USC Viterbi School of Engineering

Ming Hsieh Department of Electrical and Computer Engineering

EE 459Lx - Embedded Systems Design Laboratory

Programming the Microchip ATmega328P in C

by

Allan G. Weber

1 Introduction

This document discusses the use of the avr-gcc tool chain for programming the Microchip ATmega328P microcontroller in C. Students can install the software on their system by downloading it from a couple of web sites.

The programming environment described below is command-line based using the Mac's Terminal program or the Windows Command Prompt. If you are not comfortable using that type of programming environment, you'll have to find some other way to program the chips.

Much if this document is generic and applies to any processor in the ATmega or ATtiny family. However parts that discuss things like memory layout are specific to the ATmega328P. If the avr-gcc compiler is used for other processors the information in these areas will have to be adjusted to match the other processor.

Note: The ATmega328P microcontroller used to be made by Atmel until Atmel was acquired by Microchip Technology in 2016. Some documents on the EE459 web site may still refer to it as Atmel ATmega328P, but it's the same microcontroller and only the company name has changed.

2 Installing the Development Software

Use of the avr-gcc toolchain first requires installing the necessary software on the computer. Detailed instructions on the installation steps is available on the EE459 web site at these links:

- Macs: `https://ece-classes.usc.edu/ee459/library/toolchain_mac64`
- Windows: `https://ece-classes.usc.edu/ee459/library/toolchain_windows`
- Linux: `https://ece-classes.usc.edu/ee459/library/toolchain_linux`

3 Editing Files

In order to edit the text files associated with your project you will need a text editing program on your system. The main requirement is that it be able to edit a file of plain text characters and not turn it into something like an RTF (Rich Text Format) file or some other word processing document type. There are a large number of these available for free and it's pretty much personal preference as to which one to use.

For macOS a popular GUI editor is BBedit which is a free download. For all Unix-type systems (Mac OS X, Linux, FreeBSD, etc.) the most common command-line interface text editors are probably "`emacs`", "`vi`" or "`vim`" (improved `vi`). All are available in various forms from several Internet sites.

For Windows, the easiest one to find is probably "Notepad++". One requirement of editing text files on Windows is that the editor be able to save files without adding a filename extension like "`.txt`". If the editor insists on adding the extension it is usually necessary to then use a Command Prompt window to rename the file and remove the extension.

4 Starting a New Project

The easiest way to create a new project is to start with a couple of template files from the class web site. Create a new folder with a name for the new project.

> **Note:** It is strongly recommended that folder and file names do not contain spaces in them. You may use underscores if you wish (e.g. `lab_3`). Names with spaces are great when navigating folders in a GUI, but not so great (i.e. very painful) when using command line interfaces.

From the class web sites download the files "`Makefile`" and one of the sample AVR programs and put them both in the project folder.

The C file can be edited to create your program. The `Makefile` will also have to be edited to make sure the correct steps are done when building the project.

5 The Makefile

Programs can be compiled in several ways but one simple and efficient method is to use a utility program called "`make`". This program is run from a command line interface and will do all the necessary compiling of the source code and linking of the object files to create the executable binary file. The `make` program does its work on the basis of information in the text file "`Makefile`" and this file is the key to correct compiling and downloading of the programs. It contains all the information and commands needed to compile the program and produce binary data that can be downloaded into the ATmega328P processor on the project board.

One purpose of the `Makefile` is to describe "dependencies" between the various parts of the project. By knowing which program modules are dependent on other files, and examining the modification times of the various components, the `make` program only compiles the components of the program that need to be compiled. For a simple project this seems like more trouble than it's worth, but for a complex project that might have hundreds of source code files, the `make` program is invaluable and well worth learning how to use.

Once the proper information is in the `Makefile`, all the required program modules can be compiled and linked together by just typing the command "`make`". If a `Makefile` is copied from an existing project to a new one, only a a few lines at the top may have to be changed in order for the compiling, linking and downloading to work properly. Most of the remaining information in it can usually be left unchanged.

It's important to note that the name of the file must be "`Makefile`" (or "`makefile`"). On occasion your Mac or Windows machine may try to add an extension to the file and call it something like "`Makefile.txt`". This will prevent the `make` program from working. Some systems will add the extension, and then make the extension invisible so from the GUI the file looks like its name is `Makefile` but it's really not.

If the `make` command returns an error message about no Makefile found, use a Terminal or Command Prompt window to go to the project folder and the `ls` (Mac or Linux) or `dir` (Windows) command should list the files showing the full names. If necessary use the following command to fix the name.

```
rename makefile.txt makefile     <-- on Windows
mv Makefile.txt Makefile         <-- on Mac or Linux
```

Using your text editor, open the `Makefile` and look at the five lines at the top. These provide information on what type of processor is being used, how to program it, and what files need to be compiled to build the executable program. Regardless of where the `Makefile` came from, it's very important to make sure the correct information is at the top.

```
DEVICE     = atmega328p
CLOCK      = 9830400
PROGRAMMER = -c usbtiny -P usb
OBJECTS    = myprogram.o myfunctions.o
FUSES      = -U hfuse:w:0xd9:m -U lfuse:w:0xe0:m
```

The `DEVICE` line contains the name of the microcontroller and must be correct for the program to compile correctly. For the ATmega328P this should be "`atmega328p`".

