

# CORSO

# MPLAB X



di **FRANCESCO FICILI**  
e **VINCENZO GERMANO**

**Iniziamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip per sostituire l'MPLab IDE. Sarà l'occasione per conoscere i microcontrollori PIC32, i primi dispositivi a 32-bit prodotti da Microchip.**

**L**a nostra esperienza nello sviluppo di applicazioni embedded, ci ha insegnato che una delle problematiche con cui bisogna fare i conti è che costituisce una parte assai consistente dell'intero ciclo di sviluppo, è la scrittura del firmware della nostra applicazione.

Come ben sappiamo, lo sviluppo firmware è ben diverso dallo sviluppo di software desktop: se nel caso del software abbiamo un sistema operativo alle spalle che ci aiuta, occupandosi di organizzare l'esecuzione dei vari processi e semplificandoci l'accesso alle risorse, quando sviluppiamo un applicativo per un microcontrollore il discorso è ben diverso. Il sistema operativo spesso e volentieri non esiste (o se esiste è un sistema operativo real-time ben diverso da Linux,

Windows o MacOS) ed è nostra responsabilità scrivere il codice che ci permette di accedere alle varie risorse del microcontrollore.

Questo compito, già di per sé estremamente complesso, diventa ogni giorno più sfidante con il crescere della complessità delle applicazioni embedded.

Se in passato ci si trovava ad avere a che fare con qualche semplice periferica analogica o qualche linea seriale, oggi, anche semplici microcontrollori ad 8 bit hanno in dotazione periferiche di una certa complessità, come USB ed ethernet; inoltre si sviluppano applicazioni che sfruttano appieno ciò che il mercato offre. Si pensi, a titolo di esempio, ad un moderno smartphone: si tratta di un sistema che gestisce quantomeno un collegamento GSM/GPRS per la telefonia, un canale WiFi, un canale bluetooth ed un GPS; oltre

Fig. 1 - Logo MPLabX.

a queste periferiche wireless, il sistema dispone certamente di un'interfaccia USB (che nel migliore dei casi è una USB device, ma i dispositivi più recenti iniziano a fornire il supporto OTG, quindi anche capacità host), di un accelerometro a 3 assi, di un display touch, ecc. Il tutto, gestito da un micro a 32-bit come ad esempio un Cortex o un prodotto simile.

Per consentire lo sviluppo di applicazioni così complesse, gli strumenti di sviluppo a supporto dei vari micro si sono dovuti evolvere, passando dai primi IDE corredati di poche funzionalità, alle moderne toolchain che forniscono, oltre ad un IDE (necessariamente più evoluto rispetto al passato) potenti plug-in e framework pensati per soddisfare le più disparate necessità.

Un esempio di toolchain decisamente "moderna" è quella sviluppata da Microchip, fornita come strumento di sviluppo gratuito per le applicazioni che fanno uso dei prodotti Microchip ed in particolare dei nuovi microcontrollori a 32-bit PIC32.

La toolchain Microchip ha come nodo centrale l'IDE, la cui ultima versione è un tool di sviluppo basato su piattaforma NetBeans e denominato MPLabX IDE. Questo IDE si discosta molto dal suo predecessore (MPLab IDE 8.xx), costituendo un vero e proprio salto generazionale nei sistemi di sviluppo forniti da Microchip.

Alcune delle novità più interessanti che differenziano MPLabX dal suo predecessore sono:

- **Cross Platform**; a differenza del passato, il nuovo IDE supporta i tre più diffusi sistemi operativi per Personal Computer, ovvero Windows, Linux e MacOS;
- **Nuova Piattaforma**; il nuovo IDE si basa su una piattaforma di concezione moderna (NetBeans) che è l'antagonista di Eclipse;
- **nuova catena di compilazione**; in contemporanea all'uscita del nuovo IDE è stata lanciata la nuova catena di compilazione; i vecchi compilatori C18, C30 e C32 sono rimpiazzati da XC8, XC16 e XC32;
- **nuovi framework in formato plug-in**; a differenza del passato i framework di sviluppo non sono più forniti come pacchetti esterni, ma sono dei plug-in importabili, come avviene per lo sviluppo di applicazioni desktop.

Come si può facilmente notare dai punti elencati, che danno la portata di quanto sia ampio il

salto generazionale introdotto da questa nuova famiglia di tool, siamo di fronte ad un ambiente di nuova concezione, che si prefigge (come anche ribadito dalla stessa Microchip) di consentire una riduzione di 30-60 minuti al giorno dei tempi di sviluppo dei progetti.

## INTRODUZIONE AD MPLAB X IDE

Passiamo adesso all'esame e con un certo dettaglio, del componente fondamentale della toolchain: l'IDE MPLab X.

MPLabX è un IDE (Integrated Development Environment) specificatamente pensato per lo sviluppo di applicazioni con i microcontrollori delle varie famiglie prodotte da Microchip Technology (quindi supporta tutti i vari core ad 8, 16 e 32 bit).

Si tratta di un ambiente basato sull'IDE multilinguaggio e multi-piattaforma NetBeans, sviluppato di Oracle, e quindi beneficia di tutte le caratteristiche dell'IDE dal quale deriva, come ad esempio la possibilità di espanderne le funzionalità tramite plug-in.

La **Fig. 2** schematizza i componenti principali di questo potente sistema di sviluppo, il quale dispone delle parti seguenti.

- Un **Project Manager**, che è l'interfaccia che permette la gestione del progetto direttamente dall'IDE. Tramite questo componente è possibile inserire e rimuovere file all'interno del progetto stesso e navigare all'interno dei vari file che lo compongono.
- Un **Editor**, che permette di editare i file di progetto. L'editor di MPLabX, derivando dall'editor di NetBeans, è uno strumento molto evoluto che consente di navigare all'interno del codice tramite l'uso di un set di funzioni "Go to" estremamente evoluto (Go to File, Go to Type, Go to Symbol, Go to Declaration...). Come gli editor più recenti, è inoltre supportato il "Live Parsing" del codice (in modo da consentire agli sviluppatori di vedere gli errori di sintassi in real-time durante la stesura del codice, senza necessità di compilare) e l'auto-completamento. Questo potente strumento di sviluppo contiene uno strumento di visualizzazione dell'albero di chiamata delle funzioni, oltre ad altre interessanti caratteristiche.
- Un sistema di configuration management integrato che supporta Subversion, CVS e Mercurial, oltre al supporto integrato per il sistema

di "issue tracking" Bugzilla.

- Una serie di **Language Tools**, che sono i vari compilatori/assemblatori che compongono la toolchain di compilazione. Alcuni di questi tool sono nativamente presenti all'interno del pacchetto base dell'IDE (come ad esempio MPASM), mentre altri, come ad esempio i compilatori C, devono essere installati a parte, ma vengono integrati automaticamente all'interno dell'IDE. Oltre ai compilatori Microchip appositamente sviluppati per MPLabX (XC8, XC16 ed XC32) è possibile anche utilizzare prodotti di terze parti.
- Un **Debugger** a livello di codice, ossia uno strumento che permette (con l'ausilio di HW dedicato esterno) di debuggare l'applicazione direttamente sul codice sorgente, creando un link tra quest'ultimo e l'applicazione binaria scaricata all'interno della memoria flash del microcontrollore.
- Un'interfaccia di programmazione che permette di interfacciare i vari programmatori della Microchip (incluse anche le demoboard con on-board programmer) o programmatori di terze parti.
- Un simulatore, MPLAB SIM, che permette di

simulare il codice scritto localmente, nel caso in cui non si abbia a disposizione un debugger; esiste inoltre la possibilità di utilizzare, sempre all'interno dell'IDE, simulatori di terze parti più performanti, come, per citare un esempio, Proteus.

- Un'utility di conversione integrata, che permette di convertire automaticamente i vecchi progetti sviluppati con MPLab IDE 8.x in progetti MPLab X.

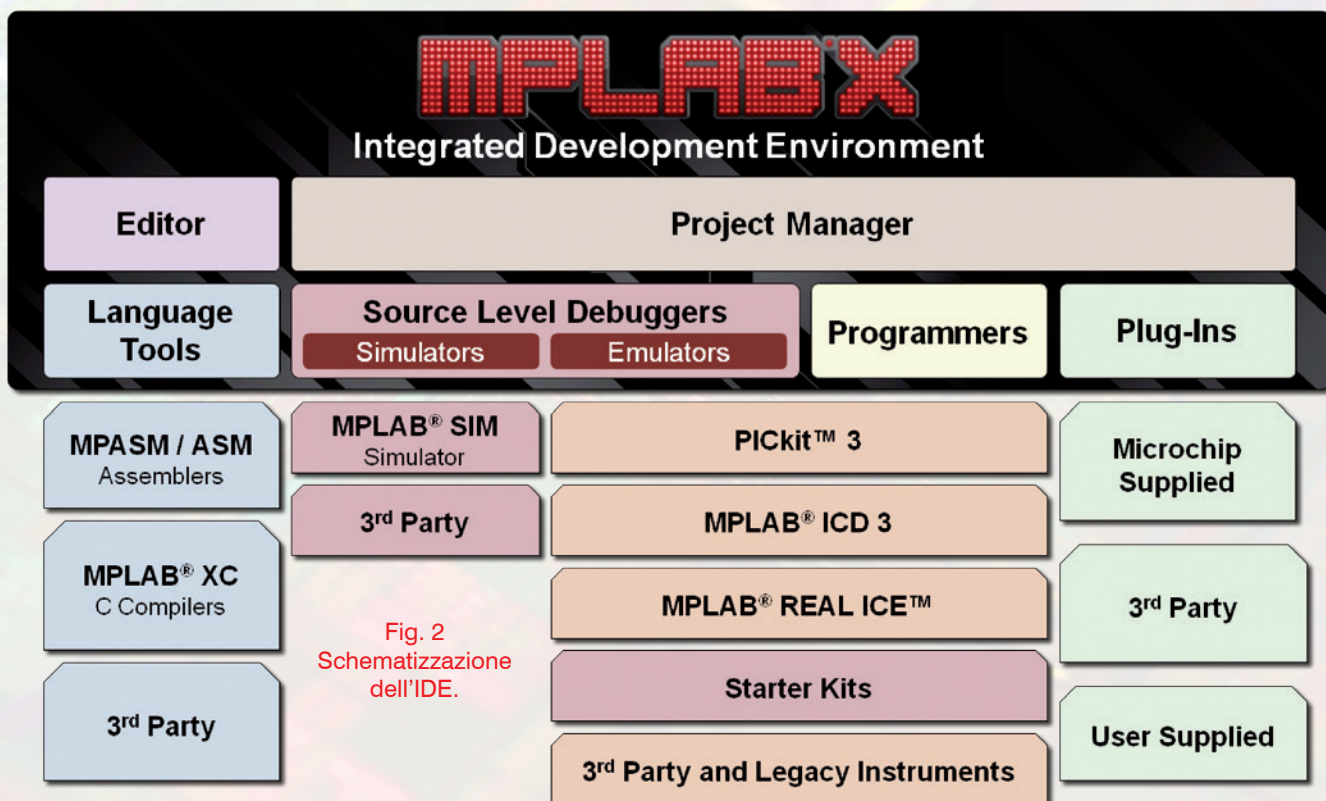
Come accennato in precedenza, l'IDE deriva direttamente da NetBeans ed è quindi possibile usufruire della moltitudine di plug-in sviluppati per l'ambiente nativo, oltre a quelli sviluppati dalla Microchip.

Essendo anche un ambiente open è possibile scaricarlo il codice sorgente e sviluppare dei plug-in proprietari.

### INSTALLAZIONE DELLA TOOLCHAIN

Passiamo adesso all'installazione della toolchain che utilizzeremo durante il corso. In questa puntata installeremo i componenti base per iniziare, ossia MPLabX IDE ed il compilatore XC32.

Per prima cosa occorre scaricare entrambi i pac-



Title	Date Published	Size	D/L
<b>Windows (x86/x64)</b>			
MPLAB® X IDE v2.20	9/3/2014	377.6 Mb	
MPLAB® X IDE Release Notes / User® Guide v2.20 (supersedes info in installer)	9/3/2014	4.0Kb	
MPLAB® X IDE Chinese Translation Files v.1.80	08/08/2013	22Mb	
<b>Linux 32-Bit and Linux 64-Bit (Requires 32-Bit Compatibility Libraries)</b>			
MPLAB® X IDE v2.20	9/3/2014	345.3Mb	
MPLAB® X IDE Release Notes / User® Guide v2.20 (supersedes info in installer)	9/3/2014	4.0Kb	
MPLAB® X IDE Chinese Translation Files v.1.80	08/08/2013	22Mb	
<b>Mac (10.X)</b>			
MPLAB® X IDE v2.20	9/3/2014	254.9Mb	
MPLAB® X IDE Release Notes / User® Guide v2.20 (supersedes info in installer)	9/3/2014	4.0Kb	
MPLAB® X IDE Chinese Translation Files v.1.80	08/08/2013	22Mb	

Fig. 3 - Schermata di download dell'IDE.

chetti dal sito web della Microchip Technology, ai seguenti indirizzi web:

- **MPLabX:** <http://www.microchip.com/pagehandler/en-us/family/mplabx/home.html>
- **XC32:** <http://www.microchip.com/pagehandler/en-us/devtools/mplabxc/home.html>

All'interno di questo pagine web sono riportati i link ai download delle versioni più recenti per i principali sistemi operativi per desktop computer, ossia Windows, Linux e MacOS. In questo corso faremo sempre riferimento alla versione per Windows, ma poiché l'ambiente è multiplatforma, quanto diremo sarà comunque valido anche per le altre versioni. In Fig. 3 è riportato uno screenshot della schermata di download dell'IDE.

Una volta scaricati i pacchetti avviare la procedura di installazione e seguire le istruzioni fino al termine. La Fig. 4 illustra la schermata iniziale del wizard di installazione di MPLab X.

Alla fine dell'installazione del compilatore verrà richiesto che tipo di versione si desidera installare (Fig. 5). I compilatori Microchip sono distribuiti in 3 differenti versioni: Free, Standard e Pro; le tre differiscono per il livello delle ottimizzazioni introdotte automaticamente dal compilatore (schematizzato nella Fig. 6).

Questa caratteristica è estremamente importante per lo sviluppo di applicazioni industriali, dove il risparmio di memoria flash può incidere in maniera determinante sui costi di produzione,

ma ha poca importanza in un contesto hobbistico/didattico come il nostro, dato che per avviare al problema basta scegliere un micro con un taglio di flash superiore. Dal nostro punto di vista è quindi sufficiente cliccare sul tasto "Next" per installare una versione free del compilatore (può anche essere ottenuta una licenza PRO in valutazione della durata di 60 giorni).

### DESCRIZIONE DELL'INTERFACCIA

Iniziamo adesso a prendere confidenza con il nuovo ambiente, cominciando con la descrizione dell'interfaccia. Avviamo quindi il nostro nuovo IDE e attendiamo che la finestra di caricamento carichi tutti i componenti necessari.

All'avvio, l'IDE presenta una schermata che si sovrappone alla finestra dell'editor (Fig. 7) e



Fig. 4 - Schermata iniziale del wizard di installazione.



Fig. 5 - XC32 Licensing information.

fornisce una serie di utili informazioni sull'ambiente, permette di visualizzare un tutorial, di scaricare plug-in, visitare il forum ecc. In Fig. 8 è riportata la struttura dell'IDE, del quale sono evidenziati i componenti principali. Come si può vedere, l'IDE è diviso in cinque componenti

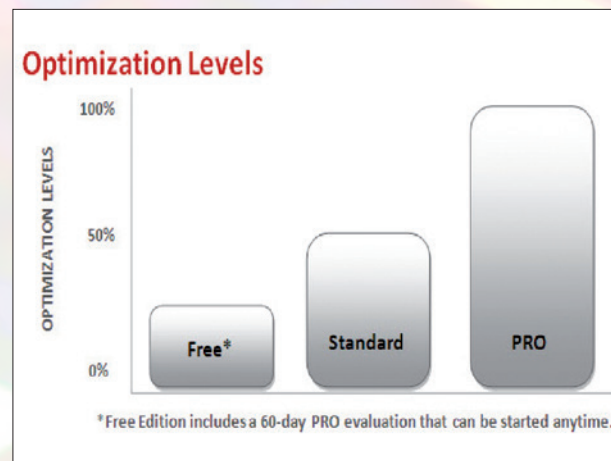


Fig. 6 - Livello di ottimizzazione delle varie versioni dei compilatori Microchip.

principali: *Project Window*, *Navigator Window*, *Output Window*, *Editor Window*, *Main Toolbar*. Di seguito sono elencate le caratteristiche di ciascuno.

- **Project Window:** è la finestra di Project Management, all'interno della quale è riportato

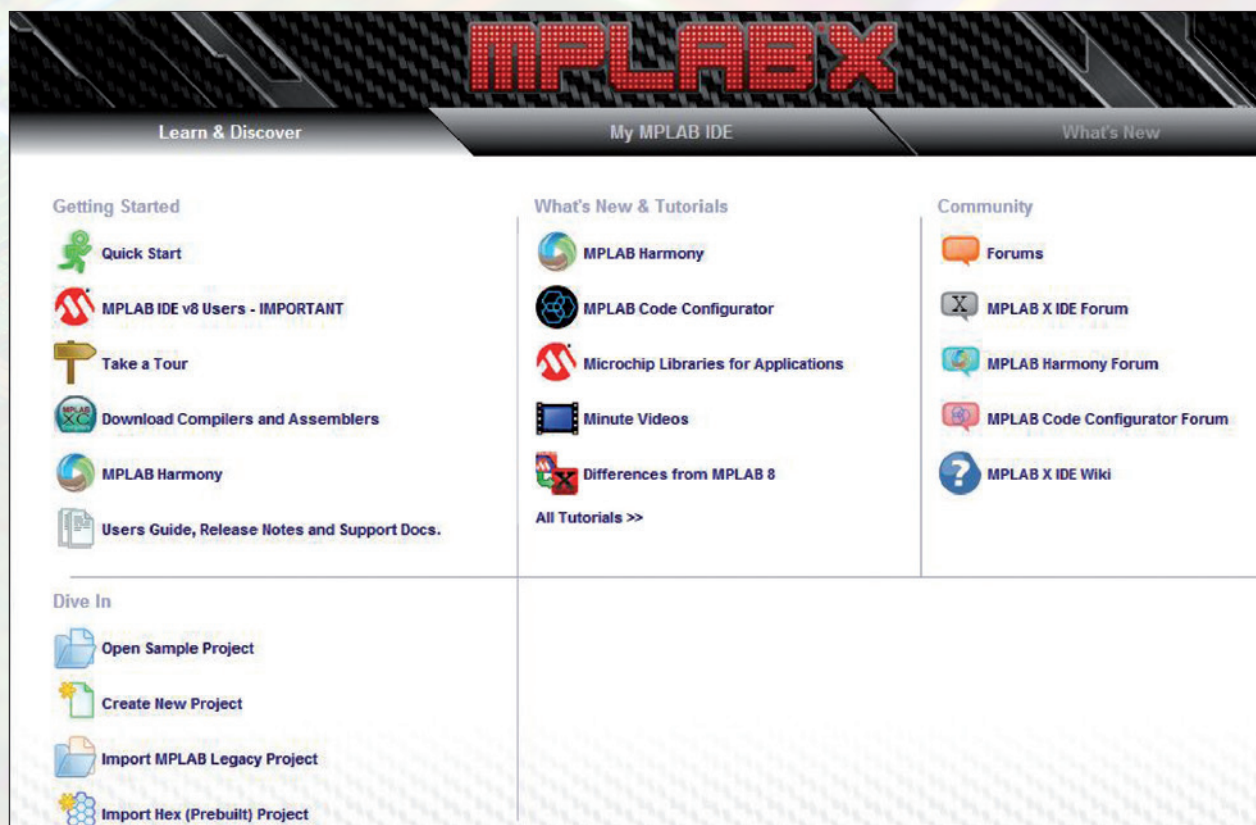


Fig. 7 - Start Page di MPLabX.

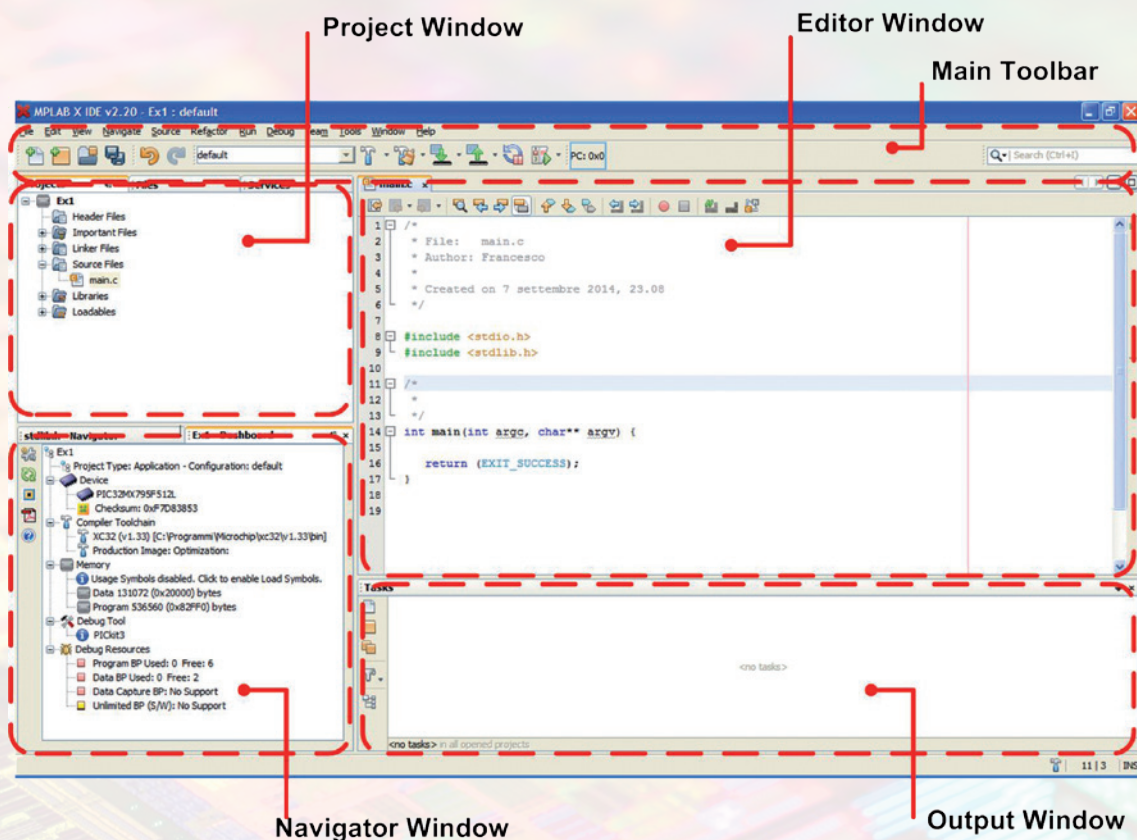


Fig. 8 - Interfaccia di MPLabX.

l'intero file system di progetto.

- **Navigator Window:** è la finestra di navigazione, che elenca tutti gli identificativi presenti all'interno del file selezionato nell'editor e permette di accedervi tramite doppio clic. Inoltre è presente anche un tab, chiamato dashboard, che riassume alcuni dettagli salienti del progetto aperto, come la toolchain ed il tool utilizzati per il progetto corrente, la quantità di memoria RAM e flash libera sul dispositivo, le risorse di debug, ecc.
- **Main Toolbar:** questa barra degli strumenti contiene tutti i principali controlli dell'IDE, come i pulsanti utilizzati per compilare il codice (in debug e release mode), tutti gli strumenti di debug (run, halt, step into, step over, etc.), i pulsanti di scelta rapida per la creazione o il caricamento di file e l'accesso a tutte le opzioni di sistema.
- **Editor Window:** è la finestra dell'editor all'interno del quale viene sviluppato il codice, e contiene anche la relativa toolbar.

- **Output Window:** è la finestra di output dell'IDE, attraverso la quale vengono fornite informazioni relative alla compilazione, alla connessione con i tool esterni, al debugger, ecc.

## CREARE UN PROGETTO CON MPLAB X IDE

Vediamo adesso come creare un progetto

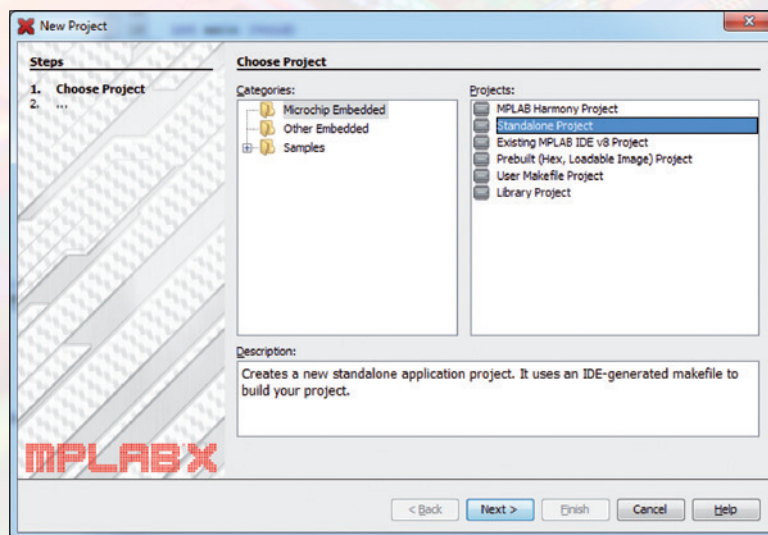


Fig. 9 - Selezione della tipologia di progetto.

usando MPLabX IDE. Per la creazione di nuovi progetti ci viene in aiuto uno strumento chiamato "New Project Wizard". A differenza di quanto avveniva per MPLab IDE, per il quale l'uso del wizard per la creazione di un nuovo progetto era opzionale, per MPLab X, l'uso di questo strumento per la creazione di un nuovo progetto è obbligatorio, in quanto, per funzionare correttamente, l'ambiente necessita di un file system predefinito e di una precisa gerarchia delle sub directory.

Per creare un nuovo progetto in MPLab X, è possibile sia fare clic sull'icona "New Project" presente all'interno della toolbar, sia selezionare l'opzione "New Project" dal menu "File". Eseguendo una di queste due operazioni, verrà aperta la finestra di dialogo visibile nella Fig. 9. Da questa finestra, selezionare l'opzione "Microchip Embedded" e poi "Standalone Project", quindi fare clic sul pulsante Next per proseguire. Notate che selezionando "Samples" è possibile accedere ad una lista di Template per le varie famiglie di microcontrollori Microchip (PIC, PIC24, dsPIC e PIC32).

La finestra successiva, riportata in Fig. 10, consente di scegliere il microcontrollore da utilizzare come target del progetto. Il combo box "Family" permette di impostare un filtro per famiglia di dispositivi, in modo da semplificare la ricerca.

Selezioniamo il modello PIC32MX795F512L e clicchiamo nuovamente sul tasto Next per andare avanti. Dalla finestra successiva (Fig. 11) è possibile scegliere il tipo di tool da utilizzare per la programmazione/debugging del dispositivo; in questo caso scegliete pure il tool di cui disponete, oppure, se non avete al momento un tool da utilizzare potete scegliere di usare il simulatore.

Premendo nuovamente Next si accede alla finestra di Fig. 12, che permette la scelta della toolchain da utilizzare per il progetto; in questo corso noi ci serviremo del compilatore XC32, nella versione free, quindi selezioniamo questa toolchain e andiamo avanti facendo ancora clic su "Next".

Siamo quasi alla fine: non ci resta che selezionare il path ed il nome del nostro progetto: inseriamo il path che desideriamo utilizzare e usiamo Ex1 come nome, quindi facciamo clic sul pulsante Finish, per consentire ad MPLab X di creare il nostro primo progetto.

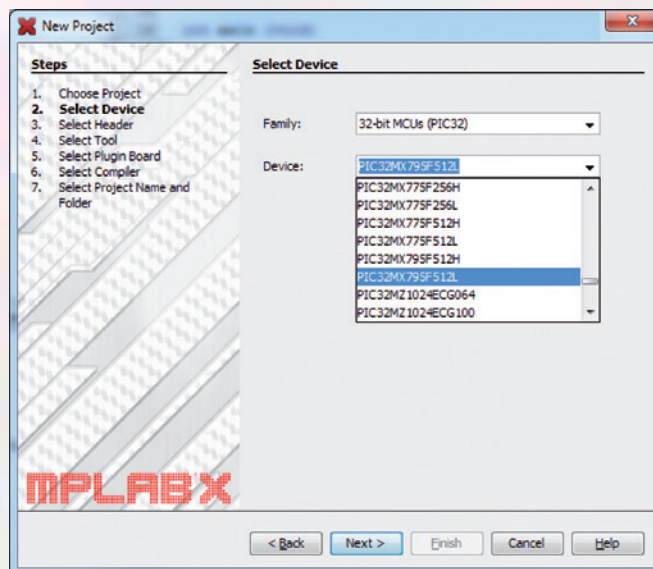


Fig. 10 - Scelta del microcontrollore.

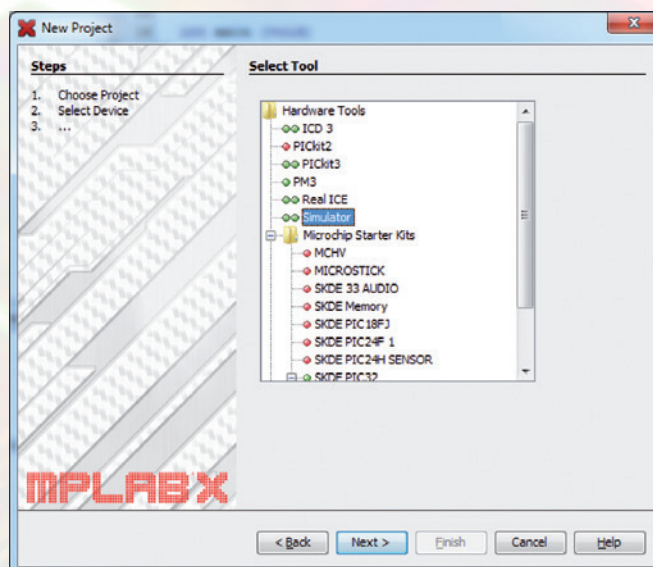


Fig. 11 - Scelta del tool hardware.

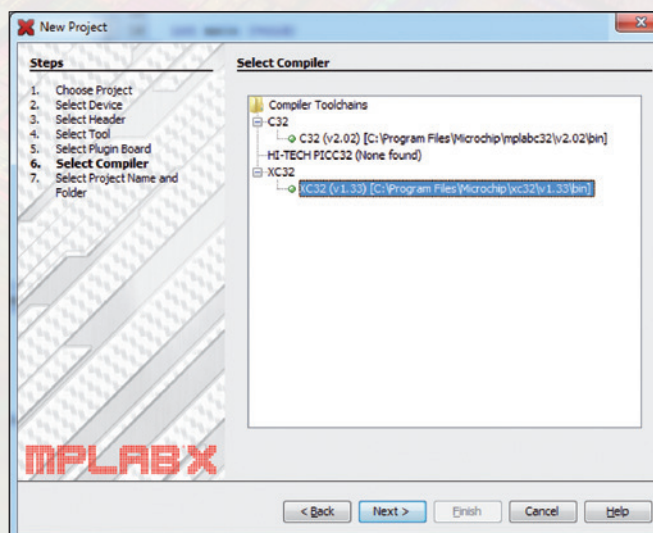


Fig. 12 - Scelta del compilatore.

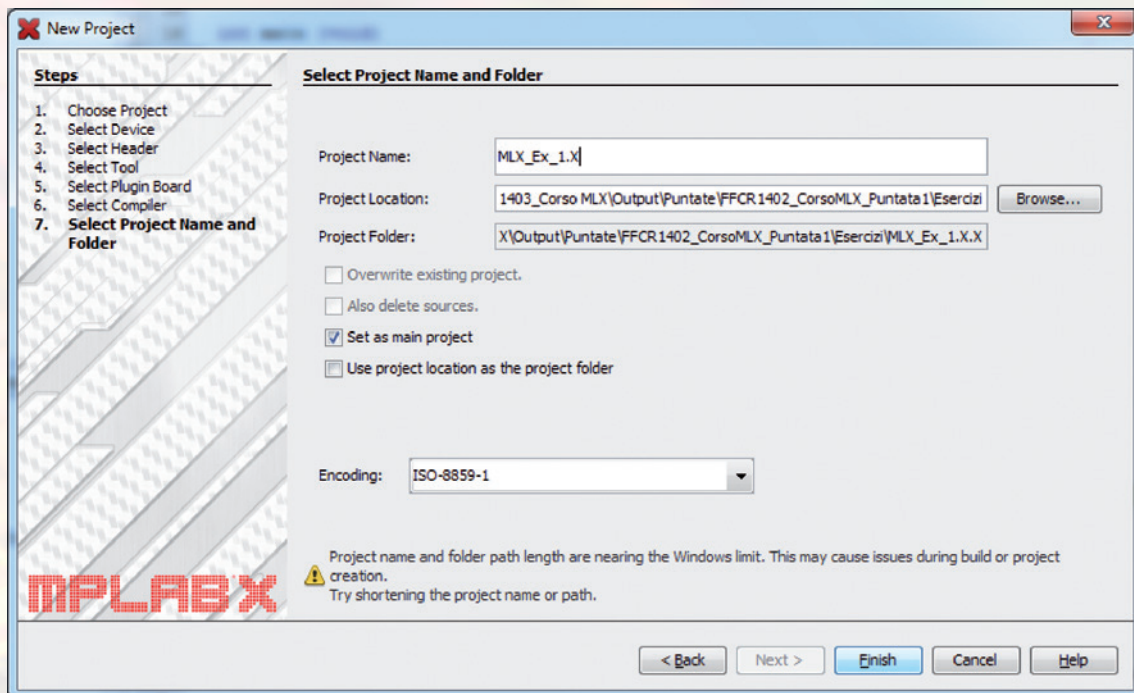


Fig. 13 - Selezione del path di progetto.

A questo punto MPLab X genererà file e directory di progetto. Notate che il nome del progetto viene riportato su una cartella all'interno della quale viene generato il file system specifico dell'IDE, aggiungendo l'estensione ".X", per rendere più

facile l'individuazione di questa categoria di progetti; quindi non esiste più il concetto di "Workspace", caratteristico di MPLab IDE 8.x e di altri IDE simili, ma solo una specifica struttura di file, che viene interpretata dall'IDE come progetto.

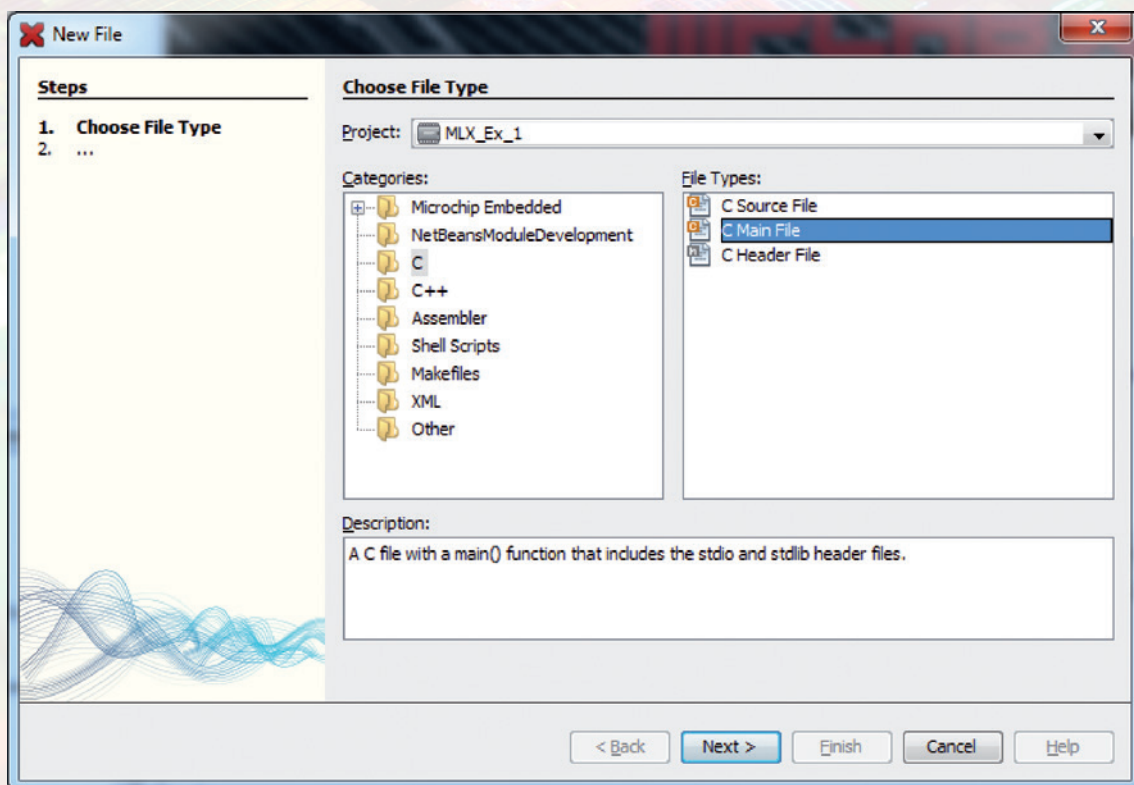


Fig. 14 - Selezione della tipologia del nuovo file.

