Embedded Systems Design

Masudul
Imtiaz

Embedded Systems Design

Utilizing the Silicon Labs EFR32XG24 BLE Microcontroller

Masudul Imtiaz, PhD
Sigmond Kukla
Aaron Storey, MSAI
& TA's of EE260

Coulson School of Engineering and Applied Sciences
Clarkson University

Table of contents

Preface

Welcome

Welcome to the **"System Design with Silicon Lab EFR32XG24 BLE Microcontroller"**. This book is designed to guide you through the process of programming, building applications, and integrating machine learning with the EFR32XG24 BLE Microcontroller. Whether you're an engineering student or a seasoned professional, this book offers hands-on examples to make advanced concepts accessible.

You'll learn how to: - Program the EFR32XG24 microcontroller using C. - Design and implement embedded systems applications. - Apply machine learning techniques to solve real-world problems. - Explore gesture recognition, anomaly detection, and audio-based ML solutions.

The book balances theory with practice, empowering readers to develop embedded systems that are robust, efficient, and intelligent.

If you're interested in broader programming concepts or other machine learning platforms, we encourage you to explore additional resources and apply your learning across domains.

> **i** This book was originally developed as part of the EE260 and EE513 courses at Clarkson University. The Quarto-based version serves as an example of modern technical publishing and open access education.

License

If you'd like to support the project or contribute, you can report issues or submit pull requests at github.com/clarkson-edge/ee513_book. Thank you for helping improve this resource for the community.

Embedded Systems Design

Chapter 1

Introduction

This chapter introduces the essential concepts of embedded systems and highlights the growing significance of BLE technology in modern designs. It emphasizes the role of microcontrollers, particularly the Silicon Labs EFR32XG24, in enabling efficient, low-power wireless communication for IoT applications. Whether you are a student starting your embedded systems journey or an engineer aiming to enhance your design skills, this booklet will serve as a valuable resource to build innovative and efficient BLE-enabled embedded solutions.

1.1 Overview

Embedded systems are specialized computing systems that are designed to perform dedicated functions or tasks within a larger mechanical or electrical system. Unlike general-purpose computers, embedded systems are optimized for specific applications, balancing constraints such as power consumption, real-time performance, and cost efficiency. They are integral to a wide range of applications, including consumer electronics, automotive systems, medical devices, industrial automation, and smart home technologies.

At the core of most embedded systems lies a microcontroller, a compact integrated circuit that combines a processor, memory, and input/output peripherals on a single chip. Microcontrollers are the brain of embedded systems, executing pre-programmed instructions to manage sensors, actuators, and communication modules. Their efficiency, reliability, and low power consumption make them ideal for embedded applications.

In recent years, the demand for wireless communication in embedded systems has surged, driven by the growth of the Internet of Things (IoT). Among the various wireless protocols, Bluetooth Low Energy (BLE) has emerged as a key technology for low-power, short-range communication. BLE enables devices to transmit small amounts of data with minimal energy consumption, making it ideal for battery-operated applications such as fitness trackers, smart home devices, and health monitoring systems.

The Silicon Labs EFR32XG24 series is one of the most advanced BLE microcontrollers available in Q4 of 2024. Built on the ARM Cortex-M33 core, it offers a powerful blend of performance, energy efficiency, and wireless connectivity. It is equipped with a robust BLE stack, extensive peripherals, and advanced security features, making it a preferred choice for designing sophisticated embedded systems.

This textbook, *System Design with Silicon Lab EFR32XG24 BLE Microcontroller*, is intended to provide a guide for students and engineers to understand and design embedded systems using the EFR32xG24

Dev Kit. The book covers both theoretical concepts and hands-on practical implementations, ensuring readers gain a deep understanding of embedded system design and BLE communication protocols.

Throughout this book, readers will learn:

- The fundamentals of embedded systems and microcontroller architecture.
- Key features and capabilities of the EFR32XG24 BLE microcontroller.
- Practical techniques for programming and debugging embedded systems.
- BLE communication protocols and integration with IoT applications.
- Real-world case studies and projects demonstrating system design principles.

1.2 Real-World Applications of Embedded Systems

Embedded systems are deeply integrated into modern life, serving as the backbone for countless devices and technologies. They are designed to execute dedicated functions efficiently while operating under constraints such as power consumption, memory limitations, and cost. Examples of embedded systems can be observed in both everyday objects and complex industrial applications, showcasing their versatility and importance in modern engineering.

One prominent example is the automotive industry, where embedded systems play a critical role in ensuring safety, efficiency, and advanced functionalities. Anti-lock Braking Systems (ABS) use embedded controllers to regulate brake pressure, preventing skidding on slippery roads and enhancing vehicle stability. Similarly, adaptive cruise control systems utilize embedded microcontrollers to monitor vehicle speed and distance through RADAR or LIDAR sensors, enabling intelligent speed adjustments. Another safety-critical application is the airbag control system, which relies on real-time sensor data to trigger airbag deployment within milliseconds during a collision.

In industrial automation, embedded systems are central to the operation of robotic arms, conveyor belts, and assembly lines. These systems handle precise sequencing, closed-loop control, and real-time signal processing to maintain efficiency and safety. For example, industrial robots are programmed to carry out repetitive tasks such as welding, painting, and packaging, each controlled by embedded microcontrollers to ensure accuracy and reliability.

Consumer electronics also heavily rely on embedded systems. Devices such as programmable engineering calculators, automated teller machines (ATMs), and smart home appliances incorporate microcontrollers to perform specific tasks seamlessly. Modern washing machines, for instance, utilize embedded controllers to monitor water levels, manage wash cycles, and adjust spin speeds dynamically. Similarly, ATMs use embedded microcontrollers to process transactions securely while managing input and output operations.

BLE microcontrollers have further extended the capabilities of embedded systems by enabling low-power wireless communication. BLE technology is particularly advantageous in battery-operated devices like fitness trackers, smart home sensors, and medical monitoring equipment. These microcontrollers facilitate energy-efficient data transmission, allowing devices to remain functional for extended periods without frequent battery replacements.

1.3 Overview of EFR32MG24 Microcontroller

The EFR32MG24 microcontroller, part of Silicon Labs' Wireless Gecko series, is specifically designed to address the growing demand for energy-efficient and high-performance wireless communication in embedded systems. Built on the ARM Cortex-**M33** core, it operates at a maximum frequency of **78** MHz, delivering sufficient computational power for real-time applications while maintaining energy

efficiency. The microcontroller integrates advanced hardware security features, including a hardware cryptographic accelerator and Secure Boot, ensuring robust protection against cyber threats. It supports multiple wireless protocols, with a primary focus on BLE 5.3, enabling reliable, low-power, short-range communication. The integrated radio transceiver offers industry-leading sensitivity and output power, ensuring stable connectivity even in challenging environments.

The EFR32MG24 is equipped with a range of analog and digital peripherals, including ADCs, DACs, timers, UART, SPI, and I2C interfaces, providing flexible options for sensor integration and peripheral control. Its low-power modes, combined with energy-saving peripherals and Sleep Timer capabilities, make it highly suitable for battery-operated devices such as IoT sensor nodes, wearable electronics, and smart home devices. This also features an on-chip AI/ML hardware accelerator, enabling edge-computing capabilities for tasks like sensor data analysis and anomaly detection. Hence, this microcontroller, available in the xG24-DK2601B Development Kit, is chosen for this book.

Chapter 2

Programming Embedded Systems with C

Embedded systems rely heavily on the C programming language due to its efficiency, low-level hardware access, and portability across different microcontroller architectures. This chapter explores the foundational concepts of C programming in the context of embedded systems, with a focus on optimizing performance, managing data types, and effectively using hardware resources. By mastering these fundamental concepts, readers will be better prepared to develop efficient and reliable embedded systems using the Silicon Labs EFR32XG24 BLE microcontroller.

2.1 Simplicity Studio IDE for Silicon Labs EFR32XG24 Microcontroller

Simplicity Studio is the official Integrated Development Environment (IDE) provided by Silicon Labs for embedded development with their microcontrollers, including the EFR32XG24 BLE microcontroller to be covered in this textbook. It is a feature-rich platform designed to streamline the development process, offering a library of example projects, application templates, and related tools for writing, debugging, profiling, and deploying firmware applications efficiently. It integrates multiple tools into one unified interface:

- **Project Management:** Create, organize, and manage embedded projects.
- **Device Configuration:** Configure peripheral modules and optimize hardware settings.
- **Debugging Tools:** Real-time debugging with SEGGER J-Link integration.
- **Energy Profiler:** Monitor and optimize power consumption of embedded applications.
- **Wireless Network Analyzer:** Analyze wireless traffic and optimize communication protocols.

Simplicity Studio supports a range of compilers tailored for embedded systems:

- **GCC (GNU Compiler Collection):** Open-source compiler widely used in embedded systems.
- **IAR Embedded Workbench Compiler:** Commercial compiler known for its optimization capabilities.
- **Keil ARM Compiler (ARMCC):** Industry-standard compiler for ARM Cortex-M series microcontrollers.

For the EFR32XG24 microcontroller, ***GCC*** is the default compiler bundled with Simplicity Studio, offering robust optimization and compatibility with ARM Cortex-M33 cores.

2.1.1 Development Workflow in Simplicity Studio

The typical workflow when using Simplicity Studio for EFR32XG24 development involves:

1. **Device Selection:** Select the target microcontroller (EFR32XG24) from the device catalog.
2. **Project Creation:** Use templates or start from scratch to create firmware projects.
3. **Peripheral Configuration:** Use the graphical configuration tool to set up GPIO, timers, UART, SPI, etc.
4. **Code Generation:** Auto-generate initialization code based on configuration settings.
5. **Build and Compile:** Compile code using GCC or other selected compilers.
6. **Debug and Test:** Use SEGGER J-Link debugger for step-by-step debugging and breakpoint management.
7. **Energy Profiling:** Use the energy profiler to optimize power consumption.

2.1.2 Key Features of Simplicity Studio for EFR32XG24

The **Graphical Peripheral Configuration Tool** provides an intuitive interface for configuring peripherals and pin assignments, reducing setup errors. The **Real-Time Energy Profiler** enables precise monitoring and analysis of energy consumption, helping developers optimize power efficiency. The **Wireless Network Analyzer** facilitates debugging and fine-tuning of Bluetooth communication channels, ensuring reliable wireless connectivity. Additionally, Simplicity Studio includes **SDK Integration** with pre-built libraries and frameworks for BLE and IoT applications. Developers can also leverage an **Extensive Example Codebase**, which contains numerous pre-written projects for rapid prototyping and reduced development time.

To maximize productivity and ensure reliable outcomes, developers should follow established best practices when using Simplicity Studio. It is essential to keep the IDE updated to the latest version to benefit from bug fixes and new features. The graphical configuration tools should be used whenever possible to minimize errors during peripheral setup. Compiler optimizations should be enabled to account for the resource-constrained nature of embedded environments. Regular energy profiling should be conducted throughout firmware development to identify and address power inefficiencies. Lastly, developers should use the **SEGGER J-Link Debugger** for precise, real-time debugging and analysis of embedded applications.

2.2 Structure of an Embedded C Program

A typical embedded C program follows a standardized structure to maintain clarity, modularity, and efficient hardware interaction. A common format that is found in Arduino IDE is as follows:

```
#include <stdint.h>
#define LED_PIN 13

void init();
void loop();

int main() {
    init();
    while (1) {
        loop();
    }
}
```

```
void init() {
    // Initialization code
}

void loop() {
    // Main functionality code
}
```

At the core lies the `main()` function, which serves as the entry point for program execution. The program begins with an `init()` function, responsible for hardware and peripheral initialization, such as configuring GPIO pins, timers, and communication interfaces. Following initialization, the program enters an infinite `while(1)` loop, where the `loop()` function is repeatedly called to handle the system's primary tasks. This structure separates setup and runtime logic, promoting code readability and easier debugging. The use of `#include <stdint.h>` ensures access to fixed-width integer types, while the `#define LED_PIN 13` macro simplifies hardware pin configuration. This modular design allows embedded systems to maintain deterministic behavior.

2.3 Structure of an Embedded C Program in Simplicity Studio

In Simplicity Studio, an embedded C program adheres to a standardized structure designed to ensure modularity, hardware abstraction, and efficient execution on microcontrollers like the EFR32XG24. A typical program format is shown below:

```
#include "em_device.h"
#include "em_chip.h"
#include "em_gpio.h"

#define LED_PIN 13

void init();
void loop();

int main(void) {
    CHIP_Init(); // Initialize the microcontroller system
    init();
    while (1) {
        loop();
    }
}

void init() {
    // GPIO and peripheral initialization code
}

void loop() {
    // Main functionality code
}
```

In Simplicity Studio, the `CHIP_Init()` function is typically called at the beginning of the `main()` function to configure essential hardware components, including the clock management unit (CMU) and device-specific registers. The `init()` function follows, serving to initialize peripherals, configure GPIO pins, and set up timers or communication interfaces. The program then enters an infinite `while(1)` loop, where the `loop()` function repeatedly executes core tasks. Header files such as `em_device.h` provide device-specific definitions, while `em_chip.h` ensures system-level configurations are applied. The use of predefined macros like `#define LED_PIN 13` simplifies hardware abstraction, improving code clarity and reducing errors. This structure leverages Simplicity Studio's hardware abstraction layer (HAL) to provide a consistent programming interface, ensuring scalability and portability across Silicon Labs microcontroller families.

2.4 Generic Data Types in Embedded Systems

Data types in embedded systems are carefully chosen based on performance requirements, memory constraints, and application-specific needs. Common data types include:

2.4.0.1 Integer Data Types (ISO C99 Standard)

- **int:** Standard integer type, typically 16 or 32 bits depending on the microcontroller.
- **uint8_t, uint16_t, uint32_t:** Unsigned integer types offering precise control over memory usage.
- **int8_t, int16_t, int32_t:** Signed integer types for representing both positive and negative values.

2.4.0.2 Floating-Point Data Types (IEEE 754 Standard)

- **float:** 32-bit floating-point type for representing decimal values.
- **double:** 64-bit floating-point type for higher precision.

A summary of these types is displayed in Table 2.1.

The EFR32XG24 microcontroller includes an FPU (Floating-Point Unit) to handle floating-point calculations, but such operations can introduce performance and power efficiency overhead. Therefore, floating-point types should be used sparingly in embedded applications.

Data type	Size	Range min	Range max
int8_t	8 bits (1 byte)	-128	127
uint8_t	8 bits (1 byte)	0	255
int16_t	16 bits (2 bytes)	-32768	32767
uint16_t	16 bits (2 bytes)	0	65535
int32_t	32 bits (4 bytes)	-2,147,483,648	2,147,483,647
uint32_t	32 bits (4 bytes)	0	4,294,967,295
int64_t	64 bits (8 bytes)	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	64 bits (8 bytes)	0	18,446,744,073,709,551,615

Table 2.1: Commonly used integer types when programming embedded systems

2.5 Choosing the Right Data Type

Selecting an appropriate data type in embedded systems programming is a crucial step to ensure optimal memory usage, computational efficiency, and prevention of data-related errors. The choice of data type must balance several key factors, including:

- **Performance:** Larger data types consume more memory and require longer processing times.
- **Overflow:** Variables must be chosen to prevent exceeding their maximum allowable value.
- **Coercion and Truncation:** Automatic type conversion can lead to unintended behavior if not managed carefully.

In embedded systems, memory is a scarce resource, and improper data type selection can lead to unnecessary overhead. For example:

- On an 8-bit microcontroller, using a 4-byte `int` instead of a 1-byte `char` for a simple counter wastes memory and processing cycles. For example,

  ```
  int counter = 0; // Uses 4 bytes unnecessarily on an 8-bit system
  uint8_t counter = 0; // Optimized for an 8-bit system
  ```

- On a 32-bit microcontroller like the ARM Cortex-M, memory access is optimized for 4-byte alignment, and using smaller data types may not yield significant performance improvements.

2.5.0.1 Handling Overflow

Overflow occurs when a variable exceeds the maximum value that can be stored in its data type. In embedded systems, overflow can lead to unpredictable behavior or silent data corruption. For example:

```
uint8_t seconds = 255;
seconds += 1; // Overflow occurs, seconds resets to 0
```

To prevent overflow:

- Use larger data types if overflow is anticipated.
- Implement overflow detection mechanisms.

