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.

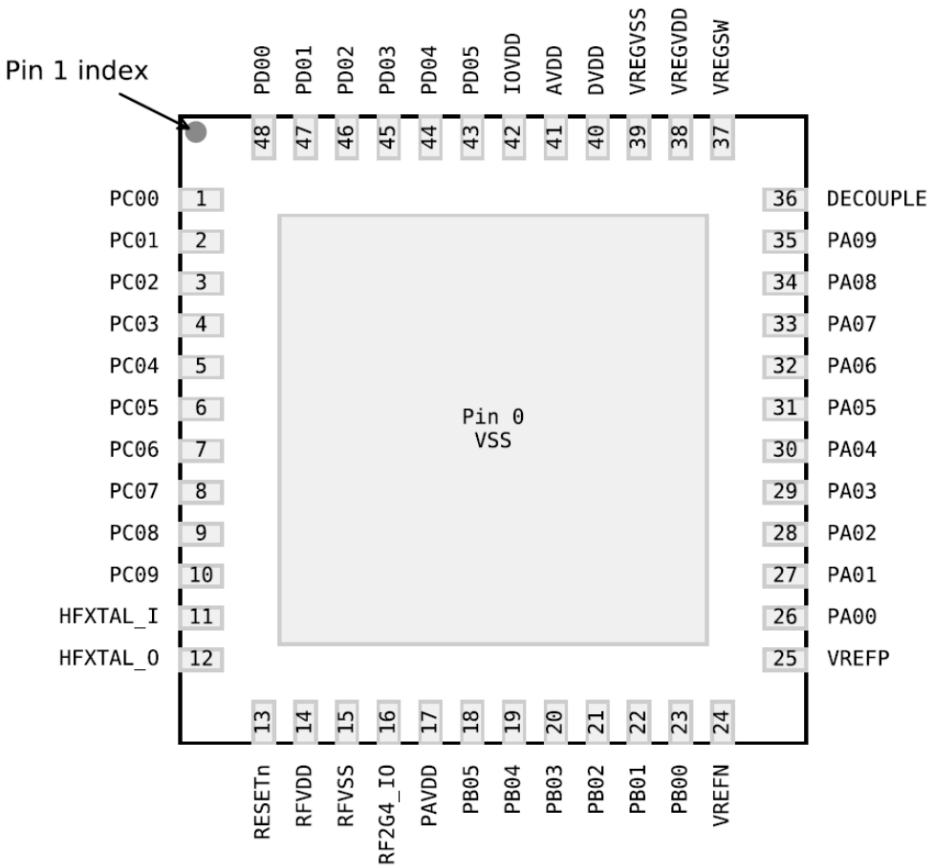


Figure 4.1: Figure 4.1: Pinout of QFN-48 packaged EFR32MG24 microcontroller (EFR32MG24 Datasheet page 107)

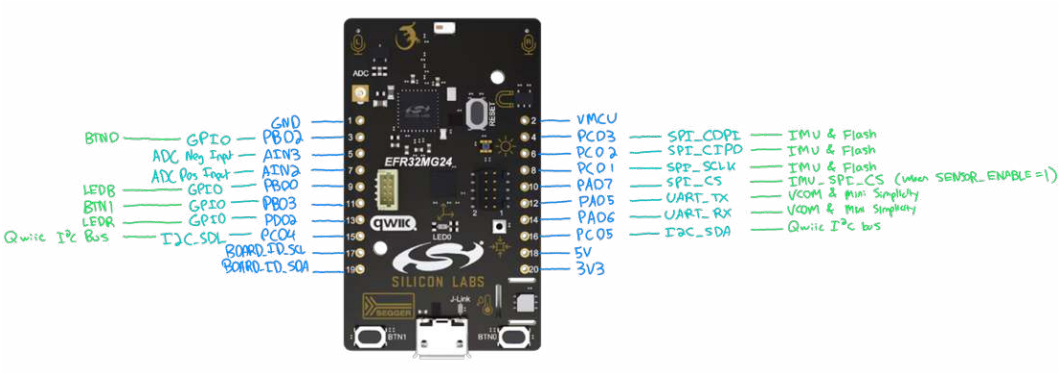


Figure 4.2: Figure 4.2: EFR32XG24 Dev Kit Expansion Header Pinout (UG524 page 19)

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.

