# ECE 375: INTRODUCTION TO COMPUTER ORGANIZATION AND ASSEMBLY

## A TekBots Course

## Internet Access

We maintain Internet systems for your use. They can be used to obtain free TekBots' software and documentation and also to purchase TekBots' products. These systems may also be used to communicate with members of TekBots and other customers. Access information is shown below:

| | |
|---|---|
| E-mail: | tekbots@eecs.oregonstate.edu |
| Web: | http://eecs.oregonstate.edu/education/tekbots.html |

# TABLE OF CONTENTS

# PREFACE

## HOW TO USE THIS MANUAL

During this course, you will work with components and circuits that you will use in your own engineering careers. The objective is to show the non-electrical engineer how electronics is important to other engineering fields. You will be designing sensors and sensor circuits to measure real phenomena—not just arbitrary circuits.

Everything learned in lecture has relevance and is useful in later courses and future careers. As various tasks are performed in these labs, pay attention to how lecture material relates to the tasks. Understanding how the lecture material is used and applied will greatly improve understanding of the topics.

## IMPORTANT SYMBOLS

This symbol indicates an important note that should be remembered. Paying attention to notes like these will make tasks easier and more efficient.

This symbol designates information that must be followed. If caution is ignored the task may appear impossible. Ignored caution symbols can lead to damaged systems.

This symbol represents something that should not be forgotten. Reminder symbols are used to make sure an important step has been completed before continuing.

The innovation symbol indicates an opportunity to advance beyond what is required. These sections will give more insight into the what, why, and how of a certain topic. Use these to learn more or get ideas for nifty innovations.

## LAB STRUCTURE

| | |
|---|---|
| **Section Overview** | The section overview will describe briefly what will be learned in the section and what will be done. |
| **Procedure** | The procedure portion of each section contains all of the tasks to be completed. All tasks are related to lecture. Keeping this in mind will help in better understanding the lecture and lab material. |
| **Study Questions** | The study questions are intended to give more practice and insight into what has been learned in lab and lecture. The study questions will be due in the next week's lab. |
| **Challenges** | The challenge sections of labs are extra credit. Performing the tasks in the challenge sections will improve understanding of what is being learned and will result in some really cool innovations. |

## LAB SAFETY

Safety is always important when working with electricity and electronics. This includes both the safety for you as well as safety for the circuit components you are working with. Concerns such as high voltage or currents can affect the human body, while static safety and proper component use can affect the life of your circuits.

### Personal Safety

When working with high voltages and currents, it is important that you remember that you can be hurt, if your body becomes the 'circuit', since the human body is a conductor of electricity.. This issue has long been combated by using the 'one hand rule.' Whenever you are working with a potentially dangerous circuit, turn it off, but if it cannot be turned off, use only one hand when working on it. This will prevent a circuit from being made through your heart, which could be potentially fatal.

### Component Safety

Many electrical components are likely to be damaged by static electricity. Static charge can build up to many thousands of volts, but with little energy. This cannot harm humans, but it can easily damage electronic components. To ensure static-safe handling, the best practice is to wear an anti-static strap and connect it to an earth ground such as a computer case or a water pipe. If you do not have an anti-static wristband, you can instead touch a ground every few minutes to discharge your static build up.

# SECTION ONE
## INTRODUCTION TO AVR DEVELOPMENT TOOLS

## SECTION OVERVIEW

**Complete the following objectives:**

- Connect your AVR microcontroller board to a TekBot.
- Create a new AVRStudio4 project.
- Download and compile the sample AVR Assembly source code.
- Understand how to run and operate the Universal Programmer.
- Upload and run the sample program on the TekBots AVR microcontroller board.

## PRELAB

In labs to come you will be required to complete a prelab for each lab. The prelab will cover concepts and knowledge that is required for the lab. The prelabs are due at the beginning of your lab section each week. If you do not have your prelab done at the beginning of the lab period you will receive no credit for the prelab. Prelabs are to be submitted on paper, not by email.
**For this lab, no prelab is required.**

## PROCEDURE

**Wiring Your TekBot**

1. The first step is to look at the wiring diagrams available on the TekBots webpage.
To make the wires for connecting your TekBot boards together you will need to use the ribbon cable that came with your kit with male headers soldered to the ends and a bit of heat shrink tubing on the connections. There is a short tutorial on the web that explains this process in detail.
2. Because we are working with a modular programmable AVR microcontroller board, you can connect the whisker inputs and motor controller outputs to any pins on any port on the board. But for the demo lab to work properly, these cables need to be connected in a certain way. This configuration should be used throughout the course. It uses the pin's alternative functions rather than its primary functions. As the course progresses, you will learn what both of these functions are. Table 1 (next page) shows the proper pin connections.

| Connection | Port | Pin | Alternative Function |
|---|---|---|---|
| Right Whisker | E | 4 | External Interrupt 4 |
| Left Whisker | E | 5 | External Interrupt 5 |
| Right Motor Enable | B | 4 | PWM Output for Timer/Counter0 |
| Right Motor Direction | B | 5 | PWM Output A for Timer/Counter1 |
| Left Motor Direction | B | 6 | PWM Output B for Timer/Counter1 |
| Left Motor Enable | B | 7 | PWM Output for Timer/Counter2 |

Table 1: TekBots Connections to AVR Microcontroller Board

3. When you have completed the wiring, be sure to ask your TA to look it over before you turn the power on. If you have wired things incorrectly (especially if you have switched Vcc and ground), you WILL destroy parts of your TekBot. Please have the TA initial in the space below that you have shown your TekBot to them.

TA Initials: _________

## Looking at some AVR Source Code

1. Download the sample code available on the webpage. This is a simple AVR source code that will run your TekBot in the simple BumpBot fashion. All code that you produce should be as well commented as this sample code. Save this code where you can find it in your UNIX file share network drive (commonly called drive Z:)
2. AVRStudio4 is the Integrated Development Environment (IDE) that you will be using to develop your AVR assembly code throughout the remainder of the course (with the exception of Lab 2). AVRStudio4 is a powerful IDE created by Atmel for their line of AVR microcontrollers. You will be using it to write assembly programs for your AVR microcontroller board that uses an ATmega128 microcontroller. Section 2 of the AVR Starter Guide, which can be found on the TekBots website, contains a good overview on how to use the program as well as some step-by-step tutorials. Briefly read through this section to gain a basic understanding of the IDE.

   (Optional) If the IDE is not already installed on the computer, or if you are doing this lab from home, read through and follow the steps in Section 2.1.1 of the AVR Starters Guide to download and install AVRStudio4 on your computer.

3. Follow the steps in Section 2.1.2 in the AVR Starters Guide to create a new project. In most IDE tools, a project is the base starting area to your program. It consists of all files you use and any settings for the program. When following this tutorial, you'll want to use the AVR assembly source code you downloaded from the web in step 1.
4. With the current project activated in step 4, follow the Project Simulation tutorial in Section 2.1.3 of the AVR Starters Guide to learn how to compile and simulate your program. By the end of this step, you should know how to successfully create an AVR project from scratch and be able to compile it into usable program hex code.
5. When assembly source code is compiled, it creates a binary program file (called a HEX file with a .hex extension). This HEX file contains the actual binary instructions that are used by the ATmega128 and is what needs to be uploaded onto the AVR Microcontroller Board. Unfortunately, AVRStudio4 does not have the ability to program the actual chip. To do this, you will need to run the Universal Programmer. This is an open source program that can upload the AVR HEX programs to the various AVR microcontrollers.
6. Follow the tutorials in Section 3 of the AVR Starter to install PonyProg and upload the AVR program into the ATmega128. Use the HEX file that was generated by AVRStudio4 from the previous steps.

HINT: When testing and debugging a program, it is useful to keep both AVRStudio4 and PonyProg running at the same time. Load only the HEX file the first time you start PonyProg. If you make changes to the code, simple recompile and then ALT-TAB over to PonyProg and hit the Reload files button. This will update the HEX file within PonyProg to reflect any changes made to the program. Then upload the code to the chip again.

7. With the program uploaded into the microcontroller, unplug the TekBot from the computer and turn it on. Observe the behavior; the TekBot should be operating in the similar BumpBot fashion. Demonstrate your TekBot to your TA.

TA Initials: __________

Theory of Operation for Lab 1 AVR Assembly Code.

- Initializes key components of the ATmega128
- Starts the TekBot moving forward
- Polls the whiskers for input
- If right whisker is hit
    - Backs up for a second
    - Turns left for a second
    - Continues Forward
- If left whisker is hit
    - Backs up for a second
    - Turns right for a second
    - Continues Forward

## STUDY QUESTIONS/ REPORT

For this lab, there will be no write up. This lab is simply for familiarization and introduction to AVR Studio and PonyProg. You will automatically get 20 points for this lab for obtaining the TA's initials as indicated.
For all other labs, you will write up short summaries that detail what you did and why you did it. Explain any problems you may have had, and answers some questions that are specific to each lab. The hard copy of write up, along with your code should, be submitted to your TA by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED.

The write up should be typed. It should be neat and free of misspelled words. Be sure you have used sufficient comments for ANOTHER STUDENT to be able to understand your code. Code that is not well documented will be penalized severely.

If you are interested in an example of what this might look like, look at the TekBots web page or the code you just downloaded.

### Study Questions:

Labs will commonly have study questions for you to think about while performing the lab. You will need to answer these questions within your lab write up. This lab has no lab write up so there are no study questions.

## CHALLENGE

Challenge problems are additional tasks that can be preformed for the labs. By successfully completing a challenge problem you can get extra credit for your lab. To get credit for the challenge problem you must successfully demonstrate it to your TA as well as document it in your lab write up.

