Lab Sheet for Week 4

Lab Instructor: **RSP**

Lab 1: Bare-metal C Programming for AVR using Microchip Studio IDE

Objectives

- Use Atmel Studio 7 or Microchip Studio for AVR to develop firmware for the ATmega328P MCU.
- Write embedded C code to perform basic I/O operations on the MCU's I/O ports and pins.
- Use the built-in simulator in Atmel Studio 7 or Microchip Studio for AVR, or the Wokwi Simulator, to simulate and test the source code.
- Use AVRDUDE to upload firmware to the Arduino Uno or Nano board without overwriting the Arduino bootloader.
- Use a digital oscilloscope and a USB Logic Analyzer to analyze the output signals.

Lab Procedure

1. Create a new C project in **Atmel Studio / Microchip Studio.**

   1.1 Select **GCC C Executable Project**, enter a project name, and choose a directory for the project.

   1.2 Select **ATmega328P** as the target device and click the "OK" button.

   1.3 Open **main.c** and edit the source code using the provided template (**Code Listing 1**).

2. Write C code in **main.c** as follows:

   2.1 Implement a C function named `gpio_init()` that configures output pins by setting pins **PB2–PB5** as outputs (**Arduino pins D10–D13**).

   2.2 Initial bit pattern of these pins is `0b0001`.

   2.3 Implement a C function named `update_output()` that implements rotating bit patterns (left-to-right rotation). When this function is called, it updates the output value on **PB2–PB5** using the following rotation sequence (one step by call):

PB5..PB2

```
0001
0010
0100
1000
0001
0010
....
```

3. Debug the program using the built-in simulator.

   3.1 Start a **debugging session** (choose **Debug → Start Debugging** and **Break** from the menu) using the simulator to begin the code execution.

   3.2 Ensure that the built-in simulator is selected as the debugger.

   3.3 Set breakpoints on selected lines of code in **main.c** before stepping through the compiled program.

   3.4 Observe the **AVR MCU state** — such as I/O registers and CPU cycle count—by opening the **Processor View, Register View**, and **I/O View** during code execution.

   3.5 Step through the code repeatedly until the next breakpoint is reached (see **Figures 1 and 2**).

4. Uploading the **.hex file** using **AVRDUDE**

   4.1 Connect the Arduino board to the host computer via a USB port.

   4.2 Open **Windows PowerShell** and run the **AVRDUDE** command which uploads the .hex file of your project to the Arduino Uno board.

   4.3 Ensure that you specify or adjust the correct file paths and the correct **serial COM port number** before executing the command.

5. Verify the correctness of your code

   5.1 Use a digital oscilloscope to visualize the at least two output signals.

   5.2 Measure the pulse width and frequency of each signal.

   5.3 Capture waveforms for inclusion in the lab report.

   5.4 Take photos of your lab setup showing the signal measurement configuration.

6. Revise the C code to implement a bidirectional (back-and-forth) sequence. Modify the output pattern sequence to:

```
0001
0010
0100
1000
0100
0010
0001
....
```

7. Rebuild the firmware in **Microchip Studio** and upload it again to the **Arduino Uno / Nano** using **AVRDUDE**.

8. Verify the revised output by using a **USB logic analyzer** to confirm the updated bidirectional bit pattern sequence.

8.1 Connect jumper wires to the USB logic analyzer. As an example, the WeAct Studio Logic Analyzer is used, as shown in **Figure 3**.

8.2 Install the USB driver for the **8-channel 24 MHz USB logic analyzer** (using the Cypress FX2(LP) / CY7C68013A or compatible chip) on Windows 10 or Windows 11. Use the **SysProgs USB Driver Tool** utility (https://visualgdb.com/UsbDriverTool/) to install the **WinUSB driver** for the device.

- **Figure 4** shows that the USB logic analyzer with **VID:PID (0x1D50:0x608C)** is detected by the **USB Driver Tool** utility. The correct USB driver must be installed before the device can be used.
- **Figure 5** shows that the USB driver (**fx2lafw / WinUSB**) is installed successfully and ready for use.

