

2.1.1 RP2040: Features and Capabilities

The RP2040 microcontroller at the heart of the Pico is notable for its performance, low power consumption, and affordability. It features a programmable input/output (PIO) subsystem that enables precise timing and protocol emulation, making it highly adaptable for custom communication protocols. With support for DMA (Direct Memory Access) and a flexible clock system, the RP2040 excels in real-time and computationally intensive applications. It also includes a rich set of peripherals, such as ADC, PWM, and multiple I/O banks, catering to diverse project needs. We will review all of these features in subsequent chapters.

2.1.2 Raspberry Pi Pico 2 and RP2350

The Raspberry Pi Pico 2, powered by the RP2350 chip, is an upgraded version of the original Pico, offering enhanced performance and expanded functionality. The RP2350 features a dual-core Arm Cortex-M4 processor, providing higher processing power and improved energy efficiency compared to the RP2040. With additional SRAM and flash memory, the Pico 2 supports more complex applications, such as machine learning, advanced signal processing, and multitasking. The RP2350 also includes advanced hardware acceleration for cryptography and DSP tasks, broadening its potential applications in security and high-performance computing.

Both the Pico and Pico 2 are well-suited for a wide range of embedded systems projects. The Pico, with its RP2040 chip, is a popular choice for DIY electronics, educational purposes, and low-power IoT applications. It has been widely used in sensor networks, motor control systems, and basic audio processing. The Pico 2, with its more powerful RP2350, targets more demanding applications, such as IoT edge computing, simple machine vision, and basic robotics, where higher computational capabilities and advanced peripherals are required.

2.1.3 Pico's Hardware Capabilities

2.1.3.0.1 Processor and Core Features

- Dual-core system with Cortex-M33 or Hazard3 processors.
- Maximum clock speed of 150 MHz.
- Integrated coprocessors for tasks like GPIO, double-precision floating-point operations, and redundancy.

2.1.3.0.2 Memory Architecture

- 520 kB on-chip SRAM divided into 10 independent banks for parallel access.
- 8 kB of one-time-programmable (OTP) storage for secure data and configurations.
- Support for up to 16 MB of external QSPI flash/PSRAM with optional secondary chip-select.

2.1.3.0.3 Peripheral Interfaces

- **UARTs:** 2 serial ports for communication.
  - You can use Universal Asynchronous Receiver-Transmitter as your serial communication interface (SCI) by transmitting and receiving data one bit at a time using a start bit, data bits, and a stop bit. It operates asynchronously, meaning it does not require a clock signal, making it suitable for point-to-point communication over longer distances. SCI is widely used in embedded systems for debugging, communication with peripherals like GPS modules, and connecting devices like PCs via USB-to-UART bridges.
- **SPI and I2C Controllers:** Two instances each for peripheral communication.
  - SPI is a high-speed, full-duplex communication protocol designed for short-distance communication between a master device and multiple slave devices. It uses separate lines for data (MOSI and MISO), a clock (SCLK), and a chip-select signal (CS) for each slave device. SPI's simplicity and speed make it ideal for applications requiring fast data transfer, such as connecting advacned sensors, LCD displays, or memory devices to a microcontroller.
  - I2C is a two-wire communication protocol that uses a clock line (SCL) and a data line (SDA) to enable communication between multiple master and slave devices on the same bus. It supports addressing and allows multiple devices to share the same lines, making it efficient for applications with limited GPIO availability. Common use cases include interfacing simpler sensors that does not require very high speeds such as temperature sensors, EEPROMs, and other simpler peripherals.
- **PWM Channels:** 24 channels of Pulse Width Modulation for precision control. PWM is a technique used to control analog values using digital signals by modulating the width of pulses in a fixed frequency signal. It is widely used for motor speed control, LED brightness adjustment, and generating audio signals.
- **Programmable IO (PIO):** 12 state machines across 3 PIO blocks, capable of handling custom protocols, signaling, and interfaces.
- **USB 1.1 Controller:** Embedded PHY for host and device connectivity.

2.1.3.0.4 Analog and GPIO

- Depending on the package, up to 48 general purpose input/output pins.
- Analog inputs range from 4 to 8 channels.

2.1.3.0.5 Security Features

- TrustZone-based partitioning of secure and non-secure states.
- Hardware hashing and SHA-256 acceleration.
- Secure boot with optional boot signing enforced by mask ROM.

2.1.3.0.6 Power Management

- Switched-mode power supply for efficient core voltage management.
- Configurable low-power states and sleep modes.

2.1.3.0.7 Bus and Fabric

- AHB5 crossbar for high bandwidth, supporting six simultaneous transfers.
- DMA controller to offload repetitive data transfers.

2.1.3.0.8 Debug and Development

- Support for SWD and JTAG for debugging and trace capabilities.
- ROM with a USB bootloader for UF2 drag-and-drop programming.

2.1.4 Understanding GPIO Pins and Pinouts

The Raspberry Pi Pico has many pins, of which 26 are GPIO pins. These pins can be configured for digital input/output, analog input, or special functions like SCI/SPI/I2C or PWM. The pinout is as follows:

- **Power Pins:** 3.3V and 5V power output, GND. Depend on the power source, you may not be able to drain more than 350 mA without damaging your USB host system (e.g., your laptops).

- **Digital GPIO:** Configurable for input or output.
- **Analog Pins:** ADC channels on GPIO26-28.
- **Communication Pins:** UART, I2C, SPI.
- **Special Pins:** GP15 for onboard LED, USB power detection.

```c
// Example: Configuring a GPIO pin for output
#include <stdio.h>
#include "pico/stdlib.h"

int main() {
  const uint LED_PIN = 25;  // Onboard LED
  gpio_init(LED_PIN);
  gpio_set_dir(LED_PIN, GPIO_OUT);

  while (true) {
    gpio_put(LED_PIN, 1);  // Turn on LED
    sleep_ms(500);
    gpio_put(LED_PIN, 0);  // Turn off LED
    sleep_ms(500);
  }
  return 0;
}
```

2.1.5 Raspberry Pi Pico SDK and Drivers

The Raspberry Pi Pico SDK is a comprehensive development framework designed to facilitate programming the Pico and other RP2040/RP2350-based boards in C and C++. It provides low-level access to the microcontroller's features and peripherals, enabling developers to create applications ranging from simple GPIO toggling to complex multi-peripheral systems. The SDK also includes support for libraries and APIs that abstract hardware details, offering both high-level and low-level drivers to suit a wide range of development needs. It includes various components that provide access to hardware peripherals, enable debugging, and facilitate efficient development. Below is a detailed list of the key components that constitute the Pico SDK:

- Hardware Abstraction Layer (HAL).
- Board Support Package (BSP).
- Standard C Library (Newlib Integration).
- PIO Support and Assembler.
- Multicore Programming Utilities.

- FreeRTOS Support.
- USB Device Library (TinyUSB).
- Debugging Tools and SWD Support.
- File System Integration (FatFS).
- DMA Library.
- CMake-based Build System.
- Examples and Templates.
- Documentation and API References.

We will briefly discuss each feature.

2.1.5.1 Hardware Abstraction Layer (HAL)

High-level drivers in the Pico SDK simplify the intricacies of hardware interaction by providing user-friendly APIs for common tasks. These drivers encapsulate complex hardware operations, making development more accessible and efficient, particularly for developers prioritizing ease of use or working on applications requiring standard functionality. At the core of these high-level drivers is the Hardware Abstraction Layer (HAL), which provides a unified interface for interacting with RP2040 peripherals such as GPIO, UART, I2C, and SPI hardware capabilities and protocols. By abstracting hardware complexities into intuitive APIs, the HAL ensures ease of development while retaining the flexibility of low-level access for advanced use cases. In the subsequent chapters, we will explore each of these peripherals in detail.

- GPIO HAL: Configure and control GPIO pins for input, output, or alternate functions.
- UART HAL: Simplify serial communication with functions for transmission and reception.
- I2C and SPI HALs: Facilitate communication with external devices via I2C and SPI protocols.
- PWM HAL: Generate PWM signals for motor control, LED dimming, etc.
- ADC HAL: Interface with the Pico's built-in 12-bit ADC.

High-level drivers in the Pico SDK abstract the intricacies of hardware interaction, providing user-friendly APIs for common tasks. These drivers simplify development by encapsulating complex hardware operations and are suitable for developers who prioritize ease of use or are working on applications requiring standard functionality.

2.1.5.1.1 GPIO Driver The GPIO driver provides easy access to the Pico's General-Purpose Input/Output (GPIO) pins. It allows developers to configure pins as input, output, or alternate functions with minimal code. For example:

```
gpio_init(25);              // Initialize GPIO pin 25
gpio_set_dir(25, GPIO_OUT); // Set pin 25 as output
gpio_put(25, 1);            // Set pin 25 to HIGH
```

2.1.5.1.2 UART Driver The UART driver abstracts serial communication, enabling data transfer via the Pico's UART hardware. It supports configurable baud rates, data bits, stop bits, and parity settings. Example usage:

```
uart_init(uart0, 9600);  // Initialize UART0 with 9600 baud
uart_puts(uart0, "Hello, UART!\n");
```

2.1.5.1.3 I2C and SPI Drivers High-level I2C and SPI drivers simplify communication with external devices. These drivers handle initialization, data transfer, and error checking. For instance, initializing SPI:

```
spi_init(spi0, 500000);  // Initialize SPI0 with 500 kHz
gpio_set_function(18, GPIO_FUNC_SPI);  // SCK
gpio_set_function(19, GPIO_FUNC_SPI);  // MOSI
```

2.1.5.1.4 PWM Driver The PWM driver facilitates the generation of pulse width modulation signals for controlling devices like motors and LEDs. Example:

```
gpio_set_function(25, GPIO_FUNC_PWM);   // Set GPIO 25 for PWM
uint slice = pwm_gpio_to_slice_num(25);
pwm_set_gpio_level(25, 128);            // Set duty cycle to 50%
```

2.1.5.1.5 Low-Level Drivers

Low-level drivers in the Pico SDK provide direct access to the RP2040's hardware registers, offering granular control over peripherals. These drivers are ideal for developers requiring fine-tuned performance or working on non-standard applications where higher-level abstractions are insufficient.

2.1.5.1.6 Hardware Registers

Direct access to registers allows precise configuration and control. For example, setting a GPIO pin directly:

```
hw_write_masked(&gpio_hw->out, 1u << 25, 1u << 25);  // Set GPIO 25 HIGH
```

2.1.5.1.7 PIO (Programmable I/O) Driver

The PIO driver allows developers to implement custom protocols and hardware behaviors. Using PIO assembly, developers can create state machines for tasks like generating waveforms or emulating communication protocols. Example of a simple PIO program:

```
// Define PIO assembly program
.program pwm_pio
pull
out pins, 1
.end
```

2.1.5.1.8 DMA (Direct Memory Access) Driver

The DMA driver enables high-speed data transfer between peripherals and memory without CPU intervention. This is crucial for performance-critical applications, such as streaming data from sensors to memory.

2.1.5.1.9 Differences Between High-Level and Low-Level Drivers

— Feature — High-Level Drivers — Low-Level Drivers — ——————————————————————————— — Abstraction — Encapsulates hardware complexity — Provides direct hardware access — — Ease of Use — User-friendly APIs — Requires in-depth hardware knowledge — — Flexibility — Limited to predefined functionality — Full control over hardware — — Performance — Moderate — High — — Examples — GPIO, UART, I2C — PIO, DMA —

2.1.5.2 Board Support Package (BSP)

The BSP includes configurations and definitions for specific boards, such as the Raspberry Pi Pico and other variants by third party vendors. It handles pin mappings, clock setups, and other board-specific details to ensure compatibility and simplify setup.

2.1.5.3 Standard C Library (Newlib Integration)

The SDK integrates the Newlib standard C library for essential functions like memory allocation (`malloc`, `free`), string manipulation (`strcpy`, `strcmp`), and input/output operations (`printf`, `scanf`).

2.1.5.4 PIO (Programmable I/O) Support

PIO is a unique feature of the Pico MCUs that enables the implementation of custom hardware protocols. We will also discuss PIO in subsequent chapters. The SDK includes:

- PIO Assembler: Define and compile custom PIO programs.
- PIO Library: Manage state machines, configure pins, and load PIO programs.

2.1.5.5 Multicore Support

The SDK provides tools to utilize the ARM cores effectively. It includes APIs for:

- Starting and managing the second core (`multicore_launch_core1()`).
- Inter-core communication using FIFOs.

2.1.5.6 FreeRTOS Integration

The SDK supports FreeRTOS for real-time multitasking. It allows developers to create multiple tasks, manage priorities, and synchronize resources using semaphores and queues.

2.1.5.7 USB Device Library (TinyUSB Integration)

The USB device library enables the Pico to act as a USB device, such as a keyboard, mouse, or mass storage device. It supports:

- HID (Human Interface Device) functionality.
- CDC (Communication Device Class) for virtual COM ports.
- Mass storage for USB flash drive-like applications.

2.1.5.8 Debugging and SWD Support

The SDK includes tools for debugging and profiling:

- SWD Debugging: Debug programs using Serial Wire Debug.
- printf and Log Support: Enhanced printf functions for debugging.

2.1.5.9 File System Integration

The SDK supports file systems like FatFS for applications requiring data logging or SD card integration. It simplifies file operations like creating, reading, and writing files.

2.1.5.10 DMA (Direct Memory Access) Library

The DMA library enables efficient data transfer between peripherals and memory without CPU involvement. It is crucial for performance-critical applications.

2.1.5.11 SDK Build System (CMake-based)

The SDK uses LLVM CMake program to manage builds, dependencies, and configurations. The build system simplifies project setup and ensures compatibility with the Pico's toolchain.

2.1.5.12 Documentation, API References, Examples, and Templates

The SDK also includes a wide range of examples and project templates demonstrating the usage of peripherals, multicore programming, USB, and more. These examples provide a quick start for developers. Also, a comprehensive documentation and API references are included in the SDK, providing in-depth explanations of functions, libraries, and hardware features.

2.1.5.13 SDK Integration and Ecosystem

The Pico SDK integrates easily with external libraries and tools, such as FatFS for file system management or TinyUSB for USB device functionality. It also supports debugging via SWD (Serial Wire Debug) and includes comprehensive documentation and examples, making it an excellent choice for beginners.

In conclusion, the Raspberry Pi Pico SDK provides a rich set of high-level and low-level drivers to accommodate various development requirements. High-level drivers simplify tasks and accelerate development, while low-level drivers offer the flexibility and performance needed for custom or advanced applications. By understanding the capabilities of both driver types, developers can harness the full potential of the RP2040 microcontroller.

2.2 Arduino Overview

An Arduino project refers to any hardware and software combination developed using the Arduino ecosystem, which includes a wide range of microcontroller boards, sensors, actuators, and an easy-to-use Integrated Development Environment (IDE). Arduino projects are highly versatile and range from simple LED blink programs to complex robotics and Internet of Things (IoT) systems. The open-source nature of the platform makes it accessible to beginners while still powerful enough for advanced users to easily develop relatively complex applications.

Arduino boards are designed to interface seamlessly with a variety of sensors, actuators, and communication modules. Using GPIO pins, users can control components like LEDs, motors, or relays and gather data from sensors such as temperature, light, or motion detectors. Popular Arduino boards, such as the Arduino Uno, Nano, and Mega, provide flexibility in terms of pin count and processing power, making them suitable for diverse projects. Additional shields and modules expand capabilities, allowing projects to include wireless communication (WiFi, Bluetooth) or GPS tracking.

The Arduino IDE drastically simplifies project development through a user-friendly interface and a rich library ecosystem. The programming language, based on C/C++, is straightforward, enabling even non-programmers to create functional systems **quickly** and **easily** with relatively flexible hardware configurations. **Arduino compatible libraries abstract the complexity** of working with specific hardware, providing ready-to-use functions for tasks like controlling motors, reading sensor data, or communicating over protocols like I2C, SPI, or UART. This modular approach accelerates development and minimizes coding errors for beginners and intermediate users.

Arduino projects are applied in numerous fields, including home automation, robotics, wearable technology, and environmental monitoring. The Arduino community is a major asset, offering extensive tutorials, **open-source code**, and forums where users can seek help or share ideas. This collaborative environment has contributed to the widespread adoption of Arduino as an educational tool and a platform for rapid prototyping, making it a major part of modern maker culture.

2.2.1 Comparison of Pico with Traditional Arduino Boards

The Raspberry Pi Pico offers several advantages compared to traditional Arduino boards:

- **Processor Speed:** Even classic Pico's ARM Cortex-M0+ runs at up to 133

MHz, faster than many Arduino models.

- **Memory:** 512 KB of SRAM is significantly more than most Arduino boards.
- **Cost:** The Pico 2 offers exceptional value, with a price point significantly lower than most Arduino boards. As of this book's publication, individual Pico 2 boards were available for just $5 USD!
- **Versatility:** Dual-core processing, RISC-V support, and extensive GPIO functions make it suitable for complex projects.
- **Programming Language:** Supports C/C++, and MicroPython, as well as Javascript and Lua while Arduino IDE primarily uses C/C++.

2.3 Getting Started with Arduino IDE for Raspberry Pi Pico

2.3.1 Installing Arduino IDE

Follow these steps to install Arduino IDE:

1. Download the Arduino IDE from the official website (`https://www.arduino.cc/en/software`).
2. Install the IDE by running the downloaded installer and following the on-screen instructions.
3. Launch the IDE and ensure it is updated to the latest version.

2.3.2 Setting Up Raspberry Pi Pico in Arduino IDE

Before connecting your boards and programming the Raspberry Pi Pico, install the necessary Pico build tools and **board package**:

1. Open Arduino IDE and navigate to `File > Preferences`. Be aware that some platforms may use `Arduino IDE > Settings`
2. Add the following URL in the "Additional Board Manager URLs" field: `https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json`

3. Go to `Tools > Board > Boards Manager`.
4. Search for "**Raspberry Pi Pico/RP2040/RP2350**" and **install the package by Earle Philhower**.

Now connect your Pico 2 and select:

1. Open the **Arduino IDE**.
2. Navigate to the top menu and click on **Tools**.
3. In the **Tools** menu locate and ensure the **Board** option is set to **"Raspberry Pi Pico 2"** which is under **Raspberry Pi Pico/RP2040/RP2350** sub menu.
4. Also under Tools menu, again scroll down to the **Port** option.
5. Under **Port**, select the appropriate serial port. You may see something like:

```
/dev/cu.usbmodem1101 (Raspberry Pi Pico 2)
```

6. Once the port is selected, you can proceed to upload or monitor your code.

2.3.3 Writing and Uploading Your First Program (Blink)

Below code is designed for the Raspberry Pi Pico 2 and leverages the Arduino IDE for deployment. The main objectives of the code are to control the onboard LED to blink at regular intervals and to communicate the LED's state through UART (Universal Asynchronous Receiver-Transmitter) serial communication. Before you proceed ensure your IDE is setup as mentioned above and your Pico is on boot-loader mode:

1. On your Raspberry Pi Pico board, press and hold the **BOOTSEL/RESET** button.
2. While holding down the **RESET** button, connect the Pico to your computer via a USB Micro cable.
3. Release the **RESET** button after a few seconds. This action puts the Pico into **bootloader mode**, allowing it to be detected as a UF2 device.
4. Open the **Arduino IDE** on your computer. Ensure that you only have a single instance (one window) of the IDE running to avoid conflicts during board and port detection.

5. Navigate to **Tools ¿ Port** in the Arduino IDE and select the port labeled as **UF2 Board**. This indicates that the IDE has successfully detected the Raspberry Pi Pico in bootloader mode.

6. Verify that the Arduino IDE reports a successful connection in the bottom-right corner of the interface. It should display something like:

```
Raspberry Pi Pico 2 on UF2 Board
```

In the Arduino IDE, create a new Arduino Sketch by navigating to **File ¿ New Sketch**. Then, copy and paste the code provided below, which is specifically designed for the Raspberry Pi Pico 2.

The code performs LED blinking functionality which is implemented using GPIO pin 25, which is the default onboard LED pin (`LED_PIN`). In the `setup()` function, the pin is configured as an output using `pinMode`. The `loop()` function toggles the LED state between ON and OFF every 500 milliseconds using the `digitalWrite` function. The LED's current state is stored in the variable `led_state`, which alternates between `HIGH` and `LOW` with each iteration of the loop.

UART communication is initialized in the `setup()` function with a baud rate of 9600 using `Serial.begin(9600)`. To ensure easy setup for the communication, flow control is disabled using `Serial.ignoreFlowControl(true)`. The program waits for the serial connection to be ready by checking `Serial.availableForWrite()` before proceeding. Each time the LED changes its state, a corresponding message is sent over the UART interface using `Serial.printf`. For example, when the LED turns ON, the message `"LED is on now ..."` is sent, and when it turns OFF, `"LED is off now ..."` is transmitted. The `Serial.flush()` function ensures that all data in the UART buffer is sent before moving on to the next operation.

The `setup()` function is executed once when the program starts. It initializes the LED pin for output and sets up UART communication. The `loop()` function, on the other hand, runs continuously and handles the core functionality of toggling the LED and sending the corresponding messages over UART. Together, these functions manage the main tasks of our program.

Two additional functions, `setup1()` and `loop1()`, are included in the code but are unused. They act as placeholders and can be safely removed if not needed. These functions are not executed during normal operation and do not affect the program's behavior.

```
#include <Arduino.h>

#define LED_PIN 25
#define UART0_TX_PIN 0
```



```
#define UART0_RX_PIN 1

short led_state = HIGH;

void setup() {
  pinMode(LED_PIN, OUTPUT);
  Serial.ignoreFlowControl(true);
  Serial.begin(9600);
  while (!Serial.availableForWrite())delay(100);
  digitalWrite(LED_PIN, led_state);  //turn the LED on GP25 on ... signal that we are ready
}

void loop() {
  led_state == HIGH ? led_state = LOW : led_state = HIGH;
  digitalWrite(LED_PIN, led_state);
  led_state == HIGH ? Serial.printf("LED is on now ...\r\n") : Serial.printf("LED is off now ...\r\n");
  Serial.flush(); // flush the usb to serial serial buffer content
  delay(500);  // waits 500 milliseconds
}

void setup1() { }
void loop1() { }
```

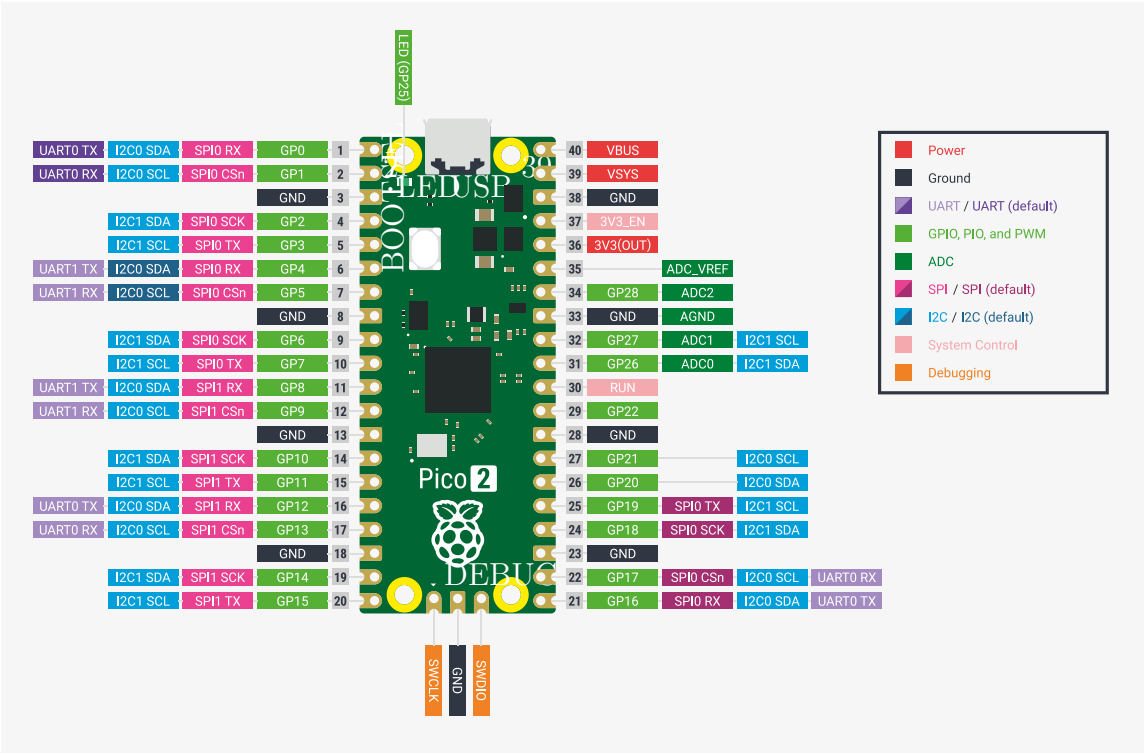
2.3.4 Raspberry Pi Pico 2 Pinouts

The table below lists all the pins for the Raspberry Pi Pico 2. The bold-colored text represents the default functions of the pins. Please always check the official Raspberry Pi documentation in case there are newer revision or changes: `https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html`

Pin Number	GP PIN	Other Functions
1	GP0	**UART0 TX**, I2C0 SDA, SPI0 RX
2	GP1	**UART0 RX**, I2C0 SCL, SPI0 CSn
3	GND	Ground
4	GP2	I2C1 SDA, SPI0 SCK
5	GP3	I2C1 SCL, SPI0 TX
6	GP4	UART1 TX, **I2C0 SDA**, SPI0 RX
7	GP5	UART1 RX, **I2C0 SCL**, SPI0 CSn
8	GND	Ground
9	GP6	I2C1 SDA, SPI0 SCK
10	GP7	I2C1 SCL, SPI0 TX
11	GP8	UART1 TX, I2C0 SDA, SPI1 RX
12	GP9	UART1 RX, I2C0 SCL, SPI1 CSn
13	GND	Ground
14	GP10	I2C1 SDA, SPI1 SCK

15	GP11	I2C1 SCL, SPI1 TX
16	GP12	UART0 TX, I2C0 SDA, SPI1 RX
17	GP13	UART0 RX, I2C0 SCL, SPI1 CSn
18	GND	Ground
19	GP14	I2C1 SDA, SPI1 SCK
20	GP15	I2C1 SCL, SPI1 TX
21	GP16	**SPI0 RX**, I2C0 SDA, UART0 TX
22	GP17	**SPI0 CSn**, I2C0 SCL, UART0 RX
23	GND	Ground
24	GP18	**SPI0 SCK**, I2C1 SDA
25	GP19	**SPI0 TX,** I2C1 SCL
26	GP20	I2C0 SDA
27	GP21	I2C0 SCL
28	GND	Ground
29	GP22	—
30	RUN	System Control
31	GP26	ADC0, I2C1 SDA
32	GP27	ADC1, I2C1 SCL
33	GND	AGND (Analog Ground)
34	GP28	ADC2
35	ADC_VREF	Analog Voltage Reference
36	3V3(OUT)	Power
37	3V3_EN	System Control
38	GND	Ground
39	VSYS	Power
40	VBUS	Power

Here is the layout mirrored from the Raspberry Pi Pico 2 website:



Chapter 3

Pico Basic GPIOs

3.1 Basic GPIO Operations

GPIO (General Purpose Input/Output) is one of the most critical aspects of working with microcontrollers. The Raspberry Pi Pico's GPIO pins enable communication with a variety of external devices such as LEDs, buttons, and sensors.

