# ECE 375 Lab 1

## Introduction to AVR Development Tools

Lab session: 015
Time: 12:00-13:50

Author: Astrid Delestine
Programming partner: Lucas Plastid

TA Signature

# 1 Introduction

This is the first Lab in the ECE 375 series and it covers the setup and compilation of an AVR Assembly Program. The student will learn how how to use the sample Basic Bump Bot assembly file and send the binaries to the AVR Microcontroller board. For the second part of the lab the student will be expected to download and compile the included C sample program and from it learn how to configure the I/O ports of the ATmega32U4 Microcontroller. The student will then write their own C program and upload it to the Microcontroller to verify that it runs as expected. The provided programs have been attached in the source code section of this report.

# 2 Design

As for part 1 of this lab assignment, no design needs to be done as the program is supplied. For part 2 of this lab assignment the C program was created to mimic the operaitons of the bump bot assembly file. Firstly the student must understand how the Bump Bot code must operate and they gain this information from the slides provided as they must program the right LED's to illuminate. For our program we decided that we wanted everything to be as readable as possible, thus we created constants for each of the LED directional cues.

# 3 Assembly Overview

As for the Assembly program an overview can be seen below

## 3.1 Internal Register Definitions and Constants

Four different registers have been setup, those being the multipurpose register (mpr), the wait counter register (waitcnt), and two loop counters, for counting the cycles of the delay function. In addition to these, there are several different constants. WTime defines the time in milliseconds to wait inside the wait loop. The rest of the defined constants are either input bits, engine enable bits, or engine direction bits.

## 3.2 Initialization Routine

The initialization routine sets up several important ports and pointers that allow the rest of the assembly to work. Firstly the stack pointer is initialized at the end of RAM so that when the program pushes and pops items into and out of it, the stack does not interfere with any other data. Port B is then initialized for output, and Port D is initialized for input. The move forward command is also in this phase, to give a default movement type.

## 3.3 Main Routine

The main program constantly checks for if either of the whisker buttons have been hit, by reading the input of the PIND. When one of the whiskers is hit, the correct subroutine is called. As long as no button is hit the bump bot will continue in a straight line.

## 3.4 Subroutines

### 3.4.1 Hit Right

The HitRight subroutine describes what happens when the right whisker bit is triggered. The robot will move backwards for a second, then turn left for a second, then it will continue forward.

### 3.4.2 Hit Left

The HitLeft subroutine describes what happens when the left whisker bit is triggered. First the bump bot will move backwards for a second, then it will turn right for a second, then it will continue forward.

### 3.4.3 Wait

The Wait subroutine controls the wait intervals while the bump bot is preforming an action. Due to each clock cycle taking a measurable amount of time, we can calculate how many times we need to loop for. This function used the olcnt and ilcnt to have two nested loops, running the dec command until they equal zero, thus waiting the requested amount of time. **The original program was changed by modifying the Wtime constant value by shifting the bit back by 1 space inside of the HitRight subroutine and the HitLeft subroutine. This effectively doubles the wait time. See Lines 167, 201**

# 4 C Program Overview

Each of the methods determined to operate the bump bot can be seen in the code section at the end of this report, their descriptions are here.

## 4.1 Definitions and Constants

Several different constant integer values are prescribed on lines 29 - 33. These constants are the binary values for what the LED's should be when enabled. Several functions are defined here as well, those being, BotActionL() , BotActionR() and goBackwards2Sec(). Each of these are quite self explanatory as to what they do.

## 4.2 Main Method

The main method initializes ports D and B for input and output respectively. Then for port D, due to the fact that it is an input, has its high 4 bits pulled high to enable inputs on those channels. It is importnat to note that all of the inputs are active low, so we must invert them, this is done on line 51. Next the main function enters an infinite while loop, that constantly checks the input of PIND and depending on the inputs, calls the BotActionL() or BotActionR() functions. It ends the while loop by setting the LEDs to forward direction and debouncing the button press by 50ms.

## 4.3 Functions

### 4.3.1 BotActionL()

first this function calls the goBackwards2sec() function, then it sets the left motor to forwards and the right motor to backwards, turning the robot right. It then waits 1 second for the action to take place, then returns to the main loop.

