CprE 288 – Introduction to Embedded Systems

Instructors:

Dr. Zhao Zhang (Sections A, B, C, D, E)

Dr. Phillip Jones (Sections F, G, J)

Overview

- Announcements
- Bitwise Operations
  - Set, clear, toggle and invert bits
  - Shift bits
  - Test bits
- Preprocessor Directives
- String functions
- Lab 3

Announcements

- Homework 3 due in class on Thursday
- Exam 1 in two weeks

BITWISE OPERATIONS

Why Bitwise Operation

Why use bitwise operations in embedded systems programming?

Each single bit may have its own meaning

- Push button array: Bit *n* is 0 if push button *n* is pushed
- LED array: Set bit *n* to 0 to light LED *n*

Data from/to I/O ports may be packed

- Two bits for shaft encoder, six bits for push button packed in PINC
- Keypad input: three bits for row position, three bits for column position

Data in memory may be packed to save space

- Split one byte into two 4-bit integers

Why Bitwise Operation

SW6 SW5 SW4 SW3 SW2 SW1
Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0

Read the input:

```
unsigned char ch = PINC;
```

Then, how does the code get to know which button is being pushed?

Connected to PINC, bits 5-0
PINC, bits 7-6 are input from shaft encoder

Bitwise Operations: What To Do?

We may want to do following programming tasks:

- Clear/Reset certain bit(s)
- Set certain bit(s)
- Test if certain bit(s) are cleared/reset
- Test if certain bit(s) are set
- Toggle/invert certain bits
- Shift bits around

Bitwise Operators: Clear/Reset Bits

C bitwise AND: &

ch = ch & 0x3C;	What does it do?

Consider a single bit *x*

x AND 1 = x	Preserve
x AND 0 = 0	Clear/Reset

Bitwise Operators: Clear/Reset Bits

ch = ch & 0x3C;

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
AND	0	0	1	1	1	1	0	0
	0	0	x_5	x_4	x_3	x_2	0	0

Clear bits 7, 6, 1, 0

Preserve bits 5, 4, 3, 2

Clear bit(s): Bitwise-AND with a mask of 0(s)

Bitwise Operators: Clear/Reset Bits

Another example:

char op1 = 1011 1100; We want to set bit 4 to 0.

char op2 = 1110 1111; We use op2 as a mask

char op3;

```
        1011 1100
AND     1110 1111
        ---------
        1010 1100
```

op3 = op1 & op2;

Class Exercise

char ch;

int n;

Clear the upper half of ch

Clear every other bits of ch starting from 0

Clear the lower half of n

Bitwise Operators: Set Bits

C bitwise OR: |

ch = ch | 0xC3;　　　What does it do?

Consider a single bit *x*

x OR 1 = 1　　　Set

x OR 0 = x　　　Preserve

Bitwise Operators: Set Bits

ch = ch | 0xC3;

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
OR	1	1	0	0	0	0	1	1
	1	1	x_5	x_4	x_3	x_2	1	1

Set bits 7, 6, 1, 0

Preserve bits 5, 4, 3, 2

Set bit(s): Bitwise-OR with a mask of 1(s)

Bitwise Operators: Set Bit

Another example:

char op1 = 1000 0101; We want to set bit 4 to 1.

char op2 = 0001 0000; We use op2 as a mask

char op3;

op3 = op1 | op2;

```
       1000 0101
OR     0001 0000
    ------------
       1001 0101
```

Bitwise Operators: Toggle Bits

C bitwise XOR: ^ ch = ch ^ 0x3C;

$$
\begin{array}{lcccccccc}
 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\
\textbf{XOR} & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
\hline
 & x_7 & x_6 & \overline{x_5} & \overline{x_4} & \overline{x_3} & \overline{x_2} & x_1 & x_0
\end{array}
$$

Toggle bits 5, 4, 3, 2

Preserve bits 7, 6, 1, 0

Toggle bit(s): Bitwise-XOR with a mask of 1(s)

Bitwise Operators: Invert Bits

C bitwise invert: ~ 		ch = ~ch;

INV	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
	$\overline{x}_7$	$\overline{x}_6$	$\overline{x}_5$	$\overline{x}_4$	$\overline{x}_3$	$\overline{x}_2$	$\overline{x}_1$	$\overline{x}_0$

Example: ch = 0b00001111;

~ch == 0b11110000

Class Exercise

char ch;

int n;