Digital I/O operations involve setting a GPIO pin to either HIGH (1) or LOW (0) for output, or reading its state for input. We start by controlling LEDs, which is one of the simplest ways to understand GPIO output. An LED is connected to a GPIO pin, and its state is toggled between HIGH and LOW to toggle turn on and off states.

```
// Example: Blinking an LED on GPIO25
#include "pico/stdlib.h"

int main() {
  const uint LED_PIN = 25; // Onboard LED
  gpio_init(LED_PIN);      // Initialize GPIO
  gpio_set_dir(LED_PIN, GPIO_OUT); // Set as output

  while (true) {
    gpio_put(LED_PIN, 1); // Turn on LED
    sleep_ms(500);        // Wait for 500 ms
    gpio_put(LED_PIN, 0); // Turn off LED
    sleep_ms(500);        // Wait for 500 ms
  }
  return 0;
}
```

Code explanations:

- `gpio_init()` initializes the specified GPIO pin.
- `gpio_set_dir()` sets the pin direction as either input or output.

- `gpio_put()` sets the GPIO pin state (HIGH or LOW).

3.1.1 Reading Button State

Buttons are used to input digital signals to the microcontroller. They are typically connected to GPIO pins configured as inputs.

A push button or tactile switch consists of two terminals (pins). To integrate it into our Pico MCU:

- The first pin is connected to the ground (GND)
- The second pin is connected to a GPIO pin on the Raspberry Pi Pico microcontroller
- The GPIO pin is configured with an internal pull-up resistor, maintaining a stable HIGH state when the button is not pressed
- When the button is pressed, it creates a connection to ground, pulling the GPIO pin to LOW state
- This configuration allows us to detect button presses by monitoring the state changes on the GPIO pin.

Code example for reading a button state:

```
// Example: Reading a button state
#include "pico/stdlib.h"

int main() {
  const uint BUTTON_PIN = 14; // Button connected to GPIO14
  gpio_init(BUTTON_PIN);
  gpio_set_dir(BUTTON_PIN, GPIO_IN); // Set as input
  gpio_pull_up(BUTTON_PIN); // Enable pull-up resistor

  while (true) {
    if (gpio_get(BUTTON_PIN) == 0) { // Check if button is pressed
      printf("Button pressed!\n");
    }
    sleep_ms(100);
  }
  return 0;
}
```

Code explanation:

- `gpio_pull_up()` enables an internal pull-up resistor to ensure a stable HIGH signal when the button is not pressed.
- `gpio_get()` reads the state of the GPIO pin.

3.1.2 Working with Multiple GPIOs

Multiple GPIO pins can be used simultaneously for tasks like controlling a sequence of LEDs or toggling multiple button states.

3.1.2.1 Sequential LED Control

This example demonstrates how to turn on LEDs one by one in a sequence.

```
// Example: Sequential LED control
#include "pico/stdlib.h"

int main() {
  const uint LED_PINS[] = {2, 3, 4}; // LEDs connected to GPIO2, GPIO3, GPIO4
  const int NUM_LEDS = 3;

  for (int i = 0; i < NUM_LEDS; i++) {
    gpio_init(LED_PINS[i]);
    gpio_set_dir(LED_PINS[i], GPIO_OUT);
  }

  while (true) {
    for (int i = 0; i < NUM_LEDS; i++) {
      gpio_put(LED_PINS[i], 1); // Turn on LED
      sleep_ms(300);
      gpio_put(LED_PINS[i], 0); // Turn off LED
    }
  }
  return 0;
}
```

Code explanation:

- The GPIO pins for LEDs are initialized in a loop for efficiency.
- Each LED is turned on and off sequentially to create a chaser effect.

3.1.2.2 Example: Button State Toggling

Button state toggling involves using a button to switch the state of an LED each time it is pressed.

```
// Example: Toggle LED state with button press
#include "pico/stdlib.h"

int main() {
  const uint LED_PIN = 25;
  const uint BUTTON_PIN = 14;

  gpio_init(LED_PIN);
  gpio_set_dir(LED_PIN, GPIO_OUT);
```



```
  gpio_put(LED_PIN, 0); // Initialize LED as off

  gpio_init(BUTTON_PIN);
  gpio_set_dir(BUTTON_PIN, GPIO_IN);
  gpio_pull_up(BUTTON_PIN);

  bool led_state = false;

  while (true) {
    if (gpio_get(BUTTON_PIN) == 0) { // Button pressed
      led_state = !led_state; // Toggle LED state
      gpio_put(LED_PIN, led_state);
      sleep_ms(300); // Debounce delay
    }
  }
  return 0;
}
```

Code explanation:

- The LED state is toggled using a boolean variable.
- A debounce delay prevents multiple toggles due to button bounce.

Chapter 4

Pico Advanced GPIOs, Timers, DMA Channels

4.0.1 Pull Up and Pull Dow Resistors

When using GPIO (General Purpose Input/Output) pins as inputs in microcontroller circuits, external signals can often become unstable or float between undefined voltage levels due to the absence of a defined input signal. This floating state makes the pin susceptible to electromagnetic interference or noise from the surroundings, leading to unreliable or erratic behavior. To prevent this, pull-up and pull-down resistors are employed to provide a stable and predictable voltage level on the GPIO pin when no external signal is present. Pull-up resistors connect the pin to a high voltage level (e.g., V_{CC}), ensuring a logical HIGH state, while pull-down resistors connect the pin to ground, ensuring a logical LOW state. These resistors are critical for eliminating the risks of floating pins and noise interference, ensuring consistent and reliable input readings. Whether implemented as external components or using the microcontroller's internal configurable resistors, they are a fundamental practice for robust and noise-tolerant digital circuit design.

- A **pull-up resistor** connects the GPIO pin to a high voltage level (e.g., V_{CC} or 3.3V/5V). When no external input is driving the pin, the resistor ensures the pin is "pulled up" to the high logic level, typically interpreted as a **logical 1**. When an external signal drives the pin low (ground or 0V), the pull-up resistor allows the pin to reach the low logic level while preventing floating states.
- A **pull-down resistor** connects the GPIO pin to ground (0V). When no external input is driving the pin, the resistor ensures the pin is "pulled down"

to the low logic level, typically interpreted as a **logical 0**. When an external signal drives the pin high (e.g., V_{CC}), the pull-down resistor allows the pin to reach the high logic level while preventing floating states.

- The value of the pull-up or pull-down resistor is crucial. It must be high enough to minimize current consumption and avoid interfering with the external input signals, yet low enough to ensure the pin voltage stabilizes quickly. Typical resistor values range from **1 kΩ to 100 kΩ**, depending on the specific requirements of the circuit.

- Many modern microcontrollers, including the Raspberry Pi Pico, have configurable **internal pull-up and pull-down resistors** that can be enabled via software. These eliminate the need for external resistors in most cases. For example, in Arduino code, you can enable an internal pull-up resistor on a GPIO pin as follows:

```
pinMode(your_pin, INPUT_PULLUP);
```

You can use pull-up and pull-down resistors to ensure reliable operation in various input scenarios. For push buttons, these resistors are essential for preventing the GPIO pin from floating when the button is not pressed, thereby providing consistent and stable readings. Similarly, sensors with open-drain or open-collector outputs often depend on pull-up resistors to function correctly, as they require a defined voltage level to produce accurate output signals. In the case of switches, pull resistors eliminate undefined behavior when the switch is open, ensuring that the GPIO pin maintains a predictable state.

4.0.2 Debouncing

As covered briefly, General Purpose Input/output pins are the primary means for interfacing the Raspberry Pi Pico with external devices. GPIO pins must be initialized before use. They can be set as input or output depending on the requirement.

In electronics, **debouncing** refers to the process of eliminating or filtering out unwanted noise or fluctuations that occur when a mechanical switch or button is pressed or released. This phenomenon, known as "bouncing," arises due to the mechanical nature of switches, which causes multiple rapid on-off transitions before settling into a stable state.

The bouncing effect occurs because of the inherent mechanical properties of the switch design. When a button is pressed or released, the internal metal contacts—often slightly curved and spring-like—make and break contact multiple times in rapid

succession before finally settling into a stable state. To visualize this, imagine releasing a droplet of water onto the surface of a bowl filled with water. The droplet initially causes ripples and bounces back and forth for a fraction of a second before merging and becoming still. Similarly, when the button is pressed, the tiny metal sheets within the switch collide and compress, generating a series of rapid "on-off" transitions. This process occurs within a fraction of a second but produces multiple electrical signals that a microcontroller might misinterpret as several button presses or releases. The rapid oscillations mimic chaotic and noisy behavior in the signal, despite the user's intention of performing a single, clean action.

This bouncing effect is characteristic of mechanical switches, stemming from the elasticity and inertia of the contact materials. The small vibrations may also be amplified by microscopic irregularities in the surfaces of the contacts, which prolong the settling time. Without handling this behavior—through hardware or software debouncing—a system might behave unpredictably, toggling an LED or registering unintended inputs multiple times for a single button press or release.

Understanding this behavior is crucial for designing reliable embedded systems. The noise caused by bouncing can disrupt the intended functionality of a program. Effective debouncing ensures that only the final, stable state of the button is registered, filtering out the transient oscillations and providing smooth, predictable responses to user inputs. Without debouncing, these unintended transitions could be interpreted as multiple presses or releases by a microcontroller, leading to erratic or undesired behavior in programs.

To recap, we can say when a button is pressed, its contacts may rapidly make and break connection several times within a few milliseconds before stabilizing. For example, pressing a button might generate a signal like `HIGH-LOW-HIGH-LOW-HIGH`, instead of the expected `HIGH-LOW`. If a microcontroller reacts to every transition, it may toggle an LED multiple times or register several unintended events. Debouncing ensures that only a single press or release event is detected, ignoring the intermediate noise caused by bouncing.

There are two primary methods for debouncing:

- **Hardware Debouncing:** Uses physical components like resistors, capacitors, or Schmitt triggers to smooth out the signal and eliminate noise before it reaches the microcontroller. This is effective but adds extra components to the circuit.

- **Software Debouncing:** Implements logic in the program to filter out transient state changes. For example, the program can check if the button state remains stable for a specified duration (e.g., 50ms) before considering it valid.

4.0.2.0.1 Debouncing Example

This example demonstrates how to wire a push button input and derive the internal on-board Pico LED. Connect one terminal of the push button to the GP22 pin and the other terminal to GND (ground). If you don't have a physical push button available, you can simulate the button press by using a jumper wire - simply touch one end to GP22 and briefly tap the other end to GND to simulate pressing and releasing the button.

```
#include <stdio.h>
#include "pico/stdlib.h"

#define LED_PIN 25
#define PUSH_BUTTON_PIN 22
#define OFF 0
#define ON 1
#define PUSH_BUTTON_PRESSED 0
#define PUSH_BUTTON_RELEASED 1

volatile short led_state = OFF;

int main() {
  gpio_init(PUSH_BUTTON_PIN);
  gpio_init(LED_PIN);

  gpio_set_dir(PUSH_BUTTON_PIN, GPIO_IN);
  gpio_set_dir(LED_PIN, GPIO_OUT);

  gpio_put(LED_PIN, OFF);
  gpio_pull_up(PUSH_BUTTON_PIN);

  sleep_ms(1000);

  int button_state = PUSH_BUTTON_RELEASED;
  int last_button_state = PUSH_BUTTON_RELEASED;
  uint32_t last_debounce_time = 0;
  const uint32_t debounce_delay = 50; // 50 ms debounce delay

  while (true) {
    // Read the current state of the button
    int current_state = gpio_get(PUSH_BUTTON_PIN);

    // Debounce: Check if the button state has changed
    if (current_state != last_button_state) {
      last_debounce_time = to_ms_since_boot(get_absolute_time());
    }

    // If the state is stable for more than the debounce delay, update the button state
    if ((to_ms_since_boot(get_absolute_time()) - last_debounce_time) > debounce_delay) {
      if (current_state != button_state) {
        button_state = current_state;

        if (button_state == PUSH_BUTTON_PRESSED) {
          led_state = ON;  // Turn LED on when button is pressed
          gpio_put(LED_PIN, ON);
          printf("Button pressed! LED is ON.\n");
        } else if (button_state == PUSH_BUTTON_RELEASED) {
```



```
          led_state = OFF; // Turn LED off when button is released
          gpio_put(LED_PIN, OFF);
          printf("Button released! LED is OFF.\n");
        }
      }
    }

    // Save the current state as the last state
    last_button_state = current_state;
    tight_loop_contents(); // Keep processor idle
  }
  return 0;
}
```

The Pico supports simultaneous configuration and management of multiple GPIOs.

```
// Example: Controlling multiple LEDs
#include "pico/stdlib.h"

int main() {
  const uint LED_PINS[] = {2, 3, 4};
  const int NUM_LEDS = 3;

  for (int i = 0; i < NUM_LEDS; i++) {
    gpio_init(LED_PINS[i]);
    gpio_set_dir(LED_PINS[i], GPIO_OUT);
  }

  while (true) {
    for (int i = 0; i < NUM_LEDS; i++) {
      gpio_put(LED_PINS[i], 1); // Turn on LED
      sleep_ms(200);
      gpio_put(LED_PINS[i], 0); // Turn off LED
    }
  }
  return 0;
}
```

4.0.3 Interrupt Requests (IRQs) and GPIOs

Synchronous (Sync) and Asynchronous (Async) are two different approaches to handling tasks and communication in programming and systems design.

Synchronous Systems:

- In a synchronous operation, tasks are performed one after another, in a sequential manner.
- When a task is initiated, the program waits for it to complete before moving on to the next task.
- The execution follows a specific order, and each task is dependent on the completion of the previous one.

- Synchronous operations block the execution until the task is finished, which can lead to delays and reduced performance if a task takes a long time to complete.
- Examples of synchronous operations include making a function call and waiting for it to return, or reading a file and waiting until all the data is read.

Asynchronous Systems:

- In an asynchronous operation, tasks are initiated and executed independently, without waiting for each other to complete.
- When a task is started, the program continues executing other tasks without waiting for the asynchronous task to finish.
- Asynchronous operations are non-blocking, meaning they allow the program to continue running while the task is being performed in the background.
- The program is notified when an asynchronous task is completed through callbacks, promises, or events.
- Asynchronous programming is often used for I/O operations, such as making HTTP requests, reading from or writing to files, or interacting with databases, to improve performance and responsiveness.
- Examples of asynchronous operations include sending an HTTP API request and handling the response with a callback, or reading a large file in chunks without blocking the main thread of execution.

Asynchronous programming is particularly useful in scenarios where tasks may take a long time to complete, such as network communication, file I/O, or complex calculations. By using asynchronous operations, the program can continue executing other tasks while waiting for the long-running tasks to finish, leading to better performance and responsiveness.

However, asynchronous programming can also introduce complexity, as the flow of execution becomes less linear and more event-driven. Proper handling of callbacks, error handling, and synchronization is crucial to avoid issues like race conditions and to ensure the correct order of execution.

At the heart of the Async programming, we have Interrupt Requests (IRQs) which they provide a hardware-driven mechanism for processors to respond to external events asynchronously and with lower latency. This approach allows the processor to focus on other tasks until an event occurs, reducing unnecessary CPU cycles.

We can check arrival of events and states in:

1. **Polling Method (Traditional):**
    - The CPU continuously checks pin states.
    - Consumes CPU cycles unnecessarily (even when the event did not happen yet!).
    - Similar to asking "Are we there yet?" repeatedly.
2. **IRQ Method (Event-Driven):**
    - Hardware automatically detects changes in the state of a pin.
    - The CPU is freed for other tasks until an event occurs.
    - Immediate response to events via callback function IRQ Handlers and Interrupt Service Routines (ISRs).

4.0.3.1 IRQ Trigger Conditions

1. **Rising Edge Detection:**
    - Transition from LOW (0V) to HIGH (3.3V/5V).
2. **Falling Edge Detection:**
    - Transition from HIGH to LOW.
3. **Level Detection:**
    - Specific voltage level maintained (HIGH or LOW).
    - Toggle detection (any change in state).

4.0.3.2 Selecting between IRQ and Polling

The IRQ/ISR approach provides a significant improvement over the traditional polling method, as it:

- Reduces CPU utilization by minimizing unnecessary checks
- Improves responsiveness and latency
- Allows for more efficient use of resources

While the IRQ (Interrupt Request) approach offers several benefits over the traditional polling method, it also has some potential side effects and cons to consider: Side Effects:

- Increased complexity: Implementing an IRQ-based system requires additional hardware and software components, such as interrupt controllers and interrupt service routines. This added complexity can make the system more difficult to design, debug, and maintain.
- Priority management: When multiple devices generate interrupts simultaneously, the system must handle interrupt priorities to ensure that critical events are processed first. Managing interrupt priorities can be challenging and requires careful design and configuration.
- Interrupt latency: Although IRQs provide faster response times compared to polling, there is still a certain amount of latency involved in processing an interrupt. This latency can vary depending on factors such as the current system load, interrupt priorities, and hardware limitations.
- Shared resources: Interrupt handlers often need to access shared resources, such as memory or peripherals. If not properly managed, concurrent access to these shared resources can lead to race conditions, data corruption, or system instability.
- Overhead: Handling interrupts introduces overhead in terms of context switching, saving and restoring processor state, and executing the interrupt service routine. This overhead can impact system performance, especially if interrupts occur frequently or if the ISRs are lengthy.
- Difficult to predict timing: The exact timing of interrupt occurrence can be difficult to predict, as it depends on external events. This uncertainty can make it challenging to ensure deterministic behavior in real-time systems or to meet strict timing requirements.
- Potential for interrupt storms: In some cases, a malfunctioning device or a poorly designed system can generate a large number of interrupts in a short period, known as an interrupt storm. Interrupt storms can overwhelm the system, causing performance degradation or even system crashes.
- Debugging challenges: Debugging interrupt-driven systems can be more complex than debugging polling-based systems. Interrupt handlers execute asynchronously,

making it harder to trace program flow and identify issues. Special debugging techniques and tools may be required.

- Power consumption: Constantly monitoring for interrupts and waking up the processor to handle them can impact power consumption, especially in low-power or battery-operated devices. Efficient interrupt management techniques and power-saving modes need to be employed to mitigate this issue.

To mitigate these side effects, you need to carefully design and implement interrupt-based systems. This includes proper interrupt priority assignment, efficient ISRs, synchronization mechanisms for shared resources, and thorough testing and debugging.

Here is a sample code that uses IRQ to changes Pico's on-board LED blinking rate upon pressing the push button connected to GP22 and GND.

```
#include <Arduino.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"

#define LED_PIN 25
#define PUSH_BUTTON_PIN 22
#define DEBOUNCE_DELAY 70 // 70ms debounce delay

const short delays[] = {100, 200, 500, 1000, 2000, 4000}; //possible delays between LED blinkings
const int num_delays = sizeof(delays)/sizeof(delays[0]);
volatile int current_index = 3; // Start at 1000ms blinking

volatile short led_state = HIGH;
volatile short led_delay = delays[current_index];
volatile uint32_t last_debounce_time = 0;

void button_irq_handler() {
  uint32_t current_time = millis();

  if ((current_time - last_debounce_time) > DEBOUNCE_DELAY) {
    last_debounce_time = current_time;
    led_delay = delays[current_index = (current_index + 1) % num_delays];
  }
}

void setup() {
  pinMode(PUSH_BUTTON_PIN, INPUT_PULLUP);
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(9600);

  attachInterrupt(digitalPinToInterrupt(PUSH_BUTTON_PIN), button_irq_handler, CHANGE); //FALLING or RISING trigger
  digitalWrite(LED_PIN, led_state);

  Serial.printf("Initial delay: %d ms\r\n", led_delay);
}

void loop() {
  led_state = (led_state == HIGH) ? LOW : HIGH;
  digitalWrite(LED_PIN, led_state);
```



```
  if (led_state == HIGH) Serial.printf("LED is on now (blinking rate: %d)...\r\n", led_delay);
  else Serial.printf("LED is off now ...\r\n");

  Serial.flush();
  delay(led_delay);
  tight_loop_contents(); // Keep processor idle
}
```

4.0.4 Timers - Advanced IRQ with no Busy/Delay Loops

In this section we discuss an advanced implementation of our previous IRQ-enabled LED blinking control system for the Raspberry Pi Pico microcontroller. While our previous version demonstrated basic interrupt handling, this new implementation showcases a more advanced timer-based control and efficient resource management. Our previous implementation faced two critical limitations: First, the busy-delay loops couldn't be interrupted mid-cycle, making dynamic changes to the LED blinking rate impossible during operation. Second, the reliance on busy-waiting loops (implemented through delay() functions, which internally used while(1) loops) resulted in significant CPU overhead and inefficient resource utilization. These limitations restricted both the functionality and efficiency of the system. The new implementation addresses these challenges through a redesign that leverages hardware timer interrupts and event-driven architecture. We improve our timing control through hardware timer interrupts, efficient CPU usage through event-driven design, immediate response to user input for real-time blinking rate adjustments, and robust interrupt handling for more reliable operation. The system utilizes the same GPIO pins for LED output (GP25) and push button input (GP22), implementing a debouncing mechanism with a 250ms window to ensure reliable and smooth button press detection. The timing system now features a progressive sequence of eight distinct delay values, ranging from 50ms to 8000ms, with each button press smoothly transitioning to the next delay value. The implementation maintains a state management through volatile variables for interrupt safety and implements a more predictable state transitions. The system exhibits resource efficiency, responsiveness, and reliability, while remaining maintainable and expandable for future enhancements such as multiple LED support, complex blinking patterns, and more advanced configuration options.

Our previous implementation faced two major challenges:

- The busy-delay loops could not be interrupted mid-cycle, preventing dynamic changes to the LED blinking rate.
- The use of busy-waiting loops (e.g., delay() which behind the scene used busy loops such as while(1)) resulted in inefficient CPU utilization and overhead.

