

# Low level programming

Federico Terraneo

22 ottobre 2011



In questo talk si parlerà di come scrivere software che si interfaccia direttamente con l'hardware.

I concetti qui presentati sono orientati alla programmazione di microcontrollori, anche se si ritrovano con poche modifiche anche per la scrittura di driver per periferiche in ambiente PC, quindi per sistemi operativi come GNU/Linux.

Ci saranno anche esempi pratici, sempre tratti dal mondo dei microcontrollori. La scelta di portare esempi relativi ai microcontrollori è dovuta sia a ragioni di semplicità (le periferiche dei microcontrollori sono più semplici di quelle di un PC), sia per il fatto che quando si usano microcontrollori è più frequente doversi “sporcare le mani” con programmazione di basso livello.

Infine, saranno introdotti dei programmi free software/open source che consentono di sviluppare software per microcontrollori in ambiente GNU/Linux.

# Come interfacciarsi con l'hardware

Molti programmatori, anche esperti, non hanno un'idea precisa di come il software interagisca con l'hardware. Questa slide si propone di risalire i vari livelli di astrazione all'interno di un sistema operativo come GNU/Linux, fino al punto che ci interessa, l'interazione diretta con l'hardware.

Libreria userspace. E' normalmente il livello di astrazione a cui i programmatori sono maggiormente abituati. Ad esempio, se un programmatore volesse interagire con la USB su GNU/Linux, utilizzerebbe libusb. (<http://www.libusb.org>)

Ok, ma... come fanno queste librerie a interagire con l'hardware? Facendo chiamate al kernel del sistema operativo.

Ok, ma... come sono implementate nel kernel queste chiamate? Facendo altre chiamate ad un driver di periferica.

(ci stiamo avvicinando...) Ok, ma... come sono implementati i device driver?, come comunicano con l'hardware?

Il metodo più comune è quello dei registri di periferica. Le periferiche hardware si presentano al software come un set registri, che non sono altro che locazioni di memoria mappate a specifici indirizzi nello spazio di indirizzamento, e quindi accessibili tramite software.

Caveat:

- I registri di periferica non vanno confusi coi registri della CPU.
- I registri di periferica sono mappati ad indirizzi fisici, non virtuali, quindi nei sistemi operativi con protezione della memoria (Linux, Mac, Windows) sono accessibili solo da dentro il kernel.  
Nei microcontrollori, invece, sono sempre accessibili.

I registri di periferica sono per certi versi paragonabili a delle variabili allocate in RAM, in quanto

- sono accessibili allo stesso modo (essendo mappati nello stesso spazio di indirizzamento)
- in molti casi sono leggibili e scrivibili dal software (alle volte però capita di avere a che fare con registri read-only).
- hanno una dimensione, solitamente di 8, 16 o 32bit, esattamente come gli unsigned char, unsigned short e unsigned int.

Ciononostante, ci sono delle differenze fondamentali tra i registri di periferica e le variabili

- Quello che viene scritto in questi registri causa azioni nel mondo reale (l'accensione di un LED, l'attivazione di un ADC, l'invio di un carattere tramite una porta seriale, etc.)
- Si trovano a specifici indirizzi di memoria. Quando una variabile viene allocata sullo stack o sull'heap, al programmatore non importa se viene allocata all'indirizzo 0xbffffc60 o 0xbffffe12, mentre se il registro di periferica si trova all'indirizzo 0x101e5018 occorre essere sicuri di stare scrivendo esattamente a quell'indirizzo, o non si otterranno i risultati voluti.
- I registri di periferica non sono ad uso esclusivo del programmatore, come le variabili. Sono condivisi tra il software e l'hardware. Per esempio l'hardware puo' decidere di flippare bit all'interno dei registri per segnalare eventi specifici, cosa che non succede con le normali variabili.

Come si fa a sapere quali periferiche si hanno a disposizione, quali registri ha una specifica periferica, a che indirizzo sono mappati e come usarli? Per un microcontrollore le periferiche disponibili sono documentate dal produttore in un documento, solitamente chiamato “datasheet” o “programming guide”.