8.3 Install the **PulseView / sigrok software** (https://sigrok.org/wiki/Downloads) that supports the **USB logic analyzer.** Capture the waveforms of the output signals using PulseView for inclusion in the lab report.

- **Figure 6** shows the initial **PulseView** window, which successfully detects and connects to the USB logic analyzer (**Sigrok FX2 LA**).

- Before starting signal capture with the USB logic analyzer, configure the **number of samples** and the **sampling rate**. Ensure that the ground of the USB logic analyzer is connected to the ground of the Arduino board.
- **Figure 7** shows the visualization of captured signal waveforms on eight channels, where the first four channels are output signals from the Arduino board.
- After capturing the signal waveforms, vertical cursors can be used to measure the pulse width or the signal period, as shown in **Figure 8**.
- **Note**: On some Windows 10/11 computers, **Microsoft Visual C++ 2010 Redistributable** may be required to run **PulseView**.

```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

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

#define T 100 // time interval in msec

void gpio_init() {
    // Configure GPIO pins here
}

void update_output() {
   // Update the LED pattern here
}

int main(void) {
   gpio_init();
   while (1) {
     update_output();
     _delay_ms( T );
   }
   return 0;
}
```

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

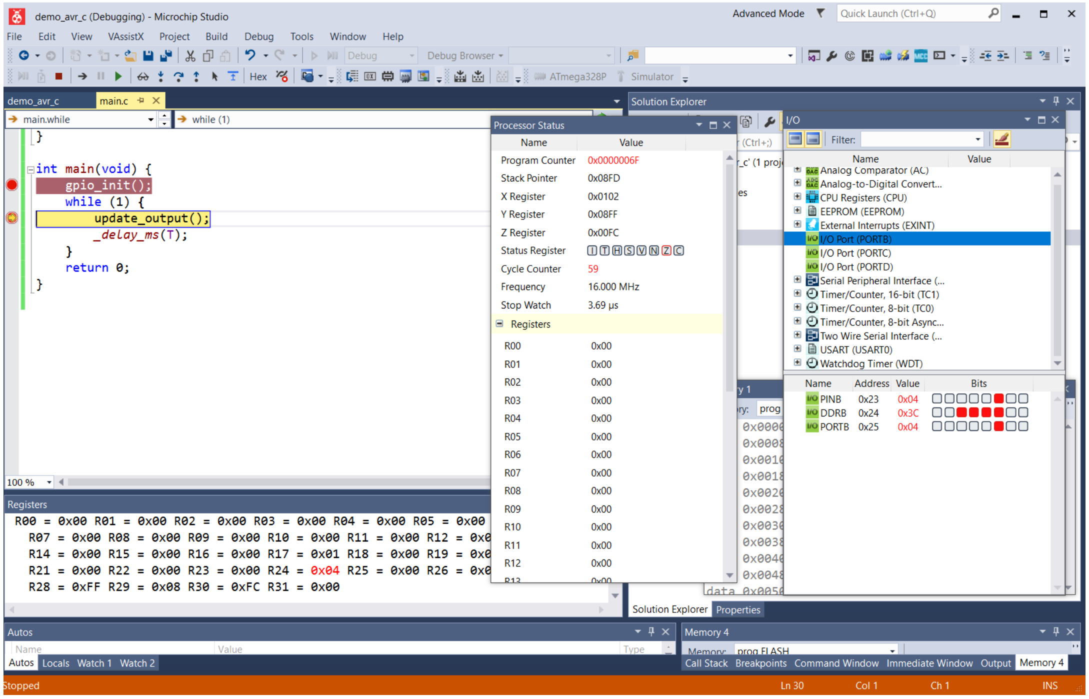


Figure 1: Code Debugging using the built-in AVR simulator

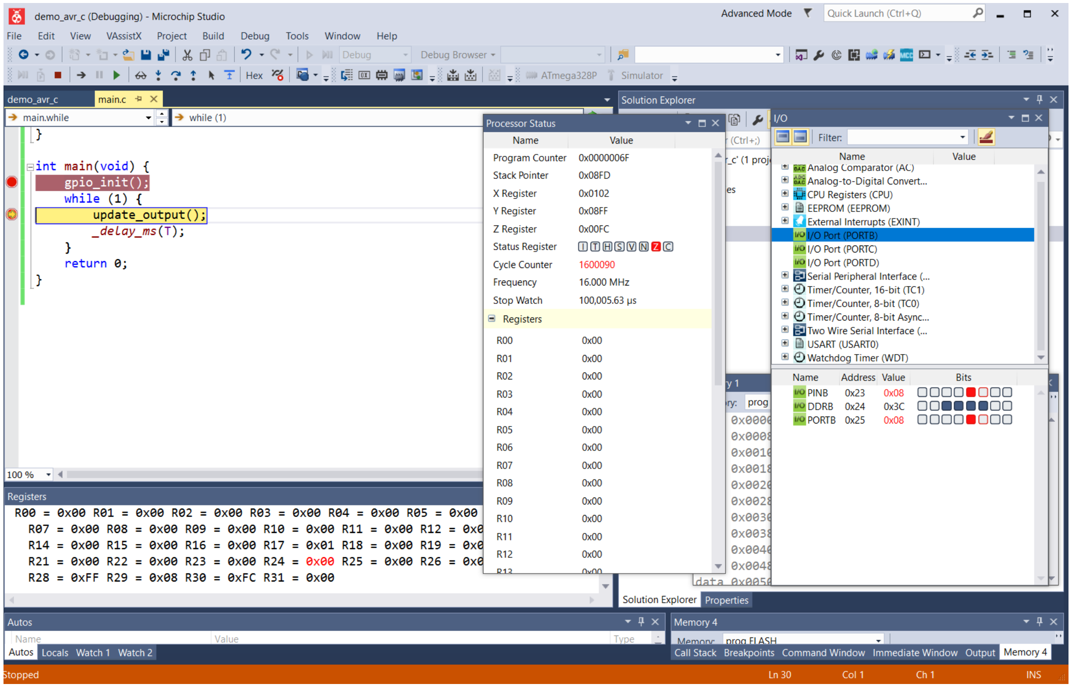
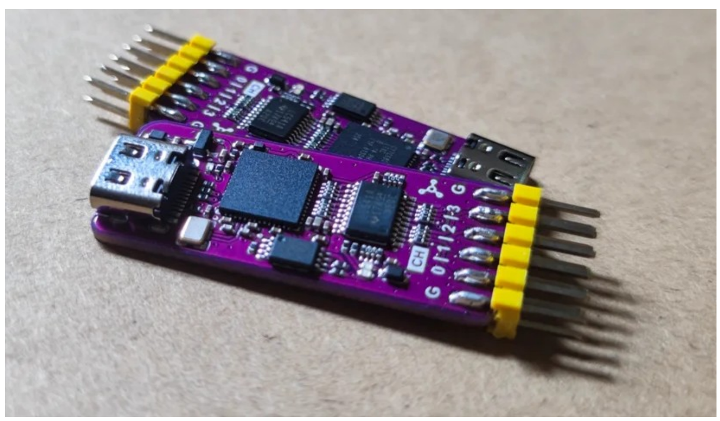


Figure 2: Run the code until the next breakpoint and observe the changes in cycle count and I/O PORT B registers