The new approach outlined below resolves these issues through:

- Hardware timer interrupts for precise timing control
- Event-driven architecture for efficient CPU usage
- Immediate response to user input and change the blinking rate.
- Clean interrupt handling for reliable operation

```
#include <Arduino.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"
#include "hardware/timer.h"

#define LED_PIN 25
#define PUSH_BUTTON_PIN 22
#define DEBOUNCE_DELAY 250 // 250ms debounce delay

const short delays[] = {50, 100, 200, 500, 1000, 2000, 4000, 8000};
const int num_delays = sizeof(delays)/sizeof(delays[0]);
volatile int current_index = 0; // Start at 50ms

volatile short led_state = HIGH;
volatile short led_delay = delays[current_index];
volatile uint32_t last_debounce_time = 0;
volatile bool timer_fired = false;
volatile static uint32_t current_time = millis();
struct repeating_timer timer;

// Timer interrupt handler
bool repeating_timer_callback(struct repeating_timer *t) {
  timer_fired = true;
  return true; // Keep repeating
}

//this would be called by the hardware whenever there is falling edge on GP22 pin which is conncted to a push button
void button_irq_handler() {
  current_time = millis();
  if ((current_time - last_debounce_time) > DEBOUNCE_DELAY) {
    last_debounce_time = current_time;

    current_index = (current_index + 1) % num_delays;
    led_delay = delays[current_index];

    // Force immediate LED state change
    led_state = HIGH;  // Start new cycle with LED ON
    digitalWrite(LED_PIN, led_state);

    // Update timer interval
    cancel_repeating_timer(&timer);
    timer_fired = false;  // Reset the timer flag
    add_repeating_timer_ms(led_delay, repeating_timer_callback, NULL, &timer);
  }
}
```



```
void setup() {
  pinMode(PUSH_BUTTON_PIN, INPUT_PULLUP);
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(9600);
  attachInterrupt(digitalPinToInterrupt(PUSH_BUTTON_PIN), button_irq_handler, FALLING);
  digitalWrite(LED_PIN, led_state);
  add_repeating_timer_ms(led_delay, repeating_timer_callback, NULL, &timer); // Initialize delay timer
  Serial.printf("Initial LED blinking delay is set to: %d ms\r\n", led_delay);
}

void loop() {
  if (timer_fired) {
    led_state = (led_state == HIGH) ? LOW : HIGH;
    digitalWrite(LED_PIN, led_state);

    if (led_state == HIGH) Serial.printf("LED is on now (blinking every %d ms)...\r\n", led_delay);
    else Serial.printf("LED is off now (blinking every %d ms)...\r\n", led_delay);

    Serial.flush();
    timer_fired = false;
  }
  tight_loop_contents(); // Keep processor idle
}
```

Code explanation:

- Onboard Pico LED, which is connected to GP25
- A push button as our input: GP22 (with internal pull-up)
- Debounce delay of 250ms
- Baud rate of 9600 symbols per second for our serial communication to the computer
- The system implements 8 distinct delay periods:

$$\text{Delays} = \{50, 100, 200, 500, 1000, 2000, 4000, 8000\} \text{ milliseconds}$$

- The hardware timer operates according to:

$$\text{Timer Operation} = \begin{cases} \text{Set flag when expired} \\ \text{Trigger LED state change} \\ \text{Auto-repeat cycle} \end{cases}$$

- Upon button press detection we do:

$$\text{Button Press} \rightarrow \begin{cases} \text{Check: } t_{current} - t_{last} > t_{debounce} \\ \text{Update Index: } i_{next} = (i_{current} + 1) \bmod n \\ \text{New Delay: } d_{new} = \text{delays}[i_{next}] \\ \text{Reset Timer} \end{cases}$$

- LED state transitions follow:

$$\text{LED State} = \begin{cases} \text{HIGH} \rightarrow \text{LOW} & \text{if timer fired} \\ \text{LOW} \rightarrow \text{HIGH} & \text{if timer fired} \end{cases}$$

- we start by system initialization:

$$\text{Initialize} \rightarrow \begin{cases} \text{Configure GPIO pins} \\ \text{Setup hardware timer} \\ \text{Enable interrupts} \\ \text{Begin serial communication} \end{cases}$$

- we continue to repeatedly perform:

$$\text{Main Loop} \rightarrow \begin{cases} \text{Check timer flag} \\ \text{Toggle LED if needed} \\ \text{Update serial output} \\ \text{Reset flag} \\ \text{Enter idle state} \end{cases}$$

- For any selected delay period d_i, the complete LED cycle time is:

$$t_{cycle} = 2d_i$$

  where:

  - t_{cycle} is the total ON-OFF cycle time
  - d_i is the current selected delay from the sequence

- The LED state transitions are governed by:

$$\text{State}_{next} = \begin{cases} !\text{State}_{current} & \text{if timer expired} \\ \text{State}_{current} & \text{otherwise} \end{cases}$$

4.0.5 DMA

Direct Memory Access (DMA) is a feature in computer systems that enables certain hardware subsystems to access system memory independently of the CPU. DMA provides an efficient mechanism for data transfer in embedded systems, offering significant advantages in terms of CPU efficiency and system performance. Consider a restaurant scenario:

- **Without DMA:** The chef (CPU) must stop cooking to deliver each plate
- **With DMA:** Waiters (DMA) handle food delivery while the chef continues cooking

Traditional Method (Without DMA) used to:

1. Use CPU to read data from source
2. Use CPU to store data temporarily
3. Use CPU to write data to destination
4. Repeat this process for each data unit

The DMA approach streamlines the process:

1. CPU configures DMA transfer parameters
2. DMA controller manages the entire transfer
3. CPU performs other tasks
4. DMA signals completion to CPU

4.0.5.1 The key benefits of using DMA

The implementation of DMA for GPIO control provides an efficient solution for handling repetitive GPIO operations without CPU intervention. This approach is particularly valuable in applications requiring CPU-efficient peripheral control with high-speed GPIO manipulation that demands precise timings:

- CPU Efficiency: CPU freed from data transfer tasks, allowing parallel operation capability, while reducing system overhead

- Direct hardware-to-hardware transfer may increas the performance, with minimal CPU intervention and enhanced system throughput
- Predictable transfer timing and consistent data handling, which make DMA to be suitable candidate for some time-critical operations

4.0.5.2 DMA Examples

For GPIOs, DMA enables efficient transfer of data between memory and peripherals without CPU intervention, making it ideal for repetitive tasks like massive array of LED pattern generation. The following example shows a basic implementation of DMA-controlled GPIO:

```
#include "pico/stdlib.h"
#include "hardware/dma.h"
#include "hardware/gpio.h"

#define LED_PIN 25

// Create pattern data for LED blinking
const uint32_t pattern[] = {
  1, 0, 1, 0, 1, 0  // LED ON/OFF sequence
};

int main() {
  // Initialize LED pin
  gpio_init(LED_PIN);
  gpio_set_dir(LED_PIN, GPIO_OUT);

  // Get a free DMA channel
  int dma_channel = dma_claim_unused_channel(true);

  // Configure DMA channel
  dma_channel_config config =
  dma_channel_get_default_config(dma_channel);

  channel_config_set_transfer_data_size(&config, DMA_SIZE_32);
  channel_config_set_ring(&config, true, 3);
  channel_config_set_read_increment(&config, true);
  channel_config_set_write_increment(&config, false);

  dma_channel_configure(
  dma_channel,
  &config,
  &gpio_hw->out,
  pattern,
  count_of(pattern),
  true
  );

  while(1) {
    tight_loop_contents();
  }
```



```
  return 0;
}
```

Here is a more advanced implementation with timer controls:

```
  #include "pico/stdlib.h"
  #include "hardware/dma.h"
  #include "hardware/gpio.h"
  #include "hardware/timer.h"

  #define LED_PIN 25
  #define PATTERN_LENGTH 8

  // LED pattern with timing
  const struct {
    uint32_t value;
    uint32_t delay_ms;
  } led_pattern[PATTERN_LENGTH] = {
    {1, 100},  // ON for 100ms
    {0, 200},  // OFF for 200ms
    // ... pattern continues ...
  };
\end{lstlisting}

\section{Key Features}
\subsection{DMA Configuration}
\begin{itemize}
  \item Channel allocation and setup
  \item Transfer size configuration
  \item Circular buffer implementation
  \item Read/write increment settings
\end{itemize}

\subsection{Pattern Control}
\begin{itemize}
  \item Flexible pattern definition
  \item Timer-based precision
  \item Variable timing support
  \item Pattern repetition handling
\end{itemize}

\section{Enhancement Functions}
\subsection{Pattern Speed Control}
\begin{lstlisting}
  void change_pattern_speed(float speed_multiplier) {
    for(int i = 0; i < PATTERN_LENGTH; i++) {
      pattern_delays[i] *= speed_multiplier;
    }
  }
```

Here is how we can do DMA Transfer and Validations:

```
  void switch_pattern(const uint32_t* new_pattern, size_t length) {
    dma_channel_abort(dma_channel);
```



```
    dma_channel_configure(
    dma_channel,
    &config,
    &gpio_hw->out,
    new_pattern,
    length,
    true
    );
  }

  bool validate_dma_transfer() {
    if(dma_channel_is_busy(dma_channel)) {
      return false;
    }
    return true;
  }
\end{lstlisting}
```

4.0.5.3 DMA Best Practices

- Always check DMA channel availability before use
- Implement proper error handling
- Clean up DMA channels when no longer needed
- Consider timing requirements in pattern design

Chapter 5

Pico's Advanced Peripherals

5.1 Universal Asynchronous Receiver-Transmitter - SCI Bus

UART (Universal Asynchronous Receiver-Transmitter), also referred to as **SCI (Serial Communication Interface)**, is a widely used communication protocol for transmitting and receiving data serially between two devices. Unlike synchronous protocols such as I2C and SPI, UART does not require a clock signal. Instead, it uses a pair of dedicated lines—`TX` (transmit) and `RX` (receive)—to exchange data asynchronously. The Raspberry Pi Pico supports multiple UART instances (`uart0` and `uart1`) with highly flexible GPIO pin assignments, making it suitable for a variety of serial communication needs.

The key parameters governing UART communication are **baud rate**, **data bits**, **stop bits**, and **parity**. The **baud rate** specifies the speed of data transfer, measured in bits per second (bps). Both devices communicating via UART must use the same baud rate to interpret data correctly. For example, common baud rates include 9600, 19200, and 115200 bps. The **data bits** parameter determines the size of each data frame, typically set to 8 bits for most applications. **Stop bits** signal the end of a data frame, and **parity** is an optional error-checking mechanism that ensures data integrity during transmission.

UART communication on the Pico follows a full-duplex model, allowing simultaneous transmission and reception of data. Each UART instance on the Pico has dedicated hardware FIFOs (First In, First Out buffers) for managing incoming and outgoing data. This buffering mechanism reduces the risk of data loss in high-speed communication scenarios. The default pin mapping for `uart0` uses GPIO0 for `TX` and GPIO1 for `RX`,

while `uart1` maps to GPIO4 and GPIO5, but these can be reassigned to other GPIOs as needed.

A key advantage of UART is its simplicity. Unlike protocols like I2C or SPI, UART does not require addressing or clock synchronization, making it easy to implement for point-to-point communication. However, it is limited to one-to-one communication unless additional hardware such as multiplexers is used. UART is commonly employed for debugging, where devices send logs to a terminal or monitor via a serial interface.

In the Raspberry Pi Pico's C/C++ SDK, UART communication is initialized using `uart_init()`, and data can be transmitted or received using `uart_putc()`, `uart_getc()`, and related functions. For example, initializing UART at a baud rate of 9600 on `uart0` can be achieved with:

```
#include "pico/stdlib.h"

uart_init(uart0, 9600);  // Initialize UART0 with a baud rate of 9600
gpio_set_function(0, GPIO_FUNC_UART);  // TX on GPIO0
gpio_set_function(1, GPIO_FUNC_UART);  // RX on GPIO1
```

MicroPython is a lightweight implementation of the Python 3 programming language specifically designed for microcontrollers and constrained systems. It provides an intuitive and high-level approach to programming devices like the Raspberry Pi Pico, Arduino boards, ESP32, and other microcontrollers. MicroPython enables developers to write concise and readable code, making it an alternative to the Arduino IDE and the Pico C/C++ SDK, especially for beginners or rapid prototyping.

Unlike the Arduino IDE, which uses a simplified version of C++, or the Pico C/C++ SDK, which requires a more detailed setup and understanding of low-level programming, MicroPython focuses on simplicity and accessibility. It offers built-in libraries for interacting with hardware peripherals, such as GPIO, I^2C, SPI, UART, and PWM. Writing MicroPython code requires minimal boilerplate, allowing developers to focus on functionality rather than hardware intricacies. For example, toggling an LED in MicroPython can be achieved with just with a fewer lines of code:

```
from machine import Pin
import time

led = Pin(25, Pin.OUT)
while True:
led.toggle()
time.sleep(1)
```

One of the standout features of MicroPython is its interactive **REPL (Read-Eval-Print Loop)**. The REPL allows users to execute Python commands directly on the microcontroller in real time, making debugging and experimentation much faster compared to the compile-upload-test cycle of Arduino and C/C++ Pico SDK

development. This interactivity reduces development time and makes MicroPython particularly appealing for prototyping and educational use.

While MicroPython is easier to learn and use, it does have some trade-offs. It is an interpreted language, meaning that it typically runs slower and uses more memory than compiled code from the Pico C/C++ SDK or Arduino IDE. This makes MicroPython less suitable for performance-critical applications, such as real-time motor control or high-frequency signal processing. However, for most general-purpose applications like environmental sensing, LED control, or basic IoT systems, MicroPython's ease of use and flexibility outweigh these limitations.

In MicroPython, the `machine.UART` class drastically simplifies UART operations. For example, initializing UART at 9600 baud and sending a message can be done as follows:

```
from machine import UART

uart = UART(0, baudrate=9600, tx=0, rx=1)
uart.write("Hello, UART!\n")
```

UART communication is widely used in embedded systems for connecting microcontrollers to peripherals such as GPS modules, GSM modules, and Bluetooth adapters. For example, a GPS module can send location data to the Pico over UART, which the Pico processes and displays on an OLED screen. Similarly, UART is indispensable for debugging, enabling the Pico to send logs and messages to a terminal such as `PuTTY`.

While UART is simple and versatile, it has limitations. It lacks built-in support for multi-device communication and advanced error-checking mechanisms (other than optional parity). Additionally, because it operates asynchronously, both devices must use closely matching baud rates and configurations to avoid data corruption.

The Raspberry Pi Pico enhances UART functionality by supporting features such as interrupt-driven communication and DMA. Interrupts allow efficient handling of data events without constant polling, while DMA enables high-speed data transfer with minimal CPU intervention, making UART suitable for applications involving large or continuous data streams.

5.1.1 Serialization and Deserialization in Communication

Serialization is the process of converting structured data, such as objects, arrays, or hierarchical information, into a format that can be transmitted or stored. This typically involves flattening the data into a sequential byte stream, making it suitable for storage in files or transmission over communication protocols like UART, I^2C, or TCP/IP. **Deserialization** is the reverse process, where the byte stream is parsed to

reconstruct the original structured data. These processes are fundamental in enabling data exchange between systems or devices that might use different architectures, languages, or formats.

One common aspect of serialization involves choosing between **text/character-based** and **binary** communication formats. Text-based communication uses human-readable formats like JSON, XML, or CSV, where data is represented as strings. For example, a JSON message like `"{"temperature": 25.5, "humidity": 60}"` provides clarity and ease of debugging. In contrast, binary communication involves encoding the data into a compact and machine-readable format, such as IEEE 754 for floating-point numbers or custom binary protocols. While text formats are easier to understand and debug, they are often less efficient in terms of size and processing speed compared to binary formats.

In **text-based communication**, the data is serialized as strings of characters. This format is platform-independent, making it ideal for cross-platform communication. However, text-based communication can introduce overhead because of verbose representations. For instance, the number `25.5` in JSON may require additional bytes for delimiters, keys, and formatting, which are unnecessary in binary formats. Additionally, parsing text-based formats can be computationally expensive, especially in embedded systems with limited resources.

Binary communication, on the other hand, is optimized for efficiency, directly representing data as byte sequences without additional formatting. For example, a 32-bit floating-point number in binary requires exactly 4 bytes, regardless of its value, while a text-based representation might require more bytes for decimal points or additional characters. However, binary formats are less human-readable and can be difficult to debug without specialized tools. The choice between text-based and binary communication depends on the application's requirements: text is preferred for readability and compatibility, while binary is ideal for speed and bandwidth efficiency.

5.1.2 Example - NAMO Serial Protocol: Bi-Directional Text and Binary Communication

The **NAMO Serial Protocol** is a framework designed specifically for educational purposes, aiming to help students and learners understand the core principles of serial communication. By bridging the gap between theory and practice, this protocol introduces both text-based and binary communication formats, offering a technical in depth view of how data can be exchanged between PCs and MCUs in an efficient and structured manner. It is important to note that this protocol and the provided code are intended solely for instructional use and should not be applied in production

environments.

With a focus on practical learning, the NAMO Serial Protocol provides a foundation for exploring bi-directional communication. Through its customizable message structure, callback-driven design, and custom command handling, students can gain hands-on experience in implementing UART-based systems. This framework allows learners to experiment with both human-readable commands and compact, machine-efficient binary data, fostering a deeper understanding of real-world communication challenges and solutions.

The protocol emphasizes key concepts such as message framing, data serialization, and deserialization, showcasing how to process and respond to commands dynamically. By supporting both text-based and binary messaging, it accommodates diverse use cases, enabling students to develop versatile communication systems that cater to a wide range of applications. The integration of dynamic callbacks further enhances its usability, allowing for the seamless handling of incoming commands and responses.

NAMO Serial Protocol is designed with simplicity and clarity in mind making it an ideal tool for students, hobbyists, and educators to explore theoretical concepts. The protocol supports two message types: **text** and **binary**. Text-based communication is human-readable, making it easier to debug and interpret commands, while binary communication is compact and efficient for data transfer. Each message consists of a **header** and a **payload**. The header contains metadata such as the key length, value length, and message type, while the payload includes the key (command) and the associated value (data). For instance, the binary command `LED_TOGGLE` with value `0410` toggles the Pico's onboard LED 4 times, with a delay of 10 $\times$ 0x10 milliseconds between toggles (all numbers are HEX-based). Messages are encoded with the following structure:

- **Header (8 bytes):**
    1. `Key Length (4 bytes)`: Indicates the length of the key string.
    2. `Value Length (4 bytes)`: Specifies the size of the value (data) payload.
- **Payload:**
    1. `Key (variable length)`: The command identifier, such as `LED_TOGGLE`. Many commands can be customized.
    2. `Value (variable length)`: The associated data, such as the number of toggles or delay interval, or motor speed and many more.

3. `Message Type (1 byte)`: Specifies whether the message is text (`0`) or binary (`1`). Other types such as Protobuf, B-JSON and many more can be added as well.

This structure ensures flexibility, allowing the protocol to handle both human-readable commands and machine-efficient binary data seamlessly.

The NAMO Serial Protocol allows for a clear distinction between text-based and binary communication. Text-based messages are ideal for debugging and sending human-readable commands, while binary messages are more compact and efficient for transmitting numerical or structured data. For example:

- Text Message: `"LED_TOGGLE"` with value `"4,100"`.
- Binary Message: Key: `LED_TOGGLE`, Value: `0x04 0x10` (indicating 4 toggles with $10 \times$ 0x10 ms delay).

The protocol efficiently handles both types, providing callbacks to process incoming messages based on their type.

The protocol uses a callback-driven architecture, where a user-defined function processes received messages. For instance, the `onMessageReceived()` callback handles incoming messages by:

1. Processing the message to execute commands, such as toggling the LED or changing motor speeds, or even reading sensor data.
2. Sending an acknowledgment back to the sender, either as a text or binary response are possible too.

This design ensures modularity, making it easy to add new command types or extend functionality.

Let's go through a sample core implementation (the code follows a header only implementation for easy integration). It can be included in any place that the protocol is needed.

```
//file name is namo_serial_proto.h
#ifndef NAMO_SERIAL_PROTO_H
#define NAMO_SERIAL_PROTO_H

#include <Arduino.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>

namespace namo {
```



```
class SerialProtocol {
  public:
  // Message types
  static const uint8_t MSG_TYPE_TEXT   = 0;
  static const uint8_t MSG_TYPE_BINARY = 1;

  // Maximum sizes
  static const uint32_t MAX_KEY_SIZE   = 256;
  static const uint32_t MAX_VALUE_SIZE = 2048;
  static const uint8_t  HEADER_SIZE    = 8;

  // Callback function type for received messages
  typedef void (*MessageCallback)(const char* key, const uint8_t* value, uint32_t valueLen, uint8_t msgType);

  private:
  uint8_t headerBuffer[HEADER_SIZE];
  uint8_t keyBuffer[MAX_KEY_SIZE + 1];
  uint8_t valueBuffer[MAX_VALUE_SIZE + 1];
  MessageCallback messageCallback;

  static uint32_t convertBytesToUint32(const uint8_t* bytes) {
    return ((uint32_t)bytes[0] << 24) |
    ((uint32_t)bytes[1] << 16) |
    ((uint32_t)bytes[2] <<  8) |
    (uint32_t)bytes[3];
  }

  static void convertUint32ToBytes(uint32_t value, uint8_t* bytes) {
    bytes[0] = (value >> 24) & 0xFF;
    bytes[1] = (value >> 16) & 0xFF;
    bytes[2] = (value >>  8) & 0xFF;
    bytes[3] =  value        & 0xFF;
  }

  bool readExactly(uint8_t* buffer, size_t length) {
    size_t bytesRead = 0;
    unsigned long timeout = millis() + 1000;

    while (bytesRead < length) {
      if (Serial.available() > 0) {
        buffer[bytesRead] = Serial.read();
        bytesRead++;
        timeout = millis() + 1000;
      }
      else if (millis() > timeout) {
        return false;
      }
      yield();
    }
    return true;
  }

  bool writeMessage(const char* key, const uint8_t* value,
  uint32_t valueLength, uint8_t type)
  {
    uint32_t keyLength = strlen(key);
    uint8_t lenBytes[4];
```



```
  if (keyLength > MAX_KEY_SIZE || valueLength > MAX_VALUE_SIZE) {
    return false;
  }

  // Write header (key length + value length)
  convertUint32ToBytes(keyLength, lenBytes);
  Serial.write(lenBytes, 4);

  convertUint32ToBytes(valueLength, lenBytes);
  Serial.write(lenBytes, 4);

  // Write payload
  Serial.write((const uint8_t*)key, keyLength);
  Serial.write(value, valueLength);
  Serial.write(type);

  Serial.flush();
  return true;
}

public:
SerialProtocol() : messageCallback(nullptr) {}

void begin(MessageCallback callback) {
  messageCallback = callback;
}

bool sendText(const char* key, const char* value) {
  return writeMessage(key, (const uint8_t*)value, strlen(value), MSG_TYPE_TEXT);
}

bool sendBinary(const char* key, const uint8_t* value, uint32_t valueLen) {
  return writeMessage(key, value, valueLen, MSG_TYPE_BINARY);
}

void process() {
  if (Serial.available() > 0) {
    // Read and validate header
    if (!readExactly(headerBuffer, HEADER_SIZE)) {
      while (Serial.available()) Serial.read();
      return;
    }

    uint32_t keyLength = convertBytesToUint32(headerBuffer);
    uint32_t valueLength = convertBytesToUint32(headerBuffer + 4);

    // Validate sizes
    if (keyLength == 0 || keyLength > MAX_KEY_SIZE ||
    valueLength == 0 || valueLength > MAX_VALUE_SIZE)
    {
      while (Serial.available()) Serial.read();
      return;
    }

    // Read key
    if (!readExactly(keyBuffer, keyLength)) {
      while (Serial.available()) Serial.read();
      return;
```



```
        }
        keyBuffer[keyLength] = '\0';

        // Read value
        if (!readExactly(valueBuffer, valueLength)) {
          while (Serial.available()) Serial.read();
          return;
        }
        valueBuffer[valueLength] = '\0';

        // Read type
        uint8_t messageType;
        if (!readExactly(&messageType, 1)) {
          while (Serial.available()) Serial.read();
          return;
        }

        // Call callback if registered
        if (messageCallback) {
          messageCallback((char*)keyBuffer, valueBuffer, valueLength, messageType);
        }
      }
    }
  };

} // namespace namo

#endif // NAMO_SERIAL_PROTO_H
```

And here is sample code for using the protocol:

```
#include <Arduino.h>
#include <string.h>
#include "namo_serial_proto.h"

#define LED_PIN 25
#define LED_TOGGLE_KEY "LED_TOGGLE"

namo::SerialProtocol serialProto;

void processMsg(const char* key, const uint8_t* value, uint32_t valueLen, uint8_t msgType)
{
  //strncmp(key, LED_TOGGLE_KEY, strlen(LED_TOGGLE_KEY)) == 0)
  //if (msgType == namo::SerialProtocol::MSG_TYPE_BINARY && key[0] == 'L' && key[1] == 'E')
  if (msgType == namo::SerialProtocol::MSG_TYPE_BINARY && strncmp(key, "LED_TOGGLE", 8) == 0)
  {
    int waitVal = 10 * value[1];
    if (valueLen > 0) {  // Make sure we have data
      for(uint8_t i = 0; i < value[0] && i < 255; i++) {
        digitalWrite(LED_PIN, HIGH);
        delay(waitVal);
        digitalWrite(LED_PIN, LOW);
        delay(waitVal);
      }
    }
  }
}
```



```
void onMessageReceived(const char* key, const uint8_t* value, uint32_t valueLen, uint8_t msgType)
{
  // Process message first
  processMsg(key, value, valueLen, msgType);

  // Then send acknowledgment
  char newKey[namo::SerialProtocol::MAX_KEY_SIZE];
  snprintf(newKey, sizeof(newKey), "%s_ack", key);

  if (msgType == namo::SerialProtocol::MSG_TYPE_TEXT) {
    char newValue[namo::SerialProtocol::MAX_VALUE_SIZE];
    snprintf(newValue, sizeof(newValue), "%s_ack", (char*)value);
    serialProto.sendText(newKey, newValue);
  }
  else if (msgType == namo::SerialProtocol::MSG_TYPE_BINARY) {
    serialProto.sendBinary(newKey, value, valueLen);
  }
}

void setup() {
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW);
  delay(500);
  digitalWrite(LED_PIN, HIGH);
  delay(500);
  digitalWrite(LED_PIN, LOW);


  Serial.begin(9600);
  while (!Serial) delay(10);
  serialProto.begin(onMessageReceived);
}

void loop() {
  serialProto.process();
  // Add your other tasks ...
}
```

Code explanation: In the provided code, the `LED_TOGGLE` command demonstrates binary communication:

```
// Process LED_TOGGLE binary command
if (msgType == namo::SerialProtocol::MSG_TYPE_BINARY && strncmp(key, "LED_TOGGLE", 8) == 0) {
  int waitVal = 10 * value[1];
  for (uint8_t i = 0; i < value[0] && i < 255; i++) {
    digitalWrite(LED_PIN, HIGH);
    delay(waitVal);
    digitalWrite(LED_PIN, LOW);
    delay(waitVal);
  }
}
```

Here, the key `LED_TOGGLE` triggers the onboard LED to toggle `value[0]` times with a delay determined by `value[1]`. This demonstrates compact and efficient binary communication.