1. You have learned to use a simple tool to download a program into your TekBot. Modify the program so the TekBot reverses twice as long before turning away and resuming forward motion. Demonstrate this to your TA and be turn in a copy of you modified assembler program

# SECTION TWO C → ASSEMBLER → MACHINE CODE → TEKBOT

## SECTION OVERVIEW

**Complete the following objectives:**

- Look at a sample C program
- Write a sample C program for the AVR microcontroller
- Compile the code using the CodeVision IDE and compiler
- Download this code to your AVR board and verify its function

## PRELAB

Remember—no late prelabs will be accepted.

Write a pseudo code program for making your TekBot move the way it did in Lab 1. Your robot should detect objects using its whiskers, reverse, and turn away from the object.

## PROCEDURE

### Looking at C Code and the CodeVision Studio

1. Download the sample code available on the web page. This is simple C code that is well commented and ready to compile. All code that you produce should be as well commented as this code. Save this code where you can find it in your UNIX file share network drive (commonly called drive Z:) There are two different programs on the web; be sure to get the correct one for your TekBot.

2. Open the CodeVision Integrated Development Environment (IDE) and select '**New Project**.' You have two options here, either run the 'Wizard' or don't. The wizard for CodeVision will quickly generate values to be stored in specific places in the AVR to setup peripheral devices like the A/D converters and the serial port. We don't need to do this, so don't use the wizard. You are asked to name your project. Put the project file in its own directory with the source code you got from the web. CodeVision creates a lot of files so you don't want to put it just anywhere. After you have named the file, a short 'selection' screen pops up, as shown in Figure 1. Click '**Add**' and add the code you downloaded. Click the top tab called '**Compiler**' to bring up the compiler options.

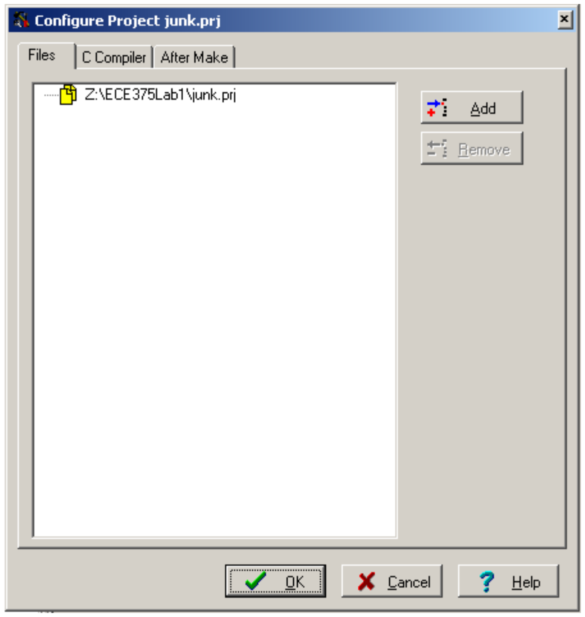
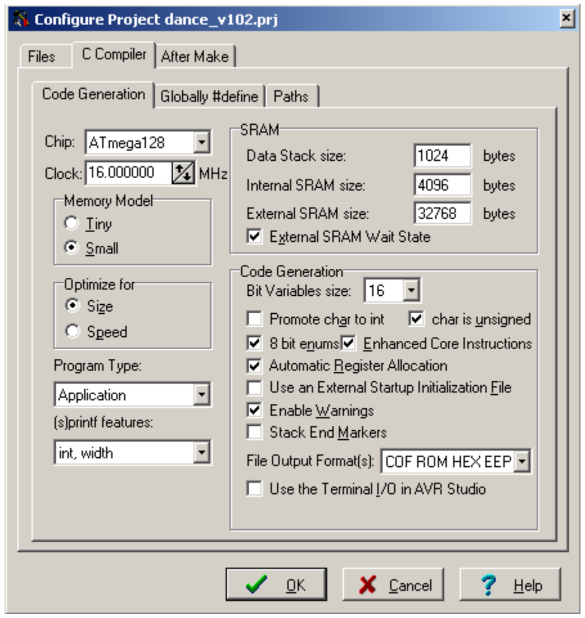


Figure 1: Project Setup Screen

3. Figure 2 shows a properly configured compiler screen. All values are specific for the AVR board you have. This configuration will not necessarily work on other chips or boards. The part is the ATMega128, running at 16Mhz. You are using a 'small' memory model with 32768 bytes of external RAM. The RAM requires a wait state. Everything else is left in its default.



Chip: ATmega128
Clock: 16.000000 MHz

Memory Model
- ○ Tiny
- ● Small

Optimize for
- ● Size
- ○ Speed

Program Type: Application

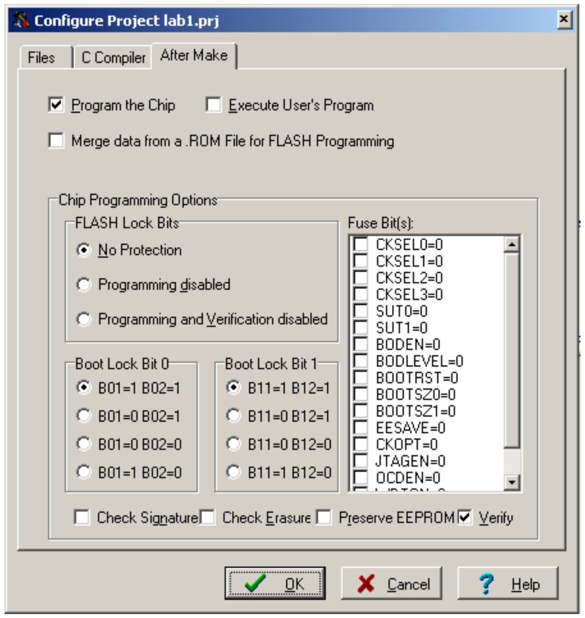
(s)printf features: int, width

SRAM
- Data Stack size: 1024 bytes
- Internal SRAM size: 4096 bytes
- External SRAM size: 32768 bytes
- ☑ External SRAM Wait State

Code Generation
- Bit Variables size: 16
- ☐ Promote char to int
- ☑ char is unsigned
- ☑ 8 bit enums
- ☑ Enhanced Core Instructions
- ☑ Automatic Register Allocation
- ☐ Use an External Startup Initialization File
- ☑ Enable Warnings
- ☐ Stack End Markers
- File Output Format(s): COF ROM HEX EEP
- ☐ Use the Terminal I/O in AVR Studio

Figure 2: Compiler Settings

4. Figure 3 shows the '**After Make**' settings. You want to program the chip after make, and you don't want to verify erasure. DO NOT MESS WITH THE FUSE BITS. Your AVR board can become unusable if you write to the wrong fuse bit without knowing what you are doing.


Configure Project lab1.prj

Files | C Compiler | After Make

[✓] Program the Chip   [ ] Execute User's Program

[ ] Merge data from a .ROM File for FLASH Programming

Chip Programming Options

FLASH Lock Bits
- [✓ No Protection]
- Programming disabled
- Programming and Verification disabled

Boot Lock Bit 0
- [✓ B01=1 B02=1]
- B01=0 B02=1
- B01=0 B02=0
- B01=1 B02=0

Boot Lock Bit 1
- [✓ B11=1 B12=1]
- B11=0 B12=1
- B11=0 B12=0
- B11=1 B12=0

Fuse Bit(s):
- [ ] CKSEL0=0
- [ ] CKSEL1=0
- [ ] CKSEL2=0
- [ ] CKSEL3=0
- [ ] SUT0=0
- [ ] SUT1=0
- [ ] BODEN=0
- [ ] BODLEVEL=0
- [ ] BOOTRST=0
- [ ] BOOTSZ0=0
- [ ] BOOTSZ1=0
- [ ] EESAVE=0
- [ ] CKOPT=0
- [ ] JTAGEN=0
- [ ] OCDEN=0

[ ] Check Signature [ ] Check Erasure [ ] Preserve EEPROM [✓] Verify

OK   Cancel   Help

Figure 3: 'After Make' Settings

5. Click **OK**, and CodeVision should open the source file. Examine this file so you understand what is happening. It is written in 'C' (you should have learned C in CS151, CS261, or elsewhere). If you are having difficulties, check the web and teach yourself or ask for assistance from your TA.

```
Pseudo Code for Lab 2 'Dance' C Code.
    • Initialize Port(s)
    • Loop Forever
        o Forward for 500mS
        o Reverse for 500mS
        o Right for 1000mS
        o Left for 2000mS
        o Right for 1000mS
```

## Compiling C Code and Downloading

1. Now that you understand the code and have connected your TekBot as expected, you will want to compile and download your code to your TekBot. This is where the beauty of an IDE is. In an IDE all you need to do is click a single button rather than use a separate program to compile your code. Select **Project** → **Compile**, or press F9. The code will compile and give you a summary. If there are any errors or warnings, ask your TA for help.
2. After you have compiled your code, it is time to 'make' the files to be downloaded. Select **Project** → **Make** or press shift + F9. Again, a window will appear showing if there were any errors or warnings. If you set up your project as asked, you should just need to press the enter key to program your TekBot. Of course your TekBot needs to be connected to your PC and powered on first.

## Your Own Code

You need to write a simple C program that will make your TekBot act like it did in Lab 1. Your TekBot should travel forward until it encounters and object, then back up and turn away from the object. If for some reason both whiskers are triggered at the same time, your TekBot needs to back up and turn to the RIGHT. HINT: Use the 'delay_ms()' function.