The `CLOCK` line tells the frequency of the clock being used in Hertz. This needs to be correct so any calls to delay functions will work and also for the communication between the your computer and the microcontroller.

The `PROGRAMMER` line contains parameters used by the program that downloads the data to the microcontroller. This information is the same on all type of systems (Mac, Windows or Linux), but will differ depending on the type of programmer being used. For the USBtiny programmers in the black plastic boxes with clear tops this should be "`-c usbtiny -P usb`". For the blue AVRISP MkII programmers should be "`-c avrispmkii -P usb`".

The `OBJECTS` line lists all object modules that need to be linked together. If the program has been divided into several source files this line will contain more entries.

The `FUSE` line give the parameters used to program the high and low fuse bytes (see Sec. 6.16). These control various aspects of the microcontroller, such as the type of clock source. For programming individual 328P's on protoboards, the settings above should be used.

Use the text editor to make any modifications to the `Makefile` that are needed and make sure that when it's saved the OS didn't add a "`.txt`" or other extension to it.

6 Writing Code

Writing C code to be run on a microcontroller requires you to manage some of the low-level details of the system since a microcontroller has no operating system. Not only are you writing the application to be run on the system but also the essential portions of OS code. The low-level details you must manage include reading and writing internal registers, writing assembly routines, registering interrupt service routines, dealing with memory addressing issues, and some basic optimization issues.

6.1 Program Initialization

The avr-gcc software automatically includes in the project the start-up code to do things like initialize global variables and set up the processor stack. Other initialization tasks are determined by the programmable fuse settings. The programmer can just start their program and assume all the housekeeping tasks related to the microcontroller have been done.

6.2 Program Termination

Unlike writing a program on a larger computer, the microcontroller does not have an operating system that can take over control once the program is finished and exits. This means the program should never exit. It should always be doing something, such as running in an endless loop.

```
while (1) {
}
```

6.3 Declaring Variables

In most cases memory can be allocated the same as it would with any C program. However the microcontrollers have much less memory than is available for C programs on a general purpose computer so it is important to make efficient use it. Fixed-point variables, as opposed to floating point variables, should be used whenever possible since they take up less space in memory. The 328P processor does not have internal floating point hardware so using floating point can slow down execution speed and increase program size.

The native size for operating on data in the microcontroller is 8 bits. Whenever possible variables should be declared as 8-bit values in order to reduce both the amount of RAM memory used and the amount of code generated to operate on the variables. Unless a values being stored in a variable are known to exceed the limits of 8-bits, variables (and functions) should be declared as "`unsigned char`" for unsigned 0 to 255 values, or "`signed char`" for -128 to +127 values.

Using the smallest variable type necessary is also important for improving execution times in the program. Since the processor can only operate on 8-bit (single byte) quantities, whenever it has to add two or four bytes quantities this requires multiple instructions. Using a two or four byte variable for something like a

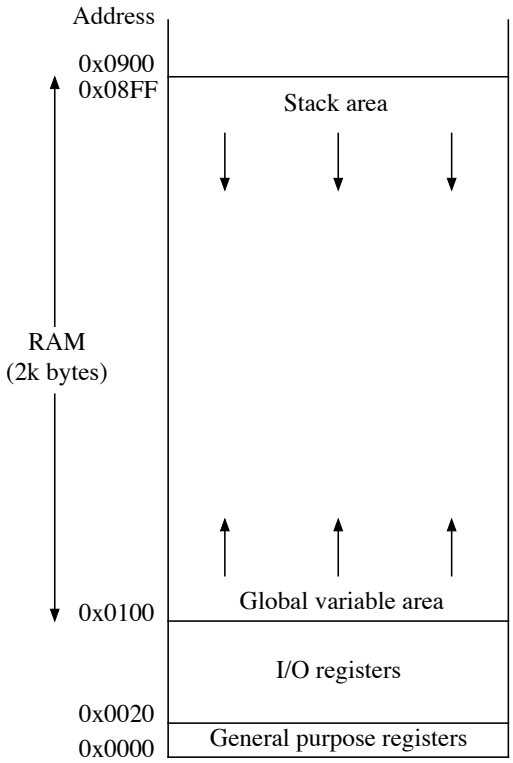

Figure 1: Memory Map of 328P RAM

loop index can cause the loop to run much slower than expected due to the amount of work that has to be done each time the index is incremented.

Table 1 summarizes the sizes and ranges of the fixed-point variables.

Traditional name	Portable name	# Bytes	Min	Max
`signed char`	`int8_t`	1	-128	+127
`unsigned char`	`uint8_t`	1	0	255
`signed int`	`int16_t`	2	-32768	32767
`unsigned int`	`uint16_t`	2	0	65535
`signed long`	`int32_t`	4	-2147483648	2147483647
`unsigned long`	`uint32_t`	4	0	4294967295

Table 1: Fixed-Point Variable Ranges

6.4 Using RAM Memory

RAM space on the 328P is very limited, only 2k bytes, and not all of it is available for use by your program. As shown in Fig. 1, the RAM occupies addresses from 0x0100 to 0x08FF in the addressing space.