After processing a command, the protocol sends an acknowledgment message back to the sender. The acknowledgment appends `_ack` to the original key and echoes the received value. For example:

```
void onMessageReceived(const char* key, const uint8_t* value, uint32_t valueLen, uint8_t msgType) {
  char newKey[namo::SerialProtocol::MAX_KEY_SIZE];
  snprintf(newKey, sizeof(newKey), "%s_ack", key);
  if (msgType == namo::SerialProtocol::MSG_TYPE_TEXT) {
    char newValue[namo::SerialProtocol::MAX_VALUE_SIZE];
    snprintf(newValue, sizeof(newValue), "%s_ack", (char*)value);
    serialProto.sendText(newKey, newValue);
  } else if (msgType == namo::SerialProtocol::MSG_TYPE_BINARY) {
    serialProto.sendBinary(newKey, value, valueLen);
  }
}
```

5.1.3 Example - NAMO Serial Protocol's Java Implementations

In this section, we review high-level drivers for the **NAMO Serial Protocol**. The provided code demonstrates sample implementations of high-level drivers designed to facilitate bi-directional communication between serial devices and TCP clients. This system showcases advanced concepts in serial communication for educational purposes. The key components include:

- `SerialCommunication.java`: Manages serial communication across multiple ports with features like message framing, automatic reconnection, and callback-driven message handling.
- `SerialServer.java`: Acts as a bridge between serial devices and TCP clients by forwarding messages between them using a TCP server.
- `SerialTcpClient.java`: Provides a TCP client implementation to send commands and data to the server, process responses, and support both text and binary communication.

The `SerialCommunication` class is the foundation of the system, responsible for managing communication across multiple serial ports using the `jSerialComm` library. It implements a structured message format consisting of an 8-byte header and a payload. The header specifies metadata such as the lengths of the key and value, while the payload contains the key (command) and value (data) along with a 1-byte message type (`TEXT` or `BINARY`). By framing messages in this structured format, the protocol ensures reliable communication and compatibility with both human-readable and machine-efficient data.

Each serial port in `SerialCommunication` is managed by a dedicated `SerialPortManager` instance. These managers handle sending and receiving messages, monitoring the status of the connection, and reconnecting if a disconnection occurs. This modular approach ensures robust communication while allowing the system to scale efficiently to multiple ports. Additionally, the class notifies registered subscribers whenever a message is received, enabling dynamic response handling.

The `SerialServer` class bridges serial and TCP communication by creating a TCP server that listens on a specified port. It forwards messages received from TCP clients to the appropriate serial devices via the `SerialCommunication` component. Conversely, responses from serial devices are broadcast to all connected TCP clients. The server includes a heartbeat mechanism to monitor the activity of each client and disconnects inactive clients after a configurable timeout. This bridging functionality makes it possible to integrate serial devices with remote systems over a network.

The `SerialTcpClient` class acts as a client for the `SerialServer`, providing functionality for sending and receiving commands. Messages consist of a message type (`TEXT` or `BINARY`), target port, key, and value. The client supports both interactive mode, where users manually input commands, and programmatic mode, where communication is automated through scripts. Heartbeat messages ensure that the client maintains an active connection with the server, while callbacks process responses dynamically.

A significant feature of the NAMO Serial Protocol is its **callback-driven design**, which simplifies the handling of received messages. For example, in the `SerialServer`, messages from serial devices are forwarded to all TCP clients, while in the `SerialTcpClient`, server responses are processed using user-defined listeners. This design modularizes message processing, making it easy to add new features, such as logging or command-specific actions, without disrupting the core architecture.

The system is designed to handle both text-based and binary communication effectively. Text-based messages are ideal for human-readable commands, such as debugging or interactive communication, while binary messages are more efficient for transmitting numerical or structured data. This flexibility demonstrates the advantages of supporting multiple communication formats in real-world scenarios.

By integrating serial and TCP communication, the NAMO Serial Protocol drivers create a versatile system suitable for diverse peripherals. For example, the system can be used for remote monitoring of devices, configuration management, or educational demonstrations of communication protocols. The message framing, error handling, and support for multiple devices make the system practical for such tasks.

The educational value of these high-level drivers lies in their ability to demonstrate core principles of serial communication, such as message framing, serialization,

deserialization, and command handling. We aim to provide a modular implementation to enable students to grasp the complexities of communication protocols while experimenting with real-world use cases.

To compile and run the Java codes for the NAMO Serial Protocol, you will need the **jSerialComm library**, which facilitates serial communication in Java. This library can be downloaded from the Maven repository at the following URL: `https://repo1.maven.org/maven2/com/fazecast/jSerialComm/2.11.0/jSerialComm-2.11.0.jar`.

Ensure that the `jSerialComm-2.11.0.jar` file is in the same directory as your Java source files. The provided commands have been tested using **OpenJDK 23.0**. Below are the compilation instructions for each component:

- To compile the `SerialCommunication.java` file:

```
javac -cp .:jSerialComm-2.11.0.jar SerialCommunication.java
```

- To compile the `SerialServer.java` file, which depends on `SerialCommunication.java`:

```
javac -cp .:jSerialComm-2.11.0.jar SerialCommunication.java SerialServer.java
```

- To compile the `SerialTcpClient.java` file, which also depends on `SerialCommunication.java`:

```
javac -cp .:jSerialComm-2.11.0.jar SerialCommunication.java SerialTcpClient.java
```

To run the `SerialServer`, which bridges serial communication and TCP clients, use the following command. Note that the serial port name for your Pico may vary depending on your operating system. Additionally, ensure that the NAMO Serial Protocol discussed in the previous chapter is correctly loaded onto the Pico:

```
java -cp .:jSerialComm-2.11.0.jar SerialServer 9000 /dev/cu.usbmodem11101 9600 8N1
```

Here:

- `9000`: The TCP port on which the server listens for client connections.
- `/dev/cu.usbmodem11101`: The serial port to which the hardware device is connected.

- `9600`: The baud rate for serial communication.
- `8N1`: The serial configuration (8 data bits, no parity, 1 stop bit).

To run the `SerialTcpClient`, which connects to the `SerialServer` over TCP, use the following command:

```
java -cp .:jSerialComm-2.11.0.jar SerialTcpClient 127.0.0.1 9000
```

Here:

- `127.0.0.1`: The IP address of the server (use `localhost` for local testing).
- `9000`: The TCP port on which the server is listening.

Additional notes:

- Ensure that your Java environment is properly configured, and the `jSerialComm` library is included in the classpath.
- Replace `/dev/cu.usbmodem11101` with the appropriate serial port name for your operating system and device. For example:
  - On Windows: `COM3`, `COM4`, etc.
  - On Linux: `/dev/ttyUSB0`, `/dev/ttyS1`, etc.
- If you encounter issues, verify that the correct serial port and baud rate are being used.
- TCP clients can send commands to the server, which relays them to the serial device, and responses from the serial device are forwarded back to the client. This enables seamless communication between multiple devices.

Codes for SerialCommunication.java:

```
import com.fazecast.jSerialComm.SerialPort;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicBoolean;
import java.lang.Thread;

/**
* Multi-port serial communication manager with automatic reconnect
```



```
* and explicit read() error detection (read == -1).
*
* Message Format (big-endian for lengths):
*  -------------------------------
*  | keyLength (4 bytes)         |
*  | valueLength (4 bytes)       |
*  | key (keyLength bytes)       |
*  | value (valueLength bytes)   |
*  | messageType (1 byte)        |
*  -------------------------------
*/
public class SerialCommunication {

  private static final int RECONNECT_DELAY_MS = 300; // Delay between reconnection attempts

  // Message Types
  public enum MessageType {
    TEXT(0),
    BINARY(1);

    private final int value;
    MessageType(int val) { this.value = val; }
    public int getValue() { return value; }

    public static MessageType fromByte(byte b) {
      return (b == 0) ? TEXT : BINARY;
    }
  }

  // Message Class
  public static class Message {
    private final String key;
    private final byte[] value;
    private final MessageType type;
    private final String sourcePort;

    public Message(String key, byte[] value, MessageType type, String sourcePort) {
      this.key = key;
      this.value = value;
      this.type = type;
      this.sourcePort = sourcePort;
    }

    public String getKey()        { return key; }
    public byte[] getValue()      { return value; }
    public MessageType getType()  { return type; }
    public String getSourcePort() { return sourcePort; }
  }

  // Subscriber interface
  public interface Subscriber {
    void onMessage(Message message);
  }

  // Internal SerialPortManager
  private class SerialPortManager {
    private final String portName;
    private final int baudRate, dataBits, stopBits, parity;
```



```
private final AtomicBoolean running = new AtomicBoolean(true);
private AtomicBoolean isConnected = new AtomicBoolean(false);

// The jSerialComm port
private volatile SerialPort serialPort;
// Worker threads
private Thread readThread;
private Thread writeThread;
private Thread reconnectThread;

// Outgoing message queue
private final BlockingQueue<Message> sendQueue = new LinkedBlockingQueue<>();

public SerialPortManager(String portName, int baudRate, int dataBits, int stopBits, int parity) {
  this.portName = portName;
  this.baudRate = baudRate;
  this.dataBits = dataBits;
  this.stopBits = stopBits;
  this.parity = parity;

  // Attempt to open port initially
  if (!initializePort()) {
    // Start a reconnection loop if the port can't open right now
    startReconnectionThread();
  }
}

/**
* Attempt to open the port. If successful, spawn read/write threads and return true.
* If unsuccessful, return false.
*/
private boolean initializePort() {
  try {
    serialPort = SerialPort.getCommPort(portName);
    serialPort.setBaudRate(baudRate);
    serialPort.setNumDataBits(dataBits);
    serialPort.setNumStopBits(stopBits);
    serialPort.setParity(parity);

    // Non-blocking I/O
    serialPort.setComPortTimeouts(SerialPort.TIMEOUT_NONBLOCKING, 0, 0);

    if (!serialPort.openPort()) {
      System.err.printf("[ERROR] Failed to open port: %s%n", portName);
      this.isConnected.set(false);
      return false;
    }

    System.out.printf("[INFO] Port %s opened successfully.%n", portName);
    startThreads();
    this.isConnected.set(true);
    return true;

  } catch (Exception e) {
    System.err.printf("[ERROR] Exception while initializing port %s: %s%n",
    portName, e.getMessage());
    this.isConnected.set(false);
    return false;
```



```
  }
}

/**
* Start read and write threads
*/
private void startThreads() {
  // Prevent double start
  if (readThread != null && readThread.isAlive()) {
    return; // Already running
  }
  if (writeThread != null && writeThread.isAlive()) {
    return; // Already running
  }

  // READ THREAD
  readThread = new Thread(() -> {
    final ByteBuffer headerBuf = ByteBuffer.allocate(8).order(ByteOrder.BIG_ENDIAN);

    while (running.get()) {
      try {
        if (!isConnected.get()){
          Thread.sleep(10);
          continue;
        }

        headerBuf.clear();
        if (!readFully(headerBuf)) {
          // Not enough data right now or disconnection
          Thread.sleep(10);
          continue;
        }

        headerBuf.flip();
        int keyLength   = headerBuf.getInt();
        int valueLength = headerBuf.getInt();

        if (keyLength <= 0 || keyLength > 1024 ||
        valueLength <= 0 || valueLength > 4096) {
          System.err.printf("[ERROR: %s] Invalid header (keyLen=%d, valueLen=%d)%n",
          portName, keyLength, valueLength);
          continue;
        }

        // Read key
        byte[] keyBytes = new byte[keyLength];
        if (!readFully(keyBytes)) {
          System.err.printf("[ERROR: %s] Partial key read%n", portName);
          continue;
        }

        // Read value
        byte[] valueBytes = new byte[valueLength];
        if (!readFully(valueBytes)) {
          System.err.printf("[ERROR: %s] Partial value read%n", portName);
          continue;
        }
```



```
      // Read type
      byte[] typeByte = new byte[1];
      if (!readFully(typeByte)) {
        System.err.printf("[ERROR: %s] Partial type read%n", portName);
        continue;
      }

      MessageType msgType = MessageType.fromByte(typeByte[0]);
      System.out.printf("[DEBUG: %s] Received: key=%s, valueLen=%d, type=%s%n",
      portName, new String(keyBytes), valueLength, msgType);

      Message inbound = new Message(new String(keyBytes), valueBytes, msgType, portName);
      notifySubscribers(inbound);

    } catch (Exception e) {}/* catch (InterruptedException ie) {
      // Possibly shutting down
      if (!running.get()) {
        // Exiting
        break;
      }
    } catch (Exception e) {
      System.err.printf("[ERROR: %s] Exception in read thread: %s%n",
      portName, e.getMessage());
      if (running.get()) {
        handleDisconnection();
        break;
      }
    }*/
  }
}, "ReadThread-" + portName);
readThread.start();

// WRITE THREAD
writeThread = new Thread(() -> {
  while (running.get()) {
    try {
      if (!isConnected.get()){
        Thread.sleep(10);
        continue;
      }

      Message msg = sendQueue.take();

      byte[] keyBytes   = msg.getKey().getBytes();
      byte[] valueBytes = msg.getValue();

      ByteBuffer header = ByteBuffer.allocate(8).order(ByteOrder.BIG_ENDIAN);
      header.putInt(keyBytes.length).putInt(valueBytes.length);

      System.out.printf("[DEBUG: %s] Sending message: key=%s, valueLen=%d, type=%s%n",
      portName, msg.getKey(), valueBytes.length, msg.getType());

      // Write the header
      int written = serialPort.writeBytes(header.array(), 8);
      if (written < 0) {
        System.err.printf("[ERROR: %s] Write error on header (disconnect?)%n", portName);
        handleDisconnection();
        continue;
```



```
          //break;
        }

        // Write key
        written = serialPort.writeBytes(keyBytes, keyBytes.length);
        if (written < 0) {
          System.err.printf("[ERROR: %s] Write error on key (disconnect?)%n", portName);
          handleDisconnection();
          continue; //break;
        }

        // Write value
        written = serialPort.writeBytes(valueBytes, valueBytes.length);
        if (written < 0) {
          System.err.printf("[ERROR: %s] Write error on value (disconnect?)%n", portName);
          handleDisconnection();
          continue;//break;
        }

        // Write type
        written = serialPort.writeBytes(new byte[]{(byte) msg.getType().getValue()}, 1);
        if (written < 0) {
          System.err.printf("[ERROR: %s] Write error on type (disconnect?)%n", portName);
          handleDisconnection();
          continue; //break;
        }

      } catch (Exception e) {} /*catch (InterruptedException e) {
        // Possibly shutting down
        Thread.currentThread().interrupt();
        break;
      } catch (Exception e) {
        System.err.printf("[ERROR: %s] Exception in write thread: %s%n",
        portName, e.getMessage());
        if (running.get()) {
          handleDisconnection();
          break;
        }
      }*/
    }
  }, "WriteThread-" + portName);
  writeThread.start();
}

/**
* Called if the port is disconnected or an I/O error occurs.
*/
private void handleDisconnection() {
  if (!isConnected.get()) return;
  System.err.printf("[WARN] Port %s disconnected or error occurred. Will attempt to reconnect...%n", portName);

  try{
    Thread.sleep(1000);
  } catch(Exception e){}

  // Stop read/write threads
  //if (readThread != null) readThread.interrupt();
  //if (writeThread != null) writeThread.interrupt();
```



```
  // Close port
  if (serialPort != null && serialPort.isOpen()) {
    serialPort.closePort();
  }
  isConnected.set(false);
  // Start reconnection
  startReconnectionThread();
}

/**
* Launches a separate thread to periodically try re-opening the port
*/
private void startReconnectionThread() {
  if (this.isConnected.get()) return;
  // If there's already a reconnect thread running, skip
  if (reconnectThread != null && reconnectThread.isAlive()) {
    return;
  }

  reconnectThread = new Thread(() -> {
    while (running.get()) {
      try {
        Thread.sleep(RECONNECT_DELAY_MS);
        System.out.printf("[INFO] Attempting to reconnect %s...%n", portName);

        if (initializePort()) {
          System.out.printf("[INFO] Successfully reconnected to %s%n", portName);
          break;  // Done reconnecting
        }
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return;
      } catch (Exception e) {
        System.err.printf("[ERROR: %s] Reconnect attempt failed: %s%n",
        portName, e.getMessage());
      }
    }
  }, "ReconnectThread-" + portName);
  reconnectThread.start();
}

/**
* Read exactly `buf.remaining()` bytes into `buf`.
* If read returns -1, treat as disconnection.
*/
private boolean readFully(ByteBuffer buf) throws InterruptedException {
  while (buf.hasRemaining() && running.get()) {
    byte[] tmp = new byte[1];
    int read = serialPort.readBytes(tmp, 1);

    if (read == -1) {
      handleDisconnection();
      return false;
    }

    if (read > 0) {
      buf.put(tmp[0]);
```



```
    } else {
      Thread.sleep(2); // no data yet
    }
  }
  return !buf.hasRemaining();
}

/**
* Overload to read into a byte[]
* If read returns -1, treat as disconnection.
*/
private boolean readFully(byte[] array) throws InterruptedException {
  int offset = 0;
  while (offset < array.length && running.get()) {
    int read = serialPort.readBytes(array, array.length - offset, offset);

    if (read == -1) {
      System.err.printf("[ERROR: %s] read == -1 (disconnect detected)%n", portName);
      handleDisconnection();
      return false;
    }

    if (read > 0) {
      offset += read;
    } else {
      Thread.sleep(2); // no data yet
    }
  }
  return (offset >= array.length);
}

/**
* Enqueue a message to be sent
*/
public void send(Message msg) {
  try {
    sendQueue.put(msg);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}

/**
* Fully close the manager
*/
public void close() {
  if (!running.compareAndSet(true, false)) {
    return; // Already closed
  }

  if (reconnectThread != null) {
    reconnectThread.interrupt();
  }
  if (readThread != null) {
    readThread.interrupt();
  }
  if (writeThread != null) {
    writeThread.interrupt();
```



```
    }
    if (serialPort != null && serialPort.isOpen()) {
      serialPort.closePort();
    }

    System.out.printf("[INFO] Port %s closed.%n", portName);
  }
}


// Aggregation of Ports + Subscription
private final Map<String, SerialPortManager> portManagers = new ConcurrentHashMap<>();
private final Set<Subscriber> subscribers = ConcurrentHashMap.newKeySet();

/**
* Add a new serial port
*/
public void addSerialPort(String portName, int baudRate, int dataBits, int stopBits, int parity) {
  portManagers.computeIfAbsent(portName,
  p -> new SerialPortManager(portName, baudRate, dataBits, stopBits, parity));
}

/**
* Publish a TEXT message
*/
public void publishText(String key, String value, String targetPort) {
  publish(key, value.getBytes(), MessageType.TEXT, targetPort);
}

/**
* Publish a BINARY message
*/
public void publishBinary(String key, byte[] value, String targetPort) {
  publish(key, value, MessageType.BINARY, targetPort);
}

/**
* Generic publish
*/
private void publish(String key, byte[] value, MessageType type, String targetPort) {
  Message msg = new Message(key, value, type, null);
  if (targetPort != null) {
    SerialPortManager spm = portManagers.get(targetPort);
    if (spm != null) {
      spm.send(msg);
    } else {
      System.err.printf("[ERROR] Port %s not found.%n", targetPort);
    }
  } else {
    // broadcast
    for (SerialPortManager spm : portManagers.values()) {
      spm.send(msg);
    }
  }
}

/**
* Subscribe
```



```
  */
  public void subscribe(Subscriber subscriber) {
    subscribers.add(subscriber);
  }

  /**
  * Unsubscribe
  */
  public void unsubscribe(Subscriber subscriber) {
    subscribers.remove(subscriber);
  }

  /**
  * Close all ports
  */
  public void close() {
    portManagers.values().forEach(SerialPortManager::close);
    portManagers.clear();
  }

  /**
  * Notify all subscribers
  */
  private void notifySubscribers(Message message) {
    for (Subscriber sub : subscribers) {
      sub.onMessage(message);
    }
  }

  /**
  * List available system ports
  */
  public static List<String> listAvailablePorts() {
    List<String> result = new ArrayList<>();
    for (SerialPort sp : SerialPort.getCommPorts()) {
      result.add(sp.getSystemPortPath());
    }
    return result;
  }
}
```

Codes for SerialServer.java:

```
import com.fazecast.jSerialComm.SerialPort;

import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.*;

public class SerialServer {
  private static final long HEARTBEAT_INTERVAL_MS = 2000; // 2 seconds
  private static final long CLIENT_TIMEOUT_MS = 9000;     // 9 seconds

  private final SerialCommunication serialComm;
  private final Set<ClientHandler> clientHandlers = ConcurrentHashMap.newKeySet();
  private final ScheduledExecutorService heartbeatScheduler = Executors.newScheduledThreadPool(1);
```



```
public SerialServer(List<PortConfig> ports) {
  serialComm = new SerialCommunication();

  serialComm.subscribe(message -> {
    StringBuilder sb = new StringBuilder();
    if (message.getType() == SerialCommunication.MessageType.TEXT) {
      sb.append("RECEIVED TEXT ")
      .append(message.getSourcePort()).append(" ")
      .append(message.getKey()).append(" ")
      .append(new String(message.getValue()));
    } else {
      sb.append("RECEIVED BINARY ")
      .append(message.getSourcePort()).append(" ")
      .append(message.getKey()).append(" ");
      for (byte b : message.getValue()) {
        sb.append(String.format("%02X", b));
      }
    }

    String outbound = sb.toString();
    broadcast(outbound);
  });

  for (PortConfig cfg : ports) {
    System.out.printf("Opening port: %s, baud=%d, dataBits=%d, parity=%d, stopBits=%d%n",
    cfg.port, cfg.baudRate, cfg.dataBits, cfg.parity, cfg.stopBits);
    try {
      serialComm.addSerialPort(cfg.port, cfg.baudRate,
      cfg.dataBits, cfg.stopBits, cfg.parity);
      System.out.println("  -> Opened successfully.");
    } catch (Exception e) {
      System.err.printf("  -> Failed to open %s: %s%n", cfg.port, e.getMessage());
    }
  }

  // Start heartbeat monitoring
  startHeartbeatMonitoring();
}

private void startHeartbeatMonitoring() {
  heartbeatScheduler.scheduleAtFixedRate(() -> {
    long currentTime = System.currentTimeMillis();
    synchronized (clientHandlers) {
      Iterator<ClientHandler> iterator = clientHandlers.iterator();
      while (iterator.hasNext()) {
        ClientHandler handler = iterator.next();
        if (currentTime - handler.getLastHeartbeatTime() > CLIENT_TIMEOUT_MS) {
          System.out.println("Client " + handler.getClientAddress() + " timed out, closing connection");
          handler.close();
          iterator.remove();
        } else {
          handler.sendHeartbeat();
        }
      }
    }
  }, HEARTBEAT_INTERVAL_MS, HEARTBEAT_INTERVAL_MS, TimeUnit.MILLISECONDS);
}
```



```
public void startServer(int tcpPort) throws IOException {
  ServerSocket serverSocket = new ServerSocket(tcpPort);
  System.out.println("SerialServer started. Listening on TCP port " + tcpPort);

  while (true) {
    Socket clientSocket = serverSocket.accept();
    ClientHandler handler = new ClientHandler(clientSocket);
    clientHandlers.add(handler);
    handler.start();
  }
}

private void broadcast(String message) {
  synchronized (clientHandlers) {
    for (ClientHandler handler : clientHandlers) {
      handler.send(message);
    }
  }
}

public void shutdown() {
  System.out.println("Shutting down the SerialServer...");
  heartbeatScheduler.shutdown();
  try {
    heartbeatScheduler.awaitTermination(5, TimeUnit.SECONDS);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }

  serialComm.close();

  for (ClientHandler handler : clientHandlers) {
    handler.close();
  }
  clientHandlers.clear();
}

public static class PortConfig {
  public String port;
  public int baudRate;
  public int dataBits;
  public int stopBits;
  public int parity;

  public PortConfig(String port, int baudRate, int dataBits, int parity, int stopBits) {
    this.port = port;
    this.baudRate = baudRate;
    this.dataBits = dataBits;
    this.parity = parity;
    this.stopBits = stopBits;
  }
}

private class ClientHandler extends Thread {
  private final Socket socket;
  private PrintWriter out;
  private BufferedReader in;
```



```
private boolean active = true;
private volatile long lastHeartbeatTime;
private final String clientAddress;

public ClientHandler(Socket socket) {
  this.socket = socket;
  this.lastHeartbeatTime = System.currentTimeMillis();
  this.clientAddress = socket.getRemoteSocketAddress().toString();
}

public String getClientAddress() {
  return clientAddress;
}

public long getLastHeartbeatTime() {
  return lastHeartbeatTime;
}

public void updateHeartbeat() {
  this.lastHeartbeatTime = System.currentTimeMillis();
}

public void sendHeartbeat() {
  send("HEARTBEAT");
}

@Override
public void run() {
  try {
    out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

    out.println("Welcome to NAMO Serial Server!");

    String line;
    while (active && (line = in.readLine()) != null) {
      // Handle heartbeat response
      if ("HEARTBEAT".equals(line.trim())) {
        updateHeartbeat();
        continue;
      }

      line = line.trim();
      if (line.equalsIgnoreCase("QUIT")) {
        out.println("Goodbye!");
        break;
      }

      String[] tokens = line.split(" ", 4);
      if (tokens.length < 4) {
        out.println("ERROR: Invalid command format. Expected 4 tokens minimum.");
        continue;
      }

      String cmdType = tokens[0].toUpperCase();
      String targetPort = tokens[1];
      String key = tokens[2];
      String value = tokens[3];
```



```
        try {
          switch (cmdType) {
            case "TEXT":
            serialComm.publishText(key, value, targetPort);
            out.printf("Sent TEXT to %s (key=%s, value=%s)\n", targetPort, key, value);
            break;

            case "BINARY":
            byte[] data = hexStringToByteArray(value);
            serialComm.publishBinary(key, data, targetPort);
            out.printf("Sent BINARY to %s (key=%s, %d bytes)\n",
            targetPort, key, data.length);
            break;

            default:
            out.println("ERROR: Unrecognized command. Use TEXT or BINARY.");
            break;
          }
        } catch (Exception e) {
          out.println("ERROR: " + e.getMessage());
        }
      }

    } catch (IOException e) {
      System.err.println("ClientHandler encountered an error: " + e.getMessage());
    } finally {
      close();
    }
  }

  public void send(String msg) {
    if (out != null && active) {
      out.println(msg);
    }
  }

  public void close() {
    active = false;
    try {
      if (out != null) out.close();
      if (in != null) in.close();
      if (socket != null && !socket.isClosed()) socket.close();
    } catch (IOException ignored) {
    }
    clientHandlers.remove(this);
  }
}

private static byte[] hexStringToByteArray(String hex) {
  hex = hex.replaceAll("\\s+", "");
  if (hex.length() % 2 != 0) {
    throw new IllegalArgumentException("Hex string must have an even number of characters");
  }
  int len = hex.length() / 2;
  byte[] data = new byte[len];
  for (int i = 0; i < len; i++) {
    data[i] = (byte) Integer.parseInt(hex.substring(2 * i, 2 * i + 2), 16);
```



```
    }
    return data;
  }

  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println("Usage: java SerialServer <tcpPort> <port> <baud> <config> [<port> <baud> <config> ...]");
      System.out.println(" Example: java SerialServer 9000 COM1 9600 8N1 COM2 115200 8N1");
      System.exit(0);
    }

    int tcpPort = Integer.parseInt(args[0]);

    List<PortConfig> portConfigs = new ArrayList<>();
    int i = 1;
    while (i < args.length) {
      if (i + 2 >= args.length) {
        System.err.println("ERROR: Each serial port config requires: <port> <baud> <config>");
        System.exit(1);
      }
      String portName = args[i];
      int baud = Integer.parseInt(args[i + 1]);
      String config = args[i + 2];
      i += 3;

      int dataBits = Character.getNumericValue(config.charAt(0));
      if (dataBits < 5 || dataBits > 8) {
        throw new IllegalArgumentException("Invalid data bits: " + dataBits);
      }

      char parityChar = Character.toUpperCase(config.charAt(1));
      int parity;
      switch (parityChar) {
        case 'N': parity = SerialPort.NO_PARITY;   break;
        case 'E': parity = SerialPort.EVEN_PARITY; break;
        case 'O': parity = SerialPort.ODD_PARITY;  break;
        case 'M': parity = SerialPort.MARK_PARITY; break;
        case 'S': parity = SerialPort.SPACE_PARITY;break;
        default:
        throw new IllegalArgumentException("Invalid parity: " + parityChar);
      }

      int stopBitsVal = Character.getNumericValue(config.charAt(2));
      int stopBits;
      switch (stopBitsVal) {
        case 1: stopBits = SerialPort.ONE_STOP_BIT; break;
        case 2: stopBits = SerialPort.TWO_STOP_BITS; break;
        default:
        throw new IllegalArgumentException("Invalid stop bits: " + stopBitsVal);
      }

      portConfigs.add(new PortConfig(portName, baud, dataBits, parity, stopBits));
    }

    SerialServer server = new SerialServer(portConfigs);

    try {
      server.startServer(tcpPort);
```



```
    } catch (IOException e) {
      System.err.println("Failed to start server: " + e.getMessage());
      server.shutdown();
    }
  }
}
```