2.5.0.2 Data Coercion and Truncation

In embedded C, implicit type conversions (coercion) and truncation can lead to unintended results:

- When a smaller data type is promoted to a larger data type (e.g., `uint8_t` to `uint16_t`), padding may occur. For example,

  ```
  uint8_t smallValue = 200;
  uint16_t largeValue = smallValue; // Coercion from 8-bit to 16-bit
  ```

- When a larger data type is truncated to a smaller one (e.g., `uint16_t` to `uint8_t`), significant data loss may occur. For example,

  ```
  uint16_t largeValue = 1025;
  uint8_t smallValue = largeValue; // Truncation, smallValue = 1
  ```

2.5.0.3 Best Practices for Data Type Selection

- Use fixed-width integer types from the `stdint.h` library (`int8_t`, `uint16_t`, etc.).
- Avoid mixing signed and unsigned data types in arithmetic operations.
- Be explicit in type casting and ensure expected results are validated.
- Always check compiler warnings related to type conversions.

2.6 Memory Alignment in Embedded Systems

Efficient memory alignment is critical in embedded systems to optimize performance, reduce access latency, and ensure compatibility with the processor's architecture. In microcontrollers like the EFR32XG24, unaligned memory access can lead to performance penalties or even cause system faults on certain architectures.

2.6.0.1 Understanding Memory Alignment

- **Aligned Access:** Data is stored in memory at addresses that are multiples of its size. For example:
  - A 2-byte `short int` should be stored at an address divisible by 2.
  - A 4-byte `int` or `float` should be stored at an address divisible by 4.
- **Unaligned Access:** Data is stored at an address that does not adhere to its size requirements. For example:
  - Storing a 4-byte `int` at an address like `0x20000003` is considered unaligned.

Aligned access is preferred because microcontrollers fetch data in word-sized chunks (e.g., 4 bytes on ARM Cortex-M processors). Misaligned data may require multiple memory accesses, increasing latency and power consumption.

2.6.0.2 Example of Memory Alignment

Aligned Memory Example (Efficient Access):

```
unsigned char a;       // 1-byte aligned at 0x20000000
unsigned short b;      // 2-byte aligned at 0x20000002
unsigned int c;        // 4-byte aligned at 0x20000004
```

Unaligned Memory Example (Potential Performance Penalty):

```
unsigned char a;       // Stored at 0x20000000
unsigned short b;      // Stored at 0x20000001 (misaligned)
unsigned int c;        // Stored at 0x20000003 (misaligned)
```

Aligned Memory Example (Efficient Access with Padding):

```
unsigned char a;       // Stored at 0x20000000
unsigned char padding; // Added for alignment
unsigned short b;      // Stored at 0x20000002
unsigned int c;        // Stored at 0x20000004
```

2.6.0.3 Best Practices for Memory Alignment

- Use compiler directives or attributes to enforce memory alignment.
- Group variables by size (e.g., group all `char`, then `short`, then `int` variables).
- Avoid unaligned data structures in performance-critical paths.

Compiler Attribute Example (ARM GCC/Keil):

```
struct __attribute__((aligned(4))) AlignedStruct {
    uint8_t a;
    uint16_t b;
    uint32_t c;
};
```

Efficient memory alignment reduces CPU cycles for memory fetches and avoids unnecessary overhead, making it a critical practice in embedded systems programming.

2.6.1 Bitwise Operations

Bitwise operators are essential in embedded systems for manipulating hardware registers and performing efficient computations. These operations are fundamental for efficiently interacting with GPIO (General Purpose Input Output) registers in embedded systems. They allow precise control over individual bits, enabling configuration, status checking, and manipulation of GPIO pins without affecting other bits in the register.

```
AND (&): Used for masking bits.
OR (|): Used for setting bits.
XOR (^): Used for toggling bits.
NOT (~): Used for inverting bits.
```

Example:

```
uint8_t reg = 0b00001111;
reg |= (1 << 4); // Set the 5th bit
reg &= ~(1 << 2); // Clear the 3rd bit
```

2.6.2 Bitwise Shift Operations

Shift operators move bits left (<<) or right (>>) and are commonly used for:

- Multiplying or dividing numbers by powers of two.
- Setting or clearing specific bits.

Example:

```
uint8_t value = 0x01;
value <<= 3; // Shift left by 3 bits (Result: 0x08)
```

2.6.3 Atomic Register Usage for GPIO Control

Atomic GPIO operations are critical in embedded systems where precise and thread-safe pin manipulation is required. Unlike standard `DOUT` register operations (data directly outputs to pin), atomic registers (`SET`, `CLR`, and `TGL`) allow direct modification of specific bits without requiring a read-modify-write cycle.

2.6.3.1 Advantages of Atomic GPIO Operations

- **Thread-Safe:** Prevents unintended side effects in concurrent operations.
- **Efficient:** Eliminates the overhead of read-modify-write cycles.
- **Precision:** Ensures only the target bit is modified.

2.6.3.2 Key Atomic Registers

- **SET Register:** Sets specific GPIO pins to a high state without affecting others.
- **CLR Register:** Clears specific GPIO pins to a low state without affecting others.
- **TGL Register:** Toggles specific GPIO pins without affecting others.

2.6.3.3 Atomic Operations Examples (Explanation in Section 2.6)

1. Setting Multiple Pins Atomically:

```
GPIO->P_SET[gpioPortB].DOUT = (1 << 2) | (1 << 4); // Set pins 2 and 4 on Port B
```

2. Clearing Specific Pins Atomically:

```
GPIO->P_CLR[gpioPortC].DOUT = (1 << 5); // Clear pin 5 on Port C
```

3. Toggling Multiple Pins Atomically:

```
GPIO->P_TGL[gpioPortD].DOUT = (1 << 1) | (1 << 3); // Toggle pins 1 and 3 on Port D
```

2.6.3.4 Avoiding Race Conditions in GPIO Control

In real-time systems, race conditions can occur when multiple threads or interrupt routines attempt to modify GPIO pins simultaneously. Atomic registers mitigate this risk by ensuring:

- Only the targeted pins are modified.
- No unintended overwrites occur during concurrent access.

Example of Thread-Safe Pin Toggle:

```
void toggleLedThreadSafe(void) {
    GPIO->P_TGL[gpioPortA].DOUT = (1 << 6); // Safely toggle pin 6
}
```

2.6.3.5 Best Practices for Using Atomic Registers

- Prefer atomic registers for time-critical pin operations.
- Avoid mixing standard `DOUT` operations with atomic operations on the same pins.
- Document atomic operations in shared resources clearly.
- Test interrupt-driven routines for predictable behavior with atomic GPIO controls.

2.6.3.6 Checking the State of a GPIO Pin:

```
uint8_t pinState = (GPIO->P[gpioPortA].DIN >> 3) & 1; // Read state of pin 3
```

2.6.3.7 Using Shift Operators for Pin Masking

Shift operators are commonly used to create masks for setting, clearing, or toggling specific bits in GPIO registers.

Example - Setting Multiple Pins:

```
GPIO->P[gpioPortA].DOUT |= (1 << 3) | (1 << 5); // Set pins 3 and 5
```

Example - Clearing Multiple Pins:

```
GPIO->P[gpioPortA].DOUT &= ~((1 << 3) | (1 << 5)); // Clear pins 3 and 5
```

2.6.3.8 Practical Example: Blinking an LED Using Bitwise Operations

The following example demonstrates how to blink an LED connected to Port D, Pin 2 using bitwise operations:

```
#define LED_PIN 2

// Configure pin as output
GPIO->P[gpioPortD].MODEL |= (1 << (4 * LED_PIN)); // MODEL is the MODE Low register

// Toggle the LED state in a loop
while (1) {
    GPIO->P[gpioPortD].DOUT ^= (1 << LED_PIN); // Toggle LED pin
    delay(1000); // 1-second delay
}
```

2.6.3.9 Best Practices for Bitwise GPIO Operations

- Always mask the specific bits you intend to modify.
- Avoid direct assignments to GPIO registers; prefer bitwise operations.
- Use clear and descriptive macros for pin numbers and masks.
- Test configurations thoroughly to prevent accidental overwrites.

Bitwise operations provide low-level control over GPIO registers, ensuring efficient and predictable manipulation of hardware pins. Mastering these operations is essential for embedded systems programming.

2.7 Understanding the -> Operator in Atomic GPIO Operations

In embedded systems programming, especially when interfacing with hardware peripherals such as GPIO registers, it is common to encounter expressions utilizing the -> operator. This operator is used to access members of a structure through a pointer. In the context of atomic GPIO operations with the Silicon Labs EFR32XG24 microcontroller, the -> operator simplifies hardware register access and enhance code clarity.

2.7.1 Pointer to Structure and the -> Operator

In C, the -> operator is used to access a member of a structure when the structure is referred to by a pointer. The syntax is:

```
pointer->member
```

This is equivalent to:

```
(*pointer).member
```

Here:

- **pointer**: Points to a structure (e.g., GPIO peripheral base address).
- **member**: Represents a specific field in the structure (e.g., registers like `P_SET`, `P_CLR`, `P_TGL`).

2.7.2 GPIO Structure and Enums in EFR32XG24

The GPIO peripheral on the EFR32XG24 microcontroller is represented as a structure, typically defined in the hardware abstraction layer (HAL). For example:

```
typedef struct {
    volatile uint32_t DOUT;
    volatile uint32_t SET;
    volatile uint32_t CLR;
    volatile uint32_t TGL;
} GPIO_Port_TypeDef;
```

Additionally, GPIO ports are often enumerated for easy reference:

```
typedef enum {
    gpioPortA,
    gpioPortB,
    gpioPortC,
    gpioPortD
} GPIO_Port_Type;
```

2.7.3 Atomic GPIO Operations with ->

When performing atomic GPIO operations, the structure pointer enables access to specific GPIO port registers. For example:

```
GPIO->P_SET[gpioPortB].DOUT = (1 << 2) | (1 << 4);
```

Explanation:

- **GPIO**: Base pointer to the GPIO peripheral structure.
- **P_SET**: Array of registers representing SET operations for each port.
- **gpioPortB**: Index to select Port B.
- **DOUT**: Data Output register for atomic SET operation.

Similarly:

```
GPIO->P_CLR[gpioPortC].DOUT = (1 << 5); // Clear pin 5 on Port C
GPIO->P_TGL[gpioPortD].DOUT = (1 << 1) | (1 << 3); // Toggle pins 1 and 3 on Port D
```

2.7.4 Advantages of Using Structures and the -> Operator

- **Code Clarity**: Clear and readable syntax for hardware register access.
- **Portability**: Standardized structure definitions across different microcontrollers.
- **Efficiency**: Direct register access through pointer dereferencing minimizes CPU cycles.

2.8 Exercise: Multiple Choice Questions

1. **What is the primary reason C is preferred for embedded systems programming?**
    1. User-friendly syntax
    2. High-level abstraction
    3. Low-level hardware access and efficiency
    4. Automatic memory management

    Correct Answer: C
2. **Which header file is commonly included in an embedded C program for fixed-width integer types?**
    1. `stdio.h`
    2. `stdint.h`
    3. `string.h`
    4. `stdlib.h`

    Correct Answer: B
3. **What happens if a variable exceeds the maximum value of its data type?**
    1. It goes back to zero
    2. It causes a system crash
    3. It triggers an interrupt
    4. It generates a compiler warning

    Correct Answer: A
4. **Which data type is best suited for a counter variable on an 8-bit microcontroller?**
    1. `int`
    2. `uint8_t`
    3. `float`
    4. `double`

    Correct Answer: B
5. **What is the key advantage of using floating-point data types sparingly in embedded systems?**
    1. Reduced memory usage
    2. Increased processing speed

3. Better precision
4. Simplified syntax

Correct Answer: A

6. **Which of the following is an example of memory alignment in embedded systems?**
    1. Address divisible by 2 for a short integer
    2. Randomly allocated memory address
    3. Using dynamic memory allocation
    4. Overwriting stack memory

    Correct Answer: A
7. **What does the bitwise 'AND' operator do in GPIO manipulation?**
    1. Sets specific bits
    2. Clears specific bits
    3. Masks specific bits
    4. Toggles specific bits

    Correct Answer: C
8. **What is the purpose of the 'SET' register in GPIO control?**
    1. Clear specific GPIO pins
    2. Toggle specific GPIO pins
    3. Set specific GPIO pins
    4. Read GPIO pin status

    Correct Answer: C
9. **Which best describes data coercion in embedded systems?**
    1. Automatic type conversion
    2. Forced memory alignment
    3. Manual data truncation
    4. Dynamic memory reallocation

    Correct Answer: A
10. **What happens when a uint16_t variable is assigned to a uint8_t variable with a value greater than 255?**
    1. Value remains unchanged
    2. Compiler error
    3. Value is truncated
    4. System crash

    Correct Answer: C
11. **Which of the following prevents race conditions in GPIO control?**
    1. Using 'DOUT' register
    2. Using 'SET' and 'CLR' registers atomically
    3. Disabling interrupts

4. Using global variables

Correct Answer: B

12. **Why is aligned memory access preferred in embedded systems?**
    1. Better energy efficiency
    2. Increased memory usage
    3. Reduced CPU latency
    4. Dynamic memory allocation

    Correct Answer: C
13. **What is the main function of bitwise shift operators (‘«’ and ‘»’) in embedded C?**
    1. Inverting bits
    2. Multiplying or dividing by powers of two
    3. Clearing specific bits
    4. Reading GPIO pin status

    Correct Answer: B
14. **What should you avoid when working with GPIO atomic operations?**
    1. Mixing standard ‘DOUT’ and atomic operations
    2. Using specific masks
    3. Documenting shared resources
    4. Testing configurations

    Correct Answer: A
15. **Which fixed-width integer type ensures consistent size across platforms?**
    1. `int`
    2. `long`
    3. `uint16_t`
    4. `short`

    Correct Answer: C

Chapter 3

EFR32xG24 Development Kit Overview

The EFR32xG24 Development Kit provides a robust platform for developing energy-efficient IoT applications. With built-in debugging tools, versatile sensors, and a flexible power supply system, it is well-suited for both prototyping and production-grade development. Mastery of the development tools and peripherals ensures efficient and scalable application design.

3.1 Key Features of the EFR32xG24 Development Kit

The EFR32xG24 Development Kit (xG24-DK2601B) is a versatile platform designed for prototyping and evaluating applications using the EFR32MG24 Wireless Gecko System-on-Chip (SoC), EFR32MG24B310F1536IM48-B. It serves as an ideal platform for developing energy-efficient IoT devices, offering advanced hardware features, debugging capabilities, and seamless integration with development tools such as Simplicity Studio. It also contains a built-in AI/ML Hardware Accelerator.

The key components and features of this kit include:

- **EFR32MG24 Wireless Gecko SoC:** ARM Cortex-M33 processor operating at 78 MHz, with 1536 KB Flash and 256 KB RAM.
- **Connectivity:** High-performance 2.4 GHz radio for Bluetooth and other wireless protocols.
- **On-Board Sensors:**
  - Si7021 Relative Humidity and Temperature Sensor.
  - Si7210 Hall Effect Sensor.
  - ICS-43434 MEMS Stereo Microphones.
  - ICM-20689 6-Axis Inertial Sensor.
  - VEML6035 Ambient Light Sensor.
  - BMP384 Barometric Pressure Sensor.
- **Memory:** 32 Mbit external SPI flash for Over-The-Air (OTA) firmware updates and data logging.
- **Power Options:** USB, coin cell battery (CR2032), or external battery.
- **Debugging Tools:**
  - SEGGER J-Link On-Board Debugger.
  - Packet Trace Interface (PTI).

  - Mini Simplicity Connector for advanced debugging.

- **User Interface:** Two push buttons, an RGB LED, and a virtual COM port.
- **Connectivity Interfaces:** I2C, SPI, UART, and Qwiic Connector.

3.1.1 Development Environment and Tools

The development kit is fully supported by Silicon Labs' Simplicity Studio, an integrated development environment (IDE) offering:

- Project creation and device configuration.
- Real-time energy profiling and debugging tools.
- Wireless network analysis with Packet Trace Interface (PTI).
- Pre-built example projects and libraries for rapid prototyping.

3.1.2 Power Management

The kit offers flexible power options, including:

- USB power supply through a Micro-B connector.
- Coin cell battery (CR2032) for portable applications.
- External battery via a dedicated header.
- Automatic power source switchover for seamless transitions.

Example Configuration for USB Power Supply:

```
Power supplied via USB Micro-B connector:
- VBUS regulated to 3.3V for SoC and peripherals.
- Automatic switchover when USB is connected.
```

3.1.3 Debugging and Virtual COM Port

The built-in SEGGER J-Link debugger allows:

- On-chip debugging via Serial Wire Debug (SWD) interface.
- Real-time packet trace using Packet Trace Interface (PTI).
- Serial communication using Virtual COM Port (VCOM).

Example UART Configuration for VCOM:

```
Baud rate: 115200 bps
Data bits: 8
Parity: None
Stop bits: 1
```

3.1.4 GPIO and Peripheral Access

The development kit provides 20 breakout pads, exposing GPIO pins, I2C, UART, and SPI interfaces. These pads follow the EXP header pinout standard, ensuring compatibility with expansion boards. Each sensor is optimized for low power consumption.

