

Writing Code for Microcontrollers

A rough outline for your program

The first few lines are comments that describe the file, its purpose, author, etc...

```
//file      : mylab2.c
//author    : R. Traylor
//date      : 7.28.05
//modified  : 7.29.05
//This code implements lab 2 for ECE473, fall 2005
```

Also included at the beginning is how the hardware is to be connected. Embedded code is intimately connected to the hardware. If you don't specification hardware connections, the code cannot be debugged efficiently.

```
//  HARDWARE SETUP:
//  PORTA is connected to the shared segments of the LED display.
//  PORTA.0 corresponds to seg a, PORTA.1 corresponds to seg b, etc.
//  PORTB bits 4-7 correspond to LED digits 0-3.
//  Switch 0: toggle function, amount to increment or decrement count.
//  Switch 1: toggle function, selects which encoder (0,1) changes count.
//  Switch 2: toggle function, sets display to bright or dim.
//  Encoder pinout:
//  encoder 0, A = PORTE.3
//             B = PORTE.2
//  encoder 1, A = PORTE.5
//             B = PORTE.4
```

Writing Code for Microcontrollers

A rough outline for your program

Next are the `#include` files.

- These tell the compiler where to look for code you are using but did not include in this file.
- Before compilation, the compiler includes the necessary header files to be compiled with the *.c file. Its is as if the header files were copied into the *.c file.

```
#include <avr/io.h>
#include <avr/delay.h>
#include "lcd.h"           //holds function prototypes for lcd.c
```

A good practice to use *function prototypes*. These are a declaration of the function that omits the function body but only specifies the function's name, argument types and return type. Declaring them all in a header file and including that file allows you to use them in any order without the compiler complaining.

Writing Code for Microcontrollers

A rough outline for your program

Then come the **#defines**.

-These are also called “macros”. Before compilation the argument of the #define is substituted everywhere the name is used. Its good practice to use all uppercase for #defines.

```
#define DELAY_COUNT 4000 // CPU speed div by 4000
#define TRUE 0x01 // logical TRUE
#define FALSE 0x00 // logical FALSE

#define MIN(A,B) ((A)<(B)) ? (A) : (B)
#define MAX(A,B) ((A)>(B)) ? (A) : (B)
```

Writing Code for Microcontrollers

A rough outline for your program

Next is the main section of code. You always have this.

-Just inside `main()` I do register setup, any initialization code such as `spi_init()` or `lcd_init()` and give some idea of the program flow.

-Note: `while(1) { }` typical in embedded applications.

```
int main()
{
DDRA = 0xFF; //set port A to all outputs
 DDRB = 0xF0; //set port bits 4-7 B as outputs
 DDRD = 0x00; //set port D to all inputs
 DDRE = 0x00; //set port E to all inputs
PORTE= 0xFF; //set port E to all pullups
PORTB= 0x00; //set port B to all zeros
PORTA= 0x00; //set port A to all zeros

//main while loop of program follows
// -checks encoders
// -increment/decrement count if something changed
// -display next digit
// -check increment/decrement amount and encoder to check

while(1){
    _delay_loop_2(500); //loop debounce required
```

Writing Code for Microcontrollers

A rough outline for your program

Next are **function calls** or **procedures**.

-For each function or procedure, clearly define how it operates.

```
/******  
//  
//          bound_count  
// Keeps the count from going below zero and above 999.  
// Takes a 16 bit signed int and returns the same.  
/******  
int16_t bound_count(int16_t in_value) {  
    if(in_value < 0)    return 0;  
    if(in_value > 999) return 999;  
    return in_value;  
} //bound_count
```

