Atmel Studio and ATmega128
A Beginner's Guide

Version 1.0:

Written by:
David Zier

Version 2.0:

Written by:
Taylor Johnson, Mohammed Sinky, and Arul Dhamodaran

Oregon State University
TekBots™

September 29, 2014

Contents

1 Introduction

The Department of Electrical Engineering and Computer Science (EECS) at Oregon State University (OSU) has recently been reevaluating the way classes are taught. With the collaboration of Tektronix, the TekBots program was born. TekBots has allowed ECE a way to educate by keeping a consistent flow in the course work. In keeping with that consistency, ECE 375: Computer Structure and Assembly Language Programming, uses Atmel's software tools and AVR RISC core chips. For more information about the TekBots Program, go to `http://eecs.oregonstate.edu/tekbots/`.

1.1 Purpose

The purpose of this document is to provide the reader with the basic knowledge to develop assembly programs for the ATmega128 using Atmel Studio 6. The intent of this document is to be used in conjuncture with lecture material from ECE 375: Computer Structure and Assembly Language Programming.

1.2 Atmel Studio 6 Overview

Atmel Studio 6 is the new professional Integrated Development Environment (IDE) for writing and debugging AVR applications in Windows environments. Atmel Studio 6 was created by the Atmel Corporation and can be downloaded for free from `http://www.atmel.com/tools/atmelstudio.aspx`.

1.3 ATmega128Overview

The ATmega128 is a low power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega128 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. The complete datasheet can be downloaded from `http://www.atmel.com/Images/doc2467.pdf`.

1.4 Nomenclature

Throughout this document, there will be several styles of writing to signify different aspects such as code examples or command line instructions.

- This style is used for normal text
- `This style is used for code examples`
- **This style used for menu selects and commands, ie. File ⇒ Exit**

1.5 Disclaimer

TekBots and Oregon State University are trademarks of OSU. Atmel Studio 6 and ATmega128 are trademarks and/or copyrighted by Atmel Co. Windows is a trademark of Microsoft Co.

2 Atmel Studio 6

Atmel Studio 6 is Atmel's official Integrated Development Environment (IDE), used for writing and debugging AVR applications on the Windows platform. Atmel Studio 6 is available for free, and can be downloaded at: http://www.atmel.com/tools/atmelstudio.aspx

This section provides general information on how to successfully use Atmel Studio 6 to create, compile, and debug AVR assembly projects. Not every aspect of Atmel Studio 6 will be covered here, but for those who choose to learn the program in more detail, additional information can be obtained from Atmel's website at: http://www.atmel.com

2.1 Startup Tutorial

This tutorial will give a step-by-step guide on how to install Atmel Studio 6, create a project, add code (new or existing) to the project, and simulate the project.

2.1.1 Installation

The installation of Atmel Studio 6 is straightforward and involves only a few steps:

1. Go to http://www.atmel.com/tools/atmelstudio.aspx, and click the download icon next to **Atmel Studio Installer**.
2. At this point, you can create a "myAtmel" account, or choose to download Atmel Studio 6 as a guest. In either case, follow the directions and download the executable installer.
3. Locate the .exe file you just downloaded and run the setup program by double-clicking on it.
4. Follow the instructions in the setup program. Most of the default installation directories will work just fine.
5. When the installer is finished, click on the **Finish** button to complete the setup process. Atmel Studio 6 is now successfully installed.

2.1.2 Project Creation

Atmel Studio 6 is an Integrated Development Environment (IDE). Just like any other IDE, Atmel Studio 6 is project-based. A project is like an environment for a particular program that is being written. It keeps track of what files are open, compilation instructions, as well as the current Graphical User Interface (GUI) selections. The following process detail the steps needed to create a new project:

1. Start Atmel Studio 6 by navigating through the Windows start menu: **Start** ⇒ **Programs** ⇒ **Atmel** ⇒ **Atmel Studio 6**. The path could be different if changed during installation.
2. Atmel Studio 6 should launch and display a Start Page. To create a new AVR project, click on the **New Project...** button, or navigate to **File** ⇒ **New** ⇒ **Project...**
3. The dialogue box that appears should look similar to Figure 1. Under **Installed Templates**, make sure **Assembler** is selected.
4. Select **AVR Assembler Project** as the project type.
5. In the **Name** text box, type the name of the project, such as **Lab1**.
6. Make sure that that the checkbox for **Create directory for solution** is checked.
7. The location of the project can be changed by clicking on the **Browse...** button next to the path name, and navigating to the desired location for the new project.
8. Click **OK** to continue.
9. The next dialogue requires a device selection. First, ensure that the drop-down menu labeled **Device Family:** selects either **All** or **megaAVR, 8-bit**.
10. Scroll through the list of devices and select **ATmega128**.
11. Click **OK** to complete the project creation.

At this point, an editor window appears within Atmel Studio 6 and you are able to begin composing your assembly program. Notice that Atmel Studio 6 has already created an empty assembly file for you, based on the name given earlier as the project name. For example, if you named your project **Lab1** as in Figure 1 then the automatically-created assembly file would be named **Lab1.asm**.

If you want to incorporate some code that you have already written into this new project, then you can do so in one of two ways. First, you can simply

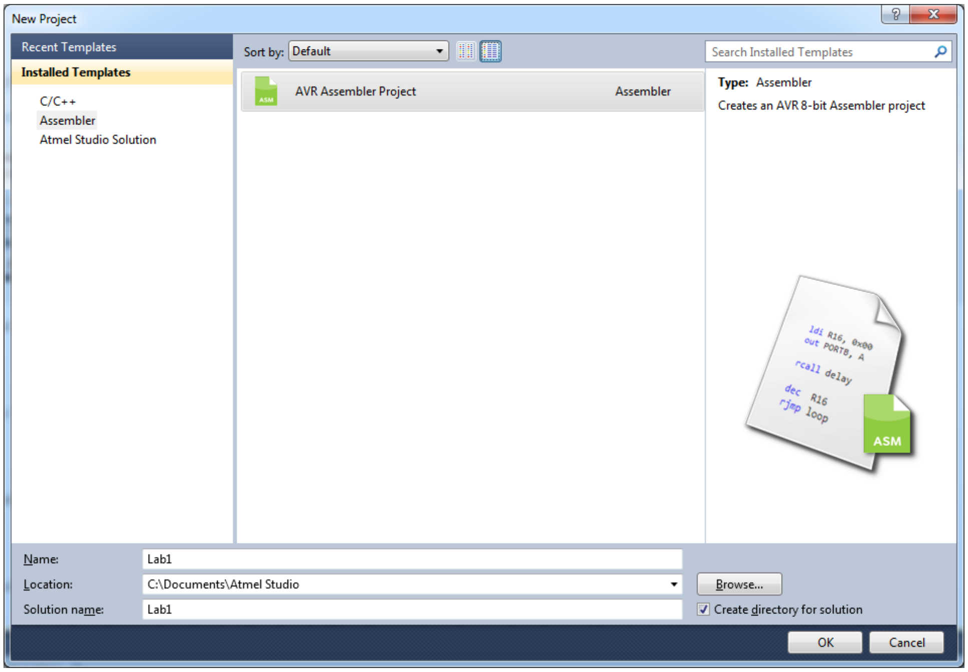

Figure 1: AVR Studio Project Creation.

open your existing code file with a text editor and copy-paste some or all of its contents directly into the open editor window within Atmel Studio 6 - this copies your code into the file created for you, e.g. **Lab1.asm**. If you want to include an entire existing file into your newly-created project, use the following steps:

1. In the **Solution Explorer** on the righthand side of the Atmel Studio 6 window, right-click on the name of your project (e.g., **Lab1**) and select **Add ⇒ Existing Item...**

2. Navigate to the existing assembly code file that you would like to use for this project, select it, and click **Add**.

3. Your existing code file will now appear in the **Solution Explorer** under the heading of your project. Double-click on the file name and it will open in a new editor tab.

4. If this existing file is to be the "main" assembly file of your project, right-click on the file name and select **Set As EntryFile**. Now this existing file that you included in the project will be considered the main entry point during compilation. Feel free to remove the automatically-created

file (e.g. **Lab1.asm**) if you are not going to use it, by right-clicking on the file name and selecting **Remove**.

2.1.3 Project Simulation

Once a project has been created, and you have written an assembly program, it will need to be tested. This is accomplished by running the program on a simulated microcontroller built into Atmel Studio 6. Atmel Studio 6 has the capability to simulate almost every AVR microcontroller offered by Atmel. For the purposes of this tutorial, the ATmega128 will be the microcontroller that will be simulated. This microcontroller was selected earlier during the project creation phase. (To change the microcontroller, right-click on your project name in the **Solution Explorer** and select **Properties**. This will open a tab that allows you to configure various properties of your project. Make sure the **Build** tab is selected, and then click the **Change Device...** button and select a different microcontroller.)

1. Before the program can be simulated, the program must first be compiled. There are three ways to do this:
   (a) In the main Atmel Studio 6 menu, navigate to **Build** ⇒ **Build Solution.**
   (b) Click on the **Build Solution icon** on the main toolbar.
   (c) Press the **F7** key.
2. If the code was successfully compiled, a message in the **Output** window at the bottom should read "Build succeeded". If it does not say this, then there were some errors in the code. Clicking on the errors in the**Error List** will highlight the line of code causing the error in the editor window.
3. Once the code has been successfully compiled, simulation can begin. There are two ways to simulate the chip: debugging mode, which allows a line-by-line simulation, and run mode, which continuously runs the program.
   (a) There are a few ways to run in debug mode:
      i. Follow the menu **Debug** ⇒ **Start Debugging and Break**.
      ii. Click on the **Start Debugging And Break** icon.
      iii. Press **Alt+F5**.
   (b) To start the run mode:
      i. Follow the menu **Debug** ⇒ **Continue**.
      ii. Click on the **Start Debugging** icon.
      iii. Press **F5**.
4. To stop the simulation at any point:
   (a) Follow the menu **Debug** ⇒ **Stop Debugging**.

(b) Click on the **Stop Debugging Icon**.

(c) Press **Ctrl+Shift+F5**.

5. That is how to simulate a program. For more detailed simulation tips and strategies, see Simulation Tips below.

2.2 Simulation Tips

Just simulating a program is not enough. Knowing how to use the simulator and debugger is essential to get results from simulation. This section will provide the necessary information needed to get the most out of a simulation.

2.2.1 Line-By-Line Debugging

Line-by-line debugging is the best way to take control of the simulation. It allows the programmer to verify data in registers and memory. There are several ways to get into line-by-line debugging mode. The first would be to start the simulation in line-by-line debug mode by clicking on the **Start Debugging and Break icon**. When the program is in run mode, hitting the **Break All icon** will halt the simulation and put it into line-by-line mode. Also, if a break point was set in the code, the simulation will automatically pause at the break point and put the simulation into line-by-line mode.

When running in line-by-line mode, several new buttons will be activated. These allow you to navigate through the program.

- Step Into (**F11**) - Steps into the code. Normal operation will run program line-by-line, but will step into subroutine calls such as the `RCALL` command.
- Step Over (**F10)** - Steps over subroutine calls. Normal operation will run program line-by-line, but will treat subroutine calls as a single instruction and not jump to the subroutine instructions.
- Step Out (**Shift+F11**) - Steps out of subroutine calls. This will temporarily put the simulation into run mode for the remainder of the subroutine and will pause at the next instruction after the subroutine call.
- Run to Cursor (**Ctrl+F10**) - Runs simulation until cursor is reached. The cursor is the blinking line indicating where to type. Place the cursor by putting the mouse over the instruction you want to stop at and hit the Run to Cursor icon.
- Reset (**Shift+F5**) - Simulates a reset of the microcontroller; returns the simulator to the first instruction of the program.

After experimenting around with these five commands, you should be able to navigate through the code with ease.

2.2.2 Workspace Window

When debugging, the **Solution Explorer** window is supplemented by tabs such as **IO View** and **Processor**, which provide a look at the current state of the microcontroller during the course of simulation. The **IO View** tab contains all the configuration registers associated with the simulated chip. By default, this window should automatically be displayed when simulation is run in line-by-line mode. Figure 2 shows an example of what the **IO View** tab looks like during simulation. By expanding some of the contents of this window, additional information is available such as the current bit values, and address, of configuration registers. It is in this window where you can simulate input on the ports.