The compiler assigns global variables to the RAM area starting at the bottom and growing upwards towards higher addresses. Stack space (see Sec 6.5) starts at the top of memory and grows downwards towards the global variable area.

6.5 Stack Space

The "stack" is a portion of memory used by the processor and programs to store temporary data. The amount of memory available for the stack is limited and misuse of the stack space on the 328P is one of the more common reasons that programs do not execute as expected. As can be seen from Fig. 1 the stack starts at location 0x08FF, and as data and addresses are pushed onto the stack, it grows downward towards the area where global variables are stored. If too many variables are pushed on the stack it is very likely that the stack will grow downward too much and eventually overwrite the contents of the global variables. Note that there is no mechanism in the 328P to prevent putting too much data on the stack. The stack data can overwrite global variables, and writing to global variables can overwrite stack data, either of which can cause big problems for the program.

When writing code, the programmer should be aware of roughly how much stack space they are using. Each time a function is called several things happen. First, all of the arguments to the function are pushed onto the stack. If a function has arguments of four 16-bit "`int`" values, this causes eight bytes to go on the stack. Next the processor pushes the two bytes for the return address on the stack. Lastly, space is allocated on the stack for all the local variables that are declared in the function. The combination of these actions can cause a significant amount of stack space to be consumed each time a function is called. The space is recovered when the function exits and control returns to the calling function. However if one function calls another, which calls another, etc., the amount of stack space used can grow beyond what is available.

When writing code, the following guidelines can help to avoid stack problems.

- **Avoid allocating large amounts of local variables.** For example, if your functions need a temporary buffer of 80 bytes in order to do some task, do not declare an array of 80 unsigned bytes in the function since these will all be allocated on the stack. Instead consider using a global variable array of 80 bytes, perhaps one that can be shared with some other functions that is not called at the same time. Even though good programming practice suggests not using global variables, the use of global variables is often the best way to prevent problems with the stack.

- **Avoid passing a data structure by value to a function.** If the "struct" is passed by value, then the calling routine puts all the values in the structure on the stack before jumping to the start of the function. A better way is probably to pass the structure by reference by passing a pointer to the structure rather than the values.

- **Do not write programs that use recursion.** Recursion is when a function calls itself and this practice can lead to excessive use of stack memory. If you think your program needs to operate recursively, try to find another way to implement the program.

6.6 Using Program Memory (ROM)

Constant values such as arrays containing character strings that will not be modified can be allocated in either RAM or ROM. Since RAM space is very limited, it's normally a good idea to allocate these in ROM. Due to the architecture of the Microchip AVR microcontrollers, this is not quite as easy as you would hope. The data to be stored in program memory has to be declared is a special way, and has to be accessed differently in the program in order to use the data.

The sample program `AT328-5.c` on the class web site contains an example of how to store a strings in program memory and then access it in the program. For more information, see the EE459 reference library web site for a link to a tutorial on using the program memory.

6.7 Using EEPROM Memory

The ATmega328P has 1024 bytes of EEPROM (Electrically Erasable Programmable Read-Only Memory) available for use by programs. EEPROM memory is similar to the ROM memory in that the data in it is retained (non-volatile) when the power to the chip is turned off, but the contents can be modified by the program similar to how RAM is used.

To use the EEPROM, a program must include the following preprocessor directive at the beginning of the code.

```
#include <avr/eeprom.h>
```

This declares a number of routines that can be called to read or write individual byte, word, long or float values, and additional routines to read or write blocks of data. For more information, see the EE459 reference library web site for a link to a tutorial on using the EEPROM memory.

EEPROM contents are erased each time the ATmega328P is reprogrammed unless one of the fuse bits is configured to have the contents retained. To have the data retained across programming operations, the "EESAVE" bit must be set to an zero. For more information see Sec. 6.16.

6.8 Determining Memory Usage

Using too much RAM memory is often the cause of problems with microcontroller software. To get an idea of how much RAM and ROM your program is using, you can examine the output produced each time the program is compiled and linked. The "`avr-size`" command is automatically run to print out some statistics on memory usage. For example, here is the output for the `AT328-4` sample program from the class web site.

```
% make
avr-gcc -Wall -Os -DF_CPU=9830400 -mmcu=atmega328p -c at328-4.c ...
avr-gcc -Wall -Os -DF_CPU=9830400 -mmcu=atmega328p -o main.elf ...
rm -f main.hex
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
avr-size --format=avr --mcu=atmega328p main.elf
AVR Memory Usage
----------------
Device: atmega328p

Program:     530 bytes (3.2% Full)
(.text + .data + .bootloader)

Data:         50 bytes (4.9% Full)
(.data + .bss + .noinit)
```

In this example, 530 bytes of program memory (ROM) are being used, and 50 bytes of RAM. Note that the RAM usage does not include the space that the stack will be using during program execution.

