Embedded Software For Non-Embedded Scientists

Test Driven Development is a useful process for all C developers, not just those creating embedded software. Ceedling, Unity, and CMock are perfectly suitable for any C development. Clearly, the focus of our course is to teach the process and tools to embedded software developers, but we'd like to make sure that it's useful to any C developer. This document gives you a brief overview of what makes embedded software development unique, so hopefully the classes' examples will make more sense to you.

Memory

Embedded software developers tend to focus on optimization more than other C developers. They are frequently working on microcontrollers that have much less memory than your average desktop PC. In fact, often the program itself isn't even running from RAM, as it would in Linux, macOS, or Windows. Many microcontrollers will run the application directly from onboard flash, allowing them to save their limited RAM resources for variables and data.

The processor cores running these types of applications will also have more constraints than working with a processor designed for a desktop. For example, the ARM series of cores require that if you're going to access memory as a 32-bit int, the address (or location) of that int must start at an address divisible by the size of the int (4 bytes). So 0 or 4 are valid addresses for an int to be stored, but 1, 2, and 3 are not. These constraints are mostly handled by the compiler for you, but they will make their presence known is surprising ways.

Let's look at just one example. What if we had the following struct defined:

```
struct TOYBOX {
    uint8_t num_toys;
    uint32_t toy_ids[4];
};
```

Depending on the compiler settings, this struct may be compiled to become 17 bytes long, or 20 bytes long. In either case, the toy_id's likely take up 16 bytes (4 elements of 4 bytes each). Our num_toys field only takes up one byte, so the obvious answer would be that the entire struct takes up 17 bytes… but toy_ids are each 4-byte ints, meaning that an ARM processor would want it to be aligned to a 4-byte address. A new struct always starts on a 4-byte alignment as well, so the compiler will often insert three unused bytes between our fields, just to make the alignment work out. This has no effect on our normal operation of the struct (they're unused!) but if we compare the memory of two "identical" structs, they may FAIL because the padding bytes may not actually contain the same data!

Often there is an option to "pack" a struct, which forces the struct to not insert this sort of extra padding. Again, the compiler will handle the details for us, but there is a hidden price. First, when we read our unaligned data, the compiler needs to translate this into multiple aligned reads, which it then assembles into the number we want. Second, we can't just cast one of these fields to a pointer, because the pointer would be pointing at an unaligned memory address. Something like this will throw an ARM Exception:

```
TOYBOX toybox = { 2, { 123, 234 } };
uint32_t* ptr = (uint32_t*)toybox.toy_ids;
*ptr = 1234; //Boom!
```

Speed

The microcontrollers of the embedded world are also significantly SLOWER than those used in the desktop or server world. This means that the developers care a LOT about the efficiency of their algorithms. They tend to focus on code with less layers, which in some ways works against the goal of having a healthy separation of concerns. To make this happen, embedded software developers often make significant use of macros and/or inlines. This allows the code to be written in a portable layered way, but removes the overhead of additional function calls. We discuss in this course how to overcome this challenge… we just wanted to mention it here so that we all understand the reason that such code practices exist is valid.

Registers

To really understand embedded software, you need to familiarize yourself with the concept of a register. Registers are special locations in memory that serve a particular purpose. They are mapped into the processor's memory just like RAM… but instead of being able to freely read and write to the location for any reason we wish, each register address serves a very specific function.

Some registers are only readable. Any attempt to write to the location will be ignored. These readable registers might be the status of a subsystem (like a USB connection for example), the result of some operation (like reading an analog input), or be a queued result (like the next character from a serial connection).

Other registers are only writable. An attempt to read from them might return nothing but zeros, but a write might trigger an operation to happen.

While most registers are readable and writable, this doesn't make them necessarily straightforward. Many registers are a collection of bits which do different things. Some registers allow you to set the state of all of those bits, while others may only allow the programmer to set a bit to 1 if it's 0, but then the subsystem itself must change it back when an operation has completed. Often a programmer must read the current state, change only the things they want changed, and then write the register again. There are even specialty registers which will set or clear bits in other registers.

There are way too many combinations to cover here. To get an idea of what kinds of things you might encounter, embedded software developers refer to a microcontroller's programming guide. It will contain the full list of registers (collected together in "peripherals") for that micro. You can find the programming manual for the Freescale K20 processor (the ARM Cortex processor we target in this class) in the supplemental materials.

No matter what microcontroller is being targeted, though, working with registers is likely going to require that you brush up on your bit operations in C.

Bit Operations

Being a low-level programming language, C has a full set of bitwise operators which allow us to manipulate data with full control. A quick search of the internet will give you all sorts of tutorials on how to work with bitwise operators, but let's look at a short version here.

Let's start by talking about how integers are actually stored. We're going to work with single byte integers (uint8_t) but the concept applies equally well to other sizes. Most modern computers like to think about numbers in binary. That means that instead of 10 digits (0-9) for each place like we do in modern mathematics, it uses only 2 digits (0-1) for each place. Remember how you "carry the one" when you're adding numbers? It still works like that!

```
Decimal   Hex      Binary

  1       0x01       0000 0001
 +1       0x01     + 0000 0001
 ---      ----     -----------
  2       0x02       0000 0010
```

On the right, we have the binary operation. We add 1 + 1 and it wants to become 2. The number 2 is bigger than the 1's column can hold, so it becomes 0 and we carry the one over to the 2's column. The columns each are a doubling of the one to the right of it… so in a single byte there are columns 1, 2, 4, 8, 16, 32, 64, and 128.