The **Processor** tab displays the current contents of the Program Counter, Stack Pointer, the 16-bit pointer registers X, Y, and Z, and the Status Register. Figure 3 shows an example of what the **Processor** tab looks like during simulation. The **Processor** tab also shows the current values contained in each of the general purpose registers (in the case of the ATmega128, registers `R00 - R31`).s

2.2.3 Memory Windows

In actuality, all of the registers are actually parts of memory within the ATmega128. In addition to the register memory, the ATmega128 has several other memory banks, including the program memory, data memory, and EEPROM memory. Of course, no good simulator is complete without being able to view and/or modify this memory, and Atmel Studio 6 is no exception.

To view the **Memory** window, follow the menu command **Debug** ⇒ **Windows** ⇒ **Memory** ⇒ **Memory 1** or hit **Alt+6**. The **Memory** window, shown in Figure 4, may pop up on top and obscure other windows, but it can be docked below the **Processor** and **IO View** tabs in order to be less intrusive.

The main area of the **Memory** window contains three sets of information; the starting address of each line of memory shown, the data of the memory in hexadecimal format, and the ASCII equivalent of that data. The pull down menu on the top left allows you to select the various memory banks available for the ATmega128. In Figure 4, the contents of Program Memory are being displayed, with `0x000000` as the starting address of the first line shown. To edit the memory, just place the cursor in the hexadecimal data area and type in the new data.

2.3 Debugging Strategies

Debugging code can be the most time consuming process in programming. Here are some tips and strategies that can help with this process:

- Comment, Comment, Comment. Unless it is absolutely blatantly obvious of what the code is doing, comment EVERY line of code. Even if the code

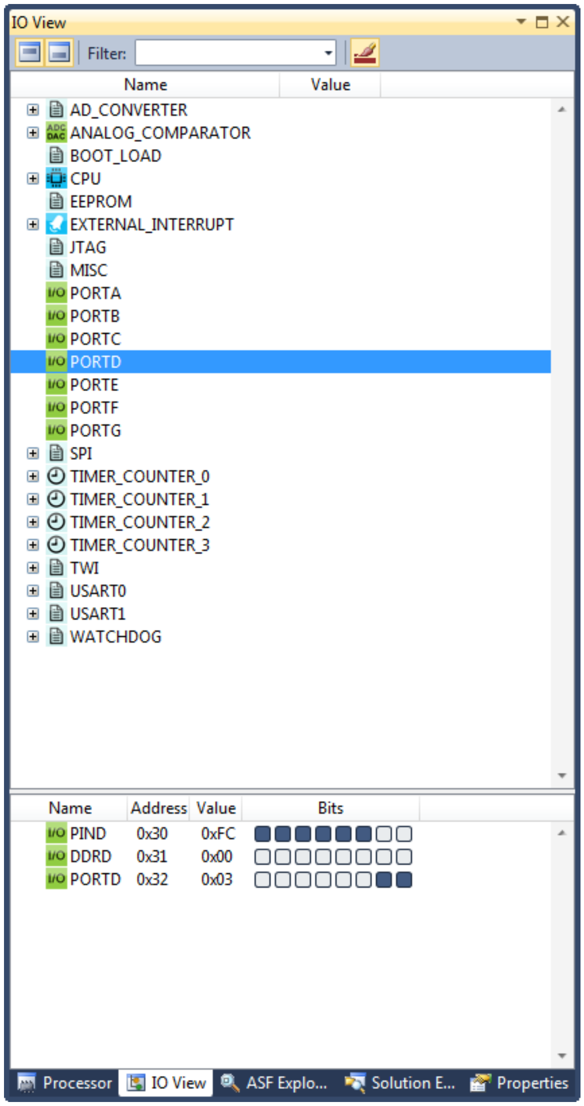

Figure 2: I/O View tab in Workspace.

is obvious, at least comment what the group of instruction is doing, for example, Initializing Stack Pointer.

- Pick a programming style and stick with it. The style is how you lay out your code, and having a consistent programming style will make reading the code a lot easier.
- Before writing any actual code, write it out in pseudo-code and convince

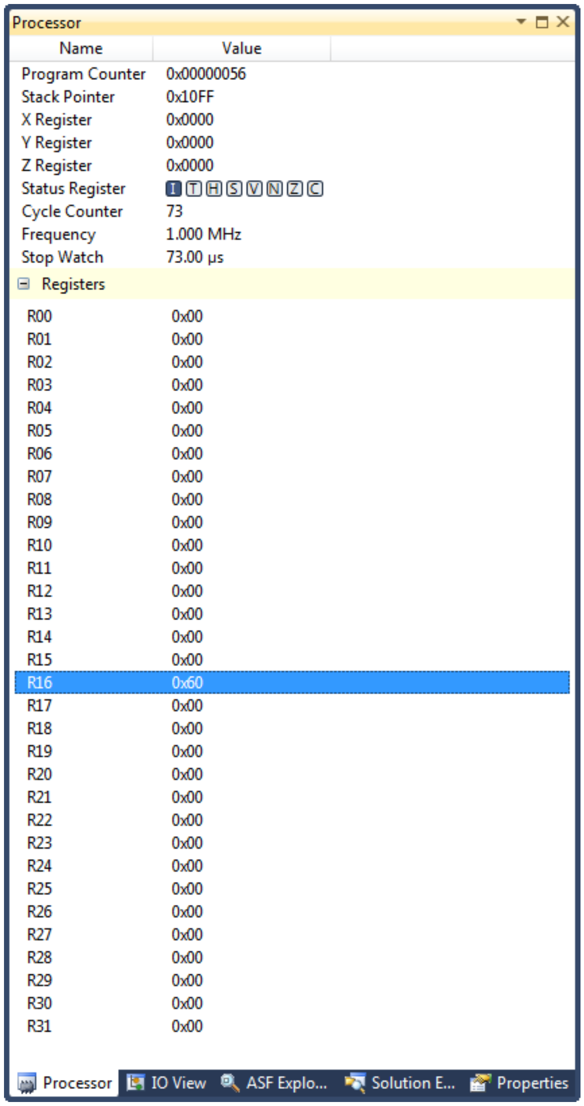

Figure 3: Processor tab in Workspace.

yourself that it works.

- Break the code down into small subroutines and function calls. Small sections of code are much easier to debug than one huge section of code.
- Wait loops should be commented out during debugging. The simulator is much slower than the actual chip and extensive wait loops take up a lot of time.

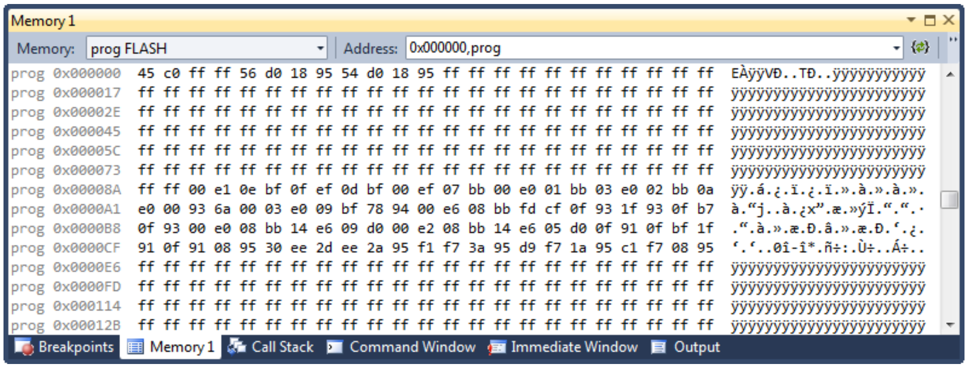

Figure 4: Memory Window.

- Use breakpoints to halt the simulation at the area known to be buggy. Proper use of breakpoints can save a lot of time and frustration.
- Carefully monitor the **I/O View** tab, **Memory** tab, and **Processor** tab throughout the simulation. These windows will indicate any problem.
- Make sure the AVR instruction is actually supported by the ATmega128.
- The ATmega128 has certain memory ranges; so make sure that when manipulating data, the addresses are within range.

3 Programming the ATmega128

Following simulation and debugging of your assembly program in AVR Studio, you should be ready to load your functional program on the actual microcontroller. This is the ultimate test in verifying your program works as it is supposed to. In many cases, your program may seem to be sound during simulation, but behaves differently when run on the actual hardware. Loading your program onto the microcontroller chip is typically referred to as “programming” or “flashing” your microcontroller. This involves transferring the “.hex” file generated by the compilation and assembly process, to the Program Memory located on chip. Recall that there exists 128 Kbytes of In-System Programmable (ISP) Flash memory dedicated to the Program Memory. The Flash is organized as 64K $\times$ 16, to accommodate one 16-bit word per entry, and is non-volatile, i.e. once you flash the chip, your code remains in memory even when powered off. This chapter will provide the steps you need to successfully program your ATmega128 chip with the dedicated Universal Programmer.

3.1 Parts and Cables Needed

The main device required for programming your ATmega128 chip, is the `osuisp2` device provided by TekBots$^{\text{TM}}$. Fig. 5 shows the device and accompanying cables. It is also referred to as the TekBots Universal Programmer. The Universal Programmer uses the Serial Peripheral Interface (SPI) standard and In-System Programming (ISP) to flash Program Memory. Thus, it is compatible with any Atmel AVR device that supports the SPI standard and has ISP capabilities.

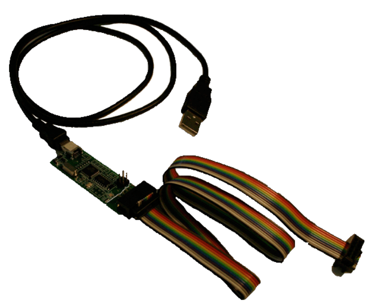

Figure 5: TekBots Universal Programmer kit: `osuisp2` programmer device connected to a USB cable and ribbon cable.

The Universal Programmer kit is not included with your ATmega128 board.

You will need to purchase it separately from TekBots^{TM}. You can find the TekBots store online by navigating to the "Store" link at `http://eecs.oregonstate.edu/education/`. Next select "TekBots Store - Oregon State University" under the drop-down menu. The Universal Programmer Kit will be under the "Spare Parts" link. Be sure to select "Walk-in" for the shipping method when you check out.

3.2 Downloading the Necessary Software

Prior to connecting the `osuisp2` programmer device to your computer and ATmega128 board, you will need to download the Universal Programmer software located at the TekBots website provided earlier. For convenience, a direct link is provided here: `http://eecs.oregonstate.edu/education/software/UniversalProgrammer3.1.zip`. Keep in mind that this link may change and to contact your TA for the most updated information. The link provided is is valid as of September 30, 2014. The software provides a graphical user interface to the AVRDUDE program (`http://www.nongnu.org/avrdude/`) which is a utility that allows modifications of the contents of ROMs and EEPROMs for Atmel microcontrollers. AVRDUDE makes use of In System Programming (ISP), which allows for the chip to be programmed while installed in the end system. In other words, you do not need to remove the chip or have it pre-programmed. This saves time and money during development and when deploying firmware updates.

3.3 Connecting the Universal Programmer

To program the ATmega128 chip, you will first need to connect your computer to the `osuisp2` programmer device with the provided USB cable. Then connect the `osuisp2` board to your ATmega128 board via the ribbon cable as depicted in Fig. 6. Double check that all your connections are solid. No additional drivers are required and Windows 7 should be able to detect the connected hardware.

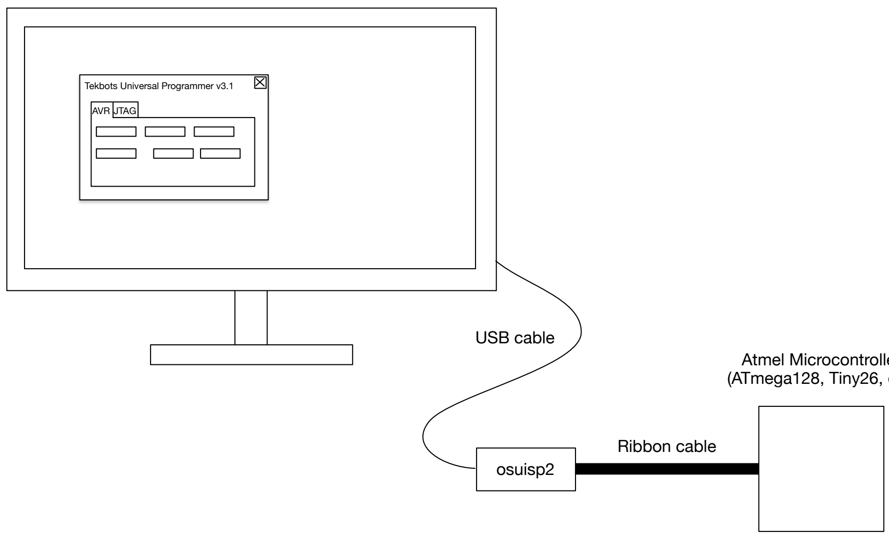

Figure 6: How to connect your laptop/computer to the Universal Progammer and Atmega128 board.