6.9 The Linker ".map" File

When your program successfully compiles and is linked together, the linker can create a map file that shows where all the procedures and variables have been allocated in ROM and RAM. The information in this file can be very useful for determining whether or not there are potential memory allocation and usage problems.

To create a `.map` file, locate the line in the `Makefile` that defines the command that will be used compile the program

```
COMPILE = avr-gcc -Wall -Os -DF_CPU=$(CLOCK) -mmcu=$(DEVICE)
```

and at the end of this line add the following options

```
-Wl,-Map=myprog.map -Wl,--cref
```

where "`myprog.map`" is replaced by whatever filename you want for the map file. The "`-Wl,...`" tells the compiler to pass the following option on to the linker during that phase of building the executable program.

Most of the .map file is useless gibberish but some useful information can be found there. Look for a line that starts with "`.text`" with no spaces before it.

```
.text           0x00000000       0x1a0
```

In the language of the linker the executable code for a program is called "text" and this line defines where in the program memory the program starts (`0x00000000`) and how big the text segment is (`0x1a0` = 416 bytes). Below that may be other entries showing the location of functions in the program, or ones that were added

by the linker to make it all work. The end of the text segment is indicated by a line with "etext" on it and in this example it says the end of the text is at address 0x1a0.

```
                0x000001a0                _etext = .
```

The next part of the map file is the "data" segment which is initialized global variables. Like the text segment part it also contains a lot of mostly useless information but some of it is shown below.

```
.data           0x00800100        0x10 load address 0x000001a0
                .
                .
                .
.data           0x00800100        0x10 at328-3.o
                0x00800100        digit_segs
                .
                .
                .
                0x00800110        _edata = .
```

This says there are 0x10 (16) global variables and that these are located starting at address 0x0100 in RAM (ignore the 0080 at the start of the address) and the data segment ends at address 0x0110.

The address shown for "edata" is probably the most important thing that can be found in map file. This tells the address of how high up in the memory the global variables extend. As discussed in Sec. 6.5, the stack memory grows downward toward the global variables so it's a good idea to be aware of how much space is being used.

6.10 Reading and Writing Internal Registers

All the internal registers in the microcontroller are defined by variables and can be read and written in the same manner as any other C variable. The names for the registers and individual register bits are defined when the line

```
#include <avr/io.h>
```

is placed at the beginning of the program. The names of the registers and the individual bits are shown in the ATmega328P manual.

For example, the input and output registers for ports B, C, and D are defined by variables named PINB, PINC and PIND, and PORTB, PORTC, and PORTD, respectively. You may perform operations on these variables just as you would any other.

```
PORTB = 0xf4;
PORTD = 5 * PINB + 2;
unsigned char my_var = PINC;
PORTC += 1;
```

The individual bits in the registers can also be accessed however the bits can not be read or written in the same way as the full registers. To write ones or zeros into bits, use the "|" (OR) and "&" (AND) operators to change the bits. For example, to set bit 3 in PORTB to a one, and clear bit 5 to a zero in PORTD do the following.

```
PORTB |= (1 << 3);
PORTD &= ~(1 << 5);
```

The bits in the device registers all have declared names in the "avr/io.h" file and these should be used for setting and clearing bits.

```
TCCR1B |= (1 << WGM12);
TIMSK1 |= (1 << OCIE1A);
```

Multiple bits can be set (or cleared) with one statement in the same manner.

```
TCCR1B |= ((1 << CS12) | (1 << CS11));
```

When writing individual bits in the registers it is important to be aware that this will cause the microcontroller to first read the full eight-bit byte, modify the byte, and then write the full byte back. Some of the internal functions of the microcontroller are affected by reading a register so it is important to be aware of any potential side effects when accessing the registers.

6.11 Floating Point

The avr-gcc compiler can generate code that does floating point calculations. This allows programs to be written in a style very similar to how it would be written for a larger computer. However programmers should keep in mind that the ATmega328P processor does not contain any floating point arithmetic hardware so all floating point calculations are implemented by doing multiple 8-bit integer arithmetic operations. The resulting code can be relatively slow to execute and the size of the program will also be larger due to the need to include routines in the final binary program that do the floating point operations. Incorporating even simple floating point operations (multiply, divide, compare, etc.) in a program can cause the program to grow by about 1900 bytes. This can be a problem if the program is close to filling out all the ROM space in the processor. Programs should not use floating point unless there are compelling reasons to do so. Whenever possible, integer arithmetic should be used instead.

Floating point variables can be declared as either "`float`" or "`double`" and the same code will be generated for both. The compiler only supports 32-bit floating-point numbers (4 bytes for each variable). Some library routines are defined with arguments declared as `double` so it is often easier to declare all floating point variables in a program as `double` to avoid having to do casts or get warning messages when calling library functions.