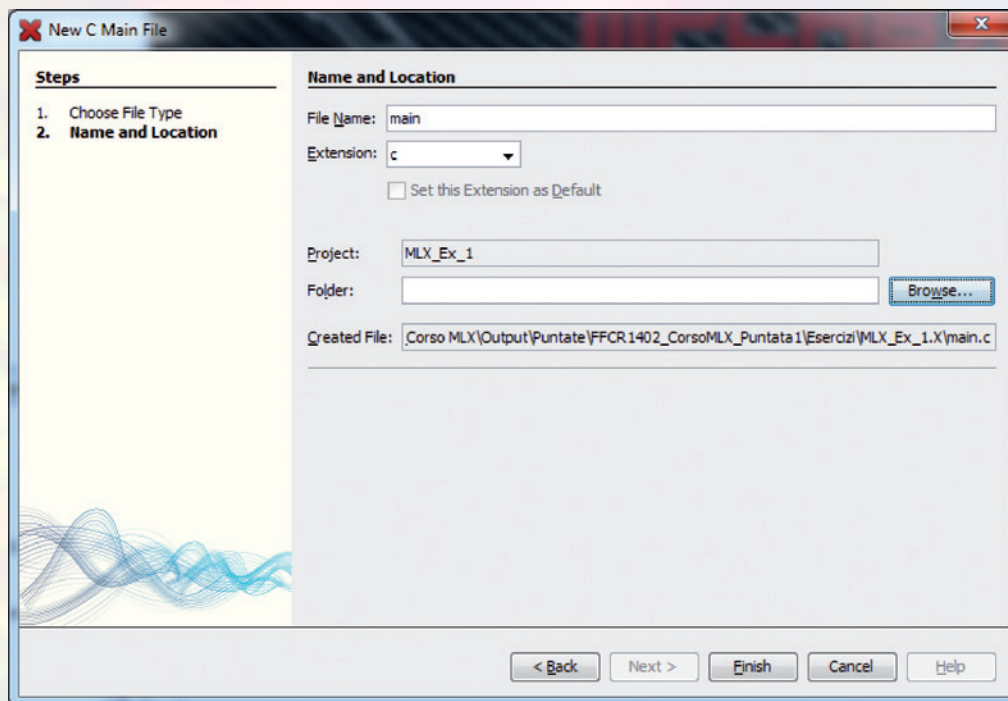


Fig. 15 - Selezione del path del nuovo file.

Fig. 16 - File main.c

## HELLO WORLD

Passiamo ora ad un primo, basilare, esempio pratico di applicazione, e lo facciamo sfruttando uno dei classici "template" della programmazione embedded: facciamo lampeggiare un LED, ovvero, dato che (usando il simulatore per questo esempio) non abbiamo un LED fisico da far lampeggiare, facciamo invertire ciclicamente lo stato logico di un pin di uscita di una qualsiasi

### Listato 1

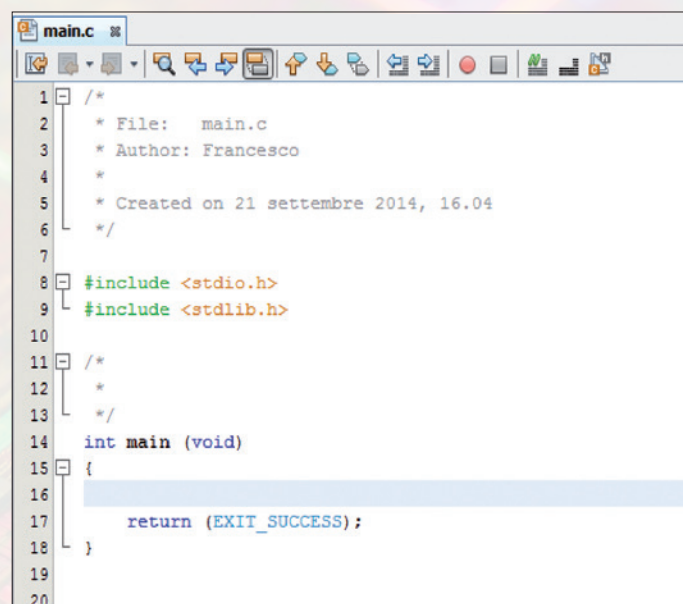
```
/* -- includes -- */
#include <stdio.h>
#include <stdlib.h>
#include <p32xxxx.h>

/* -- defines -- */
#define DELAY_VALUE      100000
#define FOREVER          1
```

porta del microcontrollore.

Chi avesse a disposizione una board fisica con un PIC32, potrà invece cercare di far realmente lampeggiare un LED, ma per effettuare questa operazione ci sono alcuni aspetti da tenere in considerazione, che riguardano il passaggio da ambiente simulato ad ambiente reale, che tratteremo in maniera approfondita nelle prossime puntate di questo corso.

Per cominciare, aggiungiamo adesso un file al nostro progetto, all'interno della sezione "source files" del Project Manager, selezionando l'icona



"New File..." dalla toolbar (o usando la short cut Ctrl + N). Si aprirà la finestra di Fig. 14: tra le varie categorie selezioniamo C file → C main file e clicchiamo su "Next". A questo punto scegliamo il path all'interno del quale andare a salvare il nuovo file (Fig. 15) e chiudiamo. Ora possiamo aprire il file dal project manager, che si presenterà come in Fig. 16; modifichiamo il file appena aperto inserendo le inclusioni e le define riportate all'interno del Listato 1.

A questo punto iniziamo a scrivere, all'interno del main, il codice riportato di seguito; per pri-



Fig. 17 - Pulsante di debug della toolbar.

ma cosa definiamo una variabile di tipo contatore, che chiamiamo genericamente counter, di tipo unsigned long ed inizializzata al valore di DELAY\_VALUE:

```
/* delay counter */
unsigned long counter = DELAY_VALUE;
```

Poi settiamo, con la seguente riga di codice, il registro direzionale della porta D in maniera che tutti i pin di quest'ultima siano configurati come uscite:

```
/* set ddr register */
TRISD = 0x0000;
```

Il registro direzionale (DDR = Data Direction Register) è un SFR (Special Function Register) del microcontrollore, la cui funzione è di impostare la configurazione di I/O dei pin di una data porta (ne esiste quindi uno per ogni porta fisica del microcontrollore).

Impostando il valore di un certo bit del registro a '0', il pin corrispondente della porta associata funzionerà come output digitale (0=output), mentre impostandolo ad '1', lo stesso pin funzionerà come input digitale. Dovendo controllare un LED, impostiamo il registro in modo con tutti i pin in uscita.

A questo punto inseriamo un ciclo while infinito all'interno del quale inseriamo un secondo while, la cui condizione di uscita è data dal valore di counter (se = 0 uscita, altrimenti continua), e all'interno del while decrementiamo "counter" ad ogni ciclo:

```
/* delay loop */
while (counter)
{
    /* decrement counter */
    counter--;
}
```

Questo è l'effettivo ciclo di ritardo.

Una volta usciti, ricarichiamo il contatore per il ciclo successivo e invertiamo lo stato precedente del pin RD0.

```
/* set delay value to DELAY_VALUE */
counter = DELAY_VALUE;
```

```
/* toggle RD0 */
PORTDbits.RD0 = ~PORTDbits.RD0;
```

Il nostro primo, semplice, programma è dunque completo. Ora possiamo lanciare la simulazione premendo il pulsante "Debug Main Project" dalla toolbar e vedere cosa succede.

Per meglio comprendere cosa sta succedendo, inseriamo un breakpoint alla riga 35 (semplicemente cliccando sul bordo della finestra dell'editor, in corrispondenza del numero della riga stessa) e lanciamo l'esecuzione. Dopo un breve periodo di tempo (dovuto al ritardo introdotto con il while più annidato) l'esecuzione si interromperà alla linea 35 (Fig. 18), dove abbiamo piazzato il breakpoint, e se la rilanciamo premendo "Continue" (o in alternativa F5) il tutto si ripeterà nuovamente, per effetto del ciclo while principale (FOREVER).

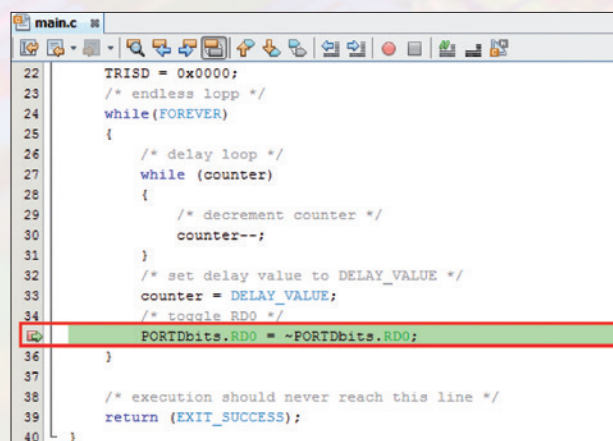
Abbiamo quindi realizzato un ciclo infinito che inverte lo stato di un pin di uscita (se colleghiamo un LED al pin vedremo il LED lampeggiare ad una certa frequenza).

## CONCLUSIONI

In questa prima puntata abbiamo introdotto l'ambiente di sviluppo MPLab X IDE, elencandone le caratteristiche più importanti e descrivendone in dettaglio l'interfaccia utente. Inoltre abbiamo utilizzato il wizard integrato per la creazione del nostro primo progetto e abbiamo illustrato un primo, semplicissimo, progetto pratico.

Nella prossima puntata entreremo nel dettaglio delle prime periferiche, analizzando alcune periferica base come i timer e gli interrupt. ■

Fig. 18 - Debug del programma Hello World.



# CORSO

# MPLAB X



di FRANCESCO FICILI  
e VINCENZO GERMANO

**Continuiamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip Technology. Iniziamo anche a conoscere i microcontrollori PIC32, i primi dispositivi a 32-bit prodotti da Microchip. Seconda puntata.**

**N**ella puntata precedente abbiamo introdotto MPLab X, il nuovo ambiente di sviluppo lanciato da Microchip per lo sviluppo di applicazioni embedded sulle sue piattaforme ad 8, 16 e 32-bit, e basato sull'IDE cross-platform ufficiale di Oracle Netbeans. Questo ambiente, nei piani di Microchip, deve andare a sostituire MPLab IDE 8.x, che è stato l'ambiente di sviluppo ufficiale di Microchip negli ultimi anni ed è ormai in obsolescenza. A corredo del nuovo ambiente, Microchip ha lanciato una nuova linea di compilatori, XC8, XC16 ed XC32, che va a soppiantare la vecchia toolchain, costituita dai compilatori MPLab C18, C30 e C32.

In questa seconda puntata concentriamo maggiormente la nostra attenzione sui

microcontrollori PIC32, che costituiranno la piattaforma hardware principale di questo corso.

## I MICROCONTROLLORI PIC32

Si tratta della più recente famiglia di microcontrollori prodotta dalla Microchip Technology ed è stata introdotta nel mercato nel novembre 2007. Questi potenti microcontrollori sono basati sul core a 32-bit MIPS M4K (licenziato dalla produttrice di IPcores inglese Imagination Technology), che gira ad una frequenza di 80 MHz, dotato di una pipeline a 5 stadi e di una ALU a 32-bit. I primi modelli prodotti sono stati mantenuti pin-to-pin compatibili con quelli della famiglia PIC24FxxGA0xx, in modo da semplificare eventuali migrazioni.

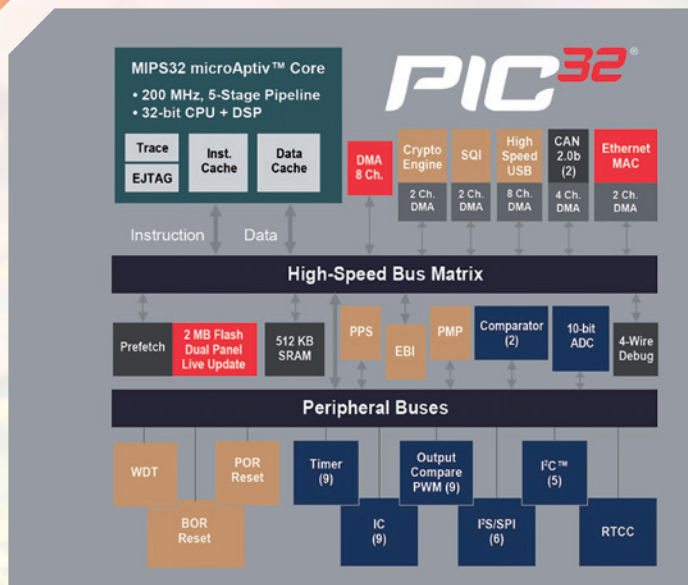


Fig. 1 - Schema a blocchi di un'architettura PIC32MZ.

I microcontrollori PIC32 costituiscono il prodotto più di alto livello commercializzato da Microchip e i chip prodotti inglobano al loro interno periferiche di elevata complessità, come USB Host/OTG/Device, Ethernet controllers, CAN, Capacitive Touch, ecc.

Attualmente esistono due famiglie principali di microcontrollori PIC32: la famiglia PIC32MX (la prima ad essere introdotta) e la famiglia PIC32MZ (attualmente la più performante). Le caratteristiche principali dei microcontrollori che vi appartengono sono:

- velocità di esecuzione 330 DMIPS @200 MHz;
- tagli di memoria Flash fino a 512k;
- una istruzione per ciclo di clock (DSP);
- possibilità di esecuzione del codice direttamente dalla memoria RAM;
- controller DMA;
- PPS (Peripheral Pin Select) per rimappare i pin del microcontrollore;
- interfaccia di programmazione/debug JTAG o 2-wire Microchip;
- supporto Execution Trace;
- Timer, Watchdog, RTCC;
- High-Speed USB (Host/Dual Role/Device), Ethernet (con MAC integrato), CAN, mTouch;

- I<sup>2</sup>C, SPI, UART, PMP;
- ADC e CCP.

Come è possibile vedere, la dotazione di periferiche di questi dispositivi è estremamente ricca, il che consente di sviluppare applicazioni embedded anche molto complesse impiegando un singolo microcontrollore e riducendo al minimo la circuiteria esterna.

## I TIMER

Analizziamo adesso una delle periferiche base dei PIC32, ossia la periferica Timer. Un timer è essenzialmente una periferica hardware deputata a compiti di conteggio. Essendo svincolati dal processore, il conteggio dei timer non influenza il tempo di esecuzione della CPU e, viceversa, un timer hardware esegue un conteggio preciso che non è influenzato dal carico di esecuzione del sistema. Esistono vari tipi di timer: ad esempio, un tipo di timer specializzato per una particolare applicazione è l'RTCC (Real Time Clock Calendar), utilizzato per l'implementazione degli orologi di sistema. Un altro esempio è il timer watchdog, che viene spesso utilizzato per resettare il sistema in seguito a situazioni di stallo. Una caratteristica molto importante dei timer è che, essendo a tutti gli effetti periferiche hardware, possono essere associati ad eventi di interrupt di diverso tipo (in genere overflow o period match con un registro di appoggio, denominato appunto period register) e quindi sono spesso utilizzati per generare interrupt con tempistiche molto precise (gli scheduler, ad esempio, sono in genere basati su un interrupt generato da un timer). Tratteremo più in dettaglio gli interrupt (e la loro possibile associazione ai timer) più avanti, nel paragrafo dedicato. I PIC32 dispongono di due tipi di timer general purpose (in questa categoria non rientra il watchdog, che, data la sua natura, non può essere considerato un general purpose timer), il Timer A ed il Timer B. Il timer A è un timer a

Fig. 2 - Tabella comparativa tra Timer A e Timer B.

Available Timer Types	Secondary Oscillator	Asynchronous External Clock	Synchronous External Clock	16-Bit Synchronous Timer/Counter	32-Bit <sup>(1)</sup> Synchronous Timer/Counter	Gated Timer	Special Event Trigger
Type A	Yes	Yes	Yes	Yes	No	Yes	No
Type B	No	No	Yes	Yes	Yes	Yes	Yes

Come accennato in precedenza, il prescaler presenta un numero maggiore di possibili selezioni (1:1, 1:2, 1:4, 1:8, 1:32, 1:64, 1:256), rendendo questo oggetto capace di risoluzioni maggiori rispetto al Timer A, e quindi più "general Purpose".

## 119

## Listato 1

```

/* -- defines -- */
#define FOREVER          1
#define SYS_CLOCK        8000000
#define TMR_ON           1
#define TMR_16_BIT_MODE  0
#define TMR_32_BIT_MODE  1
#define TMR_CS_PBCLK      0
#define TMR_PRESCALER_8   3
#define TMR_PRESCALER_VALUE 8
#define TMR_MATCH_VALUE  SYS_CLOCK/TMR_PRESCALER_VALUE

```

Listato 1 - Definisce generiche e per il settaggio dei registri di controllo del Timer 2.

una volta configurato il timer, sarà sufficiente controllare quando il registro raggiunge il valore di TMR\_MATCH\_VALUE per essere sicuri che sia trascorso un secondo.

Impostiamo quindi T2CON con un prescaler di 8, in modalità 32-bit (per poter contare fino ad un milione), sfruttando il clock delle periferiche come clock del timer (T2CONbits.TCS = TMR\_CS\_PBCLK) e accendiamo la periferica (T2CONbits.ON = TMR\_ON), come illustrato nel Listato 2.

A questo punto non ci rimane altro da fare che scrivere in C quanto detto prima in linguaggio naturale: "verificare quando il registro timer arriva al valore di TMR\_MATCH\_VALUE" e, una volta raggiunto questo valore, resettare il registro ed invertire lo stato del pin RD0, chiaramente il tutto all'interno del solito ciclo infinito while (FOREVER). Per l'implementazione guardate il Listato 3.

Per testare il funzionamento possiamo mettere il solito breakpoint sull'istruzione che esegue il toggle del pin, oppure se disponiamo di una scheda sui cui provare possiamo verificare la periodicità usando un oscilloscopio.

## INTERRUPT

Trattiamo adesso un argomento un po' più complesso rispetto ai precedenti, ma che costituisce uno dei concetti più importanti nello sviluppo

## Listato 2

```

/* set ddr register */
TRISD = 0x0000;
/* configure the PIC32 core for the best performance */
SYSTEMConfigPerformance(SYS_CLOCK);

/* set T1CON register */
T2CONbits.ON = TMR_ON;
T2CONbits.TCKPS = TMR_PRESCALER_8;
T2CONbits.T32 = TMR_32_BIT_MODE;
T2CONbits.TCS = TMR_CS_PBCLK;

```

Listato 2 - Configurazione di T2CON

## Listato 3

```

/* endless loop */
while(FOREVER)
{
    /* if TMR exceed the match value */
    if (TMR2 > TMR_MATCH_VALUE)
    {
        /* clear timer */
        TMR2 = 0;
        /* toggle RD0 */
        PORTDbits.RD0 = ~PORTDbits.RD0;
    }
}

```

Listato 3 - Hello World con Timer 2

di software per applicazioni embedded, ossia il concetto di interrupt. Prima di tutto cerchiamo di chiarire che cos'è realmente un interrupt, e lo facciamo con un parallelismo con il mondo reale, nel quale anche noi siamo spesso soggetti a delle situazioni che possono essere modellizzate come degli interrupt. Immaginate di stare leggendo un libro e, arrivati ad una certa pagina, di ricevere una chiamata al telefono. A questo punto ponete un segno sul vostro libro e lo chiudete, rispondete al telefono e, una volta

```

63  /* endless loop */
64  while(FOREVER)
65  {
66      /* if TMR exceed the match value */
67      if (TMR2 > TMR_MATCH_VALUE)
68      {
69          /* clear timer */
70          TMR2 = 0;
71          /* toggle RD0 */
72          PORTDbits.RD0 = ~PORTDbits.RD0;
73      }
74  }
75
76  /* execution should never reach this line */
77  return (EXIT_SUCCESS);
78  }

```

Fig. 5 - Verifica del funzionamento dell'esercizio 2.

conclusa la telefonata, riaprite il libro alla pagina contrassegnata e ricominciate a leggere. Quello che avviene all'interno di un microcontrollore quando la CPU riceve un interrupt è più o meno la stessa cosa: durante l'esecuzione, se viene segnalato un interrupt, nella CPU si eseguono le seguenti operazioni.

1. Viene salvato il contesto di esecuzione corrente (in particolare il valore corrente del Program Counter e tutti i dati necessari a riprendere correttamente l'esecuzione del processo interrotto). Questa operazione viene proprio detta "Salvataggio del contesto di esecuzione".
2. Viene interrogata una tabella, chiamata in genere Interrupt Vector Table, che contiene le

locazioni delle funzioni che servono le varie possibili richieste di interruzioni. Le richieste di interrupt sono chiamate IRQ (Interrupt ReQuest), mentre le funzioni che servono le richieste sono chiamate ISR (Interrupt Service Routines).

- Viene eseguita la ISR richiamata dalla Interrupt Vector Table. Tale funzione, in genere, è di breve durata, in modo da interferire il meno possibile con l'esecuzione corrente.
- Una volta terminata l'esecuzione della ISR, l'interrupt si definisce Servito e il processore recupera il contesto di esecuzione pre-interrupt, aggiorna il program counter con la locazione corretta e ripristina tutti i dati necessari allo svolgimento dell'attività interrotta e va avanti.

Il parallelismo con l'esempio riportato in precedenza è ora evidente: voi che leggete siete il processore, la lettura del libro è il processo in esecuzione, il segno apportato sul libro per ricordare la pagina è il salvataggio di contesto e la telefonata è l'interrupt.

Gli interrupt sono spesso utilizzati nel mondo embedded in una moltitudine di situazioni: ad esempio possono servire alla gestione di tutti gli eventi asincroni rispetto al processo di esecuzione (per citare solo alcuni esempi: la pressione di un pulsante in un'interfaccia HMI, o l'arrivo di un dato su una linea seriale).

Tuttavia questo contesto non è l'unico nel quale l'uso degli interrupt può risultare vantaggioso; ad esempio un altro possibile utilizzo degli interrupt in congiunzione con le periferiche di tipo timer, per la generazione di eventi di interrupt periodici (si pensi ad un timer che va in overflow o in period match ogni millisecondo, generando una interruzione: questo tipo di interruzione ciclica può essere utilizzato come segnale di temporizzazione del sistema).

È inoltre possibile utilizzare gli interrupt per la gestione di situazioni di errore particolari (eccezioni), demandando quindi ad una ISR la gestione del caso di errore particolare che viene a verificarsi (anche se in questo caso di parla in genere di trap e non di interrupt).

In sostanza gli impieghi degli interrupt sono molteplici, ma comunque tutti strettamente legati alle periferiche: virtualmente, quasi ogni periferica all'interno di un microcontrollore è in grado di generare uno o più interrupt.

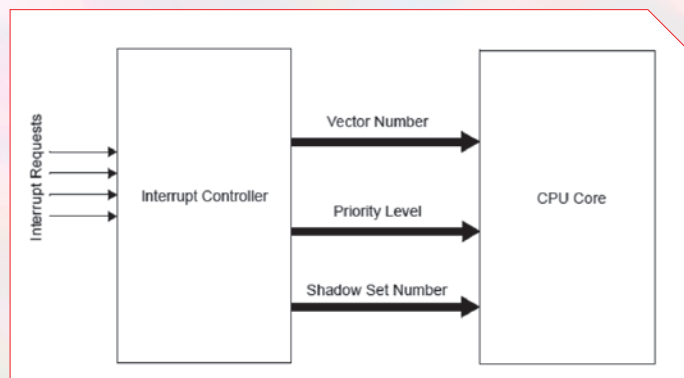


Fig. 6 – Schema a blocchi del modulo interrupt di un PIC32.

L'interrupt controller dei micro della famiglia PIC32, il cui schema a blocchi è riportato in Fig. 6, è un sistema di una certa complessità e dotato delle seguenti caratteristiche:

- fino a 96 sorgenti di interrupt;
- fino a 64 interrupt vectors;
- modalità single-vector e multi-vector;
- fino a 5 sorgenti esterne di interrupt;
- 7 livelli di priorità (priority level);
- 4 livelli di sub-priorità (sub-priority level);
- possibilità di generare interrupt da software.

La gestione e la configurazione degli interrupt è gestita tramite un particolare set di registri di controllo e di stato. Analizzeremo più in dettaglio i registri quando tratteremo l'esempio pratico, per il momento si rimandano i lettori più interessati al PIC32MX/MZ Family Reference Manual per maggiori dettagli.

## HELLO WORLD CON INTERRUPT

Vediamo adesso come sia possibile riscrivere l'esempio che abbiamo visto in precedenza utilizzando una gestione ad interrupt. Quello che vogliamo fare è, nuovamente, far oscillare il pin RD0 alla frequenza di 1 Hz. Utilizzeremo anche questa volta un timer, ma anziché impiegare il processore per determinare se il conteggio del ciclo corrente è terminato (l'istruzione if dell'esempio precedente), ci serviremo di un interrupt generato dal timer stesso, e gestiremo il cambio di stato del pin all'interno di un'apposita ISR. Inoltre vogliamo complicare leggermente l'esercizio, imponendo di utilizzare il timer in modalità 16-bit con lo stesso prescaler utilizzato in precedenza (3), in modo da dover aggiungere qualche ulteriore operazione di conteggio per arrivare alla frequenza desiderata. Creiamo quindi un nuovo progetto MPLab X, che chiamiamo MLX\_EX\_3, ed un nuovo file main al suo interno. In questo esempio avremo bisogno di utilizzare qualche funzione presente

## Listato 4

```
#define CYCLE_PER_SEC      20
#define INT_MATCH_VALUE    ((SYS_CLOCK/TMR_PRESCALER_8 -
                             VALUE)/CYCLE_PER_SEC)

#define T2_INT_DISABLE     0
#define T2_INT_ENABLE     1
#define T2_INT_PRIO_1     1
#define T2_INT_SUB_PRIO_0 0

#define LED                 PORTDbits.RD0
```

Listato 4 – Define per esercizio 3

nella libreria delle periferiche, quindi per prima cosa inseriamo la seguente inclusione all'interno del codice:

```
#include <plib.h>
```

Ricopiamo le stesse define utilizzate nell'esempio precedente, aggiungendo quelle riportate nel **Listato 4**.

Il Timer 3 dei PIC32 ha la capacità di generare un interrupt in caso di period match con il relativo registro PR2. Impostiamo quindi il timer in modalità 16-bit ed imponiamo il valore di PR2 pari alla define INT\_MATCH\_VALUE (che abbiamo calcolato per avere un periodo di interrupt pari a 50ms).

A questo punto bisogna impostare le priorità (non ha particolare importanza in questo caso, in quanto non ci sono altre sorgenti di interrupt attive al momento, decidiamo quindi di impostare un livello di priority pari ad 1 e subpriority pari a 0). Ora è sufficiente abilitare la sorgente di interrupt (Timer 3), tramite il bit T2IE del registro IEC0 ed abilitare globalmente gli interrupt (abbiamo utilizzato la funzione INTEnableSystemMultiVectoredInt presente all'interno della libreria delle periferiche).

Ora periferica e relativo interrupt sono corretta-

## Listato 5

```
/* set T1CON register */
T2CONbits.ON = TMR_ON;
T2CONbits.TCKPS = TMR_PRESCALER_8;
T2CONbits.T32 = TMR_16_BIT_MODE;
T2CONbits.TCS = TMR_CS_PBClk;
PR2 = INT_MATCH_VALUE;

/* set interrupt register */
IPC2bits.T2IP = T2_INT_PRIO_1;
IPC2bits.T2IS = T2_INT_SUB_PRIO_0;
IEC0bits.T2IE = T2_INT_ENABLE;

INTEnableSystemMultiVectoredInt();
```

Listato 5 – Configurazione registri timer ed interrupt.

## Listato 6

```
int main(void)
{
    /* set ddr register */
    TRISD = 0x0000;
    /* configure the PIC32 core for the best performance */
    SYSTEMConfigPerformance(SYS_CLOCK);

    /* set T1CON register */
    T2CONbits.ON = TMR_ON;
    T2CONbits.TCKPS = TMR_PRESCALER_8;
    T2CONbits.T32 = TMR_16_BIT_MODE;
    T2CONbits.TCS = TMR_CS_PBClk;
    PR2 = INT_MATCH_VALUE;

    /* set interrupt register */
    IPC2bits.T2IP = T2_INT_PRIO_1;
    IPC2bits.T2IS = T2_INT_SUB_PRIO_0;
    IEC0bits.T2IE = T2_INT_ENABLE;

    INTEnableSystemMultiVectoredInt();

    /* endless loop */
    while(FOREVER)
    {
        /* interrupt mode - do nothing */
    }

    /* execution should never reach this line */
    return (EXIT_SUCCESS);
}
```

Listato 6 – main function

mente configurati. Se tutto è stato eseguito correttamente, la funzione main dovrebbe assomigliare a quella riportata all'interno del **Listato 6**. Ora che la configurazione del sistema è completata, passiamo all'analisi della ISR, il cui codice è riportato all'interno del **Listato 7**.

In XC32 le ISR si identificano usando la macro `__ISR`, che predende come parametri di ingresso il vettore relativo all'intervallo e la priorità (iplx,

## Listato 7

```
void __ISR(_TIMER_2_VECTOR, ipl1)
Timer2InterruptHandler (void)
{
    /* interrupt counter */
    static unsigned int counter;

    /* increment counter */
    counter++;
    /* clear interrupt flag */
    IFS0bits.T2IF = 0;

    if (counter >= CYCLE_PER_SEC)
    {
        /* reset counter */
        counter = 0;
        /* toggle LED */
        LED = ~LED;
    }
}
```

Listato 7 – Interrupt Service Routine

in questo caso abbiamo impostato `ipl1`, come da configurazione). Dopodichè si può scegliere l'identificativo che si desidera per la ISR, in questo caso noi abbiamo scelto `Timer2InterruptHandler`. All'interno della ISR definiamo una variabile statica di conteggio (è importante che la variabile sia statica, in quanto si deve mantenere il conteggio tra un'interruzione e l'altra). Il corpo della funzione è molto semplice: sostanzialmente ad ogni occorrenza si incrementa il contatore, si resetta il flag relativo all'interrupt (`IFS0bits.T2IF = 0`, quest'operazione deve essere eseguita via software, in quanto il flag non si resetta automaticamente in hardware), e si controlla se il numero di occorrenze ha superato la soglia impostata per determinare il raggiungimento del tempo di un secondo (`if (counter > CYCLE_PER_SEC)`), e, in caso positivo, si resetta il contatore e si esegue il toggle del pin ed il ciclo ricomincia dall'inizio. Se lanciamo in esecuzione il codice con il simulatore e posizioniamo un breakpoint all'interno della ISR, dopo 50ms vedremo che l'esecuzione del codice salta all'interno della routine di interrupt, per effetto del period match tra il registro timer e PR2, come visibile in Fig. 7. Se inseriamo un secondo breakpoint all'interno dell'if, noteremo che dopo 20 occorrenze si entra all'interno di questo ramo dell'esecuzione e viene effettivamente eseguito il toggle del pin (Fig. 8).

## CONCLUSIONI

In questa seconda puntata ci siamo addentrati

```

93 void __ISR(_TIMER_2_VECTOR, ipl1) Timer2InterruptHandler (void)
94 {
95     /* interrupt counter */
96     static unsigned int counter;
97
98     /* increment counter */
99     counter++;
100     /* clear interrupt flag */
101     IFS0bits.T2IF = 0;
102
103     if (counter > CYCLE_PER_SEC)
104     {
105         /* reset counter */
106         counter = 0;
107         /* toggle LED */
108         LED = ~LED;
109     }
110 }

```

Fig. 7 – Breakpoint all'ingresso della ISR, sono trascorsi 50ms dall'inizio dell'esecuzione.

```

92 void __ISR(_TIMER_2_VECTOR, ipl1) Timer2InterruptHandler (void)
93 {
94     /* interrupt counter */
95     static unsigned int counter;
96
97     /* increment counter */
98     counter++;
99     /* clear interrupt flag */
100     IFS0bits.T2IF = 0;
101
102     if (counter >= CYCLE_PER_SEC)
103     {
104         /* reset counter */
105         counter = 0;
106         /* toggle LED */
107         LED = ~LED;
108     }

```

Fig. 8 – Breakpoint all'interno dell'if.

più in profondità della scrittura di software embedded per PIC32 tramite MPLab X, analizzando i timers e prendendo confidenza con il concetto, assolutamente fondamentale nel mondo embedded, di interrupt.

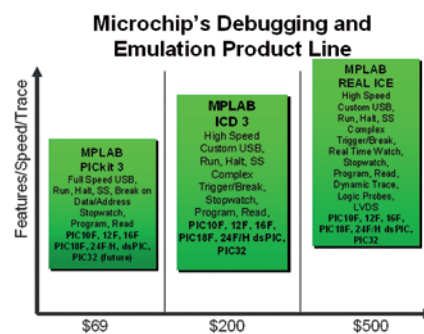
Nella prossima puntata continueremo il nostro viaggio analizzando alcune periferiche analogiche dei microcontrollori PIC32, come il convertitore analogico\digitale ed il modulo PWM. ■

## EMULATORI E DEBUGGER

A supporto dei microcontrollori che produce, la Microchip commercializza una gamma completa di emulatori e debugger, in grado di soddisfare le esigenze più disparate per quanto concerne prezzi dei tool e prestazioni. Il prodotto entry level per il debugging è il PicKit, ormai arrivato alla sua terza edizione (PicKit3); si tratta di un dispositivo con interfaccia USB che funziona sia da programmatore

che da debugger di base (comunque in grado di supportare breakpoint, lettura memoria, Stopwatch, ecc). Il fratello maggiore del PicKit è l'ICD (In-Circuit Debugger), anch'esso arrivato alla versione 3; si tratta di un dispositivo più performante rispetto al precedente, che supporta un set avanzato di funzioni di Breakpoint/Trigger ed è più veloce e protetto rispetto al pickit. Infine, il dispositivo di

Fig. 11  
Linea prodotti  
Microchip per  
Emulazione e  
Debugging.



più alta gamma fornito dalla Microchip per il debugging è il Real ICE (In-Circuit Emulator). Questo è un dispositivo

di livello professionale, che supporta oltre alle funzioni tipiche dei debuggers anche la funzione di trace.

# CORSO

# MPLAB X

# 3

di FRANCESCO FICILI  
e VINCENZO GERMANO

**Proseguiamo lo studio di MPLab X, il nuovo ambiente di sviluppo integrato di Microchip Technology, che soppianta il vecchio MPLab IDE. Approfondiamo la conoscenza di alcune periferiche di base dei microcontrollori PIC32, i primi dispositivi a 32-bit prodotti da Microchip. Terza puntata.**

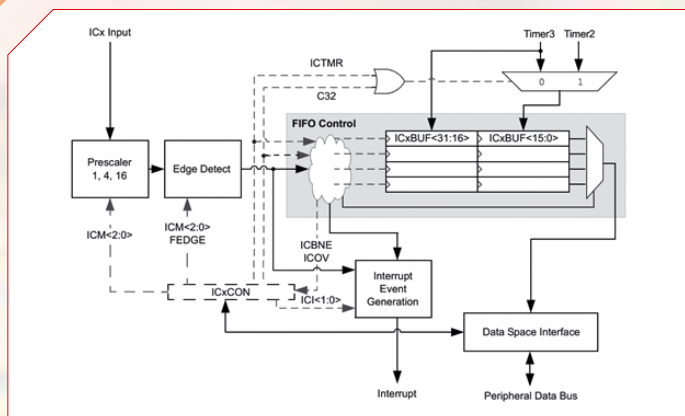
**N**ella puntata precedente abbiamo approfondito la nostra conoscenza dei microcontrollori PIC32 della Microchip e di alcune loro periferiche di base: i Timer e gli Interrupt. Ne abbiamo analizzato e compreso le principali caratteristiche e abbiamo imparato a utilizzarli, tramite facili esempi, grazie al software MPLab X. In questa terza puntata focalizziamo la nostra attenzione su altre periferiche di base dei PIC32, che verranno applicate e utilizzate su una demoboard pensata, progettata e realizzata appositamente per questo corso.

## **LE PERIFERICHE INPUT CAPTURE E OUTPUT COMPARE**

Oltre alla gestione dei Timer, come visto nella

scorsa puntata, il microcontrollore PIC32 fornisce due periferiche di base che migliorano il funzionamento dei Timer e sono in grado di semplificare svariate applicazioni, trovando largo impiego nella registrazione di eventi, nella misurazione di periodi di tempo, nei conteggi, nella generazione di impulsi e di forme d'onda periodiche; tali periferiche sono la Input Capture e la Output Compare.