3.4 Programming Your Microcontroller

To run the Universal Programmer software, after you download the corresponding zip file from the TekBots website and extract its contents you will find an executable called "Universal_GUI.exe" in the extracted folder. Double click on the executable to bring up the window shown in Fig. 7. To verify the connection, click on **Identify**. The Universal Programmer v3.1 should automatically detect your ATmega128 chip. If **Identify** does not work as expected, try setting the clock period to a higher value and double check that all connections shown in Fig. 6 are secure. You may also try another USB port on your computer. In lab, you may also run into a station with a bad USB port. In those circumstances inform your TA so that it may be noted. If you think your programmer device is bad, contact TekBots for a replacement.

Once the Universal Programmer software successfully detects your chip, you should be ready to program it. To do this, check the **Flash** checkbox. This will enable the **Browse** button, allowing you to navigate to the .hex file generated after performing a successful compilation of your assembly code. See Fig. 8.

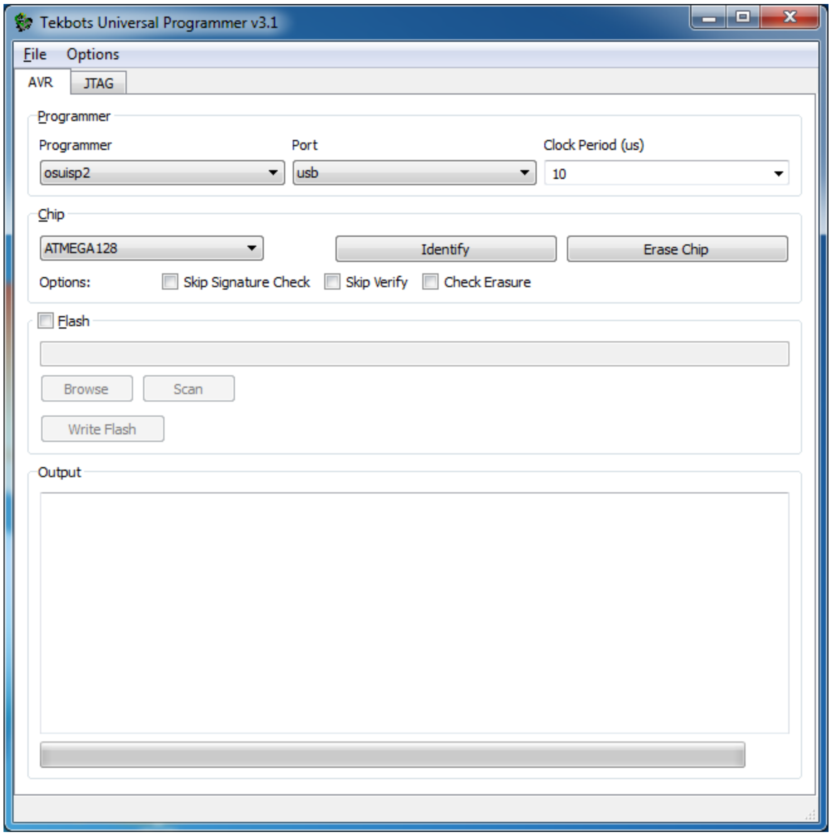

Figure 7: Universal Programmer Software with the default tab (AVR) selected.

Now click on **Write Flash** to load your program to the flash memory of the ATmega128. Always make sure the correct .hex file is chosen. Many times students will choose a .hex file for a different lab or project leading to improper operation of their TekBot according to the assigned task.

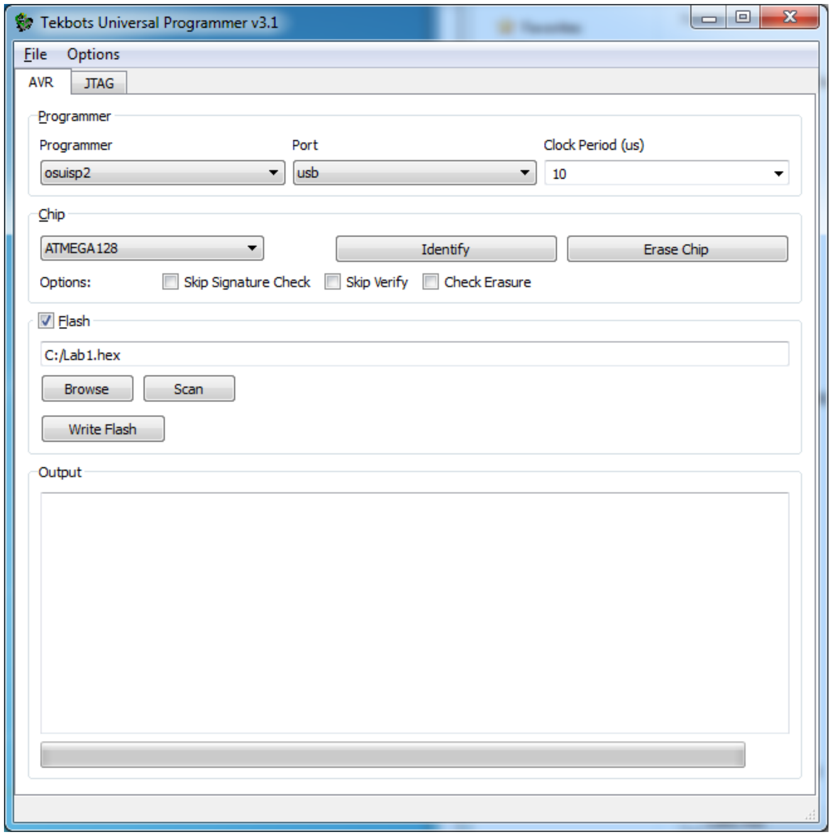

Figure 8: Selecting Flash checkbox with path to .hex file to be programmed.

4 ATmega128

This section will provide some basic information about the ATmega128 microcontroller. For more detailed information about the ATmega128, see the Complete Datasheet at http://www.atmel.com/atmel/acrobat/doc2467.pdf.

4.1 Useful Registers

The ATmega128 is equipped with two types of registers, general purpose registers and special function registers. All the registers in the ATmega128 are 8-bit, which means they are basically just 8 flip-flops in a row and are, in actuality, just part of the memory, each with their own address. Unlike other areas of memory that are tied into a multiplexor, registers are tied directly to the CPU and hold valuable data for the CPU.

4.1.1 General Purpose Registers

There are 32 General Purpose Registers in the ATmega128. These registers are connected to the ALU (Arithmetic Logic Unit) and are used to manipulate data. The registers are labeled R0 - R31.

Lower Registers The lower 16 registers, R0 - R15, work just like rest of the registers with the exception of loading immediate data i.e., the use of **LDI instruction.** These registers have access to the full range of the Data Memory, ALU, and additional peripherals. Here is an example of using the loading immediate data into the lower registers:

```
LDI R16, 30 ; Load the number 30 into R16
MOV R0, R16 ; Copy R16 into R0, R0 <- R16
INC R0 ; Increment R0, R0 <- R0+1
ADD R0,R16 ; R0<-R0+R16,value in R0 should now be 61
```

Upper Registers The upper 16 registers, R16 - R31, have additional capabilities. They have access to immediate data using the **LDI** instruction. These registers will be the ones that get the most use throughout your program. To move data into or out of these registers, the various different Load and Store instructions are needed. All arithmetic instructions work on these registers. Here is an example of using the upper registers:

```
LDI   R16, $A4 ; Load the immediate hex value into R16
LD    R17, X ; Load value from memory address in X-Pointer
ADC   R16, R17 ; Add with carry, R16 <- R16 + R17 + Carry Bit
ST    Y, R16 ; Store value in R16 to address in Y-Pointer
```

X-, Y-, Z-Registers The last six of the General Purpose Registers have additional functionality. They serve as the pointers for indirect addressing. The ATmega128 has a 16-bit addressing scheme that requires two registers for the address alone. The AVR RISC structure supports this scheme with the X, Y, and Z-Registers. These registers are the last six General Purpose Registers (R26-R31). The following table details the register assignments:

Name	Byte	Assignment
X-Register	Low	R26
	High	R27
X-Register	Low	R28
	High	R29
X-Register	Low	R30
	High	R31

Table 1: Address Register Assignments

The following code is an example of how to use these special registers. The code will read a value from SRAM, manipulate it, and then store it back at the next address in SRAM.

```
LDI   R26, $5A ; Load 0x5A into the low Byte of X
LDI   R27, $02 ; Load 0x02 into the high Byte of X
LD    R16, X+ ; Load value from SRAM, increment X
INC   R16 ; Manipulate value
ST X, R16 ; Store value to SRAM
```

4.1.2 Special Function Registers

Special Function Registers are registers in the ATmega128 that either control or monitor the various components of the chip. In this section, I will cover the commonly used Special Function Registers. Most of the Special Function Registers have read/write capabilities, check the datasheet for the ATmega128 for more details. These register reside in the ATmega128 I/O Memory and as such, require the **IN** and **OUT** instructions to read and write to these registers.

Status Register (SREG) The Status Register or SREG contains the important information about the ALU such as the Carry Bit, Overflow Bit, and Zero Bit. These bits are set and cleared during ALU instructions. This register becomes extremely useful during branching operations. The following table details the bit assignments within the SREG.

Bit	Name	Description
7	I	Global Interrupt Enable
6	T	Bit Copy Storage
5	H	Half Carry Flag
4	S	Sign Bit
3	V	Twos Compliment Overflow Flag
2	N	Negative Flag
1	Z	Zero Flag
0	C	Carry Flag

Table 2: SREG Description

As an example of using this register, look at the **BREQ** or Branch If Equal instruction. When this instruction is called, it looks at the Zero Flag in the SREG. If the Zero Flag is set, then the instruction will branch to the program address specified, otherwise it will continue on as usual.

Stack Pointer The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. The Stack Pointer Register always points to the top of the Stack. Note that the Stack is implemented as growing from higher memory locations to lower memory locations. This implies that a Stack **PUSH** command decreases the Stack Pointer.

The AVR Stack Pointer is implemented as two 8-bit Special Function Registers, Stack Pointer High Register (SPH) and Stack Pointer Low Register (SPL). The following diagram is a representation of the Stack Pointer.

Bit	15	14	13	12	11	10	9	8	
	SP15	**SP14**	**SP13**	**SP12**	**SP11**	**SP10**	**SP9**	**SP8**	**SPH**
	SP7	**SP6**	**SP5**	**SP4**	**SP3**	**SP2**	**SP1**	**SP0**	**SPL**
	7	6	5	4	3	2	1	0	

Figure 9: Stack Pointer

The following example demonstrates how to initialize the Stack Pointer. Remember to include the definition file for the ATmega128 at the beginning of the program to utilize Register naming schemes.

```
.include "m128def.inc"         ; Include definition file in program

      LDI   R16, LOW(RAMEND)  ; Low Byte of End SRAM Address
      OUT   SPL, R16           ; Write byte to SPL
      LDI   R16, HIGH(RAMEND)  ; High Byte of End SRAM Address
      OUT   SPH, R16           ; Write byte to SPH
```

I/O Ports The ATmega128 is equipped with 7 I/O Ports labeled Port A through Port G. Each port has it own unique capabilities such as External RAM Addressing, PWMs, Timers, and Counters. Unfortunately, this document will only cover the basics that are common with each port. For more detail information, refer to the Complete Datasheet for the ATmega128.

The I/O Port is the most fundamental way to move data in to or out of the ATmega128. The ports are bi-directional I/O ports with optional internal pull-ups. Throughout the rest of the description, certain names are abbreviated that are common to all registers. For example, a generic pin on an I/O port is referred to as Pxn, where x is the port name (A- G) and n is the pin number (0-7).