3.1.5 Best Practices for Overall Project Development

- Use Simplicity Studio for project management and debugging.
- Enable only necessary peripherals to conserve power.
- Use GPIO atomic operations for time-critical applications.
- Validate sensor connections using test scripts.

3.2 Sensors and Interfaces

The EFR32xG24 Development Kit integrates multiple onboard sensors interfaced through GPIO, I2C, or SPI connections, ensuring precise communication and control.

3.2.0.1 Si7021 Relative Humidity and Temperature Sensor

The Si7021 is a high-precision digital humidity and temperature sensor featuring a factory-calibrated output and low power consumption, making it suitable for IoT and embedded applications.

Key Features:

- Relative humidity accuracy: ±3%
- Temperature accuracy: ±0.4°C
- Operating voltage: 1.9V to 3.6V
- Ultra-low standby current: 60 nA

Applications:

- Environmental monitoring systems
- HVAC control
- Smart home automation

The sensor is connected through I2C, and its thermal isolation reduces self-heating effects, ensuring more accurate temperature readings.

3.2.0.2 Si7210 Hall Effect Sensor

The Si7210 is a highly sensitive Hall effect sensor capable of detecting magnetic field changes with excellent precision. It is often used in applications requiring contactless position sensing.

Key Features:

- Magnetic sensitivity: ±2.5 mT
- I2C communication interface
- Programmable magnetic thresholds
- Factory-calibrated accuracy

Applications:

- Proximity sensing
- Position detection
- Reed switch replacement

The Si7210 offers real-time magnetic field measurements and is configured via the I2C bus.

3.2.0.3 ICS-43434 MEMS Stereo Microphones

The ICS-43434 microphones are omnidirectional MEMS microphones with I2S digital output. They are suitable for audio signal processing and voice recognition systems.

Key Features:

- Frequency response: 50 Hz – 20 kHz
- Digital I2S output
- Low power consumption
- High Signal-to-Noise Ratio (SNR)

Applications:

- Voice recognition systems
- Acoustic event detection
- Environmental noise monitoring

The microphones are mounted on the bottom side of the development board, with sound pathways designed for optimal acoustic performance.

3.2.0.4 ICM-20689 6-Axis Inertial Sensor

The ICM-20689 integrates a 3-axis gyroscope and a 3-axis accelerometer for precise motion and orientation tracking.

Key Features:

- 3-axis gyroscope and 3-axis accelerometer
- Programmable digital filters
- Integrated 16-bit ADC
- SPI interface for high-speed communication

Applications:

- Motion detection systems
- Gesture-based controls
- Orientation tracking

The sensor is positioned near the geometrical center of the board, minimizing measurement bias caused by physical placement.

3.2.0.5 VEML6035 Ambient Light Sensor

The VEML6035 is a high-precision ambient light sensor that supports a digital I2C interface. It is designed for automatic brightness control and energy-saving applications.

Key Features:

- Wide dynamic range
- Low power consumption
- High accuracy
- I2C communication

Applications:

- Display backlight adjustment

- Smart lighting systems
- Proximity detection

The sensor is factory-calibrated for optimal accuracy and sensitivity across a wide range of light intensities.

3.2.0.6 BMP384 Barometric Pressure Sensor

The BMP384 is a high-precision absolute barometric pressure sensor with an integrated temperature sensor suitable for environmental monitoring and altitude estimation.

Key Features:

- Pressure accuracy: ±0.5 hPa
- Temperature accuracy: ±0.5°C
- I2C/SPI communication interface
- Integrated noise reduction filter

Applications:

- Weather station systems
- Altitude estimation
- Drone stabilization systems

The BMP384 sensor uses an internal noise-reduction filter to improve data accuracy during high-resolution measurements.

3.2.0.7 Best Practices for Sensor Integration

To ensure optimal performance when working with the onboard sensors:

- Always enable sensor power through the appropriate GPIO pins before initialization.
- Avoid floating GPIO lines connected to sensors.
- Validate sensor connections and configurations using test scripts.
- Minimize concurrent access to shared I2C or SPI lines.

Chapter 4

EFR32 I/O Programming

Embedded systems rely heavily on input/output (I/O) operations to interact with external devices such as sensors, actuators, and communication modules. The EFR32XG24 microcontroller provides versatile I/O functionalities through its General Purpose Input/Output (GPIO) pins, enabling efficient communication with hardware peripherals. Understanding GPIO configuration and control is crucial for efficient interaction with hardware peripherals. This chapter equips readers with the knowledge to configure, control, and monitor GPIO pins effectively using both register-level programming and emlib functions.

4.1 EFR32XG24 GPIO Overview

As displayed in Figure 4.1, the EFR32XG24 microcontroller series includes multiple GPIO ports (A, B, C, and D), with each port supporting up to 16 pins. The key GPIO ports and pins that are available on the specific chip used in the EFR32XG24 Dev Kit are:

- PA00-PA09
- PB00-PB05
- PC00-PC09
- PD00-PD05

Each GPIO pin can be individually configured for various modes, including input, output, and alternate functions.

Note that on the EFR32XG24 Dev Kit, only some pins are *broken out*, that is, available for use via the expansion headers on the left and right sides of the board. These pins on the expansion header, which may be found in the EFR32XG24 Dev Kit User Guide on page 19 are displayed in Figure 4.2.

4.1.1 Clock Management Unit (CMU)

The Clock Management Unit (CMU) controls the clock signals for various peripherals, including GPIO. Before using GPIO, its corresponding clock must be enabled by setting the appropriate bit in the `CMU_CLKEN0` register:

```
CMU->CLKEN0 |= 1 << 26;
```

This ensures that the GPIO module is both powered up and ready for use with a clock connected.

The `CMU_CLKEN0` register also allows activating the clock to other peripherals through the use of the same code as above. The only required change is the bit to modify, by changing the number of bits to shift the `1` left to one of the options shown in Figure 4.3.

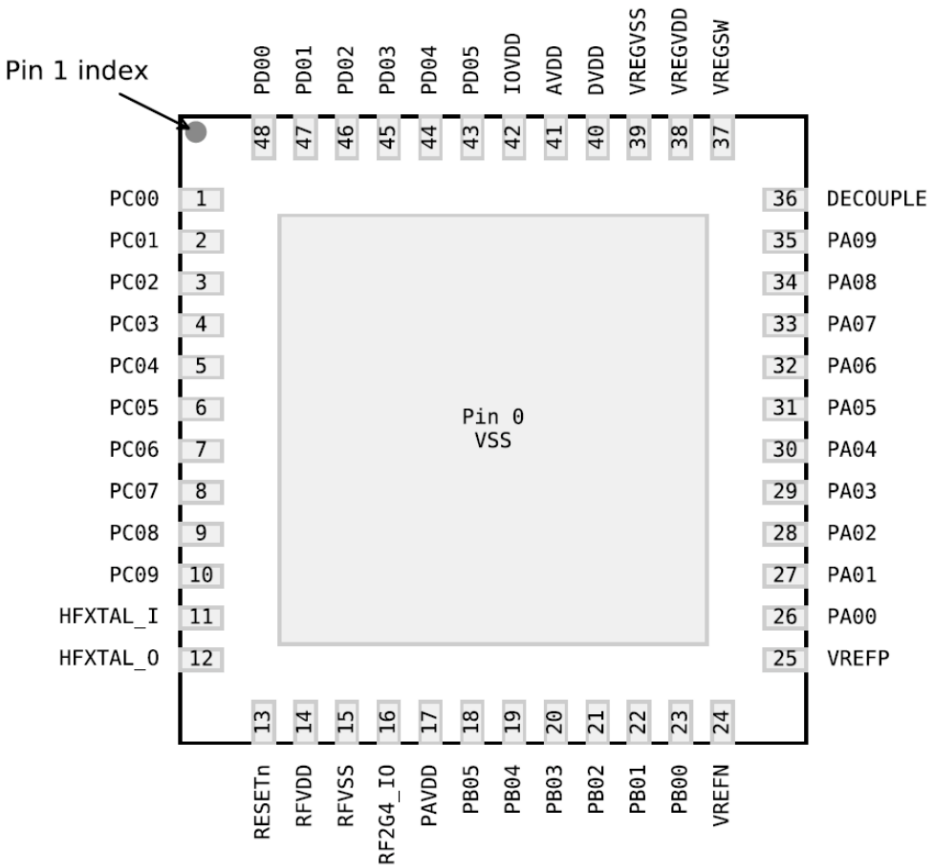

Figure 4.1: Figure 4.1: Pinout of QFN-48 packaged EFR32MG24 microcontroller (EFR32MG24 Datasheet page 107)

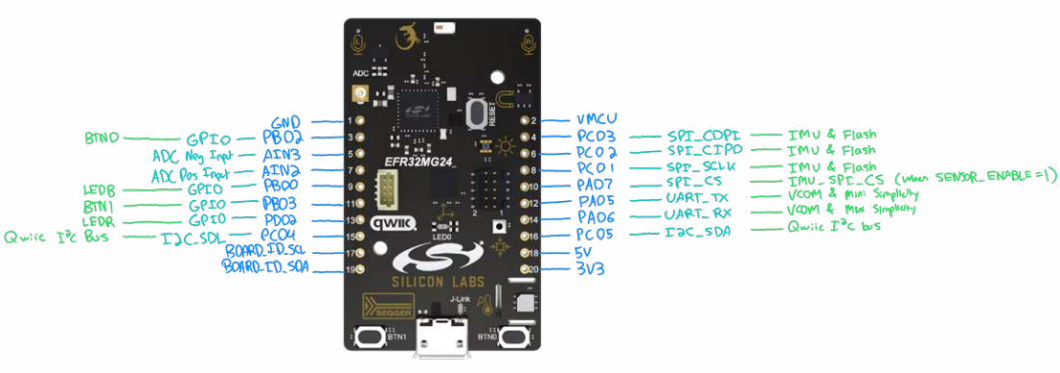

Figure 4.2: Figure 4.2: EFR32XG24 Dev Kit Expansion Header Pinout (UG524 page 19)

8.5.10 CMU_CLKEN0 - Clock Enable Register 0

Offset	Bit Position																													
0x064	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0x0	0x0	0x0	0x0	0x0	0x0		0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
Access	RW	RW	RW	RW	RW	RW		RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Name	DCDC	SYSRTC0	BURTC	BURAM	PRS	GPIO		ULFRCO	LFXO	LFRCO	FSRCO	HFXO0	HFRCOEM23	HFRCO0	DPLL0	SYSCFG	I2C1	I2C0	WDOG0	LETIMER0	AMUXCP0	IADC0	USART0	TIMER4	TIMER3	TIMER2	TIMER1	TIMER0	GPCRC	RADIOAES	LDMAXBAR	LDMA

Figure 4.3: Figure 4.3: Peripherals available to enable in the CMU_CLKEN0 register (Reference manual page 173)

GPIO_PORTA_MODEH

Offset	Bit Position																													
0x03C	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																									0x0				0x0			
Access																									RW				RW			
Name																									MODE1				MODE0			

GPIO_PORTA_MODEL

Offset	Bit Position																													
0x034	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0x0				0x0				0x0				0x0				0x0				0x0				0x0				0x0			
Access	RW				RW				RW				RW				RW				RW				RW				RW			
Name	MODE7				MODE6				MODE5				MODE4				MODE3				MODE2				MODE1				MODE0			

Figure 4.4: Figure 4.4: The MODEL and MODEH registers for EFR32MG24 GPIO Port A (Reference manual page 851)

4.1.2 GPIO Configuration

Each GPIO pin can serve multiple functions controlled by the MODEL and MODEH registers:

- **MODEL:** Configures pins 0-7 of the port.
- **MODEH:** Configures pins 8-15 of the port.

Each pin mode is represented by 4 bits, supporting modes such as:

- 0: Disabled
- 1: Input
- 2: Input pull-up/down
- 4: Push-pull (Output)

In pull-up/pull-down mode, the value of the `DOUT` register (covered later) determines the pull direction, with a `1` being pull-up and `0` being pull-down.

To set a pin mode programmatically:

```
GPIO->P[gpioPortX].MODEL |= mode << (4 * n);
```

For pin numbers 8-15:

```
GPIO->P[gpioPortX].MODEH |= mode << (4 * (n - 8));
```

A total of 16 pin modes are available for each GPIO pin. While many of these modes are not used in basic applications, Figure 4.5 displays all of the available options.

MODE n

Value	Mode	Description
0	DISABLED	Input disabled. Pullup if DOUT is set.
1	INPUT	Input enabled. Filter if DOUT is set.
2	INPUTPULL	Input enabled. DOUT determines pull direction.
3	INPUTPULLFILTER	Input enabled with filter. DOUT determines pull direction.
4	PUSHPULL	Push-pull output.
5	PUSHPULLALT	Push-pull using alternate control.
6	WIREDOR	Wired-or output.
7	WIREDORPULLDOWN	Wired-or output with pull-down.
8	WIREDAND	Open-drain output.
9	WIREDANDFILTER	Open-drain output with filter.
10	WIREDANDPULLUP	Open-drain output with pullup.
11	WIREDANDPULLUPFIL-TER	Open-drain output with filter and pullup.
12	WIREDANDALT	Open-drain output using alternate control.
13	WIREDANDALTFILTER	Open-drain output using alternate control with filter.
14	WIREDANDALTPULLUP	Open-drain output using alternate control with pullup.
15	WIREDANDALTPULLUP-FILTER	Open-drain output using alternate control with filter and pullup.

Figure 4.5: Figure 4.5: Mode register value options for each GPIO pin (Reference manual page 851)

4.1.3 GPIO Output Control

GPIO output can be managed using the following registers:

- **DOUT:** Directly outputs data to pins.
- **SET:** Atomically sets specified bits.
- **CLR:** Atomically clears specified bits.
- **TGL:** Atomically toggles specified bits.

Example of setting and clearing pins:

```
GPIO->P[gpioPortD].DOUT |= 1 << 2; // Set pin 2 of Port D
GPIO->P[gpioPortD].DOUT &= ~(1 << 2); // Clear pin 2 of Port D
```

4.1.4 GPIO Input Control

GPIO pins configured as inputs can be read using the DIN register:

```
uint8_t pinState = (GPIO->P[gpioPortX].DIN >> n) & 1;
```

This reads the state of pin `n` and returns either 0 or 1.

4.1.5 Using emlib for GPIO

EFR32 provides the **emlib** hardware abstraction layer (HAL) for GPIO configuration:

- GPIO_PinModeSet(port, pin, mode)
- GPIO_PinOutSet(port, pin)
- GPIO_PinOutClear(port, pin)
- GPIO_PinOutToggle(port, pin)
- GPIO_PinInGet(port, pin)

Example of setting a pin as output using emlib:

```
GPIO_PinModeSet(gpioPortD, 2, gpioModePushPull); // Set the Push Pull Mode of Pin 2 of Port D
GPIO_PinOutSet(gpioPortD, 2); // Set pin 2 of Port D
```

4.1.6 Practical Example: Blinking an LED

A simple example of GPIO programming is blinking an LED connected to a GPIO pin:

```
GPIO_PinModeSet(gpioPortD, 2, gpioModePushPull);
while (1) {
    GPIO_PinOutToggle(gpioPortD, 2);
    delay(1000);
}
```

This toggles the LED state every second.

Chapter 5

Applications of EFR32 I/O

Now that the basics of EFR32XG24 microcontroller input/output (I/O) are understood, the next step is learning how to add additional functionality to embedded systems using more complex hardware. This chapter will prepare readers to interact with more complex off-chip hardware using the microcontroller's GPIO pins, and is an important step towards building user-friendly and effective embedded systems.

5.1 7-Segment Displays

7-segment displays are commonly found in user-facing embedded systems, such as clock radios, household appliances, vehicles, and industrial equipment. While LED status indicators are often used for simple devices, they cannot communicate detailed information such as sensor readings or error codes. Gaining traction in the 1970s with the advent of LED technology, 7-segment displays bridge the gap between basic indicator lights and more complex graphic screens, commonly offering one or multiple digits composed of seven LED digit segments plus a decimal point or colon.

5.1.1 Segments

7-segment displays are composed of a group of LED segments arranged in an "8" pattern, allowing every digit from 0-9 plus a limited selection of letters to be readable.

These segments are commonly labeled A-G in a clockwise manner, with A being the top segment and G being the middle segment. Depending on the display, the segments may be wired in a common anode (LED positive terminal) or common cathode (LED negative terminal) configuration. Depending on the configuration, a slightly different circuit with inverted code logic may be necessary.

Additionally, as each segment is a simple LED, current-limiting resistors are a necessary inclusion in the circuit. In some cases, it may be acceptable to place a single resistor between in series with the common pin, especially if the resistor is of a high value to significantly limit the segment's brightness. However, in most cases, it is ideal to adhere to the best practice of placing a current-limiting resistor in series with each *segment* so that manufacturing discrepancies between segments do not allow any individual segment to endanger itself with a high current.