The column in the middle is hexadecimal. Just to make it extra fun, embedded C developers use hex often. It's much more concise than writing binary, but it still supports the column structure and is easier to translate to decimal at a glance. Hex takes each group of 4 bits and converts them to a single number… Since that number can now be between 0-15, we use the digits 0-9, and then fill in with A-F to be 10-15.

Confused yet? If so, maybe play around with a few numbers and see if you can get the hang of it… then come back. We're ready to start talking about bitwise operators.

It's important to know that bitwise operators work on EACH BIT individually in a number. So while || might become true if either of the values are non-zero, | will only be true for either *bit* of the corresponding number is true. Let's look at an example:

```
Decimal    Hex        Binary

 69          0x45       0100 0101
| 7        | 0x07     | 0000 0111
---        ------     -----------
 71          0x47       0100 0111
```

Notice that each column in the binary example becomes 1 if EITHER of the bits above it are 1. This is how bitwise-OR works (|). This operator is really useful to us. If we want to turn a single bit ON without touching the others, we just need to bitwise-OR the current value with the bit or bits we want enabled. In our example above, 0x45 might have been the existing value and 0x07 is what we used to make sure that the bottom three bits are set.

Similarly, while && might become true if the value on both sides is non-zero, & will only be true for each bit if the corresponding bit is true in both. Let's look at an example:

```
Decimal    Hex        Binary

 69          0x45       0100 0101
& 7        & 0x07     & 0000 0111
---        ------     -----------
  5          0x05       0000 0101
```

Notice that each column in the binary example only becomes 1 if BOTH of the bits above it are 1. This is how bitwise-AND works (&).

To see how bitwise-AND is truly useful, we first want to learn about one other operator, the bitwise-NOT operator (~). This operator is a unary operator, meaning it works on a single number at a time.

```
Decimal    Hex       Binary

~ 7        ~0x07     ~ 0000 0111
---        -----     -----------
  5         0xF8       1111 1000
```

Notice that the final number has the opposite of EACH BIT of the original number. If the original bit was 0, it is now 1, and vice versa.

So how is this useful to us?

If we combine the bitwise-AND with the bitwise-NOT operator, we gain the ability to disable (or set to zero) certain bits. Let's say we want to disable the 4th bit of a variable "a":

```
a = (a & ~(0x08) );
a &= ~0x08;
```

The two lines above are equivalent. Either performs this operation:

```
Decimal    Hex       Binary

  79         0x4F      0100 1111
& ~8       &~0x08    &~0000 1000
----       ------    -----------
  79         0x4F      0100 1111
&247       & 0xF7    & 1111 0111 // negative first
----       ------    -----------
  71         0x47      0100 0111 // then AND
```

So that's useful! There are a couple more things worth investigating before we are done with this discussion.

First, let's talk about the XOR (or *exclusive-OR*). This is similar to our bitwise-OR operation, except that the value for each bit will only be 1 if the original pair don't match (one is 1 and one is 0).

```
Decimal    Hex        Binary

  69         0x45       0100 0101
^  7       ^ 0x07     ^ 0000 0111
---        ------     -----------
  66         0x42       0100 0010
```

Notice that the only bits that are 1 in the bottom had both a 1 and a 0 above them. So how is this useful to us? Let's consider it a different way. If the top row is our starting value, notice that when the second row contains a zero, the original value passes through unchanged. When the second row contains a one, the original value is inverted. So our XOR operator has provided us a means to give a set of bits as 1's that we want to toggle to their opposite! How handy is that?

Using our OR, AND-NOT, and XOR operators this way, the second number is commonly referred to as a bit mask in embedded-speak. Depending on the operation, it is a mask of bits that we either want to set (turn to 1 using the OR), clear (turn to 0 using AND NOT), or toggle (using the XOR).

Often we will create these masks manually as binary or hex representations, as we have been above… but there are also times where it makes sense to refer to the bits as number 0, 1, 2...n, where 'n' is the number of bits in the integer. When done this way, it is sometimes useful to use the shift operator. Let's say I want to SET bit 7 in a register CHEEZ:

```
CHEEZ |= (1u << 7);
```

This starts with the number 1, which is really just a number where only the lowest bit is set. The shift operator << shifts this number 7 bits to the left, giving us the 7th bit (assuming the lowest is called 0). That gives us a mask of 0x80 or 0b10000000. We then use OR to set that bit in the CHEEZ register. Voila!

The shift operator is also used for assembly bytes into words and other such magic. Keep in mind that if you ever need to take a word back apart of look at just a portion of the word, the opposite shift is also available. For example, if I want to look at just the top nibble of a byte in the CRACKR register to see if it's equal to 3:

```
if ((CRACKR >> 4) == 3)
     //do something
```

The >> here shifts the bits DOWN by 4. The ones on bottom are pushed right off and disappear, but the remaining bits are still available to be compared to 3.

This barely scratches the surface on bit operations, but hopefully it gives enough of a taste to understand the examples in this class. There are many resources and books available to learn more, if you are interested.

Targets

Our last note is about targets. A target in embedded software is a combination of the processor core, the peripheral set, the actual hardware connected, and even the toolchain being used to compile and link. These things work together to create the perfect executable for your needs… but they also work together to create a perfect storm: If any one of these changes, there are probably corresponding changes required to your code. Hardware goes through revisions, parts go obsolete, toolchains release upgrades… It’s likely that your final release code will have slightly different needs than your first version. These often are handled with #ifdefs, changing particular values or adding / removing snippets of code.

This can be a bit challenging for your test framework. The tests will often need to understand the same compile-time options in order to match. As much as possible, these changes should be small and pushed into header files, so that they are clearly accessible by all and don’t litter the majority of the codebase. This will also help with testing.

With that said, best of luck in your endeavors!

Happy Testing!