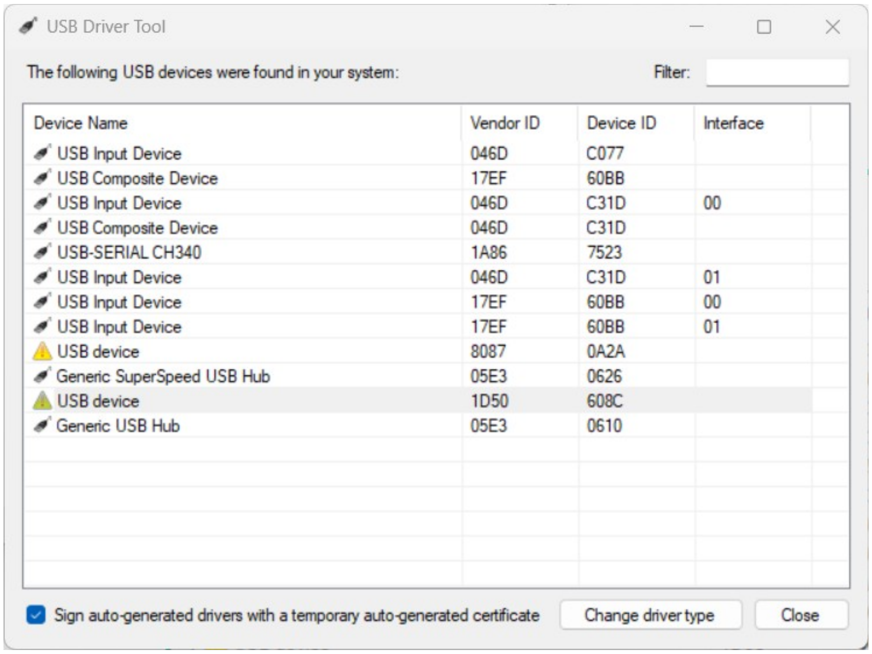


Figure 3: WeAct Studio Logic Analyzer

Figure 4: The USB device with VID:PID (0x1D50:0x608C) detected by the USB Driver Tool utility. The correct USB driver must be installed before the device can be used.

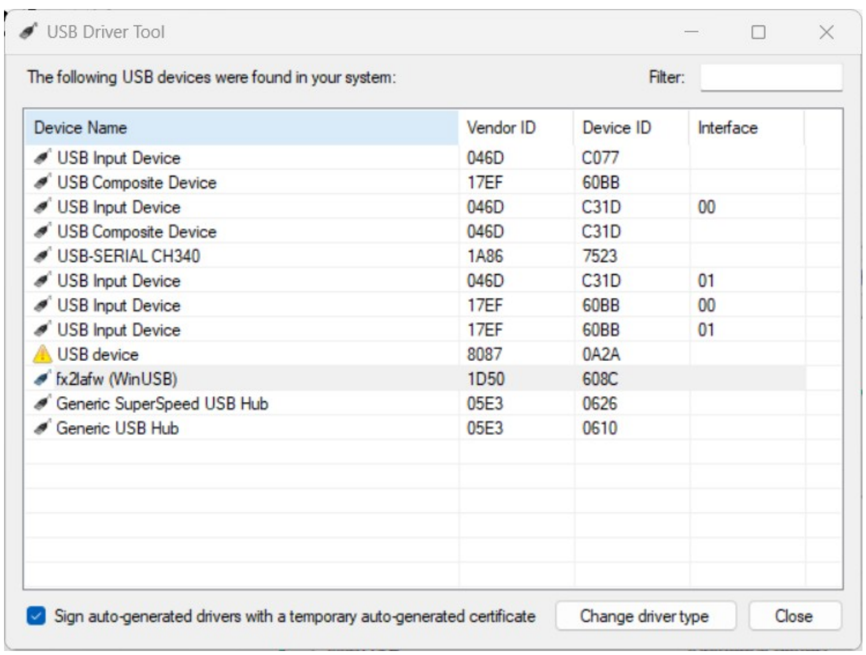


Figure 5: The USB driver (fx2law / WinUSB) is installed successfully and ready to use.

