Simple Masking Technique

by David Aldersley June 2010

Abstract

Not all articles have to be big and flashy, some are there to explain the fundamentals. Without Newton's theory of mechanics there would be no Einstein's theory of relativity. So prepare to learn the basics of simple masking, and like Einstein in his field of physics, you too will stand on the shoulders of giants to embark on your journey to embedded greatness.

Requirements

Software:

- No software requirements

Hardware:

- No hardware requirements

Using Simple Masking Technique to Send Data Over a Serial Connection

Sometimes it is not possible to work with large binary numbers 16bit long integers or larger, a simple solution for this is to split these larger numbers down into individual bytes to allow them to be manipulated in more ways.

One usage for this is for sending data over RS232 where it sends these bytes of data individually. So to send a large number we would need to break the number down into individual bytes and send each of these separately, then we can bring these bytes back together to get our original long integer back.

To break down our long integers we can use a simple masking technique to separate the two bytes from each other. For example, if we consider the decimal number 1717, this number expressed in binary becomes:

00000110 10110101

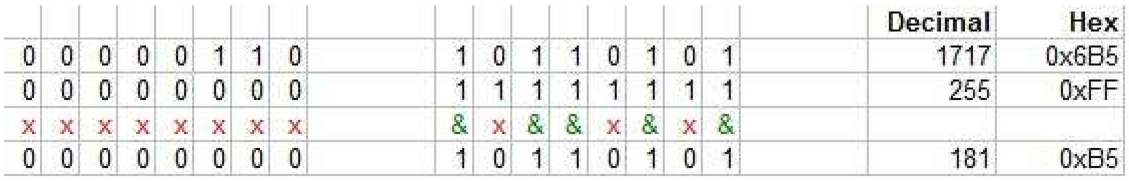
The way to break this long integer down would be into two individual bytes:

00000110 - High byte
10110101 - Low byte

To separate these bytes we use a logical '&' to 'mask' out either the higher byte or the lower byte along with 0xFF which is hexadecimal for 255 decimal or 11111111 binary.

To separate the lower byte from 1717:

low_byte = 1717 & 0xFF

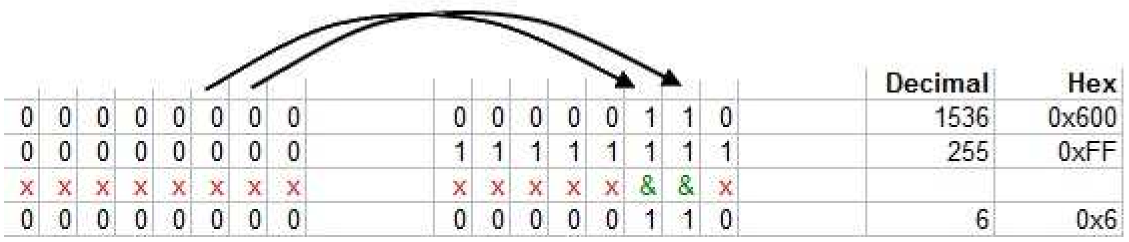


																Decimal	Hex
0	0	0	0	0	1	1	0	1	0	1	1	0	1	0	1	1717	0x6B5
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	255	0xFF
x	x	x	x	x	x	x	x	&	x	&	&	x	&	x	&		
0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	1	181	0xB5

The above table shows the first byte being “masked” off, as the ‘&’ operator only allows matching ‘1s’ through and masks off everything else.

To separate the higher byte from 1717 we need to shift the bits down to where the lower byte was located, to do this we use the ‘>>’ operator to shift the bits down.

high_byte = (1717 >> 8) & 0xFF



																Decimal	Hex
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1536	0x600
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	255	0xFF
x	x	x	x	x	x	x	x	x	x	x	x	x	&	&	x		
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	6	0x6

This leaves us with two variables:

low_byte = 0xB5
high_byte = 0x6

We can now combine these two back together in a similar way using another bitwise logical operator. Using the OR (**|**) operator we can add these two bytes together to create our original integer.

To start recombining the two bytes we must make sure our high byte is moved back to the first eight bits of the 16 bit integer. To do this we use a similar operator to before but pointing the opposite way to move the bits higher rather than lower ‘<<’. (We will use **Integer** as a temporary variable to allow calculation).

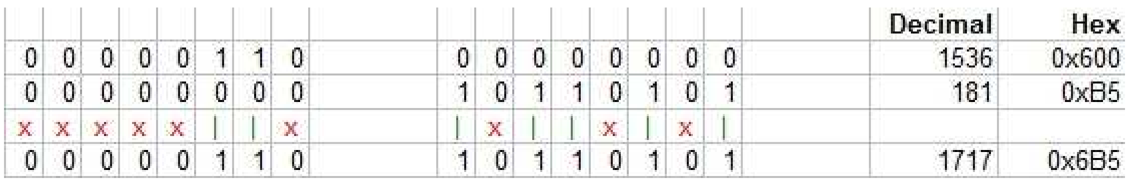
Integer = **high_byte** << 8

Now we set Integer to be the value of high_byte but shifted 8 bit places higher.

Integer = 00000110 00000000

Lastly we need to turn those last eight zeros (the lower byte) into the same value stored in our low_byte variable, to do this we use the OR operator.

Integer = **Integer | low_byte**



																Decimal	Hex
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1536	0x600
0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	1	181	0xB5
x	x	x	x	x	|	|	x	|	x	|	|	x	|	x	|		
0	0	0	0	0	1	1	0	1	0	1	1	0	1	0	1	1717	0x6B5

As can be seen from the table above, after using the OR operator and combining the values we had in **Integer** and **low_byte** the total we come out with is our original value of 1717. The OR operator allows creates a 1 value if either of the other two binary numbers have a 1 in that position, thus we can combine our two binary values to recreate our original long integer.

Flowcode Example:

Sending data over a serial port requires the data to be sent in individual bytes, therefore if we have an integer we want to send which is greater than 255 (max decimal number for one byte) then the method above can be used for this purpose.

In the example Flowcode files I am using two RS232 boards to send data to each other and outputting the result onto an LCD attached to Board A. The input is from a potentiometer which I am gathering an integer from the 0~1000 range which is attached to Board B.

To send this integer I am breaking it down into two separate bytes and sending each across the serial cable individually and then combining them back together again using the methodology explained above.

Equipment Used:

- 2x EB006 Programmer boards
- 1x EB003 Sensor board
- 2x EB015 RS232 board
- 1x EB005 LCD board
- 1x Serial cable

Next Step…

This same method could be used to deal with 32bit integers as well:

```
byte0 = ((longInt >> 24) & 0xFF) ;

byte1 = ((longInt >> 16) & 0xFF) ;

byte2 = ((longInt >> 8 ) & 0XFF);

byte3 = (longInt & 0XFF);
```

This would separate the 32bit integer into the 4 separate bytes, it would then be possible rebuild them again in a similar way to the 16bit integer. To do this in Flowcode however you would need to edit the C code manually since 32bit integers are not natively supported.

Further reading

Below are some links to other resources and articles on related subjects, and technical documentation relating to the hardware used for this project...

Flowcode:	http://www.matrixmultimedia.com/flowcode.php
Learning Centre:	http://www.matrixmultimedia.com/lc_index.php
User Forums:	http://www.matrixmultimedia.com/mmforums
Product Support:	http://www.matrixmultimedia.com/sup_menu.php