Announcements

• Homework 3: Due on Sunday 6/5
• Exam 1: In class Wed 6/8

Overview

• Announcements
• Bitwise Operations
  – Set, clear, toggle and invert bits
  – Shift bits
  – Test bits
• I/O Ports
• Lab 3

Why Bitwise Operations

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 Operations

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

Another example:

char op1 = 101 1100;  We want to clear bit 4 to 0.
char op2 = 1110 1111;  We use op2 as a mask
char op3;

op3 = op1 & op2;

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

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**

\[ ch = ch \ Or \ 0xC3; \]

<table>
<thead>
<tr>
<th>x⁷</th>
<th>x⁶</th>
<th>x⁵</th>
<th>x⁴</th>
<th>x³</th>
<th>x²</th>
<th>x₁</th>
<th>x₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

\[ \text{char op1} = 1000 \ 0101; \] We want to set bit 4 to 1.
\[ \text{char op2} = 0001 \ 0000; \] We use op2 as a mask
\[ \text{char op3;} \]
\[ \text{op3} = \text{op1} \ Or \ \text{op2}; \]
\[ \begin{array}{c}
\text{1000 0101} \\
\text{OR} \\
\text{0001 0000} \\
\hline
\text{1001 0101}
\end{array} \]

**Bitwise Operators: Toggle Bits**

C bitwise XOR: ^

\[ ch = ch \ Xor \ 0x3C; \]

What does it do?

Consider a single bit \( x \)
\[ x \ Xor \ 1 = \overline{x} \quad \text{Toggle} \]
\[ x \ Xor \ 0 = x \quad \text{Preserve} \]

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

**Bitwise Operators: Invert Bits**

C bitwise invert: ~

\[ \text{ch} = \text{~ch}; \]

\[ \begin{array}{cccccccc}
\text{x7} & \text{x6} & \text{x5} & \text{x4} & \text{x3} & \text{x2} & \text{x1} & \text{x0} \\
\text{INV} & \text{x7} & \text{x6} & \text{x5} & \text{x4} & \text{x3} & \text{x2} & \text{x1} & \text{x0} \\
\end{array} \]

Example: \( \text{ch} = 0b00001111; \)
\[ \text{~ch} = 0b11110000 \]

**Class Exercise**

char ch;
int n;
Set the lower half of ch
Set every other bits starting from 0 of ch
Set bit 15 and bit 0 of n
Toggle bits 7 and 6 of ch
Bitwise Operators: Shift-Left

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

my_result = my_reg << shift_amount;
```

<<, shifts “my_reg”, “shift_amount” places to the left
0s are shifted in from the right

Bitwise Operators: Shift-Right Logical

```c
unsigned char my_reg = 0b10000000;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shift_amount;
```

With unsigned type, >> is shift-to-right logical
0s are shifted in from the left

Bitwise Operators: Shift-Right Arithmetic

```c
signed char my_reg = 0b1000000;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shift_amount;
```

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 = 0b0000_0100

n >> k is equivalent to n / 2^k
Example: 20 >> 2 = 5
        0b0001_0100 >> 2 = 0b0000_0101
```

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);

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

unsigned char my_mask = 0000 0111;
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);

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 = 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

http://class.ece.iastate.edu/cpre288

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

http://class.ece.iastate.edu/cpre288

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

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

Memory Mapped I/O

How does a program executing within CPU communicate with the keyboard?

Get_user_ID(char *name);
Memory Mapped I/O

Memory mapped I/O: Registers within Keyboard appear to be in memory.

Get_user_ID(char *name);

char * KBDR;
KBDR = (char *) 0xff00;
while ((*(name++) = *KBDR) != newline);

How quickly does the while loop iterate?
10-100 µs?

How quickly do we type?
30-100 chars/minute?

• We need a control mechanism for the keyboard to tell us when fresh data is available in KBDR?
• Keyboard Control Register (KBCR)

Bit Fields in Structures

struct KBCR {
    // Big Endian
    unsigned int model : 4;
    unsigned int KBERROR : 1;
    unsigned int CAPLOCK : 1;
    unsigned int READY : 1;
} KBCR;

Union

• Maintain multiple views: byte or bit structure.

union KBCR_U {
    struct KBCR KBCR;
    uint8_t KBCR_Aggregate;
} KBCR_U;

KBCR_U.KBCR_Aggregate = 0xc1;
KBCR_U.KBCR.READY = 1;

Memory Mapped I/O

• In one attempt, we may not even get one character?
Memory Mapped I/O - Polling

- Without any device (keyboard) specific instructions, we can talk to the device/sensor.
- Even the future devices whose interface we do not know yet can be memory-mapped!

```c
char *KBDR;
char *KBCR;
KBDR = 0xff00;
KBCR = 0xff01;
while (!(KBCR.READY)); //polling loop
   //guaranteed fresh data
   if(*((name++) = *KBDR) == NEWLINE) return;
```

Bit fields in structures

```c
if (KBCR.READY){  
struct KBCR  *pKBCR;
pKBCR->CAPLOCK = 0;
struct student student_records[100];
```

Atmel AtMega I/O Ports

- What if sensor is not smart enough to pretend to be memory? Cannot memory-map its interface?

```c
char *KBDR;
char *KBCR;
KBDR = 0xf000;
KBCR = 0xff01;
while (!(*KBCR & 0x1)); //guaranteed fresh data
   if(*((name++) = *KBDR) == NEWLINE) return;
```
Memory-Mapped I/O Ports

- Built-in ports are memory-mapped.

I/O Ports

- ATmega128
  - 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 bit 0

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
Example: Initialize Push Buttons

```c
/// 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 5 (button SW1 to SW6)
- All input

Example: Initialize Shaft Encoder

```c
/// 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
}
```

Shaft encoder port connection
- Port C, pin 7 and 6
- Input

Example: Initialize Stepper Motor

```c
/// 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
}
```

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

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

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?

Shaft encoder

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

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?
Stepper Motor (Wikipedia)

- Full rotation divided into multiple steps.
- Motion is controllable one step at a time without need for feedback.
- Four coils giving four magnetic axes.

Stepper Motor Control

- 200 steps per 360°: 1.8 ° per step.
- 0001 → 0010 → 0100 → 1000 → 0001
- 0001 → 1000 → 0100 → 0010 → 0001 → ....
- 0001 → 1000 → 0100 → 0010 → 0001 → ....
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

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?