Ma vediamo maggiormente nel dettaglio per comprenderne meglio il funzionamento. Utilizzando la periferica Input Capture (IC) i registri funzionano come un cronometro, infatti quando si verifica un particolare evento, il tempo dell'evento viene registrato e il clock continua a funzionare normalmente (un cronometro,



**Fig. 1 - Schema a blocchi di un Input Capture.**

d'altra parte, si ferma quando viene registrato il tempo dell'evento); ciò torna molto utile nelle applicazioni che prevedono la gestione della frequenza (periodo) e una misura degli impulsi. Con riferimento alla **Fig. 1**, tale modulo cattura e registra il valore a 16-bit o 32-bit del Timer selezionato, quando si verifica un evento sul pin ICx. Su ogni canale si può scegliere tra uno dei due temporizzatori a 16-bit (Timer2 o Timer3) per un tempo base, o due temporizzatori a 16-bit (Timer2 e Timer3) insieme per formare un Timer a 32-bit. Per il Timer selezionato si può utilizzare un clock sia interno che esterno. L'evento che scatena la cattura può essere il fronte di discesa del segnale sul pin ICx, quello di salita sullo stesso, o entrambi, con la possibilità di specificare anche il primo fronte del segnale. In più, grazie al blocco del Prescaler è possibile avere come evento scatenante il quarto o il sedicesimo fronte di salita del segnale d'ingresso. Per il PIC32MX sono disponibili cinque periferiche Input Capture, ognuna controllata da un apposito registro ICxCON, in combinazione con uno o due Timer (Timer2 o Timer3). Questi moduli sono largamente utilizzati in funzioni ripetitive nel tempo, ma anche per determinare le lunghezze degli impulsi PWM allo scopo di poter attivare o controllare hardware esterno dopo un ritardo specifico. In ogni caso, il Timer corrispondente deve essere abbastanza veloce, per avere un valore significativo con una risoluzione adeguata e ottenere una rappresentazione accurata della lunghezza dell'impulso. Generalmente, dopo che si scatena una delle condizioni di cattura viste precedentemente, il modulo modifica lo stato dei suoi registri interni quando il contenuto del Timer corrisponde al valore desiderato e successivamente viene generato

un interrupt. Come mostrato nella Fig. 1, c'è anche la possibilità di un buffer FIFO (First In First Out) a quattro posizioni che permette di configurare la generazione dell'interrupt dopo il riempimento di 1, 2, 3 o 4 posizioni. Quindi il valore corrente del Timer selezionato viene registrato e memorizzato in un buffer FIFO, da recuperare leggendo il corrispondente registro ICxBUF. Nel caso della periferica Output Compare (OC), la sua funzionalità cambia e viene utilizzata per generare un singolo impulso o una serie di impulsi in risposta agli eventi di tempo selezionati. Questa modalità viene utilizzata per attivare o controllare hardware esterno con un ritardo specifico. Facendo riferimento alla **Fig. 2**, per tutte le modalità di funzionamento vengono confrontati i valori memorizzati nel registro OCxR e/o OCxRS con il valore selezionato nel Timer; non appena si verifica una corrispondenza, questo modulo genera un evento (interrupt) basato sulla modalità di funzionamento preimpostata. Tra le sue caratteristiche principali troviamo:

- generazione di interrupt programmabili;
- modalità di confronto singole e duali;
- generazione di impulsi di uscita singoli e continui;
- rilevamento di errori PWM hardware e disabilitazione dell'uscita in automatico;
- selezione programmabile della base di tempo a 16-bit o 32-bit.

La famiglia di microcontrollori PIC32MX offre cinque periferiche OC, le quali, come abbiamo visto, possono essere utilizzate per una varietà di applicazioni, tra cui: la generazione di impulso singolo, generazione di impulsi continua e modulazione di larghezza di impulso (PWM - Pulse Width Modulation). Ciascun modulo può essere associato a uno dei due temporizzatori a 16 bit (Timer2 o Timer3) o a un Timer a 32 bit (ottenuto per combinazione Timer2 e Timer3) e dispone di un pin di uscita che può essere configurato in base alle necessità per produrre fronti di salita o di discesa del segnale. È importante notare che ogni modulo ha un vettore di interrupt associato e indipendente.

La configurazione di base di questa periferica viene eseguita mediante il registro OCxCON, che ci permette di scegliere la modalità di

funzionamento desiderata. Quando viene utilizzato in modalità impulso continuo (OCM=101), in particolare, il registro OCxR viene utilizzato per determinare l'istante (rispetto al valore del temporizzatore associato) in cui il pin d'uscita viene impostato, mentre il registro OCxRS determina quando il pin di uscita viene resettato.

### LA PERIFERICA PWM

Come abbiamo visto, la periferica Output Compare può essere utilizzata come generazione di impulso singolo, generazione di impulsi continua e modulazione di larghezza di impulso (PWM). Possiamo affermare che la configurazione PWM è molto utile in svariate applicazioni ed è forse quella di gran lunga più utilizzata tra le tre.

La modalità adottata per la generazione di un segnale PWM è una combinazione tra la normale esecuzione di un Timer e la modalità Output Compare, grazie alla quale è possibile erogare un segnale PWM utilizzando il reset di un Timer a un valore specifico. Questo aspetto sarà maggiormente comprensibile nell'esempio di utilizzo del PWM.

Il modo in cui un segnale PWM funziona è piuttosto semplice: un impulso viene prodotto a intervalli regolari (T), tipicamente forniti da un Timer e da un suo registro di periodo; la larghezza di impulso (Ton), però, non è fissa ma è programmabile e può variare tra 0% e 100% del periodo del Timer. Il rapporto tra la larghezza dell'impulso (Ton) e il periodo del segnale (T) viene chiamato duty-cycle, come riportato in Fig. 3. Quindi un'uscita PWM è fondamentalmente un'onda quadra con un periodo e un ciclo di lavoro specifico.

I due valori limite del duty-cycle sono 0% e 100%: il primo corrisponderebbe a un segnale nullo, mentre il secondo corrisponde a un segnale di uscita costante.

Per quanto riguarda la risoluzione del PWM, tipicamente viene indicata come il logaritmo in base 2: per esempio, se ci sono 256 possibili ampiezze di impulso, diciamo che abbiamo un segnale PWM con una risoluzione di 8 bit.

### ESEMPIO DI UTILIZZO DEL PWM

Passiamo a un esempio pratico di generazione del segnale PWM mediante i moduli OC del microcontrollore PIC32MX, dove genere-

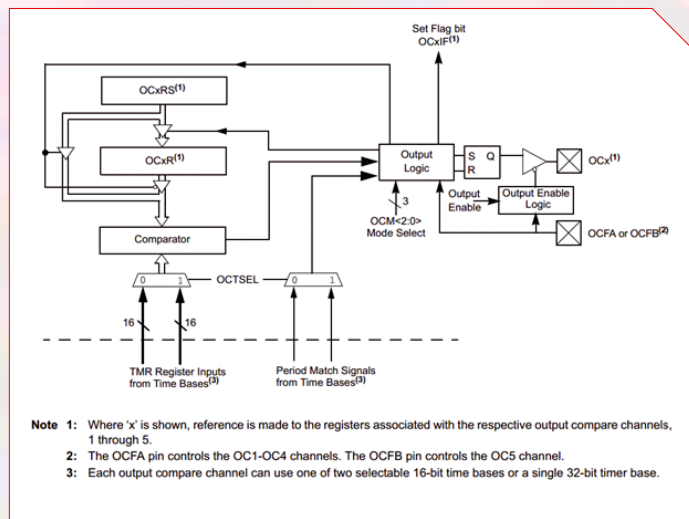


Fig. 2 - Schema a blocchi di un Output Compare.

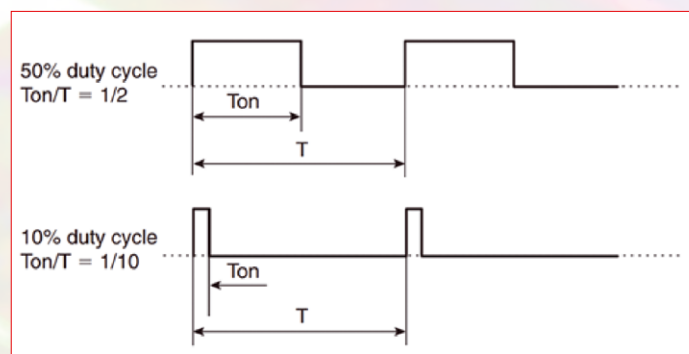


Fig. 3 - Segnale PWM di riferimento.

remo un flusso continuo di impulsi con duty-cycle desiderato.

Tutto quello che dobbiamo fare per inizializzare il modulo OC e generare un segnale PWM è impostare i tre bit del registro OCM all'interno del registro di controllo OCxCON; a riguardo, fate riferimento alla Tabella 1. Per ottenere un segnale PWM di base, conviene configurarlo come 0x110. È prevista una seconda modalità PWM (0x111) ma nel nostro caso non traiamo alcuna utilità nell'abilitare i "Fault" (guasto) sul pin, comunemente richiesto da diverse applicazioni come possono essere i meccanismi di protezione (ad esempio controllo motore o conversione di potenza).

Successivamente abbiamo bisogno di selezionare il Timer su cui basare il periodo di PWM, scegliendo tra Timer2 e Timer3. Nel caso in cui vogliamo essere in grado di produrre un periodo PWM a 40 kHz e supponendo un clock periferico di 16 MHz, impostando un prescaler a un rapporto 1:1 otteniamo un

Tabella 1 - Registro di controllo per la periferica Output Compare.

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
23:16	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
15:8	R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
	ON <sup>(1)</sup>	—	SIDL	—	—	—	—	—
7:0	U-0	U-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	OC32	OCFLT <sup>(2)</sup>	OCTSEL	OCM<2:0>		

periodo di 400 cicli. Da questo valore deriva anche la risoluzione del duty-cycle relativo al PWM per il modulo di Output Compare: infatti, dal momento che avremo 400 possibili valori del ciclo di lavoro, siamo in grado di avere una risoluzione compresa tra 8 e 9 bit, più di 256 ( $2^8$ ) passi ma meno di 512 ( $2^9$ ).

Dopo che il timer scelto viene configurato e poco prima di scrivere nel registro OCxCON, bisogna impostare, per la prima volta, il valore del duty-cycle nel registro OCxR e nel registro OCxRS. In modalità PWM, i due registri lavoreranno in configurazione master/slave e una volta che il modulo PWM verrà avviato, saremo in grado di cambiare il ciclo di lavoro scrivendo solo nel registro OCxRS. Il registro OCxR sarà aggiornato, copiando un nuovo valore dal OCxRS, solo e precisamente all'inizio di ogni nuovo periodo, allo scopo di evitare difetti (glitches) e avere un intero periodo (T) di tempo per preparare il prossimo valore del duty-cycle. Tutto questo sarà maggiormente chiaro nell'esempio *Dimmer digitale (LED con PWM da potenziometro)* che vedremo più avanti.

### LE PERIFERICHE ANALOGICHE DEI PIC32

Guardandoci intorno comprendiamo subito che viviamo in un mondo analogico, infatti le varie grandezze come temperatura, umidità, pressione ma anche tensioni e correnti sono analogiche. Se vogliamo che le nostre applicazioni elettroniche debbano interagire con il mondo esterno, abbiamo la necessità di interpretare le informazioni analogiche e convertirle nel mondo digitale in modo tale che un microcontrollore possa essere in grado di elaborarle e produrre eventualmente ancora un'uscita analogica. Il modulo convertitore analogico/digitale (ADC, acronimo di Ana-

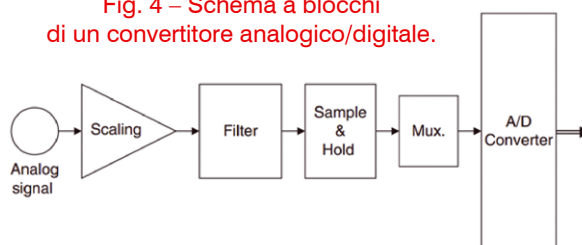
log to Digital Converter) è una delle interfacce chiave con il mondo "reale" e la famiglia PIC32MX è stata progettata per affrontare al meglio la natura analogica di questo mondo. Un convertitore veloce analogico/digitale, in grado di 500.000 conversioni al secondo, è disponibile su tutti i modelli con un multiplexer di ingresso che consente di monitorare un numero di ingressi analogici in modo rapido e con alta risoluzione. Passiamo ad analizzare le caratteristiche più salienti di questa periferica di base dei microcontrollori e nello specifico del PIC32.

#### La periferica ADC

Il convertitore analogico/digitale è una periferica che converte una tensione di ingresso analogica in un numero digitale in modo tale che possa essere elaborato da un microcontrollore o qualsiasi altro sistema digitale. Possiamo classificare le tensioni gestite dagli ADC come: unipolari e bipolari. Nel primo caso, vengono accettate tensioni di ingresso con una sola polarità, mentre nel secondo caso il range delle tensioni gestite è sia positivo che negativo. In linea generale, in Fig. 4 sono riportate le fasi tipiche coinvolte nella lettura e la conversione di un segnale analogico in forma digitale, un processo noto anche come "conditioning".

I segnali ricevuti dai sensori, di solito devono

Fig. 4 – Schema a blocchi di un convertitore analogico/digitale.



essere elaborati prima di essere processati da un convertitore ADC. Questo trattamento di solito inizia con lo "scalare" (Scaling) il segnale al valore di tensione corretto. Le componenti del segnale indesiderate vengono rimosse filtrando il segnale mediante filtri classici (ad esempio, un filtro passa-basso). Prima che il segnale entri in un convertitore ADC, esso viene fatto passare attraverso un dispositivo di campionamento e tenuta (Sample&Hold – S&H), utile per leggere segnali veloci (in tempo reale) il cui valore può cambiare tra gli istanti di campionamento; l'S&H è sostanzialmente uno switch bipolare solid state all'uscita del quale c'è un condensatore, che si carica al valore istantaneo del segnale quando lo switch è chiuso, mantenendo una tensione costante durante il processo di conversione.

Continuando lungo la catena di acquisizione mostrata in Fig. 4, molte applicazioni potrebbero necessitare di vari ADC e per sopperire a tale necessità normalmente viene utilizzato un multiplexer analogico (MUX) in ingresso al convertitore, che seleziona solo un segnale alla volta e lo porta in ingresso all'ADC. L'uscita del convertitore è digitale e tipicamente "parallela" digitale.

Possiamo riassumere il processo di conversione mediante un microcontrollore:

1. si applica il segnale da elaborare all'ingresso dell'ADC;
2. si avvia la conversione;
3. si attende fino a quando la conversione è completa;
4. si legge il dato digitale convertito.

La conversione viene avviata attivando il convertitore, che, a seconda della sua velocità, condiziona la velocità del processo di conversione stesso. Al termine della conversione, il convertitore genera un interrupt per indicare che la conversione è completa e i dati di uscita convertiti possono essere letti dal dispositivo digitale collegato all'ADC.

Facendo riferimento alla Fig. 5, in cui è riportato lo schema a blocchi dell'ADC a 10-bit di un PIC32, notiamo che ha fino a 16 pin di ingresso analogici, chiamati con AN0-AN15, oltre a due pin di ingresso analogici per l'applicazione di riferimenti di tensione esterni, che possono venire condivisi con altri pin di

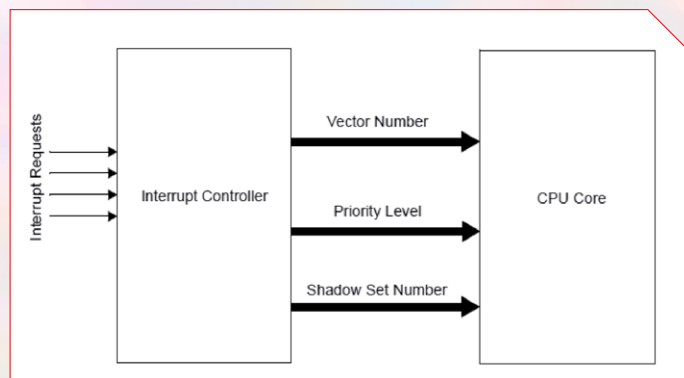


Fig. 5 – Schema a blocchi di un modulo ADC.

ingresso analogici e possono essere comuni ad altri riferimenti di moduli analogici. Quanto analizzato precedentemente in linea generale, lo riscontriamo anche in quest'architettura; infatti gli ingressi analogici sono collegati tramite due multiplexer (MUX) e possono essere commutati tra due set di ingressi. In più, si può notare la presenza del blocco di campionamento e tenuta (S&H). In una tale architettura è presente un registro di controllo che specifica quali canali di ingresso analogici saranno inclusi nella sequenza di scansione. Infine, dal diagramma a blocchi si evince che l'ADC è collegato ad un buffer (16-word) e ogni risultato a 10 bit viene convertito in uno degli otto formati di uscita a 32 bit quando viene letto dal buffer.

Riassumendo brevemente le caratteristiche di questo modulo interno al PIC32, abbiamo:

- conversione con registri ad approssimazioni successive (SAR - Successive Approximation Register);
- fino a 1 Msps di velocità di conversione e fino a 16 pin di ingresso analogici;
- pin d'ingresso per un riferimento di tensione esterna;
- sorgente di trigger selezionabile per la conversione;
- modalità di riempimento del buffer selezionabile;
- funzionamento durante la modalità di sospensione (Sleep) e inattività (Idle).

Per maggiori dettagli sulle caratteristiche/funzionalità e sul loro utilizzo, fate riferimento al data-sheet reperibile sul sito ufficiale della Microchip Technology.

## **DIMMER DIGITALE (LED CON PWM DA POTENZIOMETRO)**

Passiamo adesso a illustrare un semplice esempio di come pilotare un LED mediante



ga traccia delle modifiche di tali bit, colorando di rosso le impostazioni cambiate rispetto a quelle predefinite. Infine, grazie al tasto "Generate Source Code to Output" è possibile copiare e incollare il codice che viene automaticamente generato (secondo le impostazioni scelte) direttamente nel file d'interesse, il

quale può essere il main o un file distinto per una più corretta e pulita gestione del codice. A questo punto non ci resta che passare alla scrittura del codice relativo al nostro esempio di pilotaggio LED mediante potenziometro. Avendo già visto negli esempi delle precedenti puntate come si configurano i Timer e

## DEMOBOARD PIC32

La "Demoboard PIC32" è stata pensata e realizzata in modo da poter essere utilizzata come starter kit e quindi come strumento di supporto hardware per questo corso; prevede tutte le periferiche base, più alcune periferiche avanzate, che verranno analizzate in dettaglio nel corso delle varie puntate. Come si vede nello schema a blocchi in questo riquadro, il cuore di tutta la scheda è il microcontrollore Microchip Technology PIC32MX795F512L, dotato di 512k di Flash e 128k di RAM (più 12k di auxiliary FLASH).

La board dispone di connettività Ethernet e USB, in modo da poter fungere da piattaforma dimostrativa per questa tipologia di applicazioni.

La connessione USB è sia di tipo device (per poter supportare classi quali la HID, la CDC e la MSB, oltre a poter essere utilizzata come bus per il Bootloading), sia di tipo Host.

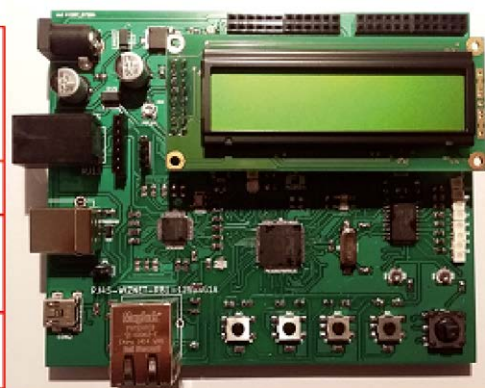
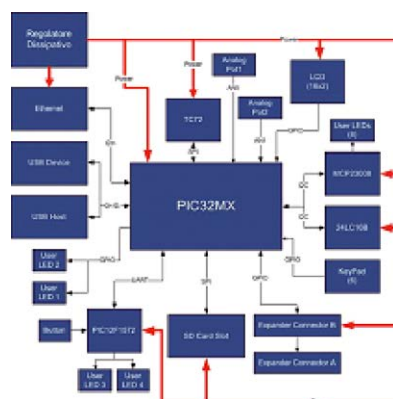
Le tensioni stabilizzate di alimentazione della scheda sono 5V e 3,3V.

Dallo schema a blocchi qui accanto, si nota facilmente che la MCU PIC32MX ha le seguenti connessioni principali:

- termometro a stato solido TC72 e slot per memoria SD mediante protocollo SPI (Serial Peripheral Interface);
- Port Expander MCP23008 (collegato a sua volta a 8 LED) e a una memoria EEPROM i2c 24LC16B mediante una piccola rete I<sup>2</sup>C, grazie alla quale è possibile sviluppare e verificare applicazioni I<sup>2</sup>C;
- microcontrollore PIC12F1572, attraverso la por-

ta UART, che a sua volta è collegato a un pulsante e due LED (gestibili anche mediante l'utilizzo di PWM);

- due potenziometri collegati ad altrettanti ingressi analogici;
- due user LED generici (User1 e User2), collegati direttamente a due pin del microcontrollore principale, utilizzabili anche mediante appositi PWM;
- display da 16x2 segmenti per permettere la visualizzazione di alcune informazioni d'interesse.



Schema a Blocchi e scheda assemblata della DemoboardPIC32.

La scheda prevede due connessioni per la programmazione dei microcontrollori, un header a sei poli per la programmazione mediante PicKit e un connettore RJ11 per la programmazione mediante ICD. In aggiunta, mediante uno switch sulla scheda è possibile selezionare quale dei due microcontrollori programmare.

Completa la dotazione della demoboard, un header di espansione a quaranta contatti su due file (20x2).

## Listato 1 – Configurazione della periferica OC2

```

OC2CON = 0x0000;           // Turn off the OC2 when performing the setup
OC2R = 0x0064;             // Initialize primary Compare register
OC2RS = 0x0064;            // Initialize secondary Compare register
OC2CON = 0x0006;           // Configure for PWM mode without Fault pin enabled
OC2CONSET = 0x8000;        // Enable OC2

```

gli Interrupt, tralasciamo questo passaggio in quanto le loro impostazioni non cambiano. Il primo passo da compiere consiste nel verificare gli ingressi e le uscite d'interesse sul microcontrollore della DemoboardPIC32 dove sono collegati; nel nostro caso, volendo pilotare con un segnale PWM il LED (identificato sulla scheda come D1), dobbiamo abilitare e configurare il modulo Output Compare corrispondente. Guardando sul datasheet del microcontrollore su quale pin è collegato D1, si nota che la periferica OC corrispondente è la OC2, quindi passiamo a configurarla come segue nel **Listato 1**.

È buona norma spegnere sempre le periferiche prima di configurarle, cosa che attuiamo con la prima istruzione. Successivamente, come visto nella trattazione del modulo dobbiamo inizializzare i registri OC2R e OC2RS per la comparazione dei valori e la generazione del PWM.

Con la penultima istruzione configuriamo la periferica a lavorare come generatore di PWM mediante il registro OC2CON e infine la riattiviamo per poterla utilizzare.

L'ultimo passo da compiere per poter avere in uscita il PWM desiderato è legato all'impostazione del registro OC2RS. Volendo ottenere un segnale variabile e pilotabile mediante potenziometro, non dobbiamo far altro che acquisire il valore del potenziometro

mediante ADC e collegarlo a tele registro.

Ancora una volta bisogna verificare mediante datasheet del microcontrollore su quale pin è collegato il potenziometro d'interesse e configurarlo come ADC. Nel nostro caso vogliamo utilizzare il potenziometro che sulla DemoboardPIC32 è indicato come RV1, perciò configuriamo la periferica come riportato nell'esempio associato a questo corso e leggiamo il valore acquisito come riportato nel **Listato 2**.

Una volta memorizzato il valore del potenziometro nella variabile "pot", non ci resta che associarla al registro OC2RS per completare questo quarto progetto pratico e avere un LED pilotato (mediante PWM variabile) con un potenziometro.

### CONCLUSIONI

In questa terza puntata abbiamo analizzato e compreso la teoria che c'è dietro ad alcune periferiche di base dei microcontrollori PIC32 come i convertitori analogico-digitale, Input Capture e Output Compare, per poi passare a utilizzarli nella pratica mediante MPLab X e la DemoboardPIC32.

Infine, con tutti questi strumenti abbiamo realizzato il "Dimming di un LED". Nella prossima puntata continueremo il nostro viaggio analizzando altre periferiche dei PIC32, come SPI, I<sup>2</sup>C e USART. ■

## Listato 2 – Lettura e acquisizione valore del potenziometro

```

while ( !mAD1GetIntFlag() )
{
    // Wait for the first conversion to complete so there will be valid data in ADC _
    // result registers
}
// Read the result of pot conversion from the idle buffer
pot = ReadADC10(offset);
mAD1ClearIntFlag();
}

```

# CORSO

# MPLAB X



di FRANCESCO FICILI  
e VINCENZO GERMANO

**Continuiamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip Technology. In questa puntata ci occupiamo dell'analisi delle più comuni periferiche di comunicazione. Quarta puntata.**

**N**elle varie applicazioni embedded di cui ci occupiamo o che realizziamo, esistono delle varianti che sono caratteristiche dell'applicazione stessa (come un algoritmo particolare, o un particolare tipo di sensore o attuttore), ma ci sono anche diversi aspetti ricorrenti, in quanto si tratta di qualcosa che nel corso degli anni è stato in qualche modo standardizzato, in maniera da renderne più semplice l'utilizzo nelle applicazioni. In questo secondo gruppo rientrano certamente le varie periferiche di comunicazione che sono utilizzate dalle unità "intelligenti" per comunicare e scambiare reciprocamente dati. Come sappiamo, gli standard di comunicazione digitale esistenti ad oggi sono svariati e possono essere più o meno

complessi a seconda del livello al quale si trovano le unità da mettere in comunicazione (a livello scheda, comunicazione tra sottosistemi, interconnessione di sistemi e via dicendo, fino ad arrivare, ad esempio, alle moderne reti satellitari).

In questa puntata ci occuperemo delle periferiche di comunicazione di basso livello che possiamo comunemente trovare già implementate nell'hardware dei PIC32. Si consideri che queste periferiche, oltre ad essere una possibile scelta per l'interconnessione tra diverse unità logiche programmabili, costituiscono lo standard per l'interfacciamento della stragrande maggioranza dei sensori, delle memorie e dei moduli specializzati.

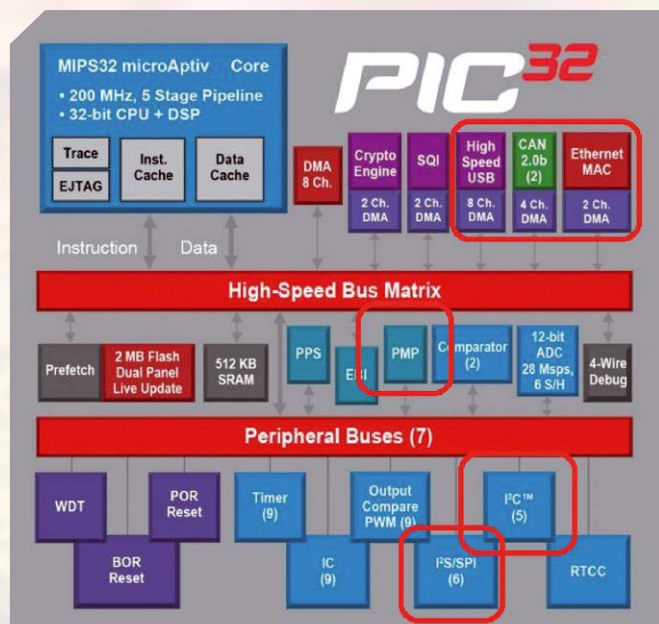


Fig. 1 - Esempio di periferiche di comunicazione in una architettura PIC32.

## LE PERIFERICHE DI COMUNICAZIONE DEI PIC32

Un'architettura PIC32 può inglobare al suo interno un set molto esteso di periferiche di comunicazione, che possono variare dalle semplici periferiche seriali o parallele per il colloquio tra circuiti integrati, fino alle complesse interfacce per la comunicazione tra sottosistemi. In Fig. 1 è riportata un'architettura PIC32 nella quale sono evidenziate alcune periferiche di comunicazione.

Alcuni esempi di periferiche di base sono:

- UART – Universal Asynchronous Receiver and Transmitter;
- SPI – Serial Peripheral Interface;
- IC – Inter Integrated Circuits;
- PMP – Parallel Mater Port;
- IS – Inter Integrated Sound.

Tra gli esempi di periferiche di comunicazione più complesse, possiamo annoverare:

- USB – Universal Serial Bus;
- CAN – Controller Area network;
- Ethernet.

Spesso queste ultime, per questioni legate alle prestazioni e alla gestione, sono associate ad uno o più canali DMA (Direct Memory Access) e sono anche supportate da stack per la gestione del protocollo di comunicazione, che in alcuni casi (come ad esempio l'USB e l'Ethernet) richiede strati (layer) aggiun-

tivi rispetto a quelli implementati a livello hardware e che quindi devono necessariamente essere inseriti nel software.

In questa puntata del corso prenderemo in esame solo le periferiche di base, rimandando l'analisi di alcune delle periferiche più complesse alle prossime puntate.

Passiamo adesso ad una breve descrizione della caratteristiche peculiari delle periferiche di comunicazione di base.

## LA PERIFERICA SPI

Lo standard SPI (acronimo di Serial Peripheral Interface), o anche bus SPI, è uno standard di collegamento tra unità logiche (microcontrollori, sensori di vario tipo, memorie), di tipo seriale sincrono, Master/Slave multidrop full-duplex, ideato dalla Motorola negli anni '80 del secolo scorso (ne esiste anche una variante della National Semiconductor, che prende il nome di Microwire). Il sistema si basa essenzialmente su quattro segnali:

- MOSI = Master Out Slave In; è la linea attraverso cui il Master invia dati ad un'unità Slave.
- MISO = Master In Slave Out; è la linea attraverso cui uno slave Slave invia dati al Master.
- SCK = Serial Clock; è la linea di clock, gestita unicamente dal Master, che lo propaga a tutti gli Slave.
- CS = Chip Select; è la linea attraverso la quale il Master "attiva" un determinato Slave (in un bus SPI possiamo trovare

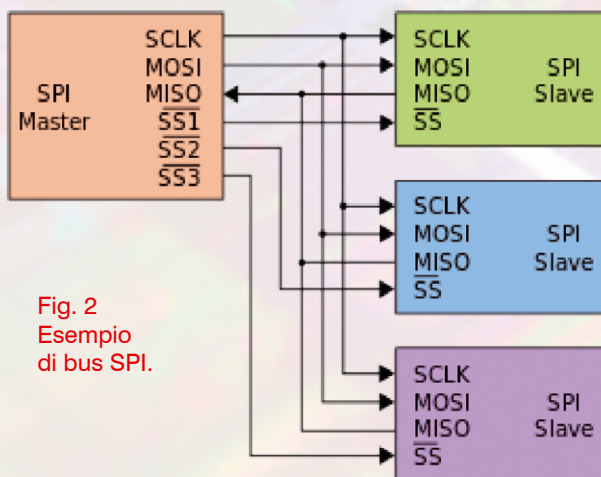


Fig. 2  
Esempio di bus SPI.

fino ad un massimo di una linea per ogni dispositivo slave).

I nomi utilizzati per identificare i segnali possono variare: ad esempio la coppia MISO/MOSI è spesso anche indicata come SDI/SDO (Serial Data Input/Output), mentre la linea CS è spesso indicata anche come SS (Slave Select). In Fig. 2 è rappresentato lo schema di collegamento tra un master e tre dispositivi slave.

Come è facile notare, per garantire la comunicazione sono necessarie almeno tre linee (MISO, MOSI, SCK) più, chiaramente, un riferimento comune (GND). Tuttavia, nel caso in cui si voglia utilizzare la linea di CS per attivare/disattivare gli slave, questo ha un costo di una linea per slave, il che può far facilmente salire il numero di interconnessioni (nel nostro esempio il consumo totale di linee è 6, dato che viene utilizzata una linea di SS dedicata per ogni slave). Nel caso non si voglia utilizzare la linea di CS, bisognerà prevedere uno strato aggiuntivo nel protocollo per la gestione dell'indirizzamento. Il funzionamento si basa sul principio dei registri a scorrimento (shift-register). Ogni dispositivo conta su un registro a scorrimento i cui bit vengono emessi e, contemporaneamente, immessi, attraverso le linee dati (MISO/MOSI), al ritmo di un bit ad ogni ciclo del segnale di clock (SCK). La dimensione degli shift-register può essere arbitraria (anche se usualmente è di 8 bit); l'unico vincolo è, chiaramente, che sia la stessa per il dispositivo master e per gli slave. Il dato di output è emesso sempre in corrispondenza del primo impulso di clock e la comunicazione può venire intrapresa solo dal master, che allo scopo, tipicamente abilita lo slave tramite la linea SS e successivamente impone il clock sulla linea dedicata. Con questa procedura ha inizio lo scambio dati, che avviene con la tecnica descritta in precedenza. Alla fine di ogni transazione, il contenuto del registro dello slave sarà passato al master e viceversa. Con opportune parole identificative si possono inviare comandi al dispositivo ricevente, che potrà effettuare l'elaborazione assegnata ponendo quindi nel suo shift-register il dato richiesto che al prossimo ciclo di trasmissione verrà trasmesso al richieden-

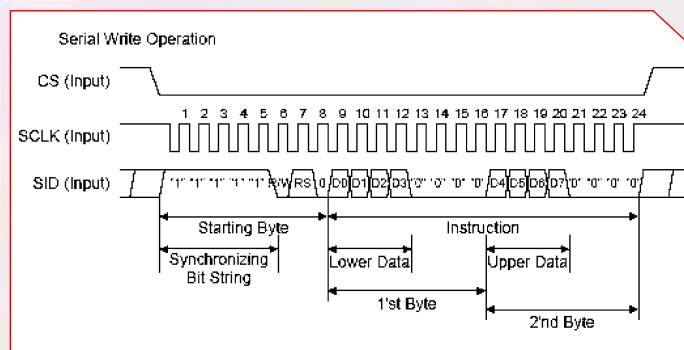


Fig. 3 - Esempio di scambio dati su bus SPI.



Fig. 4  
Logo dello standard I²C

te. In Fig. 3 è rappresentato un esempio di comunicazione su bus SPI.

Il collegamento SPI garantisce velocità di scambio dati anche abbastanza sostenute (si può arrivare fino ai 10 Mbit/s ed oltre), ed è quindi spesso utilizzato per applicazioni che richiedono un throughput elevato, come ad esempio l'interfacciamento di convertitori ADC o di memorie RAM/FLASH.

### LA PERIFERICA I²C

Lo standard I²C-Bus è un protocollo di collegamento tra dispositivi elettronici, di tipo seriale sincrono, master/slave multidrop e half-duplex. È stato standardizzato dalla divisione semiconduttori della Philips (oggi NXP) negli anni '80 del secolo scorso, ma era già largamente impiegato dall'azienda stessa come standard di interfacciamento per una vasta gamma di dispositivi prodotti (principalmente memorie, sensori, driver, interfacce, ecc...). Durante il corso degli anni ne sono state prodotte diverse declinazioni per applicazioni specifiche: l'esempio più evidente è lo standard SMBUS, brevettato dalla Intel, che viene largamente impiegato nella gestione di alcuni chip presenti nelle moderne schede madri di PC e laptop, in particolare per la gestione dei sottosistemi energetici. In Fig. 4 è rappresentato il logo dello standard. Il sistema di basa su due segnali di comunicazione:

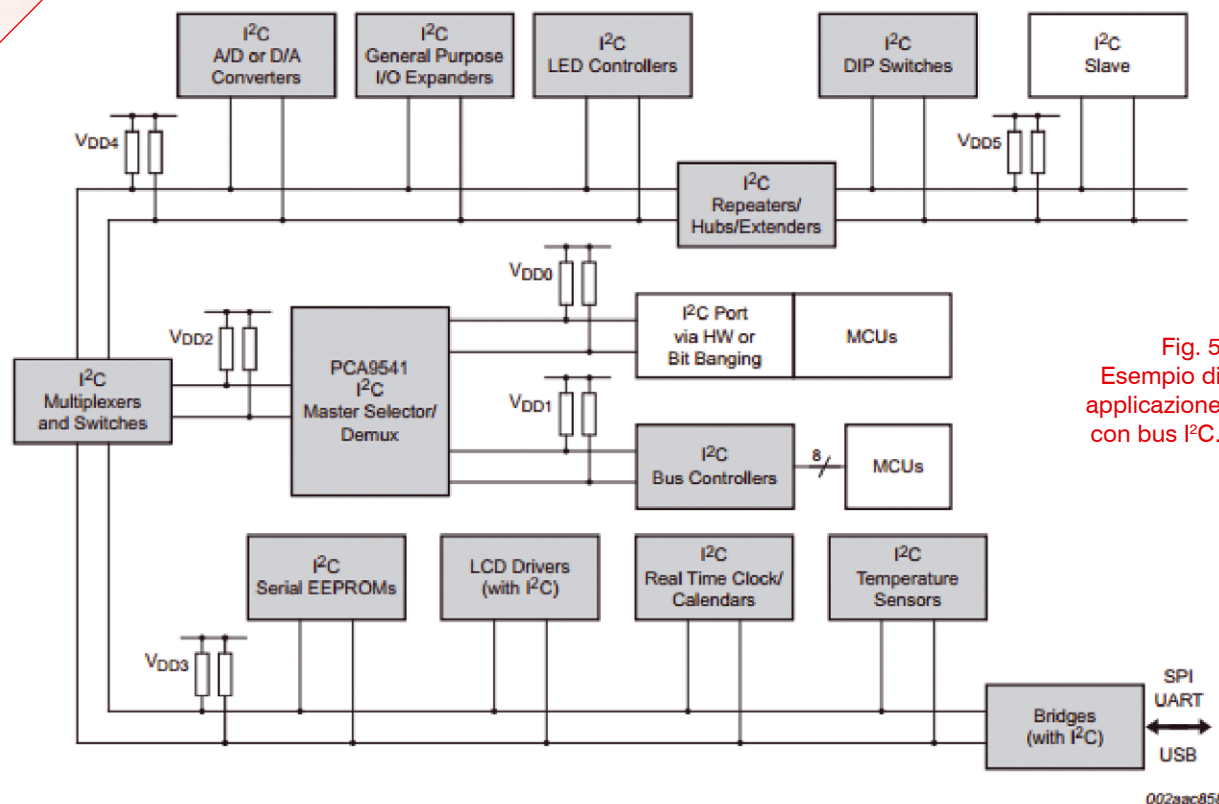
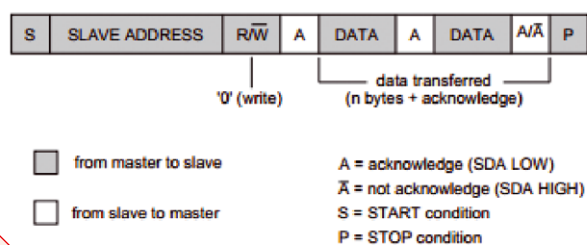


Fig. 5  
Esempio di  
applicazione  
con bus I²C.