1. Write a simple C program that performs this function. You will probably want to use the skeleton code available on the web and modify it as needed. Remember which version of the TekBot you have when designing the motor control logic. It is recommended that you use the pin out in the skeleton file, but it is not necessary.
2. Demonstrate the operation of your TekBot to your TA for credit. Have him/her sign below.

TA Signature: _________________________

## STUDY QUESTIONS/ REPORT

Write a short summary that details **what you did and why, explain any problems you may have encountered, and answer the questions below**. Your write up and code must be submitted to your TA via email by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED.

## Study Questions

1. This lab required you to begin using new tools for compiling and downloading code to your AVR-enabled TekBot using the C language. Explain why it is beneficial to write code in a language that can be ‘cross compiled.’ Also explain some of the problems of writing in this way.
2. Your program does essentially the same thing as the assembly program you downloaded for Lab 1. Compare the size of the output bin files and explain the differences in size between them.

## CHALLENGE

1. Modify your TekBot so it can move objects across a table top. Your TekBot needs to push objects that it touches a short distance. An example of how your TekBot should work is:

a. TekBot hits object.
b. TekBot continues forwards for a short period of time.
c. TekBot backs up slightly.
d. TekBot turns slightly towards the object.
e. TekBot repeats steps a through d.

There are several ways to approach this problem, but you must accomplish it by revising your firmware (change the C program or rewrite it). To get credit for this challenge, you must demonstrate the operation of your TekBot to your TA and submit a copy of the code used.

# SECTION THREE DATA MANIPULATION AND THE LCD DISPLAY

## SECTION OVERVIEW

**Complete the following objectives:**

- Understand the basics of data manipulation
- Initialize a program (i.e. stack, registers, LCD, etc)
- Use indirect addressing with the X, Y, and Z-pointers
- Read data from program memory
- Move data around in data memory
- Setup and successfully call functions and subroutines

## PRELAB

The answers to the prelab questions can be found in the AVR Starters Guide and the AVR Instruction Set.

- What is the stack pointer? How is the stack pointer used and how do you initialize it? Provide pseudo-code (NOT actual assembly code) that illustrates how to initialize the stack pointer.
- What does the AVR instruction LPM do and how do you use it? Provide pseudo-code that shows how to use the LPM instruction.
- Look at the definition file *m128def.inc*. What is contained within this definition file? What are the benefits of using this definition file? How would you include it into your AVR Assembly program?

## PROCEDURE

### Introduction

For this lab, you will learn to use the LCD Display on the AVR microcontroller board. In order to use the LCD display, you will need to learn to properly initialize your program. You will also learn how to move data from your program memory into your data memory. This data is what will be displayed on the LCD display. To help you along the way, a project skeleton file will be provided. This file will contain some code and comments directing you to what code to write and where. The file will also give you a basis for writing well-structured code as defined in the AVR Starters Guide.

### Initialization

A program initialization consists of any code that is only run once at the beginning of the program. This code does not belong in the MAIN program, this is why most Assembly programs begin with INIT and not MAIN. The INIT function is not a function in the classic sense but rather a function that is called at reset (and at power on) and jumps to the main program once it is finished.

There are several things that should be contained in the INIT function. The first and foremost is the initialization of the stack pointer. The stack in AVR operates in a higher to lower address fashion, meaning that newest element on the stack is in a lower address space than the previous element. Therefore the stack should be initialized to the highest data address space available. See the AVR Starters Guide for more information about the stack.

Other things that are initialized within the INIT function are ports, timers, interrupts, and other peripherals. For this lab, you will need to initialize the LCD display. Making a function call to the LCD Initialization Subroutine does the LCD initialization. You will also need to move data from the program memory to data memory in the INIT function.

### *LCD Driver User Manual*

To successfully use the LCD display for this lab, you will need to properly setup and call the functions in the LCD driver. To do this, you must first include it into your program. Unlike the definition file, the LCD driver contains actual assembly code and thus cannot be included at the beginning of the program file like the definition file. A quick look at the AVR Starters Guide under proper coding structure tells us that any include code file is included at the end of the program, i.e. the last lines. The actual LCD driver file is called *LCDDriver.asm* and can be downloaded from the lab website. There will also be another program file called *LCDTest.asm* that utilizes every function described below. You can download it for a sample of using the LCD.

The following are the LCD driver function definitions. In order for any function call to work, the stack pointer must be initialized prior to the function call.

#### LCDInit

This subroutine initializes the serial interface that is used to communicate with the LCD display, initializes the 'Hitachi Display Chips with 8-bit Incremental DD-RAM Pointer with no features,' and sets the display to 2x16 characters.

```
rcall	LCDInit			; Call LCD Init Subroutine
```

#### LCDWrite

This is a generic function that will write out both lines of text to the LCD display. The line data is fetched from the following AVR data memory addresses:

```
	Line 1:  $0100 - $010F
	Line 2:  $0110 - $011F
```

In order to use this function you must first put the data (i.e., ASCII string) that you want displayed on the LCD in the appropriate line address in data memory. Then call the function.

```
; Move ASCII string to line addresses $0100-$011F
rcall	LCDWrite		; Write string to LCD
```

#### LCDWrLn1

This function will write the ASCII string in the first line AVR data memory address ($0100-$010F) to the first line of the LCD Display.

In order to use this function you must first put the data (i.e., ASCII string) that you want displayed on the LCD display line 1 in the line 1 addresses in the data memory and call the function.

```
; Move ASCII string to line addresses $0100-$010F
rcall    LCDWrLn1           ; Write string to LCD
```

**LCDWrLn2**

This function will write the ASCII string in the second line AVR data memory address ($0110-$011F) to the second line of the LCD display.

In order to use this function you must first put the data (i.e. ASCII string) that you want displayed on the LCD display line 2 in the line 2 addresses in the data memory and call the function.

```
; Move ASCII string to line addresses $0110-$011F
rcall    LCDWrLn2           ; Write string to LCD
```

**LCDClear**

This subroutine will clear both lines of the LCD display and the lines in AVR data memory will be cleared to the ASCII value of ‘ ’ (space). No prior setup is required.

```
rcall    LCDClear           ; Clear both lines of LCD
```

**LCDClrLn1**

This subroutine will clear line 1 of the LCD Display and line 1 in the AVR data memory in the same fashion as LCDClear. No prior setup is required.

```
rcall    LCDClrLn1          ; Clear line 1
```

**LCDClrLn2**

This subroutine will clear line 2 of the LCD display and line 2 in the AVR data memory in the same fashion as LCDClear. No prior setup is required.

```
rcall    LCDClrLn2          ; Clear line 2
```

**LCDWriteByte**

This function allows you to write a single ASCII character or byte anywhere on the LCD display. This allows complete control over where things go within the display and does not require the AVR data memory lines as in the previous functions. There are three registers that need to be initialized prior to calling this function.

count – Holds the index value of the line to where the ASCII Char will be written, 0 – 15 (of 39). Indexes 0 – 15 are visible on the display, but can be up to 39, thus indexes 16 – 39 are off screen. If count has a value of 3 then the ACSII char will be written to the 3rd element on the LCD display.

line – Holds the line number that the char is going to be written to, 1 or 2.

mpr – Contains the value of the ASCII char to be written, 0-255.

```
; Example of writing ‘D’ to Line 2 slot 7
```

```
ldi         mpr, 'D'        ; mpr <- 'D'
ldi         line, 2         ; line <- 2
ldi         count, 7        ; count <- 7
rcall   LCDWriteByte        ; Write Byte to LCD Display
```

### Bin2ASCII

This function will convert an unsigned 8-bit binary number into the numerically equivalent ASCII string, i.e. 186 -> "186". For this function, three registers are needed:

mpr – Contains the 8-bit binary number to be converted.

X-Pointer – Contains the start address to AVR data memory when the ASCII string will be stored.

count – Will contain the number of characters that are written once the function has completed.

This function is useful for printing a number that you don't always know what the value will be (i.e., not a constant value such as a counter). When using this function, be aware that up to three characters could be written. For example, if you had the value 138 in the mpr and the address of $0112 in the X-pointer, you would get the following after calling this function:

```
DataMem($0112) ← '1'
DataMem($0113) ← '3'
DataMem($0114) ← '8'
count ← 3
```

Here is an example of calling the function described above.

```
ldi         mpr, 138                ; mpr <- 138
ldi         XL, low($0112)          ; X <- $0112
ldi         XH, high($0112)         ; X <- $0112
rcall   Bin2ASCII                   ; Call Bin2ASCII
```

There are many other functions within the LCD driver, but they are support functions for those described above and should not be called individually. Modifying any function within the driver is not recommended as it could damage the performance of the function, but feel free to play around if you like. With that said, if you do modify the driver or add another function that improves the performance, let your TA know what you did and how it works or why it is better and we might add it to future versions of the driver.

## Data Manipulation

To be able to move data from one memory to another, you first must understand the memory. The ATmega128 is an 8-bit AVR architecture. This means all data is moved around 8-bits or 1-byte at a time, thus all register and data memory are 8-bit wide.

However, the AVR Op-Code is 16 or 32-bits and for efficiency reasons the program memory is 16-bits or 2-bytes wide. This means that when you read from the program memory, you will have to read 2-bytes for every address space in the program memory.

When writing a program, it is often necessary to include data directly into program memory instead of entering it into the data memory each a time a simulation is run. The .DB (Data Byte) directive allows data to be written directly to the program memory during compilation. For example, the following code demonstrates how to create data in program space:

```
DATA:
.DB     $05, $F4
```

When the program is compiled, the hex data 0x05 and 0xF4 will be located in the program memory at the address specified by the label DATA. To read this data, use the LPM (Load Program Memory) instruction.

