AVR Toolchain Worksheet

# Introduction

This worksheet will walk you through the journey of C code from editor to MCU. Figure 1 shows the workflow Atmel Studio uses internally, however we want to explore these steps in more detail. We’ll compile and link our C code, use avr-objdump to examine the assembly code, and take a closer look at the .map file generated by the linker. We’ll also revise the Intel HEX file format, which we discussed in lecture.

# Atmel%20Studio%20to%20Executable.png

Figure 1. Atmel Studio to executable workflow

# Step 0: Setup your environment

The source code for Part 1 can be found on Canvas under Files > Labs > Lab3 – Emulator > AVR Toolchain Worksheet. You’ll need both main.c and iocompat.h!

* If you’re using Atmel Studio, you can go to Tools > Command Prompt and type commands into this window.
* If you’re using macOS or Linux, we recommend following this guide on Canvas to install the AVR toolchain, and then you can follow along in Terminal.

# Step 1: Compile

First we need to compile the source code with the commands below.

When compiling, the compiler needs to know the processor type so the -mmcu option is specified. The -Os option will tell the compiler to optimize the code for efficient space usage (at the possible expense of code execution speed). The -g is used to embed debug info. The debug info is useful for disassemblies and doesn't end up in the .hex files, so it’s useful to always specify it. Finally, the –c option tells the compiler to compile and stop -- don't link.

$ avr-gcc -g -Os -mmcu=atmega328p -c main.c

The compilation will create a main.o file.

**1.) Why can’t we directly run the object file created from the previous command on our microcontroller? i.e. What information is still missing after the previous command runs?**

*When the compiler compiles the main file, all the code within the main file is compiled into binary code that machines understand. However,* ***necessary function definitions*** *and* ***global variables*** *that do not belong to but referenced by the main program have not yet being linked.*

# Step 2: Link

Next we link it into a binary called main.elf.

$ avr-gcc -g -mmcu=atmega328p -o main.elf main.o

It is important to specify the MCU type when linking. The compiler uses the -mmcu option to choose start-up files and run-time libraries that get linked together. If this option isn't specified, the compiler defaults to the 8515 processor environment, which is most certainly what you didn't want.

**2.) What does the linker do? Why do we generate an ELF file?**

*The Linker* ***fills in any function definitions and global variables*** *that is required by the main program in order to create an executable application.*

*An elf file that is the assemble of executable, shared library, and relocatable files contains the* ***structure and necessary sections*** *of an application. It is used to* ***extract a hex file*** *for this application so that processors can execute.*

# Step 3: Objdump

Now we have a binary file. Can we do anything useful with it? The GNU Binutils suite is made up of many useful tools for manipulating object files that get generated. One tool is avr-objdump, which takes information from the object file and displays it in many useful ways. Typing the command by itself will cause it to list out its options.

For instance, to get a feel of the application's size, the -h option can be used. The output of this option shows how much space is used in each of the sections (the .stab and .stabstr sections hold the debugging information and won't make it into the ROM file).

An even more useful option is -S. This option disassembles the binary file and intersperses the source code in the output! By using –S with objdump rather than the compiler, the listing includes routines from the libraries and the vector table contents. Also, all the "fix-ups" have been satisfied. In other words, the listing generated by this option reflects the actual code that the processor will run.

$ avr-objdump -h -S main.elf > main.lst

Now let’s cat the output of the main.lst file and examine the contents in more detail. The top of the file lists the various sections of the program, followed by the interrupt vector table.

**3.) Generally speaking, what is stored in the .text, .data, and .bss memory segments?**

*.text contains program instructions that represent code, constant data and vector tables like interrupt vector table.*

*.data contains all initialized data. The linker allocates the data in Flash, and then the data is copied to RAM*

*.bss contains all uninitialized data, and they are stored in RAM.*

**4.) If the Timer 1 overflow interrupt is detected, to what address does the program jump? What is the first line of assembly of the ISR?**

***0x90****,* ***push r1***

**5.) How are other interrupt types handled by the program? To what address do these other interrupts jump?**

***Bad\_interrupt & ctor\_end*** *(interrupt end), they have starting address of* ***0x8c*** *and* ***0x68*** *respectively*

**6.) What is the “static” keyword used for? In what area of memory are static variables stored?**

*Static variables mean that the* ***life time of the variables is the same as the program****. It is stored in the* ***.data*** *segment.*

**7.) What is the endianess of the PWM variable?**

***Little endian***

**8.) What variables are stored at 0x800100, 0x800101, and 0x800102 in the data memory?**

*0x800100 stores* ***pwm***

*0x800101 stores* ***higher 8 bits of the pwm***

*0x800102 stores* ***direction***

**9.) Here we want you to learn to use the AVR Instruction Set manual, which can be found on Canvas under Files > Datasheets. What assembly lines are used to realize the switch logic? Annotate each line of assembly with an explanation of what it’s doing and why.**

*address 0xa0 – 0xda is used for realizing the switch logic*

*High level description:*

*Lines 0xa0 – 0xb2 loads the necessary variables and choose which case it should jump into*

*Lines 0xb4 – 0xc8 increments pwm, loads pwm and timer1\_ top which is 255, and compare these 2 variables, if true, direction gets assigned to 0 and jumps to 0xde, otherwise jumps to 0xde directly*

*Lines 0xca – 0xda decrements pwm, loads pwm and 0, and compare these 2 variables, if true, direction gets assigned to 1 and continues the execution, otherwise jumps to 0xde directly*