- SDA = Serial Data; è la linea utilizzata per lo scambio dati;
- SCL = Serial Clock; è la linea attraverso la quale viene distribuito il segnale di clock.

Richiedendo solo due segnali per poter funzionare (oltre, naturalmente, a un riferimento comune che è GND) il bus viene spesso anche nominato 2-wire (TWI=Two Wire Interface). Dal punto di vista fisico, i driver delle linee SDA ed SCL sono realizzati tramite uno stadio di uscita di tipo Open Collector, quindi i dispositivi possono solo imporre un livello logico basso, mentre l'impostazione del livello logico alto si realizza lasciando libera la linea che viene portata a livello alto da opportune resistenze di pull-up collegate al polo positivo della tensione

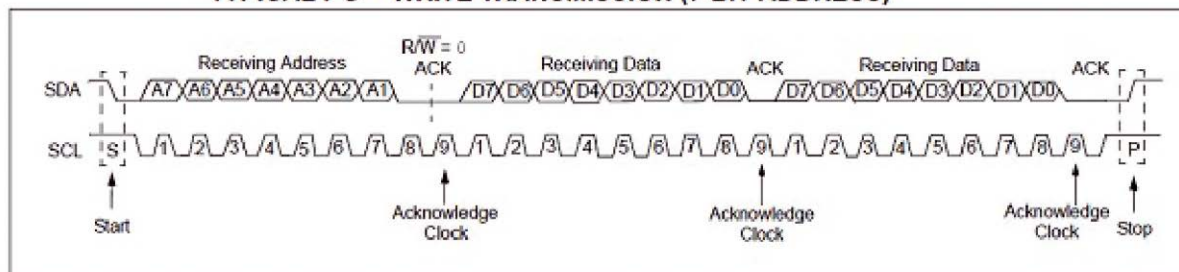
Fig. 6  
Esempio  
di  
pacchetto  
dati I²C.



di alimentazione. In Fig. 5 è rappresentato un esempio di applicazione che fa largo uso di collegamenti I²C-Bus.

Dal punto di vista della comunicazione, il protocollo I²C è leggermente più complesso rispetto a standard simili (come l'SPI), in quanto il fatto di rinunciare ad alcune linee dedicate (come la linea di CS) richiede che determinate funzioni vengano spostate a livello del protocollo di comunicazione. Queste funzioni sono l'indirizzamento dei dispositivi (tramite un opportuno campo di 7-bit all'interno del primo byte dati trasmesso) destinatari dei comandi e la tipologia di operazione richiesta (scrittura/lettura, tramite il rimanente bit). L'indirizzamento permette la gestione simultanea di oltre 100 dispositivi sul bus, ma per limitazioni fisiche il numero è di molto inferiore, aggirandosi, nelle applicazioni più spinte, sulla decina di unità (il tipico scenario è di 1-2 dispositivi collegati). Il protocollo prevede anche l'invio di un bit di acknowledge (ACK) per indicare la corretta ricezione di un blocco dati (byte). In Fig. 6 è riportato un esempio di pacchetto dati I²C, mentre in Fig. 7 trovate un esempio di comunicazione, sempre su bus I²C. Dal punto di vista prestazioni, l'I²C ha

### TYPICAL I<sup>2</sup>C™ WRITE TRANSMISSION (7-BIT ADDRESS)



### TYPICAL I<sup>2</sup>C™ READ TRANSMISSION (7-BIT ADDRESS)

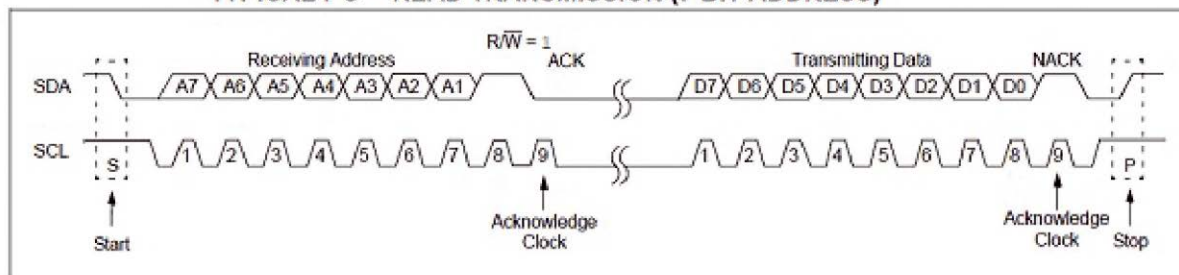


Fig. 7 - Esempio di comunicazione.

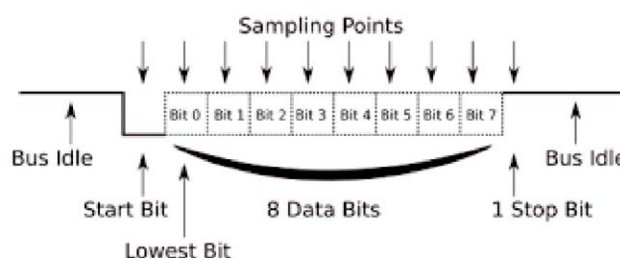
Fig. 8 - Esempio di pacchetto dati UART.

prestazioni inferiori rispetto all'SPI, con velocità di comunicazione fino a 100 kbit/s (400 kbit/s in fast mode), ma la più recente emanazione dello standard ha introdotto velocità superiori fino a 5Mbit/s, ma con pesanti limitazioni, come l'unidirezionalità del collegamento.

### LA PERIFERICA UART

Lo Universal Asynchronous Receiver Transmitter è uno dei più comuni standard di comunicazione digitale, largamente impiegato ai nostri giorni, nelle sue innumerevoli declinazioni. Si tratta di uno standard seriale asincrono di tipo full-duplex punto-punto. Essendo asincrono, la velocità di comunicazione (normalmente detta baud-rate) dev'essere nota a priori ai dispositivi che devono scambiarsi dati, altrimenti il collegamento non può avvenire. I segnali utilizzati sono due: TX ed RX, rispettivamente linea di trasmissione e di ricezione (come abbiamo detto, lo standard è full-duplex e quindi la comunicazione può avvenire in entrambi i sensi ed essere simultanea). Chiaramente il collegamento tra due unità che impiegano questo standard di comunicazione deve essere incrociato (la linea di TX del dispositivo

### UART with 8 Databits, 1 Stopbit and no Parity



A viene connessa alla linea RX del dispositivo B, e viceversa). Lo standard prevede alcuni simboli per il controllo della trasmissione dei dati e può prevedere anche controllo di flusso hardware e controllo di parità per l'individuazione degli errori di comunicazione. In Fig. 8 è riportato un esempio di pacchetto dati di un byte con un singolo stop bit e senza parità. Le tipiche velocità di comunicazione oscillano tra i 300 e i 115.200 baud, ma possono essere anche differenti; sono impostate agendo su un registro interno che genera la corretta base tempi (in genere denominato BRG, Baud Rate Generator).

### LA PERIFERICA UART DEI PIC32

I PIC32 presentano mediamente, al loro interno, diversi moduli UART, separati ed indipendenti tra di loro. Tali moduli, a

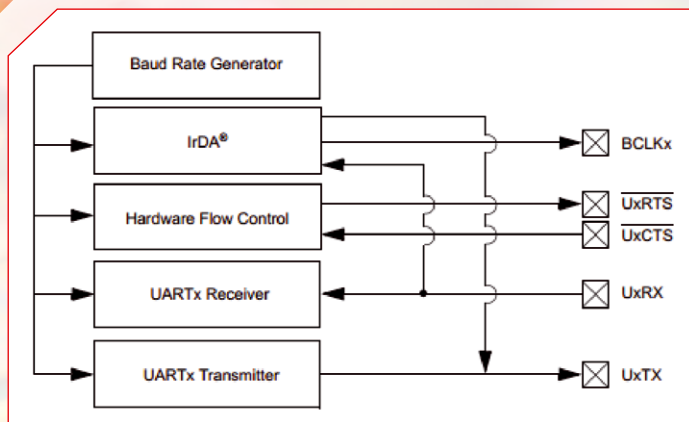


Fig. 9 - Schema a blocchi del modulo UART dei PIC32.

seconda dei modelli, possono avere dei pin dedicati, o utilizzare pin rimappabili (Peripheral Pin Select). In Fig. 9 è riportato uno schema a blocchi del modulo UART presente all'interno dei PIC32. Le caratteristiche peculiari del modulo sono:

- trasmissione dati full-duplex ad 8 o 9-bit;
- opzione controllo di parità sui dati trasmessi (solo 8 bit);
- terminazione con 1 o 2 stop bit;
- controllo di flusso Hardware integrato;
- encoder/decoder IrDA integrato;
- auto-baud mode;
- baud-rate generator integrato con prescaler a 16 bit;
- buffer FIFO di ricezione e trasmissione separati;
- interrupt di trasmissione e di ricezione separati;
- modalità loopback;
- supporto al protocollo LIN 1.2 (LIN capable UART).

Ogni modulo si configura e si gestisce tramite una coppia di registri di stato/configurazione:

- UxMODE;
- UxSTA.

dove "x" indica il generico UART all'interno del microcontrollore (gli UART sono identificati con un numero progressivo).

Le funzioni del registro UxMODE sono:

- abilitazione\disabilitazione del modulo;
- abilitazione\disabilitazione dell'encoder-decoder IrDA;
- controllo delle caratteristiche di Auto-baud e loopback;
- abilitazione\disabilitazione dei pin TX\RX;

- controllo di polarità del pin RX;
- selezione delle opzioni di parità, stop bit, data bit.

Le funzioni del registro UxSTA sono invece:

- impostazione delle modalità di Interrupt;
- abilitazione\disabilitazione della trasmissione dati;
- varie indicazioni di stato (stato dei buffer, controllo di parità, overflow, framing, eccetera).

A questi due registri (che controllano tutta la parte di configurazione) si aggiungono i buffer di ricezione/trasmmissione e il registro per il controllo del baud-rate. Questo set permette la gestione completa del modulo, anche se alcune funzioni relative agli interrupt (come l'abilitazione globale o i flag) fanno capo all'interrupt controller e sono gestite da altri registri.

### UTILIZZO DELLA PERIFERICA UART DEI PIC32

Passiamo adesso al progetto pratico relativo a questa puntata, nel quale effettueremo qualche esperimento con la porta UART dei PIC32: quello che ci prefiggiamo di fare è realizzare un programma che invii una stringa di caratteri attraverso essa. Per farlo, occorrerà aprire e configurare la porta ed utilizzare le macro di invio.

Per utilizzare la porta UART dei PIC32 è possibile procedere in diversi modi: ad esempio si potrebbe cercare nel data-sheet e scrivere tutte le funzioni necessarie per la gestione. Quest'approccio, sebbene molto lungo presenta il vantaggio di mantenere tutto il codice sotto controllo e di avere una visione molto chiara di come le periferiche del dispositivo funzionano. Purtroppo una trattazione del genere richiederebbe troppo

#### Listato 1

```
/* avoid the MLH warning message */
#define _SUPPRESS_PLIB_WARNING

/* -- includes -- */
#include <p32xxx.h>
#include <plib.h>
```

Listato 1 - Inclusioni per esempio pratico.

## Listato 2

```
#include <peripheral/adc10.h>
#include <peripheral/bmx.h>
#include <peripheral/cmp.h>
#include <peripheral/cvref.h>
#include <peripheral/dma.h>
#include <peripheral/i2c.h>
#include <peripheral/incap.h>
#include <peripheral/int.h>
#include <peripheral/nvm.h>
#include <peripheral/outcompare.h>
#include <peripheral/pcache.h>
#include <peripheral/pmp.h>
#include <peripheral/ports.h>
#include <peripheral/pps.h>
#include <peripheral/power.h>
#include <peripheral/reset.h>
#include <peripheral/rtcc.h>
#include <peripheral/spi.h>
#include <peripheral/system.h>
#include <peripheral/timer.h>
#include <peripheral/uart.h>
#include <peripheral/wdt.h>
#include <peripheral/eth.h>
#include <peripheral/CAN.h>
```

Listato 2 – Contenuto di plib.h.

spazio per essere presentata su una singola puntata del corso. Perciò seguiremo un approccio più semplice: il compilatore XC32 mette a disposizione una serie di funzioni di libreria per la gestione delle varie periferiche, le quali possono essere importate includendo il file plib.h. Tale supporto sta subendo, nell'ultimo periodo, una fase di migrazione verso un framework più evoluto, denominato MPLab Harmony, tuttavia le funzioni di libreria sono ancora utilizzabili e costituiscono un'ottima soluzione per le applicazioni più semplici. Creiamo quindi un nuovo progetto MPLab X e, nel main, cominciamo ad importare tutti i componenti che ci occorrono. Nel caso specifico, importeremo il solito file di definizione del microcontrollore e il file plib.h, come illustrato nel **Listato 1**. Si noti che prima degli `#include` è stata definita la macro `_SUPPRESS_PLIB_WARNING`, per evitare la stampa dei messaggi di warning relativi alla migrazione verso il nuovo framework. Il file plib.h contiene a sua volta tutta una serie di inclusioni di pacchetti, tra cui anche le librerie UART. Una panoramica è riportata nel **Listato 2**.

Come si vede sono diversi i componenti disponibili, dall'ADC alle varie porte seriali, passando per timer, watchdog, RTC, ecc. Una volta fatte le importazioni necessarie, passiamo alla definizione di alcune macro che ci serviranno nel resto del programma, riportate nel **Listato 3**.

Qui abbiamo definito la nostra solita co-

## Listato 3

```
/* -- defines -- */
#define DELAY_VALUE      100000
#define FOREVER          1
#define SYSTEM_FREQ      8000000
```

Listato 3 – Defines per progetto pratico.

## Listato 4

```
/* uart configuration variables */
unsigned int DesiredBaudRate = 9600;
unsigned int PbusClock = SYSTEM_FREQ/2;
unsigned int UartBrg = 0;
/* data to send */
const unsigned char DataToSend[] = {'H','e','l','l','o','!'};
/* delay counter */
unsigned int counter = 0;
```

Listato 4 – Definizione variabili.

## Listato 5

```
/* set pin direction */
TRISFbits.TRISF5 = 0;
TRISFbits.TRISF4 = 1;

/* calculate brg */
UartBrg = ((PbusClock/(4 * DesiredBaudRate)) - 1);
/* open UART2 */
OpenUART2(UART_EN|UART_BRGH_FOUR, UART_RX_ENABLE|UART_TX_ENABLE, UartBrg);
```

Listato 5 – Configurazione della porta UART.

stante `FOREVER` e un'altra costante che ci serve per la generazione di un piccolo ritardo software prima di inviare la stringa. Poi definiamo anche la frequenza di clock del sistema in hertz, parametro che utilizzeremo in seguito per il calcolo del baud rate. Importazioni e definizioni iniziali sono ultimate e possiamo ora passare al programma vero e proprio con la parte di definizione delle variabili utilizzate.

Il **Listato 4** riporta le variabili utilizzate in questo programma.

Come si vede, abbiamo un set di tre variabili (`DesiredBaudRate`, `PbusClock` e `UartBrg`) che verranno utilizzate nella funzione di inizializzazione della porta UART per il calcolo del baud-rate. Il valore desiderato si trova all'interno della variabile `DesiredBaudRate`, espresso in baud; invece `PbusClock` contiene il valore in hertz della frequenza di clock del bus delle periferiche, valore che nei nostri bit di configurazione è impostato pari a metà della frequenza di clock di sistema

## Listato 6

```
/* delay loop */
while (counter)
{
    /* decrement counter */
    counter--;
}

/* send data over UART ch */
putsUART2(DataToSend);

/* endless loop */
while(FOREVER)
{
}

/* execution should never reach this line */
return (EXIT_SUCCESS);
```

Listato 6 – Ritardo iniziale, invio della stringa dati e chiusura.

(ecco perché viene imposto pari a  $\text{SYSTEM\_FREQ}/2$ ). Infine UartBrg conterrà il risultato del calcolo.

Le rimanenti due variabili sono l'array di caratteri DataToSend, che contiene la stringa da inviare (il classico "Hello World") e una variabile contatore per il calcolo del delay. Si noti l'uso del qualificatore const per l'array di caratteri: questa operazione fa sì che il contenuto dell'array sia scritto in maniera permanente in memoria e non sia più modificabile, ma consente di trattare il dato come un normale array presente in memoria. È quindi un'operazione che alloca effettivamente delle risorse runtime sul processore, a differenza delle precedenti #define, che sono direttive per il preprocessore.

Passiamo adesso alla parte centrale del programma, ossia inizializzazione della porta UART ed invio della stringa dati.

Il PIC32MX795F512L dispone al suo interno di 6 moduli UART. Per i nostri test noi utilizzeremo il modulo UART2, i cui pin TX ed RX si trovano sul connettore di espansione 1 della demoboard di supporto al corso presentata nella puntata precedente, e precisamente ai pin 9 e 11. Le istruzioni di configurazione ed apertura della porta UART sono riportate nel Listato 5.

Per prima cosa si settano i registri direzionali della coppia di pin Tx/Rx in maniera opportuna (in modo da avere Tx come output ed Rx come input). Poi si calcola il baud-rate, applicando la seguente formula (valida per la modalità high-speed):

$$BRG = \frac{P_{busclock}}{4 \times \text{BaudRate}} - 1$$

Tale valore viene posto all'interno della variabile di servizio UartBrg, in modo da poter essere poi passato alla funzione che esegue effettivamente l'inizializzazione. La funzione utilizzata è la OpenUART2, che prende in ingresso tre parametri: config1, config2 e Brg. I primi due sono di fatto il contenuto dei due registri di configurazione del modulo (UxMODE ed UxSTA), mentre l'ultimo è appunto il baud rate calcolato in precedenza. La nostra porta UART a questo punto è inizializzata e pronta per essere utilizzata.

Il frammento di codice che invia i dati sulla porta seriale è riportato nel Listato 6.

Come si può vedere dal codice, dopo un ritardo iniziale, i dati vengono inviati semplicemente effettuando una chiamata alla funzione PutsUART2, che riceve come argomento un puntatore ad una stringa. Avendo usato un array per contenere i caratteri da inviare, possiamo passare semplicemente il nome dell'array, che in linguaggio C equivale al puntatore al primo byte dell'array stesso. Per testare il nostro programma possiamo collegare un qualsiasi convertitore USB/seriale alla demoboard e ad un Personal Computer e verificare l'invio dei dati verso il PC. Alternativamente, se non disponete di un convertitore, può essere testato il funzionamento del modulo UART in modalità loopback (modalità molto utile in fase di debug). Per farlo, occorre inserire una opportuna opzione nella funzione di configurazione, così come riportato di seguito:

```
/* open UART2 */
OpenUART2(UART_EN | UART_BRGH_
FOUR | UART_EN_LOOPBACK, UART_
```

## Listato 7

```
/* send data over UART ch */
putsUART2(DataToSend);
/* check if data are available */
if (DataRdyUART2())
{
    getsUART2(6, RxBuffer, 200);
}
```

Listato 7 – Invio e ricezione in loopback.

```

45      /* delay loop */
46      while (counter)
47      {
48          /* decrement counter */
49          counter--;
50      }
51      UART
52      /* send data over UART ch */
53      putsUART2 (DataToSend);
54      /* check if data are available */
55      if (DataRdyUART2 ())
56      {
57          getsUART2 (6, RxBuffer, 200);
58      }
59  }

```

Fig. 10 - Break Point sul ciclo di ricezione.

RX\_ENABLE | UART\_TX\_ENABLE,  
UartBrg);

A questo punto i pin TX ed RX del modulo sono collegati tra di loro e quindi ciò che viene inviato sul pin di trasmissione è ricevuto direttamente dal pin di ricezione. Per eseguire un test possiamo modificare il codice come indicato nel **Listato 7** (avendo prima avuto cura di definire ed inizializzare l'array RxBuffer).

Lo snippet presentato nel **Listato 7** invia la stringa e poi verifica che ci siano dati validi sul buffer di ricezione, ed in caso positivo effettua la lettura, trasferendone il contenuto su RxBuffer.

Se piazziamo un break point alla fine del ciclo di lettura (Fig. 10) ed analizziamo la fine-

stra di watch nella quale inseriamo RxBuffer, vedremo che essa contiene la stringa dati inviata tramite la funzione PutsUART2 (Fig. 11).

## CONCLUSIONI

In questa puntata abbiamo conosciuto e descritto le periferiche di comunicazione di base, ed abbiamo preso confidenza con una delle più semplici, ossia la porta UART. Dall'inizio del corso abbiamo preso in esame diverse periferiche, sia digitali che analogiche, ma non abbiamo ancora una infrastruttura software che ci permetta di organizzarle efficacemente all'interno di un programma. Nella prossima puntata valuteremo insieme questo aspetto, cominciando a parlare di embedded multitasking. ■

Watches	Output	Search Results	Breakpoints	Configuration ...
Name	Char			
RxBuffer	"Hello!"			
RxBuffer[0]	'H'; 0x48			
RxBuffer[1]	'e'; 0x65			
RxBuffer[2]	'l'; 0x6c			
RxBuffer[3]	'l'; 0x6c			
RxBuffer[4]	'o'; 0x6f			
RxBuffer[5]	'!'; 0x21			
<Enter new watch>				

Fig. 11 - Contenuto di RxBuffer dopo la lettura.

# CORSO

# MPLAB X



di **FRANCESCO FICILI**  
e **VINCENZO GERMANO**

**Continuiamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip Technology, che soppianta il vecchio MPLab IDE. In questa puntata ci occupiamo di come realizzare applicazioni embedded multitasking con i PIC32. Quinta puntata.**

**N**elle precedenti puntate abbiamo introdotto i microcontrollori PIC32 prodotti da Microchip Technology (e il relativo ambiente di sviluppo MPLab X) analizzando alcune delle più importanti periferiche di questi dispositivi, come i timer, la circuiteria di interrupt, le periferiche analogiche e le interfacce di comunicazione disponibili a bordo.

In questa puntata facciamo un passo in avanti e trattiamo un argomento decisamente più complesso di quelli affrontati in passato, focalizzando la nostra attenzione sull'embedded multitasking e presentando una soluzione architetturale che soddisfa questo requisito e che utilizzeremo per il resto del corso.

## PROGRAMMAZIONE EMBEDDED

In una tipica applicazione embedded, l'unità centrale di elaborazione (che nella stragrande maggioranza dei casi è un microcontrollore) non dispone di ingenti risorse interne e impone la necessità di eseguire compiti eterogenei fornendo livelli differenti di privilegio ai vari task in esecuzione. A complicare il tutto, spesso e volentieri i sistemi embedded sono anche classificati come real-time e si pone quindi il problema di avere tempi certi di esecuzione (determinismo). In questo tipo di contesto, la programmazione multitasking ha un'importanza rilevante, in quanto offre una serie di soluzioni per distribuire le (poche) risorse di

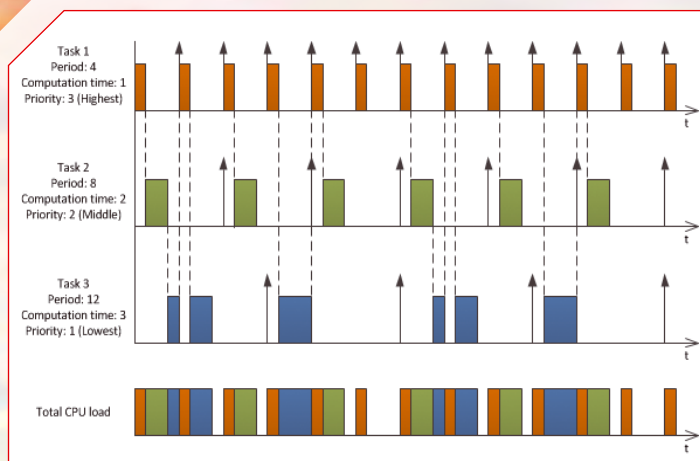


Fig. 1 - Esempio di scheduler.

sistema disponibili in maniera coerente tra i vari utilizzatori, mantenendo nel contempo prestazioni real-time.

Facciamo un esempio per chiarire la problematica: immaginiamo di avere un sistema embedded dotato di un pulsante, due LED ed una porta di comunicazione seriale. Il sistema deve fornire a un altro nodo sulla linea seriale l'informazione sulla pressione del pulsante e deve essere in grado di decodificare tre tipi differenti di pressione: pressione singola, pressione doppia e pressione lunga. Inoltre il sistema deve gestire un messaggio in ricezione che lo abilita o disabilita (quando viene disabilitato, il sistema è inerte e va in uno stato di basso consumo energetico). Quando il sistema è attivo uno dei due LED lampeggia ad intervalli regolari, mentre il secondo si accende per una durata prestabilita (ad esempio un secondo ad ogni invio di dati sulla linea seriale). Se il sistema è in modalità a basso consumo, entrambi i LED rimangono spenti. Un'applicazione di questo tipo, se approssciata con una tecnica di programmazione puramente sequenziale come quella vista sino ad ora nel corso, potrebbe risultare anche piuttosto complessa. La decodifica dello stato dei vari pulsanti richiede una certa logica per essere implementata, la gestione dei LED richiede temporizzazione ed è vincolata allo stato del sistema, e la linea seriale richiede di essere costantemente monitorata in ricezione ma anche di essere gestita in trasmissione quando un determinato evento (pressione del pulsante) si verifica. Seppure l'applicazione non sia complessa, far coesistere tutti questi task in una struttura unica (come visto in precedenza) non è cosa semplice e anche se si riuscisse a farlo, l'applicazione software sviluppata

risulterebbe -con ogni probabilità- non modulare, scarsamente scalabile e di conseguenza poco manutenibile. Insomma, ne risulterebbe un progetto software mal riuscito.

Per poter affrontare correttamente questo tipo di problematiche, si ha la necessità di un'infrastruttura software che permetta ad ogni task di avere (in forma maggiore o minore, a seconda di quanto complessa sia l'infrastruttura stessa) l'impressione che il processore e le risorse di sistema siano ad esso completamente dedicate e di avere una minima influenza sul proprio funzionamento da parte dell'attività degli altri task. In termini generali, un'infrastruttura software che risolve una problematica di questo tipo realizza il multitasking, ossia permette a più programmi di essere eseguiti contemporaneamente su di una singola CPU. In realtà l'esecuzione (se il sistema che stiamo utilizzando è un sistema single core) non è propriamente "simultanea", ma ad ogni programma viene ceduta una frazione di tempo di CPU per ciclo e il ciclo viene ripetuto nel tempo con una certa periodicità. Chiaramente, parlando di tempi di esecuzione estremamente ridotti (in base al tipo di applicazione, essi possono variare da alcuni microsecondi alle decine di millisecondi), in apparenza l'esecuzione è simultanea.

## SCHEDULER, SISTEMA OPERATIVO, RTOS

Come detto nel paragrafo precedente, per poter realizzare il multitasking abbiamo necessariamente bisogno di un'infrastruttura software che ci permetta di far credere ad ogni task che la CPU sia completamente dedicata ad esso. Ma cos'è e come è fatta questa infrastruttura? L'infrastruttura base che permette di suddividere il tempo di CPU tra vari processi in esecuzione è chiamata in genere *scheduler* (*schedulatore*, in italiano). Uno scheduler è di fatto un programma che, sfruttando un ben determinato algoritmo, regola l'accesso alle risorse di sistema da parte dei vari programmi in esecuzione. Il particolare algoritmo utilizzato da questo componente software viene in genere denominato "politica di scheduling" o "algoritmo di scheduling". Un esempio di scheduler è riportato in Fig. 1. Come si può vedere nell'esempio, lo scheduler organizza, con un certo algoritmo di

scheduling, l'esecuzione di tre task all'interno della CPU. Il risultato è visibile nella traccia in basso nella solita **Fig. 1**.

Gli scheduler possono funzionare da soli, ma spesso sono integrati da una serie di altri componenti software che forniscono funzionalità aggiuntive. Esempi di componenti aggiuntive sono i meccanismi di inter-task communication, il gestore della memoria, l'interfaccia utente, lo spooler, e altro ancora. L'insieme dello scheduler e delle altre componenti software prende il nome di Sistema Operativo e in genere lo scheduler e altri servizi di base prendono il nome di kernel (nucleo) del sistema operativo stesso.

Un particolare tipo di sistema operativo è il cosiddetto RTOS (Real Time Operating System), che è il sistema operativo più diffuso in ambito embedded. Questo particolare tipo di sistema operativo deve anche garantire che l'esecuzione sia real-time, ossia deve fare in modo che una certa operazione venga portata a termine con tempistiche ben precise e definite, senza che si verifichino fenomeni di latenza (determinismo).

Questo tipo di caratteristica è molto gradita nello sviluppo di sistemi embedded e safety critical, dove la presenza di latenze temporali non determinabili a priori può dare luogo a malfunzionamenti molto gravi dell'intero sistema (per i sistemi safety critical si parla di ha-

zard, che viene classificato in base alle possibili conseguenze, che, nel caso di sistemi operati da personale umano, possono anche mettere a rischio la vita). In **Fig. 2** è riportata una struttura tipica di un RTOS. Si noti la mancanza di elementi come l'interfaccia grafica e il gestore del file system, che essendo dipendenti da periferiche di I/O tipicamente "lente", possono compromettere il determinismo.

## COOPERATIVE E PREEMPTIVE MULTITASKING A CONFRONTO

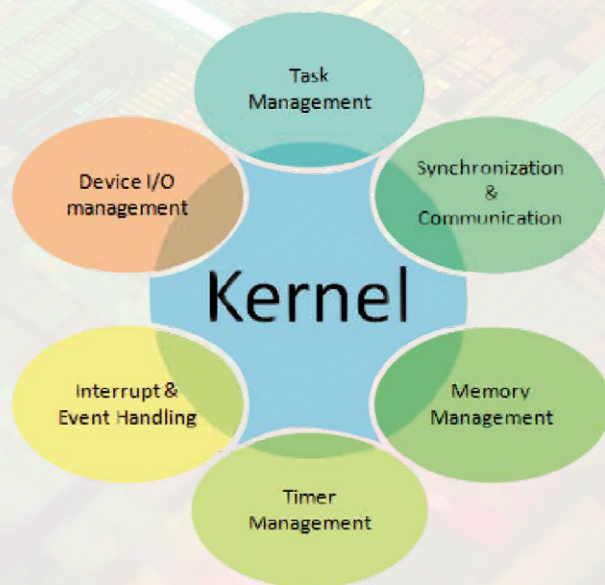
Ora che abbiamo parlato delle infrastrutture e definito anche un po' di terminologia, entriamo più nel dettaglio e classifichiamo le tipologie di sistemi multitasking con le quali possiamo trovarci a lavorare.

Esistono fondamentalmente due tipologie di multitasking:

- Cooperative multitasking;
- Preemptive multitasking.

Il cooperative multitasking è il primo tipo di multitasking che si è largamente diffuso. Ad esempio le prime versioni di Windows (3.0 e 3.1) erano sistemi operativi con multitasking cooperativo. In questo caso i programmi cedono volontariamente il controllo al programma di gestione (che può essere un sistema operativo o anche semplicemente uno scheduler), una volta terminata l'esecuzione corrente. Il vantaggio di un approccio di questo tipo è che non richiede particolari strutture hardware a supporto e può essere implementato facilmente in diverse architetture. Il principale svantaggio invece deriva dal fatto che un singolo programma che, per qualche ragione, si rifiuta di cedere il controllo, può bloccare l'intero flusso di esecuzione. Un esempio di multitasking cooperativo è riportato in **Fig. 3**. Come si può vedere, tre task vengono alternati nell'esecuzione dallo scheduler, ma sono i task stessi a restituire volontariamente il controllo, senza nessuna possibilità da parte dello scheduler di guadagnarli di sua iniziativa.

Di contro, nel caso di preemptive multitasking lo scheduler ha la possibilità di sottrarre le risorse ad un programma in esecuzione se lo ritiene necessario (ad esempio se un programma si è bloccato o se l'esecuzione sta richiedendo un tempo eccessivo e ci sono task in attesa di essere serviti). Chiaramente i task conservano il



**Fig. 2 - Struttura di un RTOS.**

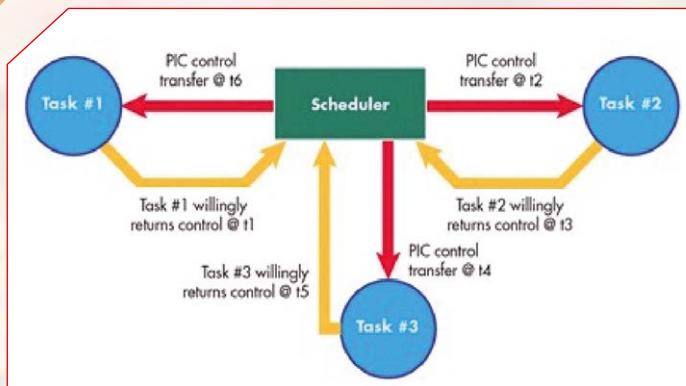


Fig. 3 - Multitasking cooperativo.

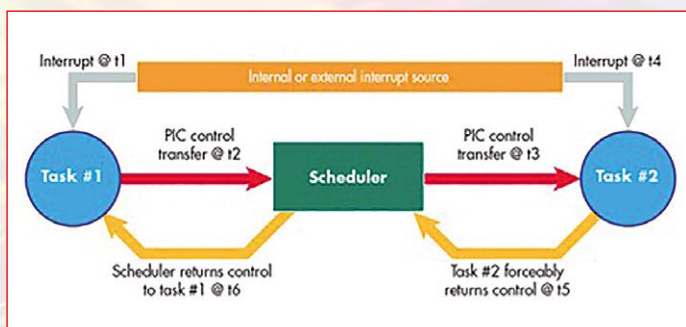


Fig. 4 - Preemptive multitasking.

diritto di cedere volontariamente le risorse di sistema nel caso in cui l'esecuzione corrente sia terminata. Un esempio è riportato in Fig. 4. In questo caso lo scheduler forza il task numero 2 a restituire il controllo per poterlo passare la task numero uno che richiede l'utilizzo delle risorse di sistema.

Chiaramente una struttura di questo tipo presenta l'enorme vantaggio di evitare che un singolo programma blocchi o rallenti l'esecuzione, in quanto lo scheduler può sempre intervenire e sottrarre le risorse al task incriminato e far eseguire i task in attesa. Lo svantaggio deriva dal fatto che una soluzione di questo tipo necessita di particolari strutture hardware per poter essere implementata e per di più queste ultime possono variare da architettura ad architettura, rendendo necessaria l'esecuzione di un porting (principalmente dello strato software di basso livello, spesso indicato come HAL – Hardware Abstraction Layer) nel caso in cui un sistema operativo implementato in questa maniera debba essere adattato ad una nuova architettura.

### UN ESEMPIO DI SCHEDULER

In questa puntata introduciamo uno scheduler che utilizzeremo per lo sviluppo delle nostre applicazioni nel resto del corso. La scelta di

presentare uno scheduler, anziché un RTOS, che chiaramente è un oggetto più performante e completo, deriva da considerazioni relative alla complessità (un RTOS è decisamente più difficile da utilizzare), al tempo (una trattazione completa sulla teoria e sulla pratica degli RTOS richiederebbe un corso dedicato) e alla didattica (si parte dalle strutture più semplici per andare poi verso quelle più complesse). Lo scheduler che presentiamo è un semplice scheduler FCFS (First Call First Serve, ossia il primo task che richiede attenzione viene servito e poi di seguito gli altri), non preemptive (quindi si implementa il multitasking cooperativo) e con gestione della periodicità. La gestione della periodicità è un sistema per gestire la priorità di esecuzione, ossia i task che richiedono mediamente più risorse perché eseguono elaborazioni più complesse, hanno un periodo di chiamata più corto nel tempo (vengono eseguiti più di frequente) rispetto ai task che necessitano di meno risorse. Lo schema viene deciso staticamente durante l'implementazione, tramite un'opportuna struttura contenente la periodicità di ogni task e non può essere più modificato a runtime.

Vediamo quindi come è stato realizzato lo scheduler in questione, andando ad analizzarne il progetto MPLab X. In Fig. 5 è riportato il project manager del progetto dello scheduler

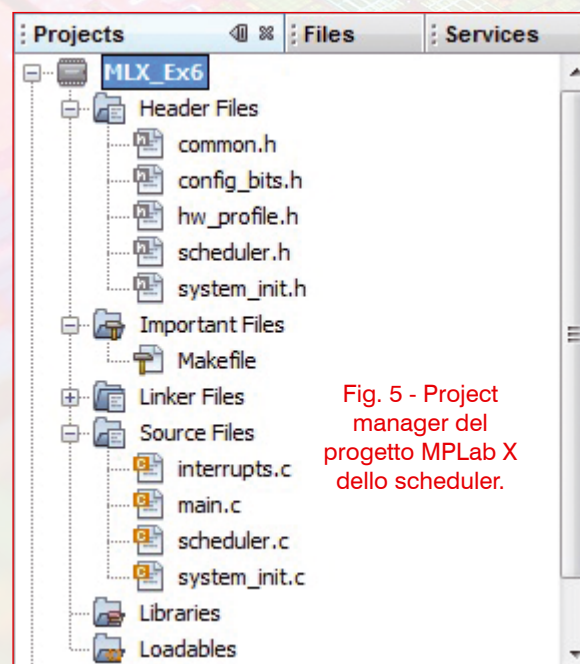


Fig. 5 - Project manager del progetto MPLab X dello scheduler.

