

LABORATORY REPORT

To: Dr.Randy Hoover

From: Benjamin Lebrun, Benjamin Garcia

Subject: Lab Assignment 2: AVR Assembly and UART Communication

Date: February 24, 2019

Introduction

This lab required us to output the value in a counter to three LEDs as well as to the UART serial port. Additionally, we were required to accept input from the serial port and use this input to set the value of the counter. A component of the output requirement was that it print/flash the LEDs every two seconds using a wait function that we created.

Our implementation holds the counter in a register and converts to and from an ASCII representation for presenting information over the serial port. Pins 0, 1, and 2 on PORTB were used for lighting up the LEDs. To handle receiving information over the serial port, we enabled the 'UART RX Complete interrupt' so that whenever a key is pressed, it will set the value of the counter for the next time the LEDs are flashed. Our wait uses a triply nested loop to consume 16,000,000 cycles (1 second when running at 16MHz).

The finished implementation blinks on a two second cycle (two seconds on, two seconds off) and key-presses are registered for the next cycle. When first turned on, the counter is initially set to 0, and the LEDs are off.

Equipment

This lab required the following equipment:

- 3 resistors (220 Ω)
- 3 LEDs
- 4 wires (male-male)
- 1 USB-A to USB-B cable
- 1 breadboard
- 1 Arduino UNO (ATMega328P)
- Atmel Studio 7
- Putty

configuration

The Arduino was connected to the computer with the USB-A to USB-B cable. On the breadboard, each LED was placed in a circuit with a single resistor connected to ground. One of the Arduino's ground pins was connected to the breadboard's ground, and the LEDs were connected to PORTB pins 0, 1, and 2 (pins numbered 8, 9, 10 on the Arduino).