5.1.2 Wiring

A 7-segment display will allocate a significant number of pins on a microcontroller, often using up nearly an entire GPIO port. If the decimal point is not used, one pin may be saved, but in many cases,

it is beneficial to use a BCD to 7-segment decoder IC, such as the 74LS147, or even an 8-bit serial-in, parallel-out shift register such as the 74HC595 for greater GPIO pin efficiency. However, for the purposes of this guide, the 7-segment display will be directly connected to the microcontroller, using 8 GPIO pins.

In an ideal design, such as when building a PCB carrier board for the EFR32MG24 chip, a bank of pins such as PC00-PC07 may be used, allowing the GPIO port C MODEL register to be written in its entirety and full bytes written to the pin set and clear ports.

However, the EFR32XG24 Dev Kit board does not break out a single port in its entirety, therefore requiring the display to share pins between Port A and Port C. Segments A-E will use PC01-PC05, while F, G, and the decimal point (DP) will be connected to PA05-PA07, as displayed in Figure 5.1. This requires in code an array of GPIO ports and pins to look up the right one for a given segment:

```
// use gpioPortC (2) pins 1-5 for A-E, and gpioPortA (0) pins 5-7 for F, G, and DP
//                              A  B  C  D  E  F  G  .
const uint8_t segment_ports[] = {2, 2, 2, 2, 2, 0, 0, 0};
const uint8_t segment_pins[] =  {1, 2, 3, 4, 5, 5, 6, 7};
```

5.1.3 Digit display logic

With these arrays created, the pattern of segments to enable for any character can now be defined. As there are eight segments in total, it makes sense to represent these patterns as bits in a byte, allowing for straightforward storage and lookup. To construct this byte, one may represent segment A as bit 0, B as bit 1, and so on until DP is bit 7. This results in Table 5.1, displaying the construction of hexadecimal codes for digits 0-9.

Digit	bit 7 .	bit 6 G	bit 5 F	bit 4 E	bit 3 D	bit 2 C	bit 1 B	bit 0 A	Hexadecimal
0	0	0	1	1	1	1	1	1	**0x3F**
1	0	0	0	0	0	1	1	0	**0x06**
2	0	1	0	1	1	0	1	1	**0x5B**
3	0	1	0	0	1	1	1	1	**0x4F**
4	0	1	1	0	0	1	1	0	**0x66**
5	0	1	1	0	1	1	0	1	**0x6D**
6	0	1	1	1	1	1	0	1	**0x7D**
7	0	0	0	0	0	1	1	1	**0x07**
8	0	1	1	1	1	1	1	1	**0x7F**
9	0	1	1	0	1	1	1	1	**0x6F**

Table 5.1: Lookup table for 7-segment display digit codes

These hexadecimal codes for each digit can then be inserted into another array, with the array index mapping a desired digit to its segment code. In a later exercise, you will be required to expand this array to support hexadecimal digits as well.

5.1.4 Display driver code

Now that this look-up array for digits is implemented, driving the 7-segment display is trivial. Each GPIO pin in use must be set up as an output. Each pin may then be looped through, and set or cleared depending on if its corresponding bit in the hexadecimal code is set. This can be achieved by shifting the hexadecimal code right by the loop iterator variable, then evaluating based on the bitwise AND of the shifted code and 1.

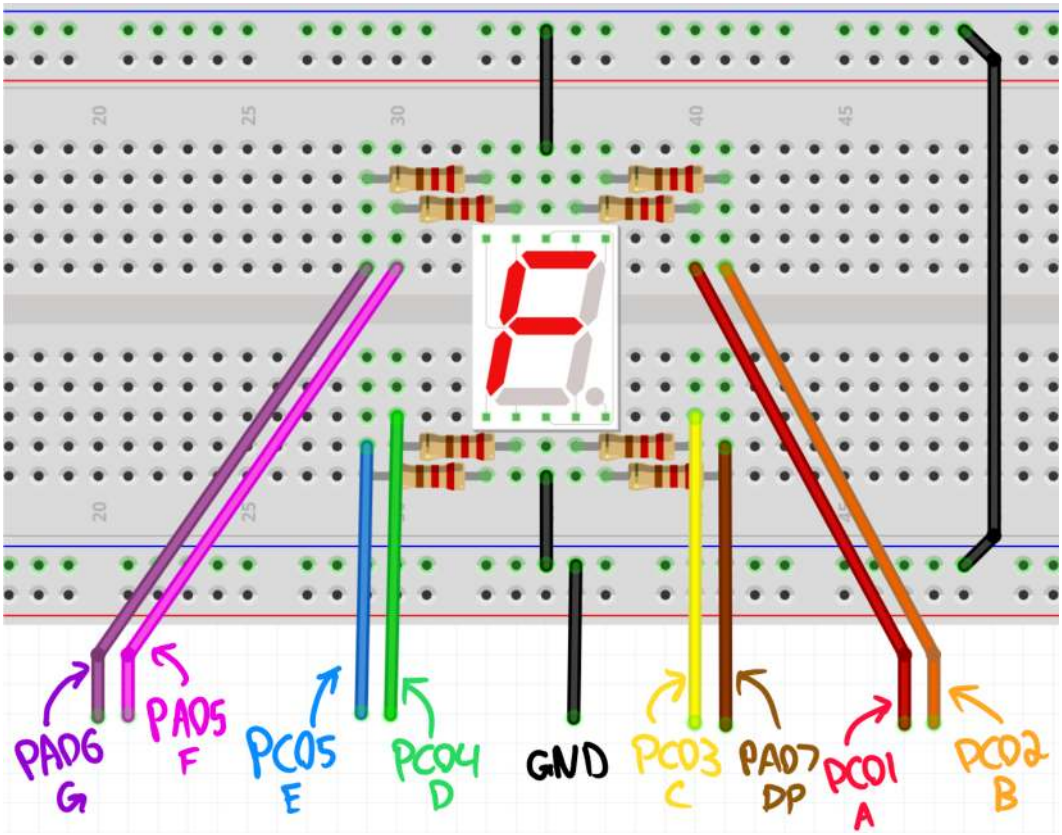

Figure 5.1: Figure 5.1: Wiring diagram for a single common cathode 7-segment display

```
for (int i = 0; i < 8; i++) // loop through all segments
{
    if ((segments[arbitrary_digit] >> i) & 1) // look up hex code, shift right, AND
    {
        GPIO->P_SET[segment_ports[i]].DOUT = 1 << segment_pins[i]; // turn on segment
    }
    else
    {
        GPIO->P_CLR[segment_ports[i]].DOUT = 1 << segment_pins[i]; // turn off segment
    }
}
```

In this example, we use the `segments` array to loop up the hex code for a given `arbitrary_digit`, which could be an integer literal or a variable. The looked up code is shifted right based on the index of the current digit to be illuminated or turned off. With the bit of interest now moved to bit 0, it is compared with 1 to determine the appropriate state for the segment.

5.1.5 Multiple digits

In many applications, multiple 7-segment digits are necessary to display a larger number or other more detailed information. Even displaying time in a 12- or 24-hour format requires four digits. Therefore, it is often beneficial to combine multiple 7-segment displays into a single module, and these are commonly available in 2, 4, 6, or 8 digit configurations. However, using 8 GPIO pins per segment can quickly waste all available microcontroller pins. Instead, all digits in a module *multiplex*, or share, segment pins, which means that all segments, if illuminated at the same time, would show the same character. To facilitate this multiplexing, each digit has its own separate common cathode or common anode pin, which can be connected or disconnected to power. Each digit may then be lit one after another, with only one on at a given time, and this process is constantly repeated to create the effect that all digits are constantly on.

This does necessitate the use of an NPN transistor for each digit to switch the common pin load of the digit on and off, as the microcontroller cannot sink this significant current into a single GPIO. An example circuit is included in Figure 5.2, demonstrating the connections of a two-digit module with a common cathode configuration to an MCU.

5.1.6 Multiple digits logic

The same code may be used for driving multiple digits as a single digit; after all, the only difference is that the lit digit must be changed repeatedly. It is therefore ideal to move the single digit driver code to its own function so that it may be reused. The infinite loop must be adjusted to drive each digit's transistor base pin high, then display a given digit, and finally switch the transistor back off before repeating the process for the next digit. This must be done quickly to avoid flicker at a frequency that is visible to the human eye, but not so quickly that the LEDs in each segment do not have time to reach their full brightness. Therefore, a few milliseconds of delay may be necessary while each digit is on before quickly switching to the next digit.

5.1.7 Exercise: Displaying hexadecimal numbers

Complete Table 5.1 with the additional hexadecimal digits A-F, and expand the array. Then, try displaying 8-bit numbers in hexadecimal format with a two-digit 7-segment display module.

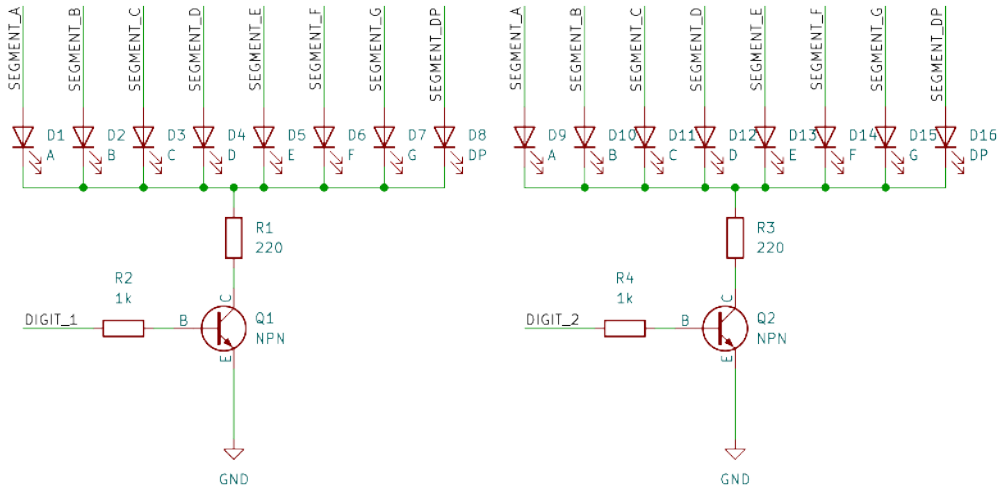

Figure 5.2: Figure 5.2: Two common-cathode 7-segment display digits switches by transistors

5.2 Parallel LCD displays

When more letters or symbols must be displayed to a user than is practical with a simple 7-segment display, or a graphical user interface is it is common to use a screen. Early computers generated signals to drive CRT screens, with a limited number of display lines and colors. Now, while full-color, high resolution monitors and other displays are widespread, it is still common to find smaller, often monochrome screens used in embedded systems due to their simplicity and minimal power consumption. In this section, we will learn about interfacing with a liquid crystal display (LCD) screen that can display two lines of 5x8 pixel characters. These low-cost displays are commonly found on budget 3D printers, control units for machinery, or in vehicle entertainment systems.

5.2.1 16x2 Character LCD

An inexpensive character-based LCD module often contains 2-4 rows of 8-20 characters. In this case, the common LCD1602 module with 2 rows of 16 characters will be used. At the top of the display is a row of pins for powering and controlling the display. Table 5.2 displays details for each pin, but most commonly found on these displays are power and ground for the display, a separate anode and cathode for the backlight LED, a contrast adjustment, and a number of data and control signals. To understand how to interface with the LCD, we must examine the display's built in controller.

5.2.2 LCD Controller

The LCD module has an on-board Hitachi HD44780U controller that generates signals for the individual pixels of the display. The HD44780U is based on an original 1980s design, retaining command and feature parity while supporting modern microcontroller interfaces. It has two host-facing I/O registers as well as internal memory, meaning that data written to the display remains until it is next updated, reducing host MCU processor load. A 4- and 8-bit interface allows writing to, or reading

from, both the instruction register or data register, which are used for configuration and character output, respectively. Therefore, these displays are known as using a parallel interface, as multiple bits of data are transferred at the same instant. More advanced displays may also offer additional interfaces that we will learn about later, such as I²C or SPI. It is important to read and understand the datasheet for the LCD controller to learn how to interface with it. The datasheet is linked here, but excerpts will be taken from it in this section: https://www.sparkfun.com/datasheets/LCD/HD44780.pdf

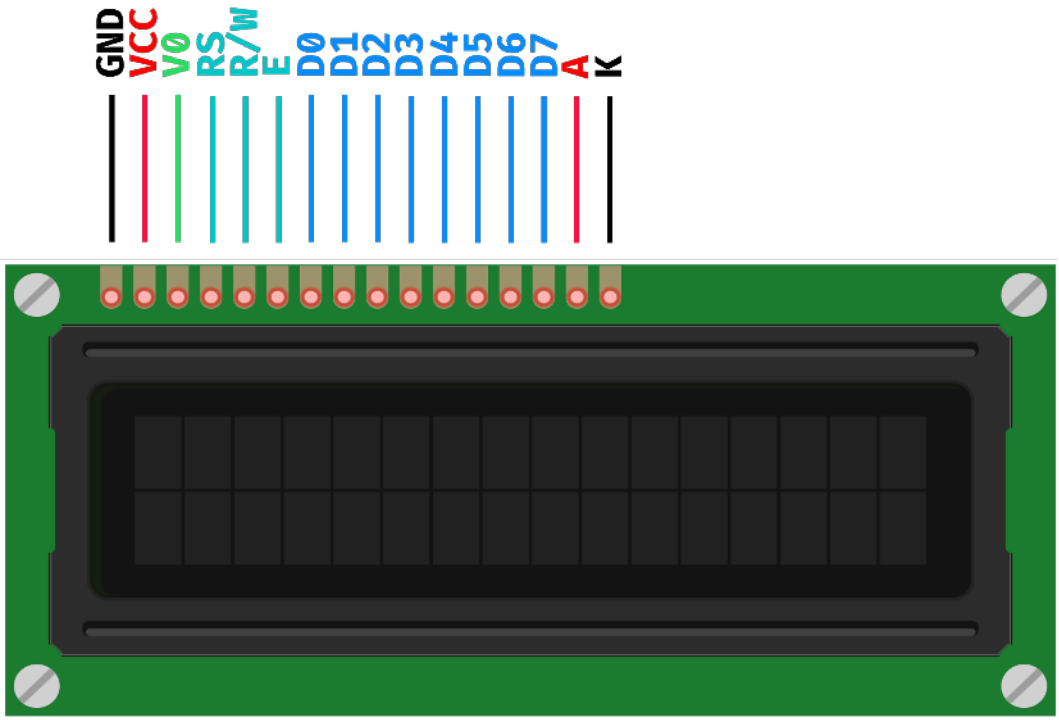

Figure 5.3: Figure 5.3: Pinout for commonly available 16x2 LCD modules

Pin	Symbol	Description
1	GND	Display ground.
2	VCC	Display power. Connect to 5V.
3	V0	Display contrast adjustment. 0-5V range.
4	RS	Register select. 0 for instructions, 1 for data.
5	R/W	Read/write. 0 for write, 1 for read.
6	E	Enable. Starts data read/write operation.
7	D0	Data bit 0, used in 8-bit mode.
8	D1	Data bit 1, used in 8-bit mode.
9	D2	Data bit 2, used in 8-bit mode.
10	D3	Data bit 3, used in 8-bit mode.
11	D4	Data bit 4, used in 4-bit and 8-bit mode.
12	D5	Data bit 4, used in 4-bit and 8-bit mode.
13	D6	Data bit 4, used in 4-bit and 8-bit mode.

Pin	Symbol	Description
14	**D7**	Data bit 4, used in 4-bit and 8-bit mode.
15	**A**	Backlight LED anode. Connect to 5V.
16	**K**	Backlight LED cathode. Connect to GND.

Table 5.2: Table 5.2: Pin designations and descriptions for the 16x2 LCD display

5.2.3 LCD Wiring

As referenced above, these displays support both a 4-bit and an 8-bit data transfer mode, with the 8-bit data length allowing for faster and simpler transmissions while the 4-bit data length increases software complexity but requires fewer GPIO pins to be allocated.

In both cases, the display also requires three additional control signals, `RS`, `RW`, and `E`. The `RS` line selects between the instruction register (if set to 0), and the data register (if set to 1) of the HD44780 controller, allowing data sent to be interpreted as a command or a character to display. The `RW` line configures the data pins for read or write mode, from the perspective of the host MCU. Because the display will receive commands and data from the MCU most of the time, the `RW` line will often be 0, however, in some cases such as reading the address of the display cursor or the display's busy signal, this line should be brought high. Finally, the `E`—or enable—signal causes a data transfer to occur. When writing to the display, the data and control lines should first be set up, and then the enable line quickly toggled on then back off, causing the HD44780U to accept the command or data.

