AVR BIT TWIDDLING

INTRODUCTION

Often when programming in a microcontroller (or on any computer, for that matter), the ability to manipulate individual bits will become useful or even necessary. Here are some situations where bit math can be helpful:

- Saving memory by packing up to 8 true/false data values in a single byte.
- Turning on/off individual bits in a control register or hardware port register.

1. BITWISE AND

The bitwise AND operator in C++ is a single ampersand, &, used between two other integer expressions. Bitwise AND operates on each bit position of the surrounding expressions independently, according to this rule: if both input bits are 1, the resulting output is 1, otherwise the output is 0. Another way of expressing this is:

```
0 & 0 == 0
0 & 1 == 0
1 & 0 == 0
1 & 1 == 1
```

In Arduino, the type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur. In a code fragment like:

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a & b;  // result:    0000000001000100, or 68 in decimal.
```

Each of the 16 bits in a and b are processed by using the bitwise AND, and all 16 resulting bits are stored in c, resulting in the value 01000100 in binary, which is 68 in decimal.

One of the most common uses of bitwise AND is to select a particular bit (or bits) from an integer value, often called *masking*. For example, if you wanted to access the least significant bit in a variable x, and store the bit in another variable y, you could use the following code:

```
int x = 5;       // binary: 101
int y = x & 1;   // now y == 1
x = 4;           // binary: 100
y = x & 1;       // now y == 0
```

2. BITWISE OR

The bitwise OR operator in C++ is the vertical bar symbol, |. Like the & operator, | operates independently each bit in its two surrounding integer expressions, but what it does is different (of course). The bitwise OR of two bits is 1 if either or both of the input bits is 1, otherwise it is 0. In other words:

```
0 | 0 == 0
0 | 1 == 1
1 | 0 == 1
1 | 1 == 1
```

Here is an example of the bitwise OR used in a snippet of C++ code:

```
int a =  92;    // in binary: 0000000001011100
int b = 101;    // in binary: 0000000001100101
int c = a | b;  // result:    0000000001111101, or 125 in decimal.
```

Bitwise OR is often used to make sure that a given bit is turned on (set to 1) in a given expression. For example, to copy the bits from a into b, while making sure the lowest bit is set to 1, use the following code: b = a | 1;

3. BITWISE XOR

There is a somewhat unusual operator in C++ called *bitwise exclusive OR*, also known as bitwise XOR. (In English this is usually pronounced "eks-or".) The bitwise XOR operator is written using the caret symbol ^. This operator is similar to the bitwise OR operator |, except that it evaluates to 1 for a given position when exactly one of the input bits for that position is 1. If both are 0 or both are 1, the XOR operator evaluates to 0.

```
0 ^ 0 == 0
```

```
0 ^ 1 == 1
1 ^ 0 == 1
1 ^ 1 == 0
```

Another way to look at bitwise XOR is that each bit in the result is a 1 if the input bits are different, or 0 if they are the same. Here is a simple code example:

```
int x = 12;     // binary: 1100
int y = 10;     // binary: 1010
int z = x ^ y;  // binary: 0110, or decimal 6
```

The ^ operator is often used to toggle (i.e. change from 0 to 1, or 1 to 0) some of the bits in an integer expression while leaving others alone. For example:

```
y = x ^ 1;   // toggle the lowest bit in x, and store the result in y.
```

4. BITWISE NOT

The bitwise NOT operator in C++ is the tilde character ~. Unlike & and |, the bitwise NOT operator is applied to a single operand to its right. Bitwise NOT changes each bit to its opposite: 0 becomes 1, and 1 becomes 0. For example:

```
int a = 103;    // binary:  0000000001100111
int b = ~a;     // binary:  1111111110011000 = -104
```

You might be surprised to see a negative number like -104 as the result of this operation. This is because the highest bit in an int variable is the so-called *sign bit*. If the highest bit is 1, the number is interpreted as negative. This encoding of positive and negative numbers is referred to as *two's complement*.

As an aside, it is interesting to note that for any integer x, ~x is the same as -x-1. At times, the sign bit in a signed integer expression can cause some unwanted surprises, as we shall see later.

5. BIT SHIFT OPERATORS

There are two *bit shift* operators in C++: the *left shift* operator << and the *right shift*operator >>. These operators cause the bits in the left operand to be shifted left or right by the number of positions specified by the right operand. For example:

```
int a = 5;        // binary: 0000000000000101
int b = a << 3;   // binary: 0000000000101000, or 40 in decimal
int c = b >> 3;   // binary: 0000000000000101, or back to 5 like we started with
```

When you shift a value x by y bits (x << y), the leftmost y bits in x are lost, literally shifted out of existence:

```
int a = 5;        // binary: 0000000000000101
int b = a << 14;  // binary: 0100000000000000 - the first 1 in 101 was discarded
```

If you are certain that none of the ones in a value are being shifted into oblivion, a simple way to think of the left-shift operator is that it multiplies the left operand by 2 raised to the right operand power.

When you shift x right by y bits (x >> y), and the highest bit in x is a 1, the behavior depends on the exact data type of x. If x is of type int, the highest bit is the sign bit, determining whether x is negative or not, as we have discussed above. In that case, the sign bit is copied into lower bits, for esoteric historical reasons:

```
int x = -16;     // binary: 1111111111110000
int y = x >> 3;  // binary: 1111111111111110
```

This behavior, called *sign extension*, is often not the behavior you want. Instead, you may wish zeros to be shifted in from the left. It turns out that the right shift rules are different for unsigned int expressions, so you can use a typecast to suppress ones being copied from the left:

```
int x = -16;               // binary: 1111111111110000
int y = unsigned(x) >> 3;  // binary: 0001111111111110
```

If you are careful to avoid sign extension, you can use the right-shift operator >> as a way to divide by powers of 2. For example:

```
int x = 1000;
int y = x >> 3;   // integer division of 1000 by 8, causing y = 125.
```

6. ASSIGNMENT OPERATORS

Often in programming, you want to operate on the value of a variable x and store the modified value back into x. In most programming languages, for example, you can increase the value of a variable x by 7 using the following code:

```
x = x + 7;    // increase x by 7
```

Because this kind of thing occurs so frequently in programming, C++ provides a shorthand notation in the form of specialized *assignment operators*. The above code fragment can be written more concisely as:

```
x += 7;    // increase x by 7
```

It turns out that bitwise AND, bitwise OR, left shift, and right shift, all have shorthand assignment operators. Here is an example:

```
int x = 1;  // binary: 0000000000000001
x <<= 3;    // binary: 0000000000001000
x |= 3;     // binary: 0000000000001011 - because 3 is 11 in binary
x &= 1;     // binary: 0000000000000001
x ^= 4;     // binary: 0000000000000101 - toggle using binary mask 100
x ^= 4;     // binary: 0000000000000001 - toggle with mask 100 again
```

There is no shorthand assignment operator for the bitwise NOT operator ~; if you want to toggle all the bits in x, you need to do this:

```
x = ~x;    // toggle all bits in x and store back in x
```

7. BITWISE OPERATORS VS. BOOLEAN OPERATORS

It is very easy to confuse the bitwise operators in C++ with the boolean operators. For instance, the bitwise AND operator & is not the same as the boolean AND operator &&, for two reasons:

They don't calculate numbers the same way. Bitwise & operates independently on each bit in its operands, whereas && converts both of its operands to a boolean value (true==1 or false==0), then returns either a single true or false value. For example, 4 & 2 == 0, because 4 is 100 in binary and 2 is 010 in binary, and none of the bits are 1 in both integers. However, 4 && 2 == true, and truenumerically is equal to 1. This is because 4 is not 0, and 2 is not 0, so both are considered as boolean true values.

Bitwise operators always evaluate both of their operands, whereas boolean operators use so-called*short-cut* evaluation. This matters only if the operands have side-effects, such as causing output to occur or modifying the value of something else in memory.

8. BIT TWIDDLING WITH AVR GCC COMPILER

The AVR GCC compiler complies with the standard ANSI C specification.

8.1. Setting Bit Values:

Bit shifting is so common in AVR programming that there's even a macro defined that gets included with io.h: it's called _BV() and stands for "bit value." The _BV() macro is just our bit-shift roll in disguise.

In fact, it's even defined as: `#define _BV(bit) (1 << (bit))`

Many times you would like to set the voltage high for only a *single* pin of a port, and to do that you create what is called a 'bit mask', which is a binary number that has single 1-bit in the location corresponding to the pin you want to set, and all the rest of the bits are zero. You then use this number with a bitwise logical operator to finish the task. To create such a number, you would use the command:

_BV(PXn) — // Where X is the port letter, and n is the pin number. For example, suppose you would like to make **pin 2 on PORTA** to be an *output*. You therefore need a bit mask with a 1 in bit number 2 location. The corresponding bit mask is easily built by the command: **_BV(PA2)**

8.2. Setting the port's data direction (input or output) on an ATmega:

DDRX = 0x00; — // Clears all the bits in the PORTx register, which makes all the associated pins to be **inputs** (keep in mind there are only ports A, B, C, and D on the ATmega328), so you can *read* various signals from things such as **switches, sensors,** or for **analog to digital conversion,** etc. (by default all pins are set to input, but you should always set them to make sure anyway.)

DDRX = 0xFF; — // Sets all the bits in the PORTx register, which makes all the associated pins to be **outputs** (keep in mind there are only ports A,B,C, and D on the ATmega328), so you can *control* devices, such as **motors, LED's, speakers**, etc., pretty much anything you

would want the microcontroller to control (keep in mind that the ATmega cannot source much current, so you will probably need to use a transistor between the microcontroller and whatever you are trying to control if it needs more than 10 mA of current).

8.3. Initializing Port Values:

PORTX = 0xFF; // Clears all the bits in the PORTx register, and assuming that the pins are outputs, will make all the pins in PORTx go high (as mentioned in the section on Hex values above). So for example, PORTA = 0xFF; would set all pins in port A to on/high/1/5V, which would be the same as writing: PORTA = 0b11111111; Keep in mind that the right-most bit corresponds to pin 0 on PORTA in this example.

PORTX = 0x00; // Clears all bits in the PORTx register and makes all pins in PORTx go **low** (as mentioned in the section on Hex values above). So for example, PORTB = 0x00; would clear all bits in the PORTB register and make all pins in PORTB go off/low/0/0V. This would be the same as writing: PORTB = 0b00000000;

8.4. Bitwise Logic:

PORTX |= 0xF0; // Method for <u>setting</u> bits. Performs a bitwise **OR** operation with the current bit values of PORTX and the bit mask represented by the binary number, 0xF0. The result is that only pins 4-7 in PORTX are set high**, and the other pins are not affected**. Equivalent in 'long hand' would be: PORTX = PORTX | 0xF0;

PORTX &= ~0x01; // Method for <u>clearing</u> bits. Performs a bitwise **AND** operation with the current bit values of PORTX and the bit mask represented by the binary number, ~(0b00000001) or 0b11111110. The result is that **pin 0 is set low** (referred to as 'cleared'), regardless of what was there before. In this example, only pin 0 is cleared, and the state of the other pins are not affected. Equivalent in 'long hand' would be PORTX = PORTX & ~0x01;

PORTX ^= 0x02; // Method for <u>toggling</u> bits. Performs a bitwise **XOR** operation with the current bit values of PORTX and the bit mask represented by the binary number, 0x02. The result is that **pin 1 is 'toggled' between the off and on** states. Think of this operation like a light switch. Each time the statement is executed, it changes the state to be the opposite of what it was previously. Similar to the prior statements, this method of toggling only affects the bits in the locations where there are ones in the bit mask hex value. 'Long hand' would be
PORTX = PORTX ^ 0x02;

The methods shown above are what you need to do to set up and control all the functions of the microcontroller.
For the purpose of class, the following are three important bit-twiddling operations.

Operation	Implementation in C	Implication
Set a bit	`PORTB |= (1<<PB1)`	Bit PB1 is set to 1 (other pins are left unchanged)
Clear bit	`PORTB &= ~(1<<PB1)`	Bit PB1 is set to 0 (other pins are left unchanged)
Toggle a bit	`PORTB ^= (1<<PB1)`	If Bit PB1 was 1, it is toggled to 0. Otherwise, it is set to 1 (other pins are left unchanged)
Read a value bit	`uint8_t bit = PORTB & (1<< PB1)`	Read and put the value of bit PB1 of PORTB into the variable bit. This is used to read switches.

Please note that it is possible to set (or clear or toggle) multiple bits of a port. For example, the following code snippet clears bit PB0 and PB1of PORTB

```
PORTB &= ~((1<<PB0)|(1<<PB1));
```

You may notice that It's "hardcore" to do the bit shifting and negation stuff by hand, and this course's code is also written in that style because I think it's good to understand what's going on under the "hood". However, if you aren't just like me, you can also define some macros to do the same thing, and this can make your code more easily readable. If you'd like to take this path, these will do the trick:

```
#define BIT (bit) (1<<(bit)) // bit position
#define SET_BIT (port,bit) (port |=BIT(bit)) // set a bit of a port
```