Tabella 1 - Descrizione dei file di progetto.

File	Descrizione
main.c	Entry point del flusso di programmazione. Il main contiene due chiamate a funzione: una alla funzione SystemInit, che inizializza l'hardware di base e una alla funzione MainScheduler, che prende il controllo dell'esecuzione e lancia lo scheduler.
scheduler.c, scheduler.h	Questa coppia di files contiene l'algoritmo di scheduling, i servizi di inter task communications e le strutture dati dello scheduler.
system_init.c, system_init.h	Questa coppia di file contiene le inizializzazioni all'hardware di base utilizzato dallo scheduler.
interrupts.c	Questo file racchiude tutte le ISR usate dallo scheduler.
hw_profile.h	Questo file contiene le ridefinizioni dell'hardware di basso livello.
config_bits.h	Questo file contiene i settaggi dei bit di configurazione.
common.h	Questo file contiene alcune direttive del preprocessore utilizzate per configurare lo scheduler e alcune macro di uso generale.

(MLX\_Ex6), scaricabile anche dal nostro sito [www.elettronica.in](http://www.elettronica.in). A differenza di quanto abbiamo fatto fino ad ora, possiamo subito notare che il progetto è suddiviso in più file. Infatti, essendo questa infrastruttura pensata per progetti di maggiore complessità, il progetto dello scheduler stesso è stato pensato modulare, in maniera da aumentarne la scalabilità.

Analizziamo i file di progetto servendoci della **Tabella 1**.

Come si può vedere, il progetto dello scheduler è costituito da un totale di 9 file. Idealmente ogni task che vogliamo aggiungere alla nostra applicazione dovrebbe contenere una coppia di file: un file .c contenente le implementazioni ed un file .h contenente tutte le interfacce. Sebbene questo tipo di struttura possa sembrare un pò pesante all'inizio, essa consente una gestione efficiente e organizzata del materiale di progetto, ne migliora la scalabilità e la manutenibilità e gli conferisce un aspetto decisamente più professionale.

Passiamo adesso ad analizzare il codice per capire il funzionamento di questo oggetto; come abbiamo detto in precedenza, ciò che vogliamo realizzare è uno scheduler FCFS di tipo non preemptive, con gestione della periodicità dei task. Per realizzare questo tipo di struttura ci vogliamo basare su un tick di sistema ad 1 ms, attorno al quale costruiremo tutte le periodicità dei task. Per generare questo tick ci serviamo di un interrupt su un timer opportunamente configurato. Il timer scelto è il timer 2, che configuriamo ed inizializziamo per generare un interrupt ogni millisecondo. Il **Listato 1** rappresenta la funzione di inizializzazione del timer, che viene lanciata nel main.

La relativa ISR è invece riportata nel **Listato 2** e si trova all'interno del file interrupt.c.

Come si può vedere, quando il numero di conteggi supera il periodo di scheduling viene resettato il contatore e settato il flag MainSystemTimebaseFlag al valore CallTaskPhase. Il MainSystemTimebaseFlag è una variabile globale che l'algoritmo di scheduling controlla costantemente per capire se un nuovo ciclo di scheduling deve avere inizio.

A questo punto abbiamo il nostro riferimento temporale di base che ci occorre per la temporizzazione; ciò che ci manca è l'algoritmo di scheduling ed una struttura dati per la gestione

### Listato 1

```

/*****
 *
 * Function:      InitSchedTimer
 * Input:         None
 * Output:        None
 * Author:        F.Ficili
 * Description:   Initialize timer 2 as scheduler timebase:
 *               Prescaler: 1/8; Actual Interrupt Time: 1 ms
 * Date:          14/02/15
 *****/
void InitSchedTimer (void)
{
    /* configure the PIC32 core for the best performance */
    SYSTEMConfigPerformance (SYSTEM_FREQUENCY_VALUE_HZ);

    /* set T1CON register */
    T2CONbits.ON = K_ON;
    T2CONbits.TCKPS = TMR_PRESCALER_8;
    T2CONbits.T32 = TMR_16_BIT_MODE;
    T2CONbits.TCS = TMR_CS_PBCCLK;
    PR2 = INT_MATCH_VALUE;

    /* set interrupt register */
    IPC2bits.T2IP = T2_INT_PRIO_1;
    IPC2bits.T2IS = T2_INT_SUB_PRIO_0;
    IEC0bits.T2IE = T2_INT_ENABLE;

    /* enable multi-vector interrupts */
    INTEnableSystemMultiVectoredInt();
}

```

Listato 1 - Inizializzazione dello schedule timer.

**Listato 2**

```

/*****
* Function:          void __ISR(_TIMER_2_VECTOR,
ipl1) Timer2InterruptHandler (void)
* Input:            None
* Output:           None
* Author:           F.Ficili
* Description:      Timer 2 ISR.
* Date:             14/02/15
*****/
void __ISR(_TIMER_2_VECTOR, ipl1) Timer2InterruptHandler
(void)
{
    /* local counter */
    static UINT16 InterruptCounter = 0;

    /* clear interrupt flag */
    IFS0bits.T2IF = 0;
    /* Increment local counter */
    InterruptCounter++;
    /* If the scheduler period is elapsed */
    if (InterruptCounter >= SCHEDULER_PERIOD_MS)
    {
        /* Reset counter */
        InterruptCounter = 0;
        /* Main scheduler timebase flag */
        MainSystemTimebaseFlag = CallTaskPhase;
    }
}

```

**Listato 2 - ISR relativa al timer dello scheduler.**

della periodicità. Analizziamo prima di tutto la struttura dati: quella che abbiamo scelto di utilizzare è una struttura basata su tre array; il primo sarà un array di puntatori a funzione, contenente le chiamate ai task che desideriamo eseguire, il secondo sarà un array di interi a 16-bit contenente i contatori dei task ed il terzo un array di interi a 16-bit contenente i valori di periodicità di ogni singolo task. In questo

**Listato 3**

```

/* Maximum number of task */
#define ACTIVE_TASK_NUMBER                ((UINT16) (0))

/* Array of task to call */
void (*TaskArray[ACTIVE_TASK_NUMBER]) (void) =
{
};

/* Array of task period timeouts */
UINT16 TaskPeriodTimeoutMs[ACTIVE_TASK_NUMBER] =
{
};

/* Array of task periods */
static UINT16 TaskPeriodCounterMs[ACTIVE_TASK_NUMBER] =
{
};

```

**Listato 3 - Struttura dati per l'implementazione dell'algoritmo di scheduling.**

modo il nostro algoritmo di scheduling avrà a disposizione una tabella che potrà interrogare ad ogni ciclo per determinare se un task debba essere eseguito oppure no. In caso positivo, potrà lanciare in esecuzione il task sfruttando l'array di puntatori a funzione, mentre in caso negativo, semplicemente andrà oltre.

Il **Listato 3** riporta la struttura dati citata in precedenza (chiaramente non inizializzata). A questo punto possiamo passare alla descrizione dell'algoritmo di scheduling vero e proprio, il cui codice C è riportato nel **Listato 4**. Come si può vedere dal codice, la funzione esegue le seguenti operazioni:

- compie una scansione degli array dei task, partendo dall'indice 0, fino ad arrivare al numero di task attivi;
- per ogni elemento viene incrementato di una unità il contatore del task (che esprime il tempo in ms);
- se il contatore ha superato il valore della periodicità del task corrente (in ms) allora l'algoritmo resetta il contatore del task attivo e lancia il task, altrimenti passa all'analisi del task successivo.

Chiaramente la funzione deve essere chiamata ogni millisecondo e questo viene effettuato dalla funzione MainScheduler, che è la funzione chiamata direttamente dal main. Il codice della funzione MainScheduler è riportato nel **Listato 5**; come possiamo vedere, inizializzazione a parte, la funzione controlla continuamente (all'interno di un ciclo infinito) se il flag MainSystemTimebaseFlag è uguale a CallTaskPhase e, in caso positivo, lancia ScheduleTasks e resetta il flag per permettere l'esecuzione di un nuovo ciclo di scheduling.

Questa semplice struttura ci permette di lanciare in sequenza, e con una certa periodicità, un numero arbitrario di task che lavorano in multitasking cooperativo. La condizione per cui tutto funzioni è chiaramente che ogni singolo task lavori in multitasking cooperativo (ossia deve essere il task stesso a cedere il controllo allo scheduler per permettere che un nuovo task venga eseguito) e che la somma del tempo di esecuzione per ciclo di tutti i task non sia superiore al singolo ciclo di scheduling. Nel caso in cui questa seconda condizione non si verifichi, si può rimediare allungando il tempo di scheduling (è possibile farlo agendo sulla

macro SCHEDULER\_PERIOD\_MS).

### ESEMPIO PRATICO: REALIZZAZIONE DI UN SEMPLICE TASK

Vediamo adesso come realizzare un task sfruttando questa architettura, ripartendo da un esempio che abbiamo visto all'inizio del corso, ossia il lampeggio di un LED (che chiameremo, appunto, LedTask). In quell'occasione l'implementazione software per generare il lampeggio era stata quella visibile nel **Listato 6**.

Il codice presentato nello snippet di tale listato non potrebbe funzionare come implementazione del LedTask che dobbiamo realizzare adesso, perché questo tipo di realizzazione non implementa il multitasking cooperativo. Infatti se scrivessimo il LedTask con questa implementazione, una volta chiamato, questo task non restituirebbe più il controllo allo scheduler, che quindi non potrebbe più passarlo ad un eventuale secondo task, e l'esecuzione rimarrebbe bloccata all'interno del while(FOREVER) iniziale.

Un modello implementativo che si concilia molto con il multitasking cooperativo è quello delle macchine a stati finiti: e quindi questa è l'implementazione che cercheremo di realizzare. Di fatto un task che esegue il lampeggio di un LED è un task che per la maggior parte aspetta che passi del tempo e poi, al momento convenuto, inverte lo stato di un pin. In una logica a macchina a stati, questo si traduce in due stati: uno stato di wait e uno stato di toggle. La macchina a stati che implementa quanto descritto in precedenza è riportata in **Fig. 6**.

#### Listato 6

```
/* endless lopp */
while(FOREVER)
{
    /* delay loop */
    while (counter)
    {
        /* decrement counter */
        counter--;
    }
    /* set delay value to DELAY_VALUE */
    counter = DELAY_VALUE;
    /* toggle RD0 */
    PORTDbits.RD0 = ~PORTDbits.RD0;
}
```

Listato 6 - Implementazione del lampeggio del LED presentata nella prima puntata.

#### Listato 4

```
/* ***** */
* Function:          void ScheduleTasks (void)
* Input:             None
* Output:            None
* Author:            F.Ficili
* Description:       Function used to call the task sequence
                    with priority
* Date:              14/02/15
/* ***** */
void ScheduleTasks (void)
{
    /* Index of active task */
    UINT8 ActiveTaskIndex = 0;

    /* Call task if timeout is expired */
    for (ActiveTaskIndex = 0; ActiveTaskIndex < ACTIVE_
        TASK_NUMBER; ActiveTaskIndex++)
    {
        /* Increment task counters */
        TaskPeriodCounterMs[ActiveTaskIndex]++;

        /* Check the task timeout */
        if (TaskPeriodCounterMs[ActiveTaskIndex] >= TaskPe_
            riodTimeoutMs[ActiveTaskIndex])
        {
            /* Reset task counter */
            TaskPeriodCounterMs[ActiveTaskIndex] = 0;
            /* Call Task */
            TaskArray[ActiveTaskIndex]();
        }
    }
}
```

Listato 4 - Listato della funzione ScheduleTasks.

Proviamo adesso a scriverla in linguaggio C (il progetto di questo esempio pratico è stato nominato MLX\_Ex7 ed è scaricabile dal sito della

#### Listato 5

```
/* ***** */
* Function:          void MainScheduler (void)
* Input:             None
* Output:            None
* Author:            F.Ficili
* Description:       Main system scheduler
* Date:              14/02/15
/* ***** */
void MainScheduler (void)
{
    /* Initialize all Tasks */
    InitializeTasks();

    /* Switch to running state */
    SystemState = RunningState;

    /* Infinite loop */
    while(FOREVER)
    {
        /* If the scheduler timer has expired */
        if (MainSystemTimebaseFlag == CallTaskPhase)
        {
            /* Start scheduling cycle */
            ScheduleTasks();
            /* Reset flag */
            MainSystemTimebaseFlag = WaitTriggerPhase;
        }
    }
}
```

Listato 5 - Listato della funzione MainScheduler.

## Listato 7

```

/*****
* Function:      LedBlinkStateMachine
* Input:         void
* Output:        LedOutType
* Author:        F.Ficili
* Description:   Blinking LED effect state machine
* Date:          01/03/15
*****/
LedOutType LedBlinkStateMachine (void)
{
    /* State machine static variables */
    static LedBlinkSmType LedBlinkSmState = WaitState;
    static UINT16 LedBlinkSmCounter = 0;
    static LedOutType LedBlinkSmOut = OffState;

    switch (LedBlinkSmState)
    {
        case WaitState:
            /* Increment counter */
            LedBlinkSmCounter++;
            /* Check time */
            if (LedBlinkSmCounter >= LED_BLINK_PERIOD_MS)
            {
                /* Reset counter */
                LedBlinkSmCounter = 0;
                /* Jump to ToggleState */
                LedBlinkSmState = ToggleState;
            }
            break;

        case ToggleState:
            /* Toggle LED output state */
            LedBlinkSmOut = ~LedBlinkSmOut;
            /* Jump to WaitState */
            LedBlinkSmState = WaitState;
            break;

        default:
            break;
    }

    /* Return out state */
    return (LedBlinkSmOut);
}

```

Listato 7 - Implementazione C della macchina a stati per il lampeggio del LED.

rivista). Un modo molto efficace per scrivere le macchine a stati in C è usare la struttura di selezione switch..case e definire le variabili di

stato come variabili static con scope interno alla funzione stessa. L'implementazione è riportata nel **Listato 7**.

Come si può vedere, sono state definite tre variabili statiche:

- **LedBlinkSmState**: rappresenta lo stato corrente della macchina;
- **LedBlinkSmCounter**: contatore di servizio utilizzato per calcolare il ritardo;
- **LedBlinkSmOut**: stato corrente dell'uscita.

La funzione entra nello stato di wait (in quanto `LedBlinkSmType LedBlinkSmState = WaitState`;) e incrementa un contatore che serve a calcolare il tempo trascorso. Il contatore si basa sulla periodicità del task, la quale, come vedremo in seguito, è stata fissata a 10 ms. Se il numero di conteggi è maggiore `LED_BLINK_PERIOD_MS` (vedremo dopo anche questo), il contatore viene resettato, si aggiorna la variabile di stato e si esce, altrimenti si esce direttamente. Nello stato di toggle si esegue solo l'operazione di toggle vero e proprio, si aggiorna la variabile di stato e si esce. Si noti come ogni stato esegue le operazioni che deve eseguire ed esce, occupando per meno tempo possibile il processore. Ad ogni esecuzione, la funzione ritorna lo stato corrente dell'uscita.

Ora abbiamo la nostra macchina a stati; quello che ci manca è costruire un task ed inserirlo nella tabella di esecuzione dello scheduler. Il listato del LED task è riportato nel **Listato 8**. Come possiamo constatare, il task stesso è una macchina a stati, che ha uno stato di inizializzazione ed uno di running; in quest'ultimo stato viene semplicemente associato il valore dell'uscita di `LedBlinkStateMachine` alla macro

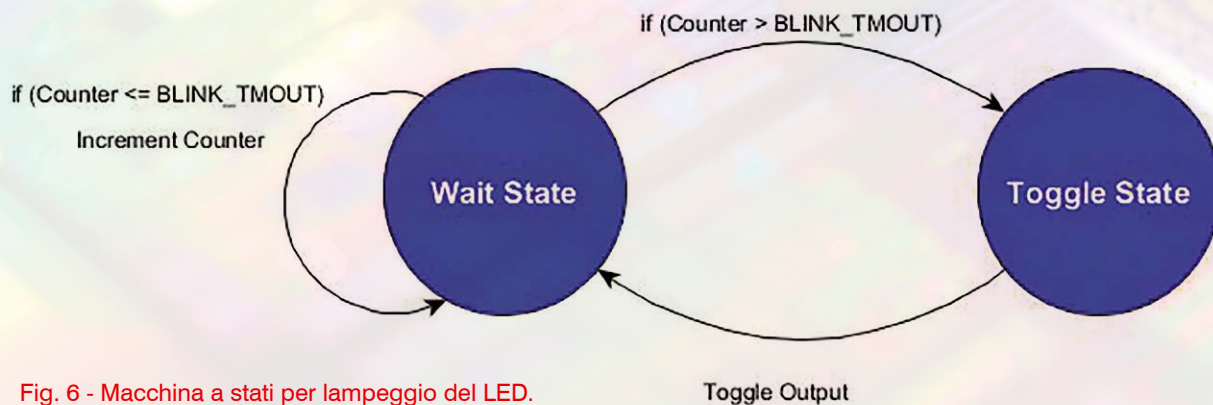


Fig. 6 - Macchina a stati per lampeggio del LED.

LED\_1, che non è altro che la ridefinizione del pin RD1.

Ora non ci rimane che inserire il task all'interno della tabella di esecuzione, per far sì che lo scheduler lo lanci. Non ci sono altri task, quindi questo sarà l'unico processo in esecuzione, ma in linea teorica potrebbero essercene un numero arbitrario.

Prima di analizzare la struttura, bisogna notare che nell'header file sono state definite queste due macro (**Listato 9**):

La prima è la periodicità del task (che come detto è fissata a 10 ms), mentre la seconda è il periodo del lampeggio del LED, che è fissato a 500 ms ed è diviso a compile time per LED\_TASK\_PERIOD\_MS, in modo da ottenere il valore corretto di conteggi potendo mantenere coerenza con il significato fisico della macro stessa (ossia un tempo in ms, come è evidenziato anche dal nome). Anche la periodicità è riferita al periodo di scheduling, che in questo caso è 1ms. L'inserimento del task nella tabella di esecuzione si effettua come illustrato nel **Listato 10**. Se proviamo a mandare in esecuzione il programma, vedremo lo stesso effetto del lampeggio del LED visto durante gli esperimenti condotti nel corso della prima puntata, ma quello che sta succedendo all'interno del programma è ben diverso: ad ogni ciclo di scheduling, lo scheduler scandisce la tabella e ogni 10 ms manda in esecuzione LedTask, che a sua volta esegue la macchina a stati per il lampeggio. Questa politica lascia libero il processore, che nella maggior parte del tempo rimane in attesa del flag che avvia un nuovo ciclo. Tutto questo tempo può essere riempito con l'esecuzione di ulteriori task, come vedremo negli esempi pratici che presenteremo nelle prossime puntate.

## CONCLUSIONI

In questa puntata abbiamo introdotto un argomento che ci servirà come base per le puntate future, in quanto nei prossimi progetti utilizzeremo sempre questa infrastruttura e implementeremo i nostri programmi utilizzando il multitasking cooperativo. Nell'esempio descritto in questa puntata l'approccio potrebbe sembrare molto più complesso rispetto a una semplice programmazione sequenziale, ma i benefici di questa filosofia si potranno apprezzare con lo sviluppo di applicazioni più complesse. ■

## Listato 8

```

/*****
* Function:          LedTask
* Input:             None
* Output:            None
* Author:           F.Ficili
* Description:       Manage LED task
* Date:             28/02/15
*****/
void LedTask (void)
{
    switch (SystemState)
    {
        /* System Initialization Phase */
        case InitializationState:
            /* Initialize LEDs pins */
            LED_1_TRIS = LINE_DIRECTION_OUTPUT;
            LED_1 = LINE_STATE_LOW;
            break;

        /* System Normal operation Phase */
        case RunningState:
            /* Run LED_1 state machine */
            LED_1 = LedBlinkStateMachine();
            break;

        /* Default */
        default:
            break;
    }
}

```

Listato 8 - LedTask.

## Listato 9

```

/* Task period in ms */
#define LED_TASK_PERIOD_MS      ((UINT16) (10)/SCHEDULER_PERIOD_MS)
/* Blink period in ms */
#define LED_BLINK_PERIOD_MS    ((UINT16) (500)/LED_TASK_PERIOD_MS)

```

Listato 9 - Macro di LedTask.

## Listato 10

```

/* Maximum number of task */
#define ACTIVE_TASK_NUMBER      ((UINT16) (1))

/* Array of task to call */
void (*TaskArray[ACTIVE_TASK_NUMBER]) (void) =
{
    LedTask,
};

/* Array of task period timeouts */
UINT16 TaskPeriodTimeoutMs[ACTIVE_TASK_NUMBER] =
{
    LED_TASK_PERIOD_MS,
};

/* Array of task periods */
static UINT16 TaskPeriodCounterMs[ACTIVE_TASK_NUMBER] =
{
    0,
};

```

Listato 10 - Inserimento di LedTask nella tabella di esecuzione dello scheduler

# CORSO MPLAB X



di FRANCESCO FICILI  
e VINCENZO GERMANO

Continuiamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip Technology, che soppianta il vecchio MPLab IDE. In questa puntata ci occupiamo di come realizzare applicazioni embedded multitasking con i PIC32 che sfruttano l'USB Device come protocollo di comunicazione.

**N**elle puntate precedenti abbiamo esposto le caratteristiche ed il funzionamento dei microcontrollori PIC32 della Microchip e proposto esempi per utilizzare in pratica varie loro periferiche, come ad esempio SPI e UART, per poi passare all'analisi e all'implementazione di uno scheduler che ci permetterà di affrontare la programmazione embedded multitasking cooperativa. A questo punto abbiamo tutti gli strumenti e gli elementi per poter passare a spiegare, nel corso di questa puntata, funzionamento e utilizzo di una delle più comuni e utilizzate periferiche: l'USB Device.

## LO STACK USB MICROCHIP

Lo standard di comunicazione seriale USB (Universal Serial Bus) venne progettato per consentire di collegare periferiche eterogenee ad un Personal Computer usando una sola interfaccia standardizzata, un solo tipo di connettore, ma anche per implementare la funzionalità "plug and play" consentendo di collegare/scollegare i dispositivi senza dover necessariamente riavviare il computer. Negli anni l'USB si è affermato come uno dei più diffusi e comuni standard di comunicazione e ad oggi è presente nella maggior parte dei dispositivi elettronici, come

v2013-02-15	Windows	Microchip Libraries for Applications	
v2013-02-15	Mac OS X	Microchip Libraries for Applications	
v2013-02-15	Linux	Microchip Libraries for Applications	
v2013-02-15		Help Files	
v2013-02-15		Release Notes ①	

Fig. 1 - File di download per le MLA.

mouse, tastiere, memorie di massa a stato solido e a disco rigido, scanner per immagini, macchine fotografiche digitali, stampanti, navigatori satellitari, televisori, cellulari, smartphone ecc.

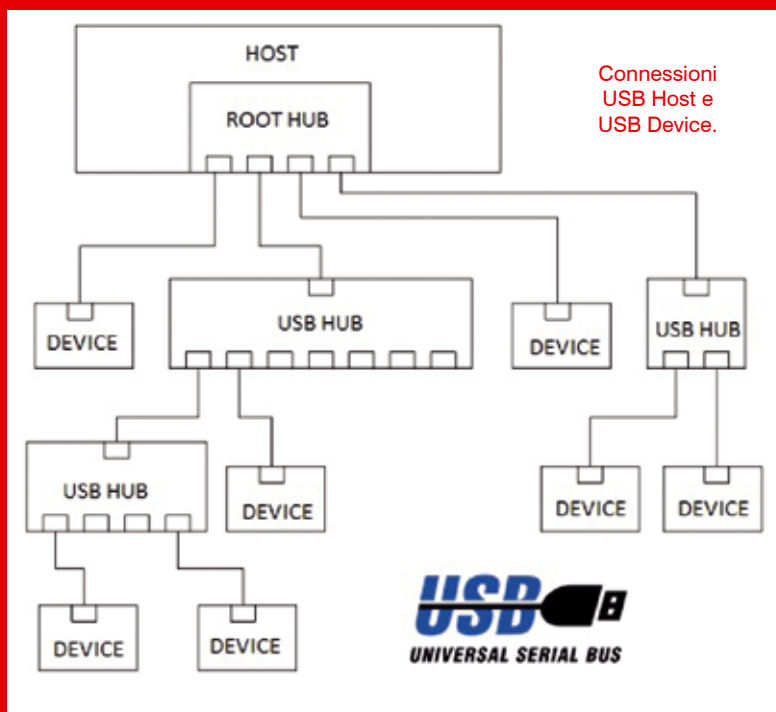
Una delle caratteristiche principali dell'USB è la sua **architettura asimmetrica** che collega un singolo Host e più periferiche Device mediante una struttura ad albero, grazie a

dei dispositivi chiamati hub (concentratori). In queste pagine affronteremo solo la parte riguardante la configurazione dell'USB Device (ricordiamo che l'USB prevede due dispositivi: l'Host e il Device; la loro funzione è spiegata nell'apposito riquadro "USB Host e USB Device") analizzando lo stack messo a disposizione dalla Microchip e integrandolo nella nostra applicazione; rimandiamo alla

## USB HOST E USB DEVICE

La prima versione dell'USB venne realizzata nel gennaio del 1996 e supportava collegamenti a solo 1,5 Mbit/s (velocità adeguate per mouse, tastiere e altri dispositivi "lenti") e una lunghezza del cavo massima di 3 metri. Con il tempo la sua evoluzione ha visto vari aggiornamenti, fino ad arrivare all'odierno USB 3.0, che è in grado di trasferire dati a una velocità di trasferimento di 4,8 Gbit/s (l'equivalente di circa 600 MB/s). Venne progettato per consentire a più periferiche di essere connesse usando una sola interfaccia standardizzata e un solo tipo di connettore, ma anche per migliorare la funzionalità "plug and play" consentendo di collegare o scollegare i dispositivi senza dover riavviare il computer. Una delle caratteristiche principali dell'USB è la sua architettura asimmetrica che consiste in un singolo gestore (identificato come Host) e più periferiche (identificate come Device) collegate ad albero, attraverso dei dispositivi chiamati hub (concentratori), come mostrato nella figura a lato; perciò nel protocollo di comunicazione si possono identificare due entità: Host e Device. Il primo comunica con il dispositivo e riceve dati di ingresso da esso rispetto alle azioni eseguite dall'utente, mentre il secondo è l'entità che interagisce direttamente con l'utente e può essere

-ad esempio- una tastiera o un mouse. Nei computer, il driver analizza i dati e consente l'associazione dinamica dei dati di I/O con la funzionalità dell'applicazione. Queste caratteristiche/funzionalità hanno permesso una rapida innovazione e lo sviluppo di tale periferica, ma anche la diversificazione di nuovi dispositivi.



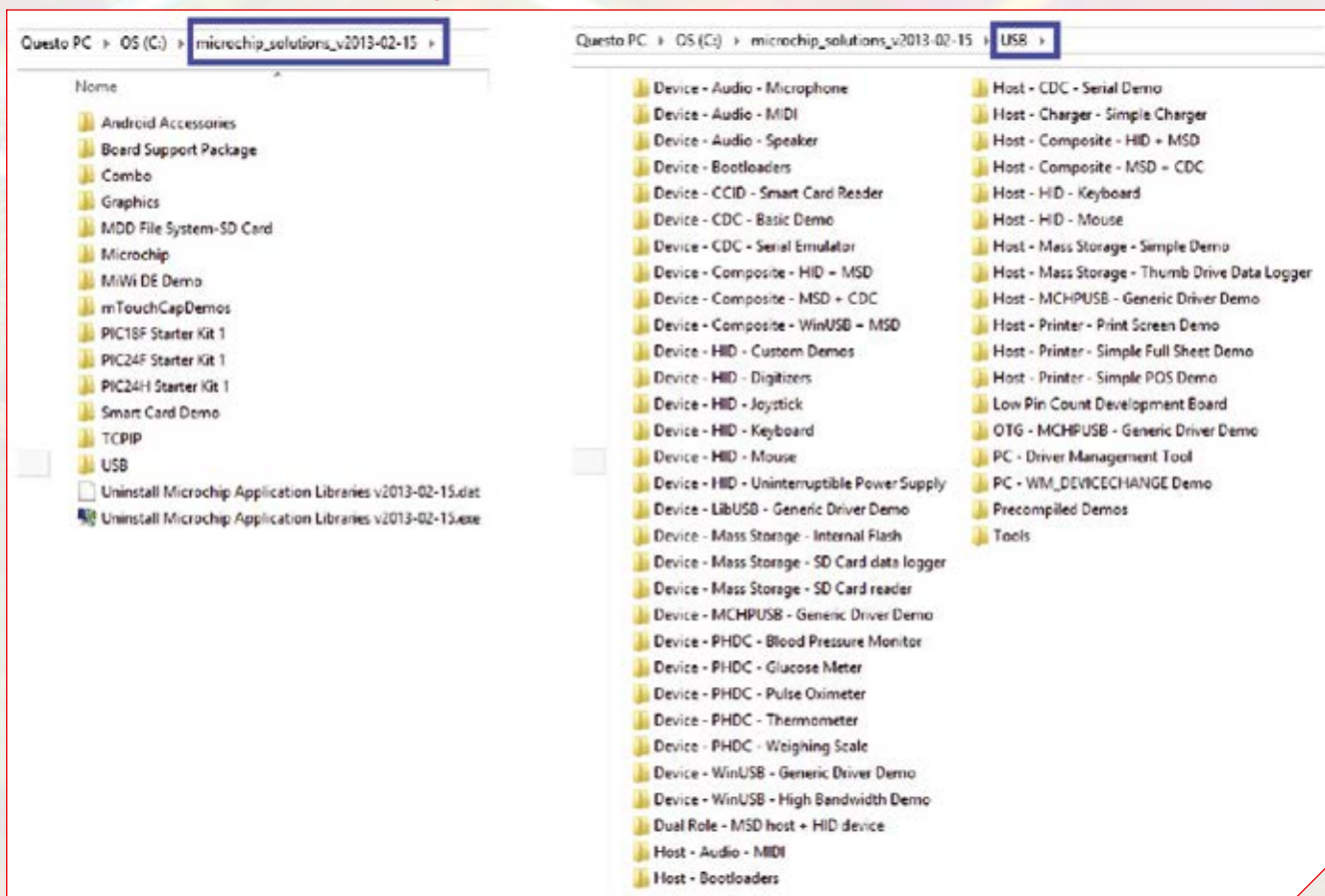
prossima puntata la trattazione e la configurazione dell'USB Host. Concentrando la nostra attenzione sullo stack Microchip, il primo passo da effettuare è scaricare dal sito ufficiale il pacchetto d'installazione delle "Microchip Libraries for Applications" (MLA) che sono le librerie di riferimento da cui poter attingere per importare tutto il necessario nei nostri progetti; esse migliorano l'interoperabilità tra le applicazioni che generalmente utilizzano più di una libreria e contengono gli strumenti base per lo sviluppo della maggior parte delle applicazioni (dal codice sorgente ai driver, dimostrazioni, documentazioni e utility), tutto preconfezionato e pronto per essere utilizzato. Per questa e le future puntate del corso, abbiamo preferito non utilizzare l'attuale versione (v2014-07-22) delle MLA perché non presenta il supporto d'interesse legato ai PIC32 (essendo nato un Framework dedicato "MPLAB® Harmony software suite") ma una delle versioni precedenti e nello specifico la

"v2013-02-15", come si vede dalla Fig. 1.

Un consiglio è quello di scaricare anche i file di "Help" perché molto utili per comprendere meglio la struttura del pacchetto complessivo delle MLA. In ogni caso, una volta scaricato e installato l'eseguibile, la situazione che ci si presenta nella cartella di destinazione selezionata è riportata in Fig. 2 (lato sinistro), da cui si evince l'organizzazione principale delle MLA. Si può notare come ci siano cartelle legate alla grafica, al TCPIP, USB e così via; all'interno di ogni cartella si possono trovare librerie, codici sorgenti, driver e utility, pronti per essere integrati nei nostri progetti. Notate che sebbene l'utilizzo di queste parti di codice messe a disposizione dalla Microchip possa sembrare immediato, non lo è, perché in ogni caso si deve comunque avere una buona conoscenza di quello che si vuole utilizzare.

La cartella che ci interessa per questa puntata del corso è ovviamente quella USB, visibile in

Fig. 2 - Struttura delle MLA e della cartella USB.



## Listato 1

```
// Hardware Configuration
#define USB_PING_PONG_MODE    USB_PING_PONG__FULL_PING_PONG

// Peripheral Configuration
#define MY_VID                0x04D8
#define MY_PID                0x0000
#define USB_POLLING
#define USB_PULLUP_OPTION     USB_PULLUP_ENABLE
#define USB_TRANSCEIVER_OPTION USB_INTERNAL_TRANSCEIVER
#define USB_EP0_BUFF_SIZE     8
#define USB_MAX_NUM_INT       (0+1)
#define USB_MAX_EP_NUMBER     6
```

Listato 1 - Esempio d'impostazione USB.

Fig. 2 (lato destro) che contiene i file d'interesse per la configurazione di questa periferica e permette una creazione abbastanza rapida di applicazioni in cui è necessaria la periferica USB; al suo interno troviamo altre sottocartelle di progetti di riferimento che coprono una gran varietà di applicazioni e casi di utilizzo.

### CONFIGURARE UN PROGETTO USB DEVICE

Per il progetto pratico che vi proponiamo abbiamo fatto riferimento a quanto presente nella cartella "Device - HID - Joystick"; in questa si trova un insieme di interfacce firmware modulari che gestiscono la maggior parte della comunicazione USB. In Fig. 3 viene mostrato un tipico flusso di un programma USB. Ogni progetto di riferimento MLA è scritto per avere un ambiente cooperativo multitasking, perciò nessuna funzione "bloccante" deve essere utilizzata (come ampiamente discusso nella puntata precedente di questo corso). Scendendo maggiormente nei dettagli dello stack USB Device della Microchip, per avere un'idea di come funzioni a basso livello, concettualmente possiamo notare quattro blocchi fondamentali (fate riferimento alla Fig. 3):

- *main.c* che contiene la funzione *main()*, cioè un ciclo infinito con servizi che attuano diversi compiti, che possono essere logicamente considerati sia come compiti USB, sia come task utente ma anche come altro;
- *USBDevice.c*, in cui sono inserite tutte le funzioni riguardanti l'USB, che vengono gestite dalla funzione *USBTasks()*, in linea con la filosofia del multitasking cooperativo spiegato nella puntata precedente;
- *hid.c* contenente le funzioni (macro) che possono essere usate da un utilizzatore umano per controllare il funzionamento del

sistema;

- *USBDescriptor.c*, che contiene le informazioni del descrittore USB per il particolare dispositivo e tali informazioni variano in base all'applicazione; a riguardo va detto che l'host non ha bisogno di capire intrinsecamente o analizzare il descrittore nello specifico, inoltre, generalmente esiste una "stringa descrittore" che fornisce una descrizione testuale del dispositivo.

Si può pensare all'*USBDescriptor.c* come a un array di byte di dati che descrivono i pacchetti del dispositivo, includendo il numero di pacchetti supportati dal dispositivo, la dimensione dei pacchetti e lo scopo di ogni byte/bit nel pacchetto.

Dopo aver installato l'eseguibile MLA, la cartella di destinazione relativa a questi file sorgente si trova nel path seguente: "*<Install Directory>\>\Microchip\USB*". Per poter utilizzare questi file bisogna importare nel proprio progetto i corrispondenti file ".h", come verrà mostrato successivamente nell'esempio pratico.

Tra i file da importare per il progetto, oltre ai quattro cardine appena illustrati, altri due meritano un'attenzione particolare: *l'HardwareProfile.h* e *l'usb\_config.h*. Il primo dei due generalmente viene usato perappare diverse configurazioni hardware per una definizione comune di codice da utilizzare. Al suo interno vengono definite varie impostazioni come, per esempio, su quale porta utilizzare i LED, la velocità di clock della scheda, che pin usare per l'acquisizione dei segnali e via dicendo.

Da quanto appena detto risulta chiaro come *l'HardwareProfile.h* possa assumere un ruolo fondamentale in ogni progetto, anche per poter continuare a mantenere una struttura ordinata e coerente. Mentre per quanto riguarda il file di intestazione *usb\_config.h*, come si può notare dal Listato 1 (un estratto del file), è un elemento chiave per la configurazione dello stack USB; infatti esso definisce vari parametri all'interno della pila, che determinano come lo stack deve funzionare e quali caratteristiche opzionali sono da includere. Ovviamente la modifica di queste opzioni può avere implicazione sulla dimensione del codice, utilizzo della RAM e throughput

dei dati. Come potete immaginare, senza un'opportuna definizione di questo file lo stack USB non funzionerà come previsto nella modalità da voi desiderata.

### LA CLASSE USB HID

Dopo aver compreso quali sono le parti fondamentali dello stack Microchip e come esse collaborano tra loro, passiamo alla trattazione della classe *Human Interface Device* (HID), legata all'Universal Serial Bus (USB).