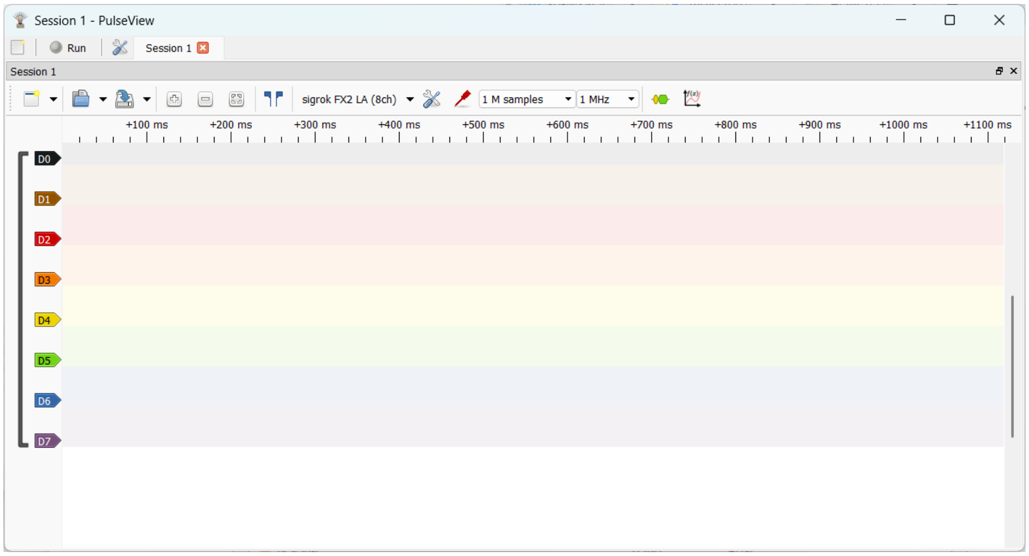


Figure 6: The initial window of PulseView

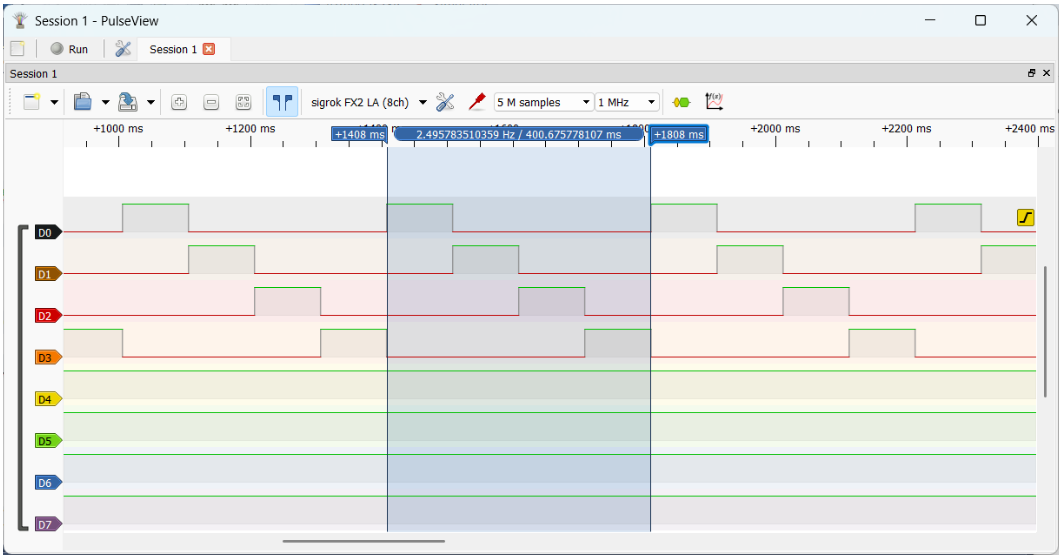


Figure 7: Visualization of captured signal waveforms on 8 channels (the first four channels are output signals from the Arduino board).