Writing Code for Microcontrollers

A rough outline for your program

Clear commenting cannot be stressed enough! For example.....

```
/******  
//                               encoder_chk  
// Takes an argument (either 0 or 1)of the encoder to check.  If the  
// encoder is moved, the function returns:  
//           1 if  CW rotation  detected,  
//           0 if  CCW rotation detected  
//           -1 if  no movement  detected.  
// Note: the return value is a _signed_ 8-bit int.  
// Expected pinout:  
//  encoder 0, A output = PORTE.3  
//             B output = PORTE.2  
//  encoder 1, A output = PORTE.5  
//             B output = PORTE.4  
// Port E is expected to be pulled up. Encoder causes switch  
// closure to ground through a 1K resistor.  
// Debounce time is 12 times each ISR run or loop time.  
// Code was adapted from Ganssel's "Guide to Debouncing"  
/******
```

Note comment style. Don't "box areas with /* */. Causes too much re-editing.

Writing Code for Microcontrollers

Programming on-board peripherals

When programming control registers for on-board peripherals, use the most appropriate form to maintain readability and portability. For instance:

Timer/Counter Control Register – TCCR0		7	6	5	4	3	2	1	0									
Bit		<table border="1"><tr><td>FOC0</td><td>WGM00</td><td>COM01</td><td>COM00</td><td>WGM01</td><td>CS02</td><td>CS01</td><td>CS00</td></tr></table>								FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00											
Read/Write		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									

To set this register up, this would be most clear:

```
TCCR0 = (1<<FOC0) | (1<<WGM00) | (1<<CS00);
```

TCCR0 would be loaded with `0b1100_0001;`

This form is most likely to work across multiple models of the AVR architecture.

Writing Code for Microcontrollers

Programming on-board peripherals

On the other hand, some control registers are best setup with a hex value. For instance:

Output Compare Register – OCR0		7	6	5	4	3	2	1	0	
Bit		OCR0[7:0]								OCR0
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	

The Output Compare Register contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0 pin.

To set this register up, this would be most clear:

```
OCR0 = 0x53; // OCR0 match is at 0x53
```

Writing Code for Microcontrollers

Programming on-board peripherals

Remember that to only set or reset certain bits, use the “|=“ or “&=“ format. This cannot be stressed enough. Ignore this at your own peril.

For example, to clear the output compare flag zero (OCF0) interrupt for counter timer 0 for we need to write a one to bit one of the TIFR register and **not touch** bit zero.

Timer/Counter Interrupt Flag Register – TIFR		Bit	7	6	5	4	3	2	1	0	
			OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

If we simply wrote a 0x02 to TIFR, a pending interrupt in the TOV0 bit would be cleared and lost. So instead we do this:

```
TIFR |= (1<<OCF0); // clear OCF0 interrupt
```

and not this:

```
TIFR = (1<<OCF0); //clear OCF0 interrupt(OOPS!)
```

Writing Code for Microcontrollers

Programming on-board peripherals

By using the predefined names in io.h, code is kept portable. For example:

```
// from code for a mega48
void spi_init(void){
    DDRB |= (1 << PB2) | (1 << PB3) | (1 << PB5); //Turn on SS, MOSI, SCLK
    SPCR=(1<<SPE) | (1<<MSTR); //enbl SPI, MSB 1st, init clk as low
    SPSR=(1<<SPI2X);           //SPI at 2x speed (0.5 MHz)
} //spi_init

//from code for a mega128
void spi_init(void){
    DDRB |= (1<<PB2) | (1<<PB1) | (1<<PB0); //Turn on SS, MOSI, SCLK
    SPCR=(1<<SPE) | (1<<MSTR); //enbl SPI, clk low init, rising edge sample
    SPSR=(1<<SPI2X);           //SPI at 2x speed (8 MHz)
} //spi_init
```

Only the pin definitions for ports differ.

Why does this work?