```
#define CLEAR_BIT(port,bit) (port &= ~BIT(bit))  // clear a bit of a port
#define TOGGLE_BIT(port,bit) (port^= BIT(bit))  // Toggle a bit of a port
```

8.5. Registers

To effectively and efficiently program microcontrollers, one needs to learn how to manipulate individual bits in *registers*, the special memory locations that the microcontroller uses to control and carry out its operations. The three registers that one must deal with for Input/Output (IO) operations with an ATmega microcontroller are:

- Data Direction (DDRx),
- Data (PORTx), and
- Port Input Pins (PINx).

[The 'x' represents the letter enumeration scheme for the associated ports. In the particular case of the ATmega328, x could be B, C, or D]. Each of these three registers is 8-bits (1 byte) wide.

Register	Comments
DDRx	Controls data direction: writing a 1 to the associated bit location makes the corresponding pin to be an OUTPUT; writing a 0 makes the pin to be an INPUT.
PORTx	For a pin configured as an OUTPUT, the PORTx register provides the way to control the digital voltage of the pin: writing a 1 to the associated bit location drives the pin to logic HIGH; writing a 0 drives the pin to logic LOW. For a pin configured as an INPUT, the PORTx register provides the way to control the pullup resistor for the pin: writing a 1 to the associated bit turns the pullup resistor on; writing a 0 turns the pullup resistor off.
PINx	Contains the almost current 'snapshot' of the digital state of the pins in the associated port. This is the register used to 'read' the digital state of a pin.

8.6. Working with bits in registers – bit masking

The technique to access or change individual bits in registers takes some getting used to. The big idea underlying the technique is that when we work with a register, we always have to handle all 8 bits of the register together – there is no direct way to work on a single bit level. Consequently, we will use bitwise logical operators and bit *masks* to 'drill down' to an individual bit or groups of bits. A bit *mask* is a construct of 8 bits that is used in much the same way that a painter uses *masking* tape to keep sections of a surface from getting painted.

As explained in the previous section, you can use the _BV(n) macro to create a bit mask with a '1' in the bit position corresponding to the value of n. The macro for _BV(n) is a #define:

```
#define _BV(n)  (1<<(n))
```

which indicates the more direct way of building bit mask: using the bit shift left operator, **<<** .

To determine if a particular bit in a register is set, use a bit mask with a bit in the position of interest and perform a bitwise AND between the 8 bits in the register and the bit mask. If the resulting value is not zero, the bit is set, else the bit is clear. For example, to check if pin 5 of PORTD is at logic HIGH, use a bit mask with a '1' in the location for bit 5, _BV(5) or (1 << 5) and bitwise AND together with the PIND register contents. The table below illustrates what is happening, supposing bit 5 is set in PIND. (The 'Xs' indicate 'don't care' what the values are: it doesn't matter if they are '1' or '0'). Go column by column doing a bitwise AND operation between the bit in the PIND register and the bit in the bit mask. The last row in the table is the result.

bit	7	6	5	4	3	2	1	0
PIND	X	X	1	X	X	X	X	X
(1 << 5) (this is the bit mask)	0	0	1	0	0	0	0	0
PIND & (1 << 5)	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**

So, if you wanted to take a particular action if bit PD5 was set, the following test would accomplish the selection:

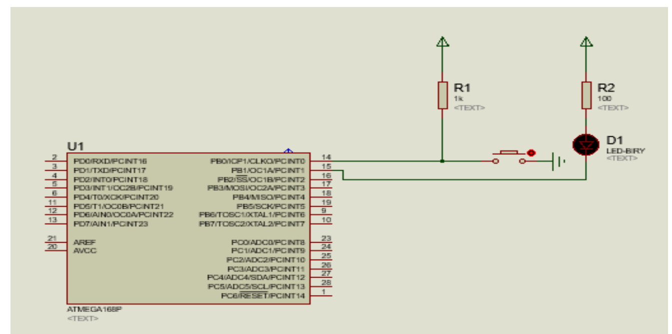
```
if( PIND & (1 << PD5) )
{
        // do stuff… ;
}
else
{
        // do other stuff… ;
}
```

Note that if bit PD5 in the PIND register were clear, the value of the result from PIND & (1 << 5) would be zero.

9. PUTTING IT ALL TOGETHER

Now we start exploring how we can combine the various bitwise operators to perform useful tasks using C++ syntax in the Arduino environment.

```
/*
 * ReadingSwitch.c
 * Created: 2/22/2015 11:19:17 AM
 * Modified 3/7/2021
 * Processor: ATmega168
 * Compiler:  AVR/GNU C Compiler
 * Author: Kizito NKURIKIYEYEZU
 * File Version: 0.3
 * Required File: None
 * Objectives: Turn ON a switch at a press of a button
 * Hardware design:
  - LED connected to PB1
  - Switch (in pull-up high) connected to PB1
 */
```




```
#include <avr/io.h>
#define PRESSED 0
#define NOT_PRESSED 1
#define LED_PIN PB1
#define SWITCH_PIN PB0
int main(void)
{
    PORTB &= ~(1<<SWITCH_PIN); // Make sure we're in the input mode
         // PORTB |= (1 << PB0); // initialize pull-up resistor on our input pin
    DDRB |= (1<<LED_PIN); // Set PB1 an output
     // Turn off the LED
    PORTB |= (1 << LED_PIN);

          while (1)
          {
                // If PB0 is not pressed
                if ((PINB & (1 << SWITCH_PIN)) == NOT_PRESSED )
                {
                        // Turn off the Led
                        PORTB |= (1<<LED_PIN); // Set PB1 to HIGH
                }
                else // If PB0 is pressed
                {
                        // Turn on the led
                        PORTB &= ~(1<<LED_PIN); // Set PB1 to LOW
                }
          }
 return 0;
}
```