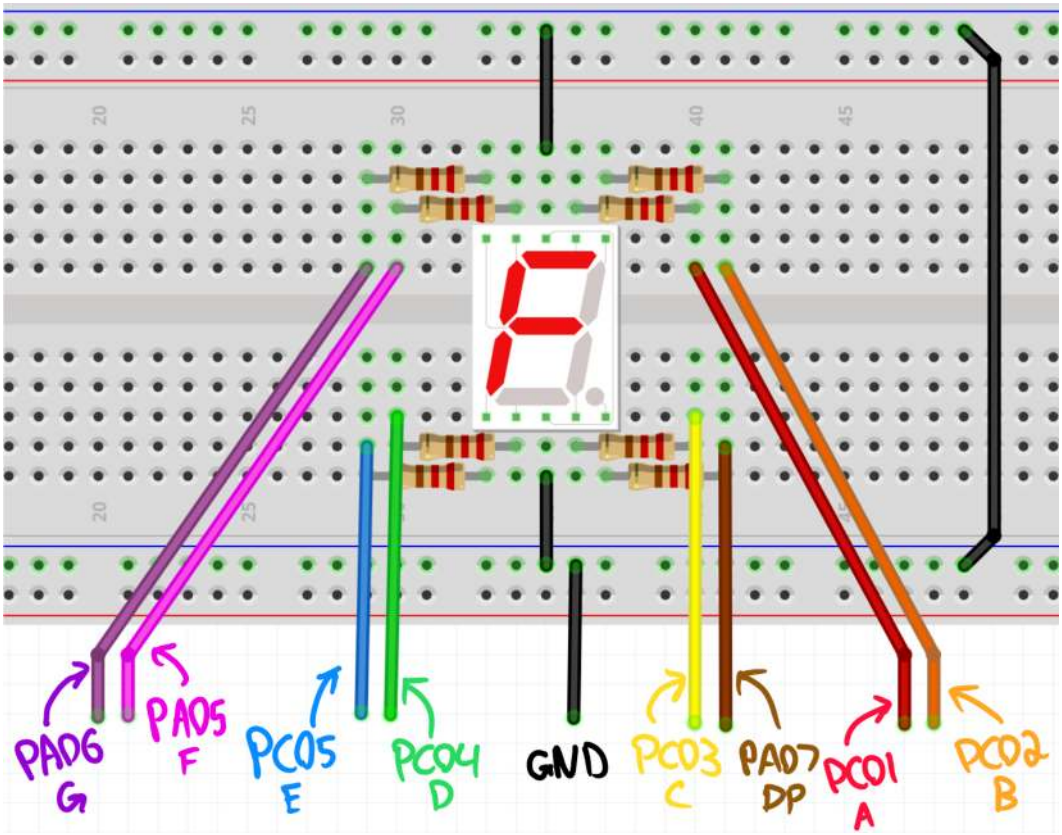


Figure 5.1: Figure 5.1: Wiring diagram for a single common cathode 7-segment display

```
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.

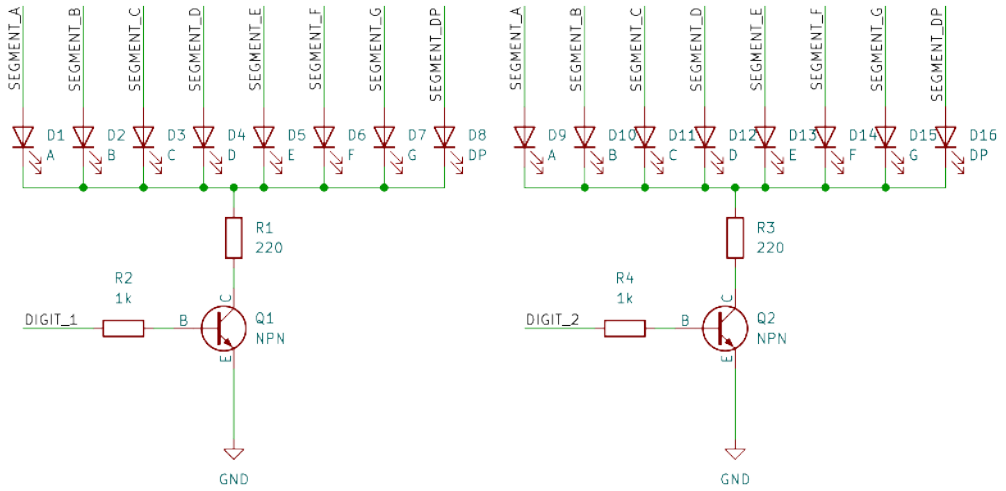


Figure 5.2: Figure 5.2: Two common-cathode 7-segment display digits switches by transistors

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

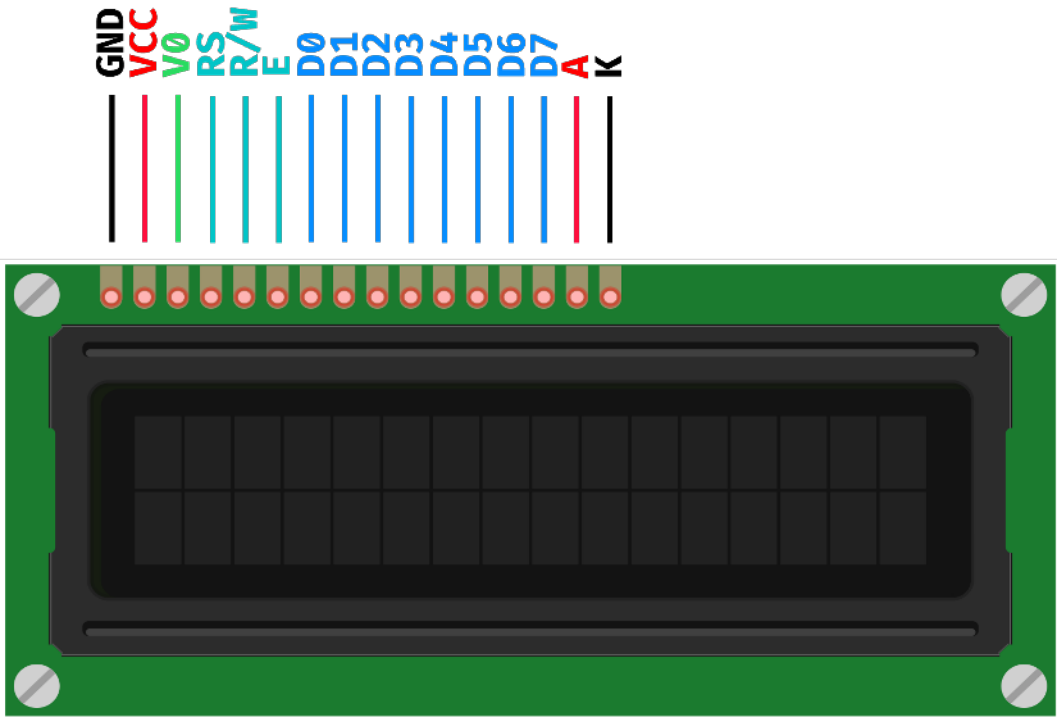


Figure 5.3: Figure 5.3: Pinout for commonly available 16x2 LCD modules

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.

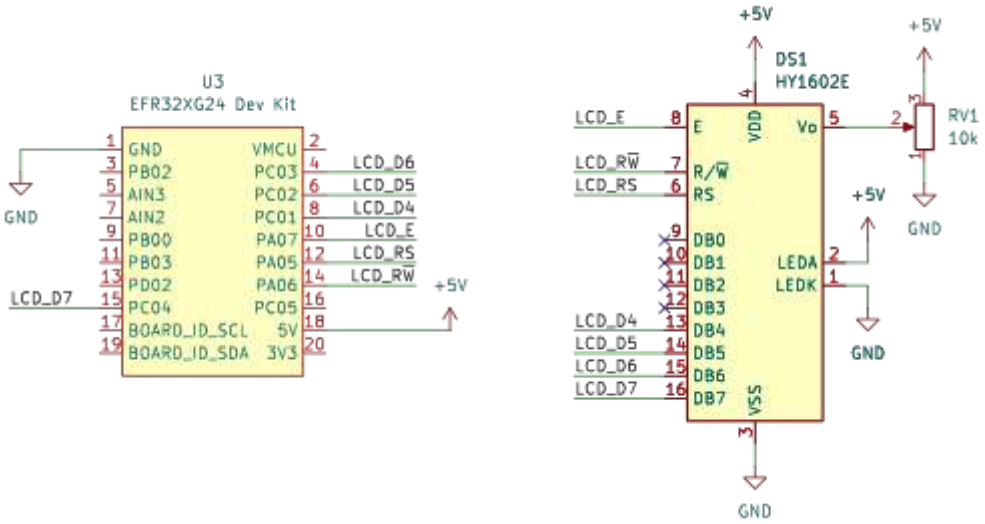


Figure 5.4: Figure 5.4: Schematic diagram of LCD connections to EFR32XG24 Dev Kit

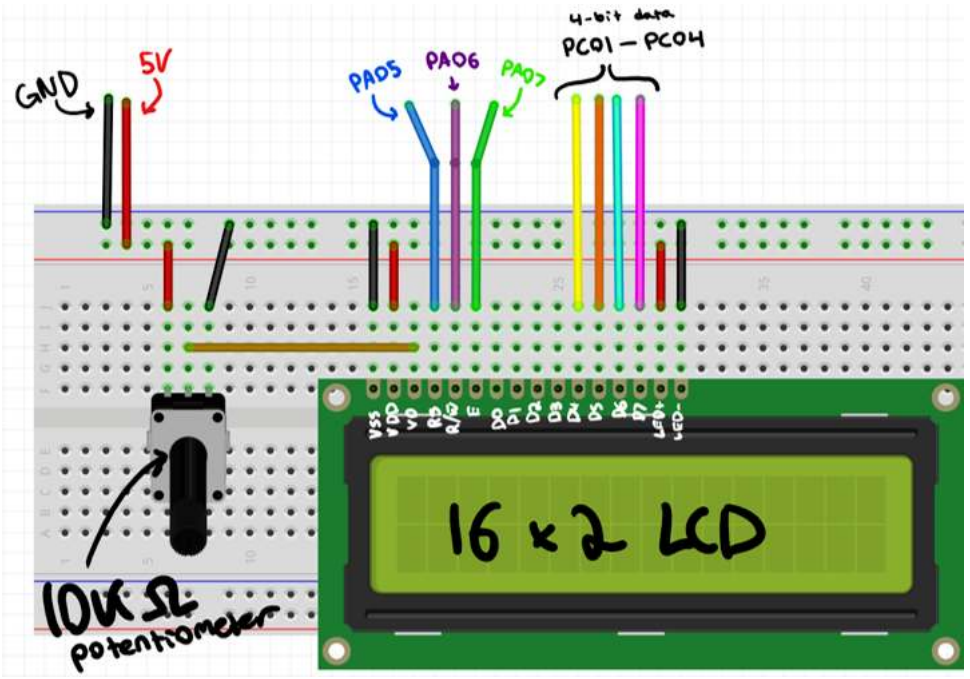


Figure 5.5: Figure 5.5: Wiring diagram for LCD connections to EFR32XG24 Dev Kit

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.

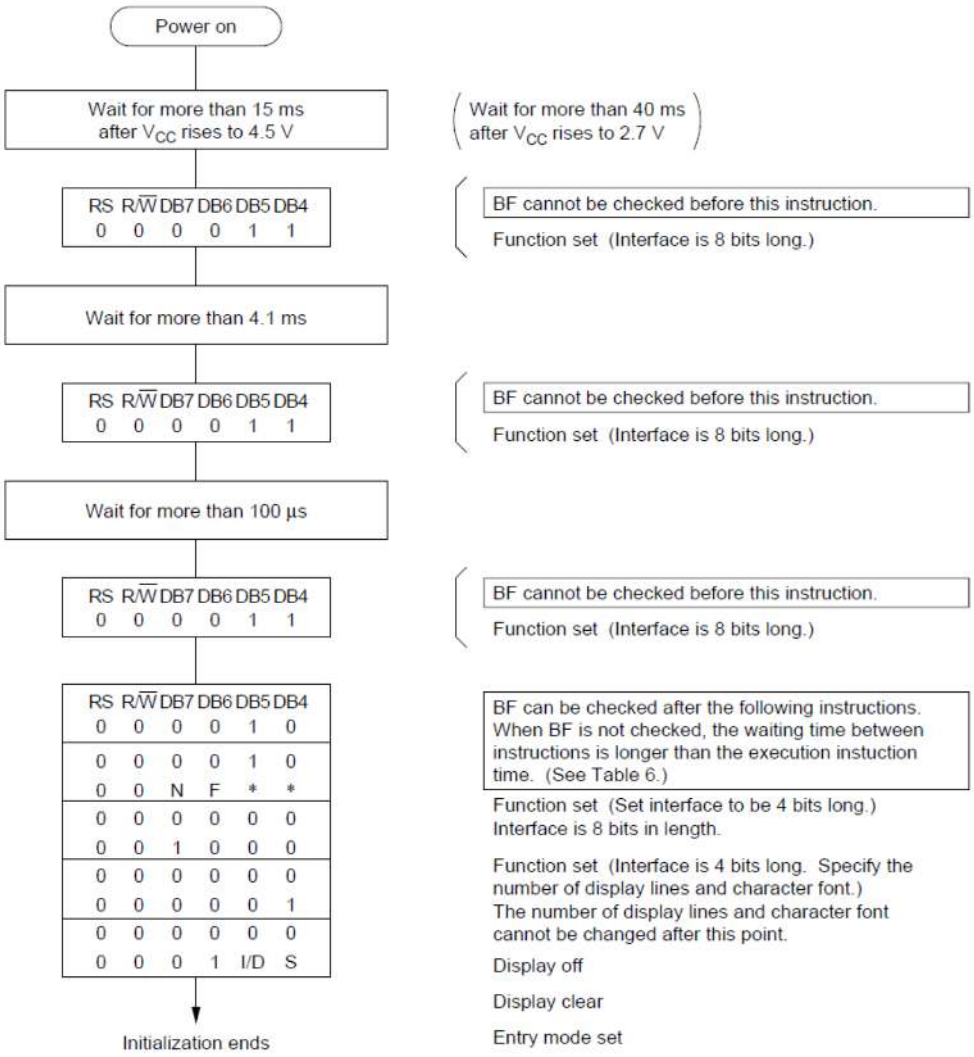


Figure 5.7: Figure 5.7: Block diagram of initialization sequence for HD44780U in 4-bit mode (HD44780U Datasheet page 46)

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.

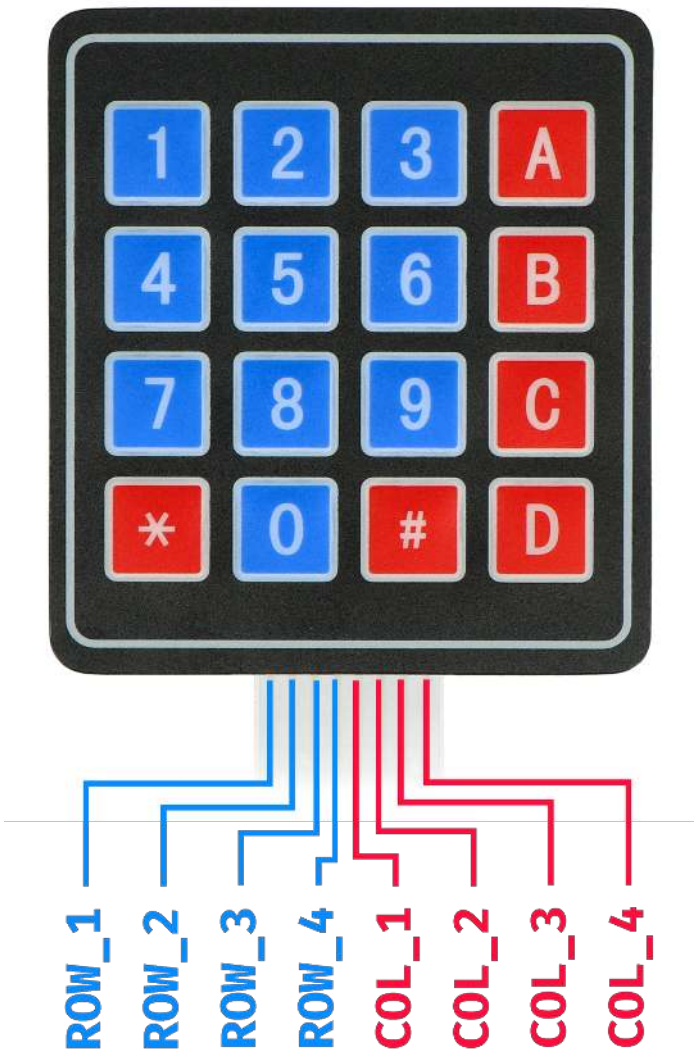


Figure 5.8: Figure 5.8: Pinout for an off-the-shelf keypad

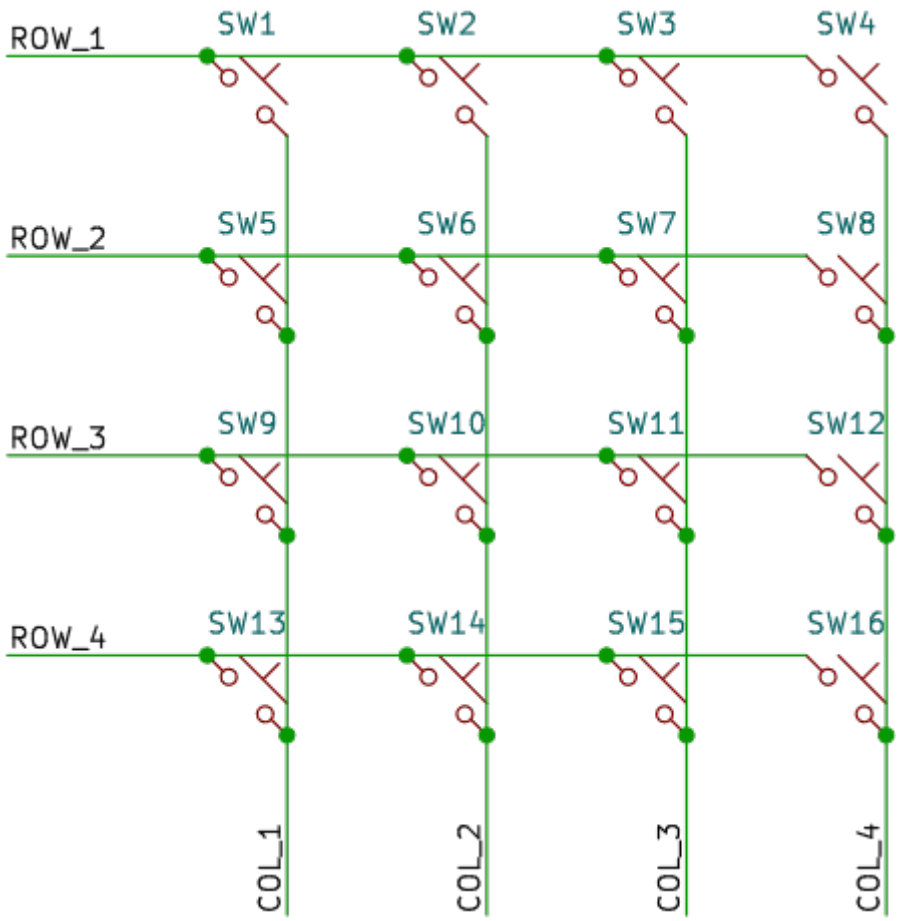


Figure 5.9: Figure 5.9: Schematic diagram of a 4x4 pushbutton switch matrix

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.

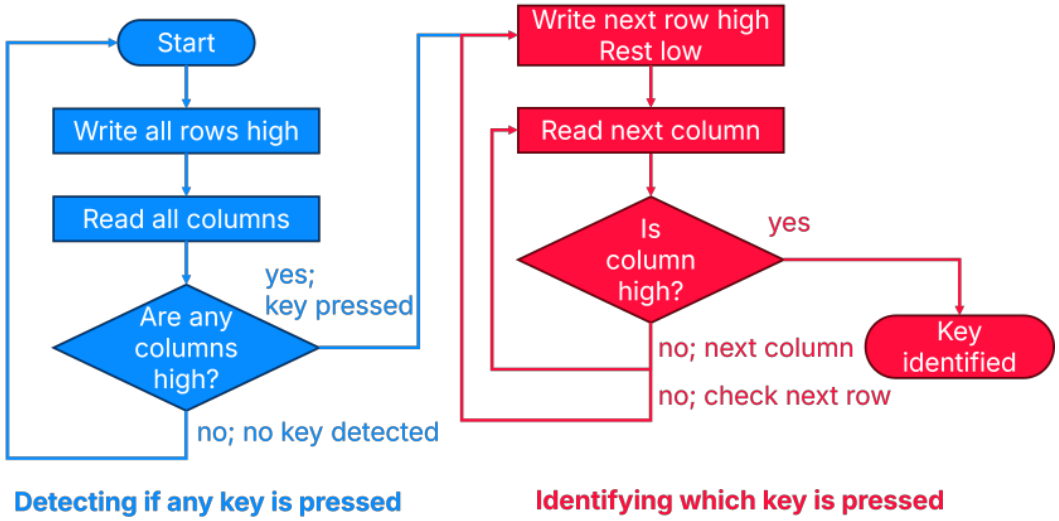


Figure 5.10: Figure 5.10: Flowchart for detecting any keypress, then identifying the speci ic key

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

The Art and Science of Machine Learning

Learning lies at the heart of intelligence, whether natural or artificial. In this chapter, we will embark on a fascinating exploration of the fundamental principles that enable machines to learn from experience. Together, we will examine both the theoretical foundations that provide a rigorous mathematical basis for machine learning, as well as the practical considerations that shape the design and implementation of modern learning systems. Our journey will take us from the historical roots of the field through to the cutting-edge research defining the current state of the art and the open challenges guiding future directions. By the end of this chapter, you will have built a comprehensive understanding of how machines can acquire, represent, and apply knowledge to solve complex problems and enhance decision-making across a wide range of domains.

To make our exploration as engaging and accessible as possible, I will aim to break down complex ideas into more easily digestible parts, building up gradually to the more advanced concepts. Along the way, I will make use of intuitive analogies, illustrative examples, and step-by-step explanations to help illuminate key points. Please feel free to ask questions or share your own insights at any point - learning is an interactive process and your contributions will only enrich our discussion!

With that in mind, let's begin our journey into the art and science of machine learning.

6.1 Origins and Evolution

To fully appreciate the current state and future potential of machine learning, it is helpful to understand its historical context and developmental trajectory. In this section, we will trace the origins of the field and highlight the pivotal advances that have shaped its evolution.

6.1.1 Historical Context

The dream of creating intelligent machines that can learn and adapt has captivated the human imagination for centuries. In mythology and folklore around the world, we find stories of animated beings imbued with 'artificial' intelligence, from the golems of Jewish legends to the mechanical servants of ancient China. These ageless visions speak to a deep fascination with the idea of breathing life and cognizance into inanimate matter.

However, the emergence of machine learning as a scientific discipline is a more recent development, tracing its origins to the mid-20th century. In a profound sense, the birth of machine learning as we know it today arose from the convergence of several key intellectual traditions:

Artificial intelligence - The quest to create machines capable of intelligent behavior

Statistics and probability theory - The mathematical tools for quantifying and reasoning about uncertainty

Optimization and control theory - The principles for automated decision-making and goal-directed behavior

Neuroscience and cognitive psychology - The scientific study of natural learning in biological systems

Each of these tributaries contributed essential ideas and techniques that merged together to form the foundations of modern machine learning.

Some key milestones in the early history of the field:

- *1943* - Warren McCulloch and Walter Pitts publish "A Logical Calculus of the Ideas Immanent in Nervous Activity", laying the groundwork for artificial neural networks
- *1950* - Alan Turing proposes the "Turing Test" in his seminal paper "Computing Machinery and Intelligence", providing an operational definition of machine intelligence
- *1952* - Arthur Samuel writes the first computer learning program, which learned to play checkers better than its creator
- *1957* - Frank Rosenblatt invents the Perceptron, an early prototype of artificial neural networks capable of learning to classify visual patterns
- *1967* - Covering numbers and the Vapnik–Chervonenkis dimension (VC dimension) introduced in the groundbreaking work of Vladimir Vapnik and Alexey Chervonenkis, providing the foundations for statistical learning theory

These pioneering efforts laid the conceptual and technical groundwork for the subsequent decades of research that grew the field into the thriving discipline it is today.

6.1.2 From Rule-Based to Learning Systems

In its early stages, artificial intelligence research focused heavily on symbolic logic and deductive reasoning. The prevailing paradigm was that of "expert systems" - computer programs that encoded human knowledge and expertise in the form of explicit logical rules. A canonical example was MYCIN, a program developed at Stanford University in the early 1970s to assist doctors in diagnosing and treating blood infections. MYCIN's knowledge base contained hundreds of IF-THEN rules obtained by interviewing expert physicians, such as:

```
IF (organism-1 is gram-positive) AND
   (morphology of organism-1 is coccus) AND
   (growth-conformation of organism-1 is chains)
THEN there is suggestive evidence (0.7) that
     the identity of organism-1 is streptococcus
```

By chaining together inferences based on these rules, MYCIN could arrive at diagnostic conclusions and treatment recommendations that rivaled those of human specialists in its domain.

However, the handcrafted knowledge-engineering approach of early expert systems soon ran into serious limitations:

- Knowledge acquisition bottleneck: Extracting and codifying expert knowledge proved to be extremely time-consuming and prone to inconsistencies and biases.
- Brittleness and inflexibility: Rule-based systems struggled to handle noisy data, adapt to novel situations, or keep up with changing knowledge.
- Opaque "black box" reasoning: The complex chains of inference generated by expert systems were often difficult for humans to inspect, understand, and debug.

- Inability to learn from experience: Once programmed, rule-based systems remained static and could not automatically improve their performance or acquire new knowledge.

These shortcomings highlighted the need for a fundamentally different approach - one that could overcome the rigidity and opacity of handcrafted symbolical rules and instead acquire knowledge directly from data.

6.1.3 The Statistical Revolution

The critical shift from rule-based to learning systems was catalyzed by two key insights: Many real-world domains are intrinsically uncertain and subject to noise, necessitating a probabilistic treatment. Expertise is often implicit and intuitive rather than explicit and axiomatic, making it more amenable to statistical extraction than symbolic codification.

Consider again the task of medical diagnosis that systems like MYCIN sought to automate. While it is possible to elicit a set of logical rules from a human expert, there are several complicating factors:

- Patients present with constellations of symptoms that are imperfectly correlated with underlying disorders.
- Diagnostic tests yield results with varying levels of accuracy and associated error rates.
- Diseases evolve over time, manifesting differently at different stages.
- Treatments have uncertain effects that depend on individual patient characteristics.
- New diseases emerge and existing ones change in their prevalence and manifestation over time. In such an environment, definitive logical rules are the exception rather than the norm. Instead, diagnosis is fundamentally a process of probabilistic reasoning under uncertainty, based on a combination of empirical observations and prior knowledge.

The key innovation that unlocked machine learning was to reframe the challenge in statistical terms:

- Instead of trying to manually encode deterministic rules, the goal became to automatically infer probabilistic relationships from observational data.
- Rather than requiring knowledge to be explicitly enumerated, learning algorithms aimed to implicitly extract latent patterns and regularities.
- In place of brittle logical chains, models learned robust statistical associations that could gracefully handle noise and uncertainty.

This shift in perspective opened up a powerful new toolbox of techniques at the intersection of probability theory and optimization. Some key formal developments:

- Maximum likelihood estimation (Ronald Fisher, 1920s): A principled framework for inferring the parameters of statistical models from observed data.
- The perceptron (Frank Rosenblatt, 1957): A simple type of artificial neural network capable of learning to classify linearly separable patterns.
- Stochastic gradient descent (Herbert Robbins & Sutton Monro, 1951): An efficient optimization procedure well-suited to large-scale machine learning problems.
- Backpropagation (multiple independent discoveries, 1970s-1980s): An algorithm for training multi-layer neural networks by propagating errors backwards through the network.
- The VC dimension (Vladimir Vapnik & Alexey Chervonenkis, 1960s-1970s): A measure of the capacity of a hypothesis space that quantifies the conditions for stable learning from finite data.
- Maximum margin classifiers and support vector machines (Vladimir Vapnik et al., 1990s): Powerful discriminative learning algorithms with strong theoretical guarantees.

Together, innovations like these provided the foundations for statistical learning systems that could effectively extract knowledge from raw data. They set the stage for the following decades of progress that would see machine learning mature into one of the most transformative technologies of our time.

6.2 Understanding Learning Systems

Having reviewed the historical context and key conceptual shifts behind the emergence of machine learning, we are now in a position to examine learning systems in greater depth. In this section, we will explore the fundamental principles that define the learning paradigm, the central role played by data, and the nature of the patterns that learning uncovers.

6.2.1 The Learning Paradigm

At its core, machine learning represents a radical departure from traditional programming approaches. To appreciate this, it is helpful to consider how we might go about solving a complex task such as object recognition using classical programming:

First, we would need to sit down and think hard about all the steps involved in identifying objects in images. We might come up with rules like:

- "an eye has a roughly circular shape"
- "a nose is usually located below the eyes and above the mouth"
- "a face is an arrangement of eyes, nose and mouth", etc.

Next, we would translate these insights into specific programmatic instructions:

- "scan the image for circular regions"
- "check if there are two such regions in close horizontal proximity"
- "label these candidate eye regions", etc.

We would then need to painstakingly debug and refine our program to handle all the edge cases and sources of variability we failed to consider initially.

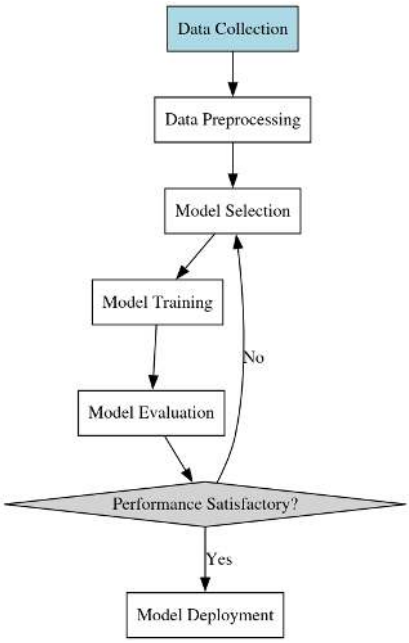
If our program needs to recognize additional object categories, we would have to return to step 1 and repeat the whole arduous process for each new class.

The classical approach places the entire explanatory burden on the human programmer - we start from a blank slate and must explicitly spell out every minute decision and edge case handling routine.

In contrast, the machine learning approach follows a very different recipe:

First, we collect a large dataset of labeled examples (e.g. images paired with the names of the objects they contain). We select a general-purpose model family that we believe has the capacity to capture the relevant patterns (e.g. deep convolutional neural networks for visual recognition). We specify a measure of success (e.g. what fraction of the images are labeled correctly) - this is our objective function.

We feed the dataset to a learning algorithm that automatically adjusts the parameters of the model so as to optimize the objective function on the provided examples. We evaluate the trained model on a separate test set to assess its ability to generalize to new cases.



Notice how the emphasis has shifted:

- Rather than having to explain "how" to solve the task, we provide examples of "what" we want and let the learning algorithm figure out the "how" for us.
- Instead of handcrafting detailed solution steps, we select a flexible model and offload the burden of tuning its parameters to an optimization procedure.
- In place of open-ended debugging, we can run controlled experiments to objectively measure generalization to unseen data.

For a more concrete illustration, consider how we might apply machine learning to the task of spam email classification:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# 1. Collect labeled data
emails, labels = load_email_data()

# 2. Select a model family
vectorizer = CountVectorizer()  # convert email text to word counts
classifier = MultinomialNB()    # naive Bayes with multinomial likelihood

# 3. Specify an objective function
```

```
def objective(model, X, y):
    return accuracy_score(y, model.predict(X))

# 4. Feed data to a learning algorithm
# learn a spam classifier from 70% of the data
train_emails, test_emails, train_labels, test_labels = train_test_split(
        emails, labels, train_size=0.7, stratify=labels)
X_train = vectorizer.fit_transform(train_emails)
classifier.fit(X_train, train_labels)

# 5. Evaluate generalization on held-out test set
X_test = vectorizer.transform(test_emails)
print("Test accuracy:", objective(classifier, X_test, test_labels))
```

