Exploring Fundamental Embedded Software Concepts Through Dedicated Hardware Demonstration

Sven van Schie – 4778758

Table of Contents

Introduction

This document presents a selection of hardware demonstrations focused on embedded software development. Each demonstration explores fundamental concepts in microcontroller-based systems, including digital signal processing, peripheral communication, and efficient hardware control. The projects highlighted here emphasize practical implementation, leveraging key techniques such as pulse-width modulation (PWM), shift registers, and direct port manipulation.

By working through these demonstrations, the goal is to gain hands-on experience in designing and optimizing embedded systems, balancing functionality, efficiency, and hardware constraints. Each demonstration is designed to reinforce essential skills such as memory management, real-time processing, and hardware interfacing, helping learners develop a strong foundation in embedded development.

LED Control System with Adjustable Speed and Brightness

This paragraph outlines the design and implementation of a system for controlling LEDs with adjustable speed and brightness. The demonstration serves as an exploration of key concepts in electronics and programming, focusing on the integration of microcontrollers, shift registers, and ADCs (analogue-digital-converters).

Requirements

To ensure the demonstration meets the goals set out in the introduction, requirements were defined and divided into two categories: functional requirements, which outline the core functionality of the prototype, and hardware requirements, which specify the components and tools needed to achieve this functionality.

Functional requirements

1. *Cycle an LED sequentially through a loop of 8 LEDs.*
   The system should light up one LED at a time in a sequence, creating a looping effect.
2. *Adjust the speed of the LED movement using a potentiometer.*
   The speed of the LED loop should be dynamically adjustable via a potentiometer, allowing for real-time control over how fast the LEDs cycle.
3. *Control the brightness of the LEDs using a potentiometer.*
   A second potentiometer should enable real-time adjustment of the LEDs' brightness, providing control over the visual intensity of the display.