Set the lower half of ch

Set every other bits starting from 0

Set bit 15 and bit 0 of n

Toggle bits 7 and 6 of ch

Bitwise Operators: Shift-Left

```
unsigned char my_reg = 0b00000001;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg << shfit_amount;
```

00000**001**

00100000

<<, shifts “my_reg”, “shift_amount” places to the left

0s are shifted in from the right

Bitwise Operators: Shift-Right Logical

unsigned char my_reg = 0b10000000;

unsigned char shift_amount = 5;

unsigned char my_result;

my_result = my_reg **>>** shfit_amount;

10000000

00000100

With unsigned type, **>>** is **shift-to-right logical**

0s are shifted in from the left

Bitwise Operators: Shift-Right Arithmetic

signed char my_reg = 0b10000000;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shfit_amount;

10000000

11111100

my_reg = 0b01111111;
my_result = my_reg >> shfit_amount;

01111111

00000011

With signed type, >> is **shift-right arithmetic**

Sign bit value are shifted in from the left

Bitwise Operators: Shift and Multiple/Divide

n << k **is equivalent to** **$n * 2^k$**

Example: 5 << 2 = 5*4 = 20

0b0000 0101 << 2 = 0b0001 0100

n >> k **is equivalent to** **$n / 2^k$**

Example: 20 >> 2 = 5

0b0001 0100 << 2 = 0b0000 0101

Bitwise Operators: Shift and Multiple/Divide

Shift-Right Arithmetic: Why shift in the sign bit?

Example: (char) 32 >> 2 = 32 / 4 = 8

0b0010 0000 >> 2 = 0b0000 1000

Example: (char) -32 >> 2 = -32/ 4 = -8

0b1110 0000 >> 2 = 0b1111 1000

Bitwise Operators: Shift and Set

What's the effect of the following state?

```
#define  BIT_POS   4
ch = ch | (1 << BIT_POS);
```

What is (1 << 4)?

0000 0001 << 4

0001 0000

In general case: (1 << n) yields **a mask of a 1 at bit n**

The effect of the statement: Set bit 4

Bitwise Operators: Shift and Set

Another example:

unsigned char my_mask = 0000 0001;

unsigned char shift_amount = 5;

unsigned char my_result = 1101 0101; Want to force bit 5 to a 1

my_result = my_result | (my_mask << shift_amount);

1101 0101 | 00100000

```
   1101 0101
OR 0010 0000
   ---------
   1111 0101
```

Shift the 1(s) of the MASK to the appropriate position, then OR with my_result to force corresponding bit positions to 1.

Bitwise Operators: Shift and Clear

What's the effect of the following state?

```
#define  BIT_POS   4
ch = ch & ~(1 << BIT_POS);
```

What is ~(1 << 4)?

0000 <u>0001</u>	<< 4
<u>0001</u> 0000	~
111**0** 1111	

In general case: ~(1 << n) yields **a mask of a 0 at bit n**

Note: Compiler does the calculation at compilation time

Bitwise Operators: Shift and Clear

unsigned char my_mask = 0000 0001;

unsigned char shift_amount = 5;

unsigned char my_result = 1011 0101; Want to force bit 5 to a 0

my_result = my_result & ~(my_mask << shift_amount);

$$\frac{1011\ 0101\ \&\ \sim 00100000}{1011\ 0101\ \&\ \ 11011111} \rightarrow \begin{array}{rr} & 1011\ 0101 \\ \text{AND} & \underline{1101\ 1111} \\ & 1001\ 0101 \end{array}$$

Shift the 0(s) of the MASK to the appropriate position, then AND with my_result to force corresponding bit positions to 0.

Exercise

unsigned char ch;

unsigned int n;

Divide n by 32 in an efficient way

Swap the upper half and lower half of ch

Exercise

```
unsigned char ch = PINC;
unsigned char shaft_encoder_reading;
```

Bits 7 and 6 of PINC are a two-bit reading of the status of the shaft encoder.

Make those two bits the only two meaningful bits in `shaft_encoder_reading`

Bitwise Testing

Remember, conditions are evaluated on the basis of zero and non-zero.

The quantity 0x80 is non-zero and therefore TRUE.

```
if (0x02 | 0x44)
```

Valid or not?

Bitwise Testing

Example

Find out if bit 7 of variable nVal is set

Bit 7 = 0x80 in hex

```
if ( nVal & 0x80 )
{       …
}
```

What happens when we want to test for multiple bits?

if statement looks only for a non-zero value

a non-zero value means at least one bit is set to TRUE

Bitwise Testing: Any Bit Is Set?

