Programming in C with the ATMega2560

CSC230 Lab 09 - Fall 2018

Although we have spent most of the semester writing assembly code for the ATmega2560, the majority of AVR programming in practice is done in C, and there is a fully featured library of C bindings available for the ATmega2560, which allows all of the features of the board (Ports, Timers, the LCD screen, etc.) to be used with C code.

Atmel Studio and C

To write C code with Atmel Studio, create a new project and select **GCC C Executable** as the project type (and configure all other aspects as for an assembly-based project). When the project is created, it will contain an empty .c file for C code.

A minimal example

Open the file in the resources section called **minimal_c.c**. Read over this program and note the following features of C:

- All the C programs we write for the ATMega2560 in this lab require us to include a file of definitions for our hardware and setup. This file is called **CSC230.h** and can be found in the Lab09 resources section of the website. Put it in the same directory as your main.c file, and include it by writing:

  ```
  #include CSC230.h
  ```

  at the beginning of main.c. Note that there is no semicolon ending this line.

- There are two functions in this program, **do_something()**, which is blank, and **main()**. Every C project has one (and only one) main() function this is where processing will begin when youre program runs. Note that there is no 'Set Entry File' option in a C project, and that functions do not have semicolons after their closing brace.

- Pretty much everything else needs a semicolon at the end.

- In C, we dont need to use registers or LD, LDS or LDI commands we instead can simply declare variables and assign them values as shown.

- We mainly use two data types in C, char and int. char may contain an 8-bit value, and int a 16-bit value. There are unsigned and signed versions of each of these data types.

- Memory locations in all uppercase represent the corresponding locations from AVR assembly programming, and can be treated as global variables. For example, instead of writing:

  LDI r16, 2

  STS PORTL, r16

  We can simply write:

  PORTL = 2;

Functions and parameters

We can pass parameters to functions and return values as we would in other high-level languages:

```
int add_two(int value_a, int value_b) {
    return value_a + value_b;
}
```

We can call this function by passing numbers or variables to it:

```
sum = add_two(457, 2783);
```

Program control

Instead of condition checking using CP, CPI and the branch family of commands, we can write conditionals using if...then constructions:

```
if (some_input == 'A') {
    value = 65;
} else if (some_input == 'B') {
    value = 66;
} else {
    value = 0
}
```

We use the **WHILE** keyword for infinite loops, and the **FOR** keyword for finite loops:

```
while (1) {
    for(int i = 0; i < 10; i++) {
        do_something(i);
    }
}
```

These nested loops will call the function **do_something()** with the numbers from zero to nine, and then repeat starting at zero again, forever. Remember we never want a program running on the ATMega2560 to end - we should always have an infinite loop either as our main program loop, or at the end of our program.

First Program - the LEDs and delay

Copy and paste the contents of the file **LCD_pattern.c** into your **main.c** file. This program lights the LEDs in order from bottom to top, and pauses for a quarter second between each. Read over the program, and note the use of two arrays:

```
unsigned char portb_pattern[] = { 2, 8, 0, 0,  0,   0};
unsigned char portl_pattern[] = { 0, 0, 2, 8, 32, 128};
```

Each array is a sequence of six one-byte values, holding a bit pattern for the LEDs for that particular port. During our main loop, we select one value from each of the two arrays and send them to the ports:

```
for (int i = 0; i < 6; i++) {
    PORTB = portb_pattern[i];
    PORTL = portl_pattern[i];
    _delay_ms(250);
}
```

The function called **_delay_ms()** has been provided for you; it implements a busy-wait loop whose duration is measured in milliseconds.

Try building and uploading this file - the process is the same as the one you used for assembler programs (build the program, and run **upload.bat** from the Debug subfolder).

The LCD

To use the LCD we need to add the file **CSC230_LCD.c** to our project - you will find it in the Lab09 folder.

Examine the provided program called **lab09_lcd_demo.c**. Note the function calls - our LCD library implements the same functions we had in assembler: **lcd_init()**, **lcd_puts()**, and **lcd_xy()**. Note also the use of a 'string' - C doesn't actually have strings; instead we use another array of type char, this time initialized with double quotes for clarity. Note that we can pass a sequence of characters directly, or save that sequence in an array, and pass the variable name to **lcd_puts()**.

Build and upload this program, and try changing the characters displayed, as well as their locations.