*Detailed description:*

*switch (direction) // This section determines the new value of the PWM*

*// loads the current direction into register 18*

*a0: 20 91 02 01 lds r18, 0x0102 ; 0x800102 <direction.1610>*

*// loads the current pwm into register 24*

*a4: 80 91 00 01 lds r24, 0x0100 ; 0x800100 <\_edata>*

*// loads the upper 8 bits of pwm into register 25*

*a8: 90 91 01 01 lds r25, 0x0101 ; 0x800101 <\_edata+0x1>*

*// update the zero flag by using register 18*

*ac: 22 23 and r18, r18*

*// test if zero flag is set, if not continues to next instruction, otherwise goto PC + 1 + 26*

*ae: 69 f0 breq .+26 ; 0xca <\_\_vector\_13+0x3a>*

*// compare the value in the Rd register with 1, updating the zero flag*

*b0: 21 30 cpi r18, 0x01 ; 1*

*// if the zero flag is cleared, go to PC + 1 + 42 otherwise continue execution.*

*b2: a9 f4 brne .+42 ; 0xde <\_\_vector\_13+0x4e>*

*{*

*case 1:*

*if (++pwm == TIMER1\_TOP)*

*// increment the pwm which is stored in r24*

*b4: 01 96 adiw r24, 0x01 ; 1*

*// store the upper 8 bits of the pwm into 0x800101*

*b6: 90 93 01 01 sts 0x0101, r25 ; 0x800101 <\_edata+0x1>*

*// update the current value of pwm in the memory*

*ba: 80 93 00 01 sts 0x0100, r24 ; 0x800100 <\_edata>*

*// compare pwm and timer1\_top*

*be: 8f 3f cpi r24, 0xFF ; 255*

*// subtract the upper 8 bits of the pwm with 3 and the C flag and store the result into r25, // updating the zero flag*

*c0: 93 40 sbci r25, 0x03 ; 3*

*// if the zero flag is cleared, go to PC+1+26, 0xdc otherwise, continue execution*

*c2: 69 f4 brne .+26 ; 0xde <\_\_vector\_13+0x4e>*

*direction = 0;*

*// set the direction to 0, r1 = 0.*

*c4: 10 92 02 01 sts 0x0102, r1 ; 0x800102 <direction.1610>*

*// jump from the current address to PC + 20 + 1, 0xdc*

*c8: 0a c0 rjmp .+20 ; 0xde <\_\_vector\_13+0x4e>*

*break;*

*case 0:*

*if (--pwm == 0)*

*// subtract pwm with 1 and store it back to r24*

*ca: 01 97 sbiw r24, 0x01 ; 1*

*// restore pwm’s upper 8 bits into 0x800101*

*cc: 90 93 01 01 sts 0x0101, r25 ; 0x800101 <\_edata+0x1>*

*// stores the new direction value back into 0x800100*

*d0: 80 93 00 01 sts 0x0100, r24 ; 0x800100 <\_edata>*

*// update the zero flag through r24 or r25 🡺 r24*

*d4: 89 2b or r24, r25*

*// branch into PC+ 6 + 1 if –pwm ! = 0*

*d6: 19 f4 brne .+6 ; 0xde <\_\_vector\_13+0x4e>*

*direction = 1;*

*// update the direction to 1*

*d8: 81 e0 ldi r24, 0x01 ; 1*

*// store the direction value into memory at 0x800102*

*da: 80 93 02 01 sts 0x0102, r24 ; 0x800102 <direction.1610>*

*break;*

*}*

# Step 5: Map File

Using avr-objdump is very useful, but sometimes it's necessary to see information about the link that can only be generated by the linker. A map file contains this information. A map file is useful for monitoring the sizes of your code and data. It also shows where modules are loaded and which modules were loaded from libraries. It is yet another view of your application.

To get a map file, add -Wl,-Map,main.map to the link command. Relink the application using the following command to generate main.map.

$ avr-gcc -g -mmcu=atmega328p -Wl,-Map,main.map -o main.elf main.o

**10.) At the top of the .map file the length of the .text segment is statically defined to be 0x00020000. What does this length represent?**

*This length is the* ***maximum number of bytes*** *the text segment can have in Atmega 328P.*

**11.) Where does the .text section start and end? In other words, what are the lowest and highest addresses of the .text segment?**

*From* ***0x00000000*** *to* ***0x0000013c***

**12.) How many bytes are in the .bss section? Why is the .bss section that size? Be specific in your justification!**

***3 bytes****, because there are* ***3 variables of size 8*** *bits each described in question number 8.*

# Step 6: Hex file

We have a binary of the application, but how do we get it into the processor? Most (if not all) programmers will not accept a GNU executable as an input file, so we need to do a little more processing. In this final step, we’ll extract portions of the binary and save the information into a .hex file. The GNU utility that does this is called avr-objcopy.

The ROM contents can be pulled from our project's binary and put into the file main.hex using the following command:

$ avr-objcopy -j .text -j .data -O ihex main.elf main.hex

The -j option indicates that we want the information from the .text and .data segment extracted. Check Lecture 4, slide 17 for more information about Intel HEX.

**13.) Fill in the C function below to calculate the CRC for a line (record) in an Intel HEX file programmatically. You may assume the byte count is always 16 (0x10), and you may add helper functions to be called by calculateChecksum.**

// this function assumes the input is always a 40 length char array. The way to add the sum is // the important thing here

*int8\_t calculateChecksum (char \* record){*

*unsigned int sum = 0x00;*

*for (int i = 0; i < 40; i+=2){*

*sum += (record[i] << 4) + record[i+1];*

*}*

*sum = ~sum;*

*sum += 0x01;*

*return sum & 0XFF;*

*}*

**14.) Use the function you wrote above to calculate the checksum for the following record:**

**:1002B00084BD15BC08958FEF87B98BB988B90E94**

***Sum = 0xaa***