Each port pin consists of three Register bits: DDxn, PORTxn, and PINxn. The DDxn bits are accessed at the the DDRx Special Function Register (SFR), the PORTxn bits at the PORTx SFR, and the PINxn bits at the PINx SFR. To alleviate confusion, these three Special Function Registers for Port A are detailed below.

Bit	7	6	5	4	3	2	1	0	
	PORTA7	**PORTA6**	**PORTA5**	**PORTA4**	**PORTA3**	**PORTA2**	**PORTA1**	**PORTA0**	**PORTA**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 10: Port A Data Register - PORTA

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 11: Port A Data Direction Register - DDRA

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Figure 12: Port A Input Pins Address - PINA

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

Table 3 summarizes the control signals for the pin value.

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Table 3: Port Pin Configurations

Note: When reading a value from the port, read from the PINx Register. When writing a value to the port, write to the PORTx register.

The following code is an example of how to initialize the ports. This example initializes Port B as output and Port D as input. Remember, this code uses the definition document that is included somewhere in the beginning of the program.

```
LDI    R16, FF ; Select Direction as Output on all pins
OUT    DDRB, R16 ; Set value in DDRB
LDI    R16, FF ; Set Initial value to high on all pins
OUT    PORTB, R16   ; Set PORTB value, Port B pins should be high
```

```
LDI      R16, 00 ; Select Direction as Input on all pins
OUT    DDRD, R16 ; Set value in DDRD
LDI      R16, 00 ; Use normal Tri-state with no Pull-up resister
OUT    PORTD, R16   ; Port D is now ready as input
```

Additional Special Function Registers Details on other Special Function Registers can be obtained from the complete datasheet for the ATmega128. Future version of this document might discuss other registers if needed for the ECE 375 class.

4.2 Interrupt Vectors

Interrupts are special functions that are automatically called when trigger in the hardware of the ATmega128. In general, interrupts are enabled or disabled through the Global Interrupt Enable, bit 7 of the SREG. There are some AVR Assembly instructions that do this as well, SEI, Set Global Interrupt, and CLI, Clear Global Interrupt. Of course, just turning the Global Interrupt Enable on and off won't activate the interrupts themselves.

Each interrupt has a specific enable bit in the Special Function Register. To find out how to enable a specific interrupt, refer to the complete datasheet for the ATmega128. Once an interrupt is triggered, the current instruction address is saved to the stack and the program address is sent to that specific Interrupt Vector. An Interrupt Vector is a specific address in the program memory associated with the interrupt. There is general enough room at the Interrupt Vector to make a call to the interrupt function somewhere else in program memory and a return from interrupt instruction.

Table 4 shows a list of all the Interrupt Vectors, as well as there addresses in program memory, and definitions.

See Section 4.4 Starter Code to find out how to code the Interrupt Vector.

4.3 Memory Specifications

This section describes the different memories in the ATmega128. The AVR architecture has two main memory spaces, the Data Memory and the Program Memory space. In addition, the ATmega128 features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

4.3.1 Program Memory

The Atmega128 contains 128K bytes of On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 64K x 16. For software security, the Flash Program memory space is divided into two sections, Boot Program section and Application Program section. The figure 13 illustrates the Flash Memory.

Constant tables can be allocated within the entire program memory address space. To access these constant, or any data within the program memory, use the

Vector No.	Program Address	Source	Interrupt Definition
1	$0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow
12	$0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	$0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	$001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	$001C	TIMER1 OVF	Timer/Counter1 Overflow
16	$001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	$0020	TIMER0 OVF	Timer/Counter0 Overflow
18	$0022	SPI, STC	SPI Serial Transfer Complete
19	$0024	USART0, RX	UASRT0, Rx Complete
20	$0026	USART0, UDRE	USART0 Data Register Empty
21	$0028	USART0, TX	USART0, Tx Complete
22	$002A	ADC	ADC Conversion Complete
23	$002C	EE READY	EEPROM Ready
24	$002E	ANALOG COMP	Analog Comparator
25	$0030	TIMER1 COMPC	Timer/Counter1 Compare Match C
26	$0032	TIMER3 CAPT	Timer/Counter3 Capture Event
27	$0034	TIMER3 COMPA	Timer/Counter3 Compare Match A
28	$0036	TIMER3 COMPB	Timer/Counter3 Compare Match B
29	$0038	TIMER3 COMPC	Timer/Counter3 Compare Match C
30	$003A	TIMER3 OVF	Timer/Counter3 Overflow
31	$003C	USART1, RX	USART1, Rx Complete
32	$003E	USART1, UDRE	USART1 Data Register Empty
33	$0040	USART1, TX	USART1, Tx Complete
34	$0042	TWI	Two-wire Serial Interface
35	$0044	SPM READY	Store Program Memory Ready

Table 4: Reset and Interrupt Vectors

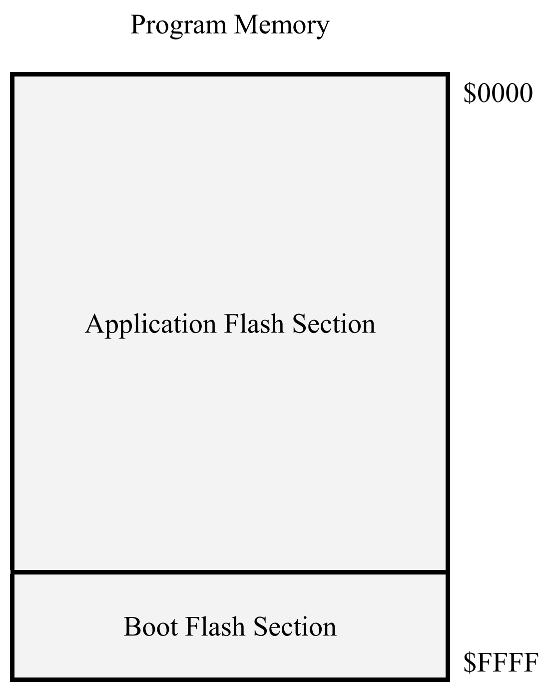

Figure 13: Program Memory Map

AVR instructions **LPM** - Load Program Memory or **ELPM** - Extended Load Program Memory. Refer to the instruction description in the AVR Instruction Set document to learn how to use them.

4.3.2 SRAM Data Memory

The ATmega128 supports two different configurations for the SRAM data memory, Normal mode and ATmega103 Compatibility mode. This mode is selected in the fuse bits of the ATmega128. For the purposes of this class and document, only the Normal mode will be discussed and used. The figure 14 illustrates the memory map for the SRAM.

The ATmega128 is a complex microcontroller with more peripheral units than can be supported within the 64-byte location reserved in the Opcode for **IN** and **OUT** instructions. For the Extended I/O space from 60−FF in SRAM, only the **ST/STS/STD** and **LD/LDS/LDD** instructions can be used.

The first 4352 Data Memory locations address the Register file, the I/O Memory and the internal data SRAM. The first 32 locations address the Register file, the next 64 locations address the standard I/O memory, 160 locations of Extended I/O memory, and then the next 4096 locations address the internal data SRAM.

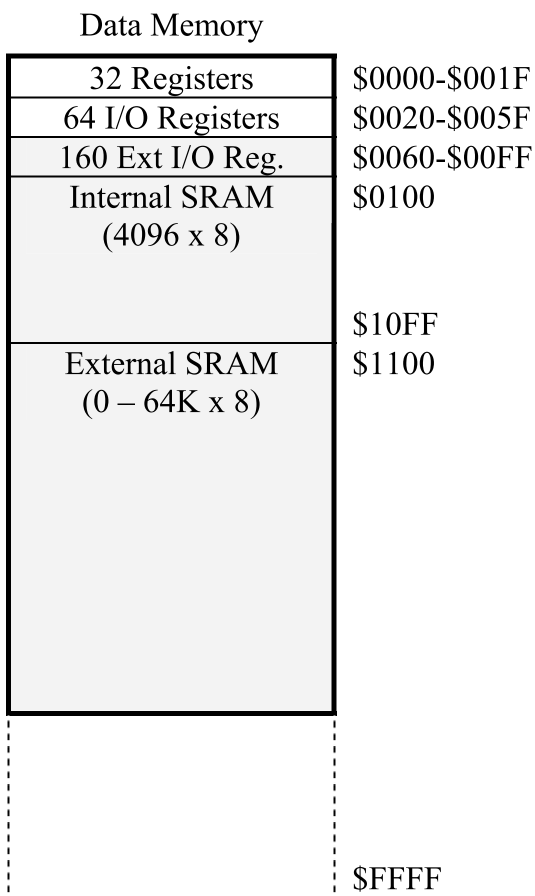

Figure 14: Data Memory Map

An optional external data SRAM can be used with the ATmega128. Refer to the datasheet for information on how to accomplish this.

The five different addressing modes for the data memory cover: Direct, Indirect with Displacement, Indirect, Indirect with Pre-decrement, Indirect with Post-increment. In the General Purpose Register file, registers R26 to R31 feature the indirect addressing pointer registers (X, Y, Z). See the section on X-, Y-, Z-Registers for more information about these registers.

The direct addressing reaches the entire data space.

The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y- or Z-register.

When using register indirect addressing modes with automatic pre-decrement and post- decrement, the address registers X, Y, and Z are decremented or incremented.

4.3.3 EEPROM Data Memory

The ATmega128 contains 4K bytes of data EEPROM memory. The EEPROM memory is useful in certain applications where modifying the data in the field becomes important. For the purposes of the class and this document, the EEPROM is not discussed in detail. For information on the EEPROM, refer to the datasheet for the ATmega128.

4.4 Starter Code

This section is an AVR program that can be used as a good starting point for any program. Just copy the program directly into a new project. The code shows a good coding technique as well as leaves areas for the interrupt vectors. It also gives some of the basic initialization routines.

```
;***************************************************************
;*
;*    Project Name Here
;*
;* Project Description Here
;* ;*************************************************************
;*
;*    Author:  Your Name
;*    Lab:  Lab Number Here
;* Date: Enter the date Here
;* ;*************************************************************
.include "m128def.inc"  ; Definition file for ATmega128
;***************************************************************
;* Program Constants
;***************************************************************
      .equ  const =00  ; Generic Constant Structure
;***************************************************************
;* Program Variables Definitions
;***************************************************************
      .def  mpr   =r16  ; Multipurpose Register
;***************************************************************
;* Interrupt Vectors
;***************************************************************
 .cseg
.org 0000     ; Define start of Code segment
      rjmp  RESET  ; Reset Handler
      reti               ; IRQ0 Handler
      nop
      reti               ; IRQ1 Handler
      nop
      reti               ; Timer2 Compare Handler
      nop
```

```
	reti			; Timer2 Overflow Handler
	nop
	reti			; Timer1 Capture Handler
	nop
	reti			; Timer1 CompareA Handler
	nop
	reti			; Timer1 CompareB Handler
	nop
	reti			; Timer1 Overflow Handler
	nop
	reti			; Timer0 Overflow Handler
	nop
	reti			; SPI Transfer Complete Handler
	nop
	reti			; USART RX Complete Handler
	nop
	reti			; UDR Empty Hanlder
	nop
	reti			; USART TX Complete Handler
	nop
	reti			; ADC Conversion Complete Handler
	nop
	reti			; EEPROM Ready Hanlder
	nop
	reti			; Analog Comparator Handler
	nop
	reti			; Two-wire Serial Interface Handler
	nop
	reti			; Store Program Memory Ready Handler
	nop
;***********************************************************************
;* Func: RESET
;* Desc: Reset Handler Routine
;***********************************************************************
RESET:
	; ***** Stack Pointer Init *****
	ldi		mpr, LOW(RAMEND)
	out		SPL, mpr
	ldi		mpr, HIGH(RAMEND)
	out		SPH, mpr
	rjmp	MAIN
;***********************************************************************
;* Func: MAIN
;* Desc: Main Entry into program
;***********************************************************************
MAIN:
```

```
;< Insert your program Here>
      rjmp        MAIN          ; Loop Indefinitely
```

To use the Interrupt Vector, create the Interrupt Handler Routine similar to the RESET routine. Then replace the NOP instruction for the interrupt and replace it with an RJMP command to the newly created Interrupt Handler Routine.