In total, the 4-bit data mode will use a minimum of 7 GPIOs, and the 8-bit mode a minimum of 11 GPIOs. While they are not prohibitive, these are significant pin allocations for a single peripheral, and care must be taken when designing an embedded system to make good use of available pins.

Therefore, this section will take into account the additional complexity of the 4-bit data mode, making the 8-bit mode comparatively trivial to implement. To begin, the LCD should be connected to the EFR32XG24 Dev Kit as shown in the schematic in Figure 5.4 and the wiring diagram in Figure 5.5.

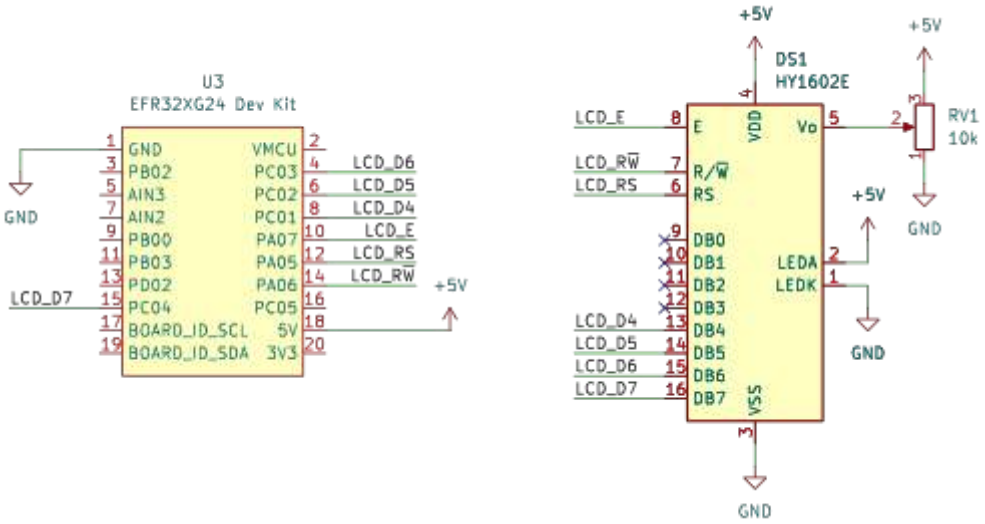

Figure 5.4: Figure 5.4: Schematic diagram of LCD connections to EFR32XG24 Dev Kit

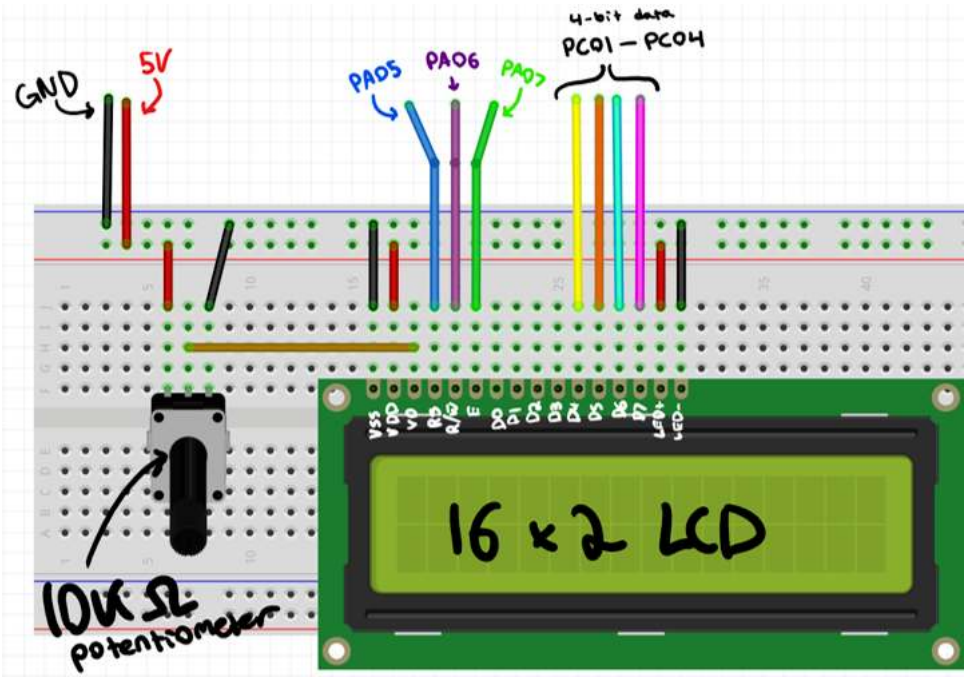

Figure 5.5: Figure 5.5: Wiring diagram for LCD connections to EFR32XG24 Dev Kit

5.2.4 LCD Data Transfer

The LCD accepts command and data bytes on the four or eight connected data lines, on the falling edge of a single pulse of the enable line. The three control lines and all of the data lines must first be written to, so that the data on them is valid at the time of the enable line pulse. In 8-bit mode, each pulse of the enable line corresponds with a single instruction or data. However, in 4-bit mode, two consecutive writes or reads are necessary to transmit a full command. The command's byte must be split into two nibbles—groups of four bits— and then transmitted with the most significant bits (MSBs) `7:4` first, followed by the least significant bits (LSBs) `3:0`. At the completion of the second data transfer, the LCD will execute the command and, after a brief period, be ready to accept more. The following code implements the protocol described above to send instructions or data to the LCD. Note that this code assumes that the control and data pins have already been configured as outputs.

```
void lcd_nibble_write(uint8_t data, uint8_t register_select)
{
    lcd_wait(); // wait until busy flag is not set (covered later in chapter)

    GPIO->P_CLR[DATA_port] = DATA_mask;               // clear data bits PC04-PC01
    GPIO->P_SET[DATA_port] = (data << 1) & DATA_mask; // set data bits shifted onto the correct pins

    if (register_select) // data
        GPIO->P_SET[CTRL_port] = 1 << RS_pin;
    else // command
        GPIO->P_CLR[CTRL_port] = 1 << RS_pin;

    GPIO->P_SET[CTRL_port] = 1 << EN_pin; // set enable
    sl_sleeptimer_delay_millisecond(1);
    GPIO->P_CLR[CTRL_port] = 1 << EN_pin; // unset enable
}
```

The `lcd_nibble_write` function takes two arguments, the first being the data (whether it be an instruction or character) to transmit, and the second being a boolean for the register select line. First, the function checks the LCD busy flag to determine if the LCD controller is ready to accept new data. This function will be discussed later, and may be implemented or replaced with a delay. The data lines are then cleared so that any previously sent data does not interfere with the new data to be written. As the data is expected in the lower four bits of the data byte, it is shifted into the correct position on the port based on the wiring diagram. Depending on the wrapper code for this function, an additional masking of the data may be wise to avoid tampering with other GPIO pins. The register select pin is also written to match the `register_select` argument, and finally the enable line toggled to complete the transmission.

5.2.5 LCD Instructions

Sending a full command or character to the LCD just requires two calls to the already-implemented `lcd_nibble_write` function, one for each nibble that must be transmitted. It may be beneficial to write a wrapper function that does this automatically, requiring only the full byte of data, and potentially a register select argument to complete the entire process. This would involve shifting the data argument right four bits, calling `lcd_nibble_write` to transmit these MSBs, then masking out the upper four bits of the data argument and again calling `lcd_nibble_write`.

With the understanding of sending full commands to the LCD, it can now be properly initialized. To do so, it is necessary to consult the HD44780U datasheet to properly form the LCD commands. An excerpt from the datasheet is included in Figure 5.6.

	Code										
Instruction	**RS**	**R/$\overline{W}$**	**DB7**	**DB6**	**DB5**	**DB4**	**DB3**	**DB2**	**DB1**	**DB0**	**Description**
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.

Figure 5.6: Figure 5.6: Instruction table for HD44780 (HD44780U Datasheet page 25)

Going through these instructions, it is clear that the command itself is determined by the place of the leftmost set bit.

The first couple of instructions at the top, Clear display and Return home, do not require arguments, and therefore require no bits lower than the leftmost set bit to change their behavior.

The next command, eEtry mode set, determines if the LCD's internal DDRAM address counter is increased or decreased after a character is sent. The DDRAM address corresponds with the cursor position, so it is most common for bit 1 to be set for this command. Bit 0 in the entry mode set command controls if the display should be shifted, as in, the cursor remains in the same position on the display and all other letters scroll around it when a character is sent. This mode is sometimes useful for displaying a wide line of scrolling text.

The Display on/off control command allows the display itself, the cursor, and the cursor blinking to be turned on or off. Setting any of the argument bits for this command turns them on. For text entry, it is common for all three bits (`2:0`) to be set, giving a blinking cursor for the next character. For status or sensor reading displays, only bit 2 (entire display) should be set, as the cursor would be visually distracting in this case.

The Cursor or display shift command manually increments or decrements the cursor position, or shifts the entire display right or left. This may be used for a backspace action or for scrolling text.

The Function set command is important for initialization of the display. The data length bit (4) selects between 4- and 8-bit modes, with 0 representing 4 bits and 1 representing 8 bits. The value for this bit will depend on the wiring for the LCD chosen previously. The number of lines bit (3) configures the display controller for 1 or 2 lines of text, with 0 representing 1 line, and 1 representing 2 lines. The value for this bit should be chosen based on the LCD hardware in use. Finally, the character font bit (2) chooses between a 5x8 or 5x10 character font, and should also be chosen depending on the LCD's capabilities.

The Set CGRAM address and Set DDRAM address are nearly identical, differing only in the number of bits available for the address and the RAM to write to. The CGRAM may be reconfigured while in operation with custom characters, and using the Set CGRAM address command is useful to select the custom character to overwrite. The DDRAM, which stores characters themselves that have been sent to the display, is more commonly used. Because the LCD DDRAM stores 80 character bytes, and the display is split into two lines, the second line begins at byte 40 of the DDRAM. This means that writing more than 16 characters on the first line will not automatically wrap to the second for many more characters; instead, the DDRAM address must be set to decimal 40 to begin the second line. For both commands, the address is specified in the bits lower than the leftmost set bit, and should be a valid address within the memory limits of the display controller.

The last command in the Figure 5.6 instruction table requires the `R/W` bit to be set and the data lines used as inputs to the host MCU. This command allows the LCD busy flag to be read using bit 6 of the LCD controller's response. It also allows for the host MCU to read the LCD controller's address counter in the lower bits `5:0`.

When using the 4-bit data length, each of these commands must be constructed by the MCU, then split into the high-order and low-order nibbles to send sequentially to the LCD.

5.2.6 LCD Initialization

With all LCD commands accounted for, the LCD may now be initialized before use. Included in Figure 5.7 is the steps necessary to initialize the LCD in 4-bit mode.

At power-up, every LCD character will be fully filled in, initialized, and cleared before characters can be written to it. Based on Figure 5.7, despite the LCD being automatically reset at power on, a manual reset sequence is necessary to synchronize the nibble transmissions. This reset sequence uses only `lcd_nibble_write`, as it is not yet ready to receive full commands. After this reset sequence is completed, the 4-bit mode, number of lines, and character size are then set in a single function set command, and further configuration commands may be used to clear the display, move the cursor, or adjust scrolling before characters are written to the LCD for the first time.

5.2.7 LCD Usage

Now that the LCD is initialized, characters may be written to the LCD by sending their ASCII codes, split up into 4-bit nibbles, to the LCD with the `RS` control line now set high. This will cause the LCD to write these characters into the DDRAM, where they are directly displayed.

Many effects may be created by combining the Set DDRAM address and cursor/display shift commands, including left, center, and right-aligned text, scrolling text, or even a small table of sensor readings.

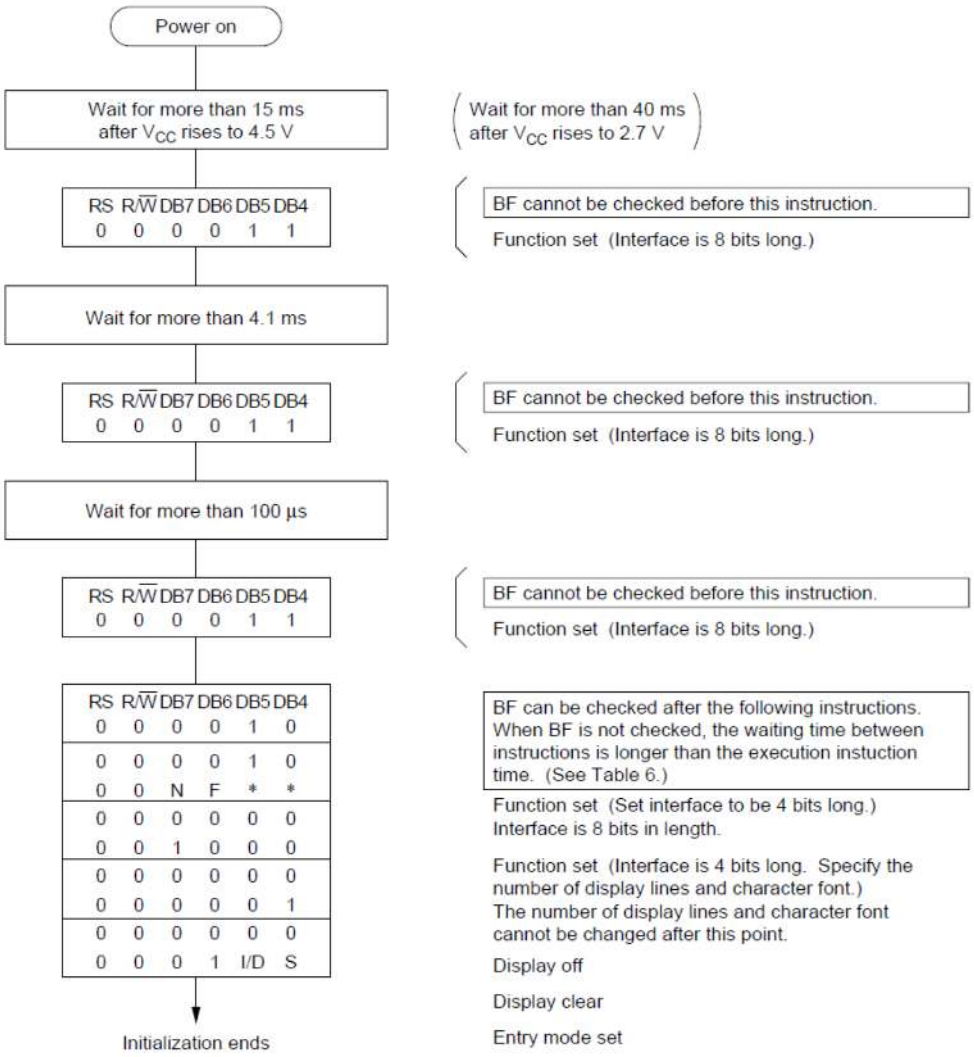

Figure 5.7: Figure 5.7: Block diagram of initialization sequence for HD44780U in 4-bit mode (HD44780U Datasheet page 46)

5.2.8 LCD Wait

The LCD's busy flag can be read while it processes commands internally. To handle this, you can implement the `lcd_wait` function, which repeatedly reads the busy flag by setting R/W to 0 and configuring the data lines as input. When the function should wait in a while loop reading the busy flag until the LCD is again ready to accept commands. Calling `lcd_wait` should be done before sending any instructions or data to the LCD, to ensure that it is ready to receive data. Alternatively, you

can use the `sl_sleeptimer_delay_ms` function to wait for a duration longer than any command processing requires. While this latter approach is simpler, it is less effective for high-frequency display updates due to the inherent required delay. For this technique, waiting for 2 milliseconds following any command is practical and easy to implement.

5.2.9 Exercise: Displaying a centered string

For this exercise, write a function that writes a c-string argument centered on the first line of the display. Check that the string passed to the function is no longer than 16 characters. With this condition met, calculate based on the length of the string the DDRAM address offset necessary to center the string.

For example, the string `EE260` is 5 characters long. The number of characters that should be blank on the line is $16-5=11$. To left align the text, the necessary offset is obviously 0. To right align the text, the offset should be 11, so that the 5 additional characters are placed directly touching the right side of the display. To center align the text, the remaining blank characters must be divided by 2. An integer division of $11/2=5$ as the decimal is truncated, meaning that the necessary DDRAM offset is 5, which will leave 6 characters to the right of the text.

The necessary DDRAM offset may then be set using the appropriate LCD command, then the characters of the string transferred to the display.

As an extension to this exercise, you may write a function that takes an additional argument to select the type of alignment and display line to place an arbitrary string of text on.