I datasheet sono in genere disponibili sul sito del produttore del microcontrollore. Per esempio il datasheet dell'ATmega328, il microcontrollore usato nell'Arduino, si trova sul sito della Atmel, mentre il datasheet dei microcontrollori stm32 si trova sul sito di ST.

# Come accedere ai registri di periferica

Metodo 1:

Assumiamo che nel microcontrollore che stiamo usando ci sia un registro a 32bit, chiamato IODIR0 all'indirizzo 0xe0028008, e che vogliamo scriverci zero.

Come si fa?

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

Ok, un minimo di spiegazione, dato che il codice potrebbe non essere così intuitivo. Prima l'indirizzo viene castato a "volatile unsigned int \*", ossia un puntatore a un intero a 32bit. Poi il "\*" sulla sinistra dereferenzia il puntatore. Infine, il valore zero viene scritto in quella locazione di memoria. Il "volatile" serve ad evitare ottimizzazioni del compilatore, come il reordering delle istruzioni, che possono creare problemi in quanto il compilatore non è a conoscenza dei side effect "nel mondo reale" causati dall'accesso ai registri di periferica.

# Come accedere ai registri di periferica

Per aumentare la leggibilità del codice, si può usare una macro come:

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

Così facendo, il codice può essere riscritto in questo modo

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

Questo codice rende meglio l'idea di quello che sta succedendo, ossia l'assegnamento del valore zero al registro IODIRO. Inoltre, l'uso del nome simbolico del registro (IODIRO) invece che dell'indirizzo di memoria (0xE0028008) rende il codice self documenting in quanto per sapere cosa sta succedendo basta aprire il datasheet e cercare cosa fa il registro IODIRO. E' pratica comune raggruppare tutte le macro che danno nomi simbolici ai registri di periferica in un header, e usare una `#include "file.h"` in tutti i sorgenti che devono accedere alle periferiche.

Ancor meglio, alcuni produttori di microcontrollori rendono disponibili degli header già pronti, da scaricare e includere nei propri progetti.

# Come accedere ai registri di periferica

Metodo 2:

Raramente una periferica, come un ADC o una seriale, ha un solo registro. Il caso tipico è quello in cui ogni periferica viene controllata da un insieme di registri, che spesso sono mappati ad indirizzi di memoria consecutivi. Questo rende possibile raggrupparli in una struct, come questa

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

#define GPIO ((struct GpioPeripheral *)0xf0000000)
```

La struct definisce in un sol colpo tutti i registri della periferica, mentre la macro da un nome self documenting ad un puntatore alla struct, mappato all'indirizzo corretto.

# Come accedere ai registri di periferica

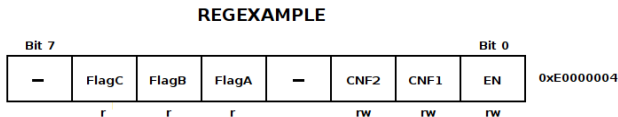
Usando questo metodo il codice per azzerare il registro CRL della periferica GPIO è questo

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

Non ci sono differenze, neanche di performance tra i due metodi, entrambi portano allo stesso risultato.

Alcuni produttori forniscono header con questa rappresentazione delle periferiche, altri con solo le macro, quindi è necessario conoscere entrambi i metodi.

# Come accedere ai registri di periferica



La figura mostra un tipico registro di periferica così come viene documentato nei datasheet. Tra le informazioni disponibili ci sono il nome del registro (REGEXAMPLE) e l'indirizzo del registro (0xE0000004). Come si può vedere ci sono tre bit leggibili e scrivibili (indicati come rw), e tre bit solo leggibili (r). Ci sono infine due bit inutilizzati.

Una possibile rappresentazione in codice di questo registro può essere