5 AVR Assembly Programming

This section concentrates on the usage of the AVR Instruction Set. This guide will not cover every single Instruction or all the Assembler Directives. Instead, it will cover the most basic and commonly used instructions and directives and give techniques on how to use them efficiently to get the most out of your program. For a detailed list of instructions, refer to the Atmel's *AVR Instruction Set*. It can be found at http://www.atmel.com/atmel/acrobat/doc0856.pdf. For a complete detailed list of all the AVR Assembler Directives, refer to Section 4.5 of *AVR Assembler*. It can be found at http://www.atmel.com/atmel/acrobat/doc1022.pdf.

5.1 Pre-compiler Directives

Pre-compiler directives are special instructions that are executed before the code is compiled and directs the compiler. These instructions are denoted by the preceding dot, i.e. .EQU. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory, and so on. The following sections will contain detailed information on the most commonly used directives. The following table contains an overview of the directives supported by the AVR Assembler.

Table 5: Pre-Compiler Directives

Directive	Description
`.BYTE`	Reserve byte to a variable
`.CSEG`	Code Segment
`.DB`	Define constant byte(s)
`.DEF`	Define a symbolic name on a register
`.DEVICE`	Define which device to assemble for
`.DSEG`	Data Segment
`.DW`	Define constant words
`.ENDMACRO`	End macro
`.EQU`	Set a symbol equal to an expression
`.ESEG`	EEPROM segment
`.EXIT`	Exit from a file
`.INCLUDE`	Read source from another file
`.LIST`	Turn listfile generation on
`.LISTMAC`	Turn macro expression on
`.MACRO`	Begin Macro
`.NOLIST`	Turn listfile generation off
`.ORG`	Set program origin
`.SET`	Set a symbol to an expression

5.1.1 CSEG – Code Segment

The CSEG directive defines the start of a Code Segment. An assembler file can contain multiple Code Segments, which are concatenated into one Code Segment when assembled. The directive does not take any parameters.

Syntax:

```
.CSEG
```

Example:

```
.DSEG                     ; Start Data Segment
      vartab: .BYTE 4     ; Reserve 4 bytes in SRAM
.CSEG
      const:       .DW 2  ; Write 0x0002 in program memory
      mov    r1, r0       ; Do something
```

5.1.2 DB – Define constant byte(s)

The DB directive reserves memory resources in the program memory or the EEPROM memory. In order to be able to refer to the reserved locations, a label should precede the DB directive.

The DB directive takes a list of expressions, and must contain at least one expression. The list of expressions is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between –128 and 255 since each expression is represented by 8-bits. A negative number will be represented by the 8-bits two's complement of the number.

If the DB directive is used in a Code Segment and the expression list contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. If the expression list contains an odd number of expressions, the last expression will be placed in a program memory word of its own, even if the next line in the assembly code contains a DB directive.

Syntax:

```
LABEL:  .DB expressionlist
```

Example:

```
.CSEG
consts:
      .DB 0, 255, 0b01010101, -128, $AA
text:
      .DB "Hello World"
```

5.1.3 DEF – Set a symbolic name on a register

The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used to the rest of the program to refer to the registers it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.

Syntax:

```
.DEF Symbol = Register
```

Example:
```
.DEF temp = R16
.DEF ior = R0
.CSEG
            ldi   temp, $F0   ; Load 0xF0 into temp register
            in    ior, $3f    ; Read SREG into ior register
            eor   temp, ior   ; Exclusive or temp and ior
```

5.1.4 EQU – Set a symbol equal to an expression

The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined.

Syntax:
```
.EQU label = expression
```

Example:
```
.EQU io_offset = $23
.EQU porta = io_offset + 2
.CSEG
            clr   r2          ; Clear register 2
            out   porta, r2   ; Write to Port A
```

5.1.5 INCLUDE – Include another file

The INCLUDE directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until the end of file (EOF) or an EXIT directive is encountered. An include file may itself contain INCLUDE directives.

Syntax:
```
.INCLUDE "filename"
```

Example:
```
                              ; iodefs.asm
.EQU  sreg = $3F              ; Status register
.EQU  sphigh = $3E            ; Stack pointer high
.EQU  splow = $3D             ; Stack pointer low
                              ; incdemo.asm
.INCLUDE "iodefs.asm"         ; Include I/O definitions
      in    r0, sreg          ; Read status register
```

5.1.6 ORG – Set program origin

The ORG directive sets the location counter to an absolute value. The value to set is given as a parameter. If an ORG directive is given within a Code Segment, then it is the Program memory counter that is set. If the directive is preceded by a label (on the same source line), the label will be given the value of the parameter. The default value of the Code location counter is zero when assembling is started. Note that the Program memory location counter counts words and not bytes.

Syntax:

```
        .ORG expression
```

Example:

```
        .CSEG
                rjmp    main            ;  Jump to the main section of code
        .ORG $0042                      ; Set location counter to address $0042 to skip the
                                        ; interrupt vectors
        main:                           ; Main section of code
                mov     r0, r1          ; Do something
```

5.2 Expressions

The Assembler incorporates expressions. Expressions can consist of operands, operators and functions. All expressions are internally 32 bits.

5.2.1 Operands

The following operands can be used:

- User defined labels that are given the value of the location counter at the place they appear.
- User defined constants defined by the EQU directive.
- Integer constants: constants can be given in several formats, including
  a) Decimal (default): 10, 255
  b) Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
  c) Binary: 0b00001010, 0b11111111
- PC – the current value of the Program memory location counter

5.2.2 Functions

The following functions are defined:

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the high byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bites 16-21 of an expression
- EXP2(expression) returns 2^expression
- LOG2(expression) returns the integer part of log2(expression)

5.2.3 Operators

The Assembler supports a number of operators that are described in section 4.6.3 of the *AVR Assembler* document. These operators can be commonly associated with C/C++ operators. Note that these operations are done only during compilation and cannot be used in place of the AVR Instructions.

5.3 Basic Instructions

Almost all AVR Instructions fall into three categories; Arithmetic and Logic Instructions, Branch Instructions, and Data Transfer Instructions. This section will not cover every instruction in the AVR Instruction Set; instead, it will support the *AVR Instruction Set* document by expanding on key instructions and their uses.

5.3.1 Common Nomenclature

With the exception of a few instructions, all AVR Assembly Instructions follow a common nomenclature. There are three parts to every instruction, the Instruction Name, Argument 1, and Argument 2.

Every AVR Instruction has an instruction name. This name is a unique three or four letter combination that identifies the instruction; for example, the AVR Instruction the Loads an Immediate Value has the Instruction Name of **LDI**.

Also, every instruction may have up to two arguments associated with it. These arguments follow the Instruction Name and are separated by a comma. When arguments are used, it is important to note that the result of the command will always be stored in the first argument. Please note the figure below.

CMD	**ARG1,**	**ARG2**
Instruction Name	Argument1	Argument2

ARG1 ← CMD(ARG1, ARG2)

Result of operation

Figure 13: AVR Instruction Nomenclature

5.3.2 Arithmetic and Logic Instructions

The arithmetic and logic instructions make use of the microcontroller's ALU. Almost all of the arithmetic and logic instructions consist of a two arguments and can modify all of the status bits in the SREG. Take note that all of the arithmetic and logic instructions are 8-bit only. The following is a breakdown of the available instructions:

- Addition: ADD, ADC, ADIW
- Subtraction: SUB, SUBI, SBC, SBCI, SBIW
- Logic: AND, ANDI, OR, ORI, EOR
- Compliments: COM, NEG
- Register Bit Manipulation: SBR, CBR
- Register Manipulation: INC, DEC, TST, CLR, SER
- Multiplication[1]: MUL, MULS, MULSU
- Fractional Multiplication[1]: FMUL, FMULS, FMULSU

[1] Multiplication and Division is very limited and restrictive

There is a common nomenclature to the naming of the instructions. The following table explains the nomenclature.

Table 6: Common Instruction Nomenclature

Ending Letter	Meaning	Description
C	Carry	Operation will involve the carry bit
I	Immediate	Operation involves an immediate value that is passed as the second argument.
W	Word	The operation is a 16-bit operation.
S	Signed	The operation handles signed numbers
SU	Signed/Unsigned	The operation handles both signed and unsigned.

5.3.3 Branch Instructions

Branch Instructions are used to introduce logical decisions and flow of control within a program. About 20% of any program consists of branches. A branch instruction is basically an instruction that can modify the Program Counter (PC) and redirect where the next instruction is fetched. There are two types of branch instructions, unconditional branches and conditional branches.

5.3.3.1 Unconditional Branches

Unconditional branches modify the PC directly. These instructions are known as jumps because they cause the program to "jump" to another location in program memory. There are several types of jump instructions (RJMP, IJMP, EIJMP, JMP), but the most common one is the relative jump, RJMP, because it takes the least amount of cycles to perform and can access the entire memory array.

There are also special unconditional branch instructions known as function calls, or calls (RCALL, ICALL, EICALL, CALL). The function calls work just like the jump instructions, except they also push the next address of the PC on to the stack before making the jump. There is also a corresponding return instruction, RET, that pops the address from the stack and loads it into the PC. These instructions are used to create functions in AVR assembly. See Section 5.6 for more details on functions.

5.3.3.2 Conditional Branches

Conditional branches will only modify the PC if the corresponding condition is meant. In AVR, the condition is determined by looking at the Status Register (SREG) bits. For example, the Branch Not Equal, BRNE, instruction will look at the Zero Flag (Z) of the SREG. If Z = 0, then the branch is taken, else the branch is not taken. At first this might not seem very intuitive, but in AVR, all the comparisons take place before the branch.

There are several things that can modify the SREG bits. Most arithmetic and logic instructions can modify all of the SREG bits. But what are more commonly used is the compare instructions, (CP, CPC, CPI, CPSE). The compare instructions will subtract the two corresponding registers in order to modify the SREG. The result of this subtraction is not stored back to the first argument. With this in mind, take a look at BRNE again. If the values in two register are equal when they are subtracted, then the resulting value would be zero and then Z = 1. If they were not equal then Z would be 0. Now when BRNE is called, the Z bit can determine the condition. Section 5.5 shows several examples of how to use this process. The following table gives a nice quick summary of all conditional tests, the corresponding instruction, and what bits in SREG are manipulated to determine the condition.

Table 7: Conditional Branch Summary

Test	Boolean	Mnemonic	Complementary	Boolean	Mnemonic	Comment
Rd > Rr	$Z \bullet (N \oplus V) = 0$	BRLT(1)	Rd ≤ Rr	$Z + (N \oplus V) = 1$	BRGE*	Signed
Rd ≥ Rr	$(N \oplus V) = 0$	BRGE	Rd < Rr	$(N \oplus V) = 1$	BRLT	Signed
Rd = Rr	Z = 1	BREQ	Rd ≠ Rr	Z = 0	BRNE	Signed
Rd ≤ Rr	$Z + (N \oplus V) = 1$	BRGE(1)	Rd > Rr	$Z \bullet (N \oplus V) = 0$	BRLT*	Signed
Rd < Rr	$(N \oplus V) = 1$	BRLT	Rd ≥ Rr	$(N \oplus V) = 0$	BRGE	Signed
Rd > Rr	C + Z = 0	BRLO(1)	Rd ≤ Rr	C + Z = 1	BRSH*	Unsigned
Rd ≥ Rr	C = 0	BRSH/BRCC	Rd < Rr	C = 1	BRLO/BRCS	Unsigned
Rd = Rr	Z = 1	BREQ	Rd ≠ Rr	Z = 0	BRNE	Unsigned
Rd ≤ Rr	C + Z = 1	BRSH(1)	Rd > Rr	C + Z = 0	BRLO*	Unsigned
Rd < Rr	C = 1	BRLO/BRCS	Rd ≥ Rr	C = 0	BRSH/BRCC	Unsigned
Carry	C = 1	BRCS	No carry	C = 0	BRCC	Simple
Negative	N = 1	BRMI	Positive	N = 0	BRPL	Simple
Overflow	V = 1	BRVS	No overflow	V = 0	BRVC	Simple
Zero	Z = 1	BREQ	Not zero	Z = 0	BRNE	Simple

Note: 1. Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr → CP Rr,Rd

Additionally, conditional branches include the skip instructions (SBRC, SBRS, SBIC, SBIS). These instructions will skip the next instruction if the condition is meant. These can be very useful in determining whether or not to call a function. Note that the skip instructions are limited to only bit testing on registers or specific areas in IO Memory.