Example

See if bit 2 or 3 is set

Bits 2,3 = 0x0C in hex

```
if (nVal & 0x0C)
{
        Some code…
}
```

What happens for several values of nVal?

nVal = 0x04	bit 2 is set	Result = 0x04	TRUE
nVal = 0x0A	bits 3,1 are set	Result = 0x08	TRUE
nVal = 0x0C	bits 2,3 are set	Result = 0x0C	TRUE

Bitwise Testing: All Bits Are Set?

Why does this present a problem?

- What happens if we want to see if both bits 2 and 3 are set, not just to see if one of the bits is set to true?
- Won't work without some other type of test

Two solutions

- Test each bit individually

```
if ( (nVal & 0x08) && (nVal & 0x04))
```

- Check the result of the bitwise AND

```
if ((nVal & 0x0C) == 0x0C)
```

Why do these solutions work?

1. Separate tests – Check for each bit and specify logical condition
2. Equality test – Result will only equal 0x0C if bits 2 and 3 are set

Exercise

```
char ch;
```

Test if any of bits 7, 6, 5, 4 is set

Test if all of bits 7, 6, 5, 4 are set

Exercise

Write a program to count the number of 1s in integer n

```
int n;
```

I/O Ports

ATmeag128

- 5 general purpose ports: Port A, B, C, D, E; two special purpose – Port F & G.
- Processor communicates with them through memory mapped I/O.
- Set of data and control registers associated with each port.

I/O Ports

- The processor communicates with attachments using ports
- Each port has three registers

PORTx – 8bit register for output

PINx – 8bit register for input

DDRx – Data direction register

- DDR
  - 0 means input
  - 1 means output

Example:

```
DDRA = 0b00000001; // all bits on port A are used for input
                   // except bit0
```

I/O Ports

DDRX Register (Data Direction Register)

•E.g. DDRA: 0 – input; 1 - output

I/O Ports

PORTX Register:

PORTA: If ***PORTxn*** is 1 when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin.

For output configured port: If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). and *vice versa*.

Write to a port through PORTX register.

E.g.:

PORTA = my_char; // set port A to be value of my_char

I/O Ports

PINX Register (is a data register):

Always keeps the current state of the physical pin.

Read only!

For an input port, the only way to read data from that port.

E.g:

my_char = PINA; //set my_char to value on port A

39

Example: Initialize Push Buttons

```
/// Initialize PORTC to accept push buttons as input
void init_push_buttons(void) {
      DDRC &= 0xC0;  //Setting PC0-PC5 to input
      PORTC |= 0x3F; //Setting pins' pull up resistors
}
```

Push Button port connection

- Port C, pin 0 to pin 6 (button SW1 to SW6)
- All input

Example: Initialize Shaft Encoder

```
/// Initialize PORTC for input from the shaft encoder
void shaft_encoder_init(void) {
      DDRC &= 0x3F;     //Setting PC6-PC7 to input
      PORTC |= 0xC0;    //Setting pins' pull-up
resistors
}
```

Sharf encoder port connection

- Port C, pin 7 and 6
- Input

Example: Initialize Stepper Motor

```
/// Initialize PORTE to control the stepper motor
void stepper_init(void) {
      DDRE |= 0xF0;      //Setting PE4-PE7 to output
      PORTE &= 0x8F;     //Init position (0b1000) PE4-PE7
      wait_ms(2);
      PORTE &= 0x0F;     //Clear PE4-PE7
}
```

Sharf encoder port connection

- Port C, pin 7 and 6
- Output
- Wait for 2 ms for stepper model to settle

PREPROCESSOR DIRECTIVES

Preprocessor Directives

- The C preprocessor runs over the code before compiling; it is a glorified text editor
- Some useful preprocessor directives:
  - #include copies a files contents to the given line
  - #define defines a preprocessor variable
  - #ifdef if a preprocessor variable is defined
  - #ifndef if a preprocessor variable is not defined
  - #else
  - #end
- The preprocessor runs before the code is compiled
- You should understand each of these preprocessor directives

Preprocessor Directives

- #include
  - Copies all of the text from a given file into the line where the #include statement is located
- #include “myprojectfile.h”
  - Use quotation marks “” to include a file from the project path
- #include <stdio.h>
  - Use brackets <> to include a header file from the library path
  - Examples:
    - #include <avr/io.h>
    - #include <stdio.h>
    - #include <string.h>
    - #include <math.h>

