Lab Sheet for Week 2

Lab Instructor: **RSP**

Lab 1: Software Delay Implementation for AVR / ATmega328P

Basic AVR I/O Programming

In this lab, students will learn how to use the three 8-bit registers associated with each I/O port of the AVR chip, namely **`DDRx`**, **`PINx`**, and **`PORTx`**, where **`x`** is the port name (e.g., B, C, or D).

1) **`DDRx`** (Data Direction Register): Controls the direction of each pin of the same I/O port.

- 1 = configure pin as output
- 0 = configure pin as input

2) **`PINx`** (Port Input Register): Used to read the logic level on an input pin.

- Writing a 1 to a **`PINx`** bit toggles the corresponding **`PORTx`** output bit.

3) **`PORTx`** (Port Output Register): Used to set the logic level on an output pin.

- When the pin is configured as output: writing to **`PORTx`** sets the output value (HIGH or LOW).
- When the pin is configured as input: writing 1 to **`PORTx`** enables the internal pull-up resistor.

Note: When a pin is configured as an output (i.e., bit *b* of **`DDRx`** = 1), writing a 1 to the corresponding bit in **`PINx`** causes the hardware to toggle the output state of that pin.

Lab Procedure

1. Create, edit, compile and simulate the test code (**main.c** in **Code Listing 1**), using **Wokwi simulator** (***This task must be completed before entering the lab session***).

2. Open the **Arduino IDE**.

- Delete all code in the default **.ino** sketch.
- Create a new tab and add a source file named **main.c.**
- Use the source code provided as an example (see **Code Listing 1**).

3. Test different values of **N**.

- Try values for **N** between **1900** and **2100**, in steps of **50**.

4. Upload the program to the **Arduino Uno** or **Nano** for each value of **N** and measure the high pulse width (in microseconds) of the output signal on the **PB5 pin (Arduino D13 pin)** using a **digital oscilloscope**.

- Record the waveforms displayed on the oscilloscope's screen for inclusion in the lab report.
- Record the measurement results in the table below.

5. Analyze the experimental data (e.g., using **linear regression**) to determine the best value of **N** that results in a correct **1 ms delay** (or as close as possible).

N	Pulse Width (usec)
1900	
1950	
2000	
2050	
2010	

Table 1: Measurement data

6. Verify correctness using the **calibrated N value** by measuring the pulse width for the following test cases using the **oscilloscope**:

- `sw_delay_ms( x ), for x = 1, 10, 100, 1000`

7. Compare the results by replacing `sw_delay_ms(…)` with `_delay_ms(…)` from the **AVR libc** library. Measure the output signal using the oscilloscope.

```
#ifndef F_CPU
#define F_CPU 16000000UL // Define your clock speed (e.g., 16 MHz)
#endif

#include <avr/io.h>      // for DDRB, PINB registers
#include <util/delay.h>  // for the _delay_ms() function

#define N   (1900)       // Number of loops per 1 msec

void sw_delay_ms(uint16_t n) {
    while (n > 0) { // outer loop
        uint16_t loops = N;
        while (loops > 0) { // inner loop
            asm volatile("nop\n nop\n nop\n nop\n");
            loops--;
        }
        n--;
    }
}
int main() {
   DDRB |= (1 << DDB5); // Set Port B pin 5 (PB5) as output.
   while (1) {
     PINB = (1<<5);     // Toggle the PB5 pin
     sw_delay_ms(1);    // Delay for 1msec
  }
  return 0;
}
```

Code Listing 1: AVR C code for **main.c** in Lab 1

Note: Defining the F_CPU macro is **essential** because the AVR libc delay functions such as `_delay_ms()` and `_delay_us()` use this value to calculate accurate timing.

AVR C Code Simulation with Wokwi

Wokwi (https://wokwi.com/) is an online hardware simulator that supports Arduino, AVR microcontrollers, ESP32, and many common electronic components. It allows users to build virtual circuits, write firmware, and observe the system's behavior directly in a web browser.