Codes for SerialTcpClient.java:

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Consumer;

public class SerialTcpClient {
  private final ClientConfig config;
  private final BlockingQueue<Message> messageQueue;
  private final List<Consumer<String>> messageListeners;
  private static final AtomicBoolean isRunning = new AtomicBoolean(true);
  private static volatile Socket currentSocket;
  private volatile long lastHeartbeatResponse;

  public static class Message {
    private final String type;        // "TEXT" or "BINARY"
    private final String targetPort;  // Serial port name
    private final String key;         // Message key
    private final String value;       // Message value or hex string for binary

    public Message(String type, String targetPort, String key, String value) {
      this.type = type.toUpperCase();
      this.targetPort = targetPort;
      this.key = key;
      this.value = value;

      if (!type.equalsIgnoreCase("TEXT") && !type.equalsIgnoreCase("BINARY")) {
        throw new IllegalArgumentException("Type must be TEXT or BINARY");
      }
    }

    public String format() {
      return String.format("%s %s %s %s", type, targetPort, key, value);
    }
  }

  public static class ClientConfig {
    private final int initialReconnectDelay;
    private final int maxReconnectDelay;
    private final int maxReconnectAttempts;
    private final int socketTimeout;
    private final int connectTimeout;
    private final int heartbeatInterval;
    private final String heartbeatMessage;
    private final boolean keepAlive;

    private ClientConfig(Builder builder) {
      this.initialReconnectDelay = builder.initialReconnectDelay;
```



```
    this.maxReconnectDelay = builder.maxReconnectDelay;
    this.maxReconnectAttempts = builder.maxReconnectAttempts;
    this.socketTimeout = builder.socketTimeout;
    this.connectTimeout = builder.connectTimeout;
    this.heartbeatInterval = builder.heartbeatInterval;
    this.heartbeatMessage = builder.heartbeatMessage;
    this.keepAlive = builder.keepAlive;
  }

  public static class Builder {
    private int initialReconnectDelay = 1000;
    private int maxReconnectDelay = 2000;
    private int maxReconnectAttempts = 0;
    private int socketTimeout = 8000; // 30 seconds
    private int connectTimeout = 5000; // 5 seconds
    private int heartbeatInterval = 1200; // 15 seconds
    private String heartbeatMessage = "HEARTBEAT";
    private boolean keepAlive = true;

    // Builder methods remain the same
    public Builder initialReconnectDelay(int delay) {
      this.initialReconnectDelay = delay;
      return this;
    }

    public Builder maxReconnectDelay(int delay) {
      this.maxReconnectDelay = delay;
      return this;
    }

    public Builder maxReconnectAttempts(int attempts) {
      this.maxReconnectAttempts = attempts;
      return this;
    }

    public Builder socketTimeout(int timeout) {
      this.socketTimeout = timeout;
      return this;
    }

    public Builder connectTimeout(int timeout) {
      this.connectTimeout = timeout;
      return this;
    }

    public Builder heartbeatInterval(int interval) {
      this.heartbeatInterval = interval;
      return this;
    }

    public Builder heartbeatMessage(String message) {
      this.heartbeatMessage = message;
      return this;
    }

    public Builder keepAlive(boolean keepAlive) {
      this.keepAlive = keepAlive;
      return this;
```



```
    }

    public ClientConfig build() {
      return new ClientConfig(this);
    }
  }
}

public SerialTcpClient(ClientConfig config) {
  this.config = config;
  this.messageQueue = new LinkedBlockingQueue<>();
  this.messageListeners = new CopyOnWriteArrayList<>();
  this.lastHeartbeatResponse = System.currentTimeMillis();
}

public void addMessageListener(Consumer<String> listener) {
  messageListeners.add(listener);
}

public void removeMessageListener(Consumer<String> listener) {
  messageListeners.remove(listener);
}

// Send text message to serial port
public void sendText(String port, String key, String value) throws InterruptedException {
  Message msg = new Message("TEXT", port, key, value);
  messageQueue.put(msg);
}

// Send binary message to serial port (value should be hex string)
public void sendBinary(String port, String key, String hexValue) throws InterruptedException {
  Message msg = new Message("BINARY", port, key, hexValue);
  messageQueue.put(msg);
}

private void handleConnection(String serverHost, int serverPort) throws IOException {
  try (Socket socket = new Socket()) {
    currentSocket = socket;

    socket.setKeepAlive(config.keepAlive);
    socket.setSoTimeout(config.socketTimeout);
    socket.connect(new InetSocketAddress(serverHost, serverPort), config.connectTimeout);

    System.out.println("Connected to server: " + serverHost + ":" + serverPort);

    try (PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

      Thread responseThread = startResponseHandler(in);
      Thread heartbeatThread = startHeartbeatSender(out);

      while (!socket.isClosed() && isRunning.get()) {
        Message message = messageQueue.poll(100, TimeUnit.MILLISECONDS);
        if (message != null) {
          try {
            String formattedMessage = message.format();
            out.println(formattedMessage);
```



```
            if (!out.checkError()) {
              System.out.println("Sent: " + formattedMessage);
            } else {
              throw new IOException("Failed to send message - connection lost");
            }
          } catch (Exception e) {
            System.err.println("Error sending message: " + e.getMessage());
            throw e;
          }
        }
      }
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    } finally {
      closeCurrentSocket();
    }
  }
}

private Thread startResponseHandler(BufferedReader in) {
  Thread responseThread = new Thread(() -> {
    try {
      String response;
      while ((response = in.readLine()) != null) {
        if (response.equals(config.heartbeatMessage)) {
          lastHeartbeatResponse = System.currentTimeMillis();
          //System.out.println(config.heartbeatMessage); // Echo heartbeat back
          continue;
        }

        // Handle "RECEIVED" messages from server
        if (response.startsWith("RECEIVED")) {
          for (Consumer<String> listener : messageListeners) {
            try {
              listener.accept(response);
            } catch (Exception e) {
              System.err.println("Error in message listener: " + e.getMessage());
            }
          }
        } else {
          // Handle other server messages
          System.out.println("Server: " + response);
        }
      }
    } catch (SocketTimeoutException e) {
      System.err.println("Server response timeout - connection may be lost");
    } catch (IOException e) {
      if (isRunning.get()) {
        System.err.println("Lost connection to server: " + e.getMessage());
      }
    }
  });
  responseThread.setDaemon(true);
  responseThread.start();
  return responseThread;
}

private Thread startHeartbeatSender(PrintWriter out) {
```



```
    Thread heartbeatThread = new Thread(() -> {
      while (!Thread.currentThread().isInterrupted() && isRunning.get()) {
        try {
          Thread.sleep(config.heartbeatInterval);
          if (System.currentTimeMillis() - lastHeartbeatResponse > config.socketTimeout) {
            throw new IOException("No heartbeat response from server");
          }
          out.println(config.heartbeatMessage);
          if (out.checkError()) {
            throw new IOException("Failed to send heartbeat - connection lost");
          }
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
          break;
        } catch (IOException e) {
          System.err.println("Heartbeat failed: " + e.getMessage());
          break;
        }
      }
    });
    heartbeatThread.setDaemon(true);
    heartbeatThread.start();
    return heartbeatThread;
  }

  private void startInputHandler() {
    Thread inputThread = new Thread(() -> {
      try (BufferedReader consoleReader = new BufferedReader(new InputStreamReader(System.in))) {
        System.out.println("Enter commands in format: <TYPE> <PORT> <KEY> <VALUE>");
        System.out.println("Example: TEXT COM1 testKey HelloWorld");
        System.out.println("         BINARY COM2 binKey DEADBEEF");
        System.out.println("Enter 'exit' to quit");

        String input;
        while (isRunning.get() && (input = consoleReader.readLine().trim()) != null) {
          if ("exit".equalsIgnoreCase(input.trim())) {
            isRunning.set(false);
            closeCurrentSocket();
            break;
          }

          try {
            String[] parts = input.split(" ", 4);
            if (parts.length == 4) {
              Message msg = new Message(parts[0], parts[1], parts[2], parts[3]);
              messageQueue.add(msg);
            } else {
              System.out.println("Invalid format. Use: <TYPE> <PORT> <KEY> <VALUE>");
            }
          } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
          }
        }
      } catch (IOException e) {
        System.err.println("Error reading console input: " + e.getMessage());
      }
    });
    inputThread.setDaemon(true);
```



```
  inputThread.start();
}

public void start(String serverHost, int serverPort, boolean interactive) {

  if (interactive) startInputHandler();

  int reconnectAttempts = 0;
  int currentDelay = config.initialReconnectDelay;

  while (isRunning.get()) {
    try {
      if (config.maxReconnectAttempts > 0 && reconnectAttempts >= config.maxReconnectAttempts) {
        System.err.println("Maximum reconnection attempts reached. Exiting...");
        break;
      }

      handleConnection(serverHost, serverPort);
      reconnectAttempts = 0;
      currentDelay = config.initialReconnectDelay;

    } catch (Exception e) {
      reconnectAttempts++;
      System.err.println(String.format("Connection attempt %d failed: %s",
      reconnectAttempts, e.getMessage()));

      try {
        System.err.println("Waiting " + (currentDelay / 1000) + " seconds before reconnecting...");
        Thread.sleep(currentDelay);
        currentDelay = Math.min(currentDelay * 2, config.maxReconnectDelay);
      } catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
        break;
      }
    }
  }
}

private void closeCurrentSocket() {
  if (currentSocket != null && !currentSocket.isClosed()) {
    try {
      currentSocket.close();
    } catch (IOException e) {
      System.err.println("Error closing socket: " + e.getMessage());
    }
  }
}

public static void main(String[] args) {
  if (args.length != 2) {
    System.out.println("Usage: java SerialTcpClient <server_host> <server_port>");
    return;
  }

  String serverHost = args[0];
  int serverPort = Integer.parseInt(args[1]);

  ClientConfig config = new ClientConfig.Builder().build();
```



```
    SerialTcpClient client = new SerialTcpClient(config);

    // Add message listener for received messages
    client.addMessageListener(message -> System.out.println("Serial message: " + message));

    boolean interactive = false;

    if (interactive) {
      // Start client interactive mode
      client.start(serverHost, serverPort, interactive);
    } else {
      // If you want API mode ... start client in a separate thread since it's blocking
      Thread clientThread = new Thread(() -> client.start(serverHost, serverPort, interactive));
      clientThread.start();
      try {
        while (true) {
          Thread.sleep(2000);
          client.sendBinary("/dev/cu.usbmodem11101", "LED_TOGGEL", "0412");

          Thread.sleep(5000);
          client.sendBinary("/dev/cu.usbmodem11101", "LED_TOGGEL", "1204");
        }
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

5.2 Analog Inputs and Outputs

Analog and digital represent two fundamentally different ways of representing and processing information. Analog signals are continuous, varying smoothly over time like a wave, and can take on any value within their range. This is similar to how a traditional clock's hands move smoothly around the face, or how the volume knob on a stereo can be adjusted to any position. In nature, most physical quantities such as temperature, pressure, and sound exist as analog signals, providing infinite resolution within their physical limits.

Digital signals, in contrast, are discrete and binary, representing information as a series of 1s and 0s (or HIGH and LOW states in electronics). Digital systems sample analog signals at specific intervals and quantize them into distinct levels, like how a digital clock shows time in distinct steps, or how a digital thermometer displays temperature in specific increments. While this discretization means some precision is lost compared to analog signals, digital systems offer numerous advantages including noise immunity, perfect reproduction of signals, easier storage and processing of information, and the ability to implement complex mathematical operations through digital logic.

Analog input and output operations expand the capabilities of the Raspberry Pi Pico, allowing it to interact with devices like sensors and actuators that rely on continuous signals rather than digital HIGH/LOW states.

Here are the details on how Pico's 12-bit ADC converts analog voltage levels into digital values:

Voltage Range:

- The Pico's ADC operates within a voltage range of 0V to 3.3V.
- Any voltage outside this range will be clamped to either 0V or 3.3V.

Bit Resolution:

- The ADC has a resolution of 12 bits.
- This means it can represent the analog voltage with $2^{12} = 4096$ discrete levels.
- The ADC converts the input voltage to a digital value using the following formula:

$$\text{Digital Value} = \frac{\text{Input Voltage}}{\text{Reference Voltage}} \times 4095$$

- Reference Voltage is typically 3.3V for the Pico.

Example Conversions:

- 0V input:

$$\text{Digital Value} = \frac{0V}{3.3V} \times 4095 = 0$$

- 1.65V input (half of 3.3V):

$$\text{Digital Value} = \frac{1.65V}{3.3V} \times 4095 = 2047$$

- 3.3V input:

$$\text{Digital Value} = \frac{3.3V}{3.3V} \times 4095 = 4095$$

- The ADC maps the input voltage range (0V to 3.3V) to the digital value range (0 to 4095).
- Each step in the digital range corresponds to a voltage step of approximately 0.8mV (3.3V / 4095).
- To get the 12-bit value, you can right-shift the result by 4 bits or divide it by 16.

5.2.1 Reading Analog Signals (ADC)

The Raspberry Pi Pico has an Analog-to-Digital Converter (ADC) built into GPIO26, GPIO27, and GPIO28, allowing the board to read analog voltages.

5.2.1.1 Example: Potentiometer

A potentiometer is a variable resistor that provides a changing voltage when adjusted. This voltage can be read by the ADC to demonstrate basic analog input.

```
#include <Arduino.h>
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/adc.h"
#include "pico/multicore.h"


#define ADC_CHANNEL 0         // ADC channel connected to the potentiometer
#define ADC_PIN 26            // GPIO pin connected to the potentiometer
#define ADC_VREF 3.3f         // ADC reference voltage
#define SAMPLE_INTERVAL 100   // Sampling interval in milliseconds
#define VOLTAGE_THRESHOLD 1.5 // Voltage threshold for LED control
#define LED_PIN 25

void setup() {
  Serial.begin(9600);
  adc_init();                           // Initialize ADC hardware
  adc_gpio_init(ADC_PIN);               // Enable ADC on the specified GPIO pin
  adc_select_input(ADC_CHANNEL);        // Select the ADC channel

  // Set up LED GPIO
  gpio_init(LED_PIN);
  gpio_set_dir(LED_PIN, GPIO_OUT);
}

void loop(){
  uint16_t raw_value = adc_read();    // Read the raw ADC value (12-bit)

  // Convert the raw value to voltage
  float voltage = raw_value * ADC_VREF / (1 << ADC_RESOLUTION);

  // Calculate the percentage of the full-scale range
  float percentage = (raw_value * 100.0f) / ((1 << ADC_RESOLUTION) - 1);

  // Print the raw value, voltage, and percentage
  Serial.printf("Raw: %d, Voltage: %.2f V, Percentage: %.1f%%\r\n", raw_value, voltage, percentage);

  // Control LED based on voltage threshold
  if (voltage > VOLTAGE_THRESHOLD) {
    gpio_put(LED_PIN, 1); // Turn on LED
  } else {
    gpio_put(LED_PIN, 0); // Turn off LED
  }
```



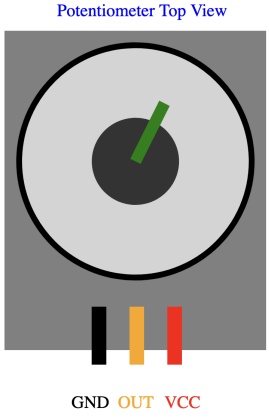
```
  sleep_ms(SAMPLE_INTERVAL); // Wait for the specified sampling interval
}
```

The above code demonstrates how to read values from a potentiometer using the ADC on the Raspberry Pi Pico and control an LED based on a voltage threshold. The code also sets up a serial communication, initializes the ADC hardware, and configures the LED GPIO pin as an output. The main functionality is implemented in the ‘loop()‘ function, which is executed repeatedly. It reads the raw ADC value, converts it to voltage, and calculates the percentage of the full-scale range. The raw value, voltage, and percentage are then printed using serial communication. Additionally, the code controls the LED based on the voltage threshold. If the voltage exceeds the threshold, the LED is turned on; otherwise, it is turned off. The loop function waits for a specified sampling interval before repeating the process.

For proper operation, connect the potentiometer to the Pico as follows:

- Potentiometer VCC → Pico 3V3 (number 36 in the list)
- Potentiometer GND → Any Pico GND pin
- Potentiometer Wiper (Out Signal) → Pico GPIO26 (ADC0)

The potentiometer acts as a voltage divider, providing a variable voltage between 0V and 3.3V to the ADC input. The Pico’s ADC converts this analog voltage to a 12-bit resolution digital value (0-4095), which is then mapped to the PWM range to control LED brightness. As the potentiometer is turned, the voltage division changes, resulting in proportional changes to the LED brightness through PWM duty cycle adjustment. For your convenience, here is the top view of the potentiometer and wiring used:



5.3 Pulse Width Modulation - PWM

Pulse Width Modulation (PWM) is a technique used to control the power delivered to electrical devices by varying the width of pulses in a signal. PWM operates by rapidly switching a digital signal between high and low states, where the proportion of time the signal remains in the high state determines the average power supplied to the device. This technique is widely used in applications such as motor control, LED dimming, and audio signal generation.

At the heart of PWM is the concept of the **duty cycle**, which represents the percentage of the signal's period that the pulse remains high. For instance, a duty cycle of 50% means the pulse is high for half of its period and low for the other half. The formula for calculating the duty cycle is:

$$\text{Duty Cycle (\%)} = \frac{\text{Pulse Width (Time High)}}{\text{Period}} \times 100$$

The **pulse width** refers to the duration for which the signal remains in the high state within a single cycle. As an example in motor control circuits, by adjusting the pulse width, the average output power can be increased or decreased, providing precise control over connected motors and actuator devices.

The **frequency** of a PWM signal is the number of cycles completed per second and is measured in hertz (Hz). The **period** is the inverse of frequency and represents the time taken for one complete cycle of the signal. For example, a frequency of 1 kHz corresponds to a period of 1 millisecond ($T = \frac{1}{f}$). The choice of frequency depends on the application; for instance, motor control typically uses lower frequencies (1-20 kHz), while LED dimming may use higher frequencies to avoid visible flicker.

PWM signals can be classified based on their alignment within the period. In **left-aligned PWM**, the pulse always starts at the beginning of the period, with the high state extending to a variable length based on the duty cycle. In contrast, **center-aligned PWM** positions the high state symmetrically around the center of the period. Center alignment is particularly useful in motor control applications, as it reduces harmonic distortion and provides smoother performance.

The **polarity** of a PWM signal determines the active state of the pulse. A positive polarity signal remains high during the active period and low during the inactive period. Conversely, a negative polarity signal is low during the active period and high during the inactive period. Polarity adjustments are crucial in systems requiring inverted logic or specific signal configurations.

PWM signals are generated by hardware peripherals or software routines that control timers and counters. Hardware PWM is often preferred due to its accuracy and

ability to operate independently of the main processor. For example, microcontrollers like the Raspberry Pi Pico feature hardware PWM modules that can generate precise signals for various applications without burdening the CPU.

One key advantage of PWM is its efficiency. Unlike analog control methods that dissipate excess power as heat, PWM switches the output between fully on and fully off states, minimizing energy loss. This makes PWM particularly suitable for battery-powered and energy-sensitive systems.

In practical applications, PWM signal can control a wide range of devices. For instance, adjusting the duty cycle of a PWM signal can control the speed of a DC motor controller, the brightness of LED controller circuits, or the position of a servo motor. Additionally, by modulating the frequency, PWM can generate audio signals for sound synthesis or communication.

Despite its versatility, PWM has limitations. For example, low-frequency PWM signals can cause audible noise in motor applications, while high-frequency signals may generate electromagnetic interference (EMI). Careful design, including appropriate filtering and grounding, is necessary to address these challenges.

5.3.1 Generating PWM Signals

PWM enables the simulation of analog-like output using digital signals. By varying the duty cycle of a square wave, you can control devices like LEDs and motors once you connect the PWM output signal to the corresponding control circuit of those devices.

In Pico-based PCUs, we can initialize a PWM signal for a specific GPIO. Here are the steps:

- Configure the GPIO pin for PWM functionality.
- Retrieve the corresponding PWM slice and channel for the pin.
- Configure the PWM clock divider and wrap value to set the frequency.
- Initialize the PWM slice with the configuration.
- Set the initial duty cycle for the PWM signal.

Here are the details for each step:

```
// Set GPIO pin to PWM functionality
gpio_set_function(PWM_MOTOR1_PIN, GPIO_FUNC_PWM);
```


This line configures the specified GPIO pin (`PWM_MOTOR1_PIN`) for PWM functionality. The `gpio_set_function()` function maps the pin to the PWM hardware block.

```
// Retrieve the PWM slice and channel for the specified pin
pwm_motor1_slice_num = pwm_gpio_to_slice_num(PWM_MOTOR1_PIN);
pwm_motor1_channel = pwm_gpio_to_channel(PWM_MOTOR1_PIN);
```

The RP2040's PWM controller is divided into multiple slices, each capable of controlling up to two channels. `pwm_gpio_to_slice_num()` determines the slice number associated with the specified GPIO pin. Similarly, `pwm_gpio_to_channel()` identifies the channel (A or B) of the slice for the pin. These values are essential for configuring and controlling PWM signals for the pin.

```
// Get and configure the default PWM settings
pwm_config config = pwm_get_default_config();
pwm_config_set_clkdiv(&config, 4.f);
pwm_config_set_wrap(&config, PWM_TOP_COUNTER);
```

The `pwm_get_default_config()` function retrieves the default configuration for a PWM slice. The clock divider is then set to `4.0`, which scales down the system clock to adjust the PWM frequency. The wrap value (`PWM_TOP_COUNTER`) determines the maximum counter value before the PWM signal resets, effectively controlling the frequency. For example, a lower wrap value results in a higher frequency.

```
// Initialize the PWM slice with the configuration
pwm_init(pwm_motor1_slice_num, &config, true);
```

The `pwm_init()` function applies the specified configuration to the PWM slice and enables it. The third parameter (`true`) ensures that the PWM output starts immediately after initialization.

```
// Set the initial duty cycle
pwm_set_chan_level(pwm_motor1_slice_num, pwm_motor1_channel, 0);
```

This line sets the duty cycle for the specified PWM channel to 0%, meaning the output will initially be LOW. The duty cycle can be updated dynamically using `pwm_set_chan_level()` to control the power delivered to the connected device.

