Accessing Hardware Registers using C

How to change only a part of a register (only modify specific bits while leaving the rest unchanged)

It will often be necessary to change only some bits in a full 32-bit register while leaving the other bits unmodified. It could be that the register has already been set-up by another part of the program (some pins are already allocated for a specific use), and by writing an assumed default value to the entire register will cause the initial set-up to be undone with adverse effects.

A common way to achieve this (modifying only the bits that you need to), is with the following steps

1. Read the current contents of the register
2. Perform a bit-wise AND operation, with a 'mask'. The mask should have a '1' in every bit position where the bit has to be left unchanged, and a '0' for bits that we want to modify
3. Add a value to the result from the previous step, where the value contains the bits that we want to write to the register in their correct bit positions. For all other bit positions (the bits that should not get modified), set the value to '0'.

For instance, for to set the pin PA1 to an output pin, we have to write the binary value 01 to MODER1 in the register below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The following table illustrates the procedure, showing only the lower 16-bits of the register for brevity.

	Binary value (bits 15:0)	Hexadecimal
Initial register value	X X X X X X X X X X X X X X X X	-
AND with Mask	1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1	FFF3
=	X X X X X X X X X X X X 0 0 X X	-
ADD Value	0 0 0 0 0 0 0 0 0 0 0 0 **0 1** 0 0	0004
=	X X X X X X X X X X X X **0 1** X X	-

This procedure can easily be written in C code in a single line (or 4 assembly instructions). In C, you would write

```
<register> = (<register> & <mask>) + <value>;
```

For instance (the same example as above)

```
GPIOA_MODER = (GPIOA_MODER & 0xfffffff3) + 0x4;
```

In this case, GPIOA_MODER has been defined to be the memory mapped address of the register, as described in the next section.

How to read from/write to a specific memory address using C

In C, we use pointers for memory access. A pointer is a variable in C that contains an address. A pointer typically also has a type associated with it, such as the 'int' in the pointer declaration 'int* my_ptr'. The type here simply refers to the type and size of data that is stored at the address. But the pointer variable will contain a 32-bit address regardless of what type of data it is pointing to.

Pointers have two operators associated with it. The ampersand (&) operator is used to obtain the address of another variable, i.e.

```
int my_var

int* my_ptr = &my_var;
```

will cause `my_ptr` to contain the address of `my_var`.

The other operator is the de-referencing (*) operator. This operator allows us to access the data at the address contained in the pointer. So for the above example, you could have code like this:

```
*my_ptr = *my_ptr + 1;
```

The *my_ptr on the right-hand side of the above code line will cause the processor to read from memory at the address contained in the my_ptr variable. It will then increment this value and write the updated value back into the same memory address.

In embedded systems, we will often have to read from or write to specific memory locations, because the registers that control the hardware peripherals are mapped to these memory addresses. We can make use of pointers to do this. In the above example we saw that one wat to assign a value to a pointer was to use the &-operator to get the address of another variable. We can also assign a constant value to the pointer, without using the &, i.e.

```
int* my_ptr = 0x12345678;  //now my_ptr 'points' to the address 0x12345678
```

If we need to access a particular register at a known memory address, we can set-up a pointer variable to point to it, and then use the '*' operator to access it.

```
uint32_t* ptr_gpioa_moder = 0x40020000;

uint32_t reg_contents = *ptr_gpioa_moder;  // read from register

*ptr_gpioa_moder =  0; // write to register.
```

It is also possible to use the '*' operator directly on a numeric constant value, without having to use a pointer variable. We do have to 'cast' the constant value to a pointer of a specific type, otherwise the C-compiler will not know what size of data to read or write

```
uint32_t reg_contents = *(0x40020000);   // this will almost work, but the
C-compiler has to know what the size of the data at address 0x4002000 is.
So rather use

uint32_t reg_contents = *( (uint32_t*) 0x40020000 );
```

And then finally, we can make use of a #define (compiler directive) to make the code look neater.

```
#define GPIOA_MODER *((uint32_t*)0x40020000)  // notice there are no
spaces in *((uint32_t*)0x40020000), and no semi-colon at the end of this
line
```

Which will allow you to write

```
GPIOA_MODER = (GPIOA_MODER & 0xfffffff3) + 0x4;
```