> Note: Since the program memory is 16-bits wide, the data after the .DB directive needs to be in multiples of 16-bits. Otherwise, the compiler will insert an extra byte of data, usually $FF.

The movement within data memory is accomplished with the different variations of Load and Store commands. There are two main addressing modes to move the data: direct addressing and indirect addressing. Direct addressing is best when you want to move data to and from a single memory location, like an extended I/O register. Indirect addressing uses the X, Y, and Z-pointers and is useful for moving data to multiple areas of data memory. For example, using indirect addressing in conjunction with a while-loop, you could move 'blocks' of data around in memory. See the *AVR Starters Guide* for more information on pointers and indirect addressing. The following is some pseudo-code you can use to properly read a bunch of data from program memory

```
Z <- Beginning Address of string in Program Memory
Y <- Beginning Address in Data Memory for string
do {
        mpr <- ProgMem(Z), Z++
        DataMem(Y) <- mpr, Y++
} while (Z != End Address of string in Prog Memory)
```

## Writing Your Name

You are to write your name on the first line of the LCD Display and a simple phrase, such as "Hello World!" on the second line of the LCD Display. In order to accomplish this, you must have you two strings in program memory using the .DB directive. Then read the data from program memory and store it into the appropriate spots in data memory that are used for Line 1 and Line 2. Then make a simple call to the correct function to display the two lines on the LCD display. You must first correctly initialize your program.

To help you along the process, skeleton code is provided. Use this as a starting point. When you are finished, demonstrate your LCD Display to the TA to get credit. Have them sign below.

TA Signature: ______________________

## STUDY QUESTIONS/ REPORT

Write a short summary that details **what you did and why, explain any problems you may have encountered, and answer the questions below**. Submit your write up and code to your TA via email by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED.

### Study Questions:

1. In this lab, you were asked to manipulate data by moving it around in memory. In fact, the AVR architecture has two different memories, a program memory and data memory. Briefly explain the differences and purposes of these memories within your write up.
2. You also learned how to make function calls. Explain how the function call works, its connection to the memory stack, and why a RET instruction must be called to return from a function. To help you understand, comment out the stack pointer initialization in the beginning of the code and try running the program. Observe and comment on this behavior.

## CHALLENGE

Not being content with displaying a static message, you would like to add some flare by creating a scrolling marquee style message. Your text will scroll across the lines from left to right. When a character is at the end a line, it will be displayed at the beginning of the opposite line, i.e. a character on slot 16 of line 1 will go to slot 1 of line 2 and a character on slot 16 of line 2 will go to slot 1 of line 1. You will also want to add a wait loop after all the characters have moved one slot so that you can read the characters. Below is an example of the display and motion where '_' signifies a space:

```
1:      Line 1 – "_____My_Name_is_"
        Line 2 – "_______John_Doe_"
wait(.25sec)
2:      Line 1 – "______My_Name_is"
        Line 2 – "________John_Doe"
wait(.25sec)
3:      Line 1 – "e______My_Name_i"
        Line 2 – "s________John_Do"
wait(.25sec)
```

4: Line 1 – “oe______My_Name_”
Line 2 – “is________John_D”
wait(.25sec)
etc...

Use the strings that are required for the lab to scroll. Submit this code with your report. Demonstrate the challenge problem to the TA for credit.

# SECTION FOUR LARGE NUMBER ARITHMETIC

## SECTION OVERVIEW

- Understand and use arithmetic and ALU operations
- Manipulate and handle large numbers
- Create and handle functions and subroutines

## PRELAB

- For this lab, you will be asked to perform arithmetic operations on numbers that are larger than 8-bits. To do this, you should understand the different arithmetic operations supported by the AVR Architecture. List and describe all the different forms of ADD, SUB, and MUL (i.e. ADC, SUBI, MULF, etc.).
- Write pseudo-code that describes a function that will take two 16-bit numbers in data memory addresses \$0110-\$0111 and \$0121-\$0122 and add them together. The function will then store the resulting 16-bit number at the address \$0100-\$0101. (Hint: The upper address in the high byte of the number and don't forget about the carry in bit.)

## PROCEDURE

Arithmetic calculations like addition and subtraction are fundamental operations in many computer programs. Most programming languages support several different data types that can be used to perform arithmetic calculations.
The ATmega128 uses 8-bit registers and has several different instructions to perform basic arithmetic operations on 8-bit operands. Examples of instructions that add or subtract 8-bit registers are:

```
ADD   R0, R1        ; R0 ← R0 + R1
ADC   R0, R1        ; R0 ← R0 + R1 + Carry Bit
SUB   R0, R1        ; R0 ← R0 – R1
```

If we are required to manipulate data with 16 or more bits with AVR instructions then we need to perform intermediate steps to manipulate 8-bit registers to produce the correct result. The following figure demonstrates the addition of two 16-bit numbers. One of the 16-bit numbers is located in registers R0 and R1 and the other number is in R2 and R3. The operation result is placed in the registers R4, R5, and R6.

```
                                 (1)    (1)    ← Possible Carry-In Bit
                                 R0     R1

Possible Carry-Out Bit →  (1)    R2     R3
                        + ----------------------
                          R4     R5     R6
```

As this calculation demonstrates, we need to add R1 to R3 and accommodate for any carry-in bit. The result of this operation is stored in R6. Next, we add R0 to R2 and accommodate for any carry-in bit from the previous operation. If there is a final carry-bit, then it is stored in R4. Note that since we want to reserve the carry out bit from the operation, we actually get a 17-bit result: 16-bit plus the carry. But since the AVR architecture inherently supports 8-bit numbers and we use 3 registers, we treat the result as a 24-bit number, where the most significant byte has the value of either 1 or 0 depending on the carry out bit.

The AVR instruction set contains a special instruction to perform multiplication, MUL. This instruction multiplies two registers and stores the result in registers R0 and R1. Therefore a multiplication of 2 single byte registers generates a 2-byte result. This instruction can be used as a fast and efficient way to multiply two 8-bit numbers. Unfortunately, it can be very complicated to use when multiplying 16 or more bit numbers.

The easiest way to understand how to multiply large numbers is to visualize it like using the paper method to multiply, this method is also known as the sum of products technique. The following figure illustrates the typical paper method.

```
   24
*   76
-------
 24      (4x6=24)
12-      (2x6=12, shifted by one)
  28-    (4x7=28, shifted by one)
+ 14--   (2x7=14, shifted by two)
-------
  1824
```

As you can see, the paper method multiplies small numbers (numbers less than 10) and then adds all the products to get the final result. We can use this same technique for large binary numbers. Since there is an instruction that supports multiply, MUL, we can multiply two 8-bit numbers at a time and then add up all the products. Below is an example of how to multiple two 16-bit numbers.

```
         A1  A2
    *     B1  B2
------------------
          H22 L22
      H21 L21
      H12 L12
+ H11 L11
------------------
  P1  P2  P3  P4
```

As you can see, the result of multiplying two 16-bit (or 2 byte) numbers gives a 32-bit (or 4 byte) result. In general, when multiplying to binary numbers, the result will be twice the size of the operands. For reference, the H and the L signify the high and low byte of the result of each multiplication and numbers signify which values where multiplied, B and A. For example, L21 is the resulting low byte of multiplying B2 and A1 together. Also note that the four results are just the following:

P1 <= H11 + carry from P2
P2 <= H21 + H12 + L11 + carry from P3
P3 <= H22 + L21 + L12
P4 <= L22

The skeleton code provided for the lab will contain an algorithm that multiplies 2 16-bit numbers in the fashion described above. You will need to expand on this function for the lab assignment.

## ASSIGNMENT

Write a program in AVR Assembly that calculates the result of (A + B)^2. A and B are 16-bit numbers or 2 bytes, thus A + B will produce a 3-byte result since we want to save the carry bit. That result will then be multiplied by itself to produce a final 6-byte result. To do this, you will need to write **two functions**, a function that adds two 16-bit numbers and produces a 24-bit result and a function that multiplies two 24-bit numbers.
The two operands will be entered in data memory during simulation time. Operand A is at the data memory location $0100 and $0101. Operand B is at data memory location $0102 and $0103. The 6-byte result will be stored in the data memory locations $0104 - $0109.

This lab will not use the TekBot platform or the AVR board. It is purely a simulation-based lab. Write and assemble the program, test it, and demonstrate the program to your TA. The TA will provide two numbers for you to run the simulation on when you submit the assignment. Turn in the write up and the source code to your TA.

## STUDY QUESTIONS/ REPORT

Write a short summary that details what you did and why, explain any problems you may have encountered, and answer the questions below. This write up and your code should be submitted to your TA by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED.

### Study Questions

No additional questions for this lab assignment.

## CHALLENGE

Complete the assignment with the following shift-and-add multiplication technique instead of sum of products technique described above. If you complete the challenge section, it will count for both the lab assignment and challenge portion.

Although the sum-of-products technique is easier for humans to perform multiplications, it is not the most efficient way for multiplying binary numbers, especially very large numbers like 1024-bit numbers. Also, it is very difficult to implement a sum-of-products multiplier in hardware. So a variation to the technique was created and used the inherent properties of a base-2 number system. This technique is known as the shift-and-add multiplier.