```
#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)
```

La prima macro definisce il registro, le altre sono opzionali e servono per dare un nome anche ai bit all'interno del registro.

# Come accedere ai registri di periferica

Il fatto che all'interno di un registro ci siano più bit con funzioni indipendenti fa sorgere la necessità di alterare un singolo bit all'interno di un registro. Il codice per fare ciò non è difficile, ma può essere non poi così immediato la prima volta che lo si vede.

Questo è un esempio di codice che setta a 1 il bit EN lasciando inalterati gli altri

```
REGEXAMPLE |= REGEXAMPLE_EN;
```

Mentre questo è il codice per settare a 0 lo stesso bit

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

Infine, il codice per testare se il bit FLAGA è a 1 è il seguente

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

Il trucco sta nell'usare le operazioni logiche di or ed and con una bitmask.

# Esempio #1: GPIO del microcontrollore ATmega32

I GPIO sono una delle periferiche più semplici, il nome sta per “General Purpose Input/Output” e la loro funzione è quella di poter controllare il livello logico su dei piedini di un circuito integrato tramite software.

PC6	1	28	PC5
PD0	2	27	PC4
PD1	3	26	PC3
PD2	4	25	PC2
PD3	5	24	PC1
PD4	6	23	PC0
VCC	7	22	GND
GND	8	21	AREF
PB6	9	20	AVCC
PB7	10	19	PB5
PD5	11	18	PB4
PD6	12	17	PB3
PD7	13	16	PB2
PB0	14	15	PB1



La figura a destra mostra il microcontrollore ATmega328, quello usato nell'Arduino. Come si può notare, ai lati ci sono due file da 14 contatti elettrici, detti “piedini”. Sulla sinistra è evidenziata la funzione di questi piedini.

# Esempio #1: GPIO del microcontrollore ATmega32

A parte alcuni piedini necessari a fornire l'alimentazione al microcontrollore (indicati con VCC, AVCC e GND), ed un piedino specifico per l'ADC (AREF), tutti gli altri sono usabili come GPIO.

## Caratteristiche dei GPIO

- sono configurabili in software come uscita, in questa modalità di funzionamento è possibile produrre sotto controllo software una tensione di uscita di zero Volt (livello logico 0), oppure una tensione pari a quella di alimentazione del microcontrollore, solitamente 5V o 3.3V (livello logico 1). La corrente che possono fornire è limitata, ma comunque sufficiente ad accendere un LED, oppure...
- ...sono configurabili come ingresso, in questa modalità è possibile leggere via software se la tensione ai loro capi è un livello logico 0 oppure 1.
- Sono raggruppati in porte, in modo da poter leggere/scrivere più GPIO contemporaneamente. Spesso le porte sono indicate con delle lettere, mentre i singoli GPIO all'interno delle porte con dei numeri. Quindi i GPIO hanno dei nomi come PA0, PA1, PB0, PB1. Le porte sono solitamente composte da 8, 16 o 32 GPIO.

## Esempio #1: GPIO del microcontrollore ATmega32

In software, i GPIO sono visti come registri di periferica. Nell'ATmega32 ogni porta ha tre registri a 8bit: PORTx, PINx e DDRx.

Quindi la porta B è accessibile tramite i registri PORTB, PINB e DDRB, mentre la porta C tramite PORTC, PINC e DDRC.

All'interno di ogni registro ogni bit si riferisce a un singolo GPIO, quindi il bit 0 di PORTB, PINB e DDRB si riferiscono a PB0, mentre il bit 1 a PB1.

Ok, quindi ci sono tre bit per GPIO, in tre distinti registri. Il loro significato è il seguente,

- Se il bit 0 di DDRB è zero, PB0 è un ingresso. Leggendo in software il registro PINB, il bit 0 di tale registro consente di sapere se l'ingresso è a livello logico 0 oppure 1.
- Se il bit 0 di DDRB è uno, PB0 è un'uscita. Scrivendo in software il registro PORTB, il bit 0 consente di decidere se generare un livello logico zero oppure uno.

Queste considerazioni si applicano allo stesso modo anche agli altri 7 bit di DDRB/PORTB che controllano i GPIO PB1..PB7, e anche alle altre porte.

# Esempio #1: GPIO del microcontrollore ATmega32

Un esempio di codice per far lampeggiare un LED su PB0 è il seguente