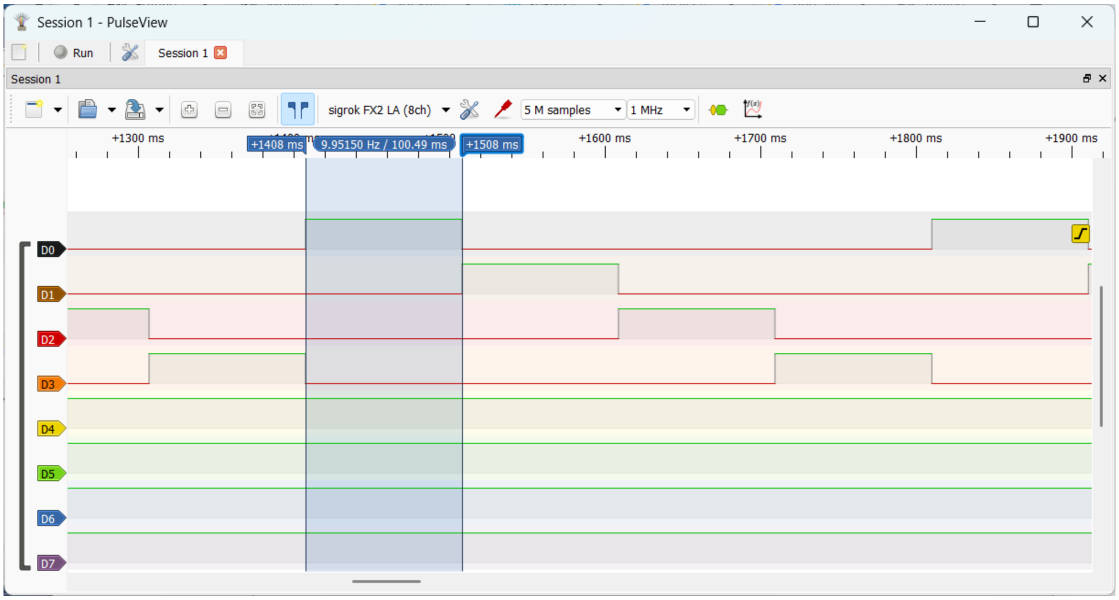


Figure 8: Vertical cursors can be used to measure the pulse width of a signal

Lab 2: AVR Assembly Programming for ATmega328P using Microchip Studio IDE

Objectives

- Use Atmel Studio 7 or Microchip Studio for AVR to develop firmware for the ATmega328P MCU.
- Write AVR Assembly code to perform basic I/O operations on the MCU's I/O ports and pins.
- Use the built-in simulator in Atmel Studio 7 or Microchip Studio for AVR to study instruction execution, CPU cycle counts, and subroutine timing behavior without using external hardware.
- Use AVRDUDE to upload firmware to the Arduino board without overwriting the Arduino bootloader.
- Use a digital oscilloscope or a USB Logic Analyzer to analyze the output signals.

Lab Procedure

1. Create a new project in **Atmel Studio / Microchip Studio.**

   1.1 Select **AVR Assembly**, enter a project name, and choose a directory for the project.

   1.2 Select **ATmega328P** as the target device and click the "OK" button.

   1.3 Open the **.asm** file to edit the source code.

2. Write AVR assembly code in **main.asm** as follows:

   2.1 Use the provided AVR assembly code as shown in **Code Listing 2**.

   2.2 Build the project to produce the output files.

3. Debug the program using the built-in simulator.

   3.1 Start a **debugging session** (choose **Debug** → **Start Debugging** and **Break** from the menu) using the simulator to begin the code execution.

   3.2 Ensure that the built-in simulator is selected as the debugger.

   3.3 Set breakpoints on selected lines of the Assembly code before stepping through the compiled code.

   3.4 Observe the AVR MCU states such as I/O registers and CPU cycle count during code execution by opening the **Processor View, Register View**, and **I/O View**.

   3.5 Step through the code repeatedly until the next breakpoint is reached.

   3.6 Analyze the program behavior using the built-in AVR simulator during the debugging session.

   - Analyze how the AVR subroutine **SW_DELAY** is called using the **RCALL** instruction and how control returns to the main program using the **RET** instruction.
   - Determine the exact number of CPU cycles consumed by the software delay subroutine **SW_DELAY** using the simulator's cycle counter.
   - Hint: Set a breakpoint at the **rcall SW_DELAY** instruction (see **Figure 9**), run the program until the breakpoint is reached, and record the CPU cycle count before and after the subroutine execution.

4. Upload the **.hex** file using **AVRDUDE** and use a digital oscilloscope to measure the pulse width and frequency of the output signals. Capture the waveforms for inclusion in the lab report.

Post-Experiment Questions

1. How does the value loaded into the register **n** (**r18**) affect the delay produced by the **SW_DELAY** subroutine and, consequently, the frequency of the output signals?