In some cases it is necessary to modify the project's "`Makefile`" to include different libraries that support the floating point routines. For example, if floating point values are to be formatted using the "`sprintf`" routine, the following must be added to the `COMPILE` definition in the `Makefile`

```
-Wl,-u,vfprintf -lprintf_flt
```

This tells the compiler to get rid of the default library code for `vfprintf` and to find another in the `printf_flt` library. Similar steps must be taken to use a floating point version of `sscanf`. Adding these libraries will also increase the size of the program as described in the next section.

6.12 Standard C Library

Avr-gcc includes a standard C library that contains many of the routines that programmers are used to having available to use in their programs. Routines like "`strlen`" and "`strcmp`" can be used the same way as on larger systems. Routines that can operate on floating point numbers like "`sqrt`" are also available but users should review the information in Sec. 6.11 before using them. Routines that would do I/O operations are not usable since the system does not have a file system or other I/O capability.

The avr-gcc linker will usually include any needed routines from the standard C library in the binary output when the program is compiled and linked. Programmers must keep in mind that these can significantly increase the size of their final executable program.

One of the more common library function used is "`sprintf`" for creating strings of formatted data. If this function is only used to format integer values and strings, the following line can be added to the `COMPILE` definition in the `Makefile` to reduce the size of the program.

```
-Wl,-u,vfprintf -lprintf_min
```

This tells the compiler to get rid of the default library code for `vfprintf` and to find another in the `printf_min` library that only supports formating of integers and strings. This can save about 400 bytes of program space. If floating point values are to be formated, see Sec. 6.11 for changes that must be made to the `Makefile` for this to work properly.

6.13 Including Assembly Language Code in C

You can write portions of your program in assembly language in two different ways. The first is by doing "inline assembly" and consists of inserting lines of assembly code directly in the C program file. While this method is relatively simple with some compilers, the AVR-gcc compiler has a rather complicated syntax that makes it difficult to use. For use in class projects, the inline assembler is not recommended.

The other method is to put the assembly code in separate "`.S`" files and assemble these into object files separate from the rest of the program. These object files can be linked with the ones produced by the compiler from C code files to form the final executable. Compared to inline assembly this is relatively easy to do. The C code in the "`.c`" file looks like normal C code, and the assembly code looks like a portion of a normal assembly language program. It does have the limitation that the assembly code has to comprise a complete function and can not be just a few lines in the middle of some C code.

The following is a summary of what has to be done with any registers that a function uses.

`r0` - Can be used by the assembler function. Does not have to be restored on return.

`r1` - Assumed be zero. If the function uses it, it must clear it to zero before returning.

`r2-r17, r28-r29` - Must be saved and later restored if used in the function.

`r18-r27, r30-r31` - Can be used freely. Function doesn't have to save and restore them.

6.14 Using Interrupts

If your program uses interrupts, either external one from the INT inputs or internal ones from the timer, ADC, etc., then you need to include the following at the beginning of the program.

```
#include <avr/interrupt.h>
```

In the program, global interrupts are enabled with the statement

```
sei();
```

and disabled with

```
cli();
```

Setting the global interrupt enable is not enough to enable interrupts for any of the modules. Each module has its own interrupt enable bit in one of the registers for the module and this bit must be set to the enable state for the module to generate an interrupt.

Any interrupt that is enabled must have a corresponding interrupt service routine (ISR) to handle the interrupt. The ISRs are written just like any other C function but in place of a function name and arguments, the ISRs contain the name of the interrupt vector. For example, an ISR for the "Timer1 Compare Match A" interrupt would look like this.

```
ISR(TIMER1_COMPA_vect)
{
    // ISR code
}
```

Table 2 lists the names of the interrupt vectors that should be used when writing interrupt service routines in C. Be advised that some of the C language vector names below are slightly different from the names of the vectors for assembly language programs. If in doubt check the contents of the `interrupts.h` file where the functions are defined.

6.15 Volatile Variables

Global variables can have their values changed and/or checked from any place in the program, including within interrupt service routines. This implies that a variable's value can change during execution of one part of the program even though there are no assignment statements affecting it at that point in the program.

Interrupt definition	Vector name
External Interrupt Request 0	INT0_vect
External Interrupt Request 1	INT1_vect
Pin Change Interrupt Request 0	PCINT0_vect
Pin Change Interrupt Request 1	PCINT1_vect
Pin Change Interrupt Request 2	PCINT2_vect
Watchdog Time-out Interrupt	WDT_vect
Timer/Counter2 Compare Match A	TIMER2_COMPA_vect
Timer/Counter2 Compare Match B	TIMER2_COMPB_vect
Timer/Counter2 Overflow	TIMER2_OVF_vect
Timer/Counter1 Capture Event	TIMER1_CAPT_vect
Timer/Counter1 Compare Match A	TIMER1_COMPA_vect
Timer/Counter1 Compare Match B	TIMER1_COMPB_vect
Timer/Counter1 Overflow	TIMER1_OVF_vect
Timer/Counter0 Compare Match A	TIMER0_COMPA_vect
Timer/Counter0 Compare Match B	TIMER0_COMPB_vect
Timer/Counter0 Overflow	TIMER0_OVF_vect
SPI Serial Transfer Complete	SPI_STC_vect
USART Rx Complete	USART_RX_vect
USART Data Register Empty	USART_UDRE_vect
USART Tx Complete	USART_TX_vect
ADC Conversion Complete	ADC_vect
EEPROM Ready	EE_READY_vect
Analog Comparator	ANALOG_COMP_vect
Two-wire Serial Interface	TWI_vect
Store Program Memory Read	SPM_READY_vect