### 4.3.2 BotActionR()

first this function calls the goBackwards2sec() function, then it sets the right motor to forwards and the left motor to backwards, turning the robot left. It then waits 1 second for the action to take place, then returns to the main loop.

### 4.3.3 goBackwards2Sec()

This function sets the LED's to the reverse motor direction for two seconds, then returns to the main loop.

# 5 Testing

Testing was only done for the modified bump bot script and for the C program, as the unchanged bump bot script was left alone.

| Case | Expected | Actual meet expected |
|---|---|---|
| D4 Pressed | Backward movement→Turn Left→Forward | ✓ |
| D5 Pressed | Backward movement→Turn Right→Forward | ✓ |

Table 1: Assembly Testing Cases

| Case | Expected | Actual meet expected |
|---|---|---|
| D4 Pressed | Backward movement→Turn Left→Forward | ✓ |
| D5 Pressed | Backward movement→Turn Right→Forward | ✓ |

Table 2: C Testing Cases

# 6 Additional Questions

1. Take a look at the code you downloaded for today's lab. Notice the lines that begin with .def and .equ followed by some type of expression. These are known as pre-compiler directives. Define pre-compiler directive. What is the difference between the .def and .equ directives? (HINT: see Section 5.4 of the AVR Assembler User Guide).

   Pre-compiler directive can be defined as just that, a program or method that is run inside of the compiler, to save certain data values to the memory of the program. these values do not typically change. The .def directive defines a human readable word or reference, that the programmer can use instead of the register directly. This makes the code more human readable. The .equ directive creates a constant variable, that references in this case a number directly. This directive also makes the code more human readable, as one can easily see what number needs to be referenced. The main difference between the two is that .equ defines numbers, while .def defines registers, or places numbers can go.

2. Read the AVR Instruction Set Manual. Based on this manual, describe the instructions listed below.

   (a) ADIW

   Adds an immiditate value to a word. This is not a text word, rather a binary word. A binary value of 16bits. The value must be from 0 - 63. (pg33 Amtel AVR Instruction Set Manual)

   (b) BCLR

   Clears a single Flag in the SREG. (pg38 Amtel AVR Instruction Set Manual)

   (c) BRCC

   Conditional branch if the carry flag is cleared. Tests the carry flag in SREG and if it is zero branches. (pg42 Amtel AVR Instruction Set Manual)

   (d) BRGE

   Branches by testing the signed flag in SREG, and branches if that flag is cleared. This works with Signed binary numbers. (pg46 Amtel AVR Instruction Set Manual)

   (e) COM

   Preforms a ones complement operation on the passed register (pg76 Amtel AVR Instruction Set Manual)

   (f) EOR

   Compares two registers using exclusive or in a bitwise fashion.(pg91 Amtel AVR Instruction Set Manual)

   (g) LSL

   Preforms a logical shift left on the passed register moving the topmost bit into the carry flag if necessary. (pg120 Amtel AVR Instruction Set Manual)

   (h) LSR

   Preforms a logical shift right on the passed register and moves the lowest bit into the carry flag if necessary (pg122 Amtel AVR Instruction Set Manual)

   (i) NEG

   Preforms the two's complement on the passed register, the value $80 is left unchanged. (pg129 Amtel AVR Instruction Set Manual)

   (j) OR

   Preforms the logical OR operation between two registers, saves result in the first one. (pg132 Amtel AVR Instruction Set Manual)

   (k) ORI

   Preforms a logical OR operation between one register and an immediate value. Results in the register (pg133 Amtel AVR Instruction Set Manual)

(l) ROL

Shifts all bits to the left by one place, taking from the carry flag if necessary, and placing the rotated out bit into the carry flag if necessary. (pg143 Amtel AVR Instruction Set Manual)

(m) ROR

Shifts all bits to the right by one place, taking from the carry flag if necessary, then placing the rotated out bit into the carry flag if necessary. (pg145 Amtel AVR Instruction Set Manual)