Before the Preprocessor

sensors.h

```
int ir_distance();

int sonar_distance();
```

sensors.c

```
#include “sensors.h”

int ir_distance() {
  // code
}

int sonar_distance() {
  // code
}
```

After the Preprocessor

sensors.c

```
int ir_distance();

int sonar_distance();

int ir_distance() {
  // code
}

int sonar_distance() {
  // code
}
```

Preprocessor Directives

- #define has three common uses
    1. defines a *name* (preprocessor variable) for the preprocessor to find and replace in the source code with a given *value*
    2. defines a preprocessor variable for use with #ifdef/#ifndef
    3. defines a macro
- Example #1:
    - #define MAX_SPEED 500
    - #define PI 3.1459
- Example #2
    - #define __SENSORS.h__
- Example #3
    - #define ADDMEANINGOFLIFE(a) (a + 42)

Before the Preprocessor

silly.c

```
#define SILLY
#define LIFE 42
#define ADDONE(x) (x + 1)

int main() {
#ifdef SILLY
  char message = "Yay!";
#end
  int x = LIFE;
  int y = ADDONE(5);
  return 0;
}
```

After the Preprocessor

silly.c

```
int main() {
  char message = “Yay!”;
  int x = 42;
  int y = 5 + 1;
  return 0;
}
```

Avoiding Circular Imports

headerA.h

```
#ifndef __HEADERA.h__
#define __HEADERA.h__

#include “headerB.h”

int functionA1();
int function A2();

#end
```

headerB.h

```
#ifndef __HEADERB.h__
#define __HEADERB.h__

#include “headerA.h”

void functionB1();
void functionB2();

#end
```

STRING FUNCTIONS

String Manipulation Functions

- int sprintf(char * str, const char * format, ...);
- int strlen(const char * str);
- int strncmp(const char * str1, const char * str2, size_t num);

String Functions: sprintf

int sprintf (char * str, const char * format, ...);

Param1: location to store the string (e.g. character array)

Param2: formatted string to store in the array

Param3-n: formatting variables that appear in the formatted string.

Example:

```
int class_num = 288;
char my_array[20];
sprintf(my_array, “Hello CPRE %d \n”, class_num);
// my_array now contains: Hello CPRE 288
```

String Functions: strlen

int strlen (const char * str);

Param1: location of a string (e.g. character array)

Return value: returns the length of the string (not counting NULL byte).

Example:

```
char my_array[20] = "Hello CPRE288";
int     my_len = 0;
my_len = strlen(my_array);

// my_len now has a value of 13
```

String Functions: strcmp

int strcmp (const char * str1, const char * str2,);

Param1: location of a string

Param2: location of a string

Return value: if equal then 0, if the first position that does not match is greater in str1 then +, else -.

Example:

```
char my_array1[20] = “apple”;
char my_array2[20] = “pair”;
int my_compare = 0;
my_compare  = strcmp(my_array1, my_array2);
// ‘p’ has a higher value than ’a’, so my_compare will be negative
```

Class Activity

- Predict the value of *message* after each line:

```
char* str1 = “hello”, str2 = “world”;
char message[100];

sprintf(message, “The meaning of life is %d.”, 42);
“The meaning of life is 42.”
sprintf(&message[0], “The meaning of life is %i.”, 42);
“The meaning of life is 42.”
sprintf(message, “%s %s”, &str1[0], str2);
“hello world”
sprintf(message, “%s”, &str1[1] );
“ello”
sprintf(message, “%20s”, str2);
“               world”
```

Lab 3

- Overview of hardware
  - Push Buttons (Switches)
  - Shaft Encoder (Control Knob)
  - Stepper Motors

Lab 3 Memory-Mapped I/O

Now write your own API functions for I/O devices

Part I. Push button

To detect which buttons are being pushed

Part II. Shaft Encoder

To take input of a shaft and emulate its behavior

Part III. Stepper Motor

To control motor movement precisely

Lab 3 Memory Mapped I/O

Part I. Push button

Return the position of the leftmost button that is being pressed. The rightmost button is position 1. Return 0 if no button is being pressed.

```
char read_push_buttons(void);
```

```
Six push buttons, connected to PINC
  bits 5-0
Active low - if a button is pushed, the
  corresponding bit is 0, otherwise 1
```

Lab 3 Memory Mapped I/O

Q1: How does it work mechanically and electronically?

