Device registers in C

Colin Walls

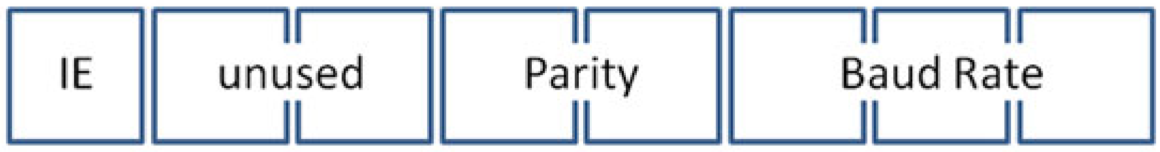
July 27, 2014

One of the key benefits of the C language, which is the reason it is so popular for embedded applications, is that it is a high-level, structured programming language, but has low-level capabilities. The ability to write code that gets close to the hardware is essential and C provides this facility. This article looks at how C may be used to access registers in peripheral devices.

Device registers
The broad issue is quite straightforward. A peripheral device is likely to have a number of internal registers, which may be read from or written to by software. These normally appear just like memory locations and can, for the most part, be treated in the same way. Typically a device register will have bit fields – groups of bits that contain or receive specific information. Such fields may be single bits, groups of bits, or a whole word. There may also be bits that are unused - reading from them or writing to them normally has no effect.

For example, a serial interface (UART) might have an 8-bit register, with the bits used like this:

- Bits 0-2: baud rate, where the values [0-7] have specific significance.
- Bits 3-4: parity, where 0=none, 1=even, 2=odd and 3 is an illegal setting.
- Bits 5-6: unused.
- Bit 7: interrupt enable.



Memory addressing
There are a few important matters to get right when accessing device registers from C. The first is data type. It is almost 100% certain that a register will be unsigned and its size (width in bits) must be accommodated. Using traditional C data types, our example register would be declared unsigned char. In newer versions of the language, there are ways to explicitly specify the bit width; in this case uint8_t would do the job.

If the devices registers are larger than 8 bits – perhaps 16 bits or 32 bits – endianity may be an issue; different CPUs locate most and least significant bytes differently. Any attempt to break down such registers into 8-bit units is likely to go awry, for example.

A really important issue to address is the suppression of unnecessary compiler optimizations.

Normally, a compiler will detect multiple accesses to a variable and make the code more efficient by accessing the memory location once, working on the data in a CPU register, and then writing it back to memory later. With a device register, it is essential that the actual device is updated exactly when the code when requires it to be. Similarly, repeated read accesses must be honored. For example, there may be some code that looks like this:

```
unsigned char dev_reg;

while ((dev_reg & 1) == 0)
    wait();
```

The idea here is to continuously poll the variable dev_reg, which is actually a device register, waiting for the least significant bit to be set. Unless dev_reg is declared volatile, most compilers would optimize the code to a single read access and the loop would never end.

Using pointers
A common question from developers, working this close to the hardware for the first time, goes something like this: "I have a variable of the right type for my device register. How do I arrange for it to be mapped to the correct address?" This is a simple question, but the answer is less simple.

Some embedded software development toolkits do make it fairly simple by providing a facility whereby a variable (or a number of variables – a program section) can be precisely located using the linker. Although this is very neat and tidy, it renders the code somewhat toolchain-dependent, which is generally unwise. There is no standard way to achieve this result in the C language.

The usual solution is to think in terms of pointers, which are effectively addresses. So, instead of creating a variable of the appropriate type, you need a pointer to that type. Then, accessing the device register is simply a matter of dereferencing the pointer. So the above example may be re-written.

```
unsigned *char dev_reg;

while ((*dev_reg & 1) == 0)
    wait();
```

There is just the question of making sure that the pointer does point to the device register. So, for example, maybe the device register in the example is located at 0x8000000. The declaration of the pointer could include an initialization, thus:

```
unsigned *char dev_reg = (unsigned char *)0x80000000;
```

Notice that the address is expressed as an integer, which is type cast to be an appropriate pointer type.
There are two drawbacks of this approach:
- The use of pointers is fraught with potential errors.
- An additional variable – the pointer – is created, which uses up valuable RAM.

Both of these issues can be addressed by creating a pointer constant and dereferencing it. The resulting horrible syntax can be hidden in a macro, thus:

```
#define DEV_REG (*(unsigned char *)0x80000000)
```

Now, DEV_REG can be used anywhere that a normal variable is valid. So our example code becomes:

```
while ((DEV_REG & 1) == 0)
    wait();
```

Using bit fields
Since device registers usually contain fields of one or more bits, each of which corresponds to specific functionality (as shown in the example above), it would seem logical to use bit fields in a C structure, thus:

```
struct uart
  {
    unsigned baud : 3;
    unsigned parity : 2;
    unsigned unused : 2;
    unsigned interrupt_enable : 1;
  };
```

This is quite neat and enables clear code to be written like this:

```
struct uart myuart;
```

```
void main()
  {
    myuart.baud = 2;
  }
```

The good news is that this code might work just fine. But the bad news is that in might not. The method by which allocation of bit fields in a word is performed is compiler dependent. So, one compiler may produce exactly the result you expect, but another may not. It is even possible that a given compiler might produce different results depending on the optimization settings.

As compiler dependent code should be avoided, the simple rule is to avoid using bit fields for this purpose.

Using C++

Using C++ instead of C for embedded applications is gradually becoming more common. Applied properly, the language has no real downsides, but can yield significant benefits for larger projects. A particular use case, which is pertinent to this topic, is the hiding of "difficult" code. Using C++, access to a device register may be carefully controlled and the application programmer can be insulated from the details. Here is a class that implements some of the functionality of the UART example:

```
class uart
  {
    unsigned char* port_address;
  public:
    uart(unsigned addr)
    {
      port_address = (unsigned char*)addr;
    }
    void set_parity(unsigned parity)
    {
      *port_address |= (parity << 3);
    }
  };

uart myuart(0x8000000);

int main()
  {
    myuart.set_parity(2);
  }
```

Here there is a pointer to the device register, which is initialized by the constructor – uart(). An example method – set_parity() – provides the program interface to the class.

To complete this code, further member functions would be needed for the other UART parameters and the use of enums would be good practice instead of anonymous numbers (what does a parity value of 2 mean?).

Other complexities

Accessing hardware can have many other challenges for the unwary embedded software developer. A couple of examples:

Some ports are "write only" – in other words you can write data to them, but reading it back is not possible. This is easily addressed by keeping a "shadow" copy of the port's current contents. This is another example of code which may be usefully encapsulated in a C++ class.

Many devices have multiple internal registers, but only respond to a single address. To access a specific register, the code first needs to write the register number to the device and then perform the required write/read operation. This is not difficult, but needs to be managed carefully. Once again, C++ can be a great help.

Conclusions

For something that is apparently quite simple and straightforward, there are a surprising number of options and pitfalls. What is even more surprising, given that this is a matter of concern to developers of almost all embedded systems, is that no standard way to approach this issue has emerged.

NOTE that the use of volatile has been omitted from several of the above examples in order to render the code more readable. In real code, it would need to be deployed. Additionally, in multi-threaded code, care needs to be taken to ensure reentrancy.

***Colin Walls** has over thirty years experience in the electronics industry, largely dedicated to embedded software. A frequent presenter at conferences and seminars and author of numerous technical articles and two books on embedded software, Colin is an embedded software technologist with Mentor Embedded (the Mentor Graphics Embedded Software Division), and is based in the UK. His regular blog is located at **blogs.mentor.com/colinwalls**. He may be reached by email at**colin_walls@mentor.com**.*