5.3.2 Calculating PWM Frequency on Raspberry Pi Pico

To calculate the **PWM frequency**, we use the relationship between the system clock frequency, clock divider, and the `TOP_COUNTER` value of the PWM:

$$f_{\text{PWM}} = \frac{f_{\text{sys}}}{\text{clkdiv} \times (\text{TOP\_COUNTER} + 1)}$$

As an example, if we use a clock divider of 4 and a `TOP_COUNTER` value of 1000, we will have:

1. **System Clock Frequency** (f_{sys}): The default system clock on the Raspberry Pi Pico 2 is 130 MHz (130×10^6 Hz).

2. **Clock Divider** (clkdiv): And we set the the clock divider to 4.0 using:

   ```
   pwm_config_set_clkdiv(&config, 4.f);
   ```

3. **PWM TOP COUNTER** (TOP_COUNTER): And we set the wrap value to 1000 using:

   ```
   pwm_config_set_wrap(&config, 1000);
   ```

We will have:

$$f_{\text{PWM}} = \frac{130 \times 10^6}{4.0 \times (1000 + 1)} \approx 32\,\text{KHz}$$

5.3.3 Example - Servo Control

A **servo motor** is a device that provides precise control over angular or linear position, velocity, and acceleration. It is commonly used in robotics, automation, and control systems. The SG90 servo motor, targeted in this example, operates based on PWM signals where specific pulse widths correspond to particular angles (e.g., 0° to 180°). Using the Arduino `Servo` library, developers can easily attach a servo to a pin and control its position by specifying the desired angle.

Important note: For servo operation, it is essential to use a dedicated power source. If you choose to use the `VSYS` pin as the power source for your project, ensure that the servo motor operates under minimal or no load. This precaution is critical because the Raspberry Pi Pico's `VSYS` pin, when powered via USB, can supply a maximum of approximately 300mA. Exceeding this current limit risks overloading the USB port, which could result in damage to the USB controller and potentially to your computer's motherboard. For applications requiring higher current, it is strongly recommended to use an external power source connected directly to `VSYS` to ensure reliable operation and to protect your hardware from potential damage.


```
#include <Arduino.h>
#include <Servo.h>

#define SERVO_PIN 9
#define LED_PIN 25  // for visual feedback

Servo myservo;  // create Servo object to control a servo
int myservo_pos = 0;  // variable to store the servo position

void setup() {
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, HIGH);  // Turn on LED to indicate we're ready
  Serial.begin(9600);
}

void setup1() {
  myservo.attach(SERVO_PIN, 544, 2400); // attaches the servo on pin 9 to the Servo object
  // 544 us corresponds approximately to 0 degrees
  // 2400 us corresponds approximately to 180 degrees
}

void loop() {

  Serial.println("Servo sweeping...");
  delay(2000);
}

void loop1() {
  for (myservo_pos = 0; myservo_pos <= 180; myservo_pos++) { // Goes from 0 to 180 degrees
    myservo.write(myservo_pos);
    delay(2); // waits 2ms between steps
  }
  delay(1000);
  for (myservo_pos = 180; myservo_pos >= 0; myservo_pos--) { // Goes from 180 to 0 degrees
    myservo.write(myservo_pos);
    delay(2);
  }
  delay(1000);
}
```

Code explanation: Above code initializes a servo motor connected to pin GP09 of an Arduino and sets up an LED on pin 25 for visual feedback. In the `setup()` function, the LED is turned on to indicate the system is ready, and serial communication is initialized for debugging purposes. In `setup1()`, the servo is attached to pin GP09 with pulse width limits of 544 μs (corresponding to 0°) and 2400 μs (corresponding to 180°), defining the full angular range of the servo.

In the main loop (`loop1()`), the servo performs a continuous sweeping motion. It moves incrementally from 0° to 180° and then back to 0°, with a 2ms delay between each position update to ensure smooth motion. A 1-second pause is included after each sweep to complete the cycle.

5.3.4 Example - LED Brightness Control

PWM is used to adjust the brightness of an LED by varying the duty cycle of the signal applied to it.

```
  // Example: Controlling LED brightness with PWM
#include "pico/stdlib.h"
#include "hardware/pwm.h"
#include "hardware/gpio.h"

#define LED_PIN 25
#define PWM_TOP 65535  // Maximum PWM value (16-bit)

int main() {
  gpio_set_function(LED_PIN, GPIO_FUNC_PWM); // Initialize LED pin for PWM

  // Get PWM slice number for the GPIO
  uint slice_num = pwm_gpio_to_slice_num(LED_PIN);
  uint channel = pwm_gpio_to_channel(LED_PIN);

  pwm_config config = pwm_get_default_config(); // Configure PWM
  pwm_config_set_clkdiv(&config, 4.f);    // Set clock divider
  pwm_config_set_wrap(&config, PWM_TOP);  // Set wrap value
  pwm_init(slice_num, &config, true);     // Initialize PWM with config

  while (true) {
    for (uint32_t i = 0; i <= PWM_TOP; i += 256) { // Fade in
      pwm_set_chan_level(slice_num, channel, i);
      sleep_ms(5);
    }
    sleep_ms(300); // Hold at full brightness briefly

    for (uint32_t i = PWM_TOP; i > 0; i -= 256) { // Fade out
      pwm_set_chan_level(slice_num, channel, i);
      sleep_ms(5);
    }
    sleep_ms(300); // Hold at zero brightness briefly
  }

  return 0;
}
```

For more details refer to `https://www.raspberrypi.com/documentation/pico-sdk/hardware.html#group_hardware_pwm`.

5.3.5 Example - Combining Analog Input and PWM Output

This example demonstrates the integration of analog input and PWM output on the Raspberry Pi Pico, creating an interactive LED brightness control system. The implementation uses a potentiometer as an analog input device to dynamically control the brightness of the Pico's onboard LED through PWM signals.

The system combines analog voltage division through a potentiometer with digital PWM control. The analog input from GPIO26 (ADC0) is converted to a digital value through the Pico's ADC, which then modulates the PWM signal on GPIO25 controlling the onboard LED's brightness.

```
// Example: Controlling LED brightness with a potentiometer
#include "pico/stdlib.h"
#include "hardware/adc.h"
#include "hardware/pwm.h"
#include "hardware/gpio.h"

#define LED_PIN 25
#define POT_PIN 26
#define ADC_CHANNEL 0
#define PWM_TOP 65535  // 16-bit resolution for smoother dimming

int main() {
  // Initialize ADC
  adc_init();
  adc_gpio_init(POT_PIN);     // Enable ADC on GPIO26
  adc_select_input(ADC_CHANNEL); // Select ADC channel 0

  // Initialize LED pin for PWM
  gpio_set_function(LED_PIN, GPIO_FUNC_PWM);

  // Get PWM slice and channel numbers
  uint slice_num = pwm_gpio_to_slice_num(LED_PIN);
  uint channel = pwm_gpio_to_channel(LED_PIN);

  // Configure PWM
  pwm_config config = pwm_get_default_config();
  pwm_config_set_clkdiv(&config, 4.f);    // Set clock divider
  pwm_config_set_wrap(&config, PWM_TOP);   // Set wrap value

  // Initialize PWM with config
  pwm_init(slice_num, &config, true);

  while (true) {
    // Read ADC and map to PWM range
    uint16_t raw_value = adc_read();

    // Map 12-bit ADC (0-4095) to 16-bit PWM (0-65535)
    uint32_t pwm_value = (uint32_t)raw_value * PWM_TOP / 4095;

    // Set PWM duty cycle
    pwm_set_chan_level(slice_num, channel, pwm_value);

    // Small delay for stability
    sleep_ms(10);
  }

  return 0;
}
```

Code explanation:

- The potentiometer's ADC reading is mapped to a PWM duty cycle.
- Adjusting the potentiometer changes the brightness of the LED dynamically.

5.3.6 Example - Basic Motor Speed Control

As we know, MOSFETs (Metal-Oxide-Semiconductor Field-Effect Transistors) are key components in motor control circuits, enabling efficient and precise speed regulation using PWM. In a typical application, the MOSFET acts as a high-speed electronic switch. The PWM signal, which alternates between high and low states, is applied to the MOSFET's gate terminal. When the PWM signal is high, the MOSFET switches on, creating a low-resistance path between its drain and source terminals, allowing current to flow through the motor. When the PWM signal is low, the MOSFET switches off, effectively cutting off the current. By adjusting the PWM duty cycle—the proportion of the "on" time in each cycle—the average voltage supplied to the motor is varied, controlling its speed.

MOSFETs can be good choices for this application due to their fast switching capabilities and high efficiency. Unlike traditional mechanical switches or linear regulators, MOSFETs operate with minimal heat generation because they have very low resistance in the on state and consume negligible power in the off state. Additionally, they can handle the high current demands of DC motors without significant losses. This efficiency makes MOSFETs good choice for battery-powered devices and other systems where energy conservation is critical. Proper selection of the MOSFET, including considerations for voltage rating, current capacity, and switching speed, ensures proper performance in motor control applications.

Figure 5.1 illustrates a basic MOSFET driver circuit. It accepts a PWM signal at its input terminal, along with a DC power source (e.g., a battery), and produces a modulated, controlled voltage at its output, which can drive a motor at the desired speed. In the example below, the ground and GPIO pin GP06 of the Raspberry Pi Pico are connected to the PWM input terminal of the MOSFET driver. The driver is then connected to the DC power source, and the motor is connected to the driver's output, which supplies the regulated, switched voltage to control the motor. A quick demo can be seen *here*

```
// Example: Controlling motor speed with PWM
#include <Arduino.h>
#include <stdio.h>
#include "pico/multicore.h"
#include "pico/stdlib.h"
#include "hardware/adc.h"
#include "hardware/pwm.h"
#include "hardware/gpio.h"
```

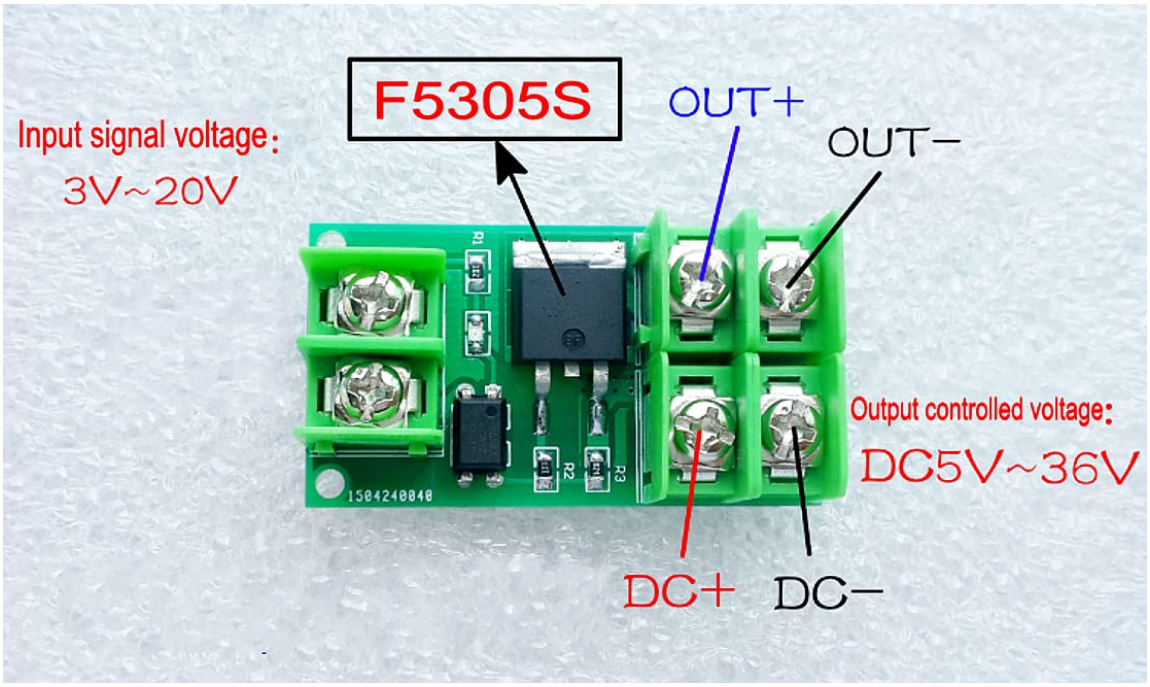



Figure 5.1: Sample MOSFET Driver Circuit (based on IR-F5305S)

```
#define PWM_MOTOR1_PIN 6      // Connect MOSFET-based motor driver to PIN6 for PWM
#define PWM_TOP 1000          // Maximum PWM value (16-bit)
#define ADC_CHANNEL 0         // ADC channel connected to the potentiometer
#define ADC_PIN 26            // GPIO pin connected to the potentiometer
#define ADC_VREF 3.3f         // ADC reference voltage
#define SAMPLE_INTERVAL 100   // Sampling interval in milliseconds
#define VOLTAGE_THRESHOLD 1.5 // Voltage threshold for LED control
#define LED_PIN 25

uint pwm_motor1_slice_num;
uint pwm_motor1_channel;

void setup() {
  gpio_set_function(PWM_MOTOR1_PIN, GPIO_FUNC_PWM); // Initialize LED pin for PWM
  pwm_motor1_slice_num = pwm_gpio_to_slice_num(PWM_MOTOR1_PIN);
  pwm_motor1_channel = pwm_gpio_to_channel(PWM_MOTOR1_PIN);
  pwm_config config = pwm_get_default_config(); // Configure PWM
  pwm_config_set_clkdiv(&config, 4.f);    // Set clock divider
  pwm_config_set_wrap(&config, PWM_TOP);  // Set wrap value
  pwm_init(pwm_motor1_slice_num, &config, true);     // Initialize PWM with config
  pwm_set_chan_level(pwm_motor1_slice_num, pwm_motor1_channel, 0);


  Serial.begin(9600);
  adc_init();                         // Initialize ADC hardware
```



```
  adc_gpio_init(ADC_PIN);              // Enable ADC on the specified GPIO pin
  adc_select_input(ADC_CHANNEL);       // Select the ADC channel

  // Set up LED GPIO
  gpio_init(LED_PIN);
  gpio_set_dir(LED_PIN, GPIO_OUT);
}

void loop(){
  uint16_t raw_value = adc_read();    // Read the raw ADC value (12-bit)

  // Convert the raw value to voltage
  float voltage = raw_value * ADC_VREF / (1 << ADC_RESOLUTION);

  // Calculate the percentage of the full-scale range
  float percentage = (raw_value * 100.0f) / ((1 << ADC_RESOLUTION) - 1);


  uint32_t duty = 0;
  // Different values are used for different motors
  if (voltage < 0.2)
  duty = 0;
  else if (voltage < 0.4)
  duty = 26;
  else if (voltage < 0.8)
  duty = 27;
  else if (voltage < 1.0)
  duty = 28;
  else if (voltage < 1.5)
  duty = 29;
  else if (voltage < 2.0)
  duty = 30;
  else if (voltage < 2.5)
  duty = 31;
  else if (voltage < 3.0)
  duty = 32;
  else if (voltage < 3.2)
  duty = 33;
  else
  duty = PWM_TOP;

  pwm_set_chan_level(pwm_motor1_slice_num, pwm_motor1_channel, duty);

  // Print the raw value, voltage, and percentage
  Serial.printf("Raw: %d, Voltage: %.2f V, Percentage: %.1f%%\r\n", raw_value, voltage, percentage);

  // Control LED based on voltage threshold
  if (voltage > VOLTAGE_THRESHOLD) {
    gpio_put(LED_PIN, 1); // Turn on LED
  } else {
    gpio_put(LED_PIN, 0); // Turn off LED
  }

  sleep_ms(SAMPLE_INTERVAL); // Wait for the specified sampling interval
}
```


5.3.7 Considerations for Driving Motors and Inductive Loads Using PWM and MOSFETs

When using PWM to control motors and inductive loads, such as DC motors or solenoids, several critical considerations must be addressed to ensure reliable and safe operation. Motors and other inductive loads generate a phenomenon called back electromotive force (back EMF) when the current through the inductor changes rapidly, such as when the PWM signal turns off. Back EMF can produce high-voltage spikes that can damage MOSFETs and other circuit components. To address this, a flyback diode (also called a freewheeling diode) is connected in parallel with the inductive load. This diode provides a safe path for the back EMF current to dissipate, preventing voltage spikes from damaging the MOSFET. Similarly, reverse polarity can occur if the power supply to the motor or load is connected incorrectly, potentially causing catastrophic damage to the MOSFET and the load. A reverse polarity protection diode placed in series with the power supply or a MOSFET-based reverse polarity protection switch can prevent such issues.

MOSFETs are commonly used for driving motors due to their high current-handling capacity and low on-resistance ($R_{DS(on)}$). Important parameters to consider include the drain-source voltage (V_{DS}), which must exceed the motor's supply voltage and any potential spikes, and the continuous drain current (I_D), which should accommodate the motor's peak current draw. Efficient gate driving ensures fast switching, minimizing power loss and preventing excessive heat generation. Slow switching leads to operation in the linear region of the MOSFET, increasing power dissipation. Proper gate driver circuits or pull-down resistors on the gate pin ensure stable and efficient operation.

The PWM frequency plays a crucial role in motor performance and the thermal behavior of MOSFETs. Higher frequencies reduce audible noise and improve motor smoothness but increase switching losses, while lower frequencies are more efficient but may produce audible noise or rough motor operation. Typical PWM frequencies for motor control range between 20 kHz and 100 kHz to balance performance and efficiency. Thermal management is equally critical, as MOSFETs dissipate heat due to conduction and switching losses. Heat sinks, thermal pads, or active cooling ensure the MOSFET operates within safe temperature limits, particularly in high-power applications.

Additional considerations include managing inrush current and mitigating electromagnetic interference (EMI). Motors draw a large inrush current during startup or sudden load increases, potentially exceeding the MOSFET's current rating. Soft-start circuits or current-limiting resistors can prevent damage to the MOSFET in these scenarios. EMI generated by high-frequency PWM switching can interfere with nearby sensitive

components or communication lines. Decoupling capacitors near the MOSFET and motor terminals, short and wide PCB traces, and snubber circuits help suppress high-frequency noise and stabilize the circuit.

For bidirectional motor control, H-Bridge configurations are commonly used. An H-Bridge employs multiple MOSFETs to allow forward and reverse operation by controlling the current direction through the motor. Dead-time control is essential to prevent both MOSFETs in a pair from being on simultaneously, which could cause a short circuit. Flyback diodes across each MOSFET handle inductive spikes. Additionally, motors can cause voltage dips and fluctuations in the power supply, which may affect the microcontroller or other components. A bulk capacitor near the motor's power terminals helps stabilize the supply voltage, while isolating the motor's power supply from the microcontroller using separate regulators ensures reliable operation. These considerations are crucial for designing robust and efficient motor control systems.

5.4 Inter-Integrated Circuit - I2C Bus

I2C (Inter-Integrated Circuit) is a serial communication protocol widely used for short-distance communication between devices such as sensors, displays, EEPROMs, and other microcontrollers. It is favored for its simplicity and efficiency, requiring only two lines: `SDA` (Serial Data Line) for data transmission and `SCL` (Serial Clock Line) for synchronization. The Raspberry Pi Pico supports I2C natively, offering multiple I2C controllers (referred to as `i2c0` and `i2c1`), which can be mapped to its GPIO pins for flexible device interfacing.

The Pico's GPIO pins are versatile and can be assigned as `SDA` or `SCL` for I2C communication. For example, the default configuration for `i2c0` uses GPIO4 for `SDA` and GPIO5 for `SCL`, but any GPIO pin can be configured for I2C. This flexibility simplifies circuit design and allows the Pico to accommodate various hardware setups. Additionally, I2C requires pull-up resistors (typically 4.7 kΩ or 10 kΩ) on the `SDA` and `SCL` lines to ensure reliable signal transmission, especially in environments with high electrical noise or long wire runs.

I2C communication operates in a **master-slave** architecture, where the master initiates and controls the communication, and slaves respond as instructed. The Pico can function as either a master or a slave, making it suitable for a wide range of applications. For instance, it can act as a master to control multiple sensors or displays, or as a slave when interfacing with another microcontroller. This versatility allows the Pico to integrate seamlessly into various embedded systems.

Communication begins with the master generating a **start condition**, where the

SDA line transitions from high to low while the SCL line remains high. The master then transmits the address of the target slave device, along with a read/write bit indicating the intended operation. Slaves on the bus monitor the address and acknowledge (ACK) if it matches their assigned address. The master and slave then exchange data in 8-bit frames, with an ACK after each frame. Communication concludes with a **stop condition**, where the SDA line transitions from low to high while the SCL line is high.

The Pico's I2C supports multiple devices on the same bus by utilizing unique 7-bit or 10-bit addresses for each slave device. For example, a temperature sensor might have an address of `0x48`, while an OLED display could use `0x3C`. By specifying the address during communication, the master can interact with each device individually. However, developers must ensure that no two devices share the same address to avoid conflicts.

In terms of speed, the Raspberry Pi Pico supports standard I2C communication modes:

- **Standard Mode:** Up to 100 kbps.
- **Fast Mode:** Up to 400 kbps.
- **Fast Mode Plus:** Up to 1 Mbps.

The choice of mode depends on the application's requirements. For instance, Fast Mode Plus is ideal for high-speed data transfer, while Standard Mode is suitable for applications prioritizing compatibility and low power consumption.

The Pico's SDK and MicroPython environment provide support for I2C communication. In C/C++, the `i2c_init()` function initializes the I2C interface, while `i2c_write_blocking()` and `i2c_read_blocking()` handle data transmission and reception. In MicroPython, the `machine.I2C` class simplifies I2C operations with functions like `scan()`, `readfrom_mem()`, and `writeto_mem()`. For example, initializing I2C on GPIO4 and GPIO5 in MicroPython can be done with:

```
from machine import I2C, Pin
i2c = I2C(0, scl=Pin(5), sda=Pin(4))
devices = i2c.scan()  # Scan for devices on the bus
print("I2C devices found:", devices)
```

For more information refer to:

- `https://docs.circuitpython.org/en/latest/README.html`
- `https://circuitpython.org/board/raspberry_pi_pico2_w`

- `https://circuitpython.org/board/raspberry_pi_pico2`
- `https://circuitpython.org/board/raspberry_pi_pico`
- `https://circuitpython.org/board/raspberry_pi_pico_w`

Another feature of the Pico's I2C is its ability to handle multi-master setups, although this is less common. In such scenarios, collision detection and arbitration mechanisms ensure smooth operation when multiple masters attempt to control the bus simultaneously.

In practical applications, I2C on the Pico is used for tasks like reading sensor data, controlling displays, and communicating with other microcontrollers. For example, a project might involve reading temperature and humidity from an HDC1080 sensor, displaying the values on an OLED screen.

5.4.1 Example - I2C with HDC1080 Temperature and Humidity Sensor

The **HDC1080** is a high-precision, low-power digital sensor developed by Texas Instruments for measuring temperature and relative humidity. It provides an I^2C interface for communication, simplifying integration into microcontroller-based projects. The HDC1080 is suitable for applications such as environmental monitoring, HVAC systems, and IoT devices. The sensor can measure temperature and humidity while allow different configurable modes, resolutions, and power efficiency.

The HDC1080 measures temperature with an accuracy of $\pm 0.2°C$ over a range of $-40°C$ to $125°C$ (different hardware revisions may have different specifications). Temperature data is reported as a 16-bit value, which can be converted using:

$$T(\text{C}) = \frac{\text{Raw Temperature Data}}{2^{16}} \times 165 - 40$$

Relative humidity (RH) is measured with an accuracy of $\pm 2\%$ over a range of 0% to 100%. The raw 16-bit RH value can be converted using:

$$\text{RH } (\%) = \frac{\text{Raw Humidity Data}}{2^{16}} \times 100$$

These capabilities make the HDC1080 a good candidate for precise environmental monitoring in both dry and humid conditions.

The HDC1080 provides configurable measurement modes and resolutions, allowing optimization for different applications:

- **Acquisition Modes:** Supports *sequential* mode (temperature and humidity measured together) and *independent* mode (measured separately).
- **Resolution:** Temperature and humidity resolutions can be set to 14-bit, 11-bit, or 8-bit to balance precision and speed.
- **Heater:** Includes an on-chip heater for defrosting or humidity stabilization.
- **Battery Monitoring:** Detects low supply voltage ($< 2.8\,\mathrm{V}$).

The HDC1080 communicates via the I^2C protocol with a fixed slave address of `0x40`. It supports standard and fast I^2C speeds (up to 400 kHz). The following registers control its operation:

- `0x00`: Temperature measurement output.
- `0x01`: Humidity measurement output.
- `0x02`: Configuration register.
- `0xFE`: Manufacturer ID (`0x5449` for Texas Instruments).
- `0xFF`: Device ID (`0x1050` for HDC1080).

The HDC1080 can be used wide range of applications, including:

- **Weather Monitoring:** Measures environmental temperature and humidity for weather stations.
- **Smart Homes:** Integrates with IoT systems to control HVAC systems based on real-time data.
- **Industrial Monitoring:** Tracks environmental conditions for process optimization.
- **Energy-Efficient Devices:** Low power consumption makes it a candidate for battery-powered systems.

In our example, we demonstrate several advanced operations:

1. **Initialization:** The HDC1080 is reset, and its device ID is verified to ensure correct wiring and functionality. Default settings configure it for sequential acquisition with 14-bit resolution.

2. **Temperature and Humidity Reading:** Measurements are performed in either sequential or independent mode, with results printed in real time.

3. **Dynamic Heater Control:** The heater is activated every 60 seconds for 5 seconds to prevent sensor defrost or humidity oversaturation.