This simple example illustrates the key ingredients:

- Data: A collection of example emails, along with human-provided labels indicating whether each one is spam or not.
- Model: The naive Bayes classifier, which specifies the general form of the relationship between the input features (word counts) and output labels (spam or not spam) in terms of probabilistic assumptions.
- Objective: The accuracy metric, which quantifies the quality of predictions made by the model in terms of the fraction of emails that are labeled correctly.
- Learning algorithm: The `fit` method of the classifier, which takes in the training data and finds the model parameters that maximize the likelihood of the observed labels.
- Generalization: The trained model is evaluated not on the data it was trained on, but rather on a separate test set that was held out during training. This provides an unbiased estimate of how well the model generalizes to new, unseen examples.

Of course, this is just a toy example intended to illustrate the basic flow. Real-world applications involve much larger datasets, more complex models, and more challenging prediction tasks. But the fundamental paradigm remains the same: by optimizing an objective function on a sample of training data, learning algorithms can automatically extract useful patterns and knowledge that generalize to novel situations.

6.2.2 Learning from Data

As the spam classification example makes clear, data plays a first-class role in machine learning. Indeed, one of the defining characteristics of the field is its focus on automatically extracting knowledge from empirical observations, rather than relying solely on human-encoded expertise. In this section, we take a closer look at how learning systems leverage data to acquire and refine their knowledge.

To begin, it is useful to clarify what we mean by "data" in a machine learning context. At the most basic level, a dataset is a collection of examples, where each example (also known as a "sample" or "instance") provides a concrete instantiation of the task or phenomenon we wish to learn about. In the spam classification scenario, for instance, each example corresponds to an actual email message, along with a label indicating whether it is spam or not.

More formally, we can think of an example as a pair (x,y), where:

- x is a vector of input features that provide a quantitative representation of the relevant properties of the example. In the case of emails, the features might be counts of various words appearing in the message.
- y is the target output variable that we would like to predict given the input features. For spam classification, y is a binary label, but in general it could be a continuous value (regression), a multi-class label (classification), or a more complex structure like a sequence or image.

A dataset, then, is a collection of n such examples:

$$D = (x_1, y_1), ..., (x_n, y_n)$$

The goal of learning is to use the dataset D to infer a function f that maps from inputs to outputs: $f : X \to Y$ such that $f(x) \approx y$ for future examples (x, y) that were not seen during training.

With this formalism in mind, we can identify several key properties of data that are crucial for effective learning:

- Representativeness: To generalize well, the examples in D should be representative of the distribution of inputs that will be encountered in the real world. If the training data is systematically biased or skewed relative to the actual test distribution, the learned model may fail to perform well on new cases.
- Quantity: In general, more data is better for learning, as it provides a richer sampling of the underlying phenomena and helps the model to avoid overfitting to accidental regularities. The amount of data needed to achieve a desired level of performance depends on the complexity of the task and the expressiveness of the model class.
- Quality: The utility of data for learning can be undermined by issues like noise, outliers, and missing values. Careful data preprocessing, cleaning, and augmentation are often necessary to ensure that the model is able to extract meaningful signal.
- Diversity: For learning to succeed, the training data must contain sufficient variability along the dimensions that are relevant for the task at hand. If all the examples are highly similar, the model may fail to capture the full range of behaviors needed for robust generalization.
- Labeling: In supervised learning tasks, the quality and consistency of the output labels is critical. Noisy, ambiguous, or inconsistent labels can severely degrade the quality of the learned model.

To make these ideas more concrete, let's return to the spam classification example. Consider the following toy dataset:

```
train_emails = [
    "Subject: You won't believe this amazing offer!",
    "Subject: Request for project meeting",
    "Subject: URGENT: Update your information now!",
    "Hey there, just wanted to follow up on our conversation...",
    "Subject: You've been selected for a special promotion!",
]

train_labels = ["spam", "not spam", "spam", "not spam", "spam"]
```

Even without running any learning algorithms, we can identify some potential issues with this dataset:

- Small quantity: Only 5 examples is not enough to learn a robust spam classifier that covers the diversity of real-world emails. With so few examples, the model is likely to overfit to idiosyncratic patterns like the specific subject lines and fail to generalize well.
- Lack of diversity: The examples cover a very narrow range of email types (mainly short subject lines). A more representative sample would include a mix of subject lines, body text, sender information, etc. that better reflect the variability of real emails.
- Label inconsistency: On closer inspection, we might question whether the labeling is fully consistent. For instance, the 4th email seems potentially ambiguous - without more context about the content of the "conversation" it refers to, it's unclear whether it should be classified as spam or not. Inconsistent labeling is a common source of problems in supervised learning.

To address these issues, we would want to collect a much larger and more diverse set of labeled examples. We might also need to do more careful data cleaning and preprocessing, for instance:

- Tokenizing the email text into individual words or n-grams
- Removing stop words, punctuation, and other low-information content
- Stemming or lemmatizing words to collapse related variants
- Normalizing features like word counts to avoid undue influence of message length
- Checking for and resolving inconsistencies or ambiguities in label assignments

In general, high-quality data is essential for successful learning. While it's tempting to focus mainly on the choice of model class and learning algorithm, in practice the quality of the results is often determined by the quality of the data preparation pipeline.

As the saying goes, "garbage in, garbage out"* - if the input data is full of noise, bias, and inconsistencies, no amount of algorithmic sophistication can extract meaningful patterns.*

6.2.3 The Nature of Patterns

Having looked at the role of data in learning, let's now turn our attention to the other central ingredient - the patterns that learning algorithms aim to extract. What exactly do we mean by "patterns" in the context of machine learning, and how do learning systems represent and leverage them?