Referring back to the basic concepts of mathematics, multiplication has two operands, the multiplicand (or the operand to be multiplied) and the multiplier. When a decimal number is multiplied, it essentially adds the multiplicand to itself for the amount specified by the multiplier. This seems pretty obvious, until you use binary. In binary, the multiplier is used to determine whether the multiplicand is added or not. The figure below will demonstrate both decimal and binary multiplications.

```
Decimal:        23    ← Multiplicand
          *     16    ← Multiplier
          ----------
               138    ← 6 * 32 = 138
          +    230    ← 1 * 23 = 23 and then shift 23 left one = 230
          ----------
               368    ← Final Multiplication Product
```

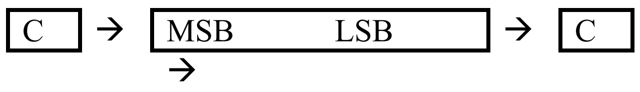
Now, we apply this same knowledge to a binary system that uses 4-bit values. The basic principle is that we shift through the multiply. When a 1 is received, we add the multiplicand to the low bit of the result, if a zero is received we do nothing. We then shift the entire result one bit to the left, thus essentially multiplying the result by 2 and repeat the process. Below is an example.

```
     Binary(4-bit):        1011  ← Multiplicand
                           *     1101  ← Multiplier
          Multiplier       ----------------
          (LSB)  1         0000  1011  ← 1 * 1011 =1011; 1011 shift left zero = 00001011
                 0         0000  0000  ← 0 * 1011 =0000; 0000 shift left one  = 00000000
                           ----------------
                           0000  1011  ← Add results together
                 1         0010  1100  ← 1 * 1011 =1011; 1011 shift left two  = 00101100
                           ----------------
                           0011  0111  ← Add results together
          (MSB)  1         0101  1000  ← 1 * 1011 =1011; 1011 shift left three = 01011000
                           ----------------
                           1000  1111  ← Add results together to get final product

          Proof:  Multiplicand   Multiplier      Product
          Bin     1011      *    1101      =     1000 1111
```

```
Dec    11    *    13    =    143        ← Correct
```

Although the method above is sound and easily understandable, there is a more efficient method. For the purpose of this example, assume you are running on a 4-bit system where the registers are 4-bits. It would take 4 registers to do the method above. A better way would be to share the low result registers with the multiplier registers so you can shift all the registers at once and only use 3 registers. In order to do this, you would shift everything to the right instead of to the left and also rotate through the carry. So if the carry bit is set after the shift, add the multiplicand to the high register of the result and shift again; otherwise, just shift. Note that when shifting to the right, you need to rotate through the carry; this means that whatever is in the carry bit will be shifted in the most significant bit and the least significant bit will be shifted out into the carry bit. For example:



Rotate Right Through Carry Bit

The following figure demonstrates this:

```
Multiply            1011   ← Multiplicand
               *    1101   ← Multiplier
      Carry    ---------------
               0000 1101   ← Load Multiplier into low register
      1        0000 0110   ← Shift right through carry
               1011        ← Carry is set, so add multiplicand
               ----------------
      0        1011 0110   ← Result of addition
      0        0101 1011   ← Shift right through carry
               -----       ← Don’t add since carry is 0
               ----------------
      0        0101 1011   ← Result thus far
      1        0010 1101   ← Shift right through carry
               1011        ← Add multiplicand since carry is set
               ----------------
      0        1101 1101   ← Result of carry
      1        0110 1110   ← Rotate right through carry
               1011        ← Add multiplicand since carry is set
               ----------------
```

```
1        0001   1110   ← Result of addition
0        1000   1111   ← Result after final shift, note that a '1' was shifted
                         in, because it was the carry that was set from
                         the last addition.
```

As you can see, the shift and add technique can be easily created with a simple for loop that loops a specific amount of times depending on the size of the data, for the case above we used 4-bit numbers so we looped 4 times. In the loop there is a simple rotate left and an add depending on the carry bit. This technique can be easily used for any sized number with minimal effort and is used internally for multiplications with most micro architectures.

# SECTION FIVE SIMPLE INTERRUPTS

## SECTION OVERVIEW

- Understand how and when interrupts can be used
- Demonstrate the use of simple interrupts
- Explore how to configure interrupts on a microcontroller

## PRELAB

Be sure to have completed these prelab questions before your lab. They will help you to perform the tasks in this lab and hopefully give you more time to experiment.

1. In computing there are traditionally two ways for a microprocessor to listen to other devices and communicate. These two methods are commonly called ‘polling’ and ‘interrupts.’ A large amount of information about these two methods exists. Please describe what each of them is and a few examples where you would choose one over the other.
2. What is the function for each bit in the following registers in the ATMega128? EICRA, EICRB, and EIMSK. You can find this information from either the AVR Instruction Set guide, or the ATMega128 Reference Manual. You can find both of these on the TekBots webpage for the ECE375 course. HINT: These registers are related to ‘external interrupts.’
3. The AVR microcontroller uses ‘interrupt vectors’ to run code when an interrupt is triggered. What is an interrupt vector? List the memory locations for the following vectors in the AVR microcontroller: Timer/Counter2 Comparison Match, External Interrupt 2, and USART1-Rx Complete.
4. In the AVR microcontroller like with many others, there are several different ways of triggering interrupts. Below is a sample signal being input onto one of the external interrupt pins. List where the interrupt would trigger on this waveform if the interrupt was set up as: a.) rising edge, b.) Falling Edge, c.) Level High, and d.) Level Low.

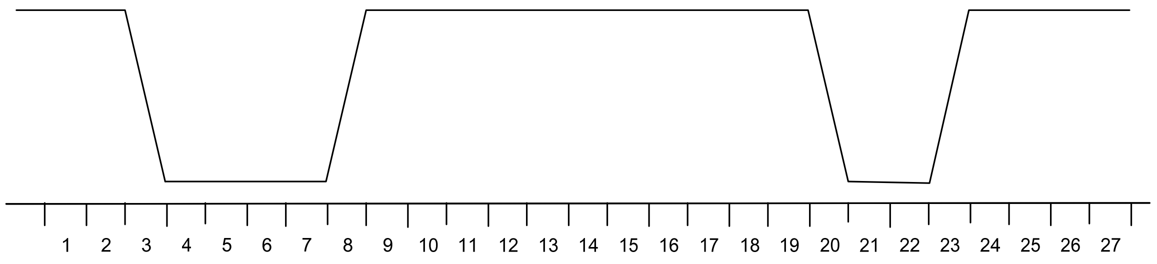


Figure 1: Sample Input to External Interrupt

## PROCEDURE

## Introduction

Most modern day computing systems use interrupts to communicate with peripheral devices. This lets the microprocessor(s) function and perform calculations until a peripheral device needs attention. Some examples of this are how PC hardware used to have 'interrupts' that a user would have to define so that maybe a sound card or joystick could ask the microprocessor for attention. Using interrupts can be tricky and sometimes wasteful. When an interrupt request comes in, the microprocessor has to stop what it is doing, store any special variables and then service the interrupt. Once the interrupt is done, the processor then must reload its original special variables and restart what it was doing. For example if a peripheral wants the microprocessor to store a single byte of data every couple of clock cycles it may try to interrupt the microprocessor every couple of clock cycles. This would force the microprocessor to spend all of its time storing its special variables, servicing the interrupt, reloading the special variables and then starting all over again when the next interrupt comes in. The cost of servicing an interrupt in this manner is called a 'context switch.' This is why many modern computers have coprocessors and peripheral controllers (like DMA for example) to handle these frequent requests.

## Interrupting a TekBot

You will need to write a short assembly program that causes your TekBot to move forward. Then when either its right or left whisker is hit it will need to react by interrupting the AVR microcontroller to back up and turn away from the point of impact. This will make your TekBot function as it has before in Lab 1 and Lab 2, but using a different method in code. This is an important point to note. There is ALWAYS more than one option/solution. Your job as an engineer is to be able to choose the BEST option based on all pertinent information.

Write your assembly program and download it to your TekBot. Show your TA the operation and the code and have them sign below for credit. Be sure your code is well commented. Skeleton code is available on the webpage.

TA Signature: ________________________________________

## STUDY QUESTIONS/ REPORT

Write a short summary that details what you did and why, explain any problems you may have encountered, and answer the questions below. Submit your write up and code to your TA via email by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED.

### Study Questions

1. As an engineer you should be able to justify your design and testing choices. You have implemented this bumper TekBot in three ways now. Explain the benefits and costs of all three implementations. Some important areas of interest include, but are not limited to, efficiency, speed, cost of context switching, time to code, understandability, etc.

2. Now that you have a basic understanding of interrupts would work, would it be possible to use a timer/counter interrupt to perform the wait loop while the robot is between transitions within the external interrupt? (HINT: The order in which the interrupt is located within the interrupt vector list is also the priority, the lower on the list, the higher the priority.) Give a reasonable argument either way.

## Challenge

1. Sometimes your TekBot can get caught in a loop where it is stuck in a corner and it continually backs up, hits the right whisker, then backs up and hits the left whisker, then the right, then left, and so on. Add a 'memory' to your TekBot so that it can detect this problem and when it has hit alternating whiskers five times, it will stop, turn around 180 degrees, and resume forward motion to get out of the corner.

In addition, correct the simple problem of the TekBot hitting the same wall several times. In this scenario the TekBot hits one of its whiskers, backs up and turns away, but does not turn far enough and hits the same object. In this case you need to make the TekBot backup and turn away twice as far as if the whisker has been hit twice in a row.

Write your program and keep it well documented. Turn in your code along with your write up and have you TA sign below that they have seen your functioning code.

TA Signature: ____________________

# SECTION SIX
## EXTREMELY SIMPLE COMPUTER (ESC)

## SECTION OVERVIEW

- Review and utilize all knowledge about computer organization
- Create the link between hardware and software
- Develop and simulate a functional simulator
- Enhance knowledge about functions and subroutines

***Note:***

- This is a group project. You can work in groups of two.

## PRELAB

No prelab is required.

