Arduino Byte Juggling, Version 1.3

By R. G. Sparber

Protected by Creative Commons.[1]

I call this one of my “lightning rod” articles. I state what I have learned and then throw it out there. Invariably, one or more people with a lot more knowledge come forward with an approach far better than I found. That is great and I then update the article to reflect the new found insights along with the proper credits. All benefit.

So far, two experts have stepped forward to expand on this topic. Thanks to Dave Kellogg and John Dammeyer for their help.

After writing this article and before publishing it, I stumbled into

https://playground.arduino.cc/Code/EEPROMReadWriteLong

which explains how to use bit shifting to accomplish the same tasks. They are dealing with the internal EEPROM but the restriction is the same: reading and writing single bytes. Of special interest to me is how they bit AND with varying lengths of all ones to define the size of the resulting variable.

Using the key words “long” and “EEPROM” you will find others discussing this issue. I’m sure there is one that uses the technique presented here but I didn’t find it.

A Few Data Formats

There are many data formats available for programming Arduinos. We can define a single byte, two bytes, or four bytes[2].

English	Arduino	bytes used	range of numbers it can hold
Byte	byte	1 byte	0 to 255
Integer	int	2 bytes	-32,768 to 32,767
Long	long	4 bytes	-2,147,483,648 to 2,147,483,647

Things get a little sticky when we want to store variables in an external EEPROM. I am using the 24LC1025 which can hold 128KB of data. It expects to deal with bytes or contiguous blocks of bytes. I chose to only use the single byte option.

A Byte

To write/read a single byte to/from the EEPROM, I used the sample code available on SparkFun[3]. Simple enough when your variables are all single bytes.

WriteEEPROM(an address, ***n***);

n = ReadEEPROM(an address);

Where ***n*** is a byte.

[2] There are also unsigned integers and unsigned longs but they use the same number of bytes as integers and longs. Floats use four bytes.

[3]See https://learn.sparkfun.com/tutorials/reading-and-writing-serial-eeproms

An Integer

When needing to store an integer, we must deal with two bytes. To do a write, we can employ the built-in functions lowByte(***m***) and highByte(***m***).

WriteEEPROM(an address, lowByte(***m***));
WriteEEPROM(an address + 1, highByte(***m***));

Where ***m*** is an integer (2 bytes).

This code will take the least significant byte (LSB) of the two byte variable called ***m*** and save it to "an address". The next line will save the most significant byte (MSB) of this variable to the next higher address. So it wasn't that hard to disassemble a two byte number and store it as separate bytes.

To do a read is a little trickier. There is no reverse of lowByte/highByte. We can get the bytes back but need to assemble them into an integer.

Byte0 = ReadEEPROM(an address);
Byte1 = ReadEEPROM(an address + 1);

Say ***m*** = 12345 and has been defined as an integer. In hexadecimal (hex) it would be 0x3039. This means that Byte1 holds 0x30 and Byte0 holds 0x39. If "an address" is 42, we will see 0x39 in address 42 and 0x30 in address 43.

To reassemble ***m*** I must fetch these bytes and put them in the correct order:

m = 256*int(Byte1) + int(Byte0); //first convert each byte to integer

Multiplying by 256 moves the LSB into the MSB position.

Going back to our example, given
Byte 1 = 0x30 which equals decimal 48
Byte 0 = 0x39 which equals decimal 57

m = (256 * Byte1) + (Byte0)

m = (256 * 48) + 57

m = 12345 which is what we put into memory in the first place.

A Long

Now that we know how to read and write integers to EEPROM, we can consider Longs which are four bytes.

Say ***p*** is a long.

To do a write, we can employ the built-in functions lowByte(***p***) and extract the LSB:
WriteEEPROM(an address, lowByte(***p***));

In fact, you can even use highByte(***p***) to extract the byte just above the LSB

WriteEEPROM(an address + 1, highByte(***p***));

But then what? I have two more bytes to write and there isn't a higherByte() and highestByte() functions.

The technique I found is to use a combination of lowByte() calls and bit shifts.

WriteEEPROM(an address, lowByte(***p***));
WriteEEPROM(an address + 1, lowByte(***p>>8***));
WriteEEPROM(an address + 2, lowByte(***p>>16***));
WriteEEPROM(an address +3, lowByte(***p>>24***));

I'm taking the long, ***p***, and shifting it right by the specified number of bit positions. Then I chop off the low byte. The most significant bit is replicated[4] as I shift right but this is harmless because I never see it.

I could have alternately written:

WriteEEPROM(an address, (***p & 0xFF***));
WriteEEPROM(an address + 1, (***p>>8***) ***& 0xFF***);
WriteEEPROM(an address + 2, (***p>>16***) ***& 0xFF***);
WriteEEPROM(an address +3, (***p>>24***) ***& 0xFF***);

The bit ANDing with 0xFF[5] zeros out the upper bytes and leaves the LSB.

[4] This is called sign extension.
[5] 0xFF is 0x00FF which is 0000 0000 1111 1111 in binary. Anding with this number zeros out the top byte.

Recall that to read an integer we used

int m; //define n as an integer
Byte0 = ReadEEPROM(an address); //get the LSB
Byte1 = ReadEEPROM(an address + 1);//get the MSB

m = 256*int(Byte1) + int(Byte0);//assembly the integer

You might think that reading a long ***should*** be an extension of what worked for integers:

long p; //define p as a long
Byte0 = ReadEEPROM(an address); //get the LSB
Byte1 = ReadEEPROM(an address + 1);//get the second byte
Byte2 = ReadEEPROM(an address + 2);//get the third byte
Byte3 = ReadEEPROM(an address + 3);//get the MSB

p = 256*256*256*long(Byte3) + 256*256*long(Byte2) + 256*long(Byte1) + long(Byte0);//assemble the long

Nope. Don't work.