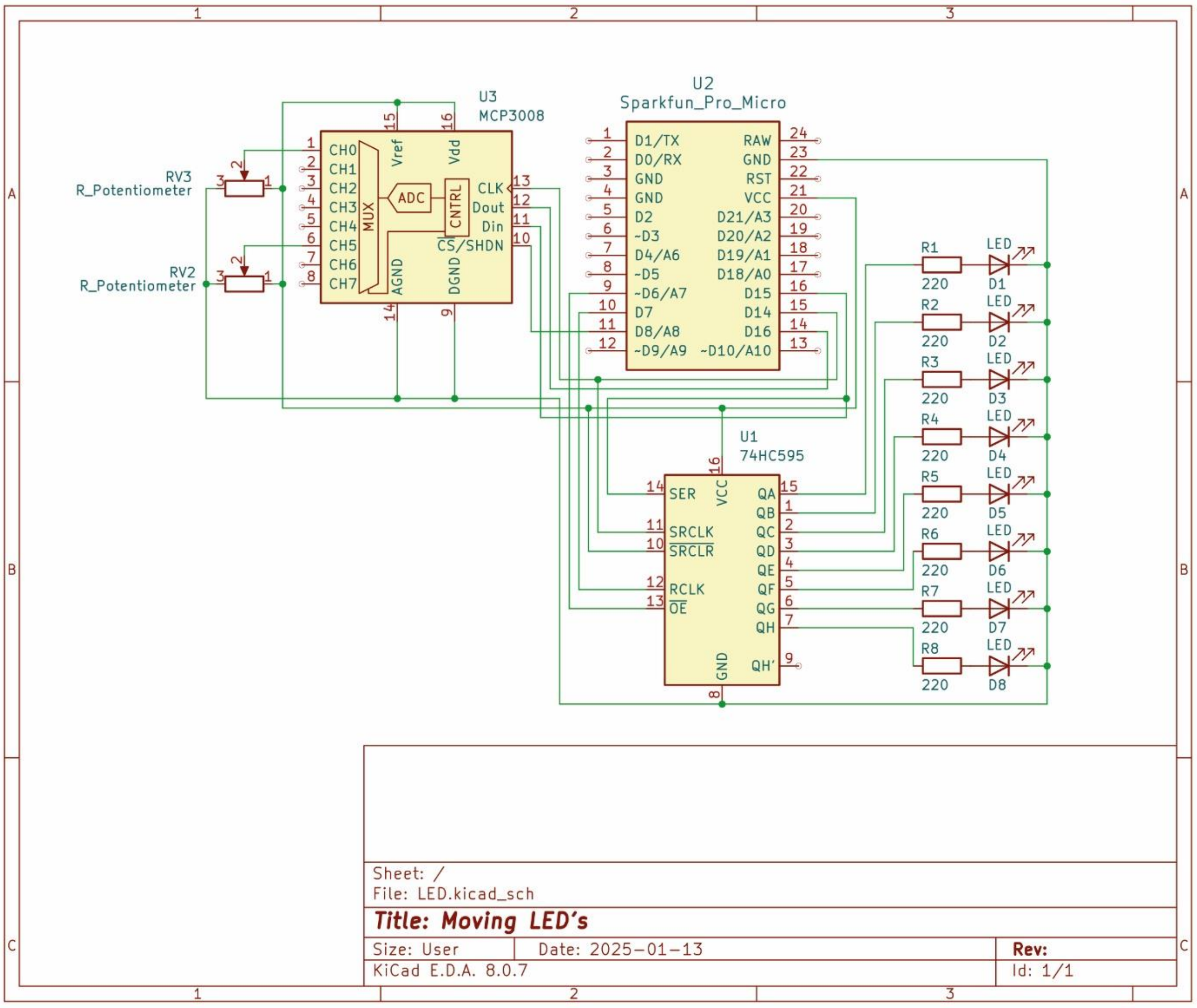
Hardware requirements

1. *Utilize a Pro Micro microcontroller.*
   The Pro Micro serves as the core processing unit, handling input from the potentiometers and controlling the LED sequence. Its compact size and ability to work with a breadboard makes it ideal for this project.
2. *Integrate a 74HC595 shift register for LED control.*
   The 74HC595 shift register expands the number of outputs from the Pro Micro, enabling efficient control of the 8 LEDs using fewer pins.
3. *Use an MCP3008 ADC (Analog-to-Digital Converter) for potentiometer inputs.*
   The MCP3008 is used to read analogue signals from the potentiometers, converting them into digital values that the Pro Micro can process for speed and brightness adjustments.

Design

To translate the requirements into a working prototype, a clear and organized design was essential. A circuit schematic was chosen as it provides a conventional and effective way to present the design. The schematic ensures that all components are properly integrated and allows for easy verification of connections in case any issues arise.

The design features the Spark fun Pro Micro microcontroller as the central unit, processing inputs and controlling outputs. It interfaces with an MCP3008 ADC, which converts analogue signals from two potentiometers: one for adjusting LED speed and the other for brightness into digital values. A 74HC595 shift register is used to efficiently control the 8 LEDs, reducing the number of GPIO pins needed. Each LED is connected via a 220-ohm resistor to limit current and protect the circuit.



Shift register

As specified in the requirements and verified in the design, 8 LEDs must be controlled using a single Pro Micro microcontroller. To extend the number of I/O (Input/Output) ports available for connecting components, a shift register is used. A commonly utilized device for this purpose is the 74HC595, which was chosen for this prototype.

74HC595

The 74HC595 converts a single serial input into 8 parallel outputs. It accomplishes this by receiving data on each positive edge of the shift register's clock signal. The data is then stored temporarily and transferred to the output register on the rising edge of the storage register's clock signal.

In simpler terms, the shift register acts as an 8-bit buffer that allows efficient control of multiple LEDs with minimal GPIO usage. The process can be summarized as follows:

1. Data is shifted into the shift register bit by bit on the positive edge of the clock signal.
2. The data remains stored in a temporary buffer until the latch pin is activated.
3. When the latch pin is set high, the shift register outputs the buffered data to its 8 parallel output pins.

This mechanism ensures that the LEDs are controlled efficiently and synchronously based on the data sent to the shift register.