## PROCEDURE

### Specifications

The details of your extremely simple computer (from here on out referred to as ESC) are as follows.

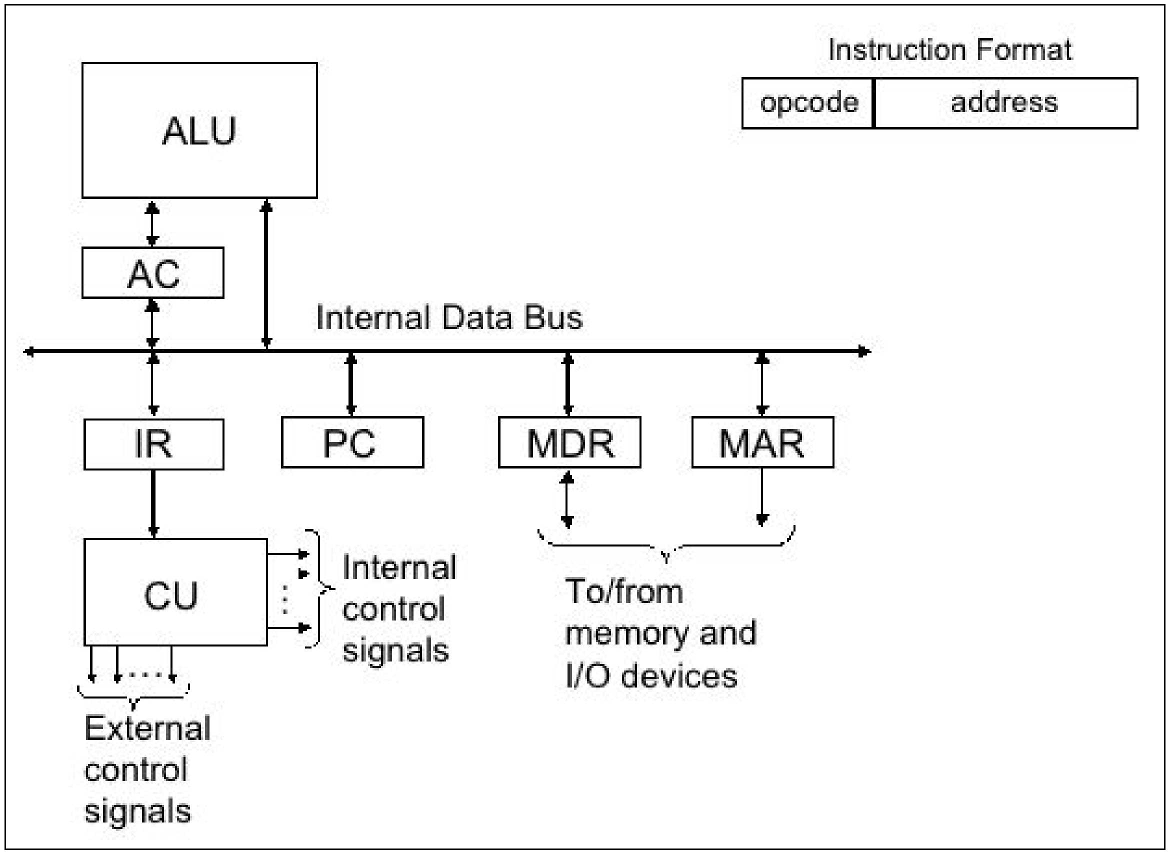


Figure 1: ESC Architecture Diagram

## *Data*

8 bits (signed)

## *Registers*

There are only 4 special registers in the ESC.

1. Accumulator (ACC)

The accumulator is not addressable. It is used in some ESC instructions, but its address does not appear in these ESC instruction bit vectors. Instead with the operations that use the accumulator, it is built in that the accumulator should be used. The ESC accumulator should be located in the ATmega128 data memory location $0100.

2. Program Counter (PC)

The use of the PC is transparent to the ESC user. The PC is initialized to $00 the beginning of the program memory space and incremented after each ESC instruction. The PC should be located in ATmega128 data memory location $0101. Although the ESC program memory only needs 5 bits to address, we need to set this to correspond to AVR data memory; this means you will have to write a function that converts the 5-bit ESC program memory address into the corresponding 16-bit AVR data memory address.

3. Instruction Register (IR)

The use of the IR is also transparent to the ESC user. The current ESC instruction (as a binary code) is stored in the IR. The ESC simulator will read the instruction that is in the IR register to know what operations to do. The ESC IR should be located at ATmega128 data memory location $0102. The IR should be initialized to $00.

4. Memory Address Register (MAR)

The MAR will be used to access data memory in ESC. This register is very similar to the PC. It contains the address in ESC data memory where data is being loaded and stored. This register should be located at ATmega128 data memory location $0103. When accessing the ESC data memory, the address of the ESC data memory must be put into this register. A function is then called that converts the 5-bit ESC data memory address into the corresponding 16-bit AVR data memory address. The MAR should be initialized to $00.

5. Memory Data Register (MDR)

The MDR is used to hold the input and output data from the data memory in the ESC. All data must go through this register before coming from or going to ESC data memory. This register should be located at the ATmega128 data memory location $0104. An example of usage for getting memory from data would be as follows. The data memory address is stored in the MAR. Then the data from the address in the MAR is stored within the MDR. Finally, the data from the MDR can be moved to the IR or ACC, depending on the instruction. This works vice versa for moving data to data memory. Data from the ACC is moved to the MDR, the data address is put into the MAR, and finally, the data within the MDR can be stored into ESC Data Memory at the memory address located in the MAR.

## *Memory*

Direct, immediate, and indirect addressing modes are allowed in ESC.

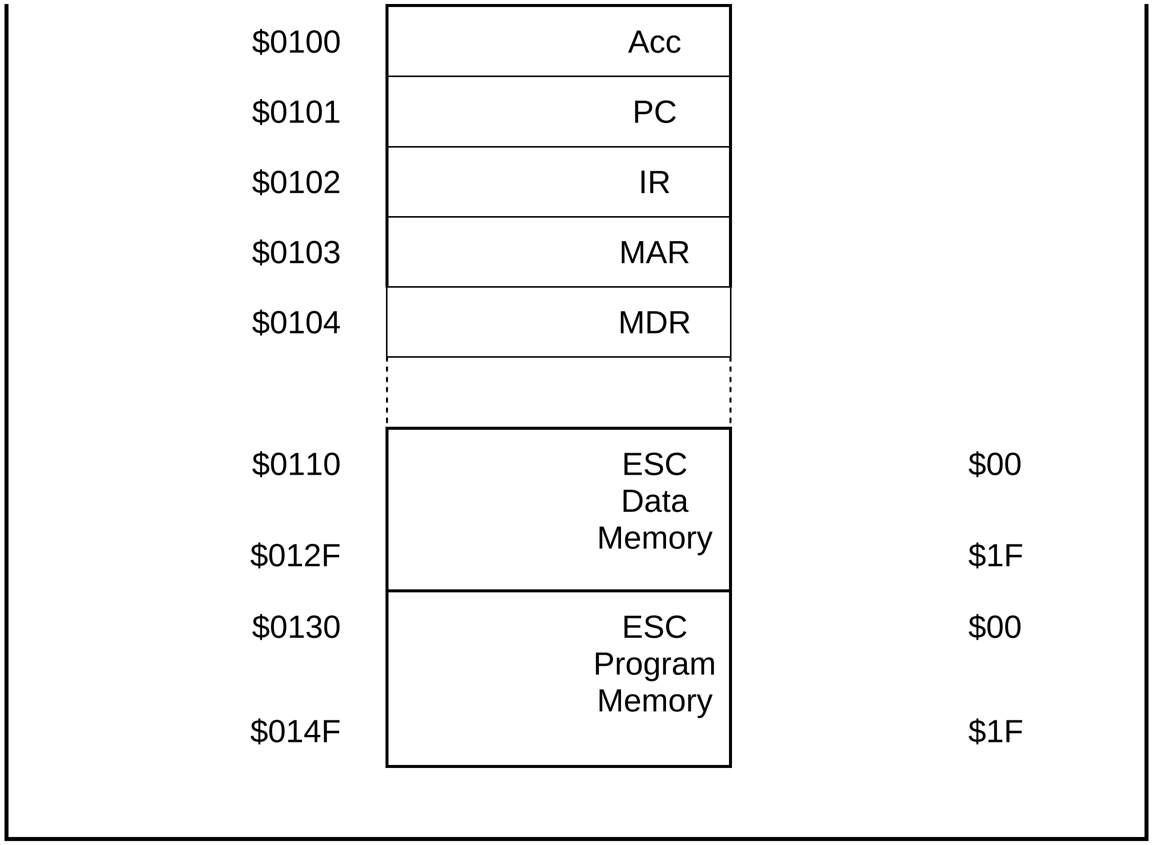


Figure 2: ESC->AVR Memory Map

## *Data Memory*

Since data address space is 5 bits, there are 2^5 = 32 bytes RAM in ESC (data address space is $00 – $1F). 5 bits in the ESC instruction specify a direct memory address. You will store the ESC data memory in the ATmega128 data memory locations $0110-$012F. Be aware of the conversion between 5-bit ESC memory addresses to 16-bit AVR memory addresses. For example, ESC memory location $07 corresponds to the ATmega128 data memory location $0117.