Writing Code for Microcontrollers

Programming on-board peripherals

from iom128.h in (.../avr/avr/include/avr)

```
/* ATmega128 SPI Control Register */
#define SPCR          _SFR_IO8(0x0D)
/* SPI Control Register - SPCR */
#define SPIE          7
#define SPE           6
#define DORD          5
#define MSTR          4
#define CPOL          3
#define CPHA          2
#define SPR1          1
#define SPR0          0
```

```
from iomx8.h (included from iom48.h)
also in (.../avr/avr/include/avr)
#define SPCR          _SFR_IO8 (0x2C)
/* SPCR */
#define SPIE          7
#define SPE           6
#define DORD          5
#define MSTR          4
#define CPOL          3
#define CPHA          2
#define SPR1          1
#define SPR0          0
```

Although the SPCR register is at a different address and the control register bits may have been at different positions, as long as these definitions are here, the code is portable across any “Mega” type processor.

Writing Code for Microcontrollers

Other Programming Perls

Comment as you code; not as a separate activity. Keep comments with the code they document. Update the comments with the code.

Leave debugging code in the source file. Comment out debugging code but leave it in place. Maybe use `#ifdef` statements for debug code.

Initialize each peripheral unit within its own function call; **nowhere else**.

Avoid cleverness. Make your intentions clear to someone else who may end up reading your code.

Put all your `#defines` in one place. Preferably in an header file.

Writing Code for Microcontrollers

Programming Perls – last words

These programs are different than any you have written before. Sloppiness will be rewarded with grief. You will have multiple asynchronous real-time events occurring that you must handle within fixed time frames or it's broken. Its not easy to do.

When you can't find a bug that you've been working on for an hour or so, print out your code, shut your laptop, and walk away. Go for a short walk to the coffee shop. Go by yourself. Let the right-brain have a go at things. Let go, take a deep breath. You are about to become productive.

Get a cup of coffee and read your code. Walk all the way through it. Several times. Red line anything that is even slightly out of sorts. Then go back, edit and recompile. You will be surprised at how many bugs you've found and fixed.

All this may take you one hour. I bet you will find its the best spent hour of the day. Too often we get stuck in the frenzied “edit, recompile, test” loop. You have confused motion with action. Pull out of the the spin. It works.

Writing Code for Microcontrollers

Programming quotations

Programming can be fun, so can cryptography; however, they should not be confused.

-Charles Kreitzberg and Ben Shneiderman

The sooner you start to code, the longer the program will take.

-Roy Carls

The most important single aspect of software development is to be clear about what you are trying to build.

-Bjarne Stroustrup

If the code and comments disagree, both are probably wrong.

-Norm Schryer

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

-Damian Conway

Documentation is a love letter that you write to your future self.

-Damian Conway

Writing Code for Microcontrollers

Programming quotations

"C is quirky, flawed, and an enormous success." - Dennis Ritchie

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” – Brian Kernighan

-See material on managing large projects at:

<http://www.fourwalledcubicle.com/AVRArticles.php>

Writing Code for Microcontrollers

Programming quotations

Refactoring

"An architect's most useful tools are an eraser at the drafting board, and a wrecking bar at the site." - Frank Lloyd Wright.

From: www.ganssle.com

Refactoring is simplifying the code without changing its behavior. Its purpose isn't to add functionality or fix bugs; rather, we refactor to improve the understandability (and hence maintainability) and/or structure of the code.

When we're afraid to modify even a comment, when the slightest change breaks a function, when the thought of any edit gives us the sweats, that's a clear indication the code is no good and should be rewritten. The goal isn't to add features or fix bugs; it's to make the function maintainable.

Writing Code for Microcontrollers

Programming quotations

Reflection and Introspection:

The notion of continuous improvement has greatly changed the world of manufacturing. It's the deliberate and methodical analysis of processes to find ways to eliminate the unneeded and optimize that which is necessary. Soul searching is an ancient idea behind improving one's self. Feedback stabilizes systems, be they electronic systems or human. We all develop bad habits. Stop and reflect, from time to time, about your development practices.

Bad habits are like entropy -- they accumulate constantly unless one works hard to keep things in order.

Every important endeavor, like engineering embedded systems, benefits from Introspection.

Close the loop. Stop and think about your practices. Change something.

-Jack Ganssle