SIPO vs PISO

The 74HC595 shift register is an example of a SIPO (Serial-In, Parallel-Out) device. This means it accepts data serially (one bit at a time) and outputs it in parallel to multiple pins. In contrast, PISO (Parallel-In, Serial-Out) shift registers work in the opposite direction, receiving data in parallel and outputting it serially.

Pulse-width modulation

To control the brightness of the LEDs, pulse-width modulation (PWM) will be used. PWM is a technique that enables the creation of analogue-like results using a digital I/O pin. Typically, a digital I/O pin outputs either a LOW or HIGH signal, representing binary information. However, to simulate an analogue signal capable of producing a range of values between LOW and HIGH, PWM toggles the signal between these two states at a high frequency.

By adjusting the proportion of time, the signal stays HIGH (the duty cycle) relative to the total cycle time, an average voltage is created. For example, a 50% duty cycle (equal time spent HIGH and LOW) results in an output that appears as half the maximum voltage, effectively dimming the LED. This method provides precise control over the brightness of the LEDs while using simple digital signals.

Since the 74HC595 lacks built-in pulse-width modulation (PWM) capabilities, the effect can be simulated by connecting a PWM-capable I/O pin from the Pro Micro microcontroller to the OE (Output Enable) pin of the 74HC595. The OE pin controls whether the outputs of the shift register are enabled or disabled. By applying a PWM signal to the OE pin, the LEDs connected to the 74HC595 can be dimmed or brightened.

Communication protocols

To read the analogue signals from the potentiometers used for speed and brightness control, an analogue-to-digital converter (ADC) is required. As specified in the requirements, the MCP3008 is used for this purpose. The MCP3008 communicates with the Pro Micro microcontroller via the SPI (Serial Peripheral Interface) protocol.

SPI is a widely used interface bus designed for communication between one controller (formerly called "master") and multiple smaller peripherals such as shift registers, ADCs, and sensors (formerly called "slaves"). It operates in a synchronous manner, using separate lines for data and clock signals to ensure data is transmitted and received in sync.

SPI communication involves the following lines:

1. Clock (SCK): Ensures synchronized data transfer between devices.
2. POCI (Peripherals Out, Controller In): Sends data from the peripherals to the controller.
3. PICO (Peripherals In, Controller Out): Sends data from the controller to the peripherals.
4. Chip Select (CS): Selects which peripheral is active by pulling its CS line LOW.

The Pro Micro microcontroller has dedicated pins for the SPI clock, MOSI, and MISO lines, ensuring seamless integration with the MCP3008. This setup allows the microcontroller to send and receive data efficiently, enabling precise adjustments to the LED speed and brightness.

Besides SPI, multiple other data protocols can be used for transmitting and receiving data. These include, for example, I²C and UART, each with its own advantages and limitations depending on the application. While this project does not explore the inner workings of these protocols in depth, the comparison table below provides a quick overview of their characteristics, advantages, and limitations:

Protocol	Type	Advantages	Limitations
SPI	Synchronous Serial	High speed, full-duplex, multiple devices	Requires more pins, no built-in addressing
I2C	Synchronous Serial	Two-wire communication, built-in addressing	Slower than SPI, half-duplex
UART	Asynchronous Serial	Simple wiring, widely supported	Limited speed, no multi-device support

Each protocol is suited to specific tasks, with SPI being ideal for high-speed, short-distance communication, while I²C and UART are better suited for simpler, lower-speed applications.

In this context, duplex refers to the communication capability of the protocol. A full-duplex protocol, like SPI, allows data to be sent and received simultaneously, enabling faster and more efficient communication. In contrast, a half-duplex protocol, such as I²C, permits data transmission in only one direction at a time, which can limit its speed but simplify implementation.

Synchronous communication occurs when data is transmitted in sync with a clock signal, ensuring coordinated timing between sender and receiver. This results in orderly data exchange but requires both devices to operate at the same rate.