5.3 Keypad

Many embedded systems with user interfaces are controlled by simple inputs, such as a joystick, multifunctional knobs, or often, a group of buttons. In cases where many buttons are required, such as for numerical or even text input, connecting a single button to its own GPIO pin is inefficient. With a 4x4 grid of buttons requiring 16 pins, an I/O expander or separate microcontroller dedicated to I/O would likely be necessary. However, a clever arrangement of switches in a grid such as this allows for pins to be multiplexed, requiring a minimum of $\sqrt{\#\text{ of buttons}}$ pins to read each button. This works by connecting one side of each switch to a common row, and the other side of the switch to a common column. For a 4x4 grid, only 8 pins are necessary to read the entire matrix layout.

5.3.1 Keypad Matrix Wiring

Commonly available matrix keypads simply implement the row and column switch connections to pushbuttons integrated in the module. Their pinout is often just a connection for each row and column, allowing them to easily be connected to GPIO pins on an MCU, as shown in Figure 5.8.

The internal connections of the keypad may be displayed differently depending on preferences for the schematic. However, it is common to see the switches aligned on a 45angle, bridging between their respective row and column common lines, as illustrated in Figure 5.9.

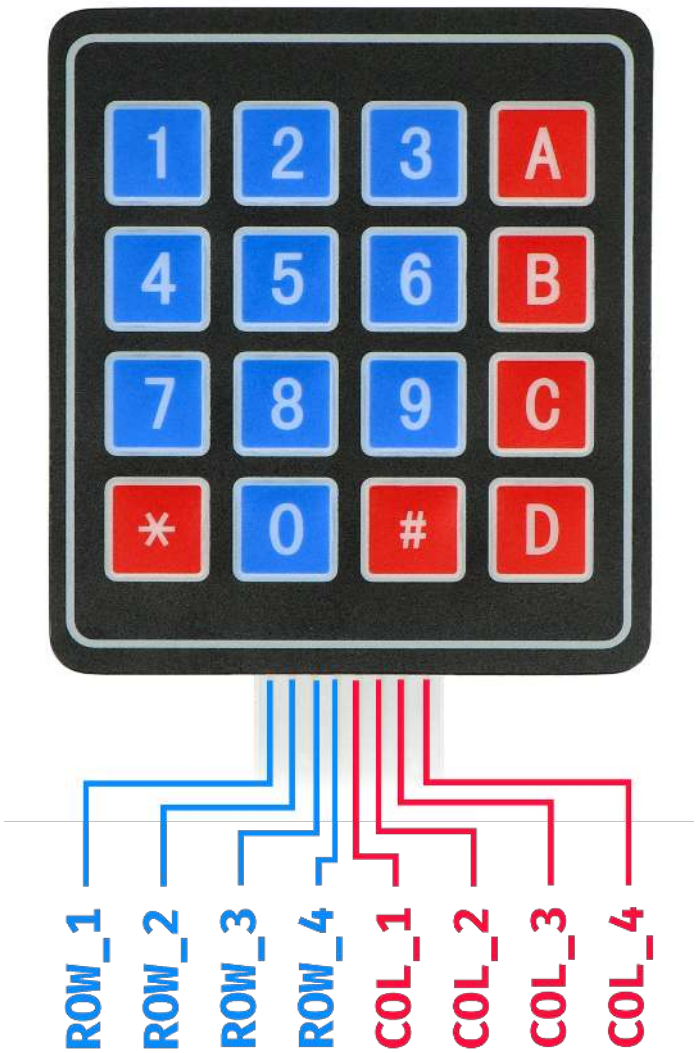

Figure 5.8: Figure 5.8: Pinout for an off-the-shelf keypad

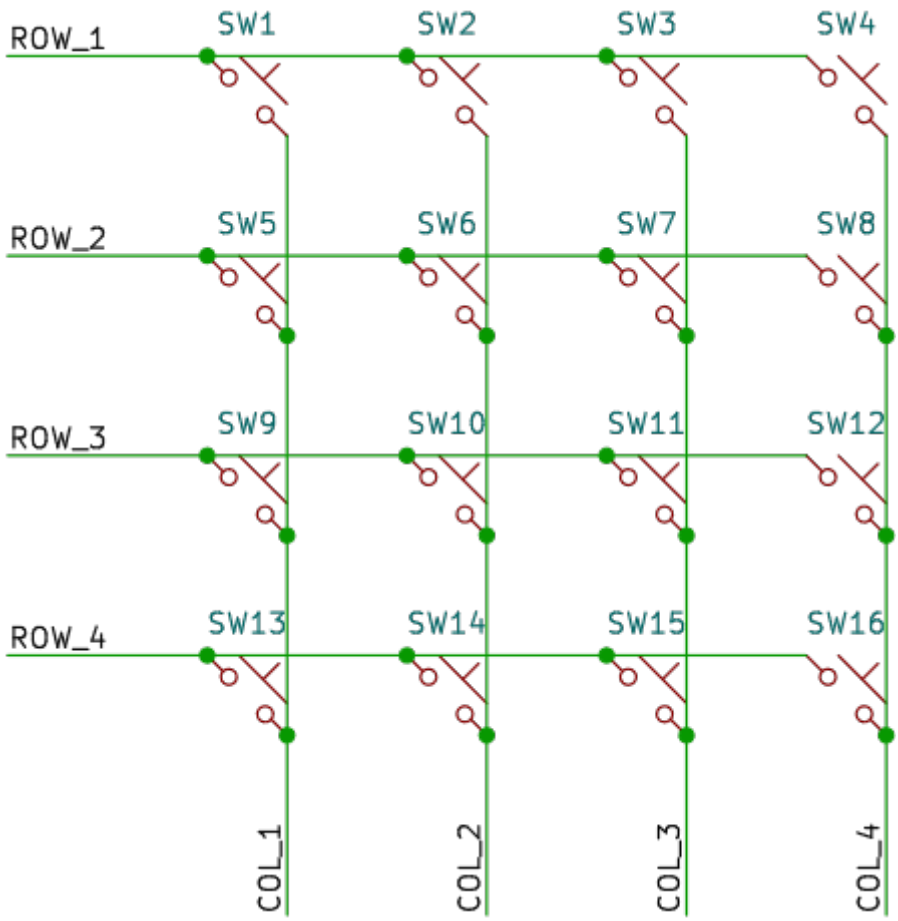

Figure 5.9: Figure 5.9: Schematic diagram of a 4x4 pushbutton switch matrix

5.3.2 Reading Keypad Matrix

To read a matrix of switches, one axis should be connected to GPIO outputs. For example, we will connect the rows to output pins, writing all pins high to begin with. The other axis should be connected to internally pulled-down inputs, meaning that when any key is pressed, one of the high rows will be connected to the input, bringing it high. When this is detected, either with polling or an interrupt-based system, the microcontroller may now identify which key has been pressed.

5.3.3 Identifying Pressed Key

Once the MCU has been alerted that any key has been pressed, it may now scan the switch matrix to determine the specific key. To do this, all rows should be written low, except for the first row. This may be achieved in practice by writing all rows low first, then immediately writing the first row high. The MCU may then check if any key in the first row has been pressed by again reading all of the column

inputs. If any of the column inputs are high, the currently checked row and column that is high correspond with the pressed key. Otherwise, the MCU must repeat the process, writing the next row low. In this way, the pressed key can quickly be determined, and other actions can be taken based on it. This process is outlined in Figure 5.10, a flowchart showing the logic necessary for efficient detection of keypresses.

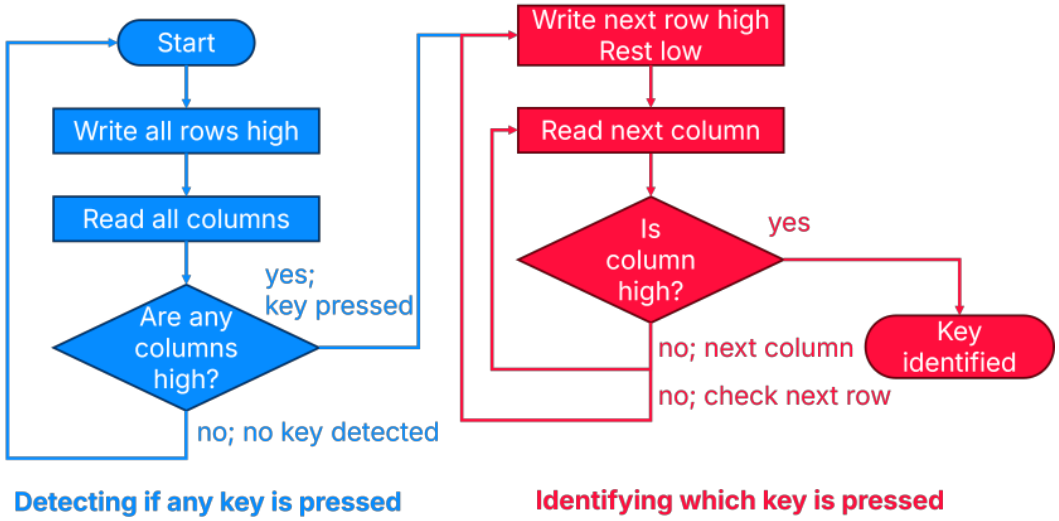

Figure 5.10: Figure 5.10: Flowchart for detecting any keypress, then identifying the speci ic key

A simplified sample implementation of this process is included below. The code implements nested-loop logic to scan a 4x4 matrix keypad using EFR32XG24 Dev Kit GPIO pins. First, the GPIO pins are initialized, with the column input pins (PC04–PC01) are configured as inputs, and the row output pins (PC05, PA07–PA05) are set to push-pull (output) mode. In the main loop, all the row pins are activated by setting them high. Then, if any of the column pins detects a high signal, representing a key press, the program iterates through each row to isolate the pressed key. During this process, all rows are brought low save for the current row of interest, and the program checks each column pin to identify the specific key pressed.

```
// column inputs are on PC04-PC01
const uint8_t row_ports[] = {2, 0, 0, 0}; // PC05, PA07-PA05 are row outputs
const uint8_t row_pins[] = {5, 7, 6, 5};  // PC05, PA07-PA05 are row outputs

int main(void)
{
    GPIO->P[gpioPortC].MODEL |= 0x1111 << (1 * 4); // input mode

    for (int i = 0; i < 4; i++)
        GPIO->P[row_ports[i]].MODEL |= 0x4 << (row_pins[i] * 4); // output mode
```

```
    while (1)
    {
        for (int i = 0; i < 4; i++)
            GPIO->P_SET[row_ports[i]] = 1 << row_pins[i]; // set row pins

        if (GPIO->P[gpioPortC].DIN & 0xF << 1) // check if any key is pressed
        {
            for (int i = 0; i < 4; i++) // loop through all columns
            {
                for (int j = 0; j < 4; j++)
                    GPIO->P_CLR[row_ports[j]] = 1 << row_pins[j]; // clear all row pins
                GPIO->P_SET[row_ports[i]] = 1 << row_pins[i];     // set current row pin

                for (int i = 0; i < 4; i++)
                {
                    if (GPIO->P[gpioPortC].DIN << 1 & 1 << i) // check if column pin is high
                    {
                        // this is the key pressed
                    }
                }
            }
        }
    }
}
```

5.3.4 Power Efficiency

A key benefit of reading a switch matrix using this technique, especially with the EFR32MG24, which supports GPIO interrupts from all Energy Management levels. This means that the processor may go into its deepest sleep state while still waiting for keypresses from the matrix. This requires interrupt logic, which will be discussed later, but the general implementation is as follows:

The GPIO pins for the rows may be set high, and configured to retain their values while in sleep mode. An interrupt to wake the processor from sleep may be enabled on all input pins, meaning that any keypresses will now trigger the interrupt, waking the MCU from sleep. It can now progress directly into the key identification phase, finding the pressed key and performing an action before returning to low-power sleep.

5.3.5 Limitations of Switch Matrix

There exist a few limitations with this naive technique of reducing GPIO pin usage for a switch matrix, the most significant being the lack of *key rollover*. This means that the MCU cannot identify multiple keys being pressed, at least not with certainty.

Finally, if the keypress time is very short, a key pressed and caught by the detection routine may already be released and missed by the identification routine. This can be compounded by switch bouncing because many keyboard matrices lack hardware debouncing, while software debouncing requires keys to be pressed for a longer period of time before the press is registered.

5.3.6 Exercise: Propose a solution to the key rollover problem

Modern computer keyboards can detect any of their keys being pressed, as well as any combination of keys being pressed. To do this, they do not even require multiple I/O expanders or additional microcontrollers. Instead, there is electronic hardware integrated into the matrix circuit in series with every switch that ensures the proper key is detected.

What could this hardware be? Explain how it may be used to support multiple simultaneous key-presses.

Embedded Machine Learning

Chapter 6

Real-Time Gesture Recognition

Edge AI combines the power of artificial intelligence and edge computing to enable real-time decision-making directly on embedded devices. This chapter explores the implementation of Edge AI for gesture recognition using the EFR32XG24 microcontroller. By leveraging AI models optimized for resource-constrained environments, embedded systems can interpret gesture inputs with low latency and high accuracy, enhancing applications in fields such as human-computer interaction, rehabilitation, and IoT-based control systems.

6.1 Introduction to Edge AI in Embedded Systems

Edge AI refers to deploying AI models on edge devices, such as microcontrollers, where data is processed locally instead of being sent to a centralized cloud server. This paradigm reduces latency, enhances data privacy, and ensures uninterrupted operation even in environments with limited connectivity.

As embedded systems become more advanced, the need for efficient on-device data processing grows. Traditional systems rely heavily on cloud infrastructure, which can introduce latency, data privacy concerns, and increased operational costs. Edge AI addresses these issues by allowing computations to occur on the microcontroller itself, ensuring responsiveness and independence from network stability. With microcontrollers like the EFR32XG24, AI algorithms are executed efficiently despite hardware and memory constraints.

```
// Example: Initialize Edge AI on EFR32XG24
void initEdgeAI() {
    configureClock();
    enableAIAccelerator();
    loadAIModel();
    initializeSensors();
}

initEdgeAI();
```

6.1.1 Advantages of Edge AI

Edge AI offers several critical advantages that make it an essential technology for embedded systems:

- **Low Latency:** Immediate processing without reliance on cloud servers.
- **Improved Privacy:** Sensitive data remains on the device.
- **Reduced Bandwidth Usage:** No need for constant data transmission.
- **Energy Efficiency:** Optimized AI models reduce power consumption.
- **Scalability:** Multiple devices can operate independently.
- **Offline Operation:** Systems continue functioning without an active internet connection.

These benefits are especially important in applications like gesture recognition, where real-time response is crucial for effective interaction. Devices deployed in remote or resource-limited environments can operate reliably without relying on continuous cloud access.

```
// Example: Optimizing AI Model
void optimizeAIModel() {
    reducePrecision();
    quantizeWeights();
    minimizeMemoryFootprint();
}

optimizeAIModel();
```

6.1.2 Why EFR32XG24 for Edge AI?

The EFR32XG24 microcontroller, equipped with an ARM Cortex-M33 core and integrated BLE capabilities, provides an ideal platform for Edge AI applications. Its features include:

- Support for TinyML frameworks.
- Dedicated hardware accelerators for AI computations.
- Energy-efficient architecture for battery-operated devices.
- Advanced security features for data integrity.
- High-speed BLE communication for real-time data transfer.
- Integrated peripherals for sensor interfacing.

```
// Example: BLE Initialization for Edge AI
void initBLE() {
    BLE_init();
    BLE_enable();
    BLE_setMode(BLE_LOW_POWER);
}

initBLE();
```

Furthermore, the microcontroller's native support for TensorFlow Lite for Microcontrollers (TFLM) allows seamless deployment of lightweight AI models. Its power efficiency ensures prolonged operational life.

6.2 Gesture Recognition System Overview

Gesture recognition systems are a subset of human-computer interaction technologies that allow users to control and interact with devices using physical gestures. In an embedded context, gesture recognition systems aim to interpret motion patterns captured by sensors like IMUs (Inertial Measurement Units). The EFR32XG24 microcontroller enables real-time gesture recognition while maintaining energy efficiency and responsiveness.

6.2.1 System Components

A gesture recognition system using the EFR32XG24 microcontroller involves several key components:

- **Sensors:** Inertial Measurement Unit (IMU) for capturing motion data.
- **AI Model:** A lightweight Convolutional Neural Network (CNN) optimized for TinyML.
- **Data Preprocessing:** Noise filtering and segmentation.
- **BLE Communication:** Real-time data transfer to mobile devices.
- **Power Management System:** Ensures long battery life.

```
// Example: Read IMU Sensor Data
float readIMU() {
    float x = IMU_getX();
    float y = IMU_getY();
    float z = IMU_getZ();
    return (x + y + z) / 3;
}
```

6.2.2 System Workflow

The overall workflow of a gesture recognition system includes the following stages:

1. Raw sensor data is captured using IMU sensors.
2. Data is preprocessed locally to remove noise.
3. Preprocessed data is fed into the AI model.
4. The AI model classifies the gesture.
5. Results are sent via BLE to a connected device.
6. Feedback is displayed in real-time on a mobile or desktop application.