5.3.4 Data Transfer Instructions

The majority of instructions in any assembly language program are data transfer instructions. These instructions essentially move data from one area in memory to another. As easy as this concept seems, it can quickly become very complicated and overwhelming. For example, the AVR Instructions Set supports five different addressing modes: Immediate, Direct, Indirect, Indirect with Pre-Decrement, Indirect with Post-Increment, and Indirect with Displacement. But each of these modes can be broken down into comprehensible sections.

5.3.4.1 Immediate Addressing

Immediate addressing is simply a way to move a constant value into a register. Only one instruction supports immediate addressing, LDI. Also note that this instruction will only work on the upper 16 General Purpose Registers, R16 – R31. The following is an example of when LDI would be used. Suppose there was a loop that needed to be looped

16 times. Well, a counter register could be loaded with the value 16 and then decremented after each loop. When the register reached zero, then the program will exit from the loop. Since the value 16 is a constant, we can load into the counter register by immediate addressing. The following code demonstrates this example.

```
.def  counter = r22           ; Create a register variable
      ldi   counter, 16       ; Load the immediate value 16 in counter
Loop: breq  Exit              ; If zero, exit loop
      adc   r0, r1            ; Do something
      dec   counter           ; Decrement the counter
      rjmp  Loop              ; Redo the loop
Exit: inc   r0                ; Continue on with program
```

5.3.4.2 Direct Addressing

Direct addressing is the simplest way of moving data from one area of memory to another. Direct addressing requires only the address to access the data. But it is limited to the use of the register file. For example, if you wanted to move a byte of data from one area in Data Memory to another area in Data Memory, you must first Load the data a register and then Store the data into the other area of memory. In general, every data manipulation instruction, except LDI, comes in a Load and Store pair. For Direct Addressing modes, the instruction pairs are LDS/STS and IN/OUT.

The point of having multiple instruction pairs is to access different areas of memory.

- LDS/STS – Move data in and out of the entire range of the SRAM Data Memory
- IN/OUT – Move data in and out of the IO Memory or $0020 - $005F of the SRAM Data Memory. IN/OUT takes less instruction cycles than LDS/STS does.

The following is an example loop that continually increments the data value at a particular address.

```
.equ  addr = $14D0            ; Address of data to be manipulated
Loop: lds   r0, addr          ; Load data to R0 from memory
      inc   r0                ; Increment R0
      sts   addr, r0          ; Store data back to memory
      rjmp  Loop              ; Jump back to loop
```

5.3.5 Bit and Bit-test Instructions

Bit and Bit-test Instructions are instructions that manipulate or test the individual bits within an 8-bit register. There are three types of Bit and Bit-testing instructions; Shift and Rotate, Bit Manipulation, and SREG Manipulation.

5.3.5.1 Shift and Rotate

Shifting a register literally means shifting every bit one spot to either the left or the right. The AVR Instruction set specifies register shifts as two types of instructions, shifts and rotates. Shifting will just shift the last bit out to carry bit and shift in a 0 to the first bit. Rotating will shift out the last bit to the carry bit and shift in the carry bit to the first bit. Therefore rotating a register will not loose any bit data while shifting a register will loose the last bit. The instruction mnemonics are LSL, LSR, ROL, and ROR for Logical Shift Left, Logical Shift Right, Rotate Left Through Carry, and Rotate Right Through Carry respectively.

There are also some special Shifting Instructions, Arithmetic Shift Right (ASR) and Swap Nibbles (SWAP). Arithmetic Shift Right behaves like a Logical Shift Right except it does not shift out to the carry bit. Instead, ASR, will right shift anywhere from 1 to 7 spaces. Swap Nibbles will swap the upper and lower 4-bits with each other.

5.3.5.2 Bit Manipulation

Bit Manipulation Instructions allow the programmer to manipulate individual bits within a register by setting, or making the value 1, and clearing, or making the value 0, the individual bits. There are three instruction pairs to manipulate the SREG, an I/O Register, or a General Purpose Register through the T flag in the SREG. BSET and BCLR will set and clear respectively any bit within the SREG register. SBI and CBI will set and clear any bit in any I/O register. BST will store any bit in any General Purpose Register to the T flag in the SREG and BLD will load the value of the T flag in the SREG to any bit in any General Purpose Register.

5.3.5.3 SREG Manipulation

Although the instructions SBI and CBI will allow a programmer to set and clear any bit in the SREG, there are additional instructions that will set and clear specific bits within the SREG. This is useful for when the programmer does not want to keep track of which bit in the SREG is for what. The following table shows the mnemonics for each set and clear instruction pair are in the table below.

Table 8: SREG Bit Manipulation Instructions

Bit	Bit Name	Set Bit	Clear Bit
C	Carry Bit	SEC	CLC
N	Negative Flag	SEN	CLN
Z	Zero Flag	SEZ	CLZ
I	Global Interrupt Flag	SEI	CLI
S	Signed Test Flag	SES	CLS
V	Two's Complement OVF Flag	SEV	CLV
T	T Flag	SET	CLT
H	Half Carry Flag	SEH	CLH

5.4 Coding Techniques

This section contains general hints and tips to produce well-structured code that can be easily debugged and save a lot of time and headaches.

5.4.1 Structure

It is important to create and maintain a consistent code structure throughout the program. Assembly Language in general can be greatly confusing; a well-structured program will ease this confusion and make the program very readable to yourself and other people. Spending several hours trying to find a specific problem area in a piece of code can become quite frustrating.

So what does a well-structured program look like? Structure includes everything that is typed in the program, where certain parts of the program are located, how an instruction looks within a line, etc. There are several ways to write out the code on the 'paper', but the most important part is to be consistent. If you start writing your code in one fashion, maintain that fashion through out the remainder of the program. Varying between different 'styles' can be quite confusing and make the code unreadable.

The one style that I recommend is using the four-column method. If this style is used consistently throughout the program, the program should look like four columns. A column is usually separated with one or two tabs depending on how long the data strings are. In general, the following table describes what goes into each column, the tab lengths and exception rules.

Table 9: Line Formatting Rules

Column	Tab Length	Includes	Comments
1	1	Pre-compiler Directives, Labels	• If a label is longer than one tab length, then the instruction mnemonic goes on the next line. • No instructions must be placed on the same line as a pre-compiler directive.
2	1	Directive Parameters, Instruction Mnemonic	• It is common for Directive Parameters to exceed one tab length.
3	2	Instruction Parameters	• If Instruction Parameters exceed two tab lengths, then place the comment on the previous line.

4	3	Comments	• Comments should only be used in the fourth column, which is roughly four tab lengths from the start of the line. • Unless the code is blatantly obvious, place a comment on every line of code. • Exceptions are Header Comments, they must start at the beginning of the line

The following is an example of well-formatted code using the rules from Table 9: Line Formatting Rules.

```
; Title:    XOR Block of Data
; Author:   David Zier
; Date:     March 16, 2003

.include "m128def.inc"        ; Include definition file

.def  tmp = r15               ; Temporary Register
.def  xor = r6                ; XOR Register
.equ  addr1 = $1500           ; Beginning Address
.equ  addr2 = $2000           ; Ending Address

                              ; This code segment will XOR all the
                              ; bytes of data between the two address
                              ; ranges.
.org  $0000                   ; Set the program starting address
INIT: ldi   XL, low(addr1)    ; Load low byte of start address in X
      ldi   XH, high(addr1)   ; Load high byte of start address in X
FETCH:                        ; Code won't fit, create a new line
      ld    tmp, X+           ; Load data from address into tmp
      eor   xor, tmp          ; XOR tmp with xor register, store in xor
      cpi   XL, low(addr2)    ; Compare high byte of X with End Address
      brne  FETCH             ; If low byte is not equal, then get next
      cpi   XH, high(addr2)   ; Compare low byte of X with End Address
      brne  FETCH             ; If high byte is not equal then get next
DONE: rjmp  DONE              ; Infinite done loop
```

The next part to proper code structure is code placement. Certain sections of code should be placed in certain areas. This alleviates confusion and allows the contents to be ordered and navigable. The following table illustrates the order in which certain code segments are to be placed.

Table 10: Code Structure

Header Comments	Title, Author, Date, and Description
Definition Includes	Specific Device Definition Includes, i.e. "m128def.inc"
Register Renaming	Register renaming and variable creation, i.e. .def tmp = r0
Constant Declaration	Constant declarations and creation, i.e. .equ addr = $2000
Interrupt Vectors	See Section 4.2 Interrupt Vectors
Initialization Code	Any initialization code goes here
Main Code	The heart of the program.
Subroutines	Any subroutine that is created follows the main code.
ISRs	Any Interrupt Subroutines will go here.
Data	Any hard coded data is best placed here, i.e. .DB "hello"
Additional Code Includes	Finally, if there is any additional source code includes, will go last.

By following these simple structure rules, the code will be more readable and understandable.

5.4.2 Register Naming

Register naming is an important part to any program. It alleviates confusion and makes the code more readable, thus it will be easier to debug. The main purpose to renaming a register is to assign a register as a specific variable type. For example, if I wanting a temporary register that I would use through out the program to hold one-shot data, I would name the register "tmp". If I was righting a program that executed a complex arithmetic routine, I might want a variable to store the result, so I name a register "res".

The reason you would rename a register is to alleviate confusion. If I just used the regular register names (r0, r1, r2, etc.), I could easily get confused as to what each register was used for. Was r0 the register to hold the result or was r13? As the program grows more complex, this can easily be the case.

Register names should be short but descriptive and unique. Short names fit well into the four-column formatting scheme. Names should be no longer the six letters, but on average, be 3 letters in length. For example, "tmp" for temporary, "res" for result, "addrl" for address low byte, or "cnt" for count. Make the names as descriptive as possible. Using a register named "tmp" that always holds the value to be compared is not good; a better name might be "cmp" for compare. And finally, the name must be unique. Don't name several registers "tmp1", "tmp2", and "tmp3". This is no better than using r1, r2, and r3. In fact it is worse, because you have to type more letters. If there is ever situations where multiple temporary register are to be used, make them unique. An example might be "otmp" and "itmp" for outer loop temporary and inner loop temporary respectively.

Proper register naming will make coding easier, so it is a good idea to get in the habit. Also, bad register naming can make code much more difficult to work with. In general, a segment of code that using register renaming can be easily understandable, even without the comments.

5.4.3 Constants and Addressing

Like register renaming, constant names should be short, descriptive, and unique. So when does one use a constant? If you ever find your self repeatedly using the same constant value over and over again, then a constant is needed. Constants are beneficial in two ways; one, they make the code more readable and thus more easy to debug, and two, allow you change the behavior of the program by adjusting the constant numbers in the beginning of the code.

If there were no constants, a programmer might have to search through the entire code to see particular value was and change, maybe even multiple times. With constants, this requires only one edit and no searching each time the programmer wants to change a value.

Common uses of constants are with set addresses. One thing to note is that addresses are 16-bits and registers are 8-bits. This means that you **must** deal with addresses as low and high bytes. There are several things to be concerned with here. First, when comparing addresses always compare both the low and the high bytes, even if the high byte doesn't change. It is very possible, that by the time the program is finished, the high byte might change and since it was not compared, the program does not function properly. Next, the programmer would need to consider how to use and access a 16-bit constant. The following code is one example of how to access the address $23D4.

```
.equ  addrl = $D4             ; Low byte of address
.equ  addrh = $23             ; High byte of address
...
      ldi   XL, addrl         ; Load low byte of address
      ldi   XH, addrh         ; Load high byte of address
```

This will works, but it is not a good method for accessing the low and high bytes. Below is a better and much preferred method.

```
.equ  addr = $23D4            ; The address
...
      ldi   XL, low(addr)     ; The low byte of the address
      ldi   XH, high(addr)    ; The high byte of the address
```

This is better because now you can see the address in its entirety. The previous method split the address into two separate byte constants, which can become confusing when changing the address. For example, what if I wanted to have another address constant and I put it as high then low. I might get confused and enter the wrong byte of the address because it is not consistent with the previous method. By putting the entire address, you can easily read or edit the address. Additionally, inserting another address will not be confusing since it is consistent. We later use the high() and low() macros to access the high and low bytes.

It is a good idea to name every constant that you use within your code. It is quick, easy, and saves you a lot of time when coding and debugging. It is definitely more beneficial for you to use constants than to not.