Asynchronous communication, on the other hand, does not rely on a clock signal. Data is transmitted independently, using start and stop bits to mark the beginning and end of each transmission, allowing for more flexibility but requiring additional handling to ensure accuracy.

Analogue to digital conversion

The MCP3008 is an 8-channel, 10-bit analogue-to-digital converter (ADC) designed to bridge the gap between analogue sensors and digital systems. It is commonly used in microcontroller projects to read analogue signals, such as those from potentiometers, temperature sensors, or light-dependent resistors (LDRs), and convert them into digital values for processing. Its 10-bit resolution allows it to represent analogue input values as digital outputs ranging from 0 to 1023, providing a high degree of precision for applications where accuracy is critical.

The MCP3008 operates using the SPI data transfer protocol and follows these steps:

1. A start bit is sent to signal the MCP3008.
2. Configuration bits are transmitted, specifying the mode (single-ended or differential) and the desired channel.
3. The MCP3008 performs the analogue-to-digital conversion.
4. The resulting 10-bit digital value is sent back to the master via the POCI line.

By using the MCP3008, multiple values can be read from potentiometers, enabling precise control of speed and brightness.

Design and Implementation of a Button-Controlled LED Switch System

This paragraph explores the process of designing and implementing a basic system to control an LED using a button. By examining core principles in electronics and programming, the project emphasizes precise control and introduces the concept of direct port manipulation to enhance understanding.

Requirements

To align the demonstration with the objectives outlined in the introduction, requirements were established and categorized into two groups: functional requirements, which detail the essential functions of the prototype, and hardware requirements, which list the components and tools necessary to implement these functions.

Functional requirements

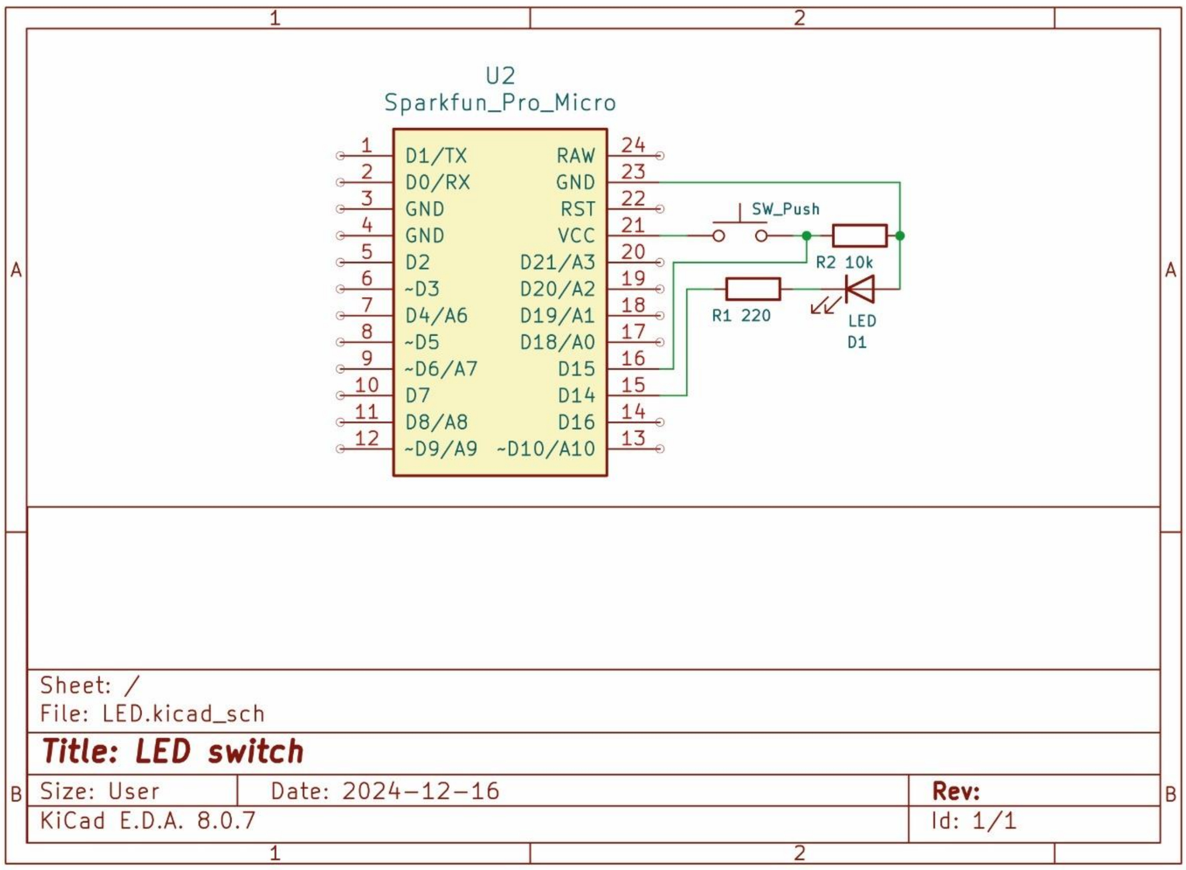
1. *Toggle an LED On and Off*
   The system must allow the LED to be toggled between on and off states by pressing a button.