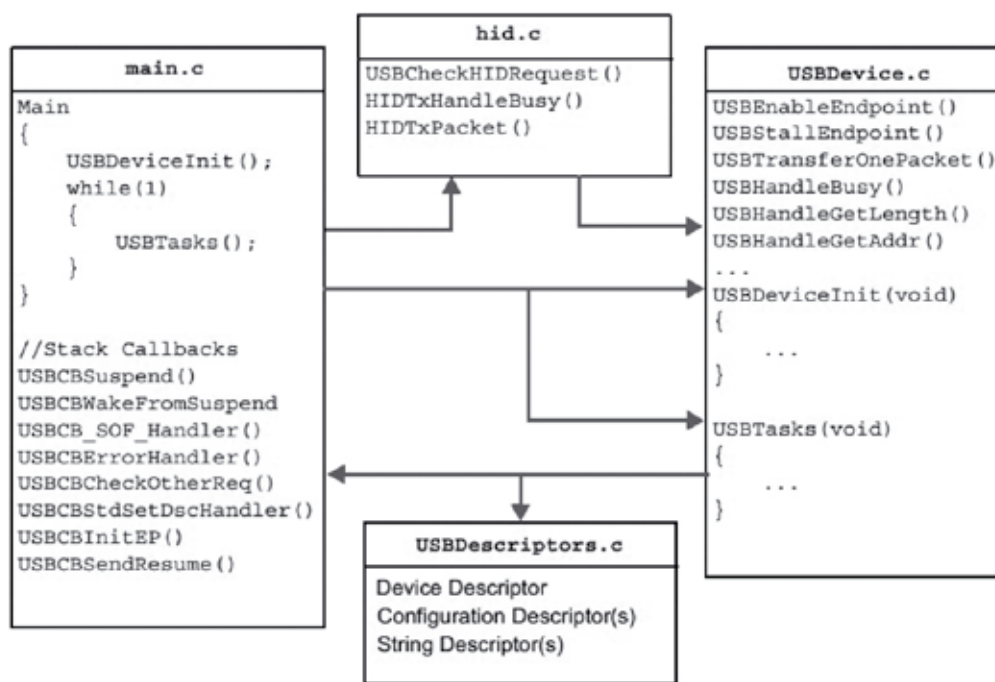
Lo standard HID, tra le altre cose, è stato adottato per consentire l'innovazione in dispositivi di input per computer e per semplificare il processo di installazione degli stessi, infatti oggi giorno la maggior parte dei dispositivi HID forniscono pacchetti di auto-installazione che possono contenere vari tipi e formati di dati. La classe HID è costituita da dispositivi che un essere umano può utilizzare per controllare il funzionamento dei sistemi come mouse, tastiera, joystick, pulsanti, interruttori e via dicendo.

È importante sapere che, oltre a fornire informazioni dalle interfacce umane, essa fornisce anche dati in output per indicare le azioni da parte del sistema operativo (computer). Per avere un'idea generale di come possa fun-

zionare una comunicazione di un dispositivo USB, possiamo affermare che è basata su *pipes* (tubi) o connessioni logiche dal controller host a un'entità logica, che si trova su un device e che viene identificata con il termine endpoint; visto che tali canali corrispondono 1-a-1 agli endpoint, i termini sono a volte usati come sinonimi.

Un dispositivo USB può avere fino a 32 endpoint (16 in, 16 OUT) ma è raro averne così tanti. Un endpoint è definito e numerato dal dispositivo durante la fase d'inizializzazione (il periodo immediatamente dopo il collegamento fisico noto come "enumeration") e quindi è permanente, mentre un canale può essere aperto e chiuso dopo una comunicazione. Utilizzando il firmware della Microchip appena presentato, la classe HID implementa le seguenti funzioni specifiche: USB Descriptor Table, Endpoint Configuration Table e Function Driver Table. Riguardo "*USB Descriptor Table*", ogni dispositivo USB deve sia fornire una serie di descrittori (strutture dati) che definiscono il dispositivo, sia informare l'host USB su quale driver di classe deve utilizzare. I descrittori dei dispositivi USB possono essere organizzati in tre gruppi: Device, Configuration e Strings.

Fig. 3 - Struttura concettuale di funzionamento dell'USB.



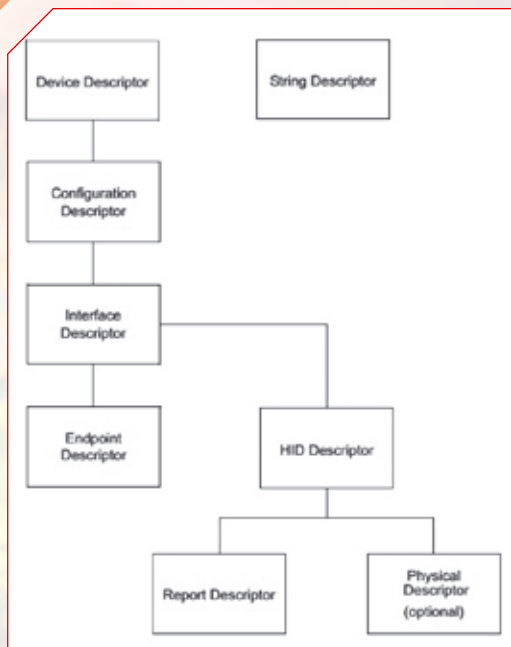


Fig. 4 - Descrittore USB per driver HID.

- *Device* identifica il tipo di dispositivo e fornisce il numero di possibili configurazioni.
- *Configuration* descrive i tipi di interfaccia e di endpoint utilizzati (inclusendo anche descrittori specifici della classe).
- *Strings*, che generalmente è opzionale, fornisce informazioni leggibili dall'utente che l'host può visualizzare.

La "Endpoint Configuration Table" viene utilizzata dallo stack USB per configurare correttamente tutti gli endpoint d'interfaccia e le impostazioni alternative, secondo quanto definito dalla tabella del particolare descrittore; identifica, inoltre, quali funzioni devono essere utilizzate per gestire l'evento che si verifica su ciascun endpoint. Infine, per quanto riguarda la "Function Driver Table", dal momento che un dispositivo può implementare più di una classe/driver della periferica USB di uno specifico fornitore, lo stack della Microchip usa una tabella per gestire l'accesso e supportare la funzione del driver. Ogni voce della tabella contiene le informazioni necessarie a gestire una singola funzione del driver. Ogni dispositivo USB ha una struttura descrittore a esso associata e può contenere più classi definite nel layer d'interfaccia, come ad esempio succede per l'HID. Una struttura descrittore ad albero è mostrata nella Fig. 4, la quale descrive un dispositivo di classe HID: come si vede, il descrittore HID indica quanti altri descrittori specifici seguo-

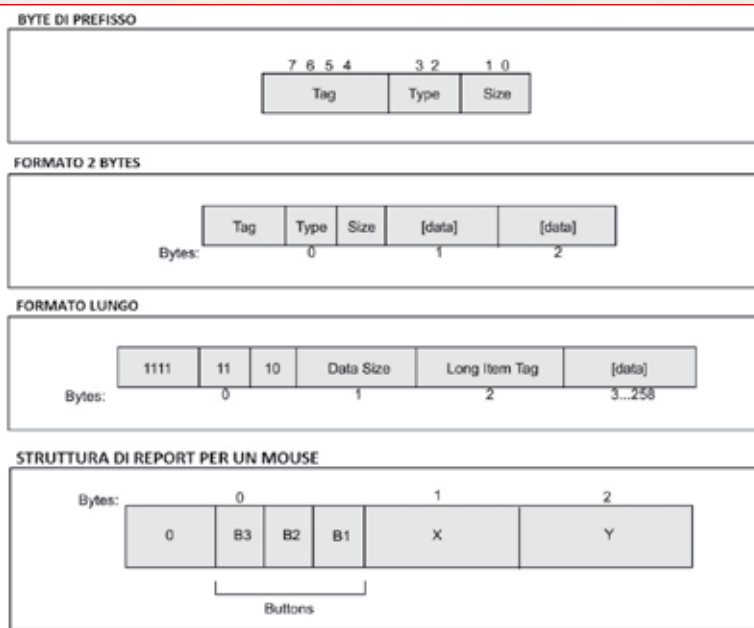


Fig. 5 - Formati dell'Item Report.

no. Notate che dev'essere presente almeno un "Report Descriptor", mentre quelli fisici sono opzionali. Un *Report Descriptor* descrive il formato e il significato di dati generati dal dispositivo, viene caricato dalla classe del driver HID dell'host utilizzando una richiesta specifica; dopo essere stato inizializzato, il dispositivo genera rapporti per indicare quando un utente interagisce con una periferica. È costituito da parti di informazione chiamate "Items" (elementi) e ognuno descrive un aspetto dei dati del report.

Generalmente un report di un item segue un formato costituito da un prefisso di un byte e un payload (informazioni utili); il primo contiene il tag, il tipo e la lunghezza del payload, come mostrato nella Fig. 5 (Byte di prefisso), nella quale sono anche riportati i due formati che può assumere un report: corto e lungo. Nell'esempio pratico che verrà mostrato successivamente verrà utilizzato il primo formato, mediante il quale sarà possibile inviare al computer gli spostamenti richiesti del mouse mediante l'interazione con un joystick (nel formato indicato come "struttura di report per un mouse").

È facile immaginare che i report di ingresso vengono utilizzati per inviare i dati sulle interazioni dell'utente con il dispositivo per l'host, mentre quelli di uscita vengono utilizzati dall'host per inviare dati al dispositivo di controllo, ad esempio un comando per accendere il LED.

## Listato 2

```

/*****
* User Joystick
*****/
/* Data direction register */
#define JOY_UP_TRIS          TRISBbits.TRISB10
#define JOY_RIGHT_TRIS       TRISBbits.TRISB9
#define JOY_LEFT_TRIS        TRISBbits.TRISB12
#define JOY_DOWN_TRIS        TRISBbits.TRISB11
#define JOY_FIRE_TRIS        TRISBbits.TRISB13

/* Resource Alias */
#define JOY_UP               PORTBbits.RB10
#define JOY_RIGHT            PORTBbits.RB9
#define JOY_LEFT             PORTBbits.RB12
#define JOY_DOWN             PORTBbits.RB11
#define JOY_FIRE             PORTBbits.RB13

```

Listato 2 - Configurazione delle porte del microcontrollore legate al Joystick.

Infine, i report vengono usati dall'host per configurare correttamente il device.

### LE MACRO USB

Con "Application Programming Interface" (API - Interfaccia di Programmazione di un'Applicazione), generalmente viene indicato un insieme di procedure disponibili al programmatore, raggruppate a formare un set di strumenti specifici per un determinato compito all'interno di un programma. La finalità è ottenere un'astrazione a più alto livello tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione, permettendo di evitare ai programmatori di riscrivere ogni volta tutte le funzioni necessarie al programma dal nulla, ovvero dal basso livello, rientrando quindi nel più vasto concetto di riuso di codice.

Ci sono due livelli di API relative al HID USB: il livello USB e il livello di sistema operativo. Il primo è un protocollo utilizzato riguardante i device e serve per poter inviare al sistema operativo le loro caratteristiche/capacità e permettere a quest'ultimo di gestirle correttamente. Invece il livello del sistema operativo offre una visione al livello superiore delle applicazioni, che non hanno più bisogno di includere il supporto per i singoli dispositivi.

Nel nostro caso la funzione più rilevante che utilizzeremo anche nell'esempio pratico sarà la macro "HIDTxPacket" contenuta all'interno del file "usb\_function\_hid.h", che ha per definizione la seguente struttura:

```
#define HIDTxPacket USBTxOnePacket
```

Grazie a questa macro è possibile inviare dei dati specifici da un particolare endpoint e

## Listato 3

```

/* Array of task to call */
void (*TaskArray[ACTIVE_TASK_NUMBER]) (void) =
{
    LedTask,
    UsbTask,
};

/* Array of task period timeouts */
UINT16 TaskPeriodTimeoutMs[ACTIVE_TASK_NUMBER] =
{
    LED_TASK_PERIOD_MS,
    USB_TASK_PERIOD_MS,
};

```

Listato 3 - Inserimento task USB.

## Listato 4

```

/*****
* Function:                UsbTask
* Input:                   None
* Output:                  None
* Author:                  F.Ficili
* Description:              Manage Usb tasks and state machines
* Date:                    28/02/15
*****/
void UsbTask (void)
{
    switch (SystemState)
    {
        /* System Initialization Phase */
        case InitializationState:

            /* Initialize port as digital */
            AD1PCFG = 0xFFFF;
            /* Initialize JOYSTICK pins */
            JOY_UP_TRIS    = LINE_DIRECTION_INPUT;
            JOY_RIGHT_TRIS = LINE_DIRECTION_INPUT;
            JOY_LEFT_TRIS  = LINE_DIRECTION_INPUT;
            JOY_DOWN_TRIS  = LINE_DIRECTION_INPUT;
            JOY_FIRE_TRIS  = LINE_DIRECTION_INPUT;

            /* Init Usb device */
            USBDeviceInit();
            break;

        /* System Normal operation Phase */
        case RunningState:
            /* Usb stack task */
            USBDeviceTasks();
            /* Emulate joystick */
            JoystickTask();
            /* Manage joystick */
            JoystickHidSendTask();
            break;

        /* Default */
        default:
            break;
    }
}

```

Listato 4 - Implementazione del task USB.

quindi poter trasferire informazioni verso il sistema operativo. La macro accetta i seguenti parametri:

- BYTE ep, che è l'endpoint verso il quale inviare i dati;
- BYTE\* data; è un puntatore ai dati che si

## Listato 5

```
void JoystickTask (void)
{
    /* Joystick state variable */
    static JoystickStateType JoystickState = DoNothing;

    if (JOY_UP == 1)
    {
        /* Jump to MoveUP */
        JoystickState = MoveUp;
    }
    else if (JOY_RIGHT == 1)
    {
        /* Jump to MoveRight */
        JoystickState = MoveRight;
    }
    else if (JOY_LEFT == 1)
    {
        /* Jump to MoveLeft */
        JoystickState = MoveLeft;
    }
    else if (JOY_DOWN == 1)
    {
        /* Jump to MoveDown */
        JoystickState = MoveDown;
    }
    else if (JOY_FIRE == 1)
    {
        /* Jump to MoveDown */
        JoystickState = FireButton;
    }
    else
    {
        JoystickState = DoNothing;
    }

    switch (JoystickState)
    {
        case MoveUp:
            /* Move the joystick up */
            HidBuffer[HID_REPORT_BUTTON] = 0;
            HidBuffer[HID_REPORT_X_AXIS] = 0;
            HidBuffer[HID_REPORT_Y_AXIS] = JOY_SPEED*MOVE_UP;
            break;

        case MoveDown:
            /* Move the joystick down */
            HidBuffer[HID_REPORT_BUTTON] = 0;
            HidBuffer[HID_REPORT_X_AXIS] = 0;
            HidBuffer[HID_REPORT_Y_AXIS] = JOY_SPEED*MOVE_DOWN;
            break;

        case MoveLeft:
            /* Move the joystick down */
            HidBuffer[HID_REPORT_BUTTON] = 0;
            HidBuffer[HID_REPORT_X_AXIS] = JOY_SPEED*MOVE_LEFT;
            HidBuffer[HID_REPORT_Y_AXIS] = 0;
            break;

        case MoveRight:
            /* Move the joystick down */
            HidBuffer[HID_REPORT_BUTTON] = 0;
            HidBuffer[HID_REPORT_X_AXIS] = JOY_SPEED*MOVE_RIGHT;
            HidBuffer[HID_REPORT_Y_AXIS] = 0;
            break;

        case FireButton:
            /* Move the joystick down */
            HidBuffer[HID_REPORT_BUTTON] = FIRE_ON;
            HidBuffer[HID_REPORT_X_AXIS] = 0;
            HidBuffer[HID_REPORT_Y_AXIS] = 0;
            break;

        case DoNothing:
            /* Move the joystick down */
            HidBuffer[HID_REPORT_BUTTON] = 0;
            HidBuffer[HID_REPORT_X_AXIS] = 0;
            HidBuffer[HID_REPORT_Y_AXIS] = 0;
            break;

        default:
            break;
    }
}
```

Listato 5 - Implementazione del task USB.

vogliono inviare;

- WORLD len; è la lunghezza dei dati da trasferire.

La macro prevede come ritorno un valore che permette di comprendere lo stato del trasferimento ("USB\_HANDLE"). Ne vediamo un'applicazione qui di seguito.

### ESEMPIO PRATICO: MINI JOYSTICK USB

Passiamo ora a utilizzare le informazioni viste fino a questo punto presentando un esempio pratico di un joystick USB. Il codice è stato realizzato e testato sulla DemoboardPIC32 pensata, progettata e realizzata per questo corso; le parti d'interesse nel nostro caso sono evidenziate con un rettangolo blu in Fig. 6, e sono il Joystick e il connettore USB. Ovviamente mediante il connettore USB ci collegheremo al computer, mentre mediante il joystick avremo la possibilità di muovere il mouse come un "trackpoint" con l'aggiunta del tasto di "conferma" (tasto sinistro del mouse).

La struttura del progetto rimane quella dello scheduler presentato nella puntata precedente ma con un'architettura mostrata in Fig. 7, dalla quale si evince la funzionalità dello scheduler in background, il blocco dei driver che comunica a basso livello con l'hardware e infine i due task che vengono richiamati per le funzionalità d'interesse (in questo caso il LED e l'USB).

L'organizzazione globale del progetto Joystick USB è riportata in Fig. 8, nella quale è possibile vedere sulla parte sinistra i file header e nella parte destra i file sorgente. È consigliabile avere una struttura di progetto ben organizzata per poter mantenere e gestire meglio la realizzazione/manutenzione dello stesso. All'interno della cartella "Bsp" (header file) sono stati inseriti i file di configurazione, come i configuration bits e il file spiegato precedentemente per la configurazione del nostro hardware, "hw\_profile.h"; mentre nella

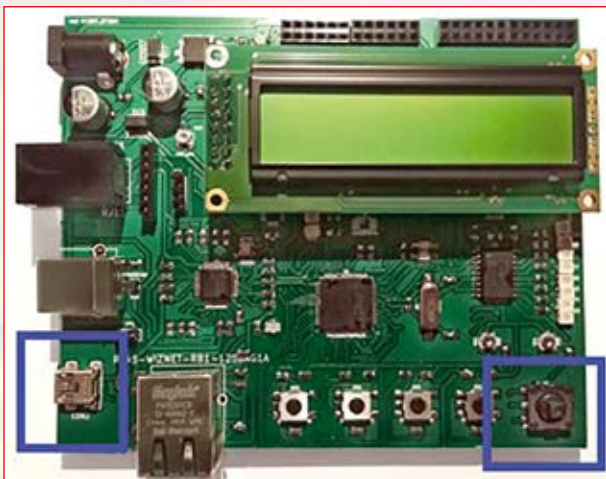


Fig. 6 - Parti d'interesse della DemoboardPIC32.

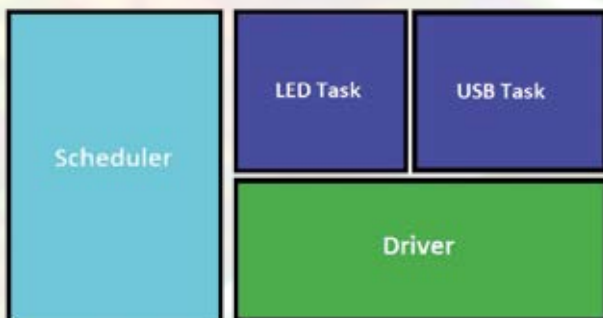


Fig. 7 - Architettura Software.

cartella equivalente del codice sorgente è stata inserita la gestione degli interrupt. Per quanto concerne la cartella "Stack", nella parte legata agli header sono stati inseriti i file per la configurazione dell'USB tra cui anche le funzioni HID (*usb\_function\_hid.h*); l'implementazione dell'USB è invece contenuta nella cartella "Stack" dei file sorgente, infatti al suo interno possiamo riconoscere i file precedentemente trattati. Le cartelle "Sched" e "Task" sono state ampiamente spiegate nel corso della puntata precedente. Avendo compreso la struttura del progetto e come quest'ultimo funziona, passiamo ai dettagli implementativi partendo dalla configurazione delle porte d'interesse sulla DemoboardPIC32. Analizzando il data-sheet del PIC32MX e guardando le interconnessioni di esso con il joystick, possiamo inserire nel file header di configurazione (*hw\_profile.h*) le definizioni visibili nel **Listato 2**. Configurare le porte da cui poter leggere i segnali provenienti dal joystick, aggiungiamo un nuovo task legato all'USB come mostrato precedentemente nell'architettura, all'interno del file "*scheduler.c*" (oltre a quello del LED già presente) e visibile nel **Listato 3**.

## Listato 6

```

/*****
* Function:   JoystickHidSendTask
* Input:      void
* Output:     void
* Author:     F.Ficili
* Description: Send HID data to Host
* Date:       07/03/15
*****/
void JoystickHidSendTask (void)
{
    /* Copy over the data to the HID buffer */
    HidReportIn[HID_REPORT_BUTTON] = HidBuffer[HID_REPORT_BUTTON];
    HidReportIn[HID_REPORT_X_AXIS] = HidBuffer[HID_REPORT_X_AXIS];
    HidReportIn[HID_REPORT_Y_AXIS] = HidBuffer[HID_REPORT_Y_AXIS];

    /* Send the 3 byte packet over USB to the host */
    LastTransmission = HIDTxPacket(HID_EP, (BYTE*)HidReportIn, 0x03);
}

```

Listato 6 - Funzione d'Invio dati mediante l'USB.

A questo punto creiamo un file dal nome "*usb\_task.c*" che inizializza la periferica e successivamente la gestisce. Facendo riferimento al **Listato 4**, possiamo vedere come anche in questo caso è stato differenziato uno stato in cui viene inizializzata la periferica d'interesse (*case InitializationState*) e una fase di normale funzionamento (*case RunningState*). Nella fase d'inizializzazione sono state definite le porte come digitali (dato che per impostazione predefinita sono impostate come analogiche) e poi inizializzate come ingressi per poter consentire la lettura degli stati del joystick. Infine viene richiamata la funzione per inizializzare la periferica USB. Sempre con riferimento al **Listato 4**, superato lo stato di inizializzazione, il codice che viene eseguito è quello di normale funzionamen-

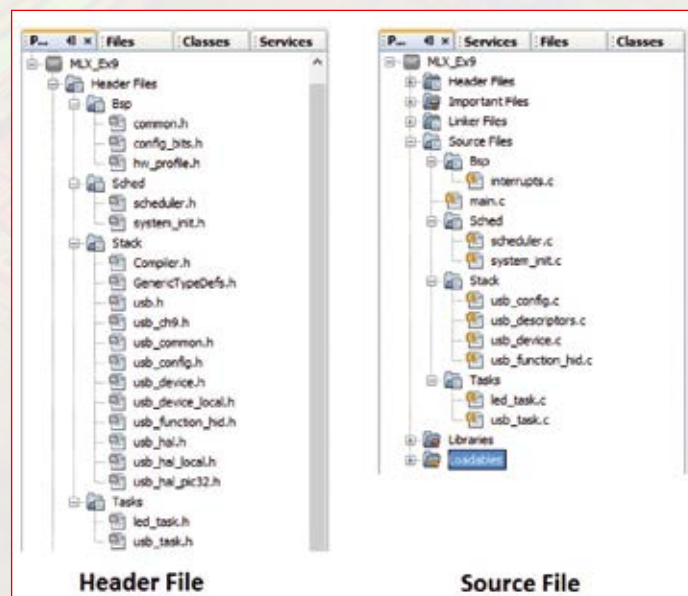


Fig. 8 - Struttura del programma Joystick USB.

## Listato 7

```
//Product string descriptor
ROM struct{BYTE bLength;BYTE bDscType;WORD string[28];}sd002={
sizeof(sd002),USB_DESCRIPTOR_STRING,
{'J','o','y','s','t','i','c','k',' ','D','e','m','o',' ','\
'E','l','e','t','t','r','o','n','i','c','a',' ','\','I','n'}
};
```

to (case *RunningState*) e infatti si può notare come al suo interno siano presenti le seguenti funzionalità:

- leggere lo stato del joystick, che si ottiene con *"JoystickTask()"*;
- inviare il comando corrispondente al computer, mediante USB *"JoystickHidSendTask()"*.

Nel **Listato 5** viene mostrata l'implementazione della funzione *JoystickTask()* che come anticipato legge lo stato del Joystick e poi la memorizza nella variabile *"JoystickState"* che viene successivamente utilizzata per capire in che direzione far muovere in mouse tramite il blocco *"switch (JoystickState)"*.

Come si può vedere, oltre alla definizione dei quattro movimenti che il mouse può fare, sono stati inseriti anche altri due casi: il primo riguarda la pressione di un tasto (pulsante sinistro del mouse) e il secondo è legato a uno stato di attesa d'azione utente. Quest'ultimo caso è fondamentale perché altrimenti il nostro mouse rimarrebbe nell'ultimo stato memorizzato di movimento e continuerebbe a scorrere verso uno dei quattro bordi dello schermo. Infine l'ultimo passo da compiere è inviare al computer i dati d'interesse e quindi gli spostamenti del mouse. Per far ciò è stata definita la funzione *JoystickHidSendTask()*, riportata nel **Listato 6**. Come si può vedere, tale funzione non fa altro che copiare i dati del mouse all'interno del buffer HID e infine trasmetterli verso il computer mediante la macro analizzata precedentemente. A questo punto il programma è completo, non bisogna far altro che compilarlo, scaricarlo sulla DemoboardPIC32 e collegarla al computer per poter utilizzare il proprio *trackpoint* "fatto

in casa". La maggior parte dei sistemi operativi riconosce i dispositivi standard HID USB come tastiere/mouse, senza bisogno di driver speciali; infatti, una volta collegati, il computer li installa e successivamente appare sullo schermo un messaggio di conferma che il dispositivo conforme è stato riconosciuto/ installato correttamente e può essere utilizzato, come mostrato nella sequenza di messaggi d'installazione in **Fig. 9**. È doveroso precisare che quanto appena presentato come progetto pratico è compatibile con le versioni di Windows XP, Vista e 7, ma non con il più recente sistema operativo di Microsoft: Windows 8. Infatti in quest'ultimo caso la periferica non verrà riconosciuta e non sarà possibile utilizzarla in quanto è presente un'incompatibilità di driver. Come visibile nell'immagine di sinistra della **Fig. 9**, il descrittore può essere personalizzato a piacimento per poter far visualizzare, in fase d'installazione, il testo desiderato: nel caso specifico abbiamo impostato Joystick **Demo Elettronica In**. La personalizzazione del descrittore è fattibile modificando, all'interno della cartella "Stack" del progetto, il file *"usb\_descriptors.c"*, come riportato nel **Listato 7**.

Fate attenzione al fatto che bisogna modificare anche la lunghezza della stringa da inviare, oltre al suo contenuto, altrimenti l'operazione non è possibile.

## CONCLUSIONI

In questa puntata abbiamo introdotto uno dei più comuni e diffusi protocolli di comunicazione, ossia l'USB lato device; in più abbiamo analizzato lo stack della Microchip per poi applicarlo ad un esempio pratico legato alla DemoboardPIC32, realizzando un mouse mediante il joystick presente su tale scheda. Nella prossima puntata passeremo ad analizzare la parte USB Host, per completare la panoramica della trattazione riguardante l'USB. ■



Fig. 9 - Sequenza di messaggi d'installazione su Windows XP.

# CORSO MPLAB X



di FRANCESCO FICILI  
e VINCENZO GERMANO

**Continuiamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato sviluppato da Microchip Technology per sostituire l'MPLab IDE. Ci occupiamo di come realizzare applicazioni che impiegano lo stack USB host di Microchip. Settima puntata.**

**N**ella scorsa puntata abbiamo visto come realizzare un'applicazione USB Device su PIC32, servendoci dello stack USB open source di Microchip e dello scheduler introdotto nella puntata 5; adesso facciamo più o meno la stessa cosa, ma utilizzando lo stack USB Host invece del device. L'applicazione di esempio che andremo a realizzare ci permetterà di "loggarci" su una comune chiavetta USB la tensione letta su un canale analogico del microcontrollore che abbiamo collegato a un potenziometro.

## **LO STACK USB HOST**

Sappiamo che la tipica struttura prevista dallo standard USB è costituita da un'unità Host e da una o più unità device. I Device USB non pos-

sono comunicare tra di loro, ma solamente con l'Host, che di fatto assume il ruolo di master del bus USB. Nella maggior parte delle applicazioni che in genere si prendono in considerazione, il ruolo dell'Host è ricoperto da un Personal Computer, il quale, disponendo di un sistema operativo piuttosto complesso (come ad esempio Microsoft Windows o Mac OS) e di risorse hardware virtualmente infinite per l'applicazione, può essere agevolmente configurato dinamicamente (se non ne dispone già) con il relativo driver della periferica USB specifica che deve andare ad interfacciare. Nel caso dell'embedded host, invece, la situazione è piuttosto differente: nella maggior parte delle applicazioni non si dispone di un sistema operativo stile Windows/

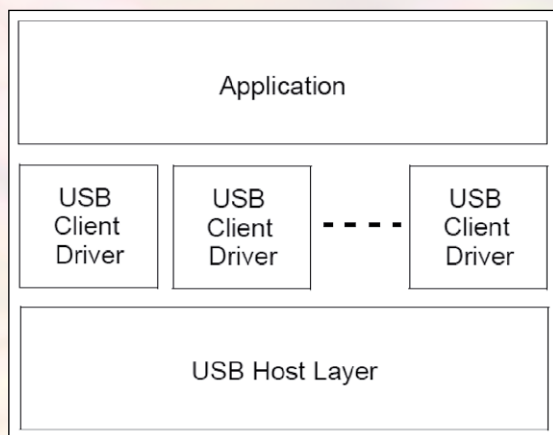


Fig. 1 - Architettura dello stack USB Host.

Linux, ma di un semplice RTOS o, come nel nostro caso, solo di uno scheduler, e non è possibile aggiornare il sistema con un nuovo driver all'occorrenza. Bisogna quindi trattare il problema in maniera differente, considerando l'applicazione specifica che utilizzerà il bus e progettando il firmware di conseguenza.

## ARCHITETTURA DELLO STACK USB HOST

L'architettura dello stack USB Host di Microchip è rappresentata schematicamente in Fig. 1, nella quale potete vedere che consiste in tre layer, ossia Application, USB Client

Driver Layer e USB Host Layer, di seguito descritti.

- **Application Layer:** questo è lo strato applicativo dell'implementazione e raggruppa tutte le funzioni di alto livello dell'applicazione specifica. In genere viene sviluppato da zero dallo sviluppatore dell'applicazione.
- **USB Client Driver Layer:** come sappiamo, ogni dispositivo USB implementa una determinata funzione (ad esempio un mouse, una tastiera, una memoria, ecc.). Ognuno di questi dispositivi ha un'implementazione specifica differente e di conseguenza necessita di un driver specifico per poter essere interfacciato. Questo strato implementa proprio i vari client driver di tutti i possibili dispositivi USB che possono essere connessi all'embedded Host.
- **USB Host Layer:** questo layer fornisce un'astrazione del protocollo USB e in particolare si occupa dell'implementazione dei seguenti servizi:
  - identificazione del dispositivo;
  - enumerazione del dispositivo;
  - gestione del client driver specifico;
  - astrazione della comunicazione con il dispositivo.

Quando un dispositivo viene connesso al bus, il layer host lo identifica leggendone i descrittori (specifiche strutture identificative definite dalla specifica USB 2.0), dopodiché controlla se un driver del dispositivo è presente all'interno di una specifica lista di client driver e se ne trova uno adatto lo inizializza per consentirne l'utilizzo ai livelli superiori.

L'operazione di identificazione ed enumerazione viene gestita in background da uno specifico task realizzato sotto forma di una macchina a stati, la cui implementazione è visibile in Fig. 2.

Come si può vedere, una volta collegato, un dispositivo passa attraverso vari stati, prima di poter essere "battezzato" come configurato e quindi passare in stato di running, dove può essere effettivamente utilizzato dall'host.

## CLIENT DRIVER TABLE

Come abbiamo visto in precedenza, ogni dispositivo USB (o famiglia di dispositivi) ha uno specifico driver (client driver) che ne permette la gestione. Dato che un host deve

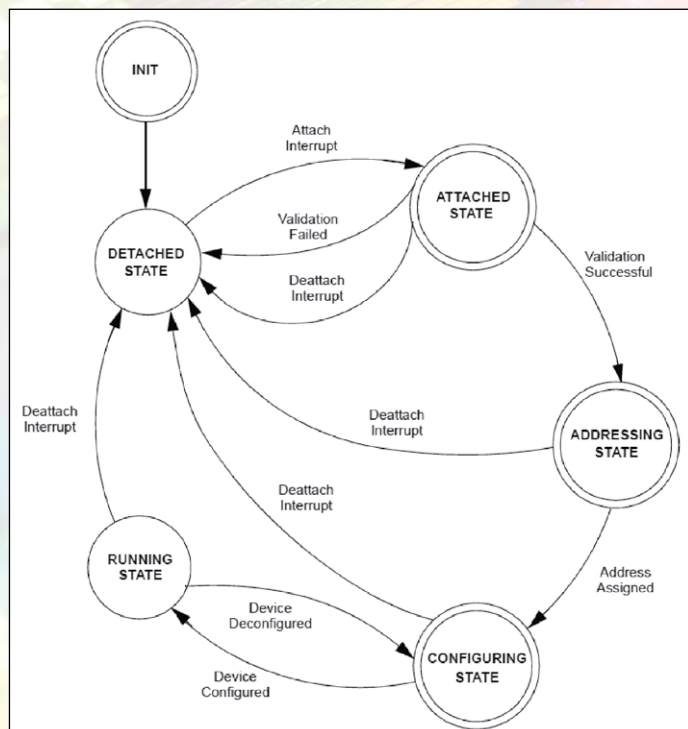


Fig. 2 - Implementazione della macchina a stati che gestisce l'enumerazione di un dispositivo USB nello stack USB Host di Microchip.

in genere supportare più di un dispositivo (questo può avvenire nel caso in cui debba essere supportata una classe di dispositivi o anche nel caso in cui si supporti un dispositivo singolo ma composito, ossia che implementa più di una funzione, con relativa interfaccia, al suo interno), quest'ultimo implementa generalmente più di un client driver. Nell'implementazione dello stack USB Host di Microchip, questo requisito viene soddisfatto implementando una tabella, che prende il nome di Client Driver Table (o CDT). Ogni entry di questa tabella corrisponde ad un singolo client driver e contiene un puntatore alla funzione di inizializzazione ed alla callback che si occupa della gestione degli eventi (event handling callback). La Fig. 3 rappresenta schematicamente l'implementazione della Client Driver Table. Quando un device viene connesso al bus, il layer host ne legge i descrittori e determina se il dispositivo può essere supportato o no. In caso affermativo, la corretta entry della client driver table viene indicizzata e viene chiamata la routine di inizializzazione. Una volta inizializzato il dispositivo, quando un particolare evento viene generato in corrispondenza di una qualche attività, lo specifico event handler viene chiamato per la gestione dell'evento specifico.

### TARGETED PERIPHERAL LIST

Come detto in precedenza, un'implementazione embedded host, a differenza di un Personal Computer, può supportare solo un limitato set di dispositivi USB. Questo set, nell'implementazione di Microchip, è definito da una Targeted Peripheral List (o TPL), ossia una lista di dispositivi supportati.

I dispositivi supportati possono essere identificati in una delle due seguenti modalità:

- Combinazione VID/PID;
- Combinazione classe/sottoclasse/protocollo.

Un esempio di un elemento di una TPL che utilizza la combinazione classe/sottoclasse/protocollo è riportato in Fig. 4.

La specifica USB impone che ogni disposi-

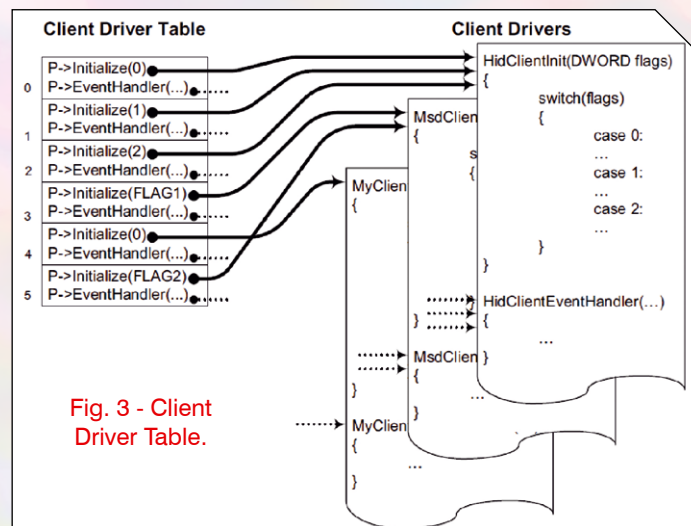


Fig. 3 - Client Driver Table.

tivo sia identificato per mezzo di una VID (acronimo di Vendor ID) e un PID (Product ID). La combinazione di questi due numeri identifica una specifica famiglia di prodotti di uno specifico venditore, in modo da poter determinare un'implementazione specifica e come supportarla. Un altro metodo fornito dalla specifica per determinare il comportamento di un generico dispositivo USB, e di conseguenza progettare l'apposito client driver, consiste nell'identificare la classe di appartenenza. Sempre secondo la specifica, i dispositivi USB sono suddivisi in classi, e all'interno di esse in sottoclassi e protocolli supportati. Identificando questa combinazione è possibile determinare il comportamento specifico di un generico dispositivo. Sia il VID/PID che la combinazione classe/sottoclasse/protocollo possono essere agevolmente ricavate da un dispositivo esaminando i suoi descrittori.