## *Program Memory*

ESC instructions are stored in ESC program memory. You should use the ATmega128 data memory locations $0130 – $014F as the ESC program memory.

| Op-code 76543210 | Mnemonic | Range of | Instruction Description |
|---|---|---|---|

| | | Data | |
|---|---|---|---|
| 000xxddd | | | This op-code identifies a group of instructions that does not need an address (or data) in LSB bits. |
| 00000ddd | STMI | $-4 \leq d \leq 3$ | Store immediate data (3 LSB sign extended) into address specified by the accumulator. |
| 00001000 | CLRA | | Clear the accumulator |
| 00010000 | LDMI | | Load indirect, read into the accumulator the value at the address specified by the contents currently in accumulator. |
| 00011000 | HALT | | ESC simulator stops execution. |
| 001aaaaa | ADDM | $0 \leq a \leq 31$ | Add direct memory location content to accumulator and place result in the accumulator. |
| 010aaaaa | SUBM | $0 \leq a \leq 31$ | Subtract direct memory location content from accumulator. |
| 011aaaaa | LDM | $0 \leq a \leq 31$ | Move the contents of a direct memory location to the accumulator. |
| 100aaaaa | STM | $0 \leq a \leq 31$ | Store the contents of accumulator into a memory location. |
| 101ddddd | JUMP | $-16 \leq d \leq 15$ | Jump unconditional. |
| 110ddddd | JMPZ | $-16 \leq d \leq 15$ | Jump if accumulator = $00. |
| 111ddddd | MOVI | $-16 \leq d \leq 15$ | Move immediate to accumulator. |

Figure 3: Instructions for the ESC Simulator

## *Instructions*

All instruction codes in ESC are 8-bits. There are three separate formats for the groups of instructions.

1. The first group consists of ADDM, SUBM, MOVI, LDM, STM, JUMP, and JMPZ instructions. For these, the most significant 3-bits are used for the op-code, and the other 5 are used for the direct address to specify an ESC Data Memory address or signed data. The format for these instructions will be:

```
Bit 7      Bit 6   Bit 5   Bit 4   Bit 3   Bit 2   Bit 1   Bit 0
|=== OPCODE ===|                   Address or Data
```

2. The second group consists of the HALT, CLRA, LDMI, and STMI. For the HALT, CLRA, and LDMI instruction, the most significant 5 bits are used to the op-code. They do not use the other 3 bits except that STMI uses the least significant 3 bits (sign extended to occupy 8 bits) as the immediate data to be stored.

```
Bit 7   Bit 6   Bit 5   Bit 4   Bit 3   Bit 2   Bit 1   Bit 0
|======== OPCODE =========|   X       X       X   |
```

Note the exception when the instruction is STMI, bits <2, 1, 0> will contain signed immediate data (limited to the range 000 to 111 (-4 – 3) when using indirect addressing mode).

NOTE: For all signed data, you will need to sign extend the signed data values to fill the 8 bits (i.e., JMP -13 = 10110011, Data value → 00010011, Sign Extended → 11110011).

## Assignment

You will be working in groups of two. You are to write, assemble, and simulate the ESC using AVR assembly language according to the specifications described above. In order to demonstrate to your TA that your ESC is properly working, you will need to write a test program (described below) and run it on your simulator. In addition, your TA will provide you with some other ESC code to test other instructions not used in your test program. NOTE: Feel free to write your own ESC program that will test all the various functionality of ESC.

## Test Program Description

You are to write a program using ESC mnemonics and convert it to binary form (fill the ESC program memory area with this program). This program should scroll the contents of 5 locations in ESC data memory in the following way: “TOM__” → “_TOM_” → “__TOM” → “M__TO” → “OM__T” → “TOM__” (Note that the “_” character indicates a blank space). Assemble the program by hand and load it into ESC program memory. Initialize by hand in ESC simulator for the first 5 ESC data memory locations with the first 3 letters of your name following by two space characters. Execute your simulator and show your TA that the program is working in ESC data memory using fixed memory locations.

## STUDY QUESTIONS/ REPORT

Write up a short summary that details what you did and why, explain any problems you may have encountered, and answer the below. Your write up and code must be submitted to your TA by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED. Since this is a group project, only one lab report needs to be submitted per group. Make sure both the group members' names are on the report.

## CHALLENGE

Since this extra credit is worth quite a bit in the overall lab scheme, you are going to have to work for it. Create an ESC compiler. This means you need to create a simple program in any language you choose that reads in an '.asm' file containing your ESC Assembly program in it. The ESC compiler will then parse through the '.asm' file, converting all the assembly mnemonic instructions (i.e. ADDM $04) into the appropriate binary op-code (i.e., 00100100 or $24). This new op-code is then written out to an '.obj' file. With this output file, you can very easily program your ESC simulator by cutting and pasting the op-codes into a .DB statement and using the LPM command to initialize the ESC program memory with it. ☺ To get credit, you must demonstrate this program to your TA and turn in a copy of the source code with your report.
NOTE: You can write this in any language you want. C/C++ is a good choice since you don't get enough of it in your undergrad curriculum and all TAs are proficient at C/C++ so they should be able to help you. If VB is more to your taste, go ahead. But if you are really courageous, write your program in PERL. PERL is a good language for text manipulation and parsing, although it is not really user friendly.

# SECTION SEVEN

## REMOTELY OPERATED VEHICLE V2.0

## SECTION OVERVIEW

- Use your knowledge of computer architecture to create a real system as a proof of concept for a possible consumer product
- Improve your knowledge of interrupts and ability to use them
- Learn how to use the Universal Synchronous Asynchronous Receive Transmit (USART) on the mega128

## PRELAB

1. In this project you are going to be given a set of functions that you need to have your 'toy' perform. Find another toy that uses a microcontroller and describe the functions that it performs. For each function, explain the interesting code (mega128 specific – setup registers ext.), mechanics, and/or electronics needed.

## PROCEDURE

### Problem

You work for TekToy Co., a global subsidiary of TekBots International. You are working on designing a new toy for the 'remotely controlled' line of products. You want to build a simple robot that can be controlled using IR. You want to use IR because it is cheaper and requires less design than an equivalent RF system.

Your first step is to build a proof of concept model so that you can sell the idea to management. You dig through your desk and come across your TekBots base from when you were in school and a couple of AVR microcontroller boards for the TekBot. Since you know how to use these parts you decide to build your proof of concept model with them.

### Specifications

Below is a list of things to consider while creating your design.

1. You would like to have multiple robots running at the same time so you need to somehow make a robot respond only to its own remote control. You decide to do this by assigning each robot a distinct address. The remote should transmit this address each time a command is transmitted. You have decided to send two eight bit codes, one address and one data. Address code must start with a '0' and data codes must start with a '1.' Figure 1 shows an example.

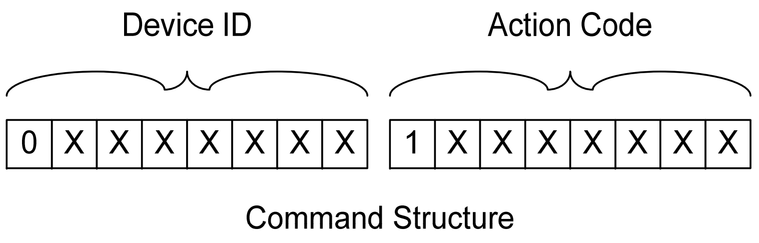


Figure 1: Example command structure

2. The AVR boards you are using have a built in IR transmitter and receiver connected to one of the serial UARTs. You feel that the UART is a good option for communicating to the robot. You will need to setup the UART on both the robot and the remote.

The AVR can set up the UART to communicate at high speeds, but you know that the IR transmitter and receiver cannot work this fast. You should use a baud rate of 2400.

3. For your simple proof of concept model, you only need a few actions: forward, stop, reverse, left, and right. To make your life easier you decide to spec out what the different commands are before you start coding. These commands are shown below in Table 1.

| Command | Action Code |
|---|---|
| Forward | 0b10110000 |
| Backward | 0b10000000 |
| Turn Right | 0b10100000 |
| Turn Left | 0b10010000 |
| Halt | 0b11001000 |
| Future Use | 0b11111000 |

Table 1: Action codes for proof of concept model

4. You can take advantage of a built in 'loop back' for the IR system on the AVR boards. When something is transmitted using the IR, the receiver can see this signal and will receive the same data. This means you can compare what you sent with what you receive to see if there has been an error in transmission. This error could be because there are multiple remotes running at one or a possible programming error.
5. You would like to have some minimal intelligence on the robot. When either of the front whiskers is triggered, the robot should reverse and turn away from the point of contact, ignoring any input from the remote control. After the robot has avoided the object it should resume what it was doing before the impact.

There is a piece of sample skeleton code available on the web. If you would like to use it go ahead. If not, feel free to do your own design.

## STUDY QUESTIONS/ REPORT

Write up a short summary that details what you did and why, explains any problems you may have encountered, and answer the questions below. Be sure to explain the operation of your code in the document. Do not rely only on the comments in your code to describe the function of the code. This write up is due by 4pm on Friday of dead week. One of the TAs will be available in the lab to take your submissions. NO LATE WORK IS ACCEPTED.

## CHALLENGE: FREEZE TAG

Since you are going to be showing your proof of concept to upper management, you feel you need to have something that can generate some 'real excitement.' The idea of having two robots play freeze tag with each other sounds like a good idea.

You think the best way to do this is to have each robot emit a certain code when the user with the remote presses a certain button. This code would then be detected by any robot that could see it (except the one 'firing') and cause the robot to 'freeze' for five seconds. When the robot unfreezes it should immediately resume what it was doing. After three freezes, the robot should stop working until it is reset.

The code your robot should transmit to freeze other robots is shown in Figure 2. The robot—not the remote— transmits this code. Make one of the buttons on the remote transmit a code to the robot that has the robot transmit the freeze code. The freeze command code from your remote should be 0b11111000.