Hardware requirements

1. *Utilize a Pro Micro microcontroller.*
   The Pro Micro serves as the core of the system, processing input from the button and controlling the LED. Its compact design and compatibility with a breadboard make it an ideal choice for this project.

Design

A clear and well-structured design is essential to ensure the system functions as intended and to simplify troubleshooting and future modifications. The design of this project centres around the integration of a Spark Fun Pro Micro microcontroller, a push button, and an LED, as shown in the schematic. The system incorporates a 10kΩ pull-up resistor to ensure reliable input from the button and a 220Ω resistor to limit current flow to the LED, preventing damage to the components.



Port manipulation and the Arduino library

To maximize understanding of a small prototype like this, a smaller goal was set: fully understanding the role of the Arduino library in the developer experience. A library is a collection of pre-written code that simplifies complex or repetitive tasks for developers. The Arduino library, for example, adds basic functionality such as the digitalWrite and digitalRead functions. These are used to control the state of the Arduino's input/output (IO) ports, setting them to LOW or HIGH when configured as outputs, and reading their values when configured as inputs.

While these functions simplify Arduino programming, they can introduce unnecessary overhead in some cases. For simple programs like this, the overhead is negligible. However, understanding these additional steps is valuable, as they could become problematic in highly time-sensitive systems.

Take digitalWrite as an example: one of its steps is disabling pulse-width modulation (PWM) on a pin before setting its value to HIGH or LOW. While this improves safety, it can reduce execution speed. Developers who are aware of these trade-offs can avoid such overhead when needed, balancing safety and performance based on the use case.

Port registers

Microcontrollers like the Arduino use a series of digital IO pins grouped into ports, labelled as Port A, Port B, Port C, etc. Each port contains several pins, which are numbered consecutively (e.g., Port B could have pins PB0, PB1, PB2, etc.). These ports can be directly controlled using specific registers, providing a more efficient way to handle multiple pins at once compared to using individual functions like digitalWrite or digitalRead.

Each port has a corresponding data direction register (DDR) that controls whether the pins are configured as inputs or outputs, and a port register (PORT) that controls the output state (HIGH or LOW) of the pins. There's also a pin register (PIN) that can be used to read the input state of the pins.

Bit manipulation

Bit manipulation allows you to access or modify the individual bits within the registers controlling the Arduino IO pins. This is done using bitwise operators, such as | (OR), & (AND), and ~ (NOT), combined with bit shifts.

These operators let you selectively set, clear, or toggle specific bits without affecting the other bits in a register. By manipulating bits directly, you gain fine-grained control over the state of the microcontrollers IO pins. This enables the efficient use of resources and faster execution in certain applications. While this approach requires a deeper understanding of the hardware, it can significantly enhance performance in time-sensitive or resource-constrained scenarios.

Debounce

Due to the mechanical and physical limitations of a simple pushbutton, the open/close transition can cause the button to register multiple presses in a short time, resulting in flickering of the LED and preventing the desired on-off effect. By checking the button state twice within a very short time interval, it can be ensured that the button press has actually been registered. This process is called debouncing.