Nello stack USB Host di Microchip, la TPL è modellata come una tabella che associa l'identificato del dispositivo (che può essere determinato con uno dei due sistemi descritti in precedenza) con un elemento della Client Driver Table. In questo modo, quando un dispositivo è collegato, la TPL viene analizzata per determinare se il dispositivo è supportato dall'host, ed in caso positivo, un indice indica quale entry della Client Driver Table deve essere utilizzata per gestire il dispositivo. Oltre a fornire un riferimento per la CDT, nella TPL

Description	Class Name	Class Code	Subclass Code	Protocol
Flash Drive	Mass Storage	0x08	0x06	0x50

Fig. 4 - Elemento di una TPL identificato tramite la combinazione classe/sottoclasse/protocollo.

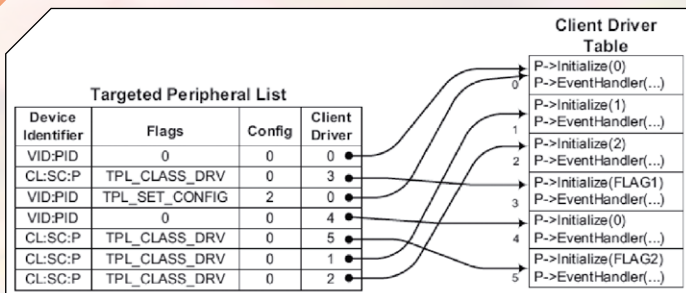


Fig. 5 - Relazione TPL - Client Driver Table.

sono contenute altre informazioni come flag e parametri di configurazione che possono essere passati allo specifico client driver in funzione dei requisiti dell'applicazione specifica. La Fig. 5 fornisce una rappresentazione schematica di quanto esposto in precedenza. Si noti, sempre dalla Fig. 5 che una singola entry della TPL può puntare allo stesso driver nella CDT (nell'esempio la prima e la terza); questo permette, quando dei dispositivi sono identificati per mezzo della combinazione VID/PID, di utilizzare lo stesso driver, nel caso l'implementazione della loro interfaccia sia effettivamente la stessa. Utilizzate in combinazione, la TPL e la CDT costituiscono un potente meccanismo per la gestione di un gruppo eterogeneo di dispositivi USB che debbano essere supportati da un singolo embedded host. Chi volesse maggiori informazioni sulla struttura dello stack USB host di Microchip può fare riferimento ad alcune interessanti application notes, come ad esempio l'AN1140 e l'AN1141.

### LA CLASSE USB MSD

Quanto descritto vale per ogni implementazione USB Host realizzata con lo stack USB di Microchip. Tuttavia, come abbiamo visto in precedenza, a seconda del dispositivo che andiamo ad interfacciare, l'implementazione specifica dell'applicazione lato Host cambierà, in quanto bisognerà implementare di sicuro lo specifico driver e anche qualche altro strato aggiuntivo in funzione dell'applicazione. Fortunatamente anche in questo Microchip ci viene incontro, fornendoci una serie di client driver per le più comuni classi di dispositivi USB (come ad esempio flash drive, dispositivi HID, stampanti, ecc.) e anche altri strati aggiuntivi necessari per alcune applicazioni (ad esempio nel caso della classe MSD vengono forniti anche dei componenti per la gestione del file system e dell'interfaccia SCSI). Tali pacchetti sono parte dello stack USB (o più

genericamente delle MLA) ed esiste anche un'implementazione general purpose (chiamata generic host) che costituisce di fatto un template per l'implementazione di client driver per le classi non ufficialmente supportate dallo stack.

L'applicazione specifica che analizzeremo nel dettaglio in questa puntata è l'implementazione di un Host USB capace di gestire una classe MSD (mass storage device) con file system, quindi implementeremo di fatto un embedded host capace di interfacciare le comuni Pen Drive USB. La struttura che andremo a realizzare si appoggia sull'architettura descritta in precedenza ed aggiunge alcuni strati proprio al di sopra del mass storage client driver, come visibile in Fig. 6. Tali strati aggiuntivi sono:

- **SCSI Command Set**; questo livello gestisce l'interfaccia SCSI verso la memoria;
- **File System**; implementa la gestione del file system (è utilizzato il tipo FAT32) e costituisce l'effettiva interfaccia di alto livello utilizzata dall'applicazione per interagire con il disco;
- **Application**; questo strato implementa l'applicazione specifica, in questo caso una applicazione di datalogging.

Lo strato di gestione del file system mette a disposizione diverse API per la gestione del disco, come funzione di scrittura, di lettura, gestione delle directory ed altro ancora. Per una descrizione dettagliata delle varie API è possibile consultare l'AN1045: "Implementing

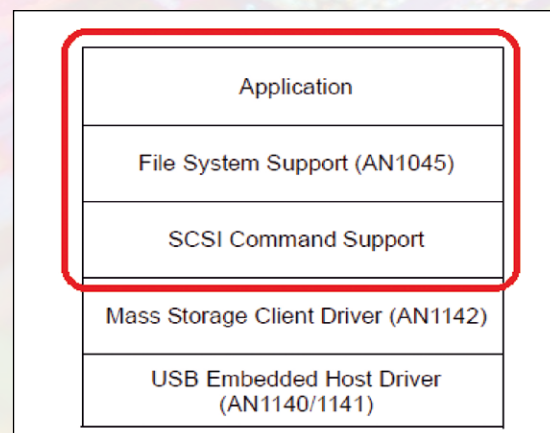


Fig. 6 - Architettura di un'applicazione host che supporta la classe USB MSD.

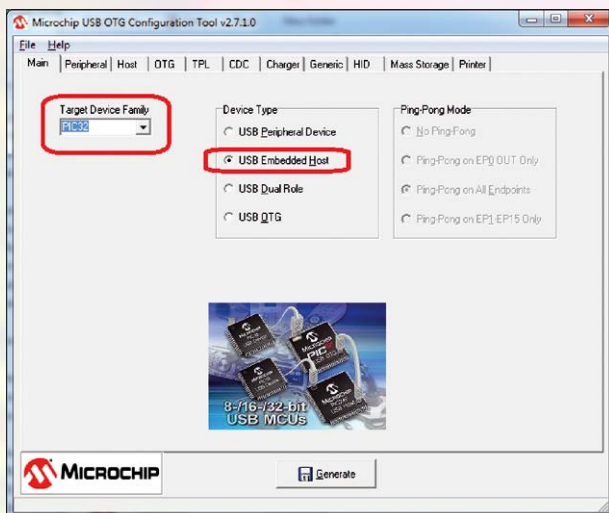


Fig. 7 - Schermata principale del tool UsbConfig.

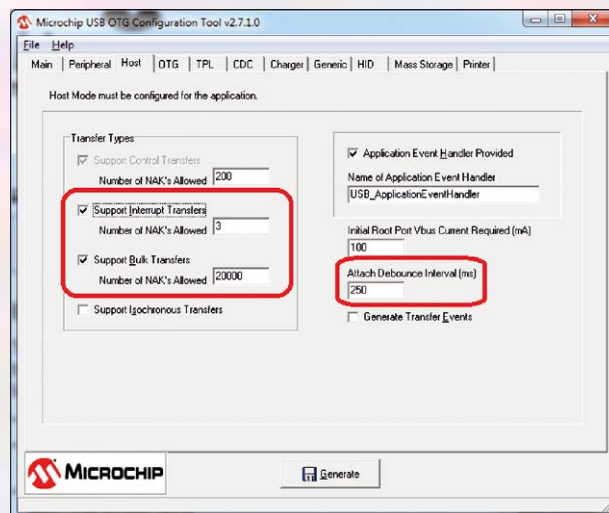


Fig. 8 - Configurazione dei parametri Host.

File I/O Functions Using Microchip's Memory Disk Drive File System Library".

### CONFIGURAZIONE DI UN PROGETTO USB HOST

Come abbiamo potuto notare dai paragrafi precedenti, l'implementazione di un'applicazione USB host che interagisca con una memoria USB può risultare un'operazione di una certa difficoltà, in quanto il framework host è ancora più complesso dell'implementazione device: in particolare, anche la sola configurazione del framework per l'applicazione specifica può essere una procedura decisamente complicata da eseguire. In un progetto USB host realizzato con lo stack USB di Microchip, tutte le configurazioni relative al framework sono contenute all'interno della coppia di file `usb_config.c/usb_config.h`. In particolare oltre alla TPL ed alla CDT, sono contenute tutte le impostazioni generiche e specifiche delle particolari classi supportate. Poiché tali configurazioni variano da applicazione in applicazione (così come anche la composizione della TPL e della CDT), Microchip ha realizzato un'applicazione, chiamata appunto UsbConfig, che permette di autogenerare il codice relativo alla configurazione tramite un'interfaccia grafica intuitiva. Vediamo come funziona questa applicazione generando i file relativi al progetto pratico che illustreremo in questa puntata. La schermata iniziale del tool si presenta come in Fig. 7 e ci permette di selezionare la piattaforma e la tipologia di dispositivo (oltre all'embedded host e al device sono presenti anche delle implementazioni ibride, come il dual role e l'OTG, che non sono

trattate in questa puntata). Scegliamo PIC32 come piattaforma e Embedded Host come "device type" e andiamo avanti, spostandoci sul tab "Host". Da qui è possibile scegliere il tipo di trasferimento da utilizzare e altre opzioni di base come il nome della callback di event handling, la corrente massima da erogare, ecc. Selezioniamo Interrupt e bulk come modalità di trasferimento dati (la modalità isocrona non è necessaria per l'applicazione), 100 mA come massima corrente e portiamo l'attach debounce interval a 250 ms, in modo da poter supportare anche i dispositivi più lenti. Non supportiamo i transfer events e manteniamo il nome predefinito per l'application event handler. In Fig. 8 è riportato uno screenshot della configurazione da inserire. Spostiamoci ora sul tab TPL per creare la "Targeted Peripheral List", come illustrato

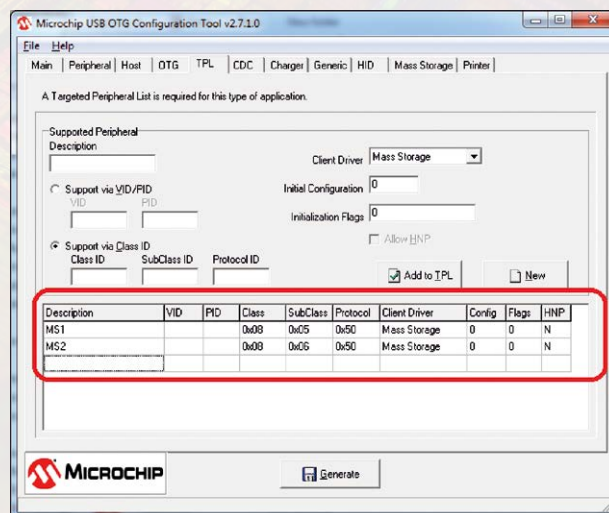


Fig. 9 - Creazione della TPL.

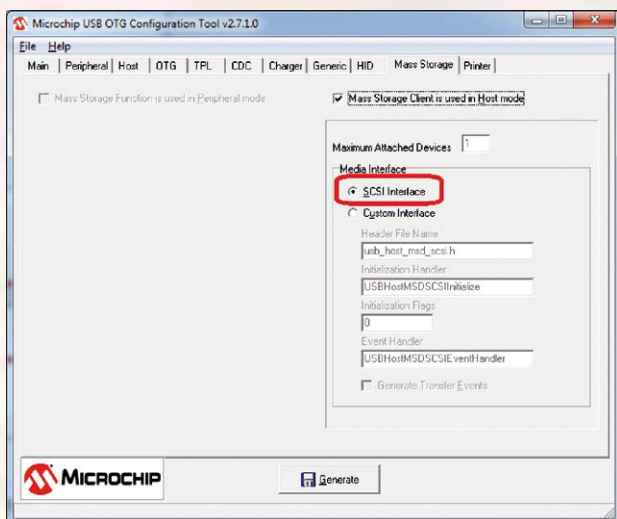


Fig. 10 - Impostazione dell'interfaccia verso il disco.

in Fig. 9. In questo tab bisogna inserire le varie entry della TPL, che consistono nei parametri identificativi del dispositivo (con i due metodi descritti in precedenza, combinazione VID/PID o combinazione classe/subclasse/protocollo), il relativo client driver (per questo è presente un dropdown box che

contiene i client driver più comuni, oppure è possibile utilizzare la generic class per implementarne uno nuovo), oltre ad eventuali flag e configurazioni. Nel nostro caso inseriremo due entry, entrambe collegate al Mass Storage Client Driver: la prima identificata dalla terna classe/subclasse/protocollo 0x08 – 0x05 – 0x50 e la seconda dalla terna 0x08 – 0x06 – 0x50. Questo perché alcune periferiche mass storage utilizzano la sottoclasse 0x05 mentre altre la 0x06. In questo modo siamo certi che il nostro embedded host le riconosca entrambe, collegando sempre lo stesso client driver. In entrambi i casi non inseriamo flag né opzioni particolari.

Non ci rimane che spostarci sul tab “Mass Storage” per inserire i dettagli dell'interfaccia specifica, come illustrato in Fig. 10. Nel nostro caso utilizzeremo l'interfaccia SCSI per la gestione del disco, quindi spuntiamo la relativa voce e completiamo. A questo punto è possibile cliccare sul pulsante “Generate”

## Listato 1

```
// *****
// USB Embedded Host Targeted Peripheral List (TPL)
// *****

USB_TPL usbTPL[] =
{
    { INIT_CL_SC_P( 0x08ul, 0x05ul, 0x50ul ), 0, 0, {TPL_CLASS_DRV} } // MS1
    ,
    { INIT_CL_SC_P( 0x08ul, 0x06ul, 0x50ul ), 0, 1, {TPL_CLASS_DRV} } // MS2
};
```

Listato 1 - Implementazione C della TPL generata tramite il tool.

## Listato 2

```
// *****
// Client Driver Function Pointer Table for the USB Embedded Host foundation
// *****

CLIENT_DRIVER_TABLE usbClientDrvTable[] =
{
    {
        USBHostMSDInitialize,
        USBHostMSDEventHandler,
        0
    },
    {
        USBHostMSDInitialize,
        USBHostMSDEventHandler,
        0
    }
};
```

Listato 2 - Implementazione C della CDT generata tramite il tool.

per creare i file; se apriamo il file `usb_config.c`, troviamo l'implementazione in C della TPL e della CDT, riportate, rispettivamente, nel **Listato 1** e nel **Listato 2**.

### ESEMPIO PRATICO: DATALOGGER USB HOST

A questo punto possiamo passare alla descrizione dell'applicazione di esempio che presentiamo in questa puntata. Come abbiamo già detto nei paragrafi precedenti, la nostra idea è quella di realizzare una applicazione di datalogging che salvi su una comune per drive USB i valori letti da un canale analogico collegato ad un potenziometro. Un'applicazione interna di controllo darà il via e terminerà la fase di log. L'architettura del software è illustrata in **Fig. 11** e come si può vedere è basata, come di consueto ormai, sullo scheduler presentato nelle puntate precedenti. Come al solito, per fare in modo che lo scheduler lavori correttamente, trattandosi di un'implementazione non pre-emptive, dovremo sviluppare i nostri task utilizzando il paradigma del multitasking cooperativo. Come possiamo vedere dallo schema architetturale, sono stati previsti i quattro task seguenti.

- **UsbTask**: questo task si occupa di chiamare il task che mantiene le macchine a stati del driver USB Host (`USBTasks()`).
- **LogTask**: questo task implementa le macchine a stati dell'applicazione di logging. In particolare viene implementata una macchina a stati che si occupa del controllo (`LogControl()`) ed una macchina a stati che si occupa della gestione effettiva della memoria (`LoggingStateMachine()`).
- **AdcTask**: questo task implementa la macchina a stati che legge i dati dal convertitore analogico digitale e genera gli eventi di logging che vengono intercettati dal task di log.
- **LedTask**: questo task si occupa della gestione dei LED di sistema.

Oltre che sullo scheduler, la nostra architettura si appoggia sullo stack USB Host presentato in precedenza (e in particolare sull'implementazione del client driver MSD) per la gestione dell'USB e sulle librerie di sistema e sui driver delle periferiche per la

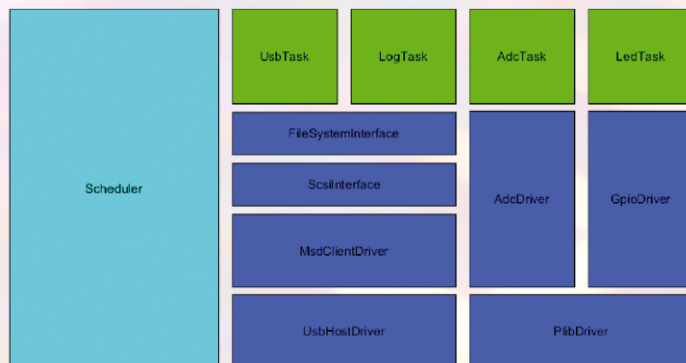


Fig. 11 - Architettura software.

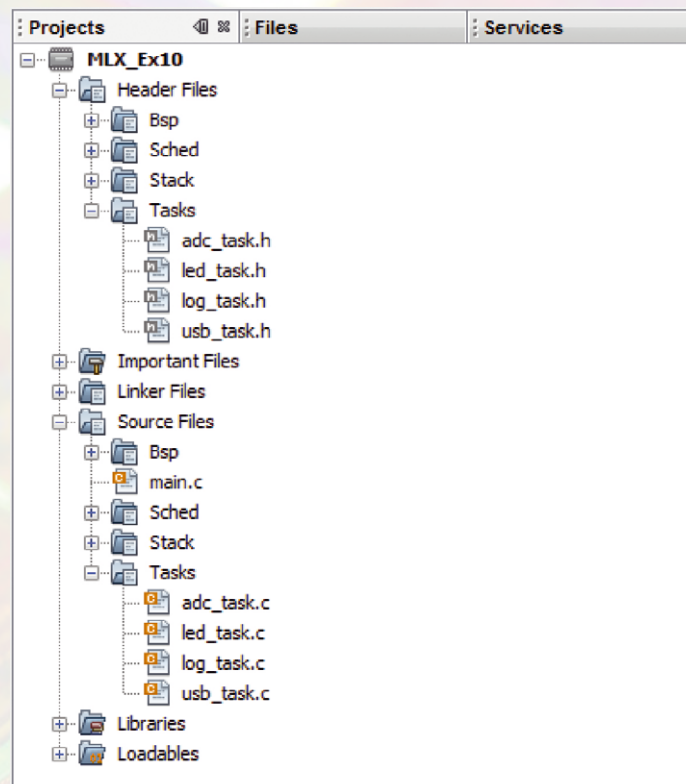


Fig. 12 - Struttura del Project Manager.

### Listato 3

```
/* Array of task to call */
void (*TaskArray[ACTIVE_TASK_NUMBER]) (void) =
{
    LedTask,
    UsbTask,
    LogTask,
    AdcTask
};

/* Array of task period timeouts */
UINT16 TaskPeriodTimeoutMs[ACTIVE_TASK_NUMBER] =
{
    LED_TASK_PERIOD_MS,
    USB_TASK_PERIOD_MS,
    LOG_TASK_PERIOD_MS,
    ADC_TASK_PERIOD_MS,
};
```

Listato 3 - Array task e periodicità.

## Listato 4

```

/*****
* Function:      void AdcStateMachine (void)
* Input:         None
* Output:        None
* Author:        F.Ficili
* Description:   Manage adc data acquisition state machine
* Date:          11/04/15
*****/
void AdcStateMachine (void)
{
    static AdcSmStateType AdcSmState = Idle;
    static UINT16 AcqDelayCounter = 0;
    UINT16 AdcDataRaw = 0;

    switch (AdcSmState)
    {
        case Idle:
            /* Check logger status */
            if (ReceiveEvt(&LoggerReady))
            {
                /* Jump to WaitLogDelay */
                AdcSmState = WaitLogDelay;
            }
            break;

        case WaitLogDelay:
            /* Increment counter */
            AcqDelayCounter++;

            /* If counter expires */
            if (AcqDelayCounter >= ACQ_DELAY_MS)
            {
                /* Reset counter */
                AcqDelayCounter = 0;
                /* Jump to AcquireData */
                AdcSmState = AcquireData;
            }
            break;

        case AcquireData:
            /* Get data */
            AdcDataRaw = ReadAdcData();
            /* Convert data to string */
            itoa(AdcDataBuffer, AdcDataRaw, 10);

            /* Jump to LogData */
            AdcSmState = LogData;
            break;

        case LogData:
            /* If a stop log evt is not received */
            if (!ReceiveEvt(&StopLog))
            {
                /* Log data event */
                GenerateEvt(&LogDataEvt);
                /* Jump to WaitLogDelay */
                AdcSmState = WaitLogDelay;
            }
            else
            {
                /* Stop log event */
                GenerateEvt(&CloseLogFile);
                /* Jump to Idle */
                AdcSmState = Idle;
            }
            break;

        default:
            break;
    }
}

```

Listato 4 - Implementazione della macchina a stati del task Adc.

UsbTask	10 ms
LogTask	50 ms
AdcTask	100 ms
LedTask	100 ms

Tabella 1 - Periodicità dei task.

gestione dell'ADC e del GPIO. La Fig. 12 illustra il Project Manager del progetto e mette in evidenza i quattro task che costituiscono lo strato applicativo. La tabella dei task dello scheduler, in questo caso, si presenta come nel Listato 3.

Da punto di vista delle periodicità sono stati utilizzati i valori riportati nella Tabella 1. Passiamo adesso ad un'analisi più dettagliata di alcuni task (analizzeremo AdcTask e LogTask, in quanto i due task UsbTask e LedTask effettuano più che altro operazioni di servizio), per comprendere come sono stati sviluppati, partendo dall'AdcTask. Questo task ha il compito di acquisire il canale analogico che deve essere loggato, ed inviare i relativi dati (già convertiti in formato stringa di testo) al task che si occupa della memorizzazione. Come al solito optiamo per una struttura di tipo macchina a stati finiti, la cui implementazione è riportata nel Listato 4. Come si può vedere sono stati previsti i seguenti quattro stati: Idle, WaitLogDelay, AcquireData, LogData. Inizialmente la macchina controlla se è stato ricevuto l'evento LoggerReady, che viene prodotto dal task LogTask,

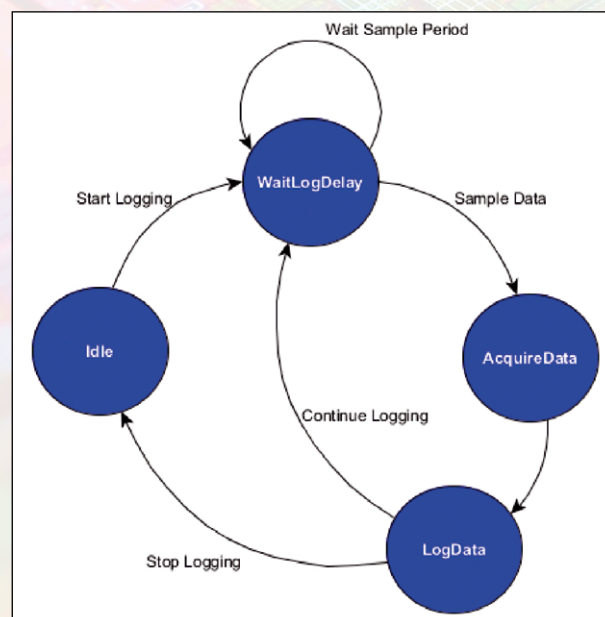


Fig. 13 - Macchina a stati AdcStateMachine.

non appena la periferica MSD è stata enumerata e correttamente inizializzata. Questo evita che vengano prodotti eventi di log prima che la memoria sia pronta a poterli memorizzare. A questo punto la macchina a stati entra in un ciclo all'interno del quale, allo scadere del periodo di campionamento il dato viene acquisito e convertito in stringa (WaitLogDelay, AcquireData). Poi, nello stato LogData, si controlla se è stato ricevuto o meno un evento di stop log; se l'evento non è stato ricevuto si invia l'evento LogDataEvt e si continua con l'acquisizione (tornando allo stato WaitLogData), altrimenti si invia l'evento CloseLogFile (per indicare al task di log di terminare e chiudere il file) e si ritorna nello stato di Idle. La rappresentazione grafica della macchina a stati relativa alla funzione AdcStateMachine è riportata in **Fig. 13**. Passiamo adesso all'esame dell'implementazione del task LogTask, che implementa tutte le operazioni relative al log dei dati sulla memoria USB. Questo task implementa due macchine a stati: la prima (LogControl) si occupa di controllare l'attività di log, generando un evento di stop dopo un certo periodo di tempo prefissato, mentre la seconda (LoggingStateMachine) si occupa di loggare i dati ricevuti dal task AdcTask sulla memoria USB. Il codice sorgente della macchina a stati LogControl è riportato nel **Listato 5**. L'implementazione è piuttosto semplice, in quanto viene semplicemente gestito un contatore per tenere traccia del tempo trascorso e generare l'evento di stop log quando il contatore stesso supera una certa soglia temporale, quindi non ci dilunghiamo e passiamo direttamente all'analisi della macchina a stati LoggingStateMachine, il cui codice è riportato nel **Listato 6**.

Questa macchina a stati sfrutta l'interfaccia di alto livello fornita dalla File System Interface per gestire direttamente la memoria. Le API utilizzate da questo task sono:

- BYTE USBHostMSDSCSIMediaDetect (void); verifica se un dispositivo MSD-SCSI è connesso al bus USB (ed è stato correttamente enumerato);

- int FSInit(void): Inizializza il file System;
- FSFILE \* FSfopen(const char \* fileName, const char \*mode); apre un file (con la modalità indicate) e ritorna un puntatore allo stream di dati;

## Listato 5

```

/*****
 * Function:      void LogControl (void)
 * Input:         None
 * Output:        UINT8
 * Author:        F.Ficili
 * Description:   Generates stop log event
 * Date:          19/04/15
 *****/
UINT8 LogControl (void)
{
    /* State machine state*/
    static SecCounterStateType SecCounterState = WaitSecDelay;
    /* Delay Counter */
    static UINT16 DelayCounter = 0;
    /* Seconds Counter */
    static UINT16 SecondsCounter = 0;

    switch (SecCounterState)
    {
        case WaitSecDelay:
            /* Increase counter */
            DelayCounter++;
            /* Check counter */
            if (DelayCounter >= SECOND_DELAY_MS)
            {
                /* Reset counter */
                DelayCounter = 0;
                /* Increase sec counter */
                SecondsCounter++;
                /* Jump to IncreaseCnt*/
                SecCounterState = IncreaseCnt;
            }
            else
            {
                /* Do nothing, satisfy MISRA */
            }
            break;

        case IncreaseCnt:
            /* Check acq timeout */
            if (SecondsCounter >= ACQ_TIMEOUT_S)
            {
                /* Generate stop log evt */
                GenerateEvt (&StopLog);
                /* Jump to Stop */
                SecCounterState = Stop;
            }
            else
            {
                /* Jump back to WaitSecDelay */
                SecCounterState = WaitSecDelay;
            }
            break;

        case Stop:
            break;

        default:
            break;
    }
}

```

Listato 5 - Implementazione della macchina a stati LogControl.

## Listato 6

```

/*****
* Function:      void LoggingStateMachine (void)
* Input:         None
* Output:        None
* Author:        F.Ficili
* Description:   Logging task state machine
* Date:          11/04/15
*****/
void LoggingStateMachine (void)
{
    /* Logging state machine state */
    static LoggingSmStateType LoggingSmState = CheckDeviceDetected;
    static MassStorageStatusType MsStatus = DeviceDetached;
    static FSFILE *LogFile;
    static UINT8 DelayCounter = 0;

    switch (LoggingSmState)
    {
        case CheckDeviceDetected:
            /* Check device status */
            if (USBHostMSDSCSIMediaDetect())
            {
                /* Device attached */
                MsStatus = DeviceAttached;
                /* Jump to CheckFsInitialized */
                LoggingSmState = CheckFsInitialized;
            }
            else
            {
                /* Device detached */
                MsStatus = DeviceDetached;
            }
            break;

        case CheckFsInitialized:
            /* If file system is initialized */
            if (FSInit())
            {
                /* Jump to OpenFile */
                LoggingSmState = OpenFile;
            }
            break;

        case OpenFile:
            /* Open file */
            LogFile = FSfopen("LogFile.txt", FS_WRITE);
            /* Jump to WaitOpening */
            LoggingSmState = WaitOpening;
            break;

        case WaitOpening:
            /* Increment counter */
            DelayCounter++;

            /* If counter expires */
            if (DelayCounter >= OPENING_FILE_DELAY_MS)
            {
                /* Ready event */
                GenerateEvt(&LoggerReady);
                /* Reset counter */
                DelayCounter = 0;
                /* Jump to WaitLogEvent */
                LoggingSmState = WaitLogEvent;
            }
            break;

        case WaitLogEvent:
            /* If a log data */
            if (ReceiveEvt(&LogDataEvt))
            {
                /* Jump to WriteLogData */
                LoggingSmState = WriteLogData;
                /* Loggin started */
                LogStatus = Logging;
            }
    }
}

```

(Continua)

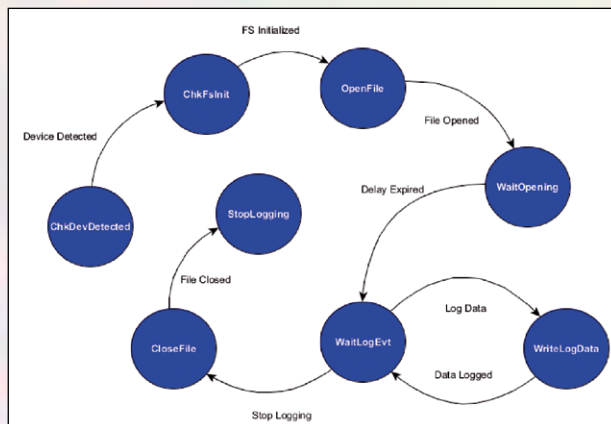


Fig. 14 - Rappresentazione grafica della macchina a stati LoggingStateMachine.

- `size_t FSfwrite(const void *data_to_write, size_t size, size_t n, FSFILE *stream);` scrive l'array di dati passato sul file puntato dallo stream;
- `int FSfclose(FSFILE *fo);` Chiude il file passato tramite il riferimento fo.

Tramite queste API di alto livello è relativamente facile scrivere un'applicazione di datalogging. Come appare dal listato, sono stati identificati 8 stati, definiti dal tipo enumerativo riportato nel **Listato 7**. I primi due stati della macchina controllano se la periferica è collegata ed è stata correttamente enumerata e se il file system è inizializzato. Successivamente viene aperto il file, utilizzando la API `FSfopen` e viene generato un ritardo per dare il tempo al client driver di eseguire l'operazione. A questo punto la macchina si mette in attesa degli eventi di log provenienti dal task `Adc`. Se viene ricevuto un evento di log si salva il dato e si torna nello stato di ascolto (`WaitLogData`) altrimenti si chiude e si va nello stato conclusivo (`StopLogging`).

Come si può notare l'implementazione (riportata in forma grafica in **Fig. 14**) è piuttosto semplice, questo perché sfruttiamo appieno le potenzialità delle infrastrutture su cui l'applicazione si appoggia (lo scheduler e lo stack USB host), che fanno la maggior parte del "lavoro sporco", permettendo all'applicazione di operare ad alto livello. Per testare la nostra applicazione è sufficiente scaricare l'applicazione sulla demoboard utilizzata a supporto del corso, collegare una chiavetta USB sulla porta USB host e dare alimentazione. L'applicazione, una volta enumerato il dispositivo USB inizierà l'attività di logging, che

## Listato 6 (segue)

```

    }
    else if (ReceiveEvt(&CloseLogFile))
    {
        /* Jump to CloseFile */
        LoggingSmState = CloseFile;
        /* Logging stopped */
        LogStatus = NotLogging;
    }
    else
    {
        /* Do nothing */
    }
    break;

case WriteLogData:
    /* Write data to USB key */
    FSfwrite(AdcDataBuffer,1,ADC_BUFFER_LENHT,LogFile);
    /* Put new line */
    FSfwrite("\r\n",1,2,LogFile);

    /* Jump back to WaitLogEvent */
    LoggingSmState = WaitLogEvent;
    break;

case CloseFile:
    /* Close file */
    FSfclose(LogFile);
    /* Jump to Stop */
    LoggingSmState = StopLogging;
    break;

case StopLogging:
    /* Do nothing */
    break;

default:
    break;
}
}

```

Listato 6 - Implementazione della macchina a stati LoggingStateMachine.

## Listato 7

```

/* Logging state machine state typedef */
typedef enum LoggingSmStateEnum
{
    CheckDeviceDetected,
    CheckFsInitialized,
    OpenFile,
    WaitOpening,
    WaitLogEvent,
    WriteLogData,
    CloseFile,
    StopLogging,
} LoggingSmStateType;

```

Listato 7 - Typedef per definire il tipo enumerativo LoggingSmStateType.

si concluderà dopo un certo periodo di tempo (circa 30 secondi). In Fig. 15 è riportato il grafico dell'acquisizione generato con Excel.

### CONCLUSIONI

In questa puntata abbiamo analizzato in dettaglio la parte host dello stack USB di microchip e costruito un'applicazione di datalogging di una certa complessità. Nella prossima ed ultima puntata analizzeremo un altro complesso e potente stack di Microchip, ossia lo stack TCP/IP.

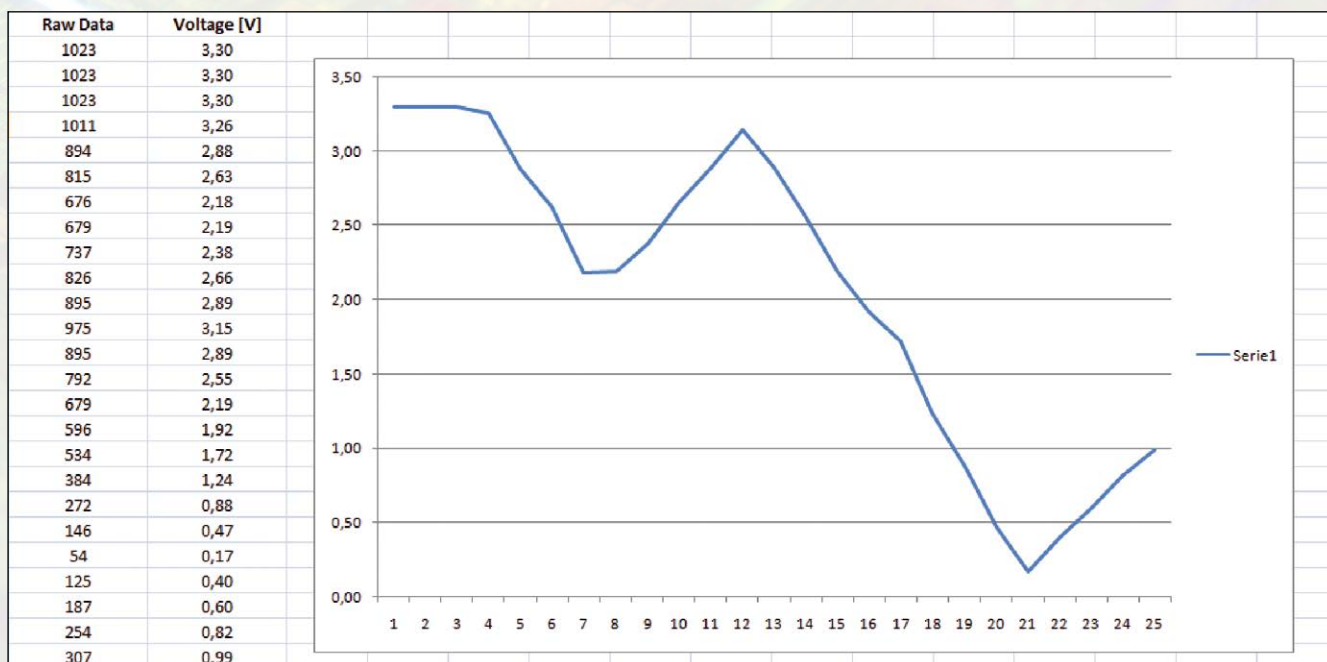


Fig. 15 - Grafico dell'acquisizione generato con MS Excel.

# CORSO

# MPLAB X

# 8

di FRANCESCO FICILI  
e VINCENZO GERMANO

**Continuiamo il nostro viaggio alla scoperta di MPLab X, il nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip per sostituire l'MPLab IDE. In queste pagine realizziamo applicazioni -come il Web Server- che impiegano lo stack TCP/IP di Microchip. Ottava puntata.**

**N**elle ultime due puntate di questo corso abbiamo focalizzato la nostra attenzione sugli stack forniti a supporto degli sviluppatori che intendono creare applicazioni con i PIC32 e con la toolchain di sviluppo open source di Microchip; in particolare, abbiamo analizzato lo stack USB, che si compone di una parte device e di una parte host. In questa ottava puntata analizziamo un altro potente e complesso stack open source di Microchip: lo stack TCP/IP. Come esempio pratico realizzeremo un semplice Web Server embedded.