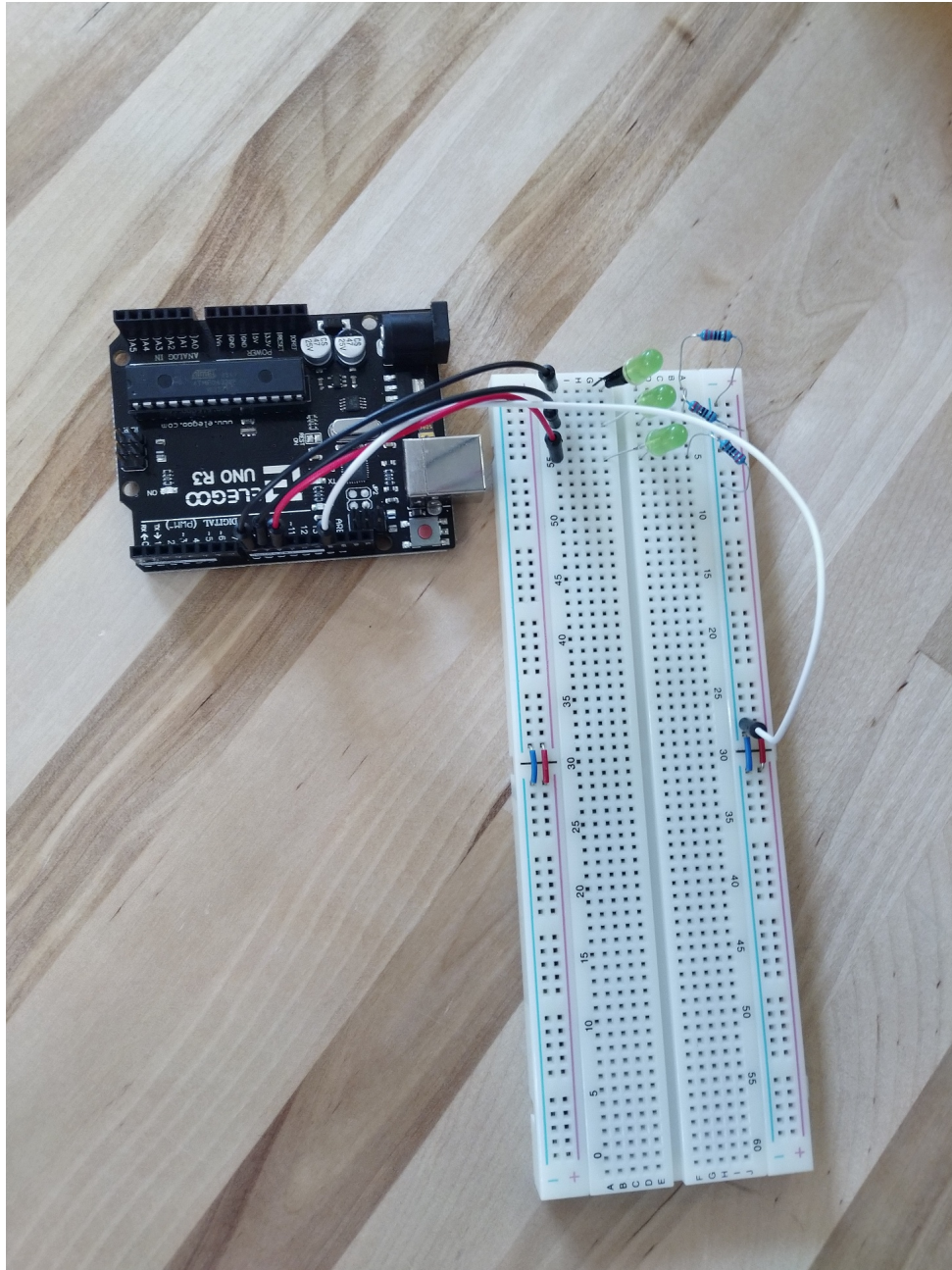


Figure 1: Picture of the wiring setup

Implementation

One Second Wait

The wait implementation uses three registers as loop variables to create a busy loop that occupies 15,999,984 cycles with variable pushing and popping and the function's return occupying a further 16 cycles to give 16,000,000 cycles.

ATOI

Handles converting numbers in ASCII representation to their depicted value (i.e. '7' is converted from 0x37 to 0x07).

It subtracts '0' (0x30) from the incoming ASCII character value to produce the numeric value. Once this is done, the value is stored in the counter for use on the next light cycle.

ITOA

Does the exact opposite of ATOI, it converts a numeric value to an ASCII character (i.e. 0x07 converts to 0x37 '7').

This is accomplished by subtracting the quantity (0 - '0') from the number (equivalent to adding 0x30). This value is sent over the serial line to the computer, where Putty can display it.

UART_RX (Interrupt)

To handle the RX complete interrupt, we first altered the Init code provided to set UCSR0B to 0x98 to enable bits 7 (RX Complete Interrupt), 4 (RX Enable), and 3 (TX Enable). Once the global interrupt bit was set, our function 'UART_RX' would be called whenever the UART buffer was filled. This function calls ATOI to set the counter and then returns control to the last instruction executed.

UART_INIT

UART_INIT is boilerplate code that was provided with the lab. The only change we have made is to enable the 'RX Complete Interrupt' as mentioned above.

This function configures the UART control and status registers to allow writing and reading using the '8n1' protocol on the UART port. Additionally, communication is configured to use a baudrate of 9600.

UART_TX

Unlike the RX function, UART_TX is not an interrupt, and is called every time the 'flash_leds' function is called. This function waits until the buffer is empty to write, then calls itoa to convert the counter to an ASCII character and outputs it to the serial port.

Program Loop and Startup

Our startup routine first assigned 0xff to DDRB to enable output. Then we initialize our counter to 0, and setup the stack to begin at RAMEND. Finally, we call the UART_INIT function to configure

the UART for the Program Loop.

The Program Loop starts by waiting for two seconds, then it will flash the LEDs, outputting the value of our counter to the LEDs and the serial port. Then, two seconds later, it turns off the LEDs. After the LEDs have been turned off, we jump back to the start of the Program Loop.

Discussion

AVR assembly is an uncomplicated approach to the larger family of assembly architectures. We have access to a decently sized array of registers but also can access special hardware and software features usually by a call to a memory address on the board.

Our first smaller issue was remembering how to convert between ASCII codes and a normal integer value. After rewriting the transmit method from the lab handout, we mistakenly sent ASCII codes being subtracted from zero into the UART, which displayed an invalid character to the screen. This was remedied by simply remembering the correct conversion between integers and ASCII codes.

