Accessing peripherals

Federico Terraneo

Accessing peripherals

Question: how can software interact with hardware?

Accessing peripherals — Peripheral registers

The most common way is through peripheral registers. Hardware peripherals expose their functionality to software through a set of "registers", that are memory locations mapped to specific addresses in the processor address space, thus accessible via software.

Caveat:

- Peripheral registers must not be confused with CPU registers.
- Peripheral registers are mapped at physical addresses, not virtual ones, so in operating systems with memory protection (Linux, Mac, Windows) are accessible only from within the OS kernel. In a microcontroller environment instead they are freely accessible since there is (usually) no memory protection.

Accessing peripherals — Peripheral registers

Peripheral registers have similarity with variables of programming languages allocated in RAM, as:

- They are accessible in the same way (for example, through load/store assembler instructions in RISC processors), as they are mapped in the same address space of RAM
- In many cases they can be read and written in software (even if some registers may be read-only).
- have a width, 8, 16 or 32bit, just like unsigned char, unigned short and unsigned int data types in C/C++.

Accessing peripherals — Peripheral registers

However, peripheral registers also have significant differences when compared to RAM allocated variables:

- What gets written in those registers causes actions in the real world, (such as a LED turning on, an ADC initiating a conversion, a character being sent through a serial port, etc.)
- They are at well specified memory addresses. When a variable is allocated in RAM, whether on the stack or the heap, to the programmer it doesn't matter that it gets allocate at the address 0xbffffc60 or 0xbffffe12, while if a peripheral register is mapped at the address 0x101e5018 the programmer needs to be sure that is writing at that address, or nothing will happen.
- Peripheral registers are not at exclusive use to the programmer, they are shared between the hardware and software. For example, the hardware can decide to change the content of a register to signal events or status flags, while variables simply keep the value stored in them by the programmer.

Accessing peripherals — Peripheral registers

How can a programmer know the peripherals available in a given architecture, which registers these peripherals have, at which address they are mapped and how to use them? For a microcontroller, available peripherals are detailed in a document, usually called “datasheet” or “reference manual”.

Datasheets are usually available at the manufacturer’s website, so the datsheet of STM32 microcontrollers are available in ST’s website, while the datasheet of the ATMega328 microcontroller used in the Arduino is available at the Atmel’s website.

Accessing peripherals — Peripheral registers

Method 1:
Let's assume that in the microcontroller we are programming there is a 32bit register called IODIR0 in the documentation, that it is mapped at the address 0xe0028008, and that we want to write zero in that register.
How can we do that from C/C++?

```
void clearReg()
{
    (*((volatile unsigned int *) 0xe0028008)) = 0;
}
```

First of all, we use a C cast operator to cast the 0xe0028008 number to a pointer to an unsigned int data type. The choice of an unsigned int is due to tha fact that the register is a 32bit register and unsigned int is a 32bit data type (on a 32bit processor). The '*' at the left dereferences the pointer, thus giving access to the memory location pointed to by the pointer, and zero is written into that address. The "volatile" keyword is necessary to disable compiler optimizations, such as instruction reordering and redundant write elimiation that cause problems as the compiler is not aware that at that memory location there is a peripheral register.

Accessing peripherals — Peripheral registers

To increase code readability, it is possible to rewrite the code using a macro:

```
#define IODIR0 (*((volatile unsigned short *) 0xe0028008))
```

This allows to rewrite the code as

```
void clearReg()
{
    IODIR0 = 0;
}
```

This code is more clear than the previous one, as it makes writing to a peripheral register more similar in syntax to writing to a variable. Moreover, the use of the symbolic name IODIR0 instead of the address makes the code more clear, as peripheral names can be easily looked up in the datsheet. It is common practice to group macros defining all registers of an architecture in an header file, and add a #include "file.h" in all source files that need to access peripheral registers.

Even better, most microcontroller manufacturers already provide that header file, so the programmer is relieved from the burden of writing it.

Accessing peripherals — Peripheral registers

Method 2:
Rarely a peripheral, such as an ADC or an SPI have only one register. Usually a peripheral is controlled through a set of registers, that are often mapped at contiguous or close-by addresses. This allows grouping all peripheral registers in a struct, like this

```
struct GpioPeripheral
{
    volatile unsigned int CRL;
    volatile unsigned int CRH;
    volatile unsigned int BSRR;
    volatile unsigned int BRR;
};

#define GPIO ((struct GpioPeripheral *)0xfeeeaba0)
```

The struct defines all the peripheral register and their relative offset in memory, while the macro defines their absolute placement in memory, giving a "self documenting" name to the address of the pointer to struct.