5.4.4 ATMega128 Definition File

A definition file is a file that contains addresses and values for common I/O registers and special registers within a specific chipset. For example, every ATMEL AVR chipset contains an SREG, but not every chipset has the SREG in the same memory location. This is where the definition file comes in. Just write your code with the common name for the I/O register such as SREG or SPH, and then include the definition file in the beginning of you code. This does two things, first, the programmer doesn't have to look up or memorize the address for each to the I/O register or chip specific registers and second, the same code can be used for different chipset by just including the proper definition file.

Since this document mostly concentrates on the ATMega128, we use the definition file so that we don't have to look up the address for specific I/O registers. The definition file is for the ATMega128 is *m128def.inc*. It contains a lot of .equ and a few .def expressions. The file also contains useful information such as the last address in SRAM (RAMEND). It is included with AVR Studio4 when you download and install the program.

5.5 Flow of Control

This section will contain several examples of C-like flow of control expressions and how to code the same thing in AVR Assembly. These flow of control examples will show the proper way to use a branch instruction and more importantly, what a branch instruction is used for.

5.5.1 IF Statement

This is probably the most simplest and straightforward control statement in program. In C, the IF statement is commonly seen as:

```
if (expr)
    statement
```

If *expr* is nonzero (*true*), then *statement* is executed; otherwise *statement* is skipped, and control passes to the next statement. This is true for assembly as well. For example, the following C-code.

```
      if (n >= 3)
      {
            expr++;
            n = expr;
      }
```

Here is the equivalent version in assembly.

```
.def  n = r0
.def  expr = r1
.equ  cmp = 3

...
      cpi   n, cmp      ; Compare value
IF:   brsh  EXEC        ; If n >= 3 then branch to NEXT
      rjmp  NEXT        ; Jump to NEXT since expression is false
EXEC: inc   expr        ; increment expr
      mov   n, expr     ; Set n = expr
NEXT: ...               ; continue on with code
```

Although this code behaves like the C-code, it is not optimal. By simply using the complementary Boolean Expression, you can save space and speed up the program.

```
.def  n = r0
.def  expr = r1
.equ  cmp = 3
...
      cpi   n, cmp      ; Compare value
IF:   brlo  NEXT        ; If n >= 3 is false then skip code
      inc   expr        ; increment expr
      mov   n, expr     ; Set n = expr
NEXT: ...               ; Continue on with code
```

This statement behaves exactly the same but uses one less branch statement and one less line of code. And more importantly, is easier to read and understand.

5.5.2 IF-ELSE Statement

This is very similar to the IF statement, except it has an additional unconditional else statement. This is not too hard to implement. Here is an example C-Code.

```
      if (n == 5)
            expr++;
      else
            n = expr;
```

And here is the equivalent code in AVR Assembly.

```
.def  n = r0
.def  expr = r1
.equ  cmp = 5
...
      cpi   n, cmp        ; Compare value
      breq  IF            ; Branch to IF if the n == 3
      rjmp  ELSE          ; Branch to ELSE if the expression is false
IF:   inc   expr          ; Increment expression
      rjmp  NEXT          ; Goto NEXT
ELSE: mov   n, expr       ; Set n = expr
NEXT: ...                 ; Continue on with code
```

We can make this more efficient if we use the complimentary Boolean expression.

```
.def  n = r0
.def  expr = r1
.equ  cmp = 5
...
      cpi   n, cmp        ; Compare value
IF:   brne  ELSE          ; Goto ELSE statement since expression is false
      inc   expr          ; Execute the IF statement
      rjmp  NEXT          ; Continue on with code
ELSE: mov   n, expr       ; Execute the ELSE statement
NEXT: ...                 ; Continue on with code
```

Again, this code has one less branch statement and one less instruction. Although this does not seem like much now, but if this were nested within a loop that looped 100 times, then it is essentially 100 less instructions to be executed.

5.5.3 IF-ELSIF-ELSE Statement

This is simply a nested mix of the IF and IF-ELSE statements. A C example would be:

```
      if (n < 3)
            expr++;
      else if (n == 5)
            n = expr;
      else
            n++;
```

Here is how to logically convert it into assembly.

```
.def  n = r0
.def  expr = r1
.equ  val1 = 3
.equ  val2 = 5
...
      cpi   n, val1       ; Compare n with val1
      brlo  IF            ; If n < 3, then execute if
      rjmp  ELIF          ; Goto ELSEIF Expression
IF:   inc   expr          ; Execute if statement
      rjmp  NEXT          ; Goto Next
ELIF: cpi   n, val2       ; Compare n with val2
      breq  ELIE          ; If n == 5, then execute ELSEIF statement
```

```
      rjmp  ELSE        ; Goto ELSE statement
ELIE: mov   n, expr     ; Execute ELSEIF statement
      rjmp  NEXT        ; Goto Next
ELSE: inc   n           ; Execute ELSE statement
NEXT: ...               ; Continue on with code
```

This seems a little complicated and confusing. By changing the Boolean expressions, the code can be optimized and less confusing.

```
.def  n = r0
.def  expr = r1
.equ  val1 = 3
.equ  val2 = 5
...
      cpi   n, val1     ; Compare n with val1
IF:   brsh  ELIF        ; If is not n < 3, then goto ELSEIF expression
      inc   expr        ; Execute if statement
      rjmp  NEXT        ; Goto Next
ELIF: cpi   n, val2     ; Compare n with val2
      brne  ELSE        ; If is not n == 5, then goto ELSE expression
      mov   n, expr     ; Execute ELSEIF statement
      rjmp  NEXT        ; Goto Next
ELSE: inc   n           ; Execute ELSE statement
NEXT: ...               ; Continue on with code
```

This optimized code has two less instructions and two less branches. In addition, it is easier to read and understand.

5.5.4 WHILE Statement

The WHILE statement is commonly used to create repetitive loops. In fact, it is common to use an infinite while loop to end a program. Consider a construction of the form:

```
      while (expr)
            statement
      next statement
```

First *expr* is evaluated, if it is nonzero (*true*), the *statement* is executed, and control is passed back to the beginning of the WHILE loop. The effect of this is that the body of the WHILE loop, namely the *statement*, is executed repeatedly until *expr* is zero (*false*). At that point control passes to *next statement*. An example is:

```
      while (n < 10) {
            sum += n;
            n++;
      }
```

In assembly, WHILE loops can be created pretty easily. Here is the equivalent assembly code:

```
.def  n = r0
.def  sum = r3
.equ  limit = 10
```

```
...
WHIL: cpi   n, limit    ; Compare n with limit
      brlo  WHEX        ; When n < limit, goto WHEX
      rjmp  NEXT        ; Condition is not meet, continue with program
WHEX: add   sum, n      ; sum += n
      inc   n           ; n++
      rjmp  WHIL        ; Go back to beginning of WHILE loop
NEXT: ...               ; Continue on with code
```

This code can also be optimized as follows:

```
.def  n = r0
.def  sum = r3
.equ  limit = 10
...
WHIL: cpi   n, limit    ; Compare n with limit
      brsh  NEXT        ; When not n < limit, goto NEXT
      add   sum, n      ; sum += n
      inc   n           ; n++
      rjmp  WHIL        ; Go back to beginning of WHILE loop
NEXT: ...               ; Continue on with code
```

By converting the BRLO to BRSH, we where able to remove one of the branch instructions and make the code look more like a WHILE loop.

5.5.5 DO Statement

The DO statement can be considered a variant of the WHILE statement. Instead of making its test at the top of the loop, it makes it at the bottom. The following is an example:

```
      do {
            sum += n;
            n--;
      } while (n > 0);
```

The assembly code for the DO statement is also very similar to the WHILE statement.

```
.def  n = r0
.def  sum = r3
.equ  limit = 0
...
DO:   add   sum, n      ; sum += n
      dec   n           ; n++
      cpi   n, limit    ; compare n to limit
      brne  DO          ; since n is unsigned, brne is same expr
NEXT: ...               ; Continue on with code
```

As you can see, a DO statement provides better performance over the optimized WHILE statement. But even this function can be optimized.

```
.def  n = r0
.def  sum = r3
```

```
.equ  limit = 0
...
DO:   add   sum, n      ; sum += n
      dec   n           ; n++
      brne  DO          ; since n is unsigned, brne is same expr
NEXT: ...               ; Continue on with code
```

Although the optimization does not affect the performance of the DO statement in general, it did for this case. Since DEC is called before the BRNE instruction, the CPI instruction is not needed. The CPI instruction forces the specific bits in the SREG to occur that are needed by the branching instructions. In this case, the DEC instruction will work as well. For example, when the DEC instruction decrements the *n* value and *n* becomes zero, then the Zero Flag in the SREG is set. This is the only bit that is checked by the BRNE command. Thus we can completely remove the CPI instruction.

5.5.6 FOR Statement

The FOR statement, like the WHILE statement, is used to execute code iteratively. We can explain its action in terms of the WHILE statement. The construction

```
      for (expr1; expr2; expr3)
            statement
      next statement
```

is semantically equivalent to

```
      expr1;
      while (expr2) {
            statement
            expr3;
      }
      next statement
```

provided that *expr2* is present. FOR loops are commonly used to run through a set of data. For example, the following is some code that iterates 10 times.

```
      for (n = 0; n < 10; n++)
            sum += n;
```

The following assembly is the equivalent of the C code.

```
.def  n = r0
.def  sum = r3
.equ  max = 10
...
      ldi   n, 0        ; Initialize n to 0
FOR:  cpi   n, max            ; Compare n to max value
      brlo  EXEC        ; If n < max, the goto EXEC
      rjmp  NEXT        ; Statement is false, break out of FOR loop
EXEC: add   sum, n            ; sum += n
      inc   n           ; decrement n
      rjmp  FOR         ; goto the start of FOR loop
```

```
NEXT: ...               ; rest of code
```

There are several things to do to optimize this code, first, use a DO loop instead of a WHILE loop. Next use the complemented form of the expression. And lastly, initialize the variable *n* to max and decrement it. This will allow use to use the SREG technique from Section 5.5.5.

```
.def  n = r0
.def  sum = r3
.equ  max = 10
...
      ldi   n, max      ; Initialize n to max
FOR:  add   sum, n      ; sum += n
      dec   n           ; decrement n
      brne  FOR         ; repeat loop if n does not equal 0
NEXT: ...               ; rest of code
```

This removed seven instructions and two branches. In addition, the code is simpler and easier to read. And more importantly, it has the same functionality as the C code. This is also a good example of why to name constants. If we wanted the FOR loop to loop 25 times, then all we would have to do is change the max constant from 10 to 25. No sweat!

5.5.7 SWITCH Statement

The SWITCH statement is a multiway conditional statement generalizing the IF-ELSE statement. The following is a typical example of a SWITCH statement:

```
      switch (val) {
      case 1:
            a_cnt++;
            break;
      case 2:
      case 3:
            b_cnt++;
            break;
      default:
            c_cnt++;
      }
```

The case statement is probably the most complicated to write in assembly. Here is the logical form for the above C code example.

```
.def  val = r0
.def  a_cnt = r5
.def  b_cnt = r6
.def  c_cnt = r7
...
SWITCH:                 ; The beginning of the SWITCH statement
      cpi   val, 1      ; Compare val to 1
      breq  S_1         ; Branch to S_1 if val == 1
      cpi   val, 2      ; Compare val to 2
      breq  S_3         ; Branch to S_3 if val == 2
      cpi   val, 3      ; Compare val to 3
```

```
      breq  S_3         ; Branch to S_3 if val == 3
      inc   c_cnt       ; Execute Default
      rjmp  NEXT        ; Break out of switch
S_1:  inc   a_cnt       ; Execute case 1
      rjmp  NEXT        ; Break out of switch
S_3:  inc   b_cnt       ; Execute case 2
NEXT: ...               ; The rest of the code
```

This is the general idea, although some might even nest the execution within condition expressions to make it more logically correct. Yet, using the complementary Boolean expression can optimize this code segment. (Do you a similar pattern yet!)

```
.def  val = r0
.def  a_cnt = r5
.def  b_cnt = r6
.def  c_cnt = r7
...
SWITCH:                 ; The beginning of the SWITCH statement
S_1:  cpi   val, 1      ; Compare val to 1
      brne  S_2         ; Branch to S_2 if val != 1
      inc   a_cnt       ; Execute case 1
      rjmp  NEXT        ; Break out of switch
S_2:  cpi   val, 2      ; Compare val to 2
      brne  S_3         ; Branch to S_3 if val != 2
      inc   b_cnt       ; Execute case 2
      rjmp  NEXT        ; Break out of switch
S_3:  cpi   val, 3      ; Compare val to 3
      brne  DFLT        ; Branch to DFLT if val != 3
      inc   b_cnt       ; Execute case 3
      rjmp  NEXT        ; Break out of switch
DFLT: inc   c_cnt       ; Execute default
NEXT: ...               ; The rest of the code
```

Believe it or not, this code actually has better optimization than the former. In the former, any given case statement will have to go through two branches before it is executed. In the optimized version, it will only have to go through one branch. If you take a look at the worse case scenario, there are fewer jumps to get to the default statement as well. Therefore this is the optimal code, even though there are more instructions.