4. **Power Modes:** The sensor automatically enters sleep mode between measurements to conserve power.

5. **Resolution Configuration:** Temperature and humidity resolutions are adjustable via specific bits in the configuration register.

Important Notes: When connecting the HDC1080 to the Raspberry Pi Pico, ensure the following connections are made correctly to guarantee proper operation and avoid potential damages:

1. **Power and Ground Connections:** The HDC1080 operates exclusively at **3.3V**. Connect the Pico's `3V3` pin to the `VCC` pin of the HDC1080 to supply power. Ensure the ground (`GND`) pin of the HDC1080 is connected to the Pico's `GND` pin to establish a common ground reference.

2. **I²C Communication Lines:** Use the Raspberry Pi Pico's `I2C0` interface for communication:

   - Connect `SDA` on GPIO4 of the Pico to the `SDA` pin on the HDC1080.
   - Connect `SCL` on GPIO5 of the Pico to the `SCL` pin on the HDC1080.

3. **Voltage Considerations:** Do not use the Pico's `VSYS` pin or any other voltage source exceeding 3.3V to power the HDC1080, as the sensor is not designed to handle higher voltages and may be permanently damaged. Double-check all connections before powering the circuit.

4. **Pull-Up Resistors:** Some I²C devices require external pull-up resistors on the `SDA` and `SCL` lines. Verify if the HDC1080 module you are using includes built-in pull-up resistors; if not, add 4.7kΩ pull-up resistors between `SDA` and `VCC`, and between `SCL` and `VCC`.

```
#include <Arduino.h>
#include <Wire.h>

#define I2C_HDC1080_ADDR 0x40
#define LED_PIN 25
#define I2C0_SDA_PIN 4
```



```
#define I2C0_SCL_PIN 5

// HDC1080 Registers
#define HDC1080_TEMP_REG 0x00      // Temperature measurement output
#define HDC1080_HUM_REG 0x01       // Humidity measurement output
#define HDC1080_CONFIG_REG 0x02    // Configuration and status
#define HDC1080_SERIAL_ID1 0xFB    // First 2 bytes of serial ID
#define HDC1080_SERIAL_ID2 0xFC    // Mid 2 bytes of serial ID
#define HDC1080_SERIAL_ID3 0xFD    // Last 2 bytes of serial ID
#define HDC1080_MANUFID 0xFE       // Manufacturer ID
#define HDC1080_DEVICEID 0xFF      // Device ID

// Configuration Register Bits
#define HDC1080_CONFIG_RST    0x8000  // Software reset
#define HDC1080_CONFIG_HEAT   0x2000  // Heater
#define HDC1080_CONFIG_MODE   0x1000  // Mode of acquisition
#define HDC1080_CONFIG_BTST   0x0800  // Battery status
#define HDC1080_CONFIG_TRES   0x0400  // Temperature resolution
#define HDC1080_CONFIG_HRES   0x0300  // Humidity resolution

// Expected IDs
#define HDC1080_MANUFID_DEFAULT 0x5449  // TI
#define HDC1080_DEVICEID_DEFAULT 0x1050  // HDC1080

uint16_t temperatureRaw;
uint16_t humidityRaw;
int led_state = HIGH;

struct HDC1080_Config {
  bool heater_enabled;
  bool acquisition_mode;  // 0: Independent, 1: Sequential
  bool battery_status;    // 0: >2.8V, 1: <2.8V
  uint8_t temp_resolution;  // 0: 14 bit, 1: 11 bit
  uint8_t hum_resolution;   // 0: 14 bit, 1: 11 bit, 2: 8 bit
} config;

void I2C_HDC1080_WriteRegister(uint8_t address, uint16_t value) {
  Wire.beginTransmission(I2C_HDC1080_ADDR);
  Wire.write(address);
  Wire.write(value >> 8);
  Wire.write(value & 0xFF);
  Wire.endTransmission();
}

uint16_t I2C_HDC1080_ReadRegister(uint8_t address) {
  Wire.beginTransmission(I2C_HDC1080_ADDR);
  Wire.write(address);
  Wire.endTransmission();

  delay(10);
  Wire.requestFrom(I2C_HDC1080_ADDR, 2);
  uint16_t value = Wire.read() << 8;
  value |= Wire.read();
  return value;
}

void I2C_HDC1080_Reset() {
  I2C_HDC1080_WriteRegister(HDC1080_CONFIG_REG, HDC1080_CONFIG_RST);
```



```
  delay(15);  // Wait for reset to complete
}

void I2C_HDC1080_Sleep() {
  // HDC1080 goes to sleep automatically between measurements
  // No specific command needed
  Serial.println("Device in sleep mode");
}

void I2C_HDC1080_Wake() {
  // Any I2C communication will wake the device
  I2C_HDC1080_ReadRegister(HDC1080_CONFIG_REG);
  Serial.println("Device woken up");
}

void I2C_HDC1080_SetHeater(bool enabled) {
  uint16_t current_config = I2C_HDC1080_ReadRegister(HDC1080_CONFIG_REG);
  if (enabled) {
    current_config |= HDC1080_CONFIG_HEAT;
  } else {
    current_config &= ~HDC1080_CONFIG_HEAT;
  }
  I2C_HDC1080_WriteRegister(HDC1080_CONFIG_REG, current_config);
  config.heater_enabled = enabled;
}

void I2C_HDC1080_SetResolution(uint8_t temp_res, uint8_t hum_res) {
  uint16_t current_config = I2C_HDC1080_ReadRegister(HDC1080_CONFIG_REG);

  // Clear resolution bits
  current_config &= ~(HDC1080_CONFIG_TRES | HDC1080_CONFIG_HRES);

  // Set new resolution
  if (temp_res == 1) current_config |= HDC1080_CONFIG_TRES;  // 11 bit
  if (hum_res == 1) current_config |= 0x0100;  // 11 bit
  else if (hum_res == 2) current_config |= 0x0200;  // 8 bit

  I2C_HDC1080_WriteRegister(HDC1080_CONFIG_REG, current_config);
  config.temp_resolution = temp_res;
  config.hum_resolution = hum_res;
}

void I2C_HDC1080_SetAcquisitionMode(bool sequential) {
  uint16_t current_config = I2C_HDC1080_ReadRegister(HDC1080_CONFIG_REG);
  if (sequential) {
    current_config |= HDC1080_CONFIG_MODE;
  } else {
    current_config &= ~HDC1080_CONFIG_MODE;
  }
  I2C_HDC1080_WriteRegister(HDC1080_CONFIG_REG, current_config);
  config.acquisition_mode = sequential;
}

void I2C_HDC1080_ReadTempHumid() {
  if (config.acquisition_mode) {
    // Sequential mode
    Wire.beginTransmission(I2C_HDC1080_ADDR);
    Wire.write(HDC1080_TEMP_REG);
```



```
    Wire.endTransmission();
    delay(15);  // Wait for both measurements

    Wire.requestFrom(I2C_HDC1080_ADDR, 4);
    temperatureRaw = Wire.read() << 8 | Wire.read();
    humidityRaw = Wire.read() << 8 | Wire.read();
  } else {
    // Independent mode
    // Read temperature
    Wire.beginTransmission(I2C_HDC1080_ADDR);
    Wire.write(HDC1080_TEMP_REG);
    Wire.endTransmission();
    delay(7);  // Wait for temperature measurement

    Wire.requestFrom(I2C_HDC1080_ADDR, 2);
    temperatureRaw = Wire.read() << 8 | Wire.read();

    // Read humidity
    Wire.beginTransmission(I2C_HDC1080_ADDR);
    Wire.write(HDC1080_HUM_REG);
    Wire.endTransmission();
    delay(7);  // Wait for humidity measurement

    Wire.requestFrom(I2C_HDC1080_ADDR, 2);
    humidityRaw = Wire.read() << 8 | Wire.read();
  }
}

float I2C_HDC1080_GetTemp() {
  return ((float)temperatureRaw) * 165.0f / 65536.0f - 40.0f;
}

float I2C_HDC1080_GetRelativeHumidity() {
  return ((float)humidityRaw) * 100.0f / 65536.0f;
}

void I2C_HDC1080_PrintInfo() {
  uint16_t manufID = I2C_HDC1080_ReadRegister(HDC1080_MANUFID);
  uint16_t deviceID = I2C_HDC1080_ReadRegister(HDC1080_DEVICEID);
  uint16_t configReg = I2C_HDC1080_ReadRegister(HDC1080_CONFIG_REG);

  Serial.printf("Manufacturer ID: 0x%04X (Expected: 0x%04X)\r\n", manufID, HDC1080_MANUFID_DEFAULT);
  Serial.printf("Device ID: 0x%04X (Expected: 0x%04X)\r\n", deviceID, HDC1080_DEVICEID_DEFAULT);
  Serial.printf("Config Register: 0x%04X\r\n", configReg);
  Serial.printf("Heater: %s\r\n", config.heater_enabled ? "ON" : "OFF");
  Serial.printf("Mode: %s\r\n", config.acquisition_mode ? "Sequential" : "Independent");
  Serial.printf("Battery: %s\r\n", config.battery_status ? "<2.8V" : ">2.8V");
  Serial.printf("Temperature Resolution: %d-bit\r\n", config.temp_resolution ? 11 : 14);
  Serial.printf("Humidity Resolution: %d-bit\r\n",
  config.hum_resolution == 0 ? 14 :
  config.hum_resolution == 1 ? 11 : 8);
}

bool I2C_HDC1080_Init() {
  Wire.setSCL(I2C0_SCL_PIN);
  Wire.setSDA(I2C0_SDA_PIN);
  Wire.begin();
  Wire.setClock(400000);  // Fast mode 400kHz
```



```
  // Reset device
  I2C_HDC1080_Reset();

  // Verify device IDs
  uint16_t manufID = I2C_HDC1080_ReadRegister(HDC1080_MANUFID);
  uint16_t deviceID = I2C_HDC1080_ReadRegister(HDC1080_DEVICEID);

  if (manufID != HDC1080_MANUFID_DEFAULT || deviceID != HDC1080_DEVICEID_DEFAULT) {
    Serial.println("HDC1080 not found! Check wiring.");
    return false;
  }

  // Default configuration
  config.heater_enabled = false;
  config.acquisition_mode = true;  // Sequential mode
  config.temp_resolution = 0;      // 14-bit
  config.hum_resolution = 0;       // 14-bit

  uint16_t config_reg = 0x1000;    // Sequential mode, 14-bit temp/humidity
  I2C_HDC1080_WriteRegister(HDC1080_CONFIG_REG, config_reg);

  return true;
}

void setup() {
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, led_state);

  Serial.begin(9600);
  while (!Serial.availableForWrite()) delay(100);

  if (!I2C_HDC1080_Init()) {
    while(1) {
      digitalWrite(LED_PIN, !digitalRead(LED_PIN));
      delay(100);  // Fast blink indicates error
    }
  }

  I2C_HDC1080_PrintInfo();

  Serial.println("\r\nTimeElapsed(ms), Temperature(°C), RelativeHumidity(%), HeaterMode(HTM), AcquisitionMode(ACQM)");
}

void loop() {
  digitalWrite(LED_PIN, led_state);
  led_state = !led_state;

  I2C_HDC1080_ReadTempHumid();

  Serial.printf("%lu ms, %.2f °C, %.2f RH, HTM: %d, ACQM: %d\r\n",
  millis(),
  I2C_HDC1080_GetTemp(),
  I2C_HDC1080_GetRelativeHumidity(),
  config.heater_enabled,
  config.acquisition_mode);

  // Turn on heater every 60 seconds for 5 seconds for defrost
```



```
  if ((millis() / 1000) % 60 == 0) {
    I2C_HDC1080_SetHeater(true);
    delay(5000);
    I2C_HDC1080_SetHeater(false);
  }

  delay(2000);
}
```

5.5 Serial Peripheral Interface - SPI Bus

SPI (Serial Peripheral Interface) is a high-speed, full-duplex serial communication protocol widely used for interfacing microcontrollers with peripheral devices such as sensors, displays, SD cards, and memory chips. Unlike I2C, SPI uses separate lines for transmitting and receiving data, making it faster and more efficient for applications requiring high data transfer rates. The Raspberry Pi Pico provides native support for SPI, with multiple SPI controllers (`spi0` and `spi1`) and configurable GPIO mappings.

The SPI protocol uses four main signals:

- `MOSI` (Master Out, Slave In): Data sent from the master to the slave.
- `MISO` (Master In, Slave Out): Data sent from the slave to the master.
- `SCLK` (Serial Clock): Clock signal generated by the master to synchronize data transfers.
- `CS` (Chip Select): Signal used by the master to activate a specific slave device.

SPI operates in a **master-slave** configuration, where the master initiates communication and controls the clock. Multiple slave devices can share the same `MOSI`, `MISO`, and `SCLK` lines, with each slave connected to a unique `CS` pin.

On the Raspberry Pi Pico, the SPI interface can operate at clock speeds up to 133 MHz, making it suitable for high-speed data transfer applications. The GPIO pins are highly configurable, allowing any pin to serve as `MOSI`, `MISO`, `SCLK`, or `CS`. For example, the default configuration for `spi0` uses GPIO16, GPIO17, GPIO18, and GPIO19 for `CS`, `SCK`, `TX (MOSI)`, and `RX (MISO)`, respectively, but these can be reassigned to suit the hardware design.

SPI communication is governed by several modes that define the clock polarity (CPOL) and clock phase (CPHA). These modes, represented as combinations of CPOL and CPHA (Mode 0 to Mode 3), determine the timing and alignment of data transfers:

- **Mode 0 (CPOL = 0, CPHA = 0):** Clock idle low, data sampled on the rising edge.
- **Mode 1 (CPOL = 0, CPHA = 1):** Clock idle low, data sampled on the falling edge.
- **Mode 2 (CPOL = 1, CPHA = 0):** Clock idle high, data sampled on the falling edge.
- **Mode 3 (CPOL = 1, CPHA = 1):** Clock idle high, data sampled on the rising edge.

The choice of SPI mode depends on the requirements of the slave device being interfaced. Ensuring the correct mode is critical for successful communication.

The Raspberry Pi Pico's SDK and MicroPython environments provide support for SPI. In C/C++, the `spi_init()` function initializes the SPI interface, while `spi_write_blocking()` and `spi_read_blocking()` handle data transmission and reception. In MicroPython, the `machine.SPI` class simplifies SPI operations. For example, initializing `spi0` in MicroPython can be done with:

```
from machine import SPI, Pin
spi = SPI(0, baudrate=1000000, polarity=0, phase=0,
sck=Pin(18), mosi=Pin(19), miso=Pin(16))
spi.write(b"Hello")
```

SPI offers high-speed, full-duplex communication, making it suitable for devices requiring rapid data transfer. For instance, an SPI-connected SD card can handle file read/write operations efficiently, while an SPI OLED display can render graphics quickly. Multiple devices can be connected to the same SPI bus by assigning unique `CS` pins to each device, with the master activating only one device at a time.

Despite its advantages, SPI has limitations. It requires more physical lines compared to I2C, as each slave device needs a dedicated `CS` pin. Additionally, SPI lacks built-in acknowledgment or error-checking mechanisms, relying on higher-level protocols or manual handling to ensure data integrity.

The Raspberry Pi Pico's SPI capabilities also include DMA (Direct Memory Access) support, enabling efficient data transfer without CPU intervention. This is particularly beneficial for high-speed applications where minimal latency is required. For example, using DMA to stream data from a sensor to memory ensures smooth operation even at high sampling rates.

5.5.1 Pico to Pico Communication via SPI

Our example below demonstrates a complete SPI communication between two Raspberry Pi Pico boards, where one acts as a master and the other as a slave. The implementation uses the Arduino framework and their SPI libraries to handle the communication.

The master device initializes SPI communication on pins 2-5 (SCK, TX/MOSI, RX/MISO, and CS) and operates at 1 MHz clock speed. In its main loop, it prepares numbered messages (e.g., "Master msg #1"), sends them to the slave using asynchronous transfer, and waits for a response. It includes debug outputs over serial line to monitor the exchange and tracks how many cycles it waited for the slave's response.

The slave device uses the same pin configuration but with TX and RX crossed over in the physical connection (master's TX connects to slave's RX and vice versa). It operates using a callback-based system, where `onDataReceived()` handles incoming messages from the master, and `onDataRequested()` prepares responses (e.g., "Slave response #1"). The slave continuously monitors for incoming data and prints received messages over serial when a complete message arrives.

Both devices use the same SPI configuration (1 MHz speed, MSBFIRST bit order, and `SPI_MODE0`) to have compatible communication. They implement a 64-byte buffer size for messages, and the slave responds to each master message with its own independent numbered response. The master includes a 3-second delay between transmissions to allow human monitoring of the communication channel, while the slave remains continuously ready to receive and respond to messages.

Please note most SPI devices use MSBFIRST, which specifies the bit order for transmission. Another alternative approach is LSBFIRST (Least Significant Bit First). Example of Most Significant Bit First: When sending a byte like `10110011`, the '1' on the far left is sent first (e.g., most significant digit)

For clock characteristics we know that:

- CPOL (Clock Polarity): Determines if clock idles at LOW states or HIGH states
- CPHA (Clock Phase): Determines if data is sampled on first (0) or second (1) clock edge

And for SPI Modes as we covered we know that:

- SPI Mode0: 00 (CPOL=0, CPHA=0):
  - Clock idles at LOW (CPOL=0)

  - Data is sampled on the rising edge (LOW to HIGH transition)
  - Data is changed/shifted on the falling edge
  - This is the most common mode used by SPI devices
- SPI Mode1: 01 (CPOL=0, CPHA=1):
  - Clock idles at LOW (CPOL=0)
  - Data is sampled on the falling edge (HIGH to LOW transition)
  - Data is changed/shifted on the rising edge
  - Clock starts with a rising edge before first data sampling
- SPI Mode2: 10 (CPOL=1, CPHA=0):
  - Clock idles at HIGH (CPOL=1)
  - Data is sampled on the falling edge (HIGH to LOW transition)
  - Data is changed/shifted on the rising edge
  - Less common than modes 0 and 1
- SPI Mode3: 11 (CPOL=1, CPHA=1):
  - Clock idles at HIGH (CPOL=1)
  - Data is sampled on the rising edge (LOW to HIGH transition)
  - Data is changed/shifted on the falling edge
  - Clock starts with a falling edge before first data sampling

Master Pico board code:

```
// Pico SPI Master Board, which communicates with slave device
#include <Arduino.h>
#include <SPI.h>
#include "pico.h"
#include "hardware/spi.h"
#include "hardware/regs/spi.h"

// Pin definitions for SPI communication
const uint8_t PIN_SPI_SCK = 2;   // Serial Clock
const uint8_t PIN_SPI_TX = 3;    // MOSI (Master Out Slave In)
const uint8_t PIN_SPI_RX = 4;    // MISO (Master In Slave Out)
const uint8_t PIN_SPI_CS = 5;    // Chip Select

// SPI configuration
const uint32_t SPI_SPEED = 1000000;  // 1 MHz
const size_t BUFFER_SIZE = 64;
```



```
SPISettings spiConfig(SPI_SPEED, MSBFIRST, SPI_MODE0);


// Message counter
uint32_t messageCount = 0;

void setup() {
  // Setup and stabilize serial communication for debugging
  delay(200);
  Serial.begin(115200);
  delay(200);

  // Configure SPI pins
  SPI.setSCK(PIN_SPI_SCK);
  SPI.setTX(PIN_SPI_TX);
  SPI.setRX(PIN_SPI_RX);
  SPI.setCS(PIN_SPI_CS);

  // Initialize SPI as master
  SPI.begin(true);

  Serial.println("SPI Master initialized");
}

void loop() {
  char txBuffer[BUFFER_SIZE] = {0};
  char rxBuffer[BUFFER_SIZE] = {0};
  uint32_t waitCycles = 0;

  // Prepare message
  snprintf(txBuffer, BUFFER_SIZE, "Master msg #%lu", messageCount);
  Serial.printf("Sending: '%s'\r\n", txBuffer);

  // Perform SPI transaction
  SPI.beginTransaction(spiConfig);
  SPI.transferAsync(txBuffer, rxBuffer, BUFFER_SIZE);

  // Wait for transfer completion
  while (!SPI.finishedAsync()) {
    waitCycles++;
  }
  SPI.endTransaction();

  // Display results
  Serial.printf("Received: '%s'\r\n", rxBuffer);
  Serial.printf("Waited %lu cycles before getting response from slave\r\n\r\n", waitCycles);

  messageCount++;
  delay(3000);  // Wait 3 seconds before our next transmission
}
```

Slave Pico board code:

```
// Pico SPI Slave Board, which communicates with master device
#include <Arduino.h>
#include <SPISlave.h>
#include "pico.h"
#include "hardware/spi.h"
```



```
#include "hardware/regs/spi.h"

// Pin definitions for SPI communication
const uint8_t PIN_SPI_SCK = 2;   // Serial Clock
const uint8_t PIN_SPI_TX = 3;    // MOSI (Master Out Slave In)
const uint8_t PIN_SPI_RX = 4;    // MISO (Master In Slave Out)
const uint8_t PIN_SPI_CS = 5;    // Chip Select
// Note: Reverse the physical wires RX and TX wires between master and slave (e.g., master TX to Slave RX)


// SPI configuration
const uint32_t SPI_SPEED = 1000000;  // 1 MHz mianly for consistency and not really used here
const size_t BUFFER_SIZE = 64;
SPISettings spiConfig(SPI_SPEED, MSBFIRST, SPI_MODE0);

// Communication buffers
volatile bool dataReceived = false;
char receiveBuffer[BUFFER_SIZE] = {0};
char transmitBuffer[BUFFER_SIZE] = {0};
uint32_t receiveIndex = 0;
uint32_t responseCount = 0;

// Callback for receiving data from master
void onDataReceived(uint8_t *data, size_t length) {
  memcpy(receiveBuffer + receiveIndex, data, length);
  receiveIndex += length;

  if (receiveIndex >= BUFFER_SIZE) {
    dataReceived = true;
    receiveIndex = 0;
  }
}

// Callback for preparing data to send to master
void onDataRequested() {
  memset(transmitBuffer, 0, BUFFER_SIZE);
  snprintf(transmitBuffer, BUFFER_SIZE, "Slave response #%lu", responseCount++);
  SPISlave.setData((uint8_t*)transmitBuffer, BUFFER_SIZE);
}

void setup() {
  // Setup and stabilize serial communication for debugging
  delay(200);
  Serial.begin(115200);
  delay(200);

  // Configure SPI pins
  SPISlave.setRX(PIN_SPI_RX);
  SPISlave.setTX(PIN_SPI_TX);
  SPISlave.setSCK(PIN_SPI_SCK);
  SPISlave.setCS(PIN_SPI_CS);

  // Prepare initial response
  onDataRequested();

  // Set up callbacks
  SPISlave.onDataRecv(onDataReceived);
  SPISlave.onDataSent(onDataRequested);
```



```
  // Initialize SPI as slave
  SPISlave.begin(spiConfig);

  Serial.println("SPI Slave initialized");
}

void loop() {
  if (dataReceived) {
    Serial.printf("Received from master: '%s'\r\n", receiveBuffer);
    dataReceived = false;
  }
}
```


5.6 Dual-Core Processing

The RP2040 and RP2350 chips are equipped with dual-core ARM processors that can execute tasks simultaneously, enabling parallelism in embedded systems. For instance, one core can handle sensor data while the other manages communication.

```
#include <Arduino.h>
#include "pico.h"
#include "pico/stdlib.h"
#include "pico/multicore.h"

void setup() {
  // Setup and stabilize serial communication for debugging
  delay(200);
  Serial.begin(115200);
  delay(200);
}

void loop() {
  Serial.printf("Hello from Pico Core1 CPU\r\n");
  delay(3000);
}

void setup1() {
  // Setup and stabilize serial communication for debugging
  delay(200);
  Serial.begin(115200);
  delay(200);
}

void loop1() {
  Serial.printf("Hello from Pico Core2 CPU\r\n");
  delay(1000);
}
```

5.7 PICO Programmable IO (PIO)

The Programmable IO (PIO) block is a distinctive feature of the RP2040 and RP2350 microcontrollers, providing unparalleled flexibility in handling custom IO protocols and interfaces. PIO is a hardware block designed to allow developers to implement complex IO functions that are not natively supported by the microcontroller's standard peripherals, such as UART, SPI, or I2C. Key features of PIO are:

- **State Machines:** Each PIO block contains multiple state machines (up to 4 per block in RP2040 and 12 total across 3 blocks in RP2350). These are programmable, independent units that execute custom IO logic.

- **Custom Protocol Support:** PIO can emulate or implement protocols such as VGA video signals, WS2812 LEDs (NeoPixels), custom serial interfaces, and more.
- **High-Speed Performance:** PIO operates independently of the CPU and can run at clock speeds matching the system's maximum frequency, allowing it to handle high-speed signaling.
- **Dual FIFOs:** Each state machine is equipped with two FIFOs (Transmit and Receive) for efficient data communication between the CPU and PIO.
- **GPIO Flexibility:** PIO allows arbitrary GPIO pins to be assigned as inputs or outputs, providing extreme flexibility in hardware design.
- **Interrupt Integration:** The PIO block can raise interrupts to the CPU for event-driven applications.

5.7.1 PIO Programming Model

PIO programs are written in a simple assembly-like language designed specifically for controlling GPIO pins. The instructions include commands for setting, clearing, or toggling GPIOs, waiting for conditions, and branching. The PIO assembly language allows precise control over timing and signal behavior.

Once a program is written, it is loaded into the PIO's instruction memory, where state machines execute the instructions independently of the CPU. The CPU interacts with the PIO via its FIFOs, loading data to be transmitted or reading received data.

