Episode 7.04 – Setting Bits using the Bitwise-OR

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series, we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. This episode has direct consequences for our code. You can find coding examples on the episode worksheet, a link to which can be found on the transcript page at intermation.com.

Individual bits within a binary value can be set to one using the bitwise logical OR. To do this, OR the original value with a binary mask that has ones in the positions to be set and zeros in the positions to be left alone. It's as simple as that. For example, we can set all four of the least significant bits of an integer, bits 3, 2, 1, and 0, by performing a bitwise-OR of the integer with the binary mask 1111, which in hexadecimal is 0xf. Since the bitwise-OR operator is the vertical pipe, sometimes referred to as the vertical bar, the expression to do this would be integerValue = integerValue | 0xf.
Let's assume that we are writing software to control the lighting in a small auditorium. There are eight banks of lights, each of which is controlled by the value contained in one of the bits of a byte named lightingControl. The bit assignments are as follows:

```
Bit 7 controls the house lighting
Bit 6 controls the work lighting
Bit 5 controls the aisle lighting
Bit 4 controls the exit lighting
Bit 3 controls the emergency lighting
Bit 2 controls the stage lighting
Bit 1 controls the orchestra pit lighting
Bit 0 controls the curtain lighting
```

For example, if the house lighting, exit lighting, and stage lighting are all on, the value of the lightingControl should be 10010100 in binary. The challenge meant to be handled with the bitwise-OR is to turn on lights without turning off lights that are already on. For example, we don't want to turn on the orchestra pit lighting by assigning the binary value 00000010 to the lightingControl variable. That would turn off all the other lights including exit lighting.

So what we could do is create eight constants to use as bit masks to turn on the lighting. In decimal, these masks could be set like this:

```
const houseLightingMask = 128;
const workLightingMask = 64;
const aisleLightingMask = 32;
const exitLightingMask = 16;
const emergencyLightingMask = 8;
const stageLightingMask = 4;
const orchestraPitLightingMask = 2;
const curtainLightingMask = 1;
```

Now if we wanted to turn on the aisle lighting and the emergency lighting, it would be a matter of performing a bitwise-OR of the lightingControl variable with the aisleLightingMask and with the emergencyLightingMask. This could be done in a single line of code:

```
lightingControl |= (aisleLightingMask | emergencyLightingMask);
```

We could turn off the lights by clearing the bits with a bitwise-AND. In that case, our bitmasks would be the binary inverse of the masks we're using to turn the lights on. We will save that for our next episode where we invert bits using the bitwise exclusive-OR and the bitwise inverse.

Incidentally, numerous functions available in libraries from many programming languages, give us the opportunity to turn on and off features using the bitwise-OR. Predefined constants exist in the libraries, and using them to turn on and off functions is merely a matter of passing a parameter with each of these flags OR-ed together.

The bitwise-OR is perfect if we want to build an integer from the ground up. Let's try something simple, building the 24-bit value to represent the web color gray, which uses 50% levels for red, green, and blue. Fifty percent of the way from 0 to 255 is about 128. Combining the bitwise-OR with the left shift operator, which is two less-than operators next to each other, we get the code to create gray from scratch.

```
fiftyPercentGray = (128 << 16) | (128 << 8) | 128.
```

If you were to print this result in hexadecimal, you would get 0x808080, the 24-bit representation for gray.

We could use a similar tactic to combine the red, green, and blue components into a single 24-bit value. Let's say we have a value between 0 and 255 representing red, another value between 0 and 255 representing green, and a third value between 0 and 255 representing blue. The bitwise-OR can be used to create the 24-bit color with the code:

```
color = (red << 16) | (green << 8) | blue.
```

Yes, I know that this could have been done just as easily with addition, but remember, the bitwise-OR is about setting the bits, not adding values. Writing our code this way makes it so that our code is self-documenting. In other words, the shifts and the bitwise-ORs make it clear how we are building this integer. Additionally, older microprocessors can perform bitwise operations slightly faster than addition, so the code using the shifts and the bitwise-ORs would have been a little faster. Advances in computer architecture, however, have sped up addition so that this is no longer the case.

Let's do one last example. In Episode 7.03 – Coding for Bitwise Operations, we performed a bitwise-AND of an IPv4 address with its subnet mask in order to reveal its subnet. Let's do some bit manipulation on IPv4 addresses by creating the IPv4 address and its corresponding subnet mask using bitwise-ORs. Remember that both the IPv4 address and the subnet mask consist of four integers between 0 and 255 separated by dots. If a user has entered these values as separate integers, we need to assemble them into the 32-bit values used by the network. Let's start with the IPv4 address. Assume that the four integers are passed to our code in the variables addressByte1, addressByte2, addressByte3, and addressByte4, where addressByte1 is the most significant byte.

In order to avoid creating a bad address, we might want to check to see if any of these bytes is greater than 255 or less than 0. This can be done with a simple if-statement, or we could simply perform a bitwise-AND of each byte with the hexadecimal value 0xff. That would clear any bits beyond the valid range.

Now that we've protected ourselves from values outside of our range, let's build our IPv4 address using the bitwise-OR.

```
address = (addressByte1 << 24) | (addressByte2 << 16) | (addressByte3 << 8) |
addressByte4;
```

The subnet mask can be built using the same operations.

```
subnetMask = (maskByte1 << 24) | (maskByte2 << 16) | (maskByte3 << 8) |
maskByte4;
```

In our next episode, we will take a look at inverting bits and the code we'd use to do that. For episode transcripts, worksheets, links, or other podcast notes, please visit us at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages. Until the next episode, remember that while the scope of what makes a computer is immense, it's all just ones and zeros.