Aug 14, 2012

AVR Tutorial

A Hobbyist's Guide to Hacking 8-Bit AVR Microcontrollers

< Getting Started: Blinking an LED Table of Contents

AVR C Programming Basics

In this chapter you will explore some of the key concepts of C programming for AVR microcontrollers.

3.1 AVR Registers

A **register** is a special storage space in which the state of the bits in the register have some special meaning to the AVR microcontroller. Most of the registers are 8-bits wide (there are a few exceptions).

Each register has a name and individual bits may also have unique names. All of the register and bit names are described in the ATmega48/88/168/328 datasheet. The AVR Libc library will define identifiers for the register names and bit names so that they can be easily accessed in C.

Manipulating the bits in the various registers of the AVR is the basis for AVR programming. Everything from configuring the AVR device's built-in peripherals to using the AVR's pins for digital I/O is done by manipulating bits in these registers.

3.2 Bits and Bytes

A **bit** represents one of two possible states: 1 or 0 (aka: on/off, set/clear, high/low). Several bits together represent numerical values in binary, where each bit is one binary digit. An AVR microcontroller groups 8 bits together to form one **byte** wtih the **L**east **S**ignificant **B**it (LSB) on the right. Each bit is numbered, starting from 0, at the LSB.

Consider the decimal number `15` for example. `15` represented in 8-bit binary is `00001111`. The 4 least significant bits 0, 1, 2, and 3, are set.

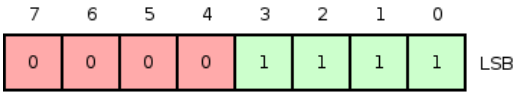
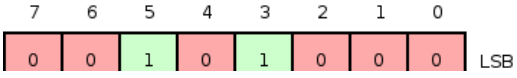


Figure 3.1 - The decimal number 15 represented in 8 bits

> If you do not already know how to convert numbers between decimal, binary, and hexadecimal, you can find numerous lessons online or use my Printable Decimal-Binary-Hex Conversion Chart

Another example, the decimal number `40` is represented in binary as `00101000`. Bits 3 and 5 are set.



Hello, my name is Micah and I write software. I love Python, GTK+, Linux, and robots.

Email Facebook Twitter

Github RSS Feed

LinkedIn Stack Overflow

Need some work done? Hire me!

SSL behind an Nginx Reverse Proxy in Django 1.4

Using libgdx without Eclipse

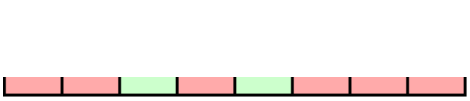


Figure 3.2 - The decimal number 40 represented in 8 bits

Numerical values in a C program for an AVR microcontroller may be defined using a decimal, hexadecimal, or binary notation depending on the context and the programmer's preference. A hexadecmial number is defined using the `0x` prefix and a binary number is defined using the `0b` prefix.

The following C code shows 3 ways in which a variable might be initialized to the decimal value of 15.

```
uint8_t a = 15;         /* decimal */
uint8_t b = 0x0F;       /* hexidecimal */
uint8_t c = 0b00001111; /* binary */
```

> The `uint8_t` data type is one of the fixed width integer types from the C99 standard. It defines an 8-bit unsigned integer. The C99 style data types will be used throughout this tutorial series.

3.3 Bitwise Operations

Since individual bits often have a significant meaning when programming AVR microcontrollers, bitwise operations are very important.

A **bitwise AND** operation results in bits being set only if the same bits are set in both of the operands. In other words: bit n will be set in the result if bit n is set in the first operand **and** the second operand.

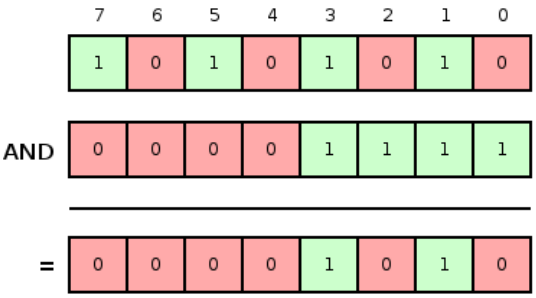


Figure 3.3 - Bitwise AND operation

A single ampersand character (`&`) is the bitwise AND operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a & b;  /* 00001010 */
```

A **bitwise OR** operation results in bits being set if the same bits are set in either of the operands. In other words: bit n will be set in the result if bit n is set in the first operand **or** the second operand.

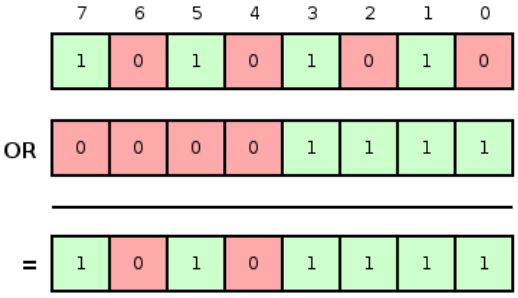


Figure 3.4 - Bitwise OR operation

A single pipe character (`|`) is the bitwise OR operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a | b;  /* 10101111 */
```

A **bitwise XOR** operation ("exclusive or") results in bits being set if, and only if, the same bit is set

in one of the operands but not the other. In other words: bit n will be set in the result if bit n is exclusively set in only one of the operands.

