# ECE 375 LAB 2

C Code

**Lab Time: Thursday 1000-1200**

*Eric Prather*

## PRELAB QUESTIONS

The following questions and answers concern the ATmega128 I/O pins.

*1. 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?*
The DDRB (DDRx + B) register is used to make this configuration. To configure all 8 outputs in this mode, the value b11111111 must be written to it. DDRx is short for "Data Direction Register X", where 1 is output and 0 is input. As a caveat, all pull-ups are disable is SFIOR's "pull up disable bit" is on.

*2. Suppose all 8 of Port D's pins have been configured as inputs. Which I/O register must be used to read the current state of Port D's pins?*
In this case, the PIND register must be read from to determine the values on the input ports.

*3. Does the function of a PORTx register differ depending on the setting of its corresponding DDRx register? If so, explain any differences.*
Yes: The PORTx pins have two functions depending on the corresponding DDRx pins:
* If the DDRx bit is unset, the corresponding PORTx bit determines if a pullup is present.
* If the DDRx bit is set, the corresponding PORTx bit determines the driven pin value.

## INTRODUCTION

Modern programming languages can be compiled in a variety of different ways depending on the settings passed into a compiler. Even though in previous labs and future labs in this class, there will be an emphasis on the writing and compilation of assembly code, it is also possible to write instructions for systems such as the ATmega128 using a higher-level language such as C. This lab demonstrates this concept by requiring the development of a simple robot locomotion program, like Lab 1, but written in C instead of assembly. Special attention is paid to the specific advantages and disadvantages of coding in C. Conducting this lab and critical analysis of the processes involved substantiate the need for further understanding of- and development in- assembly instead of higher-level programming languages in certain situations. It also helps to bridge the gap between higher-level and lower-level programming.

While it is not a required part of this lab, it is possible to compile C to assembly code, which would allow two implementations of an algorithm to be compared side-by-side. Following this lab, it is reasonable to guess that the compiler's interpretation of C code produces "worse" assembly code than a human directly writing assembly code.

## PROGRAM OVERVIEW

The program written, "BumpBot", allows the mega128 board to control a "TekBot" and make it move forward until it collides with an object. Its behavior then diverges based on whether it is executing the challenge or standard code. In the standard version, after experiencing a collision, the TekBot will back up and turn away from the obstacle collided with (based on whether the collision was detected by its left or right bumper). However, in the challenge version of the code, the Tekbot instead attempts to *push* the object collided with.

Unlike other labs in this course, this lab is written in C, rather than assembly. As such, design choices were made at a higher level of abstraction of the mega128 boards' internal processes. The software "Atmel Studio" provides a

**Commented [PEA1]:** The lab write-up should be done in the style of a professional report/white paper. Proper headers need to be used and written in a clean, professional style. Proof read the report to eliminate both grammatical errors and spelling. The introduction should be a short 1-2 paragraph section discussing what the purpose of this lab is. This is not merely a copy from the lab handout, but rather your own personal opinion about what the object of the lab is and why you are doing it. Basically, consider the objectives for the lab and what you learned and then briefly summarize them. For example, a good introduction to lab 1 may be as follows.
The purpose of this first lab is to provide an introduction on how to use AVRStudio4 software for this course along with connecting the AVR board to the TekBot base. A simple pre-made "BumpBot" program was provided to practice creating a project in AVRStudio4, building the project, and then using the Universal Programmer to download the program onto the AVR board.

**Commented [PEA2]:** This section provides an overview of how the assembly program works. Take the time to write this section in a clear and concise manner. You do not have to go into so much detail that you are simply repeating the comments that are within your program, but simply provide an overview of all the major components within your program along with how each of the components work. Discuss each of your functions and subroutines, interesting program features such as data structures, program flows, and variables, and try to avoid nitty-gritty details. For example, simple state that you "First initialized the stack pointer," rather than explaining that you wrote such and such data values to each register. These types of details should be easily found within your source code. Also, do not hesitate to include figures when needed. As they say, a picture is worth a thousand words, and in technical writing, this couldn't be truer. You may spend 2 pages explaining a function which could have been better explained through a simple program-flow chart.

direct C to machine binary pipeline which is utilized- although this process could hypothetically be interrupted to retrieve assembly instructions (but this is out of the scope of this lab).