6.3 AI Model Design for Gesture Recognition

The AI model for gesture recognition is implemented using a TinyML-compatible CNN architecture.

6.3.1 Model Architecture

The CNN architecture is carefully designed to balance accuracy, memory consumption, and computational efficiency:

- **Input Layer:** Processes time-series IMU data.
- **Convolutional Layers:** Extract spatial and temporal patterns.
- **Dropout Layers:** Prevent overfitting.

- **Fully Connected Layer:** Classifies gestures.
- **Softmax Layer:** Provides final classification probabilities.

```
// Example: Preprocessing IMU Data
void preprocessIMUData(float* data, int size) {
    for (int i = 0; i < size; i++) {
        data[i] = normalize(data[i]);
    }
}
```

Layer	Type	Output Shape
Input	Time-Series Data	(128, 3, 1)
Conv2D	Feature Extraction	(64, 128, 3, 8)
MaxPooling2D	Downsampling	(32, 64, 8)
Dropout	Regularization	-
Flatten	Vectorization	(512)
Dense	Classification	(16)
Output	Softmax	(4)

Table 6.1: Table 6.1: AI Model Layers and Parameters

6.4 Methodology

This section outlines the methodology used to design and implement an Edge AI-based gesture recognition system on the EFR32XG24 BLE microcontroller. The methodology consists of four main stages: Data Acquisition, Data Preprocessing, AI Model Development, and System Integration. Each stage is elaborated below.

6.4.1 Data Acquisition

The gesture recognition system utilizes an Inertial Measurement Unit (IMU) sensor to capture motion data. The IMU consists of accelerometers, gyroscopes, and magnetometers to measure linear acceleration, angular velocity, and orientation, respectively.

Sensor Configuration:

- **Sensor Type:** 6-axis IMU sensor.
- **Sampling Rate:** 50 Hz.
- **Data Format:** Time-series data with X, Y, and Z-axis readings.

The IMU sensor outputs raw motion data, which is collected in real-time and fed into the microcontroller for preprocessing.

6.4.2 Data Preprocessing

Raw data from the IMU sensor is noisy and requires preprocessing before being fed into the AI model. The preprocessing pipeline includes the following steps:

1. **Noise Filtering:** A low-pass filter is applied to remove high-frequency noise.
2. **Normalization:** Sensor readings are normalized to a common scale between 0 and 1.
3. **Segmentation:** The data is divided into fixed-size time windows for analysis.

The preprocessed data ensures consistency and reduces variability, enabling robust AI model performance.

6.4.3 AI Model Development

The AI model is implemented using a TinyML-compatible Convolutional Neural Network (CNN). The architecture is optimized for low memory and computational constraints typical of embedded devices.

Model Architecture:

- **Input Layer:** Accepts preprocessed IMU time-series data.
- **Convolutional Layers:** Extract spatial and temporal patterns.
- **Pooling Layers:** Reduce dimensionality while retaining critical features.
- **Dropout Layers:** Prevent overfitting during training.
- **Fully Connected Layers:** Map learned features to output classes.
- **Output Layer:** Softmax layer providing probabilities for each gesture class.

Training and Optimization:

- **Framework:** TensorFlow Lite for Microcontrollers (TFLM).
- **Training Dataset:** Recorded gesture datasets.
- **Optimization Techniques:** Weight quantization, reduced precision arithmetic, and model pruning.

The model was trained offline on a high-performance server and deployed onto the EFR32XG24 microcontroller after optimization.

6.4.4 System Integration

The final deployment involves integrating the AI model with the EFR32XG24 microcontroller and establishing BLE communication for data transfer and feedback display.

System Workflow:

1. IMU sensors capture real-time motion data.
2. Data preprocessing is performed locally on the microcontroller.
3. The preprocessed data is passed to the AI model for gesture classification.
4. Results are transmitted via BLE to connected mobile or desktop applications.
5. Feedback is displayed in real-time.

Power Management:

- Adaptive power management is implemented to minimize battery consumption.
- Low-power BLE mode is enabled for data transmission.

6.4.5 Evaluation Metrics

The system's performance was evaluated using the following metrics:

- **Accuracy:** Percentage of correct gesture classifications.
- **Latency:** Time taken for end-to-end gesture recognition.
- **Power Consumption:** Average energy used per gesture recognition cycle.

6.4.6 Hardware and Software Tools

- **Hardware:** EFR32XG24 BLE microcontroller, IMU sensor module.
- **Software:** TensorFlow Lite for Microcontrollers, Embedded C, BLE SDK.

The methodology ensures an efficient, real-time, and scalable gesture recognition system optimized for embedded hardware constraints.

6.5 Challenges and Limitations

While implementing the Edge AI-based gesture recognition system on the EFR32XG24 microcontroller, several challenges and limitations were encountered. These are discussed below:

1. **Hardware Constraints:** The limited computational power and memory resources of the microcontroller posed restrictions on the complexity and size of the AI model. Optimizing the AI model for memory efficiency while maintaining accuracy was a significant challenge.
2. **Power Consumption:** Real-time gesture recognition is computationally intensive, and ensuring prolonged battery life for portable devices required careful power management strategies.
3. **Sensor Noise:** IMU sensors are susceptible to noise and environmental disturbances, which can introduce inaccuracies in gesture data. Filtering and preprocessing techniques had to be carefully designed to address this issue.
4. **Latency Constraints:** Edge AI systems require minimal latency for real-time performance. Achieving low latency while balancing model accuracy and computational load was challenging.
5. **BLE Communication Bottlenecks:** Real-time data transfer via BLE can be affected by interference, limited bandwidth, and power constraints, impacting system responsiveness.
6. **Scalability:**** Scaling the system for multi-gesture recognition or increasing the number of edge devices posed integration challenges due to resource limitations.
7. **Environmental Variability:** Gesture recognition accuracy can degrade under varying environmental conditions, such as lighting changes, device orientation, or sudden movements.

Addressing these challenges required a combination of hardware optimization, software fine-tuning, and iterative testing to ensure a balance between performance, accuracy, and efficiency.

Chapter 7

Magic Wand via Gesture Recognition

The EFR32xG24 board has high capabilities for edge computing necessary for capturing, processing, and analyzing data in real time with improved security and reduced latency. The Magic Wand with BLE Gesture Recognition (BLE-MW) is an ideal project to exemplify this concept through the implementation of a TinyML classifier for gesture recognition. It incorporates a quantized TinyML, accelerometer data processing of the onboard IMU, and a BLE communication protocol. The project files are available at `github.com/Ijnaka22len/ble_magic_wand`, following the original implementation in `github.com/SiliconLabs/machine_learning_applications`.

7.1 System Overview

A convolutional neural network (CNN) model was trained in this TinyML project to recognize gestures such as Swipe Up, Swipe Down, and Circle based on the onboard accelerometer data. The detected gestures (Arrow Up, Arrow Down, and Play/Pause) are mapped to media control actions and transmitted over BLE as media key presses and over the UART interface. Figure 7.1 shows an instance of the output.

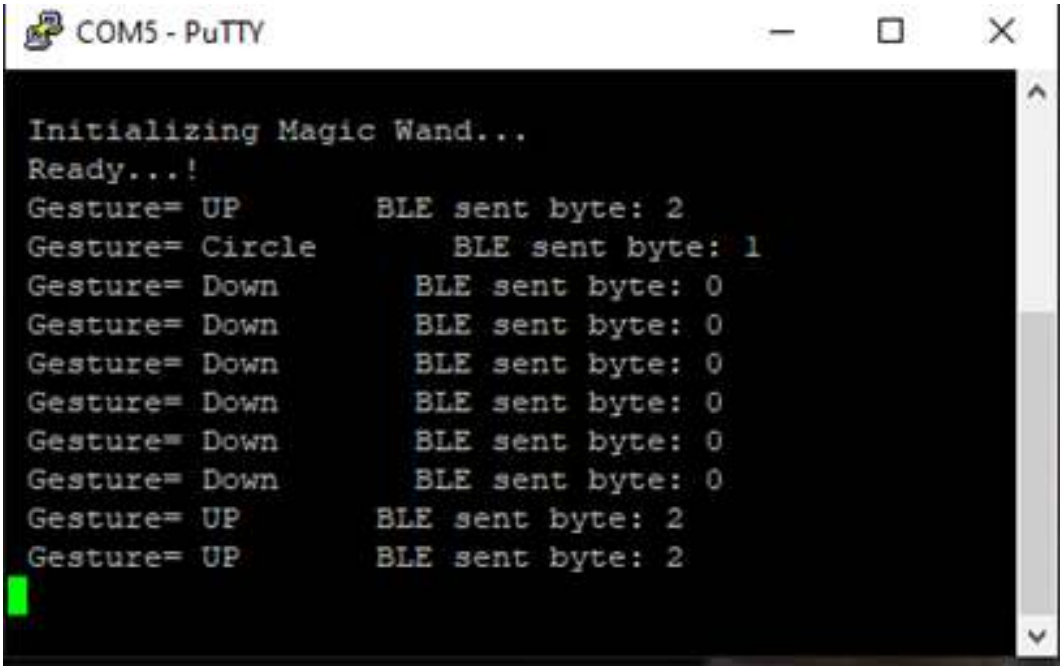

Figure 7.1: Figure 7.1: Sample Data Output via UART Terminal

7.2 Data Collection and Processing

The IMU data for this TinyML model was collected at 25Hz with a sequence length of 50 samples to form the input buffer for the CNN. The fastest way to do this is by following the `github.com/Ijnaka22len/FastDataCollection4MagicWangProjects` repository, which provides a detailed description of IMU data capture for TinyML projects. The captured data is preprocessed and fed into the model as an input image for pattern recognition.

```
if __name__ == "__main__":
    delFile(file_path="data/complete_data")
    delFile(file_path="data/test")
    delFile(file_path="data/train")
    delFile(file_path="data/valid")
    delFile(file_path="netmodels/CNN/weights.h5")

    folders = os.listdir("data")
    names = [file.split('_')[1].lower() for file in os.listdir(f"data/{folders[0]}")]
    data = []  # pylint: disable=redefined-outer-name

    for idx1, folder in enumerate(folders):
        files = os.listdir(f"data/{folder}")
        for file in files:
```

```
            name = file.split('_')[1].lower()
            prepare_original_data(folder, name, data, f"data/{folder}/{file}")

    for idx in range(5):
        prepare_original_data("negative", "negative%d" % (idx + 1), data, "data/negative/negative_%
    generate_negative_data(data)
    print("data_length: " + str(len(data)))

    if not os.path.exists("./data"):
        os.makedirs("./data")

    write_data(data, "./data/complete_data")
```

7.3 Model Architecture and Deployment

TensorFlow Lite is used to create the CNN model, which takes the processed IMU data as an input image for multiclass classification of the various classes. The AIDrawPen chapter can be reviewed to show how to train a TinyML model for this microcontroller.

```
"""Trains the model."""
calculate_model_size(model)
epochs = 50
batch_size = 64
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
if kind == "CNN":
    train_data = train_data.map(reshape_function)
    test_data = test_data.map(reshape_function)
    valid_data = valid_data.map(reshape_function)

test_labels = np.zeros(test_len)
idx = 0
for data, label in test_data:  # pylint: disable=unused-variable
    test_labels[idx] = label.numpy()
    idx += 1

train_data = train_data.batch(batch_size).repeat()
valid_data = valid_data.batch(batch_size)
test_data = test_data.batch(batch_size)

history = model.fit(train_data,
              epochs=epochs,
              validation_data=valid_data,
              steps_per_epoch=1000,
              validation_steps=int((valid_len - 1) / batch_size + 1),
```

```
              callbacks=[tensorboard_callback, early_stop, checkpoint])

loss, acc = model.evaluate(test_data)
pred = np.argmax(model.predict(test_data), axis=1)
confusion = tf.math.confusion_matrix(labels=tf.constant(test_labels),
                                     predictions=tf.constant(pred),
                                     num_classes=4)
```

7.4 Firmware and Model Inference

The microcontroller firmware contains a code segment to capture the accelerometer data at the predetermined frequency and sampling rate. This data is stored in a buffer and updated every 100ms during gesture detection.

```
#include "accelerometer.h"
#include "config.h"

#if defined(SL_COMPONENT_CATALOG_PRESENT)
#include "sl_component_catalog.h"
#endif

#if defined (SL_CATALOG_ICM20689_DRIVER_PRESENT)
#include "sl_icm20689_config.h"
#define  SL_IMU_INT_PORT SL_ICM20689_INT_PORT
#define  SL_IMU_INT_PIN  SL_ICM20689_INT_PIN
#elif defined (SL_CATALOG_ICM20648_DRIVER_PRESENT)
#include "sl_icm20648_config.h"
#define  SL_IMU_INT_PORT SL_ICM20648_INT_PORT
#define  SL_IMU_INT_PIN  SL_ICM20648_INT_PIN
#else
#error "No IMU driver defined"
#endif

// Accelerometer data from sensor
typedef struct imu_data {
  int16_t x;
  int16_t y;
  int16_t z;
} imu_data_t;

sl_status_t accelerometer_setup(GPIOINT_IrqCallbackPtrExt_t callbackPtr)
{
  sl_status_t status = SL_STATUS_OK;
  int int_id;

  // Initialize accelerometer sensor
  status = sl_imu_init();
```

```
  if (status != SL_STATUS_OK) {
    return status;
  }
  sl_imu_configure(ACCELEROMETER_FREQ);
  // Setup interrupt from accelerometer on falling edge
  GPIO_PinModeSet(SL_IMU_INT_PORT, SL_IMU_INT_PIN, gpioModeInput, 0);
  int_id = GPIOINT_CallbackRegisterExt(SL_IMU_INT_PIN, callbackPtr, 0);
  if (int_id != INTERRUPT_UNAVAILABLE) {
    GPIO_ExtIntConfig(SL_IMU_INT_PORT, SL_IMU_INT_PIN, int_id, false, true, true);
  } else {
    status = SL_STATUS_FAIL;
  }
  return status;
}

sl_status_t accelerometer_read(acc_data_t* dst)
{
  if (!sl_imu_is_data_ready()) {
    return SL_STATUS_FAIL;
  }
  sl_imu_update();
  int16_t m[3];
  sl_imu_get_acceleration(m);
  CORE_DECLARE_IRQ_STATE;
  CORE_ENTER_CRITICAL();
  if (dst != NULL) {
    dst->x = m[0];
    dst->y = m[1];
    dst->z = m[2];
  }
  CORE_EXIT_CRITICAL();
  return SL_STATUS_OK;
}
```

The quantized CNN model interprets the processed accelerometer data to classify gestures through periodic inference. Results are evaluated against the accepted threshold (#define DETECTION_THRESHOLD 0.9f).

```
#include "sl_tflite_micro_model.h"
#include "sl_tflite_micro_init.h"
#include "sl_sleeptimer.h"
#include "magic_wand.h"
#include "accelerometer.h"
#include "sl_simple_button_instances.h"
#include "math.h"
#include "config.h"
// BLE header
#include "sl_bluetooth.h"
```

```
#include "app_assert.h"
#include "gatt_db.h"
#include "em_common.h"
//
static int input_length;
static TfLiteTensor* model_input;
static tflite::MicroInterpreter* interpreter;
static acc_data_t buf[SEQUENCE_LENGTH] = { 0.5f, 0.5f, 0.5f };
static bool infer = false;
static bool read_accel = false;
static int head_ptr = 0;
static int inference_trigger_samples_num = round(INFERENCE_PERIOD_MS / ACCELEROMETER_FREQ);
static acc_data_t prev_data = { 0.5f, 0.5f, 0.5f };

static void listen_for_gestures(bool enable)
{
  if (enable) {
    for (uint8_t i = 0; i < SEQUENCE_LENGTH; i++) {
      acc_data_t _d = { 0.5f, 0.5f, 0.5f };
      buf[i] = _d;
    }
    read_accel = true;
  } else {
    read_accel = false;
    head_ptr = 0;
  }
}

void sl_button_on_change(const sl_button_t *handle)
{
  if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_PRESSED) {
    if (&sl_button_btn0 == handle) {
      listen_for_gestures(true);
    }
  } else if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_RELEASED) {
    if (&sl_button_btn0 == handle) {
      listen_for_gestures(false);
    }
  }
}

// Called when the IMU has data available using gpio interrupt.
static void on_data_available(uint8_t int_id, void *ctx)
{
  (void) int_id;
  (void) ctx;
  acc_data_t data = { 0, 0, 0 };
```

```
  sl_status_t status = accelerometer_read(&data);
  if (status == SL_STATUS_FAIL || !read_accel) {
    return;
  }

  data.x /= 2000;
  data.y /= 2000;
  data.z /= 2000;

  acc_data_t delta_data = { 0 };
  delta_data.x = data.x - prev_data.x;
  delta_data.y = data.y - prev_data.y;
  delta_data.z = data.z - prev_data.z;

  delta_data.x = (delta_data.x / 2 + 1) / 2;
  delta_data.y = (delta_data.y / 2 + 1) / 2;
  delta_data.z = (delta_data.z / 2 + 1) / 2;

  buf[head_ptr].x = delta_data.x;
  buf[head_ptr].y = delta_data.y;
  buf[head_ptr].z = delta_data.z;

  head_ptr++;
  prev_data.x = data.x;
  prev_data.y = data.y;
  prev_data.z = data.z;
  if (head_ptr >= SEQUENCE_LENGTH) {
    head_ptr = 0;
  }
  if (head_ptr % inference_trigger_samples_num == 0) {
    infer = true;
  }
}
```

7.5 Data Transfer and BLE Communication

Predicted data is transferred through BLE using the GATT server as well as logged to the Putty terminal.

```
void send_gesture_via_ble(uint8_t gesture)
{
    printf(" BLE sent byte: %u\r\n", (unsigned int)gesture);
    sl_status_t sc;
    sc = sl_bt_gatt_server_notify_all(gattdb_gesture_data,
                                      sizeof(gesture),
                                      &gesture);
    app_assert_status(sc);
```

```
}
```

Chapter 8

IMU Anomaly Detection Using Hierarchical Temporal Memory

Analyzing sensor data to identify rare or suspicious events, items or observation that differ significantly from standard patterns. This process is called anomaly detection, which is highly important in product production. This chapter demonstrates the identification of anomalies in real-time accelerometer data using the Hierarchical Temporal Memory (HTM) algorithms. This project aims to show how edge computing can be used to provide sensitive anomaly detection for diverse applications such as machinery monitoring, motion analysis, and safety-critical systems. The project files are available at `github.com/Ijnaka22len/imu_anomaly_detection`, following the original implementation in `github.com/SiliconLabs/machine_learning_applications`.