5.7.1.0.1 Advantages of PIO

- **CPU Offloading:** PIO handles IO tasks independently, freeing the CPU for other processing.
- **Customizability:** Developers can implement virtually any protocol or timing-critical IO operation.
- **Scalability:** Multiple state machines can operate in parallel, handling different tasks simultaneously.
- **Deterministic Timing:** PIO provides deterministic and low-latency signal control, critical for real-time applications.

Appendices

A Setting Up Java, Maven, and Gradle on macOS Using Homebrew

This appendix provides a step-by-step guide for setting up Java, Maven, and Gradle on macOS using Homebrew. Follow these instructions to configure your development environment efficiently.

A.1 Installing the Tools via Homebrew

To install Java, Maven, and Gradle, use the following command:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install openjdk@23 gradle maven
```

A.2 Locating Homebrew Installation Paths

Homebrew typically installs these tools in the following directories (check your installed versions):

- **Java:** `/opt/homebrew/Cellar/openjdk/23.0.1/libexec/openjdk.jdk/Contents/Home`
- **Gradle:** `/opt/homebrew/Cellar/gradle/8.12/bin/gradle`
- **Maven:** `/opt/homebrew/Cellar/maven/3.9.9`

To confirm the paths, use:

```
brew info openjdk@23
brew info gradle
brew info maven
```

A.3 Copying the Installations to a Personal Target Directory

For easier management, copy the installations to a target directory in your home folder:

```
export TARGET_DIR=${HOME}/java  # Replace with your preferred directory

mkdir -p ${TARGET_DIR}
cp -aR /opt/homebrew/Cellar/openjdk/23.0.1/libexec/openjdk.jdk/Contents/Home ${TARGET_DIR}
cp -aR /opt/homebrew/Cellar/gradle/8.12 ${TARGET_DIR}/gradle
```



```
cp -aR /opt/homebrew/Cellar/maven/3.9.9 ${TARGET_DIR}/maven

# Update ownership to the current user
chown -R $(whoami) ${TARGET_DIR}
```

A.4 Setting Environment Variables

To configure the environment variables, add the following lines to your shell configuration file (e.g., `~/.zshrc` or `~/.bashrc`):

```
export TARGET_DIR=${HOME}/java  # Or the directory you selected earlier
export JAVA_HOME=${TARGET_DIR}
export JDK_HOME=${JAVA_HOME}
export GRADLE_HOME=${JAVA_HOME}/gradle
export MAVEN_HOME=${JAVA_HOME}/maven
export GRADLE_USER_HOME=${JAVA_HOME}/gradle.cache
export MAVEN_USER_HOME=${JAVA_HOME}/maven.cache

export PATH=${JAVA_HOME}/bin:${GRADLE_HOME}/bin:${MAVEN_HOME}/bin:$PATH

export JAVA_OPTS="-Xmx1g -Xms512m"
export GRADLE_OPTS="-Xmx2g -Dorg.gradle.daemon=true"
export MAVEN_OPTS="-Xmx2g"
```

Apply the changes by running:

```
source ~/.zshrc  # Or ~/.bashrc, depending on your shell
```

A.5 Verifying the Setup

Verify that the tools are correctly configured by checking their versions:

```
java --version
gradle --version
mvn --version
```

The outputs should point to the versions installed in your target directory.

B NAMO Serial Protocol's Python Implementations

In this section, we discuss Python Serial Drivers for NAMO Protocol. Our codes demonstrates a high-level implementation of a NAMO Serial communication framework leveraging Redis for coordination and serial communication for device interaction.

Redis, is an open-source in-memory, key-value data store known for its high performance and versatility. Unlike traditional databases, Redis operates as an in-memory store, allowing it to handle extremely fast read and write operations. It supports a variety of data structures, including strings, lists, sets, hashes, and sorted sets, making it suitable for a broad range of use cases such as caching, session management, real-time analytics, and **message brokering**. Redis also includes features like persistence, where data can be saved to disk and restored after a restart, providing durability, **logging, and audit capabilities** alongside its speed.

One of Redis's standout features is its support for **publish-subscribe messaging**, which allows applications to implement real-time communication patterns. Through this mechanism, clients can **subscribe to specific channels** and receive messages published to those channels in real time. This makes Redis an ideal choice for **distributed systems** requiring coordination, notification systems, or communication between components. Its simplicity, combined with support for complex use cases, has made Redis a widely adopted tool across various industries. Additionally, its compatibility with multiple programming languages and lightweight architecture make it a go-to solution for developers.

We decided to include three primary components: a Redis Serial Client, a Redis Serial Server, and the original serial communication layer:

Redis Serial Client: This client interacts with Redis channels to send commands and receive responses. Commands are encapsulated as objects containing fields like type, targetPort, key, and value. Commands can either be text or binary, allowing versatile messaging capabilities. The RedisSerialClient connects to a Redis server using redis-py, handles commands via a publish-subscribe pattern, and provides functions like send_text and send_binary to publish commands to the serial_commands channel. Responses are captured via the serial_responses channel, where a listener mechanism notifies subscribers of incoming messagesrver: Acting as a bridge, the server listens to commands from Redis and communicates with serial devices.

The server parses the commands received from the serial_commands channel, identifies whether they are text or binary, and forwards them to the appropriate serial port using the SerialCommunication layer. It also publishes the responses or errors back to the Redis serial_responses channel. The server supports multiple serial ports, specified via configurations passed at runtime, and uses error-handling to ensure

reliability in serial communication (in case connections are lost, they can be resumed).

Serial Communication Component handles direct interaction with serial ports. It supports multi-port setups and provides methods for publishing text and binary messages. Internally, it manages threads for reading and writing data, reconnection handling, and subscriber notifications. Messages are framed with headers, which specify the lengths of keys and values, followed by the message payload and type (text or binary). The layer includes features like non-blocking I/O, reconnection threads, and message queues to ensure proper operation even in scenarios like abrupt disconnections .

To run the provided codes, ensure that you have **Python 3.13.1** installed on your system. This is the version we tested and the code may work with other versions. The setup requires the installation of two essential Python packages: `pyserial` and `redis`. These packages enable serial communication and interaction with the Redis server, respectively. Use the following `pip` commands to install the required packages:

```
python3 -m venv .venv
source .venv/bin/activate
pip install pyserial==3.5
pip install redis==5.2.1
```

- **pyserial (version 3.5):** This library provides a simple interface for accessing and interacting with serial ports in Python, essential for handling communication with connected hardware devices.

- **redis (version 5.2.1):** This is a Python client library for Redis, enabling communication with the Redis server using the publish-subscribe messaging model. You need to have working redis server on your machine or your network. Setting up a Redis server is straightforward and involves downloading, installing, and running the Redis server application. On most operating systems, you can install Redis using a package manager. For example, on macOS, you can use Homebrew with the command brew install redis, while on Linux, you can use apt install redis-server (for Ubuntu) or the equivalent package manager for your distribution. For Windows, the recommended approach is to use the precompiled binaries available from the official Redis website or use the Windows Subsystem for Linux (WSL) to run Redis. Once installed, you can start the Redis server by running redis-server from the terminal. By default, the server listens on port 6379.

- To verify that the Redis server is running, you can use the command redis-cli ping. If the server is active, it will respond with PONG. The Redis configuration

file (redis.conf) can be used to customize settings such as the port number, data persistence, or security options. For more advanced setups, you can enable features like authentication and clustering by modifying the configuration file. It's important to secure the Redis server, especially in production environments, by setting a strong password and limiting access to trusted clients.

The `redis_serial_server.py` script serves as the bridge between the serial device and the Redis server. Use the following command to run the server:

```
python redis_serial_server.py /dev/cu.usbmodem11101 9600 8N1
```

- `/dev/cu.usbmodem11101`: Replace this with the appropriate serial port for your device. The serial port may vary depending on your operating system (e.g., `COM3` on Windows or `/dev/ttyUSB0` on Linux).
- `9600`: This specifies the baud rate for communication with the serial device.
- `8N1`: The serial configuration, representing 8 data bits, no parity, and 1 stop bit.

The `redis_serial_client.py` script acts as the client, interacting with the Redis server to send commands and receive responses. Run the client using the following command:

```
python redis_serial_client.py
```

The client listens for commands and messages published to Redis channels and processes them according to the protocol. Ensure that the Redis server is running and accessible, as the client depends on it for communication.

Codes for serial_communication.py

```
import serial
import threading
import queue
import time
import struct
from enum import Enum
from dataclasses import dataclass
from typing import List, Dict, Optional, Callable, Set
from concurrent.futures import ThreadPoolExecutor

class MessageType(Enum):
TEXT = 0
BINARY = 1

@dataclass
```



```
class Message:
key: str
value: bytes
type: MessageType
source_port: Optional[str] = None

class SerialCommunication:
RECONNECT_DELAY_MS = 300  # Delay between reconnection attempts

def __init__(self):
self.port_managers: Dict[str, 'SerialPortManager'] = {}
self.subscribers: Set[Callable] = set()
self._lock = threading.Lock()

def add_serial_port(self, port_name: str, baud_rate: int, data_bits: int, stop_bits: int, parity: str):
with self._lock:
if port_name not in self.port_managers:
self.port_managers[port_name] = SerialPortManager(
port_name, baud_rate, data_bits, stop_bits, parity, self._notify_subscribers
)

def publish_text(self, key: str, value: str, target_port: str):
self.publish(key, value.encode(), MessageType.TEXT, target_port)

def publish_binary(self, key: str, value: bytes, target_port: str):
self.publish(key, value, MessageType.BINARY, target_port)

def publish(self, key: str, value: bytes, msg_type: MessageType, target_port: str = None):
msg = Message(key, value, msg_type, None)
if target_port:
if target_port in self.port_managers:
self.port_managers[target_port].send(msg)
else:
print(f"[ERROR] Port {target_port} not found.")
else:
# Broadcast
for manager in self.port_managers.values():
manager.send(msg)

def subscribe(self, callback: Callable[[Message], None]):
with self._lock:
self.subscribers.add(callback)

def unsubscribe(self, callback: Callable[[Message], None]):
with self._lock:
self.subscribers.discard(callback)

def _notify_subscribers(self, message: Message):
with self._lock:
for subscriber in self.subscribers:
try:
subscriber(message)
except Exception as e:
print(f"[ERROR] Subscriber error: {e}")

def close(self):
for manager in self.port_managers.values():
manager.close()
```



```
self.port_managers.clear()

class SerialPortManager:
def __init__(self, port_name: str, baud_rate: int, data_bits: int, stop_bits: int,
parity: str, callback: Callable[[Message], None]):
self.port_name = port_name
self.baud_rate = baud_rate
self.data_bits = data_bits
self.stop_bits = stop_bits
self.parity = parity
self.callback = callback

self.running = True
self.is_connected = threading.Event()
self.send_queue = queue.Queue()
self.serial_port = None

# Start threads
if not self._initialize_port():
self._start_reconnection_thread()

def _initialize_port(self) -> bool:
try:
self.serial_port = serial.Serial(
port=self.port_name,
baudrate=self.baud_rate,
bytesize=self.data_bits,
parity=self.parity,
stopbits=self.stop_bits,
timeout=0  # Non-blocking mode
)

print(f"[INFO] Port {self.port_name} opened successfully.")
self.is_connected.set()
self._start_threads()
return True

except Exception as e:
print(f"[ERROR] Failed to open port {self.port_name}: {e}")
self.is_connected.clear()
return False

def _start_threads(self):
# Read thread
self.read_thread = threading.Thread(target=self._read_loop, daemon=True)
self.read_thread.start()

# Write thread
self.write_thread = threading.Thread(target=self._write_loop, daemon=True)
self.write_thread.start()

def _read_loop(self):
while self.running:
if not self.is_connected.is_set():
time.sleep(0.01)
continue

try:
```



```
# Read header (8 bytes: 4 for key length, 4 for value length)
header = self._read_exactly(8)
if not header:
continue

key_length, value_length = struct.unpack('>II', header)

if not (0 < key_length <= 1024 and 0 < value_length <= 4096):
print(f"[ERROR: {self.port_name}] Invalid header lengths")
continue

# Read key
key_bytes = self._read_exactly(key_length)
if not key_bytes:
continue

# Read value
value_bytes = self._read_exactly(value_length)
if not value_bytes:
continue

# Read type (1 byte)
type_byte = self._read_exactly(1)
if not type_byte:
continue

msg_type = MessageType(type_byte[0])
key = key_bytes.decode()

print(f"[DEBUG: {self.port_name}] Received: key={key}, "
f"valueLen={len(value_bytes)}, type={msg_type}")

msg = Message(key, value_bytes, msg_type, self.port_name)
self.callback(msg)

except Exception as e:
print(f"[ERROR: {self.port_name}] Read error: {e}")
self._handle_disconnection()

def _write_loop(self):
while self.running:
try:
if not self.is_connected.is_set():
time.sleep(0.01)
continue

msg = self.send_queue.get()

key_bytes = msg.key.encode()
value_bytes = msg.value

# Create header
header = struct.pack('>II', len(key_bytes), len(value_bytes))

# Send header
if self._write_all(header) <= 0:
continue
```



```
# Send key
if self._write_all(key_bytes) <= 0:
continue

# Send value
if self._write_all(value_bytes) <= 0:
continue

# Send type
if self._write_all(bytes([msg.type.value])) <= 0:
continue

print(f"[DEBUG: {self.port_name}] Sent message: key={msg.key}, "
f"valueLen={len(value_bytes)}, type={msg.type}")

except Exception as e:
print(f"[ERROR: {self.port_name}] Write error: {e}")
self._handle_disconnection()

def _read_exactly(self, size: int) -> Optional[bytes]:
buffer = bytearray()
while len(buffer) < size and self.running:
if not self.is_connected.is_set():
return None

try:
chunk = self.serial_port.read(size - len(buffer))
if chunk:
buffer.extend(chunk)
else:
time.sleep(0.002)
except Exception as e:
self._handle_disconnection()
return None

return bytes(buffer) if len(buffer) == size else None

def _write_all(self, data: bytes) -> int:
try:
return self.serial_port.write(data)
except Exception as e:
self._handle_disconnection()
return -1

def _handle_disconnection(self):
if not self.is_connected.is_set():
return

print(f"[WARN] Port {self.port_name} disconnected. Attempting to reconnect...")
self.is_connected.clear()

if self.serial_port:
try:
self.serial_port.close()
except:
pass

time.sleep(1)
```



```
self._start_reconnection_thread()

def _start_reconnection_thread(self):
def reconnect_loop():
while self.running and not self.is_connected.is_set():
try:
time.sleep(SerialCommunication.RECONNECT_DELAY_MS / 1000)
print(f"[INFO] Attempting to reconnect {self.port_name}...")

if self._initialize_port():
print(f"[INFO] Successfully reconnected to {self.port_name}")
break
except Exception as e:
print(f"[ERROR: {self.port_name}] Reconnect attempt failed: {e}")

threading.Thread(target=reconnect_loop, daemon=True).start()

def send(self, msg: Message):
self.send_queue.put(msg)

def close(self):
self.running = False
self.is_connected.clear()
if self.serial_port:
self.serial_port.close()

@staticmethod
def list_available_ports() -> List[str]:
import serial.tools.list_ports
return [port.device for port in serial.tools.list_ports.comports()]
```

Codes for redis_serial_server.py

```
import redis
import json
import time
import threading
from dataclasses import dataclass
from typing import List
import serial
from serial_communication import SerialCommunication, Message, MessageType

@dataclass
class PortConfig:
port: str
baud_rate: int
data_bits: int
parity: str
stop_bits: int

class RedisSerialServer:
COMMAND_CHANNEL = "serial_commands"
RESPONSE_CHANNEL = "serial_responses"

# Redis configuration
REDIS_HOST = "192.168.40.1"
REDIS_PASSWORD = "YOURPASSWORD"
REDIS_PORT = 6379
```



```
def __init__(self, ports: List[PortConfig]):
# Initialize Redis connection pool
self.redis_pool = redis.ConnectionPool(
host=self.REDIS_HOST,
port=self.REDIS_PORT,
password=self.REDIS_PASSWORD,
decode_responses=True
)

self.serial_comm = SerialCommunication()
self.is_running = True

# Initialize serial ports
self.initialize_ports(ports)

# Setup serial message handler
self.setup_serial_handler()

# Start command listener thread
self.subscribe_thread = self.start_command_listener()

def initialize_ports(self, ports: List[PortConfig]):
for cfg in ports:
print(f"Opening port: {cfg.port}, baud={cfg.baud_rate}, dataBits={cfg.data_bits}, "
f"parity={cfg.parity}, stopBits={cfg.stop_bits}")
try:
self.serial_comm.add_serial_port(
cfg.port, cfg.baud_rate, cfg.data_bits, cfg.stop_bits, cfg.parity
)
print("  -> Opened successfully.")
except Exception as e:
print(f"  -> Failed to open {cfg.port}: {e}")

def setup_serial_handler(self):
def message_handler(message: Message):
response_parts = []
if message.type == MessageType.TEXT:
response_parts.extend([
"RECEIVED TEXT",
message.source_port,
message.key,
message.value.decode() if isinstance(message.value, bytes) else message.value
])
else:
response_parts.extend([
"RECEIVED BINARY",
message.source_port,
message.key,
''.join(f"{b:02X}" for b in message.value)
])

self.publish_response(" ".join(response_parts))

self.serial_comm.subscribe(message_handler)

def start_command_listener(self):
def listener_thread():
```



```
while self.is_running:
try:
redis_client = redis.Redis(connection_pool=self.redis_pool)
pubsub = redis_client.pubsub()
pubsub.subscribe(self.COMMAND_CHANNEL)

print(f"Connected to Redis server at {self.REDIS_HOST}")

for message in pubsub.listen():
if not self.is_running:
break

if message['type'] == 'message':
self.handle_command(message['data'])

except Exception as e:
print(f"Redis connection error: {e}")
if self.is_running:
time.sleep(5)
continue

thread = threading.Thread(target=listener_thread, daemon=True)
thread.start()
return thread

def handle_command(self, json_command: str):
try:
cmd = json.loads(json_command)
print(f"Received command: {json_command}")

cmd_type = cmd['type'].upper()
if cmd_type == "TEXT":
self.serial_comm.publish_text(cmd['key'], cmd['value'], cmd['targetPort'])
self.publish_response(
f"Sent TEXT to {cmd['targetPort']} (key={cmd['key']}, value={cmd['value']})"
)
elif cmd_type == "BINARY":
# Convert hex string to bytes (remove spaces and ensure even length)
hex_value = cmd['value'].replace(" ", "")
if len(hex_value) % 2 != 0:
raise ValueError("Hex string must have an even number of characters")

data = bytes.fromhex(hex_value)
print(f"Converted hex to bytes: {data.hex()}")

self.serial_comm.publish_binary(cmd['key'], data, cmd['targetPort'])
self.publish_response(
f"Sent BINARY to {cmd['targetPort']} (key={cmd['key']}, {len(data)} bytes)"
)
else:
self.publish_response(f"ERROR: Invalid command type: {cmd_type}")

except Exception as e:
import traceback
traceback.print_exc()
self.publish_response(f"ERROR: {str(e)}")

def publish_response(self, response: str):
```



```
try:
redis_client = redis.Redis(connection_pool=self.redis_pool)
redis_client.publish(self.RESPONSE_CHANNEL, response)
print(f"Published response: {response}")
except Exception as e:
print(f"Error publishing response: {e}")

def shutdown(self):
self.is_running = False
self.serial_comm.close()
self.redis_pool.disconnect()

def parse_port_config(port_name: str, baud_rate: int, config: str) -> PortConfig:
# Parse data bits
data_bits = int(config[0])
if not (5 <= data_bits <= 8):
raise ValueError(f"Invalid data bits: {data_bits}")

# Parse parity
parity_char = config[1].upper()
parity_map = {
  'N': serial.PARITY_NONE,
  'E': serial.PARITY_EVEN,
  'O': serial.PARITY_ODD,
  'M': serial.PARITY_MARK,
  'S': serial.PARITY_SPACE
}
if parity_char not in parity_map:
raise ValueError(f"Invalid parity: {parity_char}")
parity = parity_map[parity_char]

# Parse stop bits
stop_bits_val = int(config[2])
stop_bits_map = {
  1: serial.STOPBITS_ONE,
  2: serial.STOPBITS_TWO
}
if stop_bits_val not in stop_bits_map:
raise ValueError(f"Invalid stop bits: {stop_bits_val}")
stop_bits = stop_bits_map[stop_bits_val]

return PortConfig(port_name, baud_rate, data_bits, parity, stop_bits)

def main():
import sys

if len(sys.argv) < 4:
print("Usage: python redis_serial_server.py <serial port path> <baud> <config> [<port> <baud> <config> ...]")
print("Example: python redis_serial_server.py /dev/tty.usbmodem11101 9600 8N1")
return

port_configs = []
i = 1
while i < len(sys.argv):
if i + 2 >= len(sys.argv):
print("ERROR: Each serial port config requires: <port> <baud> <config>")
sys.exit(1)
```



```
port_name = sys.argv[i]
baud_rate = int(sys.argv[i + 1])
config = sys.argv[i + 2]
i += 3

try:
port_config = parse_port_config(port_name, baud_rate, config)
port_configs.append(port_config)
except ValueError as e:
print(f"Error parsing port config: {e}")
sys.exit(1)

# Create and start the server
server = RedisSerialServer(port_configs)

try:
print(f"Server started. Listening for commands on Redis channel: {RedisSerialServer.COMMAND_CHANNEL}")
while True:
time.sleep(1)
except KeyboardInterrupt:
print("\nShutting down...")
server.shutdown()

if __name__ == "__main__":
main()
```

Codes for redis_serial_client.py:

```
import redis
import json
import time
import threading
from typing import Callable, List
from dataclasses import dataclass
from datetime import datetime


@dataclass
class SerialCommand:
type: str
target_port: str
key: str
value: str
timestamp: str = None

def __post_init__(self):
self.type = self.type.upper()
if self.type not in ["TEXT", "BINARY"]:
raise ValueError("Type must be TEXT or BINARY")
self.timestamp = str(int(time.time() * 1000))

def to_json(self) -> str:
return json.dumps({
  "type": self.type,
  "targetPort": self.target_port,
  "key": self.key,
  "value": self.value,
  "timestamp": self.timestamp
```



```
})


class RedisSerialClient:
COMMAND_CHANNEL = "serial_commands"
RESPONSE_CHANNEL = "serial_responses"
REDIS_HOST = "192.168.40.1"
REDIS_PASSWORD = "YOURPASSWORD"
REDIS_PORT = 6379

def __init__(self):
# Create Redis connection pool
self.redis_pool = redis.ConnectionPool(
host=self.REDIS_HOST,
port=self.REDIS_PORT,
password=self.REDIS_PASSWORD,
decode_responses=True
)
self.message_listeners: List[Callable] = []
self.is_running = True

# Start response listener thread
self.subscribe_thread = threading.Thread(target=self._start_response_listener, daemon=True)
self.subscribe_thread.start()

def add_message_listener(self, listener: Callable[[str], None]):
"""Add a message listener function."""
self.message_listeners.append(listener)

def remove_message_listener(self, listener: Callable[[str], None]):
"""Remove a message listener function."""
if listener in self.message_listeners:
self.message_listeners.remove(listener)

def _start_response_listener(self):
"""Listen for responses from the serial server."""
while self.is_running:
try:
# Create a new Redis connection for subscription
redis_client = redis.Redis(connection_pool=self.redis_pool)
pubsub = redis_client.pubsub()
pubsub.subscribe(self.RESPONSE_CHANNEL)

print(f"Connected to Redis server at {self.REDIS_HOST}")

for message in pubsub.listen():
if not self.is_running:
break

if message['type'] == 'message':
response = message['data']
# Notify all listeners
for listener in self.message_listeners:
try:
listener(response)
except Exception as e:
print(f"Error in message listener: {e}")
```



```
except Exception as e:
print(f"Redis connection error: {e}")
if self.is_running:
time.sleep(5)  # Wait before retry
continue

def send_text(self, port: str, key: str, value: str):
"""Send a text message to the specified serial port."""
cmd = SerialCommand("TEXT", port, key, value)
self._publish_command(cmd)

def send_binary(self, port: str, key: str, hex_value: str):
"""Send a binary message to the specified serial port."""
cmd = SerialCommand("BINARY", port, key, hex_value)
self._publish_command(cmd)

def _publish_command(self, cmd: SerialCommand):
"""Publish a command to Redis."""
try:
redis_client = redis.Redis(connection_pool=self.redis_pool)
json_cmd = cmd.to_json()
result = redis_client.publish(self.COMMAND_CHANNEL, json_cmd)

if result > 0:
print(f"Published command: {json_cmd}")
else:
print(f"Published command but no subscribers: {json_cmd}")

except Exception as e:
print(f"Error publishing command: {e}")

def shutdown(self):
"""Shutdown the client and cleanup resources."""
self.is_running = False
time.sleep(1)  # Give time for threads to cleanup
self.redis_pool.disconnect()


def main():
# Create Redis client
client = RedisSerialClient()

# Add message listener
def message_handler(message: str):
print(f"Received response: {message}")

client.add_message_listener(message_handler)

try:
print(f"Client started and connected to Redis at {RedisSerialClient.REDIS_HOST}")

while True:
time.sleep(2)
client.send_binary("/dev/cu.usbmodem11101", "LED_TOGGLE", "0412")

time.sleep(5)
client.send_binary("/dev/cu.usbmodem11101", "LED_TOGGLE", "1204")
```



```
except KeyboardInterrupt:
print("\nShutting down...")
client.shutdown()
except Exception as e:
print(f"Error: {e}")
client.shutdown()


if __name__ == "__main__":
main()
```