In order to correctly execute, this program requires interfacing with the various port mappings as they relate to the TekBot. These mappings are provided on the Lab website. For this lab, we are also provided with two critical resources from which code is derived: The "skeleton code" and the "DanceBot.c" project file.

First, this program calls init_io(). Then, there is a loop in "main()" that calls the "Update()" method repeatedly. This branches into conditional logic driven by two pins on port. The behavior executed depends on whether the source code is the challenge version or the basic version. The basic version mimics the bump-bot routine from Lab 1, whereas the challenge version does a similar behavior that attempts to "push" objects rather than turning *away* from them.

### INIT_IO()

Defines IO registers and initializes their values.

### MAIN()

Calls init_io() and then repeatedly calls update()

### UPDATE()

Branches based on bumper input.

No bumper hit:

Drive forward for one cycle

Left or right bumper hit:

**Either** reverse and turn away from the hit bumper (left/right) ***or*** push in the direction of the hit bumper (decided by whether the code is running in the base or challenge version).

### REVERSE_STEP()

Go backwards for 1000ms

### REVERSE_STEP_SHORT()

Go backwards for 450ms

## ADDITIONAL QUESTIONS

*1. 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 mega128 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.*

Higher level programming languages serve an essential function in modern software development, despite some of their inefficiencies. C, for example, has a rigid standard that persists between many different platforms, and the programmer can generally rely on the compiler to do a certain set of exact behaviors with the syntax. Additionally, higher level languages have several useful abstractions that can improve code legibility and cut down development costs. This makes it useful for enterprise solutions where the system is to be architecture-independent or when a high volume of code needs to be produced in a short period of time. If optimal program size and efficiency aren't required, it is typically far easier to work in a higher-level programming language such as C.

However, working in a language more abstract than assembly introduces another layer of interpretation by a compiler. As such, the code can be mangled or otherwise made less efficient due to compiler implementations. Furthermore, if an architecture is specifically designed to perform certain operations better than others (or has other notable emergent properties), this cannot be explicitly taken advantage of by C. Since C can be cross-compiled, it does not assume any particular strengths or weaknesses of various architectures besides the broadest level of architecture properties such as address sizes.

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

The size of the C program is _B, whereas the size of the assembly program from lab 1 is 497B. While it would be very difficult to make an exact assertion about the cause of this without a detailed knowledge of the compiler, some assumptions can be made based on the difference between higher and lower level programming to explain this difference. Firstly, the C compiler may be imperfect and not translate C code into machine code in the most optimal level. However, even if the C compiler *were* ideal, it still must work with the abstractions of C as a programming language and cannot make perfect optimizations without breaking C standards. As such, assembly code will always have the potential to write more compact and efficient code than C, as it is unbound by the limitations of translating abstract code to functional code. Another reason for the difference may be that C code will typically assign data RAM that an assembly program would be able to keep in a register for longer. A good compiler may be able to automatically optimize C code in this regard, but for reasons discussed in Question #1, this will never be an ideal operation.

## DIFFICULTIES

During my first time compiling the program, I did not call my initialization routine. This resulted in the output registers being low.

I could not change my output values from what was set in init_io(). I thought that my issue was with the input whiskers, but those might have been fine all along- I just couldn't see my output.

I misunderstood the PIND and PINDX macros.

My biggest misunderstanding was that the whisker buttons were active low, not active high.

## CONCLUSION

In this lab, it was a major expected outcome to learn about the compilation pipeline. This ended up being the case, with lots of C code compilation and library references. However, concurrently and independently of this, the work

also evaluated skills related to I/O ports and their correct usage. Overall, the skills built in this lab will be highly applicable to future labs thanks to the pertinence of port usage.

## SOURCE CODE

**Important:** Challenge code is included as its own separate section following Source Code.

**Important:** The behavior when both whiskers are pressed defaults to the behavior of when the left whisker is pressed. I meant to document this in the code for the source but did not leave a comment.