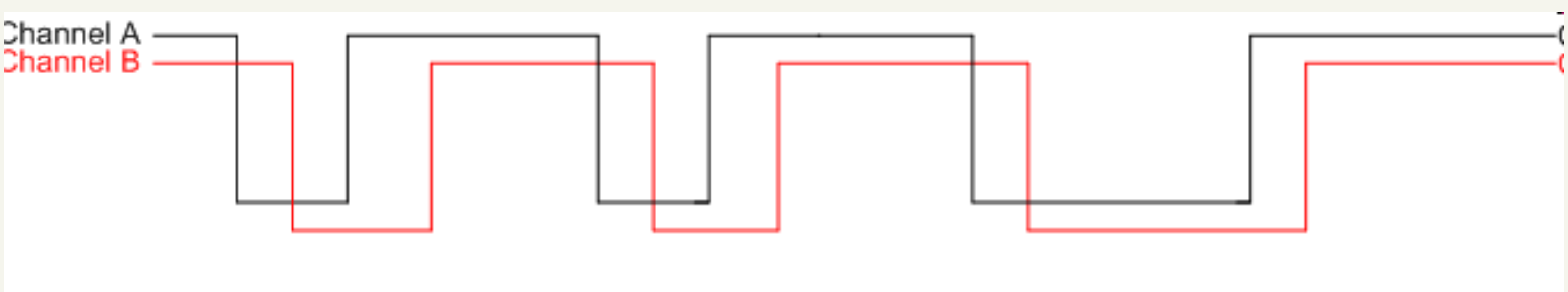
Q2: How to read the raw input from the push buttons?

Q3: How to read a port?

Lab 3 Memory Mapped I/O

Part II. Shaft Encoder

- The device generates two waveforms to two input pins of ATmega128 (PC6 and PC7)
- The direction of the shaft encoder is reflected by the ordering of the two waveforms
- A leading B is clockwise, B leading A is counter-clockwise
- Channel B connected to PINC bit 7, Channel A connected to PINC bit 6



Shaft Encoder Turned Clockwise

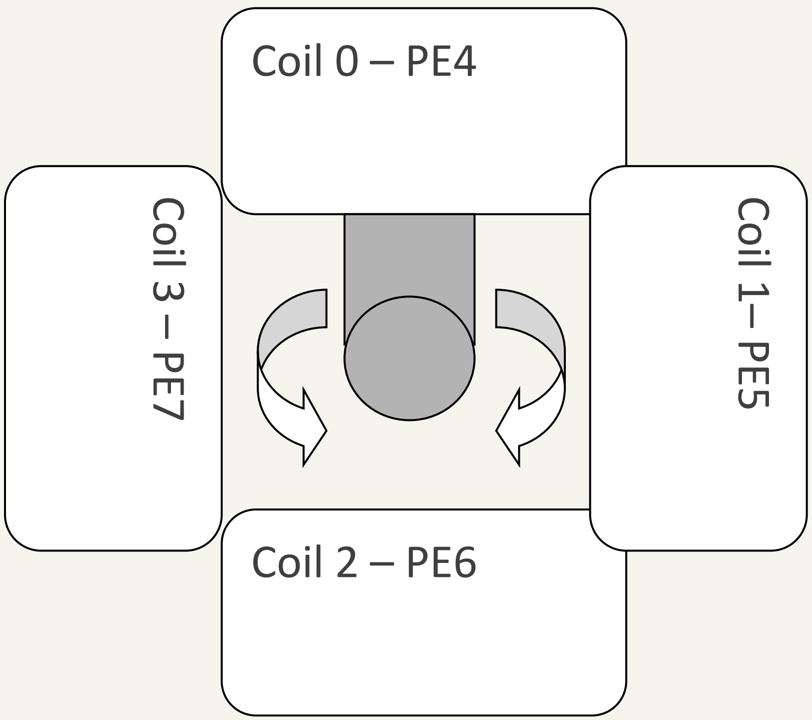
Lab 3 Memory Mapped I/O

Q1: How does your program read and represent the waveform?

Q2: How do you decide the ordering of the waveform, i.e. A leads B or B leads A?

Lab 3 Memory Mapped I/O

Part III. Stepper Motor



To rotate clockwise: send to PE7-PE4 the following sequence: 0001, 0010, 0100, 1000, 0001, …

Allow 2ms gap between two outputs

Lab 3 Memory Mapped I/O

Q1: How to rotate the four bits?

Q2: How to send out the four bits to PE7-PE4 without affecting the other four bits of PORTE?

Q3: How to couple the shaft encoder with the stepper motor?