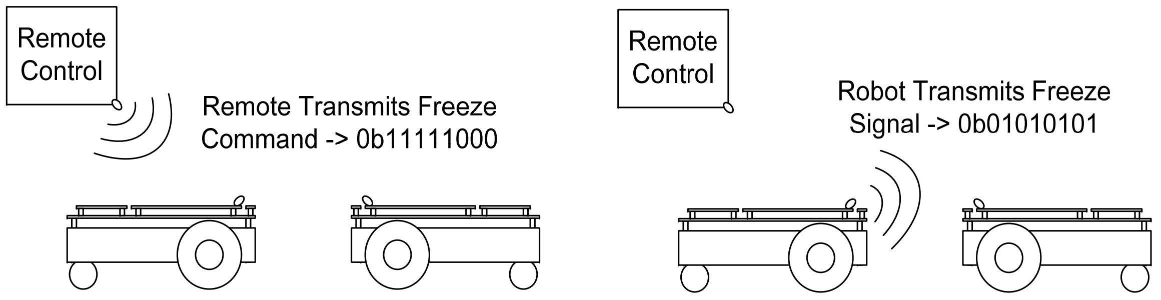

Remote Control
Remote Transmits Freeze Command -> 0b11111000
Remote Control
Robot Transmits Freeze Signal -> 0b01010101

Figure 2: Freeze Tag Operation

To show your design you will need to work with another team to have the two robots compete. Show your TA and have each team turn in a copy of their code with a written description.

TA Signature: ______________________

# APPENDIX A PARTS LISTS

## PARTS LIST

| t y | cription | ig ur e | Supplier | plier # |
|---|---|---|---|---|
| | ;a128.2 | | TekBots | ;a128.2 |
| | isp | | TekBots | isp |
| | ;gramming table | | TekBots | isp cable |
| | ıale Jumper Blocks | | Digikey | ›228-ND |

Fig. A

Fig. B

Fig. C

# APPEN

## SCHE

## EGA128.2

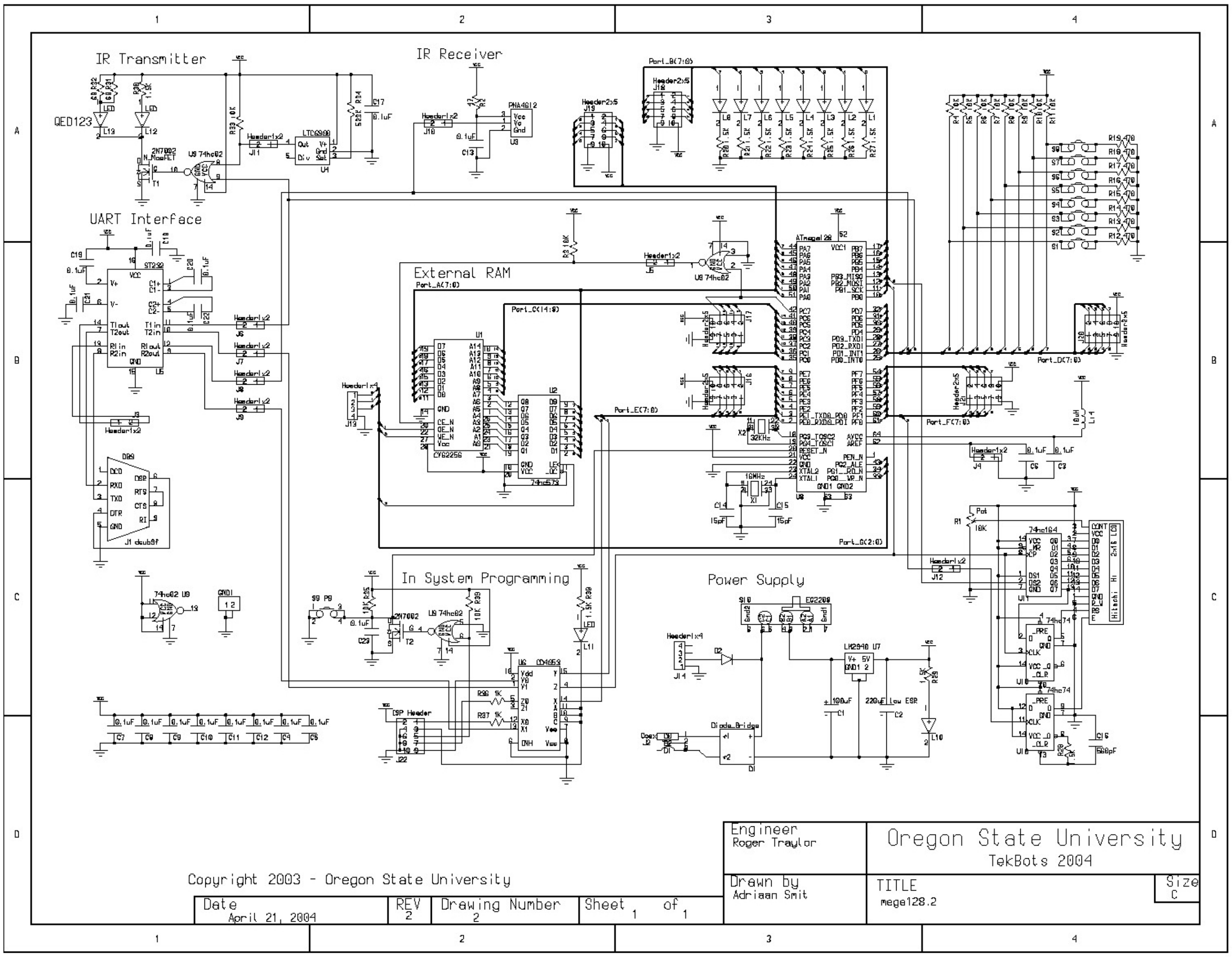


Figure 51: Block diagram of the charger board

## VR-ISP

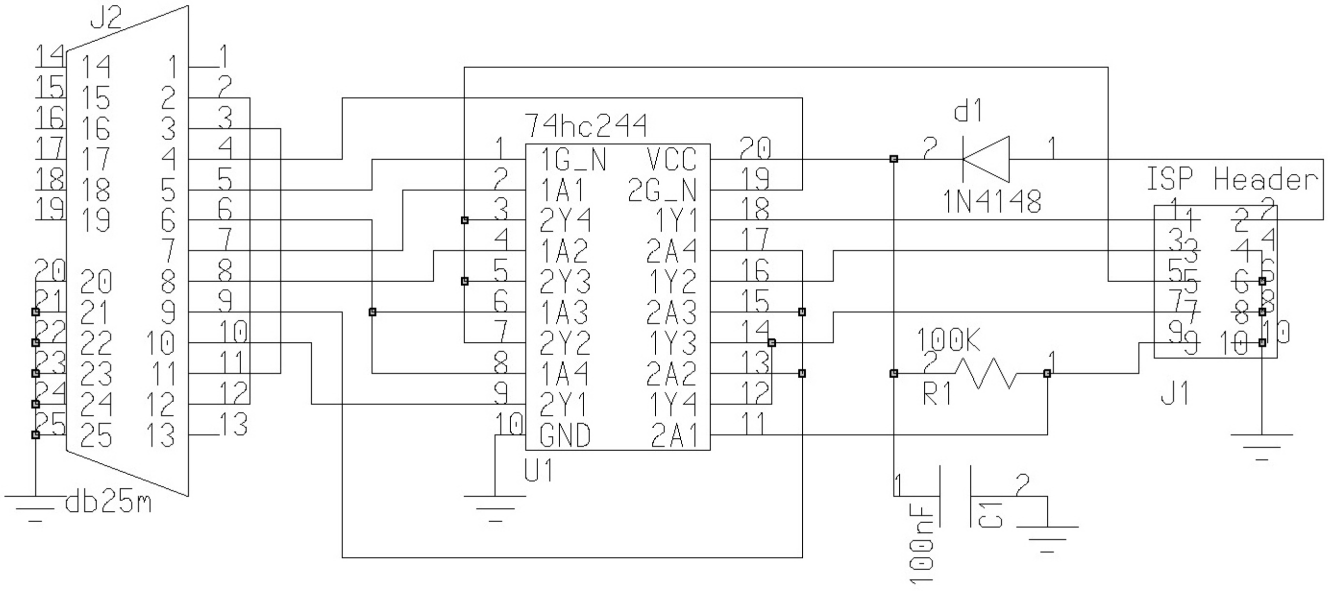


**Figure 54: Block diagram of the analog board**

# APPENDIX C
## SUPPLIERS

| | |
|---|---|
| DigiKey | 701 Brooks Ave. South<br>Thief River Falls, MN 56701-0677<br>(800) 344-4539<br>**http://www.digikey.com** |
| Mouser Electronics | 1000 N. Main Street,<br>Mansfield, TX 76063<br>(800) 346-6873<br>**http://www.mouser.com** |
| TekBots | 220 Owen Hall<br>Oregon State University<br>Corvallis, OR 97331<br>tekbots_support@eecs.oregonstate.edu<br>**http://eecs.oregonstate.edu/tekbots** |
| Solarbotics | 179 Harvest Glen Way N.E.<br>Calgary, Alberta, Canada T3K 4J4<br>(866) B-ROBOTS<br>**http://www.solarbotics.com** |
| McMaster-Carr | P.O. Box 7690,<br>Chicago, IL 60680-7684<br>(562) 692-5911<br>**http://www.mcmaster.com** |
| Jameco Electronics | 1355 Shoreway Rd,<br>Belmont, CA 94002<br>(800) 831-4242<br>**http://www.jameco.com** |