**Important:** It is assumed that the motors are active high in the source code. **However**, in the challenge code, it is assumed that the motors are active low.

```
/*
; Eric Prather (932580666): prathere@oregonstate.edu
; January 15, 2019
File: main.c
Version: sourcecode

The following text is provided by the skeleton code framework and provides
a useful reference for port mapping:

        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 4 -> Output -> Right Motor Enable
        Port B, Pin 5 -> Output -> Right Motor Direction
        Port B, Pin 7 -> Output -> Left Motor Enable
        Port B, Pin 6 -> Output -> Left Motor Direction
        Port D, Pin 1 -> Input -> Left Whisker
        Port D, Pin 0 -> Input -> Right Whisker
*/

// Frequency of the CPU:
#define F_CPU 16000000

// Board library:
#include <avr/io.h>

// C libraries:
#include <util/delay.h>
#include <stdio.h>

// Useful Macros
// Active low.
#define LEFT_BUMBER_PRESSED !bit_is_set(PIND, PD1)// (PIND & 0x01)
#define RIGHT_BUMBER_PRESSED !bit_is_set(PIND, PD0)//(PIND & 0x02)

// Initialize IO registers
// Uses macros defined in <avr/io.h>
// Manual: https://www.nongnu.org/avr-libc/user-manual/io_8h_source.html
// Therefore our specific IO file is avr/iom128.h, available at:
// https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/iom128.h
// Don't import this file directly though. Just reference it to know the macros you need.
void init_io()
{
        // Port B: Motors
        // IMPORTANT: The endianness of these registers is not consistent: Typically they are:
        // [7, 6, 5, 4, 3, 2, 1, 0]
        // or maybe they are just active low
        // Utilized motors are output, un-utilized pins are input.
        DDRB = 0b11110000; // 1 = output, 0 = input
```

```
	PORTB = 0b11110000; // Meaning varies based on io
	// Port D: Bumpers
	DDRD = 0b11111100; // Might as well all be input
	PORTD = 0b11111100; // Put pullup on unused pins, no pullup on used pins
	// However, the behaviors isn ot different based on whether pullup is enabled on the
whiskers.
	// So should PORTD set the pullup or not?
	// Shouldn't I be using `PORTX |= (1<<PX#) | ...`?
}


// Drive using intuitive true/false values (instead of whatever the motor needs)
void drive(int leftOn, int leftDir, int rightOn, int rightDir)
{
	//This doesn't work.
	//PORTB |= (leftOn<<PB7) || (leftDir<<PB6) || (rightOn<<PB4) || (rightDir<<PB5);
	if(leftOn)
		PORTB |= (1<<PB7);
	else
		PORTB &= ~(1<<PB7);
	if(rightOn)
		PORTB |= (1<<PB4);
	else
		PORTB &= ~(1<<PB4);
	if(leftDir)
		PORTB |= (1<<PB6);
	else
		PORTB &= ~(1<<PB6);
	if(rightDir)
		PORTB |= (1<<PB5);
	else
		PORTB &= ~(1<<PB5);
}

// Go backwards for 1000 seconds
// Formerly this function took a parameter, but _delay_ms must only have a compile time constant
void reverse_step()
{
	drive(1,0,1,0);
	_delay_ms(1000); // Yield
}


void Update()
{
	if(LEFT_BUMBER_PRESSED) // left
	{
		reverse_step();
		drive(1,1,1,0); // turn right
		_delay_ms(1000);
	}
	else if(RIGHT_BUMBER_PRESSED)
	{
		reverse_step();
		drive(1,0,1,1); // turn right
		_delay_ms(1000);
	}
	else // Just go forward
		drive(1,1,1,1);
}

// Main method called at program start
int main(void) // No arguments
{
	init_io();
	while (1) // loop forever
	{
		Update();
	}
	return 1; // If we reach this, an error has occured (break; in while loop)
}
```