(n) SBC

Subtracts two registers with the carry flag being subtracted as well if necessary. Places result in first register(pg147 Amtel AVR Instruction Set Manual)

(o) SBIW

Subtracts an immediate value from a word. (pg154 Amtel AVR Instruction Set Manual)

(p) SUB

Subtracts two registers and puts the result in the first register. (pg181 Amtel AVR Instruction Set Manual)

3. The ATmega32U4 microcontroller has six general-purpose input-output (I/O) ports: Port A through Port F. An I/O port is a collection of pins, and these pins can be individually configured to send (output) or receive (input) a single binary bit. Each port has three I/O registers, which are used to control the behavior of its pins: PORTx, DDRx, and PINx. (The "x " is just a generic notation; for example, Port A's three I/O registers are PORTA, DDRA, and PINA.)

   (a) Suppose you want to configure Port B so that all 8 of its pins are configured as outputs. Which I/O register is used to make this configuration, and what 8-bit binary value must be written to configure all 8 pins as outputs?

   DDRB would be configured and to enable all 8 pins as outputs it would need to be set to 0b11111111 or $ff.

   (b) Suppose Port D's pins 4-7 have been configured as inputs. Which I/O register must be used to read the current state of Port D's pins?

   To read from port D's pins, you must use the PINx command, for example PIND

   (c) Does the function of a PORTx register differ depending on the setting of its corresponding DDRx register? If so, explain any differences.

   Yes it does, due to the fact that if DDRx is set to logical 1, it then uses the PORTx as a voltage Sink or Source. PORTx can only be used. This means that PORTx can only be used as an output port if DDRx is set to a logical 1. For other operations, one would be configuring the pull up resistor.

4. This lab required you to modify the sample AVR program so the TekBot can reverse for twice as long before turning away and resuming forward motion. Explain how you have done it with reasons.

   This has been done by shifting the bit that had been preset as WTime to the left by 1, thus effectively multiplying it by 2. We did this in this way because we wanted to maintain the value of WTime outside of the move backward function. Looking back on the project I believe it would have been easier to just call the Wait method twice.

5. The Part 2 of this lab required you to compile two C programs (one given as a sample, and another that you wrote) into a binary representation that allows them to run directly on your ATmega32U4 board. Explain some of the benefits of writing code in a language like C that can be "cross compiled". Also, explain some of the drawbacks of writing this way.

   Some benefits of having a cross compiled program are that they should be easy to migrate to a different piece of hardware as the code itself should not need to change, only what hardware the code references. Some drawbacks are that it may be difficult to setup for the first time, or it may run slower than a program written specifically for a chip. C however is known for being almost as good as writing directly to hardware, like we are doing with Assembly

6. The C program you wrote does basically the same thing as the sample AVR program you looked at in Part 1. What is the size (in bytes) of your Part 1 & Part 2 output .hex files? Explain why there is a size difference between these two files, even though they both perform the same BumpBot behavior?

   The Assembly hex file came out to 485 bytes while the C file came out to 1020 bytes. This is over double the size. The main reason for the discrepancy in size can be attributed to possible bloat or automatic functions built into the C program to make it run smoothly. C works for you in this way, however the Assembly file is smaller due to the fact that we have defined exactly what needs to happen at every step thus making it a smaller file and possibly more precise.

# 7 Difficulties

This Lab was quite trivial, as such the only difficulties that we encountered were agreeing on how we would preform the requested task with modifying the original bump bot script. After this programming the bump bot in C was quite simple. As such this lab did not have many difficulties.

# 8 Conclusion

This lab reinforced the ideas of how assembly code is assembled, and how we can make this code run on our boards. Additionally it allowed the students to gain a better understanding of C programming when it relates to bare metal, and programming directly to a chip.