The debounce is used by setting a timer to wait for a brief period after the initial button press or release before checking the button state again. During this waiting period, any fluctuations caused by the mechanical bounce of the button are ignored. After the timer expires, the button state is checked again, and if it has changed, the action (such as toggling an LED) is executed. By using this timer-based approach, only a stable, intentional button press is considered, preventing multiple triggers from a single press.

Interrupt

In embedded systems, interrupts are used to handle events asynchronously, allowing a microcontroller to respond immediately to external or internal signals without constantly polling for changes. This technique improves efficiency by enabling the system to execute other tasks until an interrupt occurs, at which point the processor temporarily stops its current execution, processes the interrupt, and then resumes normal operation.

In this demonstration, using an interrupt-based approach presents an additional challenge. Instead of polling a button press in a loop, an interrupt service routine (ISR) can be used to trigger a response when the button state changes. This reduces CPU load and ensures faster, more efficient handling of input events.

Key considerations when implementing an interrupt for this system include:

1. Debounce Handling – Since mechanical buttons can produce unintended multiple signals (bouncing), a software or hardware debounce method must be used to ensure a clean transition.
2. Interrupt Priority and Timing – If multiple interrupts exist, their priority levels must be managed to avoid conflicts.
3. Minimizing ISR Execution Time – The ISR should execute quickly to prevent blocking other processes. If complex operations are needed, the ISR should set a flag for the main loop to handle later.

By implementing an interrupt-driven approach for the button-controlled LED system, this demonstration provides a deeper understanding of real-time event handling in embedded software, further optimizing responsiveness and system performance.

I^{2}C and SPI Integration for LED Brightness Control

This paragraph explores LED brightness control through I^{2}C and SPI communication, demonstrating how these protocols expand microcontroller capabilities. By integrating I/O expanders, shift registers, and ADCs, the system enables flexible control over LED behaviour, showcasing the efficiency of I^{2}C and SPI for managing multiple components.

Requirements

To ensure the successful implementation of I^{2}C and SPI for LED brightness control, requirements were defined and categorized. Functional requirements specify the necessary operations of the system, while hardware requirements outline the components and tools needed to achieve these functions.

Functional requirements

1. *Mode switching with button*
   The system must allow the user to switch between two modes, brightness selection mode and LED on/off mode, using a button.
2. *Brightness Adjustment via Potentiometer*
   In brightness selection mode, the user can adjust the desired brightness level using the potentiometer.

Hardware requirements

1. *Utilize a Pro Micro microcontroller.*
   The Pro Micro acts as the central controller, managing communication between components via I^{2}C and SPI. It processes input signals, controls LED brightness, and coordinates data transfer. Its compact size and breadboard compatibility make it well-suited for prototyping.
2. *Utilize a MCP23017*
   The MCP23017 extends the number of available GPIO pins using I^{2}C communication. It allows control of multiple LEDs or buttons without occupying additional microcontroller pins, improving scalability.
3. *Utilize a MCP3008*
   The MCP3008 converts analogue signals from sensors or potentiometers into digital values using SPI. This enables precise adjustments to LED brightness based on external inputs.
4. *Utilize a 74HC595*
   The 74HC595 expands the number of digital output pins using SPI. It is used to drive multiple LEDs efficiently, reducing the number of direct connections required on the microcontroller.

Design

Work in progress.

I ^{2}C

Part of this demonstration was gaining a deeper understanding of the I^{2}C data protocol. By connecting a button to the MCP23017 and communicating with it via I^{2}C, a better understanding of its practical workings could be formed. Below is a detailed analysis of the protocol.

I^{2}C (Inter-Integrated Circuit) is a synchronous, multi-master, multi-slave serial communication protocol commonly used for low-speed peripherals like sensors, RTCs, and displays. I^{2}C uses only two wires: SDA (Serial Data Line) and SCL (Serial Clock Line).