```
#define F_CPU 1000000UL //Richiesto da delay.h
#include <avr/io.h>      //Definizioni dei registri di periferica
#include <util/delay.h> //Funzione _delay_ms()

int main()
{
    DDRB=0x01; //00000001b : PB0=uscita, tutti gli altri=ingresso
    for(;;)
    {
        PORTB |= 0x01; //Setta il bit 0 di PORTB, uscita a livello 1
        _delay_ms(500);
        PORTB &= ~0x01; //Resetta il bit 0 di PORTB, uscita a livello 0
        _delay_ms(500);
    }
}
```

Da notare l'uso di `|=` e `&=~` introdotti prima, per alterare un solo bit.

## Esempio #1: GPIO del microcontrollore ATmega32

In questo esempio invece PB1 è configurato come uscita, e il valore prodotto è il not logico del valore di PB0, configurato come ingresso.

```
#include <avr/io.h>

int main()
{
    DDRB=0x02; //00000010b : PB0=ingresso, PB1=uscita, tutti gli altri=ingresso
    for(;;)
    {
        if(PINB & 0x01) PORTB &= ~0x02;
        else PORTB |= 0x02;
    }
}
```

Anche in questo caso è interessante notare l'uso di una bitmask nell'if per effettuare il check sul solo bit 0 di PINB.

## Esempio #2: Timer 7 del microcontrollore STM32

Un timer è una periferica che genera eventi periodicamente. In genere questi eventi possono fare scattare degli interrupt.

Un interrupt, è un evento hardware che porta la CPU a sospendere temporaneamente l'esecuzione del codice che stava eseguendo, e ad eseguire al suo posto una routine, detta ISR "Interrupt Service Routine". Una volta conclusa l'ISR la CPU riprende ad eseguire il codice precedente.

Semplificando al massimo, un interrupt è uno strumento che consente all'hardware di chiamare una funzione (la ISR) scritta in un linguaggio di programmazione come il C.

## Esempio #2: Timer 7 del microcontrollore STM32



Il microcontrollore STM32F100RB è dotato di più di un timer, l'esempio si concentra sul timer 7.

Questo timer prende in ingresso la frequenza di clock della CPU, pari a 24MHz, e la divide per un numero configurabile tramite il registro PSC. La frequenza ottenuta viene poi usata per alimentare un contatore hardware.

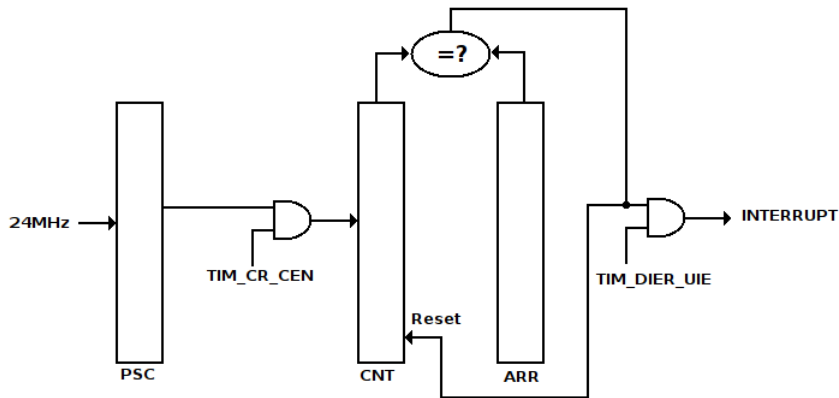
Questo contatore, accessibile tramite il registro CNT conta in avanti fino a raggiungere un valore programmabile tramite un terzo registro, ARR.

Quando ciò avviene il contatore viene azzerato, e viene generato un evento.

Esiste infine un registro di controllo, CR1 il cui bit zero serve ad abilitare o meno il contatore, e un registro di gestione dell'interrupt (DIER) che consente di attivare o meno la generazione di un interrupt in seguito a un evento.

## Esempio #2: Timer 7 del microcontrollore STM32

Questo è uno schema a blocchi di come funziona il timer 7



## Esempio #2: Timer 7 del microcontrollore STM32

Prima di continuare è necessario spiegare brevemente come dichiarare una routine di interrupt. I metodi più comuni sono 2

- Occorre dichiarare una funzione con un nome specifico, il compilatore riconosce che è un interrupt dal nome.
- Occorre salvare un puntatore alla funzione all'interno di un registro di periferica.

L'STM32 utilizza il primo metodo, e la ISR del timer 7 si deve chiamare

