Bit Manipulation

It is often necessary to assemble a byte to be written from parts of other variables. For example the DA converter that we are using (Microchip MCP4822) requires the 4 most significant bits (MSB) of the first byte sent to the DA to be control bits that define the way the DA operates. The next 4 bits of the first byte are the 4 most significant bits of the 12 bit data that defines the value of the DA. The second byte sent to the DA is the lower 8 bits of the 12 bit data that defines the value of the DA. See Appendix A for a description of basic bit manipulation techniques.

Since the data from the AD converter is 10 bits it must be stored in a 16 bit variable. The following code segment illustrates the method to move the proper bit segments into the variables that are written to the SPI bus.

```
unsigned char   spi_data_0;
unsigned char   spi_data_1;
unsigned int    adc_output;

    adc_output = ADCW;                          // Read AD value

    spi_data_0 = 0x00;                          // Zero spi_data_0
    spi_data_0 = (adc_output & 0x0F00) >> 8;    // Set up the first byte to write by mapping bits 8-11
                                                // to the lower 4 bit positions and
    spi_data_0 = spi_data_0 + 0b00110000;       // Adding the upper 4 DA control bits
    spi_data_1 = (adc_output & 0xFF);           // Set up the second byte to write by mapping
                                                // bits 0-7  to the lower 8 bit positions

    cbi(PORTD,7);                               // Activate the chip - set chip select to zero
    dummy_read = spi_write_read(spi_data_0);    // Write/Read first byte
    dummy_read = spi_write_read(spi_data_1);    // Write/Read second byte
    sbi(PORTD,7);                               // Release the chip  - set chip select to one
```

The following explains each code segment.

These statements declare the variables that are used. The unsigned char designation declares n 8 bit variable whose value is 0 – 255. The unsigned int designation declares an 16 bit variable whose value is 0 – 65525.

```
unsigned char   spi_data_0;
unsigned char   spi_data_1;
unsigned int    adc_output;
```

This statement reads the 10 bit AD value into the 16 bit variable adc_output.

```
    adc_output = ADCW;                          // Read AD value
```

The following statements set up the two bytes that are written to the DA converter.

```
spi_data_0 = 0x00;                            // Zero spi_data_0
spi_data_0 = (adc_output & 0x0F00) >> 8;      // Set up the first byte to write by mapping bits 8-11
                                              // to the lower 4 bit positions and
spi_data_0 = spi_data_0 + 0b00110000;         // Adding the upper 4 AD control bits
spi_data_1 = (adc_output & 0xFF);             // Set up the second byte to write by mapping
```

The first statement zeros spi_data_0.

```
spi_data_0 = 0x00;                            // Zero spi_data_0
```

The 0x00 is hexadecimal notation and is equivalent to 0b00000000 but is more compact. Each of the numbers in hex notation corresponds to 4 bits in binary notation.

A lot is happening in this next line.

```
spi_data_0 = (adc_output & 0x0F00) >> 8;      // Set up the first byte to write by mapping bits 8-11
                                              // to the lower 4 bit positions and
```

Let's start with the term inside the parentheses.

adc_output & 0x0F00

The hex term **0x0F00** is equal to 0b0000111100000000 and is used to mask or to isolate the 4 bits of interest in adc_output. If adc_output is represented symbolically as adc_output = **0bxxxxxxxxxxxxxxxx**

adc_output & 0x0F00 = 0bxxxxxxxxxxxxxxxx & 0b0000111100000000 = 0b0000xxxx00000000

So the statement above sets all of the bits in adc_output to zero except the 4 bits of interest.

The last part of the statement >>**8** simply shifts the bits 8 places to the right.

(adc_output & 0x0F00) >> 8 = (0b0000xxxx00000000) >> 8 = 0b000000000000xxxx

So the end result of the complete statement **spi_data_0 = (adc_output & 0x0F00) >> 8** is to mask bits 8 to 11 of adc_output , shift those bits 8 places to the right and set the result to **spi_data_0**. So this statement places the 4 most significant bits (upper 4 bits) of the 12 bit DA data in the proper location in **spi_data_0**.

The next statement defines the upper 4 bits of **spi_data_0** by adding the 4 DA control bits to the lower 4 bits of **spi_data_0** defined above.

```
spi_data_0 = spi_data_0 + 0b00110000;         // Adding the upper 4 DA control bits
```

The final statement places the lower 8 bits of adc_output into **spi_data_1**.

```
spi_data_1 = (adc_output & 0xFF);             // Set up the second byte to write by mapping
                                              // bits 0-7 to the lower 8 bit positions
```

The following statements activate the DA chip, write the two bytes of data and release the DA chip.

```
cbi(PORTD,7);                                // Activate the chip - set chip select to zero
dummy_read = spi_write_read(spi_data_0); // Write/Read first byte
dummy_read = spi_write_read(spi_data_1); // Write/Read second byte
sbi(PORTD,7);                                // Release the chip  - set chip select to one
```

Appendix A: Basic Bit Manipulation

A good tutorial on bit manipulation is:
http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=37871&highlight

The following summarizes the most important aspects of bit manipulation.

The C operators that allow bit manipulation are shown below:

```
| bit OR
& bit AND
~ bit NOT
^ bit EXLUSIVE OR (XOR)
<< bit LEFT SHIFT
>> bit RIGHT SHIFT
```

The following is a truth table that shows the operation of OR, AND and XOR operators.

Operator Truth Table

OR operation	AND operation	XOR operation
0 OR 0 = 0	0 AND 0 = 0	0 XOR 0 = 0
0 OR 1 = 1	0 AND 1 = 0	0 XOR 1 = 1
1 OR 0 = 1	1 AND 0 = 0	1 XOR 0 = 1
1 OR 1 = 1	1 AND 1 = 1	1 XOR 1 = 0

The NOT operator reverses the bit, so a 1 becomes a 0, and a 0 becomes a 1. The bit left shift (<<) and bit right shift (>>) shift the bits in the variable left or right. Zeros are used to fill the bit locations that are vacated when the shift operation takes place.

Appendix B: Links

Microchip MCP4822 (http://ww1.microchip.com/downloads/en/DeviceDoc/21953a.pdf).