Wokwi is especially useful for AVR or Arduino programming because it enables basic **virtual prototyping** without needing physical hardware. Users can test pin outputs, run timing experiments, check logic behavior, and debug their code using built-in tools such as the virtual logic analyzer. This helps students and developers verify their designs early, quickly iterate on code, and understand how the microcontroller interacts with external components—all before moving to real hardware.

Figure 1 shows the Wokwi simulator using an **Arduino Uno board**. During simulation, the output signal from the Arduino Uno can be captured with an **8-channel virtual logic analyzer**. The analyzer generates a waveform data file in VCD format (**wokwi-logic.vcd**), which can be viewed using open-source tools such as **GTKWave** and **Surfer**, as shown in **Figures 2 and 3**. Users can measure parameters such as pulse width by placing vertical cursors (markers) on the waveform.

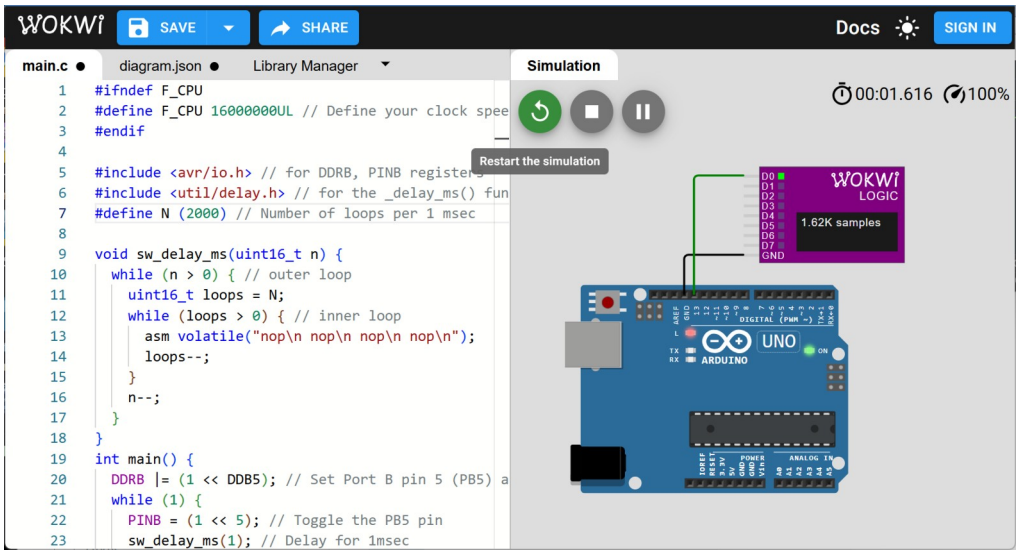

Figure 1: AVR C code simulation with **Wokwi**

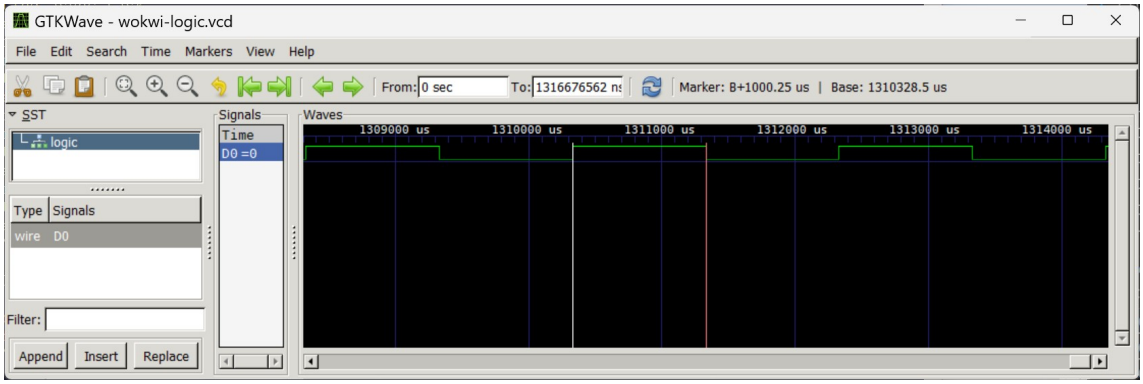

Figure 2: VCD waveform view with **GTKWave**

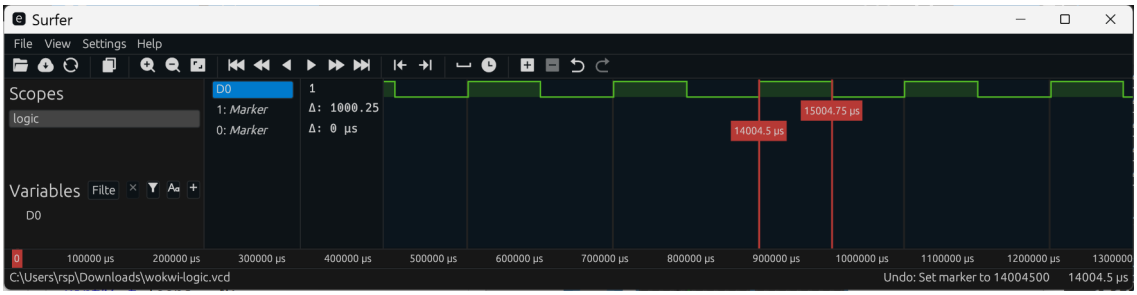

Figure 3: VCD waveform view with **Surfer**

Lab 2: Bare-metal Register-based I/O Programming for AVR / ATmega328P

Lab Procedure

1. Create, edit, compile and simulate the test code (**main.c** in **Code Listing 2**), using **Wokwi simulator** (***This task must be completed before entering the lab session***).

2. Use the **Arduino IDE** to build and upload program to an **Arduino Uno** or **Nano** board during the lab session. Measure the output signal with a **digital oscilloscope**. Capture the waveform for inclusion in the lab report.

3. Record the following results:

- High pulse width (µs) _____
- Low pulse width (µs) _____
- Period (µs) _____
- Frequency (MHz) _____

4. Explain the C statements marked with (1) and (2) in the source code (**Code Listing 2**). What are their functions?

```
#include "avr_io.h"

int main() {
  PortB.DDR->bits.B5 = 1;    // (1) _________________
  while (1) {