Table 2: Interrupt Vector Names

If an interrupt occurs, the variable can be changed and when execution continues in the program it will have a different value than it did before. This can be a problem if the compiler removes what it feels are unnecessary references to a variable in order to optimize the code. For example, in the following code the function "`check()`" preforms a loop waiting for the variable "`flag`" to become non-zero. The variable is not changed in `check()` but is changed in the ISR. Once an interrupt occurs and the ISR sets the variable, the function then does something.

```
uint8_t flag;

check()
{
    while (flag == 0);

    \\ Do something
}

ISR(SOME_INTERRUPT_vect)
{
    flag = 1;
}
```

The problem occurs because the compiler will see that the value of "`flag`" is never changed in the function so it will simplify the code by replacing it with something like this.

```
check()
{
```

```
    if (flag == 0)
        while (1);
    else {
        \\ Do something
    }
}
```

The above code only checks the value of `flag` once and then goes into an infinite loop if the value is zero. As the compiler sees it, there is no reason to check the value of `flag` each time through the loop since it isn't being changed. To prevent this from happening, all global variables that can be modified outside the normal flow of the program, such as in an ISR, must include the "`volatile`" keyword in the declaration.

```
volatile uint8_t flag;
```

The `volatile` keyword tells the compiler that the variable's value can change and that it should never assume that the value that was in the last time it used it will be the same the next time. With the `flag` variable declared with the `volatile` keyword, the compiler will generate the correct code for this situation.

6.16 Fuse Settings

Some of the operation of the ATmega328P is determined by three bytes that can be programmed by the user. These are referred to as the fuse bytes and are separate from the program memory. In most cases these can be programmed once when a new ATmega328P is installed in the project board and do not have to be reprogrammed each time the firmware is changed in the chip.

The fuses for the 328P contain the bits shown it the Table 3. The default settings are fine for most of the things that can be configured. The only changes that are usually made are to a couple of bits in the low fuse byte to make the chip use an external clock source such as a TTL oscillator and to not divide the clock by eight. All the other settings can be left unchanged from the default settings. These settings result in the high fuse byte value of `0xd9` and the low byte value of `0xe0`. These values are put in the Makefile on the `FUSES` line.

7 Building your Application

You can compile and link your project by entering the command "`make`". This will compile any source files that have changed since the last time the command was run, and then link it all together. If the compiling and linking are successful, a file named "`main.hex`" will be created in the project folder containing the binary data that will be loaded into the microcontroller.

When the compiler runs to compile your code it will print out any error or warning messages. Make sure to read through these to see if something has to be fixed. Don't assume that just because the compilation finished that all went correctly. In many cases the compiler will build the `main.hex` file even though there were errors in the process. Students often waste hours or days trying to debug a program simply because they didn't read a message generated during the compiling process that tells them where the problem is. Warning messages should not be ignored even though those are usually for non-fatal problems in the code. Make every effort to fix the issues in the code that cause these so the messages are not generated. If a program is compiled and generates a lot of warnings, these can make it difficult to spot other messages that may be reporting more important issues.

8 Programming the ATmega328P

Once the program has been compiled and linked, the binary data can be downloaded to the 328P using one of the USBtiny programming modules available in the lab. These connect to a USB port on the computer and have a six-conductor ribbon cable connector that mates with a 2×3 pin header on the project board. The basic programming sequence is as follows.

High Fuse Byte				
Bit	Name	Description	Default	Recommended Value = `0xd9`
7	RSTDISBL	External Reset Disable	1	1 ⇒ Reset enabled
6	DWEN	debugWIRE Enable	1	1 ⇒ debugWIRE disabled
5	SPIEN	Enable Serial Program Downloading	0	0 ⇒ SPI enabled
4	WDTON	Watchdog Timer On	1	1 ⇒ Watchdog timer off
3	EESAVE	EEPROM memory preserved	1	1 ⇒ Don't preserve EEPROM
2	BOOTSZ1	Select Boot Size	0	0 ⇒ 2048 words
1	BOOTSZ0		0	0
0	BOOTRST	Select Reset Vector	1	1

Low Fuse Byte				
Bit	Name	Description	Default	Recommended Value = `0xe0`
7	CKDIV8	Divide Clock by 8	0	1 ⇒ Don't divide by 8
6	CKOUT	Clock output	1	1 ⇒ Don't ouput clock on PB0
5	SUT1	Select start-up time	1	1 ⇒ 14 clocks + 65ms
4	SUT0		0	0
3	CKSEL3	Select clock source	0	0 ⇒ External clock
2	CKSEL2		0	0
1	CKSEL1		1	0
0	CKSEL0		0	0

Extended Fuse Byte				
Bit	Name	Description	Default	Recommended Value = `0xff`
7			1	1
6			1	1
5			1	1
4			1	1
3			1	1
2	BODLEVEL2	Brown-Out Detector Level	1	1 ⇒ BOD disabled
1	BODLEVEL1		1	1
0	BODLEVEL0		1	1

Table 3: ATmega328P fuse bits

1. Connect the programmer's USB cable to one of the computer's USB ports. A red LED inside the programmer should light up.

