Name: Lab Section:

CprE 288 Fall 2012 – Homework 5
Due Thu. Oct. 4 in the class

Notes:

- **Start early on homework.**
- Homework answers must be typed using a word editor. Hand in a hard copy in the class.
- Late homework is accepted within three days from the due date. **E-mail late homework to both of the grading TAs, Min Sang Yoon (my222@iastate.edu) and Zhen Chen (zchen@iastate.edu)**. *Late penalty is 10% per day (counting from 10:45am of the due date).*

Question 1: C Operator Precedence (15 pts)

Does each of the code pieces serve for its stated purpose? You must **explain** to get the full credit. Assume the following variable declaration:

```
unsigned a, b, c, d, mask;
unsigned *p = &a;
```

Hint: Check the operator precedence chart, slide 49 of week 5.

a. [3] Check if a is equal to b or c is equal to d.

```
if (a == b || c == d)
   … // code
```

Yes. The precedence of || is higher than that of ==. The condition is equivalent to ((a == b) || (c == d)).

b. [3] Check if the upper half of a is equal to the upper half of b.

```
mask = 0xFF00;
if (a&mask == b&mask)
   … // code
```

No. & is lower than == in the precedence table, so the condition is actually (a & (mask == b) & mask(.

c. [3] Increment a through pointer p.

```
*p++;
```

No. ++ is higher than * in the precedence table, so the statement is actually *(p++). The pointer p will be increment, not *p.

d. [3] Swap the upper half and the lower half of a.

```
a = a<<8 + a>>8;
```

No. Because + has higher precedence than >> and <<, the statement is a << (8+a) >> 8.

Grading: The question was originally written as "a = a<<16 + a>>16;" For this reason, give *full credit* to either "yes" or "no" answer.

e. [3] Swap the upper half and the lower half of b.

```
b = b<<8 | b>>8;
```

Yes. | has lower precedence than << and >>, so it is indeed (b << 8) | (b >> 8).

Grading: The question was originally written as "b = b<<16 | b>>16;" For this reason, give *full credit* to either "yes" or "no" answer.

Question 2: C Type Conversion and Casting (15 pts)

Note: This question was not graded because of an ambiguity in the question. "char" type in C can be "singed char" or "unsigned char" depending on the platform. On the current AVR platform, it seems to be "unsigned char" by default but can be defined as "signed char" through a compiler option.

As a general rule, it's always a good practice to use either "signed char" or "unsigned char" in C code when using 8-bit integer.

The following solutions are given assuming "char" to be "unsigned char" on the AVR platform.

What is the outcome of each of the following assignment statements or if-condition? For an assignment statement, state the value of the variable being assigned after the execution of the statement. For an if-condition, state it is True or False. You must **explain** to get the full credit. Assume the following variable declaration:

```
char byte = -24;
short ADC_input = 1020;
unsigned n;
```

Name: Lab Section:

Hint: If you are not sure about an outcome, you can write a program to find out (using AVR studio).

a. [3] `byte = ADC_input;`

1020 is 0x 03FC. The upper half is cut off, and the lower half 0xFC is put into type. Byte will have a value of 0xFC or 252.

b. [3] `n = (int) byte;`

8-bit -24 is hex 0xE8. When it's assigned to byte, byte gets the value of 0xE8 or unsigned decimal 232. It's then sign-extended to 16-bit 232 and assigned to n.

c. [3] `if ((unsigned) byte > (unsigned) ADC_input)`

"unsigned" means "unsigned int" in C. "(unsigned) byte" is unsigned decimal of 232. ADC_input is signed decimal of 1024 and it's converted to unsigned decimal of 1024, so the condition is false.

d. [3] `n = ADC_input + byte;`

8-bit unsigned value of byte, which is 232, is extended to 16-bit signed value, which is also 232. It's then added to 1024, and the result is 1256.

e. [3] `n = ADC_input + (unsigned) byte;`

8-bit unsigned value of byte, which is 232, is extended to 16-bit unsigned value, which is also 232. It's then converted to 16-bit signed value, which is still 232. The it's added to 1024, and the result is 1256.

Question 3: C Standard Library (10 pts)

Assume in a programming assignment you need a function to test whether a C string is a substring of another C string; for example, yes for "CprE" being a substring of "This is CprE288". Find out such a function in C Standard Library. Write a short program to verify that the function you found works. Test it in a programming environment, e.g. Linux, Microsoft Visual Studio, or Dev-C++ (use only C code). Cut and paste your program code and test output here.

Name: Lab Section:

Question 4: Interrupt Service Routine (10 pts)

Consider the following code which is a partial implementation of interrupt-based clock implementation of Lab 4 assignment:

```
static int hours;
static int minutes;
static int seconds;

ISR (TIMER1_COMPA_vect)
{
  char buffer[12];

  … // code for advancing the clock

  /* print out time in HH:MM:SS format*/
  sprintf(buffer, "%02d:%02d:%02d", hours, minutes, seconds);
  lcd_puts(buffer);
}
```

a. Discuss generally why it's a bad idea to call time-consuming functions like "lcd_puts" from the inside of an ISR function.

Developers should keep ISRs short and simple, avoiding loops and other constructs which can increase latency and complexity. When an interrupt fires, the microprocessor typically disables interrupts globally before transferring control to the ISR; then either the ISR must re-enable interrupts or the ISR return instruction will re-enable interrupts automatically. By keeping ISRs short and simple, developers will avoid the common pitfall of leaving interrupts disabled for too long, thereby increasing the latency of higher priority interrupts. In addition, ISRs are notoriously difficult to debug and often must be analyzed by inspection – practical only for simple implementations. Keeping ISR short will minimize interrupt response time, testing and debugging time.

Grading: It's OK if one has covered the points in above paragraph.

b. Discuss specifically what bad program behavior a user may experience with this implementation.

If the ISR is too long then there are possibilities of missing certain functions or dropping off some data. For example in lab 4, if the ISR is long (runs for more than 200ms), it would miss the pushbutton function.

Grading: It's OK to use other examples.

Name: Lab Section:

Question 5: Interrupt Service Routine (10 pts)

Explain why a variable shared by an ISR and the foreground program execution must be declared as "volatile". In particular, discuss what will happen if "volatile" is NOT used on the shared variable and complier optimization is enabled. The following is an example:

```
volatile static int clock_flag = 0;

ISR (TIMER1_COMPA_vect)
{
  clock_flag = 1;
}

int main()
{
   … // other code

   while (1)
     {
       if (clock_flag == 1)
         {
            … // advance the clock, print clock on LCD
         }
     }
}
```

The 'volatile' qualifier tells the compiler that the variable may change spontaneously. The compiler shouldn't rely on a cached copy of the variable which may be left over from a previous calculation, but should fetch a fresh copy each and every time it uses the variable. If a variable is used in an ISR and in a function outside the ISR, it should be declared 'volatile'.