    PortB.PIN->bits.B5 = 1;  // (2) _________________
  }
  return 0;
}
```

Code Listing 2: AVR C code for **main.c** in **Steps 1 - 3**

4. Experiment with the following C code (**Code Listing 3**). Measure the signal using a digital oscilloscope and capture the waveform for inclusion in the lab report.

- What is the duty cycle of the output signal?
- Does it produce a signal different from the code in **Step 3**? If yes, explain why.

```
#include "avr_io.h"

int main() {
  PortB.DDR->bits.B5 = 1;     // (1) _________________
  while (1) {
    PortB.PORT->bits.B5 = 1;  // (2) _________________
    PortB.PORT->bits.B5 = 0;  // (3) _________________
  }
  return 0;
}
```

Code Listing 3: AVR C code for **main.c** in **Step 4**

5. Revise the code in **main.c** to use the C functions defined in the C header file "**avr_io.h**" (see: **Code Listing 4**), while keeping the same behavior.

5.1 To configure **PB5** as an output, use one of the following functions:

- `io_set_bit(…)`
- `io_pin_mode(…)`

5.2 To read, set, or clear the output value of the **PB5** pin, use the following functions:

- `io_read_bit(…)`
- `io_clear_bit(…)`
- `io_set_bit(…)`

5.3 To toggle the **PB5** output pin, use one of the following functions:

- `io_toggle(…)`
- `io_set_bit(…)`

```
#ifndef AVR_IO_H
#define AVR_IO_H

#include <avr/io.h>
#include <stdint.h>

typedef union {
    uint8_t REG;
    struct {
        uint8_t B0:1;
        uint8_t B1:1;
        uint8_t B2:1;
        uint8_t B3:1;
        uint8_t B4:1;
        uint8_t B5:1;
        uint8_t B6:1;
        uint8_t B7:1;
    } bits;
} reg8_t;

typedef struct {
    volatile reg8_t *PIN;
    volatile reg8_t *DDR;
    volatile reg8_t *PORT;
} avr_port_t;

#define REG8(addr) ((volatile reg8_t *) (addr))
#define MAKE_AVR_PORT(port_letter) { \
        .PIN  = REG8(&PIN ## port_letter), \
        .DDR  = REG8(&DDR ## port_letter), \
        .PORT = REG8(&PORT ## port_letter) \
    }

avr_port_t PortB = MAKE_AVR_PORT(B);
avr_port_t PortC = MAKE_AVR_PORT(C);
avr_port_t PortD = MAKE_AVR_PORT(D);

// Code continues on the next page ...
```