To implement the receive function, we used interrupts instead of waits or delays to properly receive data. The transmit function, because of the nature of our application, could be called in code with every flash of the LEDs at the prescribed interval from our lab requirements and specs and therefore did not require the direct use of an interrupt to constantly check the state of the buffer or carefully navigate writing to the assigned register buffer. However, we found out that simply setting the global interrupt flag and creating our interrupt vectors in code wasn't enough to activate the interrupt. As noted in implementation we also had to set a specified interrupt flag to activate the correct interrupt.

Responses

Respond intelligently and at necessary length to any questions posed in the lab assignment.

1 It is extremely unlikely that your one-second delay is exactly one second. How many clock cycles are actually consumed inside your one-second delay? What is your relative error?

1. Our wait uses exactly 16 Million cycles plus 3 for the initial call. Our function does account for the two cycles used if our break if equal command does attempt to set the comparison register. For 16 million cycles on our board with three for the call, our delay lasts for 16,000,003 cycles comes within 0.18 nanoseconds of one second, or approximately 0.00001875% error.

2 The USB serial communication protocol and the UART communication we've implemented on the ATmega328P are very different, yet we are able to plug a cable between the two and communicate! How is this possible on the Arduino Uno R3?

2. The Arduino Uno R3 uses a dedicated memory address to write to a built-in USB to serial adaptor. This adaptor has a specialized chip which attempts to carry UART communication through USB protocol, and generally has its own built in EEPROM, voltage regulators, receivers and transmitters and internal timing oscillator and a host of other features to allow communication conversion between USB and UART.

3 Explain each step of the UART RX call provided in this document.

3. UART RX first pushes the contents of register 17 onto the program stack for storage, then loads the status of the UART buffer from the UCSR0A register. We use this register to read the status of the UART, such as if the buffer is full or if communication has ended. In our case we are waiting for communication to end and compare the RCX0 register to do this. We use a comparison to check if this bit is set; if it is, we then load the contents of the UDR0 data register into register 16. Then we pop the data from the program stack back onto register 17 where we previously carried our UART status register and carry on.

Appendices

The following files are included as appendices:

- main.asm
- wait2.asm (the version of the wait function used in the program)