In the most general sense, a pattern is any regularity or structure that exists in the data and captures some useful information for the task at hand. For instance, in spam classification, some relevant patterns might include:

- Certain words or phrases that are more common in spam messages than in normal emails (e.g. "special offer", "free trial", "no credit check", etc.)
- Unusual formatting or stylistic choices that are suggestive of marketing content (e.g. excessive use of capitalization, colorful text, or images)
- Suspicious sender information, like mismatches between the stated identity and email address, or sending from known spam domains

A key insight of machine learning is that such patterns can be represented and manipulated mathematically, as operations in some formal space. For instance, the presence or absence of specific words can be encoded as a binary vector, with each dimension corresponding to a word in the vocabulary:

```
vocabulary = ["credit", "offer", "special", "trial", "won't", "believe", ...]

def email_to_vector(email):
    vector = [0] * len(vocabulary)
```

```
    for word in email.split():
        if word in vocabulary:
            index = vocabulary.index(word)
            vector[index] = 1
    return vector

# Example usage
message1 = "Subject: You won't believe this amazing offer!"
message2 = "Subject: Request for project meeting"

print(email_to_vector(message1))
# Output: [0, 1, 0, 0, 1, 1, ...]

print(email_to_vector(message2))
# Output: [0, 0, 0, 0, 0, 0, ...]
```

In this simple "bag of words" representation, each email is transformed into a vector that indicates which words from a predefined vocabulary are present in it. Already, some potentially useful patterns start to emerge - notice how the spam message gets mapped to a vector with more non-zero entries, suggesting the presence of marketing language.

Of course, this is a very crude representation that discards a lot of potentially relevant information (word order, punctuation, contextualized meanings, etc.). More sophisticated approaches attempt to preserve additional structure, for instance:

- Using counts or tf-idf weights instead of binary indicators to capture word frequencies
- Extracting n-grams (contiguous sequences of n words) to partially preserve local word order
- Applying techniques like latent semantic analysis or topic modeling to identify thematic structures
- Learning dense vector embeddings that map words and documents to points in a continuous semantic space

What these approaches all have in common is that they define a systematic mapping from the raw data (e.g. natural language text) to some mathematically tractable representation (e.g. vectors in a high-dimensional space). This mapping is where the "learning" in "machine learning" really takes place - by discovering the specific parameters of the mapping that lead to effective performance on the training examples, the learning algorithm implicitly identifies patterns that are useful for the task at hand.

To make this more concrete, let's take a closer look at how a typical supervised learning algorithm actually goes about extracting patterns from data. Recall that the goal is to learn a function $f : X \to Y$ that maps from input features to output labels, such that $f(x) \approx y$ for examples (x, y) drawn from some underlying distribution.

In practice, most learning algorithms work by defining a parametrized function family F_θ and searching for the parameter values θ that minimize the empirical risk (i.e. the average loss) on the training examples:

$$\theta* = argmin_\theta 1/n \sum_i L(F_\theta(x_i), y_i)$$

Here L is a loss function that quantifies the discrepancy between the predicted labels $F_\theta(x_i)$ and the true labels y_i, and the summation ranges over the n examples in the training dataset.

Different learning algorithms are characterized by the specific function families F and loss functions L that they employ, as well as the optimization procedure used to search for $\theta*$. But at a high level, they all aim to find patterns - as captured by the parameters θ - that enable the predictions $F_\theta(x)$ to closely match the actual labels y across the training examples.

Let's make this more vivid by returning to the spam classification example. A very simple model family for this task is logistic regression, which learns a linear function of the input features:

$$F_\theta(x) = \sigma(\theta^T x)$$

Here x is the vector of word-presence features, θ is a vector of real-valued weights, and σ is the logistic sigmoid function that "squashes" the linear combination $\theta^T x$ to a value between 0 and 1 interpretable as the probability that the email is spam.

Coupled with the binary cross-entropy loss, the learning objective becomes:

$$\theta* = argmin_\theta 1/n \sum_i [-y_i log(F_\theta(x_i)) - (1-y_i) log(1-F_\theta(x_i))]$$

where $y_i \in 0, 1$ indicates the true label (spam or not spam) for the ith training example.

Solving this optimization problem via a technique like gradient descent will yield a weight vector $\theta*$ such that:

- Weights for words that are more common in spam messages (like "offer" or "free") will tend to be positive, increasing the predicted probability of spam when those words are present.
- Weights for words that are more common in normal messages (like "meeting" or "project") will tend to be negative, decreasing the predicted probability of spam when those words are present.
- The magnitude of each weight corresponds to how predictive the associated word is of spam vs. non-spam - larger positive weights indicate stronger spam signals, while larger negative weights indicate stronger non-spam signals.

In this way, the learning process automatically discovers the specific patterns of word usage that are most informative for distinguishing spam from non-spam, as summarized in the weights $\theta*$. Furthermore, the learned weights implicitly define a decision boundary in the high-dimensional feature space - emails that fall on one side of this boundary (as determined by the sign of $\theta^T x$) are classified as spam, while those on the other side are classified as non-spam.

This simple example illustrates several key properties that are common to many learning algorithms:

The parameters θ provide a compact summary of the patterns in the data that are relevant for the task at hand. In this case, they capture the correlations between the presence of certain words and the spam/non-spam label.

The learning process is data-driven - the specific values of the weights are determined by the empirical distribution of word frequencies in the training examples, not by any a priori assumptions or hand-coded rules.

The learned patterns are task-specific - the weights are tuned to optimize performance on the particular problem of spam classification, and may not be meaningful or useful for other tasks.

The expressiveness of the learned patterns is limited by the model family - in this case, the assumption of a simple linear relationship between word presence and spam probability. More complex model families (like deep neural networks) can capture richer, more nuanced patterns.

Of course, this is just a toy example intended to illustrate the basic principles. In practice, modern learning systems often employ much higher-dimensional feature spaces, more elaborate model families, and more sophisticated optimization procedures. But the fundamental idea remains the same -

by adjusting the parameters of a flexible model to minimize the empirical risk on a training dataset, learning algorithms can automatically discover patterns that generalize to improve performance on novel examples.

6.3 The Nature of Machine Learning

Having examined the fundamental components of learning systems - the data they learn from and the patterns they aim to extract - we now turn to some higher-level questions about the nature of learning itself. What does it mean for a machine to "learn" in the first place? How does this process differ from other approaches to artificial intelligence? And what challenges and opportunities does the learning paradigm present?

6.3.1 Learning as Induction

At a fundamental level, machine learning can be understood as a form of inductive inference - the process of drawing general conclusions from specific examples. In philosophical terms, this contrasts with deductive inference, which derives specific conclusions from general premises.

Consider a classic example of deductive reasoning:

- All men are mortal. (premise)
- Socrates is a man. (premise)
- Therefore, Socrates is mortal. (conclusion)

Here, the conclusion follows necessarily from the premises - if we accept that all men are mortal and that Socrates is a man, we must also accept that Socrates is mortal. The conclusion is guaranteed to be true if the premises are true.

Inductive reasoning, on the other hand, goes in the opposite direction:

- Socrates is a man and is mortal.
- Plato is a man and is mortal.
- Aristotle is a man and is mortal.
- Therefore, all men are mortal.

Here, the conclusion is not guaranteed to be true, even if all the premises are true - we can never be certain that the next man we encounter will be mortal, no matter how many examples of mortal men we have seen. At best, the conclusion is probable, with a degree of confidence that depends on the number and diversity of examples observed.

Machine learning can be seen as a form of algorithmic induction - instead of a human observer drawing conclusions from examples, we have a learning algorithm that discovers patterns in data and uses them to make predictions about novel cases. Just as with human induction, the conclusions of a machine learning model are never guaranteed to be true, but can be highly probable if the training data is sufficiently representative and the model family is appropriate for the task.

To make this more concrete, let's return to the spam classification example. Recall that our goal is to learn a function f that maps from email features x to spam labels y, such that $f(x) \approx y$ for new examples (x, y) drawn from the same distribution as the training data.

In the logistic regression model we considered earlier, f takes the form:

$$f(x) = \sigma(\theta^T x)$$

where θ is a vector of learned weights and σ is the logistic sigmoid function.

Now, imagine that we train this model on a dataset of 1000 labeled emails, using gradient descent to find the weights $\theta*$ that minimize the average cross-entropy loss on the training examples. We can then apply the learned function f* to classify new emails as spam or not spam:

```
def predict_spam(email, weights):
    features = email_to_vector(email)
    score = weights.dot(features)
    probability = sigmoid(score)
    return probability > 0.5

# Example usage
weights = train_logistic_regression(train_emails, train_labels)

new_email = "Subject: Amazing opportunity to work from home!"
prediction = predict_spam(new_email, weights)

print(prediction)  # Output: True
```

This process is fundamentally inductive:

- We start with a collection of specific examples (the training emails and their labels).
- We use these examples to learn a general rule (the weight vector $\theta*$) for mapping from inputs to outputs.
- We apply this rule to make predictions about new, unseen examples (e.g. classifying the new email as spam).

Just as with human induction, there is no guarantee that the predictions will be correct - the learned rule is only a generalization based on the limited sample of examples in the training data. If the training set is not perfectly representative of the real distribution of emails (which it almost never is), there will necessarily be some errors and edge cases that the model gets wrong.

However, if the inductive reasoning is sound - i.e. if the patterns discovered by the learning algorithm actually capture meaningful regularities in the data - then the model's predictions will be correct more often than not. Furthermore, as we train on larger and more diverse datasets, we can expect the accuracy and robustness of the learned patterns to improve, leading to better generalization performance.

Of course, spam classification is a relatively simple example as far as machine learning tasks go. In more complex domains like computer vision, natural language processing, or strategic decision-making, the input features and output labels can be much higher-dimensional and more abstract, the model families more elaborate and multilayered, and the optimization procedures more intricate and computationally intensive.

However, the fundamental inductive reasoning remains the same:

- Start with a set of training examples that (hopefully) capture relevant patterns and variations.
- Define a suitably expressive model family and objective function.
- Use an optimization algorithm to find the model parameters that minimize the objective on the training set.
- Apply the learned model to predict outputs for new, unseen inputs.
- Evaluate the quality of the predictions and iterate to improve the data, model, and optimization as needed.

The power of this paradigm lies in its generality - by framing the search for patterns as an optimization problem, learning algorithms can be applied to an extremely wide range of domains and tasks without the need for detailed domain-specific knowledge engineering. Given enough data and compute, the same basic approach can be used to learn patterns in images, text, speech, sensor readings, economic trends, user behavior, and countless other types of data.

At the same time, the generality of the paradigm also highlights some of its limitations and challenges:

- Dependence on data quality: The performance of a learning system is fundamentally limited by the quality and representativeness of its training data. If the data is noisy, biased, or incomplete, the learned patterns will reflect those limitations.
- Opacity of learned representations: The patterns discovered by learning algorithms can be highly complex and challenging to interpret. While simpler model families like linear regression produce relatively transparent representations, the internal structure of large neural networks is often inscrutable, making it difficult to understand how they arrive at their predictions.
- Lack of explicit reasoning: Learning systems excel at discovering statistical patterns, but struggle with the kind of explicit, logical reasoning that comes naturally to humans. Tasks that require careful deliberation, causal analysis, or manipulation of symbolic representations can be challenging to frame in purely statistical terms.
- Potential for bias and fairness issues: If the training data reflects societal biases or underrepresents certain groups, the learned models can perpetuate or even amplify those biases in their predictions. Careful auditing and debiasing of data and models is essential to ensure equitable outcomes.