2. Using the AVR simulator, how many CPU cycles are consumed by a single call to **SW_DELAY**? Does this value include the execution time of the **RCALL** and **RET** instructions? Explain.

```
.include "m328pdef.inc"

.def temp = r16
.def mask = r17
.def n = r18

.org 0x0000
    rjmp RESET          ; [2C] jump to RESET

RESET:
    ; Set PB5 and PB4 as outputs
    ldi temp, (1<<PB5)|(1<<PB4)
    out DDRB, temp
    ; Set bit at PB4 of PORTB
    ldi temp, (1 << PB4)
    out PORTB, temp
    ldi mask, (3 << PB4)

MAIN:
    in temp, PORTB       ; [1C] read PORTB into temp
    eor temp, mask       ; [1C] perform an XOR operation
    out PORTB, temp      ; [1C] write temp to PORTB
    ldi n, 100           ; [1C] set n = 100
    rcall SW_DELAY       ; [3C] call the SW_DELAY subroutine
    rjmp MAIN            ; [2C] go back to MAIN

SW_DELAY:
    tst r18              ; [1C] test if r18 is zero
    breq DONE            ; [1C/2C] branch if equal zero (check Z flag)
OUTER_LOOP:
    ldi r24, low(3999)   ; [1C] load the load byte into r24
    ldi r25, high(3999)  ; [1C] load the high byte into r25
INNER_LOOP:
    sbiw r25:r24, 1      ; [2C] 16-bit subtract
    brne INNER_LOOP      ; [1C/2C] branch if not equal zero
    dec r18              ; [1C] decrement r18 by 1
    brne OUTER_LOOP      ; [1C/2C] branch if not equal zero
DONE:
    ret                  ; [4C] return from subroutine
```

Code Listing 2: AVR assembly code template for Lab 2

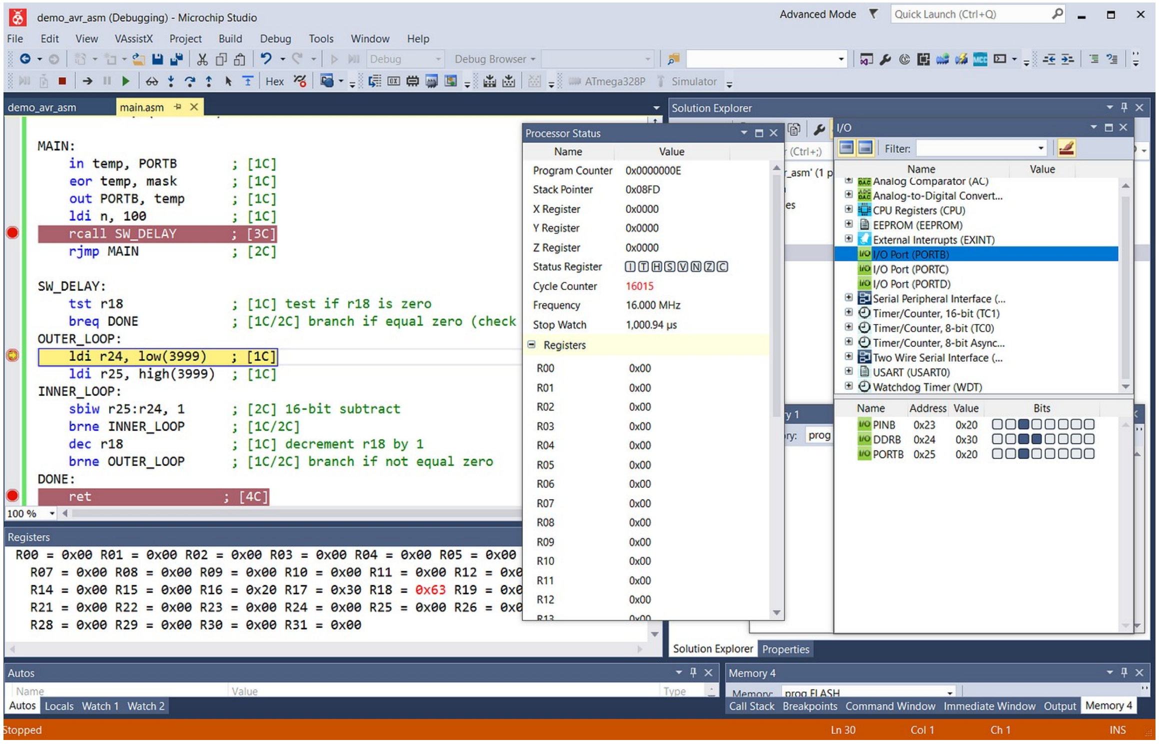


Figure 9: Code debugging using the built-in simulator