## CHALLENGE CODE

**Important**: It is assumed that the motors are **active low**.

```
/*
; Eric Prather (932580666): prathere@oregonstate.edu
; January 15, 2019
File: main.c
Version: sourcecode

IMPORTANT: MOTORS ARE ASSUMED To BE ACTIVE LOW

The following text is provided by the skeleton code framework and provides
a useful reference for port mapping:

        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 4 -> Output -> Right Motor Enable
        Port B, Pin 5 -> Output -> Right Motor Direction
        Port B, Pin 7 -> Output -> Left Motor Enable
        Port B, Pin 6 -> Output -> Left Motor Direction
        Port D, Pin 1 -> Input -> Left Whisker
        Port D, Pin 0 -> Input -> Right Whisker
*/

// Frequency of the CPU:
#define F_CPU 16000000

// Board library:
#include <avr/io.h>

// C libraries:
#include <util/delay.h>
#include <stdio.h>

// Useful Macros
// Active low.
#define LEFT_BUMBER_PRESSED !bit_is_set(PIND, PD1)// (PIND & 0x01)
#define RIGHT_BUMBER_PRESSED !bit_is_set(PIND, PD0)//(PIND & 0x02)

// Initialize IO registers
// Uses macros defined in <avr/io.h>
// Manual: https://www.nongnu.org/avr-libc/user-manual/io_8h_source.html
// Therefore our specific IO file is avr/iom128.h, available at:
// https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/iom128.h
// Don't import this file directly though. Just reference it to know the macros you need.
void init_io()
{
        // Port B: Motors
        // IMPORTANT: The endianness of these registers is not consistent: Typically they are:
        // [7, 6, 5, 4, 3, 2, 1, 0]
        // or maybe they are just active low
        // Utilized motors are output, un-utilized pins are input.
        DDRB = 0b11110000; // 1 = output, 0 = input
        PORTB = 0b01100000; // Meaning varies based on io
        // Port D: Bumpers
        DDRD = 0b11111100; // Might as well all be input
        PORTD = 0b11111100; // Put pullup on unused pins, no pullup on used pins
        // However, the behaviors isn ot different based on whether pullup is enabled on the
whiskers.
        // So should PORTD set the pullup or not?
        // Shouldn't I be using `PORTX |= (1<<PX#) | ...`?
}


// Drive using intuitive true/false values (instead of whatever the motor needs)
```

```
void drive(int leftOn, int leftDir, int rightOn, int rightDir)
{
	//This doesn't work.
	//PORTB |= (leftOn<<PB7) || (leftDir<<PB6) || (rightOn<<PB4) || (rightDir<<PB5);
	if(leftOn)
		PORTB |= (1<<PB7);
	else
		PORTB &= ~(1<<PB7);
	if(rightOn)
		PORTB |= (1<<PB4);
	else
		PORTB &= ~(1<<PB4);
	if(leftDir)
		PORTB |= (1<<PB6);
	else
		PORTB &= ~(1<<PB6);
	if(rightDir)
		PORTB |= (1<<PB5);
	else
		PORTB &= ~(1<<PB5);
}

// Go backwards for 1000 seconds
// Formerly this function took a parameter, but _delay_ms must only have a compile time constant
void reverse_step()
{
	drive(0,0,0,0);
	_delay_ms(1000); // Yield
}

void short_reverse_step()
{
	drive(0,0,0,0);
	_delay_ms(450); // Yield
}

void Update()
{
	if(LEFT_BUMBER_PRESSED) // left
	{
		drive(0,1,0,1);
		_delay_ms(400);
		short_reverse_step();
		drive(0,0,0,1);
		_delay_ms(400);
	}
	else if(RIGHT_BUMBER_PRESSED)
	{
		drive(0,1,0,1);
		_delay_ms(400);
		short_reverse_step();
		drive(0,1,0,0);
		_delay_ms(400);
	}
	else // Just go forward
		drive(0,1,0,1);
}

// Main method called at program start
int main(void) // No arguments
{
	init_io();
	while (1) // loop forever
	{
		Update();
	}
	return 1; // If we reach this, an error has occured (break; in while loop)
}
```