5.6 Functions and Subroutines

AVR Assembly language has the ability create and execute functions and subroutines. By simply using a subroutine or a function, a programmer can drastically reduce the size and complexity of the code. Because of the importance, this topic is given its own subsection. This section will cover the general topics of functions and subroutines, included how to create one, when to create one, and how to use one.

5.6.1 Definitions

A function or a subroutine can generally be thought as "reusable code". Reusable code is any segment of code that can be used over and over through out the program without duplicity or two copies of the same code. You can think of functions and subroutines as they are implemented in a C program. The function is created outside of the main program and then is later called within the main program, sometimes in multiple areas.

Reusable code within an assembly program can be thought of as either a function or a subroutine. Both are very similar in terms of how they are implemented, but have subtle differences.

A subroutine is a reusable piece of code that requires no input from the main program. Generally, the state of the program is saved upon entering the subroutine and is restored before leaving the subroutine. This is perfect for when servicing interrupts.

A function, on the other hand, is involved within the main code and requires some interaction. This usually means that some registers or other memory has to be initialized before the function is called. In addition, a function will most likely alter the state of the program.

In general, a subroutine must not alter the state of a running program and no care must be taken to ensure the proper operation of a subroutine. For a function, the main program must initialize data for the function to work properly and must be able to handle any changes caused by the function.

5.6.2 Operational Overview

A subroutine or function is called via the CALL, RCALL, ICALL, or EICALL instructions and is matched with an RET instruction to return to the instruction address after the call. The function or subroutine is preceded by a label that signifies the name of function or subroutine. When a CALL instruction is implement, the processor first pushes the address of the next instruction after the CALL instruction onto the stack. This is important to realize since it means that the stack **must** be initialized before functions or subroutines can be used. The CALL instruction will then jump to the address specified by label used as the parameter. The next instruction to be executed will then be the first instruction with the subroutine or function. Upon exiting the subroutine or function, the return instruction, RET, must be called. The RET instruction will then pop the address of the next instruction after the CALL instruction from the stack and load into the PC. Thus the next instruction to be executed is the instruction after the CALL instruction.

It is important to keep track of what is pushed and popped on the stack. If within a subroutine or function, data is not popped correctly, the RET instruction can pop the wrong data values for the address and thus the program will not function correctly. Additionally, **never** exit a subroutine or function via another jump instruction other than RET. Doing so will cause the data in the stack to never be popped and thus the stack will become out of sink.

So when does a programmer decide to create a subroutine or function? This can often be a tough call and can get really complicated. The general rule of thumb is that a subroutine or function is needed when the programmer finds that the same segment of code is being written in several places within the program. This can be very troublesome since an error within the code segment result in several hours of work trying to fix every instance of the code segment. On the other hand, if the program were to use a function or subroutine, the fixing the code is easy since there is only one instance and it is in a common location.

5.6.3 Implementation

In the last section, we briefly talked about how a function or a subroutine works. This section will give detailed explanations on how to implement both the function and the subroutine.

5.6.3.1 Setup

The first thing to do for any function and subroutine is to initialize the stack. This can be done in four lines of code at the beginning of the program. Optimally, the stack should be initialized for any program. Here is the code:

```
.include "m128def.inc"

INIT:                        ; Initial the stack pointer
      ldi   r16, low(RAMEND)  ; Load the lo byte of the ram's end addr
      out   SPL, r16          ; Set the Stack Pointer Low register
      ldi   r16, high(RAMEND) ; Load the hi byte of the ram's end addr
      out   SPH, r16          ; Set the Stack Pointer High register
```

After this point, any function or subroutine will correctly in regards to the stack.

5.6.3.2 Subroutine Implementation

The subroutine does not require any outside influence for its performance. Therefore it is a good idea to save the state of the program before executing the subroutine. This means that certain registers must be pushed to the stack in the beginning of the subroutine and popped just before the subroutine ends. It is important to remember to pop registers in reverse order from which they where pushed. These registers include the SREG (essentially the state of the program) and any general purpose registers that are used throughout subroutine.

A good example of a routine is a wait loop that will wait for a specific amount of time. For our case, we will want the wait subroutine to wait for 1000 cycles (not including the subroutine overhead cycles.)

```
.def  ocnt = r16              ; Outer loop count variable
.def  icnt = r17              ; Inner loop count variable
WAIT:                         ; Wait subroutine
      push  icnt              ; Save icnt register
      push  ocnt              ; Save ocnt register
      in    ocnt, SREG        ; Get the SREG value
      push  ocnt              ; Save the value of the SREG
      ldi   ocnt, 55          ; Loop outer loop 55 times
WTL1: ldi   icnt, 5           ; Loop inner loop 5 times
WTL2: dec   icnt              ; Decrement inner loop counter
```

```
      brne  WTL2              ; Continue through inner loop
      dec   ocnt              ; Decrement outer loop counter
      brne  WTL1              ; Continue through outer loop
      ldi   ocnt, 3           ; This next loop just uses 9 cycles
WTL3: dec   ocnt              ; that is need for the 1000 cycles
      brne  WTL3              ; Repeat last loop 3 times
      nop                     ; We still come up 1 cycle short
      pop   ocnt              ; Get the SREG value
      out   SREG, ocnt        ; Restore the value of the SREG
      pop   ocnt              ; Restore ocnt register
      pop   icnt              ; Restore icnt register
      ret                     ; Return from subroutine
```

We are going to stray off topic for a second to discuss this code. From the time first LDI instruction is called to the last NOP, there are exactly 1000 cycles of execution. To calculate this, you must take into consideration the number of cycles it takes to execute each instruction. With the exception of the branches, every instruction takes one cycle. The branches take 1 cycle if false and 2 cycles if true. With this in mind, the main loop follows the equation $((3*icnt + 3)*ocnt$. The optimal values for *icnt* and *ocnt* are 5 and 55 respectively. This yields the total number of cycles for the main to be 990 cycles. This is unfortunately 10 cycles short of our goal. We could just shove 10 consecutive NOPs at the end, but instead opted for a second small loop. This second loop follows the equation $3*ocnt$ and with a value of *ocnt* being 3 yields 9 cycles. Therefore, with the addition of a single NOP instruction, our total number of cycles is 1000 cycles.

Now back to the topic of subroutines. As you can see in the example code, the very first thing we do is push the SREG to the stack. Additionally, we also push the registers *icnt* and *ocnt* since they are used within the subroutine. We then execute the main subroutine code. This followed up by popping the data from the stack in the reverse order that it is pushed. And finally, we leave the subroutine with the RET instruction. Another item to notice is that PUSH and POP only deal with the general-purpose registers. Therefore, if we wanted to push an I/O Register, such as the SREG, we must first load it into a general-purpose register. Since we were already using the *ocnt* register within the subroutine, we used it to temporarily hold the SREG value. We made sure the *ocnt* register was first pushed to the stack so that we didn't loose any data that might have been there.

So now how does a programmer use a subroutine once it has been created? Well, this is easier than it sounds. Since a label precedes the subroutine, we can just make a call to the label with one of the CALL instructions.

```
MAIN: ldi   r16, 4            ; Set r16 to 4
LOOP: rcall WAIT              ; Call our WAIT routine
      dec   r16               ; Decrement r16
      brne  LOOP              ; Call the wait statement 4 times
      ...                     ; Additional code
      rcall WAIT              ; Call our WAIT routine
      ...                     ; Even more code
DONE: rjmp  DONE              ; Program complete
```

As you can see, this program will call the WAIT routine in several places. Now if the routine did not exist, then we would have to write all that code twice in several places. While this is not hard with modern technologies such as cut and paste, the might still initially be incorrect. This means that we now have to search through the rest of the code and this can quickly become a headache. By now, you should be able to understand how subroutines can make your programming experience better.

5.6.3.3 Function Implementation

A function is a bit different from a subroutine in the fact that it alters the state of the program. Unlike the subroutine, the function will most likely need to be initialized prior to the function call. This is how to create input to a function. Also the function will most likely modify some registers to provide output. With the exceptions of the input and output, a function behaves just like a routine. Any registers that used within a function but do not correspond to any input or output, must still be pushed and popped from the stack.

We will use a common example of a function that takes two 8-bit numbers and multiplies them together. Since an 8-bit multiplication results in a 16-bit number, the result will be stored in the same registers that were used for the input. The two input registers are the multiplier (*mplr*) and the multiplicand (*mplc*). The high byte of the result will be stored in *mplr* and the low byte will be stored in the *mplc*. The basic multiplication algorithm will be to repeatedly add the multiplicand to itself for the amount specified in the multiplicand. So here is the function.

```
.def  mplr = r18              ; The multiplier
.def  mplc = r19              ; The multiplicand
.def  resh = r1               ; The high byte of the result
.def  resl = r2               ; The low byte of the result
.def  zero = r0               ; Zero register that always contains 0

MUL:                          ; The multiplication function
      push  resh              ; Save the state of resh
      push  resl              ; Save the state of resl
      push  zero              ; Save the zero register
      clr   zero              ; Enforce the 0 in the zero register
      clr   resh              ; Clear the result high byte
      clr   resl              ; Clear the result low byte
      cpi   mplr, 0           ; Initially check mplr for 0
      breq  EXIT              ; Check for end condition
ADD:  add   resl, mplc        ; Add multiplicand to result
      adc   resh, zero        ; This just add the carry bit, if any
      dec   mplr              ; Decrement mplr
      brne  ADD               ; Repeat loop if mplr is not 0
EXIT: mov   mplr, resh        ; Move high byte of result to mplr
      mov   mplc, resl        ; Move low byte of result to mplc
      pop   zero              ; Restore zero register
      pop   resl              ; Restore resl register
      pop   resh              ; Restore resh register
      ret                     ; Return from function
```

As you can see, the general structure and layout is very similar to a subroutine. The only difference is that we did not push and pop the registers *mplr* and *mplc*, since they are used for input and output. You will also note the SREG was not saved. Sometimes, a function will want to return the value of the SREG and therefore it is not necessarily vital to save it.

Now we will see how to utilize a function. The process is very similar to subroutine except that the input values need to be initialized before the function is called. It is also important to note that register renaming for the functions input and outputs should be done at the beginning of the program with the programs renamed registers. Like the subroutine, any register that is used internally can be renamed before the instance of the function. So here is an example of how to use the function we created in the above example.

```
.def  mplr = r18
.def  mplc = r19

INIT: ldi   r16, high(RAMEND) ; Initialize the stack pointer high byte
      out   SPH, r16
      ldi   r16, low(RAMEND)  ; Initialize the stack pointer low byte
      out   SPL, r16
MAIN: ...                     ; Other code in the main program
      ldi   mplr, 25          ; Load 25 into multiplier
      ldi   mplc, 93          ; Load 93 into multiplicand
      rcall MUL               ; Multiply 93 * 25
      st    X+, mplr          ; Do something with the result
      st    X, mplc           ; by storing it into memory
      ...                     ; Additional Code
      mov   mplr, r5          ; Setup another multiplication function
      mov   mplc, r8          ; with the registers r5 and r8
      rcall MUL               ; Multiply r8 * r5
      mov   r0, mplr          ; Do something with the result
      mov   r1, mplc          ; by storing the result into r0:r1
      ...                     ; More code
DONE: rjmp  DONE              ; Program complete
```

As you can see, we initialize the function by storing the value we want to multiply into the *mplr* and *mplc*. We then call the multiplication function, MUL. And finally we handle the results. What values you want to multiply and what to do with the result depends on the program. By now, you should have a good understanding as how to create and use a function.