# 9 Source Code

Listing 1: Assembely Bump Bot Script

```
;
; Lab1_Sourcecode.asm
;
; Created: 1/13/2023 12:15:20 PM
; Author : Astrid Delestine and Lucas Plaisted!
```

```
;

;***********************************************************
;*
;*	BasicBumpBot.asm	-	V3.0
;*
;*	This program contains the neccessary code to enable the
;*	the TekBot to behave in the traditional BumpBot fashion.
;*      It is written to work with the latest TekBots platform.
;*	If you have an earlier version you may need to modify
;*	your code appropriately.
;*
;*	The behavior is very simple.  Get the TekBot moving
;*	forward and poll for whisker inputs.  If the right
;*	whisker is activated, the TekBot backs up for a second,
;*	turns left for a second, and then moves forward again.
;*	If the left whisker is activated, the TekBot backs up
;*	for a second, turns right for a second, and then
;*	continues forward.
;*
;***********************************************************
;*
;*	 Author: David Zier, Mohammed Sinky, and Dongjun Lee
;*	                    (modification August 10, 2022)
;*	   Date: August 10, 2022
;*	Company: TekBots(TM), Oregon State University - EECS
;*	Version: 3.0
;*
;***********************************************************
;*	Rev	Date	Name		Description
;*----------------------------------------------------------
;*	-	3/29/02	Zier		Initial Creation of Version 1.0
;*	-	1/08/09	Sinky		Version 2.0 modifictions
;*	-	8/10/22	Dongjun The chip transition from Atmega128 to Atmega32U4
;***********************************************************

.include "m32U4def.inc"				; Include definition file

;************************************************************
;* Variable and Constant Declarations
;************************************************************
.def	mpr = r16				; Multi-Purpose Register
.def	waitcnt = r17				; Wait Loop Counter
.def	ilcnt = r18				; Inner Loop Counter
.def	olcnt = r19				; Outer Loop Counter