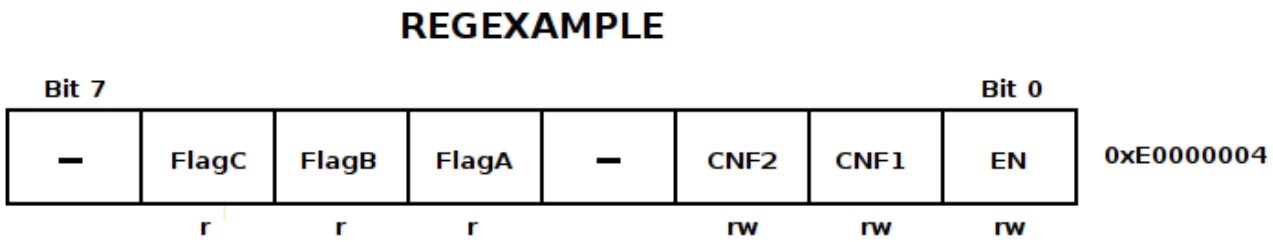
Accessing peripherals — Peripheral registers

Using this method, the code to clear register CRL in the GPIO peripheral is

```
void clearReg()
{
    GPIO->CRL = 0;
}
```

There are no differences, not even in performance between the two methods, both achieve the same result.
Some manufacturers however use the first method, some the second one, so it is necessary to know both.

Accessing peripherals — Peripheral registers



The figure shows a typical peripheral register as documented in a datasheet. Among the information available there are its name (REGEXAMPLE), and its address in memory (0xE0000004). As can be seen, there are three readable and writable bits (shown as rw), and three bits are read-only (r). There are also two unused bits.
A possible representation in code of this register can be

```
#define REGEXAMPLE (*((volatile unsigned char *) 0xe0000004))

#define REGEXAMPLE_EN (0x01)
#define REGEXAMPLE_CNF1 (0x02)
#define REGEXAMPLE_CNF2 (0x04)
#define REGEXAMPLE_FLAGA (0x10)
#define REGEXAMPLE_FLAGB (0x20)
#define REGEXAMPLE_FLAGC (0x40)
```

The first macro defines the register, the other are optional and allow to represent in code the individual bits with a self documenting name.

Accessing peripherals — Peripheral registers

Given that inside a register there are more bits with different functions, there is the need to alter a single bit inside a register. This requires working with bit fields.

This is the code to set the bit EN to 1, leaving the other unaffected

```
REGEXAMPLE |= REGEXAMPLE_EN;
```

While this is the code to set the same bit to 0

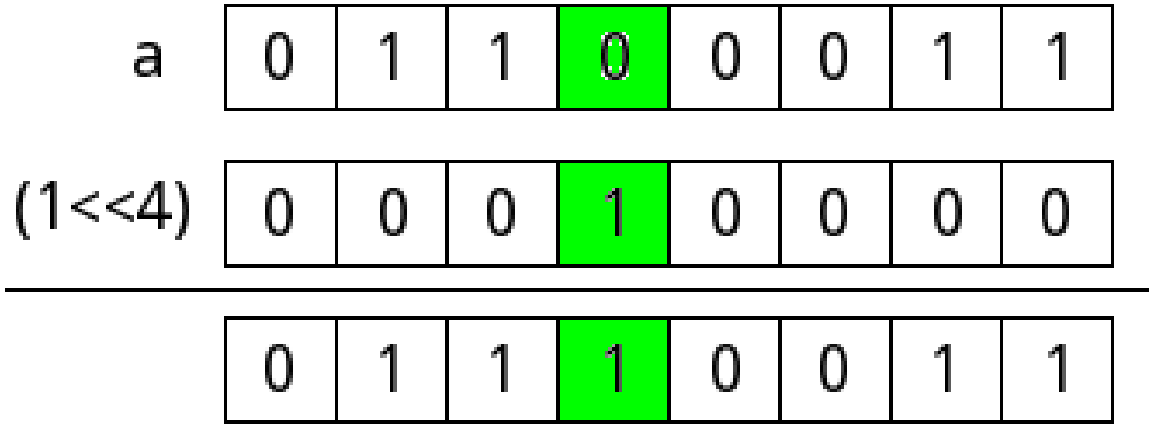
```
REGEXAMPLE &= ~REGEXAMPLE_EN;
```

Last, this is the code to test if the bit FLAGA is set to 1

```
if(REGEXAMPLE & REGEXAMPLE_FLAGA)
{
    [...]
}
```

Accessing peripherals — Peripheral registers

```
a |= (1<<4);
```




```
a &= ~(1<<4);
```