```
void TIM7_IRQHandler();
```

Dopo aver introdotto brevemente i timer e gli interrupt, nella prossima slide verrà mostrato un esempio di codice che fa lampeggiare un LED usando un timer.

## Esempio #2: Timer 7 del microcontrollore STM32

```
#include "stm32f10x.h"

void TIM7_IRQHandler()
{
    TIM7->SR=0;
    static int status=0;
    if(status) ledOn(); else ledOff();
    status=1-status;
}

int main()
{
    //Enable peripherals clock gating
    RCC->APB1ENR |= RCC_APB1ENR_TIM7EN;

    //Configure TIM7
    TIM7->DIER=TIM_DIER_UIE;
    TIM7->PSC=10000-1;
    TIM7->ARR=2400-1;
    TIM7->CNT=0;
    TIM7->CR1=TIM_CR1_CEN;

    //Enable TIM7 interrupts
    NVIC_SetPriority(TIM7_IRQn,10);
    NVIC_EnableIRQ(TIM7_IRQn);

    for(;;) ;
}
```

## Esempio #2: Timer 7 del microcontrollore STM32

Il codice comincia includendo `stm32f10x.h`, l'header con la definizione dei registri di periferica.

Poi c'è la routine di interrupt, che tramite una semplice statemachine a due stati accende/spegne il LED. Per semplicità si assume che esistano due funzioni `ledOn()` e `ledOff()` che si prendono cura di interagire coi registri dei GPIO.

Il `main()` invece inizia con l'abilitare la periferica TIM7 (per ragioni di minimizzazione del consumo negli stm32 le periferiche sono tutte disattivate al boot).

Poi configura il timer in modo da generare un interrupt ogni secondo, chiama due funzioni, `NVIC_SetPriority()` e `NVIC_EnableIRQ()` per configurare un'altra periferica non descritta qui, l'interrupt controller in modo da abilitare l'accettazione dell'interrupt.

Infine entra in un loop infinito, tanto la routine di interrupt sarà chiamata in automatico una volta al secondo.

- GCC, il compilatore più usato sotto Linux è anche in grado di essere usato come crosscompilatore, ossia compilatore che gira su una macchina (ad esempio un PC) ma genera binari per una altra CPU (ad esempio quella di un microcontrollore).  
E' possibile avere installato più di una istanza di GCC, per diverse architetture. Questo perchè le versioni di GCC per crosscompilazione sono contraddistinte da un prefisso, ad esempio "avr-gcc" è il GCC per AVR, mentre "arm-eabi-gcc" è il GCC per architettura ARM.
- GDB+OpenOCD, una combo molto usata per fare in circuit debugging, ossia poter eseguire il codice single step, mettere breakpoint e analizzare il contenuto delle variabili, il tutto mentre il codice sta girando sull'hardware vero e proprio, non in un simulatore.  
OpenOCD è orientato all'architettura ARM.
- Vari tool, come avrdude, stm32flash, versaloon etc. per poter "flashare" i microcontrollori, ossia trasferire un firmware, ossia un binario al loro interno.

# Sessione live di in circuit debugging

Se avanza tempo, sessione live di in circuit debugging.



Problem?