Despite these challenges, the inductive learning paradigm has proven remarkably effective across a wide range of applications. In domains from medical diagnosis and scientific discovery to robotics and autonomous vehicles, machine learning systems are able to match or exceed human performance by discovering patterns that are too subtle or complex for manual specification. As the availability of data and computing power continues to grow, it's likely that the scope and impact of machine learning will only continue to expand.

6.3.2 The Role of Uncertainty

One of the most crucial things to understand about machine learning is that it is, at its core, a fundamentally probabilistic endeavor. When a learning algorithm draws conclusions from data, those conclusions are never absolutely certain, but rather statements of probability based on the patterns in the training examples.

Think back to our spam classification example. Even if our training dataset was very large and diverse, covering a wide range of both spam and legitimate emails, we can never be 100% sure that the patterns it captures will hold for every possible future email. There could always be some new type of spam that looks very different from what we've seen before, or some unusual legitimate email that happens to share many features with typical spam.

What a good machine learning model gives us, then, is not a definite classification, but a probability estimate. When we use logistic regression to predict the "spamminess" of an email, the model's output is a number between 0 and 1 that can be interpreted as the estimated probability that the email is spam, given its input features.

This might seem like a limitation compared to a deterministic rule-based system that always gives a definite yes-or-no answer. However, having a principled way to quantify uncertainty is actually

a key strength of the probabilistic approach. By explicitly representing the confidence of its predictions, a probabilistic model provides valuable information for downstream decision-making and risk assessment.

For instance, consider an email client that uses a spam filter to automatically move suspected spam messages to a separate folder. If the filter is based on a probabilistic model, we can set a confidence threshold for taking this action - say, only move messages with a 95% or higher probability of being spam. This allows us to trade off between false positives (legitimate emails moved to the spam folder) and false negatives (spam emails left in the main inbox) in a principled way.

More generally, having access to well-calibrated probability estimates opens up a range of possibilities for uncertainty-aware decision making, such as:

- Deferring to human judgment for borderline cases where the model is unsure
- Gathering additional information (e.g. asking the user for feedback) to resolve uncertainty
- Hedging decisions to balance risk and reward in the face of uncertain outcomes
- Combining predictions from multiple models to improve overall confidence

Of course, for these benefits to be realized, it's essential that the probability estimates produced by the model are actually well-calibrated - that is, they accurately reflect the true likelihood of the predicted outcomes. If a model consistently predicts 95% confidence for events that only occur 60% of the time, its uncertainty estimates are not reliable.

There are various techniques for quantifying and calibrating uncertainty in machine learning models, including:

- Explicit probability models: Some model families, like Bayesian networks or Gaussian processes, are designed to naturally produce probability distributions over outcomes. By incorporating prior knowledge and explicitly modeling sources of uncertainty, these approaches can provide principled uncertainty estimates.
- Ensemble methods: Techniques like bagging (bootstrap aggregating) and boosting involve training multiple models on different subsets or weightings of the data, then combining their predictions. The variation among the ensemble's predictions provides a measure of uncertainty.
- Calibration methods: Post-hoc calibration techniques like Platt scaling or isotonic regression can be used to adjust the raw confidence scores from a model to better align with empirical probabilities.
- Conformal prediction: A framework for providing guaranteed coverage rates for predictions, based on the assumption that the data are exchangeable. Conformal predictors accompany each prediction with a measure of confidence and a set of possible outcomes.

The importance of quantifying uncertainty goes beyond just improving decision quality - it's also crucial for building trust and promoting responsible use of machine learning systems. When a model accompanies its predictions with meaningful confidence estimates, users can make informed choices about when and how to rely on its outputs. This is especially important in high-stakes domains like healthcare or criminal justice, where the consequences of incorrect predictions can be serious.

Finally, reasoning about uncertainty is also central to more advanced machine learning paradigms like reinforcement learning and active learning:

- In reinforcement learning, an agent learns to make decisions by interacting with an environment and receiving rewards or penalties. Because the environment is often stochastic and the consequences of actions are uncertain, the agent must reason about the expected long-term value of different choices under uncertainty.

- In active learning, a model is allowed to interactively query for labels of unlabeled examples that would be most informative for improving its predictions. Selecting these examples requires estimating the expected reduction in uncertainty from obtaining their labels, based on the model's current state of knowledge.

As we continue to push the boundaries of what machine learning systems can do, the ability to properly quantify and reason about uncertainty will only become more essential. From building robust and reliable models to enabling effective human-AI collaboration, embracing uncertainty is key to unlocking the full potential of machine learning.

6.3.3 Model Complexity and Regularization

Another fundamental challenge in machine learning is striking the right balance between model complexity and generalization performance. On one hand, we want our models to be expressive enough to capture meaningful patterns in the data. On the other hand, we don't want them to overfit to noise or idiosyncrasies of the training set and fail to generalize to new examples.

This tradeoff is commonly known as the bias-variance dilemma:

- **Bias** refers to the error that comes from modeling assumptions and simplifications. A model with high bias makes strong assumptions about the data-generating process, which can lead to underfitting if those assumptions are wrong.
- **Variance** refers to the error that comes from sensitivity to small fluctuations in the training data. A model with high variance can fit the training set very well but may overfit to noise and fail to generalize to unseen examples.

As an analogy, think of trying to fit a curve to a set of scattered data points. A simple linear model has high bias but low variance - it makes the strong assumption that the relationship is linear, which limits its ability to fit complex patterns, but also makes it relatively stable across different subsets of the data. Conversely, a complex high-degree polynomial has low bias but high variance - it can fit the training points extremely well, but may wildly oscillate between them and make very poor predictions on new data.

In general, as we increase the complexity of a model (e.g. by adding more features, increasing the depth of a neural network, or reducing the strength of regularization), we decrease bias but increase variance. The goal is to find the sweet spot where the model is complex enough to capture relevant patterns but not so complex that it overfits to noise.

One way to control model complexity is through the choice of hypothesis space - the set of possible models that the learning algorithm can consider. A simple hypothesis space (like linear functions of the input features) will have low variance but potentially high bias, while a complex hypothesis space (like deep neural networks with millions of parameters) will have low bias but potentially high variance.

Another key tool for managing complexity is *regularization* - techniques for constraining the model's parameters or limiting its capacity to overfit. Some common regularization approaches include:

- **Parameter norm penalties**: Adding a penalty term to the loss function that encourages the model's weights to be small. L2 regularization (also known as weight decay) penalizes the squared Euclidean norm of the weights, while L1 regularization penalizes the absolute values. These penalties discourage the model from relying too heavily on any one feature.
- **Dropout**: Randomly "dropping out" (setting to zero) a fraction of the activations in a neural network during training. This prevents the network from relying too heavily on any one pathway and encourages it to learn redundant representations.

- **Early stopping**: Monitoring the model's performance on a validation set during training and stopping the optimization process when the validation error starts to increase, even if the training error is still decreasing. This prevents the model from overfitting to the training data.

The amount and type of regularization to apply is a key hyperparameter that must be tuned based on the characteristics of the data and the model. Too much regularization can lead to underfitting, while too little can lead to overfitting. Techniques like cross-validation and information criteria can help guide the selection of appropriate regularization settings.

It's worth noting that the bias-variance tradeoff and the role of regularization can vary depending on the amount of training data available. In the "classical" regime where the number of examples is small relative to the number of model parameters, regularization is essential for preventing overfitting. However, in the "modern" regime of very large datasets and overparameterized models (like deep neural networks with millions of parameters), the risk of overfitting is much lower, and the role of regularization is more subtle.

In fact, recent research has suggested that overparameterized models can exhibit "double descent" behavior, where increasing the model complexity beyond the point of interpolating the training data can actually improve generalization performance. This challenges the classical view of the bias-variance tradeoff and suggests that our understanding of model complexity and generalization is still evolving. Despite these nuances, the basic principles of managing model complexity and using regularization to promote generalization remain central to the practice of machine learning. As we train increasingly powerful models on ever-larger datasets, finding the right balance between expressiveness and constrainedness will be key to achieving robust and reliable performance.

6.4 Building Learning Systems

Now that we've explored some of the key theoretical principles behind machine learning, let's turn our attention to the practical considerations involved in building effective learning systems. What are the key components of a successful machine learning pipeline, and how do they fit together?

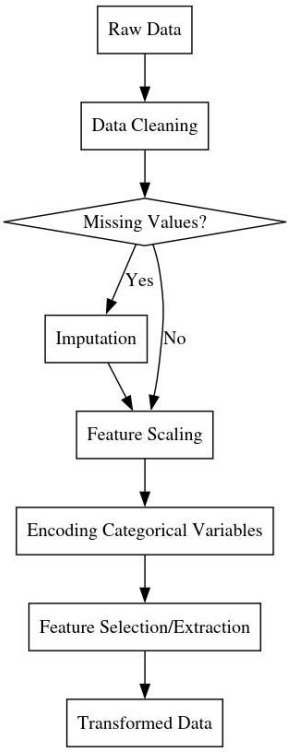
6.4.1 Data Preparation

The first and arguably most important step in any machine learning project is preparing the data. As we saw in Section 6.2.2, the quality and representativeness of the training data is essential for learning meaningful patterns that generalize well to new examples. No amount of algorithmic sophistication can make up for fundamentally flawed or insufficient data.

Key aspects of data preparation include:

- **Data cleaning**: Identifying and correcting errors, inconsistencies, and missing values in the raw data. This can involve steps like removing duplicate records, standardizing formats, and imputing missing values based on statistical patterns.
- **Feature engineering**: Transforming the raw input data into a representation that is more amenable to learning. This can involve steps like normalizing numeric features, encoding categorical variables, extracting domain-specific features, and reducing dimensionality.
- **Data augmentation**: Increasing the size and diversity of the training set by generating new examples through transformations of the existing data. This is especially common in domains like computer vision, where techniques like random cropping, flipping, and color jittering can help improve the robustness of the learned models.
- **Data splitting**: Dividing the data into separate sets for training, validation, and testing. The training set is used to fit the model parameters, the validation set is used to tune hyperparame-

ters and detect overfitting, and the test set is used to evaluate the final performance of the model on unseen data.



The specifics of data preparation will vary depending on the domain and the characteristics of the data, but the general principles of ensuring data quality, representativeness, and suitability for learning are universal. It's often said that data preparation is 80% of the work in machine learning, and while this may be an exaggeration, it underscores the critical importance of getting the data right.

To make this more concrete, let's consider an example of data preparation in the context of a real-world problem. Suppose we're working on a machine learning system to predict housing prices based on features like square footage, number of bedrooms, location, etc. Our raw data might look something like this:

```
Address,Sq.Ft.,Beds,Baths,Price
123 Main St,2000,3,2.5,$500,000
456 Oak Ave,1500,2,1,"$350,000"
789 Elm Rd,1800,3,2,425000
```

To prepare this data for learning, we might perform the following steps:

- Standardize the 'Price' column to remove the "$" and "," characters and convert to a numeric type.
- Impute missing values in the 'Beds' and 'Baths' columns (if any) with the median or most frequent value.