2. Connect the programmer to the 2×3 pin header on the project board. The USBtiny programmers provide +5 power to the board being programmed so you may not need to connect the board's power wires to the power supply in order for the board to operate.

3. Enter the command "`make flash`". The `avrdude` software will now run and download the data into the 328P program memory.

4. For new chips or if the fuse setting have changed, enter the command "`make fuse`". This will cause the fuse data to be programmed into the 328P. If the "`make fuse`" operation was done previously for this microcontroller chip, and you aren't trying to change the fuse settings, you can skip this step.

At this point the microcontroller will reset and start running the new program. The programmer can be left connected to the computer and the project board while the operation of the board is being observed

and changes are made to the program source files. After the changed program has been compiled and linked and is ready to be downloaded, only step 3 above (`make flash`) has to be done.

Important: If the programmer is attached to your project board it must also be connected to the computer in order for the program on the project board to run properly. If the programmer is not connected to the computer so it is not powered up, it may prevent the program in the microcontroller from running.

8.1 Programming Problems

Several things can go wrong that will prevent the microcontroller from being programmed. If the `make flash` command results in the error message

```
avrdude: Initialization failed, rc=-1
```

this means the programmer can not communicate with the microcontroller. This can be for any of the number of reasons.

- Microcontroller is not plugged securely into the socket.
- Clock signal not getting to the microcontroller (check with oscilloscope.)
- +5 volt power not present at the microcontroller (check with oscilloscope.)
- Wiring errors between the 6 pin header and the microcontroller pins.
- Microcontroller has become defective.

If the micro was programming fine and then this error started appearing, one of the first three listed above is probably the cause.

9 Sample Programs

The following files are available on the EE 459 class web site and may be of interest to students writing C programs for the ATmega328P.

at328-0.c A very simple program for showing the micro is working. It loops forever turning bit zero in port C (PC0) on and off as rapidly as possible.

at328-1.c This program reads a switch input and turns an LED on and off.

at328-2.c This program counts up and down on a seven-segment display. The program uses a library function to implement the counter delay. See Appendix A for a listing.

at328-3.c Similar to `at328-2.c` but uses an internal timer and interrupts to implement the delay. See Appendix B for a listing.

at328-4.c Demonstrates interfacing to an LCD display using an 8-bit interface. Puts up a short message on the display.

at328-5.c Similar to `at328-4.c` but uses a 4-bit interface to the LCD and stores the strings in the ROM memory instead of RAM.

at328-6.c This program demonstrates using an RS-232 serial interface to control an LCD display.

at328-7.c Examples of reading and writing data to an EEPROM using an IIC (I^2C) bus.

Appendix A: Sample program at328-2.c

```
/************************************************************
*       at328-2.c - Demonstrate simple I/O functions of ATmega328P
*
*       This program will cause a 7-segment display to either count up in
*       hexadecimal (0,1,2,...,E,F,0,1,...) or count down in decimal
*       (9,8,...,1,0,9,8,..) depending on whether or not a switch is
*       pressed.
*
*       Port C, bit 1 - input from switch (0 = pressed, 1 = not pressed)
*               When the switch is not pressed, the 7-segment display
*               counts up in hexadecimal. When the switch is pressed,
*               the 7-segment display counts down in decimal.
*       Port B, bits 0-1 and Port D, bits 2-6 - Outputs to data inputs of
*               the 74LS374 register.
*               Bit 6 -> segment A, 5->B, ... , 1->F, 0->G
*               A low output bit will cause the LED segment to light up.
*       Port C, bit 2 - Output to positive edge-triggered clock input
*               of 74LS374 register.
*
************************************************************/

#include <avr/io.h>
#include <util/delay.h>

unsigned char digit_segs[16] = {
    0x7e,0x30,0x6d,0x79,0x33,0x5b,0x5f,0x70,
    0x7f,0x73,0x77,0x1f,0x4e,0x3d,0x4f,0x47 };

void display_digit(unsigned char);

#define SWITCH         (1 << PC1)
#define SEG_DATA_B     0x03     // Bits in Port B for LED display
#define SEG_DATA_D     0xfc     // Bits in Port D for LED display
#define SEG_CLOCK      (1 << PC2)

int main(void) {
    unsigned char up;
    unsigned char cnt = 0;

    PORTC |= SWITCH;            // Turn on pull-up on PC1 for switch

    DDRC |= SEG_CLOCK;          // Set PORTC bit 2 for output
    DDRB |= SEG_DATA_B;         // Set PORTB bits 0-1 for output
    DDRD |= SEG_DATA_D;         // Set PORTD bits 2-6 for output

    while(1) {                  // Loop forever
        display_digit(cnt);

        _delay_ms(500);         // wait 500 ms

        up = PINC & SWITCH;     // read the button
        if (up) {               // if button is not pressed, up = 1
```

```
            if (++cnt > 15)     // and we count up in hex
                cnt = 0;
        }
        else {                          // if button is pressed, up = 0
            if (--cnt < 0 || cnt > 9)   // and we count down in decimal
                cnt = 9;
        }
    }
    return 0;   /* never reached */
}

void display_digit(unsigned char digit)
{
    unsigned char x;
    x = digit_segs[digit] ^ 0xff;  // invert bits (active low outputs)
    PORTB |= x & SEG_DATA_B;    // put low two bits in B
    PORTB &= (x | ~SEG_DATA_B);
    PORTD |= x & SEG_DATA_D;    // put high five bits in D
    PORTD &= (x | ~SEG_DATA_D);
    PORTC |= SEG_CLOCK;         // toggle the clock bit to 1
    PORTC &= ~SEG_CLOCK;        // toggle the clock bit to 0
}
```