Code Listing 4: AVR C code for **avr_io.h** (part 1)

```
// Code continues from the previous page.

inline void io_set_bit(volatile reg8_t *r, uint8_t b) {
    r->REG |= (1 << b);
}

inline void io_clear_bit(volatile reg8_t *r, uint8_t b) {
    r->REG &= ~(1 << b);
}

inline void io_toggle_bit(volatile reg8_t *r, uint8_t b) {
    r->REG ^= (1 << b);
}

inline uint8_t io_read_bit(volatile reg8_t *r, uint8_t b) {
    return (r->REG >> b) & 1;
}

inline void io_pin_mode(avr_port_t *p, uint8_t bit, uint8_t output) {
    if (output)
        io_set_bit(p->DDR, bit);
    else
        io_clear_bit(p->DDR, bit);
}

inline void io_write(avr_port_t *p, uint8_t bit, uint8_t value) {
    if (value)
        io_set_bit(p->PORT, bit);
    else
        io_clear_bit(p->PORT, bit);
}

inline uint8_t io_read(avr_port_t *p, uint8_t bit) {
    return io_read_bit(p->PIN, bit);
}

inline void io_toggle(avr_port_t *p, uint8_t bit) {
    io_toggle_bit(p->PORT, bit);
}

#endif
```

Code Listing 4: AVR C code for **avr_io.h** (part 2)

Questions for C Programming Knowledge

1) What is the **memory size** required to store the data structure **`reg8_t`**, and why?

2) What is an **inline C function**? How is it different from a normal C function?

3) What is the purpose of including the C header file `<stdint.h>`?

4) Explain the purpose of the C macro **REG8(...)**. What does it do?
   Give a usage example of this C macro for the AVR MCU.

```
#include <stdint.h>

typedef union {
    uint8_t REG;
    struct {
        uint8_t B0:1;
        uint8_t B1:1;
        uint8_t B2:1;
        uint8_t B3:1;
        uint8_t B4:1;
        uint8_t B5:1;
        uint8_t B6:1;
        uint8_t B7:1;
    } bits;
} reg8_t;

#define REG8(addr) ((volatile reg8_t *) (addr))
```

Lab 3: I/O Polling AVR / ATmega328P

Lab Procedure

1) Write C code in **main.c** using the **Arduino IDE** to perform the following tasks:

- In an endless loop, the MCU reads the logic level on the **PD2** pin (**Arduino D2 pin**) and immediately writes the inverted value to the **PB4** pin (**Arduino D12 pin**) configured as output.
- Enable the internal pull-up resistor on the **PD2 pin** configured as input.
- Use a push button in an **active-low** configuration to provide the input signal on **PD2 pin**.

2) Test your code using **Wokwi simulator** (***This task must be completed before entering the lab session***).

3) Test the program using an **Arduino Uno** or **Nano** board during the lab session:

- Create a push button circuit on a breadboard and connect the input signal to the Arduino board.
- Use a **digital oscilloscope** with **CH1** and **CH2** to measure the signals on **PD2** and **PB4**, respectively.
- Measure the I/O delay between the input and output signals when a transition occurs.
- Capture the waveform for inclusion in the lab report.

Questions

- What are the range of the delay values (L-to-H and H-to-L delays) measured during the experiment? What are the key factors that affect these delay values? Compare your measured delays with typical propagation delay values of a 74HC/HCT14 logic gate.
- Did you observe any bouncing effects occurred on the input signal? If yes, include the captured oscilloscope waveform showing the bouncing of the input button. How did the MCU respond in this case, considering the logic level changes on the output signal?
- In cases where button bouncing occurs, how can the problem be solved using both hardware and software approaches? Explain how each method works.