- Normalize the 'Sq.Ft.' column by subtracting the mean and dividing by the standard deviation.
- One-hot encode the 'Address' column into separate binary features for each unique location.
- Split the data into training, validation, and test sets in a stratified fashion to ensure representative price distributions in each split.

The end result might look something like this:

```
Sq.Ft._Norm,Beds,Baths,123_Main_St,456_Oak_Ave,789_Elm_Rd,...,Price
,3,2.5,1,0,0,...,500000
-0.58,2,1,0,1,0,...,350000
,3,2,0,0,1,...,425000
...
```

Of course, this is just a toy example, and in practice the data preparation process can be much more involved. The key point is that investing time and effort into carefully preparing the data is essential for building successful learning systems.

6.4.2 Model Selection and Training

Once the data is prepared, the next step is to select an appropriate model family and training procedure. As we saw in Section 6.3.3, this involves striking a balance between model complexity and generalization ability, often through a combination of cross-validation and regularization techniques.

Some key considerations in model selection include:

- *Inductive biases*: The assumptions and constraints that are built into the model architecture. For example, convolutional neural networks have an inductive bias towards translation invariance and local connectivity, which makes them well-suited for image recognition tasks.
- *Parameter complexity*: The number of learnable parameters in the model, which affects its capacity to fit complex patterns but also its potential to overfit to noise in the training data. Regularization techniques can help control parameter complexity.
- *Computational complexity*: The time and memory requirements for training and inference with the model. More complex models may require specialized hardware (like GPUs) and longer training times, which can be a practical limitation.
- *Interpretability*: The extent to which the learned model can be inspected and understood by humans. In some domains (like healthcare or finance), interpretability may be a key requirement for building trust and ensuring regulatory compliance.

The choice of model family will depend on the nature of the problem and the characteristics of the data. For structured data with clear feature semantics, "shallow" models like linear regression, decision trees, or support vector machines may be appropriate. For unstructured data like images, audio, or text, "deep" models like convolutional or recurrent neural networks are often used.

Once a model family is selected, the next step is to train the model on the prepared data. This typically involves the following steps:

- Instantiate the model with initial parameter values (e.g. random weights for a neural network).
- Define a loss function that measures the discrepancy between the model's predictions and the true labels on the training set.
- Use an optimization algorithm (like stochastic gradient descent) to iteratively update the model parameters to minimize the loss function.

- Monitor the model's performance on the validation set to detect overfitting and tune hyperparameters.
- Stop training when the validation performance plateaus or starts to degrade.

The specifics of the training process will vary depending on the chosen model family and optimization algorithm, but the general goal is to find the model parameters that minimize the empirical risk on the training data while still generalizing well to unseen examples.

To illustrate these ideas, let's continue with our housing price prediction example. Suppose we've decided to use a regularized linear regression model of the form:

$$price = w_0 + w_1 * sqft + w_2 * beds + w_3 * baths + ...$$

where $w_0, w_1, ...$ are the learned weights and `sqft`, `beds`, `baths`, ... are the input features.

We can train this model using gradient descent on the mean squared error loss function:

```
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def gradient_descent(X, y, w, lr=0.01, num_iters=100):
    for i in range(num_iters):
        y_pred = np.dot(X, w)
        error = y_pred - y
        gradient = 2 * np.dot(X.T, error) / len(y)
        w -= lr * gradient
    return w

# Add a bias term to the feature matrix
X = np.c_[np.ones(len(X)), X]

# Initialize weights to zero
w = np.zeros(X.shape[1])

# Train the model
w = gradient_descent(X, y, w)
```

We can also add L2 regularization to the loss function to prevent overfitting:

```
def mse_loss_regularized(y_true, y_pred, w, alpha=0.01):
    return mse_loss(y_true, y_pred) + alpha * np.sum(w**2)

def gradient_descent_regularized(X, y, w, lr=0.01, alpha=0.01, num_iters=100):
    for i in range(num_iters):
        y_pred = np.dot(X, w)
        error = y_pred - y
        gradient = 2 * np.dot(X.T, error) / len(y) + 2 * alpha * w
        w -= lr * gradient
    return w
```

```
# Train the regularized model
w = gradient_descent_regularized(X, y, w)
```

The `alpha` parameter controls the strength of the regularization - larger values will constrain the weights more strongly, while smaller values will allow the model to fit the training data more closely.

By tuning the learning rate `lr`, regularization strength `alpha`, and number of iterations `num_iters`, we can find the model that achieves the best balance between fitting the training data and generalizing to new examples.

Of course, linear regression is just one possible model choice for this problem. We could also experiment with more complex models like decision trees, random forests, or neural networks, each of which would have its own set of hyperparameters to tune. The key is to use a combination of domain knowledge, empirical validation, and iterative refinement to find the model that best suits the problem at hand.

6.4.3 Model Evaluation and Deployment

Once we've trained a model that performs well on the validation set, the final step is to evaluate its performance on the held-out test set. This gives us an unbiased estimate of how well the model is likely to generalize to real-world data.

Common evaluation metrics for *classification problems* include:

- **Accuracy**: The fraction of examples that are correctly classified.
- **Precision**: The fraction of positive predictions that are actually positive.
- **Recall**: The fraction of actual positives that are predicted positive.
- **F1 Score**: The harmonic mean of precision and recall.
- **ROC AUC**: The area under the receiver operating characteristic curve, which measures the trade-off between true positive rate and false positive rate.

For *regression problems*, common metrics include:

- **Mean squared error (MSE)**: The average squared difference between the predicted and actual values.
- **Mean absolute error (MAE)**: The average absolute difference between the predicted and actual values.
- **R-squared (R^2)**: The proportion of variance in the target variable that is predictable from the input features.

It's important to choose an evaluation metric that aligns with the business goals of the problem. For example, in a fraud detection system, we might care more about recall (catching as many fraudulent transactions as possible) than precision (avoiding false alarms), while in a medical diagnosis system, we might care more about precision (avoiding false positives that could lead to unnecessary treatments).

If the model's performance on the test set is satisfactory, we can proceed to deploy it in a production environment. This involves integrating the trained model into a larger software system that can apply it to new input data and surface the predictions to end users.

Some key considerations in model deployment include:

- Scalability: Can the model handle the volume and velocity of data in the production environment? This may require techniques like batch processing, streaming, or distributed computation.

- Latency: How quickly does the model need to generate predictions in order to meet business requirements? This may require optimizations like model compression, quantization, or hardware acceleration.
- Monitoring: How will the model's performance be monitored and maintained over time? This may involve tracking key metrics, detecting data drift, and periodically retraining the model on fresh data.
- Security: How will the model and its predictions be protected from abuse or unauthorized access? This may involve techniques like input validation, output filtering, or access controls.

Deploying and maintaining machine learning models in production is a complex topic that requires close collaboration between data scientists, software engineers, and domain experts. It's an active area of research and development, with new tools and best practices emerging regularly.

To bring everything together, let's return one last time to our housing price prediction example. After training and validating our regularized linear regression model, we can evaluate its performance on the test set:

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Generate predictions on the test set
y_pred = np.dot(X_test, w)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Test MSE: {mse:.2f}")
print(f"Test MAE: {mae:.2f}")
print(f"Test R^2: {r2:.2f}")
```

If we're satisfied with the model's performance, we can deploy it as part of a larger housing price estimation service. This might involve:

- Wrapping the trained model in a web service API that can accept new housing features and return price predictions.
- Integrating the API with a user-facing application that allows homeowners or real estate agents to input property information and receive estimates.
- Setting up a data pipeline to continuously collect new housing data and periodically retrain the model to capture changing market conditions.
- Defining monitoring dashboards and alerts to track the model's performance over time and detect any anomalies or degradations.
- Establishing governance policies and processes for managing the lifecycle of the model, from development to retirement.

Again, this is a simplified example, but it illustrates the end-to-end process of building a machine learning system, from data preparation to model development to deployment and maintenance.

6.5 The Ethics and Governance of Machine Learning

As machine learning systems become more prevalent and powerful, it's crucial that we grapple with the ethical implications of their development and deployment. In this final section, we'll explore some key ethical considerations and governance principles for responsible machine learning.

6.5.1 Fairness and Bias

One of the most pressing ethical challenges in machine learning is ensuring that models are fair and unbiased. If the training data reflects historical biases or discrimination, the resulting model may perpetuate or even amplify those biases in its predictions.

For example, consider a hiring model that is trained on past hiring decisions to predict the likelihood of a candidate being successful in a job. If the training data comes from a company with a history of discriminatory hiring practices, the model may learn to penalize candidates from underrepresented groups, even if those factors are not actually predictive of job performance.

Detecting and mitigating bias in machine learning systems is an active area of research, with techniques like:

- Demographically balancing datasets to ensure equal representation of different groups
- Adversarial debiasing to remove sensitive information from model representations
- Regularization techniques to penalize models that exhibit disparate impact
- Post-processing methods to adjust model outputs to satisfy fairness constraints

However, these techniques are not perfect, and there is often a tradeoff between fairness and accuracy. Moreover, fairness is not a purely technical issue, but a sociotechnical one that requires ongoing collaboration between machine learning practitioners, domain experts, policymakers, and affected communities.

6.5.2 Transparency and Accountability

Another key ethical principle for machine learning is transparency and accountability. As models become more complex and consequential, it becomes harder for humans to understand how they arrive at their predictions and to trace the provenance of their training data and design choices.

This opacity can make it difficult to audit models for bias, safety, or compliance with regulations. It can also make it harder to challenge or appeal decisions made by machine learning systems, leading to a loss of human agency and recourse.

Some techniques for promoting transparency and accountability in machine learning include:

- Model interpretability methods that provide human-understandable explanations of model predictions
- Provenance tracking to document the lineage of data, code, and models used in a system
- Audit trails and version control to enable reproducibility and historical analysis
- Participatory design processes that involve affected stakeholders in the development and governance of models

However, like fairness, transparency is not purely a technical problem. It also requires institutional structures and processes for oversight, redress, and accountability. This might involve things like:

- Designating responsible individuals or teams for the ethical development and deployment of machine learning systems
- Establishing review boards or oversight committees to assess the social impact and governance of models

- Creating channels for affected individuals and communities to provide input and feedback on the use of machine learning in their lives
- Developing legal and regulatory frameworks to enforce transparency and accountability standards

6.5.3 Safety and Robustness

As machine learning systems are deployed in increasingly high-stakes domains, from healthcare to transportation to criminal justice, ensuring their safety and robustness becomes paramount. Models that are brittle, unreliable, or easily fooled can lead to serious harms if they are not carefully designed and tested.

Some key challenges in machine learning safety and robustness include:

- Distributional shift, where models trained on one data distribution may fail unexpectedly when applied to a different distribution
- Adversarial attacks, where malicious actors can craft inputs that fool models into making egregious errors
- Reward hacking, where optimizing for the wrong objective function can lead models to behave in unintended and harmful ways
- Safe exploration, where models need to learn about their environment without taking catastrophic actions

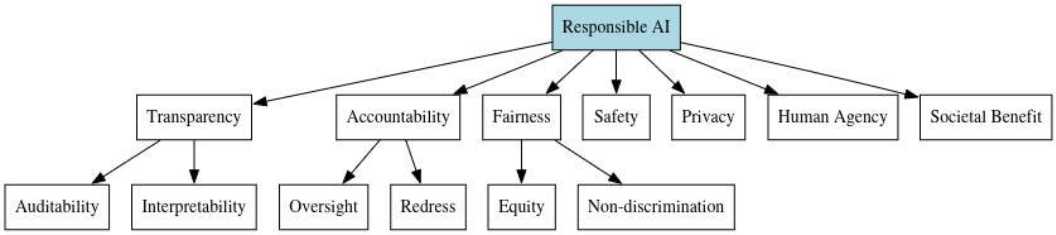
Techniques for improving the safety and robustness of machine learning systems include:

- Anomaly and out-of-distribution detection to flag inputs that are far from the training data
- Adversarial training and robustness regularization to make models more resilient to perturbations
- Constrained optimization and safe reinforcement learning to respect safety boundaries during learning
- Formal verification and testing to provide guarantees about model behavior under different conditions

However, building truly safe and robust machine learning systems requires more than just technical solutions. It also requires:

- Rigorous safety culture and practices throughout the development and deployment lifecycle
- Close collaboration between machine learning practitioners, domain experts, and safety professionals
- Proactive engagement with policymakers and the public to align the development of machine learning with societal values and expectations
- Ongoing monitoring and adjustment of deployed systems to catch and correct errors and unintended consequences

6.5.4 Ethical Principles and Governance Frameworks



To navigate the complex ethical landscape of machine learning, we need clear principles and governance frameworks to guide responsible development and deployment. Some key principles that have been proposed include:

- **Transparency**: Machine learning systems should be auditable and understandable by humans.
- **Accountability**: There should be clear mechanisms for oversight, redress, and enforcement.
- **Fairness**: Machine learning should treat all individuals equitably and avoid discriminatory impacts.
- **Safety**: Machine learning systems should be reliable, robust, and safe throughout their lifecycle.
- **Privacy**: The collection and use of data for machine learning should respect individual privacy rights and provide appropriate protections.
- **Human agency**: Machine learning systems should respect human autonomy and dignity, and provide meaningful opportunities for human input and control.
- **Societal benefit**: The development and deployment of machine learning should be guided by considerations of social good and collective wellbeing.

Translating these high-level principles into practical governance frameworks is an ongoing challenge, but some key elements include:

- Ethical codes of conduct and professional standards for machine learning practitioners
- Impact assessment and risk management processes to identify and mitigate potential harms
- Stakeholder engagement and participatory design to ensure affected communities have a voice
- Regulatory sandboxes and policy experiments to test new governance approaches
- International standards and coordination to address the global nature of machine learning development

Ultimately, the goal of machine learning governance should be to ensure that the technology is developed and deployed in a way that aligns with human values and enhances, rather than undermines, human flourishing. This is a complex and ongoing process that will require sustained collaboration across disciplines, sectors, and geographies.

6.6 Conclusion

In this chapter, we've embarked on a comprehensive exploration of the foundations of machine learning, from its historical roots to its modern techniques to its future challenges. We've seen how the field has evolved from rule-based expert systems to data-driven statistical learning, powered by the

explosion of big data and computing power. We've examined the fundamental components of machine learning systems - the data they learn from, the patterns they aim to extract, and the algorithms that power the learning process. We've discussed key concepts like inductive bias, generalization, overfitting, and regularization, and how they relate to the art of building effective models. We've walked through the practical steps of constructing a machine learning pipeline, from data preparation to model selection to deployment and monitoring. And we've grappled with some of the ethical challenges and governance principles that arise when building systems that can have significant impact on people's lives.

The field of machine learning is still rapidly evolving, with new breakthroughs and challenges emerging every year. As we look to the future, some of the key frontiers and open questions include:

- *Continual and lifelong learning*: How can we build models that can learn continuously and adapt to new tasks and domains over time, without forgetting what they've learned before?
- *Causality and interpretability*: How can we move beyond purely associational patterns to uncover causal relationships and build models that are more interpretable and explainable to humans?
- *Robustness and safety*: How can we guarantee that machine learning systems will behave safely and reliably, even in the face of distributional shift, adversarial attacks, or unexpected situations?
- *Human-AI collaboration*: How can we design machine learning systems that augment and empower human intelligence, rather than replacing or undermining it?
- *Ethical alignment*: How can we ensure that the development and deployment of machine learning aligns with human values and promotes beneficial outcomes for society as a whole?

Advancing machine learning requires collaboration across disciplines—from computer science and statistics to psychology, social science, philosophy, and ethics. It also demands engagement with policymakers, industry leaders, and the public to ensure responsible and inclusive development. The goal is to build systems that learn from experience to make decisions that benefit humanity, whether in healthcare, scientific discovery, or improving daily life.

However, realizing this potential goes beyond technical progress; it requires addressing fairness, accountability, transparency, and safety while navigating ethical and governance challenges. Machine learning practitioners must not only push technological boundaries but also consider the broader impact of their work. By embracing diverse perspectives and collaborating beyond our field, we shape the future of AI. Staying curious, critical, and committed to responsible development will ensure machine learning serves society for generations to come.

Chapter 7

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.

7.1 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();
```

7.1.1 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();
```

7.1.2 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.

7.2 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.

7.2.1 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;
}
```

7.2.2 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.

7.3 AI Model Design for Gesture Recognition

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

7.3.1 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 7.1: Table 6.1: AI Model Layers and Parameters

7.4 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.

7.4.1 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.

7.4.2 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.

7.4.3 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.

7.4.4 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.

7.4.5 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.

7.4.6 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.

7.5 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 8

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`.

8.1 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.

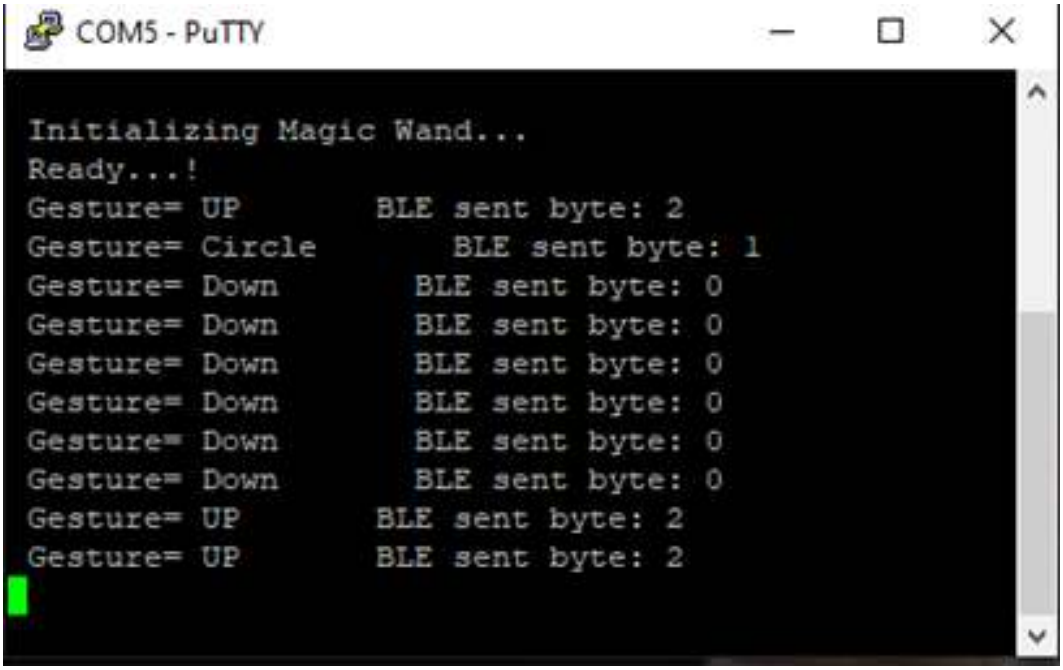


Figure 8.1: Figure 7.1: Sample Data Output via UART Terminal

8.2 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")
```

8.3 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)
```

8.4 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;
  }
}
```

8.5 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 9

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`.

9.1 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.

9.2 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;
```

9.3 HTM Model Architecture

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

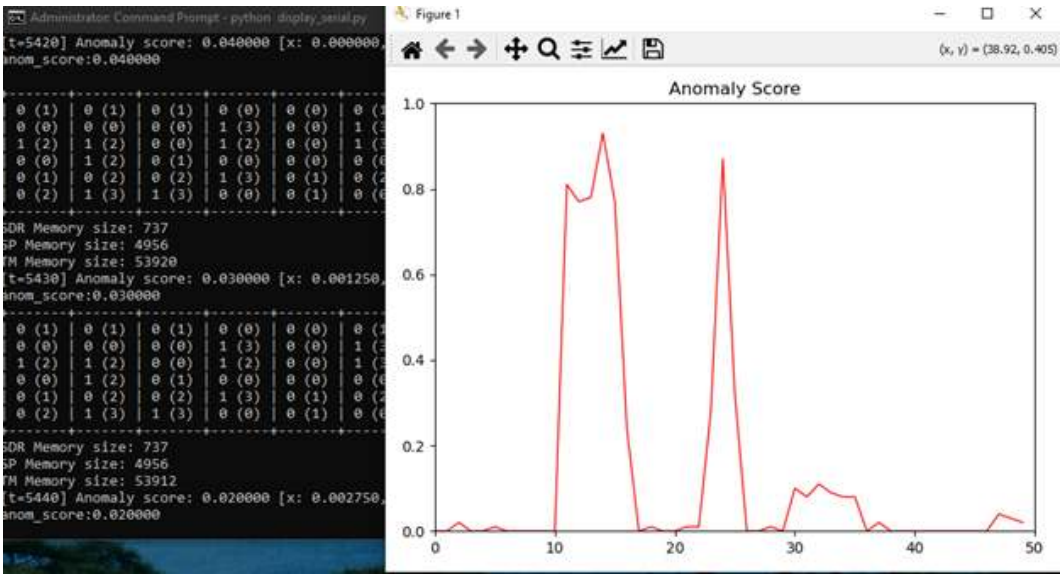


Figure 9.1: Figure 8.1: Example of IMU Anomaly

```
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);
```

9.4 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 10

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.

10.1 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.

10.2 Training the Model in TensorFlow

10.2.1 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)
```

10.2.2 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')
```

10.3 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.

10.4 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.

10.4.1 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.

10.4.2 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*

10.4.3 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;
}
```

10.4.4 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();
    }
}
```

10.4.5 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.