Appendix B: Sample program at328-3.c

```
/************************************************************
*       at328-3.c - Demonstrate simple I/O functions of ATmega328P
*           and the use of an internal timer and interrupt
*
*       This program will cause a 7-segment display to either count up in
*       hexadecimal (0,1,2,...,E,F,0,1,...) or count down in decimal
*       (9,8,...,1,0,9,8,..) depending on whether or not a switch is
*       pressed.
*
*       Port C, bit 1 - input from switch (0 = pressed, 1 = not pressed)
*               When the switch is not pressed, the 7-segment display
*               counts up in hexadecimal. When the switch is pressed,
*               the 7-segment display counts down in decimal.
*       Port B, bits 0-1 and Port D, bits 2-6 - Outputs to data inputs of
*               the 74LS374 register.
*               Bit 6 -> segment A, 5->B, ... , 1->F, 0->G
*               A low output bit will cause the LED segment to light up.
*       Port C, bit 2 - Output to positive edge-triggered clock input
*               of 74LS374 register.
*
************************************************************/

#include <avr/io.h>
#include <avr/interrupt.h>

uint8_t digit_segs[16] = {
    0x7e,0x30,0x6d,0x79,0x33,0x5b,0x5f,0x70,
    0x7f,0x73,0x77,0x1f,0x4e,0x3d,0x4f,0x47 };
uint8_t cnt;

void display_digit(uint8_t);

#define SWITCH         (1 << PC1)
#define SEG_DATA_B     0x03     // Bits in Port B for LED display
#define SEG_DATA_D     0xfc     // Bits in Port D for LED display
#define SEG_CLOCK      (1 << PC2)

int main(void) {

    PORTC |= SWITCH;            // Turn on pull-up on PC1 for switch

    DDRC |= SEG_CLOCK;          // Set PORTC bit 2 for output
    DDRB |= SEG_DATA_B;         // Set PORTB bits 0-1 for output
    DDRD |= SEG_DATA_D;         // Set PORTD bits 2-6 for output

    /*
       The demo board has a 9.8304 MHz clock.  We want the timer to
       interrupt every half second (2 Hz) so we need to count clocks to
       9.8304MHz/2Hz = 4,915,200.  This is too big for the 16 bit counter
       register so use the prescaler to divide the clock by 256 and then
       count that clock to 19,200.
    */
```

```
    // Reset clears register bits to zero so only set the 1's
    TCCR1B |= (1 << WGM12);     // Set for CTC mode.  OCR1A = modulus
    TIMSK1 |= (1 << OCIE1A);    // Enable CTC interrupt
    sei();                      // Enable global interrupts
    OCR1A = 19200;              // Set the counter modulus
    TCCR1B |= (1 << CS12);      // Set prescaler for divide by 256,
                                // also starts timer

    display_digit(cnt);

    while(1) {                  // Loop forever while interrupts occur
    }

    return 0;   /* never reached */
}

void display_digit(uint8_t digit)
{
    uint8_t x;
    x = digit_segs[digit] ^ 0xff;  // invert bits (active low outputs)
    PORTB |= x & SEG_DATA_B;    // put low two bits in B
    PORTB &= (x | ~SEG_DATA_B);
    PORTD |= x & SEG_DATA_D;    // put high five bits in D
    PORTD &= (x | ~SEG_DATA_D);
    PORTC |= SEG_CLOCK;         // toggle the clock bit to 1
    PORTC &= ~SEG_CLOCK;        // toggle the clock bit to 0
}

ISR(TIMER1_COMPA_vect)
{
    uint8_t up;

    up = PINC & SWITCH;         // read the button
    if (up) {                   // if button is not pressed, up = 1
        if (++cnt > 15)         // and we count up in hex
            cnt = 0;
    }
    else {                      // if button is pressed, up = 0
        if (--cnt < 0 || cnt > 9)   // and we count down in decimal
            cnt = 9;
    }

    display_digit(cnt);
}
```