```

```
.equ    WTime = 100                 ; Time to wait in wait loop

.equ    WskrR = 4                   ; Right Whisker Input Bit
.equ    WskrL = 5                   ; Left Whisker Input Bit
.equ    EngEnR = 5                  ; Right Engine Enable Bit
.equ    EngEnL = 6                  ; Left Engine Enable Bit
.equ    EngDirR = 4                 ; Right Engine Direction Bit
.equ    EngDirL = 7                 ; Left Engine Direction Bit

;/////////////////////////////////////////////////////////////
;These macros are the values to make the TekBot Move.
;/////////////////////////////////////////////////////////////

.equ    MovFwd = (1<<EngDirR|1<<EngDirL)    ; Move Forward Command
.equ    MovBck = $00                    ; Move Backward Command
.equ    TurnR = (1<<EngDirL)               ; Turn Right Command
.equ    TurnL = (1<<EngDirR)               ; Turn Left Command
.equ    Halt = (1<<EngEnR|1<<EngEnL)         ; Halt Command

;==============================================================
; NOTE: Let me explain what the macros above are doing.
; Every macro is executing in the pre-compiler stage before
; the rest of the code is compiled.  The macros used are
; left shift bits (<<) and logical or (|).  Here is how it
; works:
;    Step 1.   .equ   MovFwd = (1<<EngDirR|1<<EngDirL)
;    Step 2.      substitute constants
;              .equ   MovFwd = (1<<4|1<<7)
;    Step 3.      calculate shifts
;              .equ   MovFwd = (b00010000|b10000000)
;    Step 4.      calculate logical or
;              .equ   MovFwd = b10010000
; Thus MovFwd has a constant value of b10010000 or $90 and any
; instance of MovFwd within the code will be replaced with $90
; before the code is compiled.  So why did I do it this way
; instead of explicitly specifying MovFwd = $90?  Because, if
; I wanted to put the Left and Right Direction Bits on different
; pin allocations, all I have to do is change thier individual
; constants, instead of recalculating the new command and
; everything else just falls in place.
;==============================================================

;*************************************************************
;* Beginning of code segment
;*************************************************************
.cseg
```

```

;-----------------------------------------------------------
; Interrupt Vectors
;-----------------------------------------------------------
.org    $0000                   ; Reset and Power On Interrupt
        rjmp    INIT            ; Jump to program initialization

.org    $0056                   ; End of Interrupt Vectors
;-----------------------------------------------------------
; Program Initialization
;-----------------------------------------------------------
INIT:
    ; Initialize the Stack Pointer (VERY IMPORTANT!!!!)
        ldi     mpr, low(RAMEND)
        out     SPL, mpr        ; Load SPL with low byte of RAMEND
        ldi     mpr, high(RAMEND)
        out     SPH, mpr        ; Load SPH with high byte of RAMEND

    ; Initialize Port B for output
        ldi     mpr, $FF        ; Set Port B Data Direction Register
        out     DDRB, mpr       ; for output
        ldi     mpr, $00        ; Initialize Port B Data Register
        out     PORTB, mpr      ; so all Port B outputs are low

    ; Initialize Port D for input
        ldi     mpr, $00        ; Set Port D Data Direction Register
        out     DDRD, mpr       ; for input
        ldi     mpr, $FF        ; Initialize Port D Data Register
        out     PORTD, mpr      ; so all Port D inputs are Tri-State

        ; Initialize TekBot Forward Movement
        ldi     mpr, MovFwd     ; Load Move Forward Command
        out     PORTB, mpr      ; Send command to motors

;-----------------------------------------------------------
; Main Program
;-----------------------------------------------------------
MAIN:
        in      mpr, PIND       ; Get whisker input from Port D
        andi    mpr, (1<<WskrR|1<<WskrL)
        cpi     mpr, (1<<WskrL) ; Check for Right Whisker input
                                ;(Recall Active Low)
        brne    NEXT            ; Continue with next check
        rcall   HitRight        ; Call the subroutine HitRight
        rjmp    MAIN            ; Continue with program
NEXT:   cpi     mpr, (1<<WskrR) ; Check for Left Whisker input
```

```
                                ;(Recall Active Low)
        brne    MAIN            ; No Whisker input, continue program
        rcall   HitLeft         ; Call subroutine HitLeft
        rjmp    MAIN            ; Continue through main

;***********************************************************
;* Subroutines and Functions
;***********************************************************

;----------------------------------------------------------------
; Sub:  HitRight
; Desc: Handles functionality of the TekBot when the right whisker
;       is triggered.
;----------------------------------------------------------------
HitRight:
        push    mpr             ; Save mpr register
        push    waitcnt         ; Save wait register
        in      mpr, SREG       ; Save program state
        push    mpr             ;

        ; Move Backwards for a second
        ldi     mpr, MovBck     ; Load Move Backward command
        out     PORTB, mpr      ; Send command to port
        ldi     waitcnt, (WTime<<1)     ; Shifted bit back by 1,
                                        ; making the wait time two seconds
        rcall   Wait            ; Call wait function

        ; Turn left for a second
        ldi     mpr, TurnL      ; Load Turn Left Command
        out     PORTB, mpr      ; Send command to port
        ldi     waitcnt, WTime  ; Wait for 1 second
        rcall   Wait            ; Call wait function

        ; Move Forward again
        ldi     mpr, MovFwd     ; Load Move Forward command
        out     PORTB, mpr      ; Send command to port

        pop     mpr             ; Restore program state
        out     SREG, mpr       ;
        pop     waitcnt         ; Restore wait register
        pop     mpr             ; Restore mpr
        ret                     ; Return from subroutine

;----------------------------------------------------------------
; Sub:  HitLeft
; Desc: Handles functionality of the TekBot when the left whisker
```

```
;           is triggered.
;-----------------------------------------------------------
HitLeft:
            push    mpr             ; Save mpr register
            push    waitcnt             ; Save wait register
            in      mpr, SREG       ; Save program state
            push    mpr             ;

            ; Move Backwards for a second
            ldi     mpr, MovBck ; Load Move Backward command
            out     PORTB, mpr  ; Send command to port
            ldi     waitcnt, (WTime<<1) ; Wait for 1 second
            rcall   Wait                ; Call wait function

            ; Turn right for a second
            ldi     mpr, TurnR  ; Load Turn Left Command
            out     PORTB, mpr  ; Send command to port
            ldi     waitcnt, WTime  ; Wait for 1 second
            rcall   Wait                ; Call wait function

            ; Move Forward again
            ldi     mpr, MovFwd ; Load Move Forward command
            out     PORTB, mpr  ; Send command to port

            pop     mpr     ; Restore program state
            out     SREG, mpr   ;
            pop     waitcnt     ; Restore wait register
            pop     mpr     ; Restore mpr
            ret             ; Return from subroutine

;-----------------------------------------------------------
; Sub:  Wait
; Desc: A wait loop that is 16 + 159975*waitcnt cycles or roughly
;       waitcnt*10ms.  Just initialize wait for the specific amount
;       of time in 10ms intervals. Here is the general eqaution
;       for the number of clock cycles in the wait loop:
;           (((((3*ilcnt)-1+4)*olcnt)-1+4)*waitcnt)-1+16
;-----------------------------------------------------------
Wait:
            push    waitcnt             ; Save wait register
            push    ilcnt               ; Save ilcnt register
            push    olcnt               ; Save olcnt register

Loop:       ldi     olcnt, 224          ; load olcnt register
OLoop:      ldi     ilcnt, 237          ; load ilcnt register
ILoop:      dec     ilcnt               ; decrement ilcnt
```

```
        brne    ILoop           ; Continue Inner Loop
        dec     olcnt       ; decrement olcnt
        brne    OLoop           ; Continue Outer Loop
        dec     waitcnt     ; Decrement wait
        brne    Loop            ; Continue Wait loop

        pop     olcnt       ; Restore olcnt register
        pop     ilcnt       ; Restore ilcnt register
        pop     waitcnt     ; Restore wait register
        ret             ; Return from subroutine
```