String Manipulation

You are welcome to use parts of the C standard library in your C code, including the string functions provided in **string.h** (see the list of C string functions on Wikipedia for more details). The **lab09_count.c** file contains a short program which uses the strcpy function to copy a constant string into an array, then modifies an element of the array at regular intervals to display a one-second counter on the LCD screen. In order to use the string functions in your code, include them into your text file:

```
#include <string.h>
```

The sprintf function

The printf function is not useable in AVR code because there is no "standard output" stream for printf to write its result. However, one of the variants of printf, called sprintf, allows the result of printf to be written to a string instead of an output stream. The sprintf function is available in **string.h**, can make the process of construction formatted strings extremely easy.

The file **lab09_count_sprintf.c** uses the sprintf function to display a simple one-second counter on the LCD screen using the _delay_ms() function to keep time.

The ADC and black buttons

The **lab09_show_adc_result.c** file contains code to poll the black buttons using the ADC. The program contains a main loop which polls the buttons at one second intervals, then displays the result in hex on the LCD screen.

The code contains a set of #define statements to create named constants for the various button values. You may want to use these in your code (but you may need to change them for the "v1.1" AVR boards).

```
#define ADC_BTN_RIGHT 0x032
#define ADC_BTN_UP 0x0C3
#define ADC_BTN_DOWN 0x17C
#define ADC_BTN_LEFT 0x22B
#define ADC_BTN_SELECT 0x316
```

The 16 bit ADC result is returned as a single short value from the poll_adc function, so you can test button values using a single equality comparison. For example, to test if the UP button is pressed, you might use code like the following.

```
short adc_result = poll_adc();
if (adc_result >= ADC_BTN_RIGHT && adc_result < ADC_BTN_UP){
  //Up button pressed
}
```

Advanced: Interrupts

We can use interrupts in our C code, with a few minor changes:

- We use a pre-defined function called **sei()** rather than the sei command to set the interrupt flag.
- We don't need to create an interrupt vector or worry about register protection or the SREG.
- Our interrupt handler must be declared using the ISR macro, which takes as a parameter a specific, predefined name (listed on the next page by vector number). For example, the handler for Timer1 overflow, which we covered in the last lab, would be a function declared like this:

  ```
  ISR(TIMER1_OVF_vect){
      // something you want to do every time Timer1 overflows.
  }
  ```

The file **lab09_blink_.isr.c** uses the 8-bit TIMER0 to blink an LED using interrupts; consult it for an example of using timer interrupts in C.

Challenge: Starting with **LCD_pattern.c**, modify the code to make the LED pattern scroll up, then back down, in an endless loop.

Challenge: Modify **lab09_lcd_demo.c**. Create a 16-character string containing whatever you want, then display it to the LCD with a short delay between each character. For example, if my string is 'Assembler is fun', it would be displayed first as just the letter A on the first line, then As and so on. Once the line is full, blank it and start again.

This document contains material written by Bill Bird as part of CSC230 Lab9 Summer 2018.

Vector number	Handler name	Vector number	Handler name
1	INT0_vect	28	ANALOG_COMP_vect
2	INT1_vect	29	ADC_vect
3	INT2_vect	30	EE_READY_vect
4	INT3_vect	31	TIMER3_CAPT_vect
5	INT4_vect	32	TIMER3_COMPA_vect
6	INT5_vect	33	TIMER3_COMPB_vect
7	INT6_vect	34	TIMER3_COMPC_vect
8	INT7_vect	35	TIMER3_OVF_vect
9	PCINT0_vect	36	USART1_RX_vect
10	PCINT1_vect	37	USART1_UDRE_vect
11	PCINT2_vect	38	USART1_TX_vect
12	WDT_vect	39	TWI_vect
13	TIMER2_COMPA_vect	40	SPM_READY_vect
14	TIMER2_COMPB_vect	41	TIMER4_CAPT_vect
15	TIMER2_OVF_vect	42	TIMER4_COMPA_vect
16	TIMER1_CAPT_vect	43	TIMER4_COMPB_vect
17	TIMER1_COMPA_vect	44	TIMER4_COMPC_vect
18	TIMER1_COMPB_vect	45	TIMER4_OVF_vect
19	TIMER1_COMPC_vect	46	TIMER5_CAPT_vect
20	TIMER1_OVF_vect	47	TIMER5_COMPA_vect
21	TIMER0_COMPA_vect	48	TIMER5_COMPB_vect
22	TIMER0_COMPB_vect	49	TIMER5_COMPC_vect
23	TIMER0_OVF_vect	50	TIMER5_OVF_vect
24	SPI_STC_vect	51	USART2_RX_vect
25	USART0_RX_vect	52	USART2_UDRE_vect
26	USART0_UDRE_vect	53	USART2_TX_vect
27	USART0_TX_vect	54	USART3_RX_vect
28	ANALOG_COMP_vect	55	USART3_UDRE_vect
		56	USART3_TX_vect