# Project 1: Microcontroller Programming - GPIO, Timers & Interrupts

**Author:** Manus AI **Date:** December 28, 2025 **Target Microcontroller:** ATmega328P (Arduino Uno)

## 1. Introduction

The objective of this project is to gain a fundamental understanding of embedded C programming and **microcontroller register-level manipulation** by implementing core functionalities: General Purpose Input/Output (GPIO), Timers, and Interrupts. The project utilizes the **ATmega328P** microcontroller, the core of the Arduino Uno platform, allowing for direct interaction with hardware registers to control peripherals. This approach provides a deeper insight into the microcontroller's architecture compared to using high-level Arduino functions.

## 2. Methodology

All tasks were implemented using pure embedded C, directly manipulating the microcontroller's registers. The ATmega328P's datasheet was the primary reference for configuring the Data Direction Registers (**DDRx**), Port Registers (**PORTx**), Pin Input Registers (**PINx**), Timer/Counter Control Registers (**TCCRx**), and Interrupt Control Registers (**EICRA**, **EIMSK**, **TIMSKx**).

## 3. Section A: GPIO Programming

This section focuses on basic input and output operations, demonstrating both software and hardware-assisted methods for timing.

### 3.1. `led_blink_delay.c`: LED Blink using Software Delay

This task implements a simple LED blink using a **software-based delay** function, `_delay_ms()`, which is a blocking function that consumes CPU cycles while waiting.

| Register | Configuration | Purpose |
|---|---|---|
| `DDRB` | `DDRB \|= (1 << PB5)` | Sets Pin 5 of Port B (Digital Pin 13) as an **Output**. |
| `PORTB` | `PORTB ^= (1 << PB5)` | Toggles the state of the LED by using the XOR assignment operator. |

### 3.2. `led_blink_timer.c`: LED Blink using Hardware Timer (Polling)

To avoid the inefficiency of software delays, this task uses a hardware timer (**Timer1**) in a **polling** mechanism to create a non-blocking delay. The CPU is still technically blocked waiting for the flag, but the timing is precise and independent of other code execution time.

| Register | Configuration | Purpose |
|---|---|---|
| `TCCR1B` | `(1 << CS12)` | Sets the clock source with a **256 prescaler**. |
| `TCNT1` | `65536 - 31250` | Preloads the counter to achieve a 500ms delay. |
| `TIFR1` | `(1 << TOV1)` | Checks the **Timer Overflow Flag** to determine when the delay is complete. |

### 3.3. `button_input_read.c`: Read Button Input

This task demonstrates reading a digital input from a push-button and reflecting its state on an LED. An **internal pull-up resistor** is enabled to simplify the external circuit (the button connects the pin to ground).

| Register | Configuration | Purpose |
|---|---|---|
| `DDRD` | `DDRD &= ~(1 << PD2)` | Sets Pin 2 of Port D (Digital Pin 2) as an **Input**. |
| `PORTD` | `PORTD \|= (1 << PD2)` | Enables the **internal pull-up resistor** for the input pin. |
| `PIND` | `PIND & (1 << PD2)` | Reads the current state of the input pin. |

## 4. Section B: Timer Programming

This section explores advanced timer functionalities, specifically using the timer to generate precise waveforms and to build a custom delay function.

### 4.1. `square_wave_timer.c`: Generate Square Waves

This code uses **Timer1** in **Clear Timer on Compare Match (CTC)** mode to generate a square wave on the output pin **OC1A** (Digital Pin 9). The frequency is controlled by the value loaded into the Output Compare Register (`OCR1A`).

| Frequency (Hz) | `OCR1A` Value |
|---|---|
| 1 Hz | 31249 |
| 2 Hz | 15624 |
| 5 Hz | 6249 |

The formula used for calculation is: $OCR1A = \frac{F_{CPU}}{2 \times Prescaler \times F_{out}} - 1$

### 4.2. `timer_delay.c`: Build `delay_ms()` using Timer Registers

A custom `delay_ms_timer0()` function was implemented using **Timer0** in **CTC mode**. The timer is configured to generate an interrupt (or in this case, a flag) every 1 millisecond, and the function loops for the requested number of milliseconds.

| Parameter | Value | Calculation |
|---|---|---|
| CPU Frequency ($F_{CPU}$) | 16,000,000 Hz | |
| Prescaler | 64 | |
| Target Time | 1 ms | |
| `OCR0A` Value | 249 | $\frac{16{,}000{,}000}{64} \times 0.001 - 1 = 249$ |

## 5. Section C: Interrupts

Interrupts allow the microcontroller to respond to external or internal events without constantly polling, leading to more efficient and responsive code.

### 5.1. `exti_button_toggle.c`: External Interrupt

This task uses **External Interrupt 0 (INT0)**, triggered by a button press, to toggle the LED state. The interrupt is configured to trigger on a **falling edge** (when the button is pressed, pulling the pin from VCC to GND).

| Register | Configuration | Purpose |
|---|---|---|
| `EICRA` | `(1 << ISC01)` | Sets the trigger to **falling edge** for INT0. |
| `EIMSK` | `(1 << INT0)` | **Enables** the external interrupt INT0. |
| `sei()` | N/A | Enables **Global Interrupts**. |
| `ISR(INT0_vect)` | N/A | The Interrupt Service Routine that executes when the button is pressed. |

### 5.2. `timer_interrupt_blink.c`: Timer Interrupt

This task uses a **Timer Compare Match Interrupt** from **Timer1** to achieve a precise, non-blocking LED blink. The main loop is empty, and all timing and toggling logic resides within the Interrupt Service Routine.

| Register | Configuration | Purpose |
|---|---|---|
| `TCCR1B` | `(1 << WGM12)` | Sets Timer1 to **CTC mode**. |
| `TIMSK1` | `(1 << OCIE1A)` | **Enables** the Output Compare Match A Interrupt. |
| `OCR1A` | 7811 | Sets the compare value for a 500ms period (1Hz blink). |
| `ISR(TIMER1_COMPA_vect)` | N/A | The Interrupt Service Routine that toggles the LED every 500ms. |

## 6. Circuit Diagrams

The following diagrams illustrate the basic connections required for the GPIO and Interrupt tasks.

### LED Circuit

LED Circuit Diagram

### Button Input Circuit (with Internal Pull-up)

Button Circuit Diagram

## 7. Conclusion

This project successfully demonstrated the implementation of fundamental microcontroller programming concepts using register-level embedded C. By directly manipulating registers, a deeper understanding of the ATmega328P's hardware was achieved, covering basic GPIO control, precise timing with hardware timers, and efficient event handling using external and internal interrupts. The resulting code is highly optimized and provides a solid foundation for more complex embedded systems development.