Listing 2: C Bump Bot Script

```
/*
 * Lab1C.c
 *
 * Created: 1/14/2023 12:51:47 PM
 * Author : Astrid Delestine and Lucas Plaisted
 */

/*
This code will cause a TekBot connected to the AVR board to
move forward and when it touches an obstacle, it will reverse
and turn away from the obstacle and resume forward motion.

PORT MAP
Port B, Pin 5 -> Output -> Right Motor Enable
Port B, Pin 4 -> Output -> Right Motor Direction
Port B, Pin 6 -> Output -> Left Motor Enable
Port B, Pin 7 -> Output -> Left Motor Direction
Port D, Pin 5 -> Input -> Left Whisker
Port D, Pin 4 -> Input -> Right Whisker
*/

#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>

// Led final integer values

const int FORWARD = 0b10010000,
HALT = 0b11110000,
BACKWARD = 0b00000000,
 RIGHT = 0b00010000,
 LEFT = 0b10000000;

```

```
void BotActionL();
void BotActionR();
void goBackwards2Sec();

int main(void)
{
    DDRB = 0b11110000; // set 7-4th bits as outputs
    //PORTB = 0b01100000; // turn on LEDs connected to 5-6th bits
    DDRD = 0b00000000; // set 5th and 4th pins on D as inputs
    PORTD = 0b11110000; //enable pull up resistors for port D pins 7-4


    while (1) // loop forever
    {
        // read and extract only 4-5 th bit
        uint8_t mpr = PIND & 0b00110000;
        mpr = ~mpr; //flip bits since PINDD is active low
        if (mpr & 0b00010000) // check if the right whisker is hit
        {
            BotActionR(); // call BotAction
        }
        else if (mpr & 0b00100000) // check if the left whisker is hit
        {
            BotActionL(); // call BotAction
        }
        PORTB = FORWARD; //resume forward movement
        _delay_ms(50); //delay for 50ms to help prevent switch bouncing
    }
}


void BotActionL(){
    goBackwards2Sec(); //self explanatory
    //left motor forwards, right motor backwards = turn right
    PORTB = LEFT;
    _delay_ms(1000); //wait 1 second
    return;
}

void BotActionR(){
    goBackwards2Sec(); //self explanatory :)
    //right motor forwards, left motor backwards = turn left
    PORTB = RIGHT;
    _delay_ms(1000); //wait 1 second
    return;
}
```

```

void goBackwards2Sec(){
    PORTB = BACKWARD; //turn both motors to reverse
    _delay_ms(2000); //delay for 2 seconds
    return;
}
```