The reason is subtle. By default, arithmetic is done using what is called the "C Operator Precedence Table". It says that we do multiplications first and start at the left end of each group of multiplication. A second set of rules called the "Usual Arithmetic Conversions" tells us how the compiler deals with different data types.

p = 256*256*256*long(Byte3) + 256*256*long(Byte2) + 256*long(Byte1) + long(Byte0);//assemble the long

The compiler looks at the first two numbers in the first multiplication:

256*256*256*long(Byte3)

Since they are not marked, it is assumed they are both integers. But the result is larger than what an integer can hold so the result is 0. That zeros out the first term.

Next, we have

256*256*long(Byte2)

Again we multiply the integer 256 by the integer 256 and the result is too large to be held in an integer. We get 0 instead. So this zeros out the second term.

Then we have

256*long(Byte1)

The compiler sees that one of these values is a long so coverts 256 to a long and the result is a long. Plenty of room for the result.

And the last term. long(Byte0), is just a long so is fine.

The solution is to tell the compiler you need to do all of the arithmetic using longs. This is done by defining the first constant in each of the two large products as a long. For example,

256L*256*256*long(Byte3)

When the compiler sees 256L*256, the "Usual Arithmetic Conversions" rules say to change the second number to a long and the result is a long. Then we have the product of the first two numbers stored as a long times the third 256. Since we are multiplying by a long, the rule is to convert 256 into a long before multiplying. Then when we multiply the resulting constant, which is a long, by long(Byte3), the result is a long.

This same logic applies to the next term

256L*256*long(Byte3)

Since each of the multiplication results is a long, the sum must be a long too[6]. The result is assigned to ***p*** which was previously defined as a long.

We end up with

p = 256L*256*256*long(Byte3) + 256L*256*long(Byte2) + 256*long(Byte1) + long(Byte0);

[6] As per the Usual Arithmetic Conversions document, as long as one of the two numbers being added is a long, the other number will be converted to a long and the result will be a long.

More Than One Variable Occupying the Same Space

John Dammeyer told me about the ***union***[7]. This approach enables me write a long and turn around and read each of the bytes separately. Conversely, it lets me write each of the bytes separately and then read a long made up of those bytes.

Say I have

```
unsigned long LongPort;
```

This defines 4 bytes to be accessed when the program wants to read or write the variable LongPort.

I can also have

```
byte BytePort[4];
```

This defines 4 bytes to be accessed individually as the elements of an array. So if I wrote

```
unsigned long LongPort;
byte BytePort[4];
```

data I assigned to LongPort would have nothing to do with data assigned to the BytePort[] array.

However, I can get LongPort and BytePort[ ] to occupy the same memory if I write

```
union{

  unsigned long LongPort;
  byte BytePort[4];

} TwoWay;
```

union tells the compiler that we want to assign different variables to the same memory space. After the { we have a list of the variables that point to the same place. In this example, I have a long which is 4 bytes and my array is also 4 bytes

[7] See http://www.cplusplus.com/doc/tutorial/other_data_types/ For a full explanation.

so the sizes match. In general, the amount of memory set aside will equal the size of the largest variable. So if I had

```
unsigned long LongPort;
byte BytePort;
```

BytePort would point to the LSB of the LongPort variable.

After } we have the name of the objects, TwoWay. This name will precede each variable and have a "." after it.

For example, I can populate my BytePort array with

```
TwoWay.BytePort[0] = 0x00; //LSB
TwoWay.BytePort[1] = 0x01;
TwoWay.BytePort[2] = 0x02;
TwoWay.BytePort[3] = 0x03; //MSB
```

Then I can access LongPort by calling out TwoWay. LongPort. It will contain 0x03020100.

Since the variables LongPort and BytePort[4] were defined within the union{} they are local variable. As you can see above, we must have the object's name, TwoWay with a "." in front of each variable to use it globally. This also means that I can have another union{} with the same variable names but with a different object name and there won't be a conflict.

I can also go from writing a long and reading its bytes. For example, say I write

```
TwoWay.LongPort = 2881249614; //in hex it is 0xABBC614E
```

and then access each of the bytes that makes it up:

```
Serial.println( TwoWay.BytePort[3] ); //0xAB
Serial.println( TwoWay.BytePort[2] ); //0xBC
Serial.println( TwoWay.BytePort[1] );//  0x61
Serial.println( TwoWay.BytePort[0] ); // 0x4E
```

This will print out the decimal equivalents of 0xAB, 0xBC, 0x61, and 0x4E.

```
171
188
97
78
```

As a final check, let's put TwoWay.LongPort back together:

```
171*(256*256*256) + 188*(256*256) + 97*(256) + 78 = 2881249614
```

A word of warning on unions. They do not work the same way across all types of computing environments[8]. As long as you only use them in your Arduino programming, all should be fine.

[8] See https://en.wikipedia.org/wiki/Endianness for how the order of the bytes can vary.

Acknowledgments

Thanks to Dave Kellogg for pointing me to the “C Operator Precedence Table” and the “Usual Arithmetic Conversions” set of rules.

Thanks to John Dammeyer for sharing the unique method of performing this task.

I welcome your comments and questions.

If you wish to be contacted each time I publish an article, email me with just "Article Alias" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org