8.1 System Overview

Hierarchical Temporal Memory (HTM) is an algorithm that mimics the brain's neocortical learning mechanisms and constantly learns time-based patterns in unlabeled data. Anomalies (suspicious data) are flagged when data (IMU data) deviate significantly from learned patterns. In addition to the firmware, a Python script (`display_serial.py`) is written to monitor patterns and deviations.

8.2 Data Collection and Preprocessing

Like in Chapter 7, the IMU captures accelerometer data in real-time at a frequency of *25Hz*. The previous and current readings are processed to normalize motion along the x, y, and z axes into a $[-1, 1]$ range as shown below:

```
imu_data_normalized.x = imu_data_current.x - imu_data_prev.x;
imu_data_normalized.y = imu_data_current.y - imu_data_prev.y;
imu_data_normalized.z = imu_data_current.z - imu_data_prev.z;

imu_data_normalized.x /= 4000;
imu_data_normalized.y /= 4000;
imu_data_normalized.z /= 4000;
```

8.3 HTM Model Architecture

The IMU data is encoded into Sparse Distributed Representations (SDRs) to facilitate efficient anomaly detection.

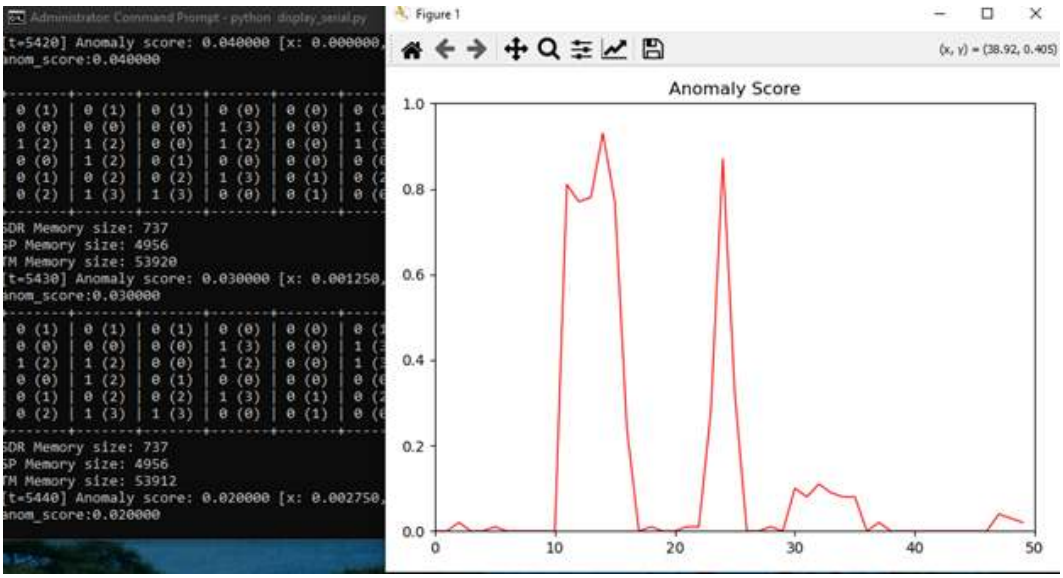

Figure 8.1: Figure 8.1: Example of IMU Anomaly

```
sl_htm_encoder_simple_number(imu_data_normalized.x, -1.0f, 1.0f, 9, &sdr_x);
sl_htm_encoder_simple_number(imu_data_normalized.y, -1.0f, 1.0f, 9, &sdr_y);
sl_htm_encoder_simple_number(imu_data_normalized.z, -1.0f, 1.0f, 9, &sdr_z);

sl_htm_sdr_insert(&input_sdr, &sdr_x, 0, SDR_WIDTH / 3 * 0);
sl_htm_sdr_insert(&input_sdr, &sdr_y, 0, SDR_WIDTH / 3 * 1);
sl_htm_sdr_insert(&input_sdr, &sdr_z, 0, SDR_WIDTH / 3 * 2);
```

8.4 Visualization and Real-Time Monitoring

The Python script `display_serial.py` visualizes anomaly scores in real-time by reading from a serial port connected to the microcontroller at a *baudrate=115200*. The script maintains a rolling buffer of scores and updates a live plot. This script helps visualize identified anomalies, such as irregular vibrations or sudden movements.

```
if line.startswith("anom_score"):
    line_info = line.split(":")
    anomaly_score = float(line_info[1])
    buffer.append(anomaly_score)
    buffer = buffer[1:]
```

```
    axs.plot(buffer, color="red", linewidth=1)
    fig.tight_layout()
    fig.canvas.draw()
    plt.pause(0.001)
    axs.clear()
```

Chapter 9

Audio ML for EFR32

Implementing audio machine learning (ML) on microcontrollers has become increasingly common in recent years. Optimizing the model is crucial to achieve accurate results while maintaining energy efficiency, ensuring suitability for high-performance embedded systems. This chapter details the implementation of a basic *Yes/No* audio detection system using ML. Initially, a neural network is trained to classify two audio classes: *Yes* and *No*. The trained model is then deployed on the EFR32 Dev kit.

9.1 Overview

In the implementation below, the system processes raw audio input and categorizes it as either *Yes* or *No* using the trained ML model. The workflow includes three key stages:

- Training the ML model using TensorFlow.
- Converting the model to a TensorFlow Lite format.
- Deploying the model on the EFR32 MCU for real-time inference.

The explanations and code examples are presented below for clarity. The required concepts are as follows:

- **TensorFlow:** An open-source framework widely used for developing and deploying machine learning models across platforms, including mobile and embedded systems. TensorFlow provides tools for building, training, and optimizing neural networks, along with TensorFlow Lite for Microcontrollers, which allows ML models to run efficiently on resource-constrained devices.
- **Audio Features:** Raw audio data, typically represented as waveforms, is transformed into meaningful numerical representations suitable for ML models. Commonly used features include Mel Frequency Cepstral Coefficients (MFCCs), which capture the spectral properties of audio signals, and Spectrograms, which represent the frequency content over time. These features enable neural networks to identify patterns and classify audio inputs accurately.
- **Edge ML:** Optimization of ML models for performance and memory efficiency on embedded devices.

9.2 Training the Model in TensorFlow

9.2.1 Preparing the Data

Labeled audio clips containing the words *Yes* and *No* are required for training a model. A pre-recorded audio dataset, available in WAV format, will be used in this example. These audio files are first loaded, processed to extract MFCC features, and splitted into training and test sets.

```
import tensorflow as tf
import librosa
import numpy as np
import os

# Extract MFCC features from an audio file
def extract_mfcc(file_path):
    y, sr = librosa.load(file_path, sr=None)  # Load the audio file
    mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)  # Extract MFCCs
    return np.mean(mfcc, axis=1)  # Return the average of the MFCCs


# Dataset preparation
def prepare_dataset(audio_dir):
    features = []
    labels = []
    for label in ['yes', 'no']:
        for file in os.listdir(os.path.join(audio_dir, label)):
            if file.endswith('.wav'):
                file_path = os.path.join(audio_dir, label, file)
                mfcc_features = extract_mfcc(file_path)
                features.append(mfcc_features)
                labels.append(0 if label == 'no' else 1)  # 0 for "No", 1 for "Yes"
    return np.array(features), np.array(labels)

X, y = prepare_dataset('data/audio')

# Splitting dataset into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

9.2.2 Training the Neural Network Model

A neural network is defined for binary classification of audio features.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(13,)),  # 13 MFCC features
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Sigmoid for binary classification

])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

# Save the trained model
model.save('yes_no_model.h5')
```

9.3 Converting the Model for MCU Deployment

The trained TensorFlow model is converted into TensorFlow Lite (TFLite) format for efficient deployment on resource-constrained devices.

```
model = tf.keras.models.load_model('yes_no_model.h5')
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the TensorFlow Lite model
with open('yes_no_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

The conversion produces a '.tflite' file suitable for embedded deployment.

9.4 Implementing the Model on the EFR32xG24

The TensorFlow Lite model is integrated into the EFR32xG24 environment using the appropriate software development kit (SDK) and TensorFlow Lite for Microcontrollers library.

9.4.1 Setting Up the Development Environment

The following components are required for setting up the development environment:

- **EFR32xG24 SDK:** The latest version of the Silicon Labs Gecko SDK must be installed.
- **TensorFlow Lite for Microcontrollers:** This library should be set up within the development environment.

9.4.2 Loading the Model and Running Inference

The TensorFlow Lite model is loaded onto the MCU, input data is prepared, and inference is performed.

```
#include "tensorflow/lite/c/common.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/model.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/kernels/register.h"

#define INPUT_SIZE 13

// Declare tensors and interpreter
tflite::MicroInterpreter* interpreter;
```

```
tflite::Model* model;
tflite::MicroAllocator* allocator;

float input_data[INPUT_SIZE];  // Input data (MFCCs)
float output_data[1];  // Output data (prediction)


// Load the TensorFlow Lite model
void LoadModel(const uint8_t* model_data) {
    model = tflite::GetModel(model_data);
    tflite::ops::micro::RegisterAllOps();
    tflite::MicroInterpreter interpreter(model, tensor_arena, kTensorArenaSize, &resolver, &allocat
    interpreter.AllocateTensors();
}

// Perform audio classification
int ClassifyAudio(float* mfcc_input) {
    // Copy MFCC data into the input tensor
    memcpy(interpreter.input(0)->data.f, mfcc_input, sizeof(float) * INPUT_SIZE);

    // Perform inference
    interpreter.Invoke();

    // Get the output prediction
    float prediction = interpreter.output(0)->data.f[0];

    // Return classification result: 1 for "Yes", 0 for "No"
    return prediction > 0.5 ? 1 : 0;

}
```

The inference results are interpreted as:

- 1: Detected *Yes*
- 0: Detected *No*

9.4.3 Integrating Audio Capture

An onboard microphone or an external microphone is often used to interface with the MCU to process real-time audio input. The EFR32xG24 does not include a dedicated audio processing block, requiring integration with a microphone module that outputs either analog signals (such as those from an electret microphone) or digital signals (like those from an I2S microphone). For simplicity, an analog microphone with an ADC on the MCU is employed here, with audio signals sampled, preprocessed, and then classified using the following steps:

1. Configure the ADC to sample audio signals.
2. Capture the raw samples from the ADC at a suitable rate (e.g., 16 kHz or 8 kHz, depending on requirements).
3. Preprocess the audio to extract features such as MFCC (Mel-frequency cepstral coefficients), which are suitable for ML models.

4. Feed these features into the model for classification.

Here is an example of how to collect and process audio data using an ADC for feature extraction using the EFR32 SDK.

```
#include "em_device.h"
#include "em_chip.h"
#include "em_adc.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_interrupt.h"

// ADC buffer for storing captured samples
#define BUFFER_SIZE 1024
static uint16_t adc_buffer[BUFFER_SIZE];
static volatile uint32_t adc_index = 0;  // Index for storing samples in the buffer

// ADC interrupt handler to collect samples
void ADC0_IRQHandler(void) {
    // Read the ADC data from the ADC data register
    adc_buffer[adc_index++] = ADC_DataSingleGet(ADC0);

    // If the buffer is full, stop the ADC conversion
    if (adc_index >= BUFFER_SIZE) {
        ADC0->CMD = ADC_CMD_STOP;
        adc_index = 0;
    }
}

// Initialize the ADC for audio sampling
void ADC_InitAudio(void) {
    // Enable the clock for ADC and GPIO
    CMU_ClockEnable(cmuClock_ADC0, true);
    CMU_ClockEnable(cmuClock_GPIO, true);

    // Configure the GPIO pin for the microphone (assuming it is connected to a pin, e.g., PA0)
    GPIO_PinModeSet(gpioPortA, 0, gpioModeInput, 0);

    // Configure the ADC to sample at a reasonable rate for audio (e.g., 16 kHz)
    ADC_Init_TypeDef adcInit = ADC_INIT_DEFAULT;
    adcInit.prescale = ADC_PrescaleCalc(16000, 0);  // Calculate prescaler for 16kHz sampling rate
    ADC_Init(ADC0, &adcInit);

    // Configure the ADC single conversion mode
    ADC_InitSingle_TypeDef adcSingleInit = ADC_INITSINGLE_DEFAULT;
    adcSingleInit.input = adcSingleInputPin0;  // Set the input channel (e.g., PA0)
    adcSingleInit.acqTime = adcAcqTime4;       // Set acquisition time
    ADC_InitSingle(ADC0, &adcSingleInit);
```

```
    // Enable the ADC interrupt and start ADC conversions
    NVIC_EnableIRQ(ADC0_IRQn);
    ADC0->CMD = ADC_CMD_START;
}
```

9.4.4 Audio Preprocessing and Classification

The audio data is stored in an array adc_buffer where the ADC samples are placed at regular intervals following these steps:

- ADC samples the microphone data at a fixed rate.
- The interrupt service routine (ISR) will be triggered each time the ADC completes a conversion.
- The ISR stores the data into the adc_buffer.

Once the raw ADC samples are in the buffer, preprocessing is needed for the ML model. An example is as follows:

- Convert the ADC samples to a window of audio (e.g., 25 ms).
- Apply a Fourier transform to convert the time-domain signal to the frequency domain.
- Extract MFCC features from the frequency-domain signal.

An example is provided on how to use the buffer data to classify using a Tensorflow Lite model.

```
void ProcessAudioAndClassify() {
    // Preprocess the raw ADC samples (simplified; actual MFCC extraction would be more complex)
    float mfcc_input[INPUT_SIZE];  // Assumed 13 MFCC features

    // For simplicity, the ADC data is copied directly into the input array
    // In a real case, processing is required (e.g., via FFT and MFCC extraction)

    float mfcc_input[INPUT_SIZE];
    for (int i = 0; i < INPUT_SIZE; i++) {
        mfcc_input[i] = (float)adc_buffer[i];
    }

    // Run the model to classify the audio
    int prediction = ClassifyAudio(mfcc_input);
    if (prediction == 1) {
        printf("Detected: Yes\n");
    } else {
        printf("Detected: No\n");
    }
}
```

The main loop manages continuous audio capture and classification.

```
int main(void) {
    CHIP_Init();
    ADC_InitAudio();
```

```
    while (1) {
        ProcessAudioAndClassify();
        EMU_EnterEM1();
    }
}
```

9.4.5 Considerations for Optimization

- **ADC Resolution:** The ADC resolution and sampling rate must align with audio requirements.
- **MFCC Extraction:** Complex preprocessing, such as Fourier Transform and MFCC extraction, may require optimizations.
- **Performance:** Model complexity and sampling rates should be adjusted for available memory and processing capabilities.