Appendix A: main.asm

```
;
; lab_02.asm
;
; Created: 2/16/2019 13:50:57
; Author : Ben & Ben
;
;
; USING:
; PORTB Pins 0–7
;
; DDRB – The Port D Data Direction Register – read/write
; PORTB – The Port D Data Register – read/write
; PINB – The Port D Input Pins Register – read only
;
; PORTB = B10101000; // sets digital pins 7,5,3 HIGH
;
; (from https://www.arduino.cc/en/Reference/PortManipulation)
;
;
.cseg
; reset interrupt vector, should just immediately restart the program
.org 0x0000
    rjmp start

; UART receive interrupt
.org 0x0024
    rcall UART_RX
    reti

.org 0x0034

; UART initialization
UART_Init:
    push    uart_buf
    push    uart_buf2
    lds     uart_buf, UCSR0B
    ;ori     uart_buf, 0x18 ; original line (rx and tx)
    ori     uart_buf, 0x98 ; (enable rx,tx, and rx interrupt)
    sts     UCSR0B, uart_buf ; enables rx tx

    lds     uart_buf, UCSR0C
    ori     uart_buf, 0x06
    sts     UCSR0C, uart_buf ; 8n1 async

    ldi     uart_buf, high(BAUD_PRESCALE)
    ldi     uart_buf2, low(BAUD_PRESCALE)
    sts     UBRR0H, uart_buf
    sts     UBRR0L, uart_buf2
    pop     uart_buf2
    pop     uart_buf
    ret

; UART transmit, messages should be put in the uart_buf register
UART_TX:
    push    uart_buf
    lds     uart_buf, UCSR0A
    sbrs    uart_buf, UDRE0 ; Check if there's data to read
    rjmp    UART_TX ; Loop if there is waiting data

    rcall   itoa ; convert num to 'alpha'

    sts     UDR0, alpha ; output alphanumeric
    pop     uart_buf
    ret

; UART receive, reads from out_buf register
; Uses the UART RX Complete interrupt
UART_RX:
    lds     out_buf, UDR0
    mov     alpha, out_buf ; move what we read into the 'alpha' register
    rcall   atoi ; set num from 'alpha'
    ret

flash_leds:
    out     PORTB, num
    rcall   UART_TX
    ret

clear_leds:
    ldi     tmp, 0x00
    out     PORTB, tmp
    ret

; Replace with your application code
.org 0x0100

; UART configuration
.equ F_CPU = 16000000
.equ BAUDRATE = 9600
.equ BAUD_PRESCALE = F_CPU / (BAUDRATE*16) - 1
; defines r16 as output buffer
```

```

.def out.buf = r16          ; uart rx buffer
.def uart.buf = r17         ; uart tx buffer
.def uart.buf2 = r18        ; overflow buffer
.def num = r24              ; value we flash to LEDs
.def alpha = r23            ; value we send to uart
.def tmp = r22              ; tmp location

.include "wait2.asm"
; Initialization function, should be called for every reset interrupt
start:
    clr     r1
    out     SREG, r1          ; clear sreg for safety

    ldi     r16, 0xff         ; load 0x11111111 to reg 16
    out     DDRB, r16        ; set portb to output

    clr     r16              ; clear it because it's one of our buffers

    ldi     num, 0
    rcall   itoa

    ldi     r28, LOW(RAMEND)
    ldi     r29, HIGH(RAMEND)
    out     SPL, r28
    out     SPH, r29

    rcall   UART_INIT
    sei

    rjmp    prgmloop
; Main program loop
prgmloop:
    rcall   wait_1.second
    rcall   wait_1.second
    rcall   flash_leds
    rcall   wait_1.second
    rcall   wait_1.second
    rcall   clear_leds
    rjmp    prgmloop

; itoa function for our bit to ascii code for digits 0 through 10
; in - num in byte
; out - alpha in ASCII code
itoa:
    mov     tmp, num
    subi    tmp, 0 - '0'
    mov     alpha, tmp
    ret

; atoi function for our ascii to bit representation for digits 0 through 10
; in - alpha in ASCII code
; out - num in byte code
atoi:
    mov     tmp, alpha
    subi    tmp, '0'
    mov     num, tmp
    ret

```

Appendix B: wait2.asm

```
/*
 * wait2.asm
 *
 * Triple nested loop wait implementation. consumes exactly
 * 16,000,000 cycles (1 second at 16MHz), bringing the total
 * cost of the function to 16,000,003 cycles including an rcall
 * to the procedure.
 *
 * uses registers r16, r17, and r18, pushing their states to the
 * stack. rjmp's past the definition to allow .include directive
 * to work as expected.
 *
 * Created: 2/19/2019 2:20:32 PM
 * Author: Benjamin Garcia
 */

;.def i=r16
;.def j=r17
;.def k=r18

rjmp end_wait.2.asm

; void wait.1.second();
; waits 1 second (16,000,000 cycles)
; 16,000,000 = 4i + 4ij + 4ijk + 16
wait.1.second:
    ; pushing = 6 cycles
    push r16
    push r17
    push r18

    ldi r16, 0x6c ; load 108
    loop.i.wait.1.second: ; 4i + 4ij + 4ijk - 1
        ldi r17, 0xbc ; load 188
        loop.j.wait.1.second: ; 4j + 4jk - 1
            ldi r18, 0xc4 ; load 196
            loop.k.wait.1.second: ; 4k - 1
                dec r18
                breq end.k.wait.1.second ; 1 cycle (2 on last iter)
                rjmp loop.k.wait.1.second ; 2 cycles (not executed on last iter)
            end.k.wait.1.second:
                dec r17
                breq end.j.wait.1.second ; 1 cycle (2 on last iter)
                rjmp loop.j.wait.1.second ; 2 cycles (not executed on last iter)
            end.j.wait.1.second:
                dec r16
                breq end.i.wait.1.second ; 1 cycle (2 on last iter)
                rjmp loop.i.wait.1.second ; 2 cycles (not executed on last iter)
        end.i.wait.1.second:

    ; popping and return = 10 cycles
    pop r18
    pop r17
    pop r16
    ret

end_wait.2.asm:
```