- SDA is used to transmit data.
- SCL is used to provide the clock signal for synchronization.

Devices on the I^{2}C bus are identified by a unique address. Communication between the controller and peripherals is established by the controller sending the device address. This allows for multiple masters and multiple slaves, unlike SPI, which typically uses one master and multiple slaves. However, I^{2}C is generally slower.

I^{2}C communication typically follows these steps:

1. The controller pulls the SDA line low while the SCL line remains high. This marks the beginning of communication.
2. The controller sends a 7-bit address to identify the peripheral, along with a read/write bit.
3. After the address and read/write bit, the peripheral acknowledges the request by pulling the SDA line low (ACK).
4. In a write operation, the controller sends the data byte by byte, with the peripheral acknowledging each byte after it's received. The reverse happens in a read operation, with the peripheral sending data byte by byte and the controller acknowledging each byte.
5. Stop Condition: Once the data transfer is complete, the controller sends a stop condition, which involves releasing both the SDA and SCL lines. This indicates the end of communication.

Through the practical use of I^{2}C with the MCP23017, this demonstration highlights how a simple button press can be used to interact with a microcontroller via I^{2}C. By understanding the protocol’s key concepts, such as device addressing, data transmission, and acknowledgment, the operation of I^{2}C communication can be effectively utilized for various low-speed peripheral devices in embedded systems.

MCP23017 Overview and Functionality

The MCP23017 is a 16-bit I/O expander that interfaces with a microcontroller via the I²C protocol. This component is typically used in embedded systems to increase the number of available GPIO pins, providing 16 additional pins that can be configured as either inputs or outputs. By using I²C, the MCP23017 reduces the complexity of wiring, allowing multiple devices to be connected to a single data bus with unique addresses.

Key Features:

- GPIO Expansion: Provides 16 additional GPIO pins that can be independently configured as inputs or outputs.
- I²C Communication: Communicates using the I²C protocol, allowing multiple devices to share the same bus.
- Pull-up Resistors: Internal pull-up resistors can be enabled or disabled for the GPIO pins, useful when interfacing with switches or other devices.
- Interrupt Handling: Supports interrupt functionality, notifying the microcontroller when a pin state changes. This is useful in applications like real-time monitoring of inputs (e.g., button presses or sensor readings).

Device Registers:

The MCP23017's functionality is managed by various internal registers that control different aspects of the device:

- IODIR: Controls the direction (input or output) of the GPIO pins.
- GPIO: Reads the state of input pins or sets the state of output pins.
- GPPU: Manages the internal pull-up resistors for the GPIO pins.
- INTCON: Enables interrupt functionality, allowing the microcontroller to respond to changes in pin states.

Each of these registers is accessed through a unique address, allowing the microcontroller to configure the MCP23017's behavior for a specific application.

The MCP23017 is a highly versatile I/O expander that is integral for projects requiring additional GPIO pins. By utilizing the I²C protocol and configuring the device through its registers, the MCP23017 can be customized to meet the specific needs of a wide variety of applications.

Conclusion

This document has presented a variety of hardware demonstrations aimed at enhancing understanding of embedded software development, focusing on key principles such as microcontroller-based systems, digital signal processing, peripheral communication, and efficient hardware control. The demonstrations explored practical applications of technologies like pulse-width modulation (PWM), shift registers, and analogue-to-digital conversion, providing valuable hands-on experience in designing and optimizing embedded systems.

Throughout these projects, the integration of different components, such as the Pro Micro microcontroller, shift registers, and ADCs, demonstrated the balance between functionality, efficiency, and hardware constraints that is essential in embedded development. The use of SPI and I^{2}C communication, PWM, and direct port manipulation in systems highlighted the importance of understanding both the hardware and software aspects of system design.

Looking ahead, the knowledge gained from these projects can be expanded upon through further experimentation and additional projects. As new hardware and software challenges are introduced, the ability to design, test, and refine embedded systems will grow, opening doors to more complex applications and innovations. The lessons learned here will serve as valuable stepping stones toward mastering embedded systems development and applying these skills in real-world scenarios.