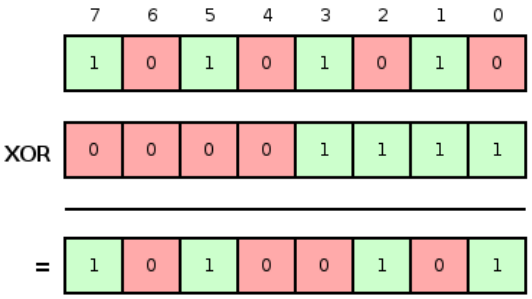


Figure 3.5 - Bitwise XOR operation

The caret character (^) is the bitwise XOR operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a ^ b;  /* 10100101 */
```

A **NOT** operation, also known as a **one's complement**, is a unary operation. That means the operation is performed on a single value instead of two. The NOT operation will simply negate each bit. Every `1` becomes `0` and every `0` becomes `1`.

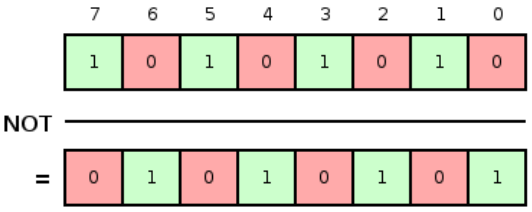


Figure 3.6 - Bitwise NOT operation

A tilde character (~) is the NOT operator in C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = ~a;   /* 01010101 */
```

A **shift** operation shifts all of the bits to the left or the right. In a left shift, bits get "shifted out" on the left and `0` bits get "shifted in" on the right. The opposite goes for a right shift.

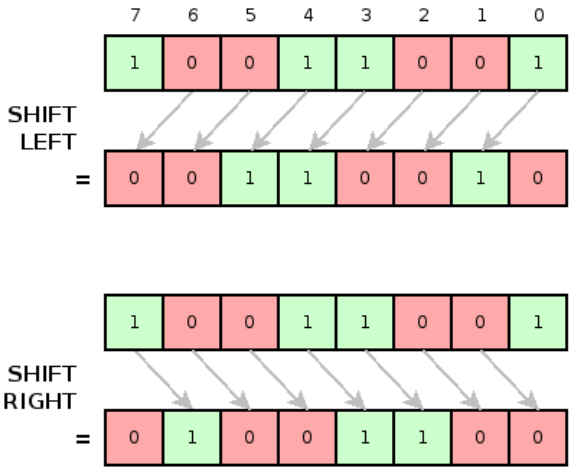


Figure 3.7 - Bitwise shift operation

Two less-than symbols (<<) is the left shift operator and two greater-than symbols (>>) is the right shift operator in C. The right side of the operator is the number of bits to shift.

```
uint8_t a = 0x99; /* 10011001 */
uint8_t b = a<<1; /* 00110010 */
uint8_t c = a>>3; /* 00010011 */
```

3.4 Clearing and Setting Bits

Setting and clearing a single bit, without changing any other bits, is a common task in AVR microcontroller programming. You will use these techniques over and over again.

When manipulating a single bit, it is often necessary to have a byte value in which only the bit of interest is set. This byte can then be used with bitwise operations to manipulate that one bit. Let's call this a **bit value mask**. For example, the bit value mask for bit 2 would be `00000100` and the bit value mask for bit 6 would be `01000000`.

Since the number `1` is represented in binary with only bit 0 set, you can get the bit value mask for a given bit by left shifting `1` by the bit number of interest. For example, to get the bit value mask for bit 2, left shift `1` by 2.

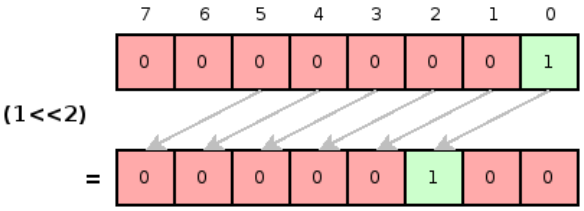


Figure 3.7 - Bit value mask

To **set a bit** in C, OR the value with the bit value mask.

```
uint8_t a = 0x08; /* 00001000 */
                  /* set bit 2 */
a |= (1<<2);      /* 00001100 */
```

Use multiple OR operators to set multiple bits.

```
uint8_t a = 0x08;   /* 00001000 */
                    /* set bits 1 and 2 */
a |= (1<<2)|(1<<1); /* 00001110 */
```

To **clear a bit** in C, NOT the bit value mask so that the bit of interest is the only bit cleared, and then AND that with the value.

```
uint8_t a = 0x0F; /* 00001111 */
                  /* clear bit 2 */
a &= ~(1<<2);     /* 00001011 */
```

Use multiple OR operators to clear multiple bits.

```
uint8_t a = 0x0F;       /* 00001111 */
                        /* clear bit 1 and 2 */
a &= ~((1<<2)|(1<<1)); /* 00001001 */
```

To **toggle a bit** in C, XOR the value with the bit value mask.

```
uint8_t a = 0x0F; /* 00001111 */
                  /* toggle bit 2 */
a ^= (1<<2);      /* 00001011 */
a ^= (1<<2);      /* 00001111 */
```

3.5 The Bit Value Macro _BV()

AVR Libc defines a the `_BV()` macro which stands for "bit value". It is a convenience macro to get the bit value mask for a given bit number. The idea is to make the code a little more readable over using a bitwise left shift. Using `_BV(n)` is functionally equivelent to using `(1<<n)`.

```
/* set bit 0 using _BV() */
a |= _BV(0);

/* set bit 0 using shift */
a |= (1<<0);
```

Which method you choose is entirely up to you. You will see both in wide use and should be comfortable seeing either. Since the `_BV()` macro is unique to [illegible] it is not as portable to other