## **LO STACK TCP/IP DI MICROCHIP**

Nel pacchetto Microchip Libraries for Applications (MLA), insieme allo stack USB già utilizzato e ad altri stack e librerie non trattate in questo corso, viene

fornito un completo stack TCP/IP integrabile in progetti che fanno uso di microcontrollori Microchip (PIC18, PIC24/dsPIC e PIC32). Lo stack supporta diverse configurazioni hardware, permettendo di utilizzare sia i moduli interni presenti nei PIC18 e PIC32, sia i controller esterni ENC28J60 (10Base-T) ed ENC624J600 (10/100Base-T) Microchip. La **Fig. 1** illustra le soluzioni HW supportate dallo stack, evidenziando le caratteristiche di ognuna. In questa puntata ci occuperemo della soluzione rappresentata in alto a destra nello schema di **Fig. 1**, ossia quella che sfrutta il controller interno ai PIC32 con MAC integrato (in **Fig. 2** è riportato lo schema a blocchi di un PIC32, dove è evidenziato il modulo Ethernet interno, collegato a due canali DMA). In questa configurazione non è previsto il livello fisico all'interno

## Ethernet Portfolio

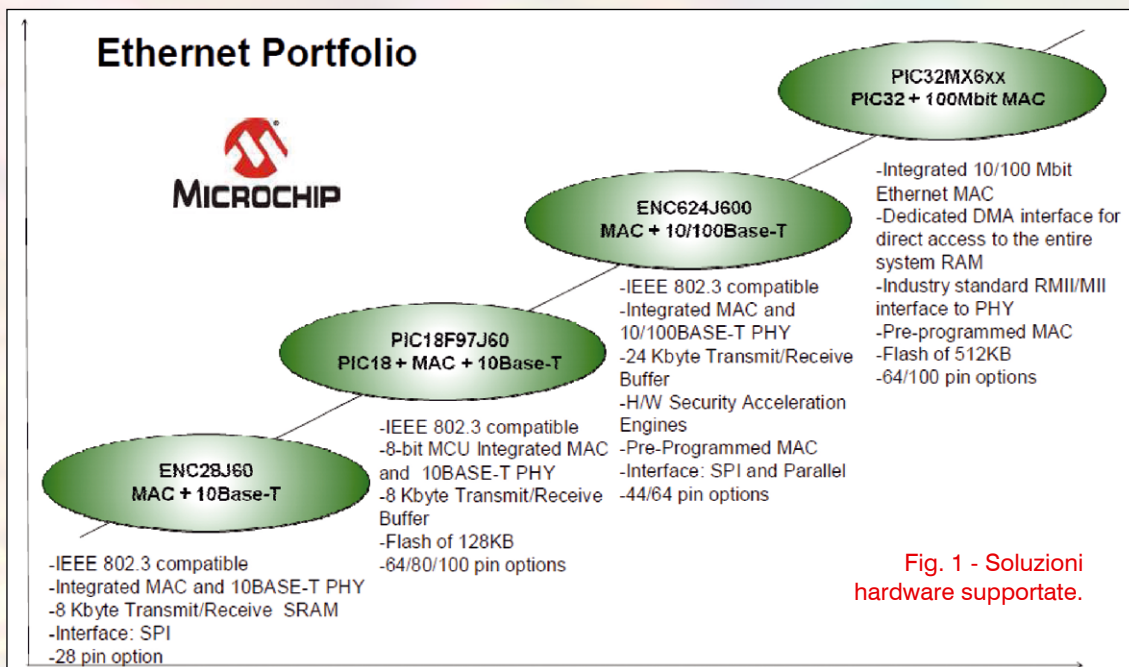


Fig. 1 - Soluzioni hardware supportate.

del micro, che dovrà quindi essere aggiunto esternamente (ad esempio, nella demoboard presentata a supporto del corso è stato utilizzato il chip National Semiconductor DP83848C, connesso al PIC32MX795F512L tramite lo standard RMII).

### ARCHITETTURA DELLO STACK TCP/IP

L'architettura dello stack TCP/IP di Microchip segue un approccio a layer, come la teoria implementativa di uno stack di comunicazione suggeri-

sce: il software è diviso in strati sovrapposti l'uno all'altro e ogni strato utilizza servizi provenienti dagli strati sottostanti e fornisce servizi agli strati superiori. Un modello semplificato dello stack TCP/IP è riportato in Fig. 3. Naturalmente, siccome l'implementazione dello stack non è pensata per un ambiente desktop, dotato di risorse virtualmente infinite e di una capacità di calcolo elevata, l'implementazione Microchip racchiude un subset dei vari livelli che possono comporre lo stack. In Fig. 4 è riportata l'implementazione Microchip, comparata con il modello di riferimento del protocollo TCP/IP. L'implementazione dello stack segue il paradigma del multitasking cooperativo e può quindi essere integrata in qualsiasi altro sistema che sfrutti il medesimo approccio, come ad esempio nello scheduler che abbiamo realizzato a supporto del corso (fascicolo n° 194) e che abbiamo già utilizzato per l'integrazione dello stack USB. Come si può vedere dallo schema, l'implementazione dello stack è modulare e ogni modulo implementa un particolare livello o porzione di livello. Nella Tabella 1 sono riportati i principali moduli che compongono lo stack con la relativa descrizione e la lista dei file necessari per includere il modulo nel progetto. Chi volesse approfondire i dettagli dello stack TCP/IP open source di Microchip può leggere un'interessantissima application note sull'argomento: l'AN833 - The Microchip TCP/IP Stack.

### CONFIGURAZIONE DELLO STACK

Come abbiamo visto in precedenza, lo stack TCP/IP di Microchip è composto da diversi moduli.

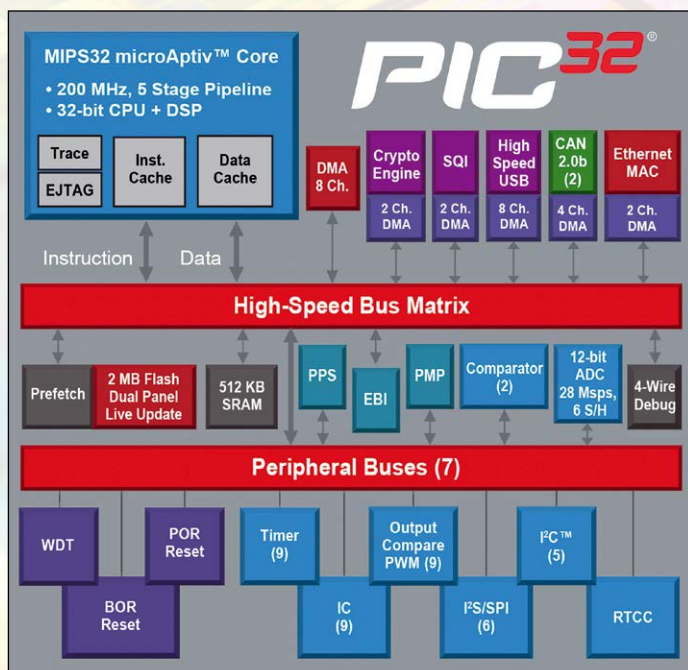


Fig. 2 - Schema a blocchi dell'architettura hardware interna di un PIC32.

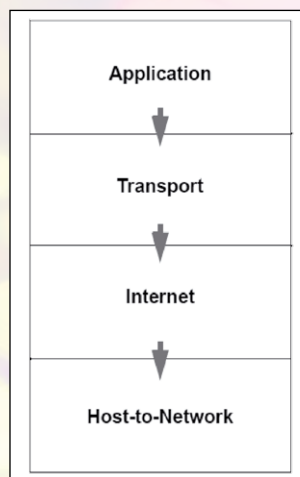


Fig. 3 - Modello semplificato dello stack TCP/IP.

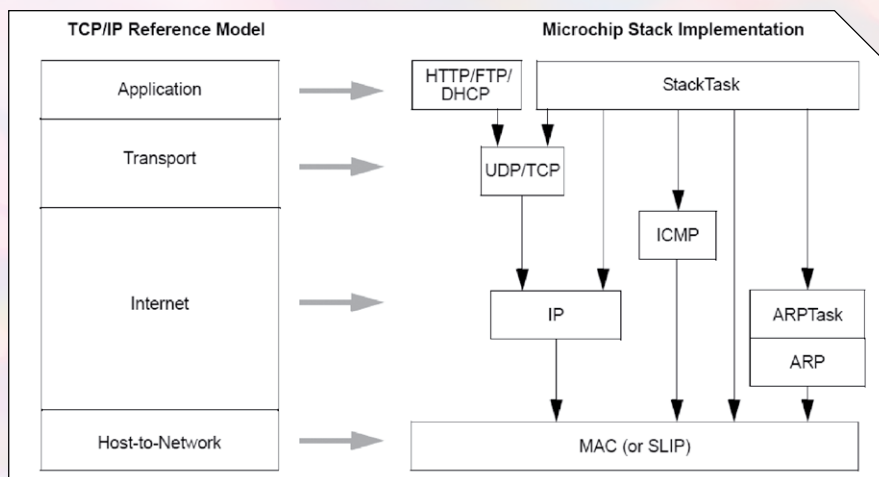


Fig. 4 - Comparazione tra l'implementazione Microchip e il modello di riferimento per il protocollo TCP/IP.

Ogni modulo, inoltre, può essere opportunamente configurato in base all'applicazione specifica e tale configurazione risiede in un particolare file, chiamato *TCPIPConfig.h*, che deve essere incluso nel progetto. L'impostazione delle configurazioni del file *TCPIPConfig.h* può essere un'operazione piuttosto complessa, ecco perché la Microchip ha realizzato un'applicazione, che prende il nome di *TCPIPConfig.exe*, che ci permette di configurare

in maniera semplice questo file attraverso un'interfaccia grafica, esattamente come avevamo fatto per lo stack USB. Vediamo quindi come funziona, utilizzandola per configurare il file *TCPIPConfig.h* dell'esempio pratico proposto in questa puntata. Per prima cosa lanciamo l'applicazione, che, una volta avviata, ci presenterà la schermata di Fig. 5, dove è possibile scegliere il percorso del file *TCPIPConfig.h* da modificare. Nel caso

## IL LINGUAGGIO HTML

Il linguaggio HTML, acronimo di HyperTextMarkup Language, è il linguaggio base utilizzato per la realizzazione delle pagine web. È un linguaggio open source, la cui sintassi è stabilita dal World Wide Web Consortium (W3C), e che è derivato da un altro linguaggio avente scopi più generici, l'SGML. È basato sull'utilizzo di Tags che permettono di formattare il testo nel modo desiderato. Non può essere considerato un linguaggio di programmazione perché non possono essere definite alcune variabili e cicli di controllo ma unicamente un linguaggio di formattazione testuale, ed è generalmente indicato come *linguaggio di markup*. Per ovviare a questa limitazione è però possibile inserire all'interno della pagina dei contenuti esterni come immagini, video o script in altri linguaggi di programmazione come ad esempio il JavaScript. I tags sono racchiusi tra parentesi angolari ed ogni volta che un tag viene aperto deve poi anche essere chiuso inserendo prima del nome del tag, la barra. Ad esempio il per inserire il testo "Testo formattato" in grassetto dovremo scrivere:

```
<b>Testo Formattato</b>
```

La struttura base di un file Html è raffigurata in Fig. A. Partendo dalla testa del file, si ha una descrizione

del documento, alcune informazioni di servizio ed il corpo del documento nel quale viene formattato l'intero contenuto della pagina.

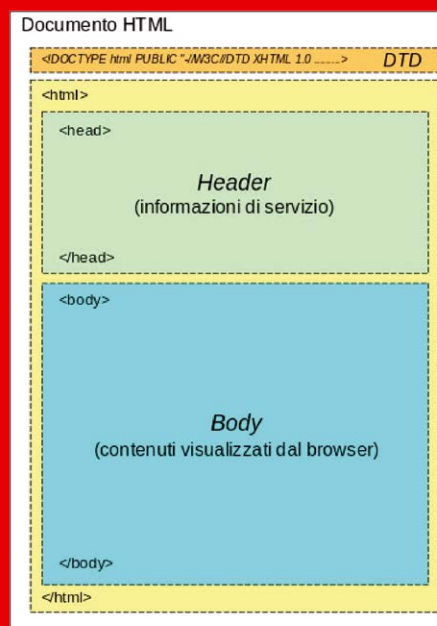


Fig. A - Schermata iniziale del tool TCPIPConfig.

Modulo	File Necessari	Descrizione
Stack Manager	StackTsk.c TCP.c IP.c ICMP.c ARPTsk.c ARP.c MAC.c or SLIP.c Tick.c Helpers.c	Stack Manager ("StackTask"), che coordina l'esecuzione di tutti gli altri moduli dello stack.
MAC	MAC.c Delay.c	Media Access Layer.
SLIP	SLIP.c	Media Access Layer per il protocollo SLIP.
ARP	ARP.c ARPTsk.c MAC.c or SLIP.c Helpers.c	Address Resolution Protocol.
IP	IP.c MAC.c or SLIP.c Helpers.c	Internet Protocol.
ICMP	ICMP.c StackTsk.c IP.c MAC.c or SLIP.c Helpers.c	Internet Control Message Protocol.
TCP	StackTsk.c TCP.c IP.c MAC.c or SLIP.c Helpers.c Tick.c	Transmission Control Protocol.
UDP	StackTsk.c UDP.c IP.c MAC.c or SLIP.c Helpers.c	User Datagram Protocol.
HTTP Server	HTTP.c TCP.c IP.c MAC.c or SLIP.c Helpers.c Tick.c MPFS.c	HyperText Transfer Protocol Server.
FPT Server	FTP.c TCP.c IP.c MAC.c or SLIP.c	File Transfer Protocol Server.
DHCP	DHCP.c UDP.c IP.c MAC.c Helpers.c Tick.c	Dynamic Host Configuration Protocol.

**Tabella 1 - Elenco dei moduli che compongono lo stack TCP/IP della Microchip.**

si utilizzi anche il WiFi, si deve inserire anche il percorso del file *WF\_Config.h*, che contiene le opzioni di configurazione del modulo WiFi. Nella schermata successiva, visibile in **Fig. 6**, è possibile selezionare i moduli base da includere nell'applicazione. Nel nostro caso vogliamo realizzare un Web Server, quindi spuntiamo semplicemente la casella "Web Sever", sotto "Application Protocols" e deseleggiamo il resto. Andando ancora avanti troveremo la schermata che ci permette di

selezionare alcuni moduli di esempio. Nel nostro caso non ci interessa averne, quindi deseleggiamo tutto e procediamo.

La schermata successiva è riportata in **Fig. 7**, e ci permette di selezionare i moduli di supporto.

Selezioniamo i seguenti moduli:

- DHCP Client e Server;
- ICMP Client;
- DNS Client;
- NetBIOS Name Server;
- AutoIP Client.

e andiamo avanti. Nella schermata successiva possiamo impostare un po' di indirizzi della scheda. Lasciamo pure inalterati il MAC e l'indirizzo IP, come anche la SubNet Mask, l'indirizzo del gateway e dei server DNS, mentre invece modifichiamo il nome dell'Host, usando quello che preferiamo (nell'esempio abbiamo usato il nome "EINBOARD"). Si consideri che, avendo incluso il modulo DHCP, i valori inseriti per IP Address, Subnet Mask, Gateway Address, e DNS Server, sono solo valori predefiniti, dato che poi ci penserà il modulo DHCP ad inserire dei valori dinamici quando collegheremo la scheda. A questo punto bisogna configurare il Web Server, operazione che possiamo eseguire nelle successive due schermate. Per prima cosa, nella schermata di **Fig. 9**, inseriamo il nome della pagina di default, che è quella che conterrà la schermata iniziale del nostro web server. Poi inseriamo il numero massimo di socket simultanei che possono essere mantenuti (nel nostro esempio 2). Nella sezione "Additional Features" e "Demo Applications" deseleggiamo tutto. Nella schermata successiva (**Fig. 10**) bisogna solo impostare dove verrà memorizzata la pagina (o le pagine) web del Web Server. Ci sono diverse opzioni; in questo caso scegliamo la più semplice, ossia "Internal Program Memory". Con questa opzione la pagina web viene caricata direttamente nella memoria Flash del PIC32 che stiamo utilizzando, e vedremo dopo come questa operazione viene eseguita. Non sempre è possibile seguire questa strada, dato che se le pagine web sono piuttosto pesanti, la Flash del microcontrollore potrebbe non riuscire a contenerle; in questo caso ci sono altre possibilità, come usare memorie EEPROM o Flash esterne o addirittura un supporto FAT (come ad esempio una SD card o una memoria USB). Nella schermata di **Fig. 11** è mostrata la configurazione dei socket TCP; potete lasciare i valori predefiniti,

ma se volete modificare qualcosa consultate -per saperne di più- la sezione relativa al modulo TCP sulla Application Note 833. Una volta completata anche la configurazione dei socket, verrà visualizzata la schermata che permette di completare la configurazione.

### MICROCHIP FILE SYSTEM (MPFS)

Come abbiamo visto in precedenza, l'implementazione di Microchip dello stack TCP/IP prevede anche l'implementazione di un Web Server HTTP. Tuttavia, se viene implementato un Web Server ci saranno sicuramente una o più pagine web da visualizzare, che devono essere immagazzinate in una qualche memoria. Nell'im-

mentazione Microchip del Web Server embedded viene utilizzato un semplice tipo di file system (Microchip File System o MPFS) per immagazzinare le pagine web da pubblicare. Come abbiamo già visto prima, tali pagine possono essere memorizzate nella memoria Flash del micro stesso oppure su un supporto esterno (EEPROM esterna, SPI Flash o anche una memoria SD o USB esterna). Il formato di un'immagine MPFS è riportato in Fig. 12: dopo un blocco iniziale riservato, la cui dimensione è fissata dalla macro MPFS\_RESERVE\_BLOCK, l'immagine contiene una serie di FAT (File Allocation Table) entry, seguite da altrettanti blocchi di dati. Ogni FAT entry ha il formato riportato in Fig. 13. Come si

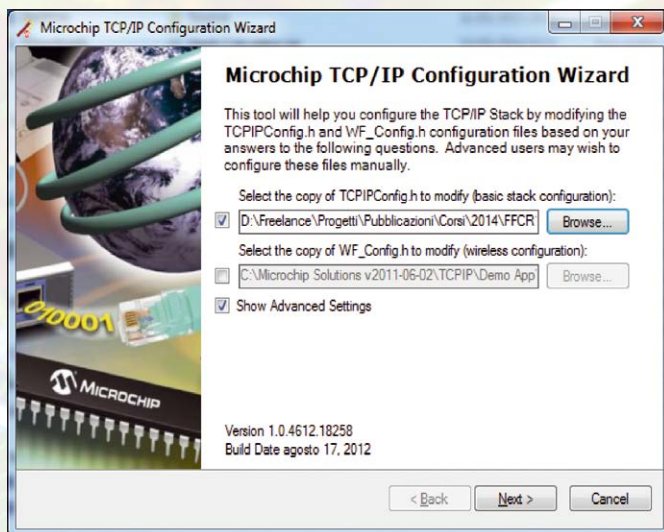


Fig. 5 - Schermata iniziale del tool TCPIPConfig.

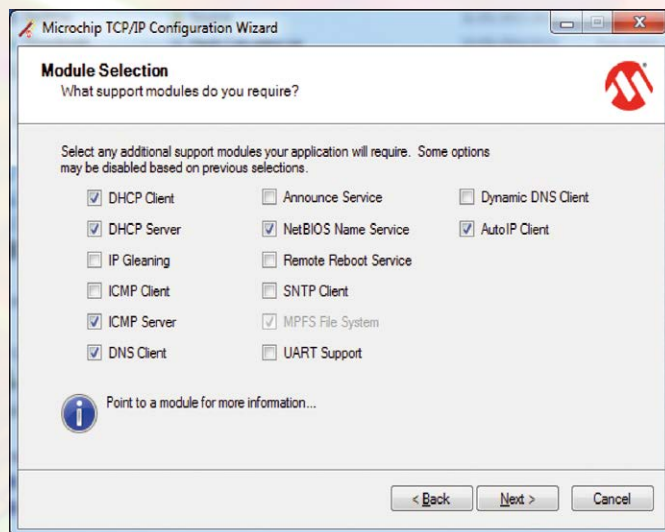


Fig. 7 - Selezione dei moduli di supporto da includere nell'applicazione.

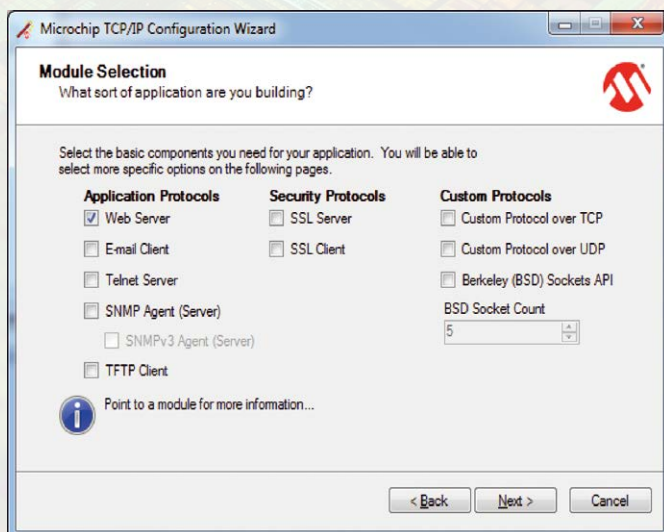


Fig. 6 - Selezione dei moduli base da includere nell'applicazione.

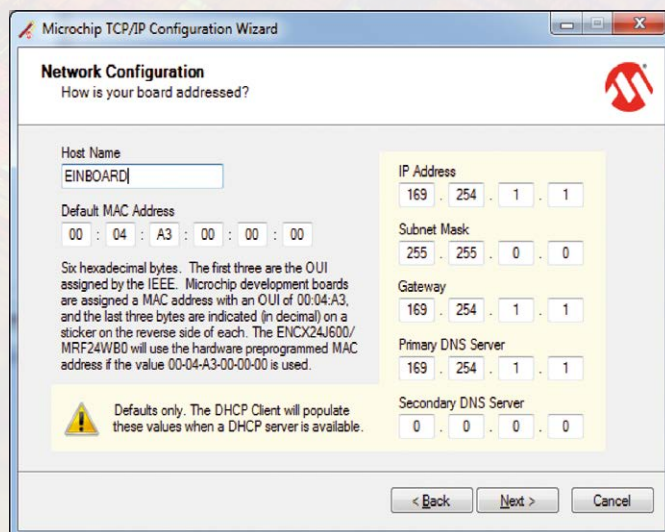


Fig. 8 - Impostazione degli indirizzi.

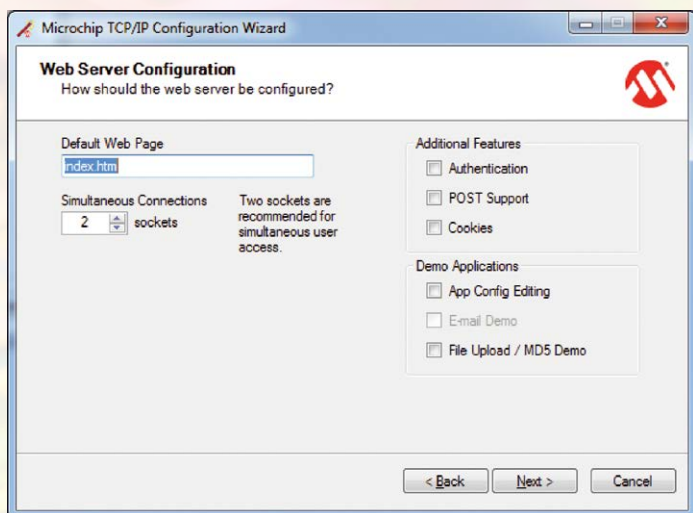


Fig. 9 - Configurazione del WebServer.

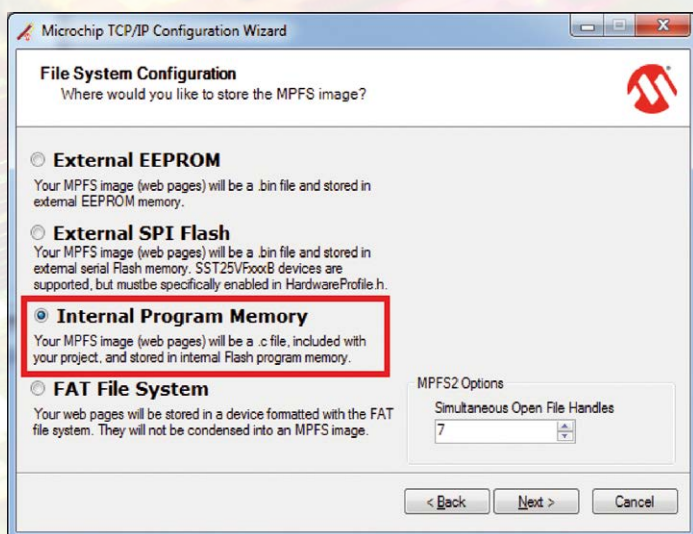


Fig. 10 - Opzioni di store delle pagine web del WebServer.

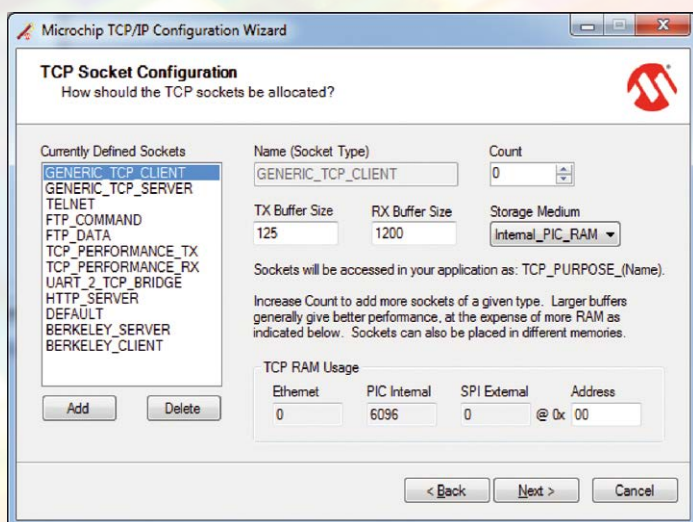


Fig. 11 - Configurazione dei sockets.

può facilmente notare, l'insieme delle FAT entry costituisce una tabella che associa un indirizzo al nome del blocco dati. L'indirizzo può essere a 16 o 24-bit in funzione del supporto utilizzato per la memorizzazione. Ad esempio, se le pagine vengono immagazzinate nella memoria Flash e per compilare un progetto è utilizzato il modello di memoria SMALL, allora gli indirizzi sono a 16-bit. Per quanto riguarda -invece- il file name, è utilizzato il nome di tipo "short", nel formato "8 + 3". L'indirizzo di ogni FAT entry punta al relativo blocco che contiene i dati binari della pagina, il cui formato è riportato in Fig. 14. La generazione di un'immagine di tipo MPFS è indispensabile per la realizzazione di un'applicazione che utilizzi il Web Server (quindi come quella del nostro esempio pratico). Per semplificare il processo, Microchip ha introdotto un'applicazione desktop (chiamata MPF2) che, partendo da una normale pagina web realizzata in HTML, genera la corrispondente immagine MPFS da utilizzare nel nostro progetto embedded. L'applicazione, la cui schermata è riportata in Fig. 15, permette di generare immagini binarie (da immagazzinare su supporti esterni), file .c (nel caso di memorizzazione nella Flash del micro stesso), come anche codice per file system di tipo FAT (SD card o chiavetta USB). Nella nostra applicazione di esempio faremo uso di questo tool per la conversione della nostra pagina web.

## VARIABILI DINAMICHE

Un'applicazione che implementa un web server all'interno del nostro PIC32 sarebbe piuttosto inutile se non fosse in grado di interagire in qualche modo con il resto del sistema per scambiare dati. Microchip ha pensato anche a questo e per risolvere il problema ha introdotto il concetto di variabile dinamica. Tali variabili permettono una interazione tra modulo Web Server e il resto dell'applicazione che gira sul microcontrollore, permettendo di avere delle variabili condivise tra il modulo stesso e il resto del codice. Un esempio dell'utilità di questa soluzione è schematizzato in Fig. 16: in questo schema è stata realizzata un'applicazione che al proprio interno implementa un Web Server e che è anche collegata ad un particolare sensore. Utilizzando il concetto di variabile dinamica, le letture del sensore sono combinate all'interno del Web Server in modo da essere visualizzate sulla pagina web e poi venire trasmesse via Internet, da cui sono disponibili per ogni client che ne faccia

richiesta. Le variabili dinamiche possono essere utilizzate, in combinazione con il Web Server, in modi differenti. Noi ne esamineremo due, che sono i due casi utilizzati nel nostro esempio pratico: visualizzazione di dati sulla pagina web ed uso del metodo GET. Nel primo caso abbiamo la possibilità di visualizzare sulla nostra pagina una variabile di sistema (ossia un generica variabile presente nel nostro progetto embedded). Per farlo sono necessari due step: dal lato pagina web, la variabile da visualizzare va racchiusa tra due simboli tilde (~), mentre dal lato embedded deve essere implementata un'opportuna callback, il cui prototipo viene generato dal tool MPFS2 quando la pagina web viene tradotta in immagine binaria. Ad esempio, se volessimo visualizzare una generica variabile "var" presente nel nostro progetto embedded, sulla pagina web non dovremmo fare altro che inserire questa variabile tra i simboli tilde:

```
~var~
```

Poi, dal lato embedded dovremmo implementare la seguente callback:

```
HTTPPrint_var()
```

Al resto penserà lo stack. Vedremo più chiaramente questa procedura nell'esempio pratico. Nel secondo caso, invece, l'esecuzione di un metodo GET lato web fa sì che tutti i dati passati con il metodo vengano accumulati sul buffer *HTTPcurr.data* e che venga chiamata la callback *HTTPExecuteGet()*. Combinando questi due eventi è possibile gestire l'occorrenza di una GET in maniera opportuna lato embedded. Anche questo aspetto risulterà più chiaro con la descrizione del progetto pratico.

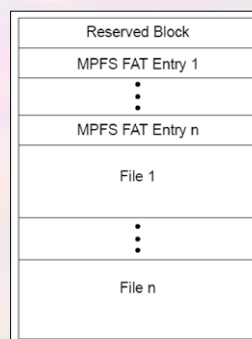


Fig. 12 - Formato di una immagine MPFS.

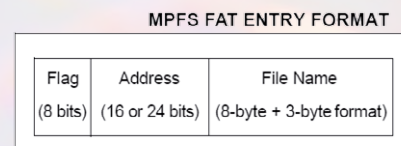


Fig. 13 - Formato della FAT Entry.

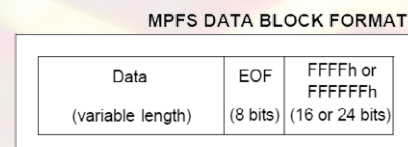


Fig. 14 - Formato del data block.

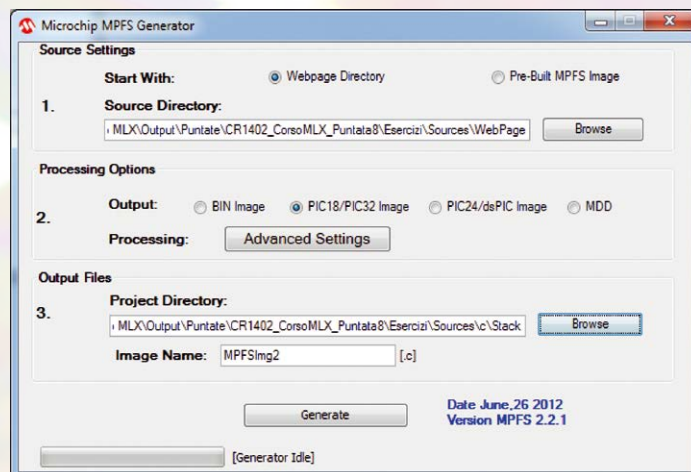


Fig. 15 - Schermata principale dell'applicazione MPFS2.jar.

## ESEMPIO PRATICO: WEB SERVER EMBEDDED

Passiamo quindi alla parte pratica, nella quale ci proponiamo di realizzare un semplice Web Server embedded, che ci permetta di controllare da remoto un LED. L'applicazione in sé è molto semplice, ma ci permette di validare in maniera pratica quanto è stato presentato nel resto della puntata. Partiamo quindi dalla solita impostazione architetturale, il cui schema è visibile in Fig. 17. Come si può vedere, in questo



Fig. 16 - Esempio d'uso di una variabile dinamica.

## Listato 1

```

/* Maximum number of task */
#define ACTIVE_TASK_NUMBER                ((UINT16) (2))

/* Array of task to call */
void (*TaskArray[ACTIVE_TASK_NUMBER]) (void) =
{
    LedTask,
    TcpIpTask,
};

/* Array of task period timeouts */
UINT16 TaskPeriodTimeoutMs[ACTIVE_TASK_NUMBER] =
{
    LED_TASK_PERIOD_MS,
    TCPIP_TASK_PERIOD_MS,
};

```

Listato 1 - Strutture dati.

## Listato 2

```

/*****
* Function:      TcpIpTask
* Input:         None
* Output:        None
* Author:        F.Ficili
* Description:   Manage TCPIP Task.
* Date:          31/05/15
*****/
void TcpIpTask (void)
{
    switch (SystemState)
    {
        /* System Initialization Phase */
        case InitializationState:
            /* Initialize tcpip stack */
            TcpIpInitStack();
            break;

        /* System Normal operation Phase */
        case RunningState:
            /* Call tcpip tasks */
            TcpIpStackTasks();
            break;

        /* Default */
        default:
            break;
    }
}

```

Listato 2 - Implementazione del TcpIpTask

caso utilizziamo solo due task per implementare la nostra applicazione: un task dedicato al TCP/IP e il solito "Led task" che ci permette di avere un minimo di feedback dalla nostra applicazione (nel nostro caso è sempre un lampeggio del LED alla frequenza di 1 Hz). Chiaramente il TcpIpTask si appoggia sullo stack TCP/IP e sull'implementazione del web server HTTP di Microchip.

In Fig. 18 è invece riportato il project manager di MPLab X relativo al progetto pratico (che abbiamo chiamato MLX\_Ex11), mentre la Tabella 2 riassume, come di consueto, le periodicità dei due task in esecuzione.

L'infrastruttura utilizzata è sempre la stessa,

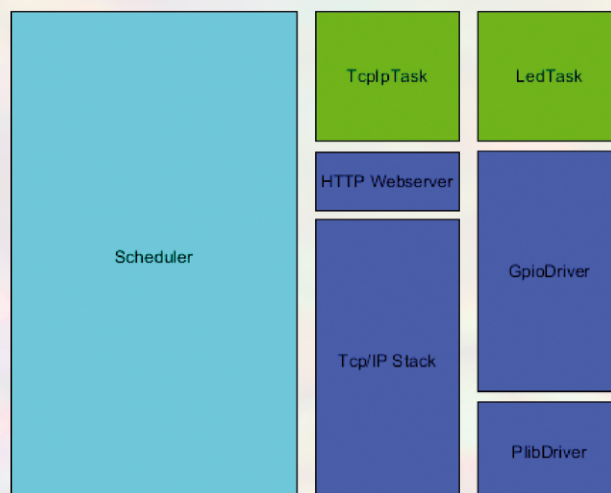


Fig. 17 - Architettura Software.

così come le definizioni delle strutture dati che supportano l'esecuzione dello scheduler, riportate nel Listato 1. Spostiamo ora la nostra analisi sul TcpIpTask, che è il task principale di questo esempio e vediamo come è stato implementato, analizzando il Listato 2: il task si compone di una sezione di inizializzazione, che serve per inizializzare correttamente le strutture dati dei vari task dello stack utilizzati in questa applica-

Task	Periodicità [ms]
LedTask	100
TcpIpTask	10

Tabella 2 - Periodicità dei task

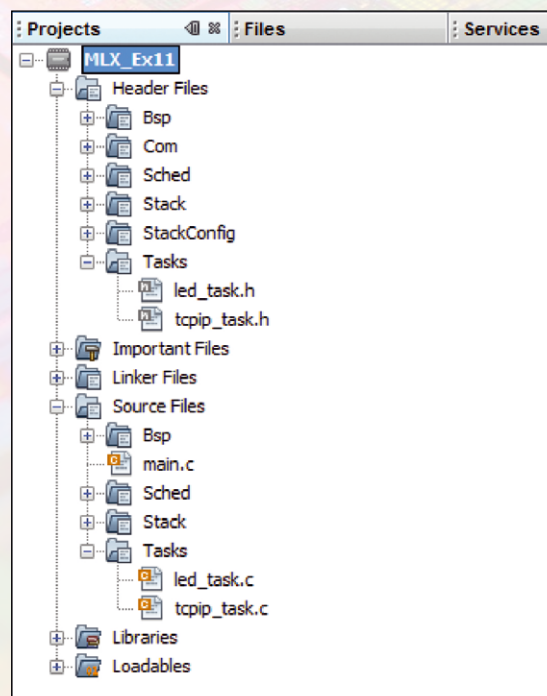


Fig. 18 - Project Manager MPLab X.

zione, e di una sezione "running" che fa girare le macchine a stati dello stack stesso. Analizziamo la struttura della funzione di inizializzazione, il cui codice è riportato all'interno del **Listato 3**. In questa funzione vengono inizializzati in sequenza:

- il componente Tick, che serve a generare tutte le necessarie tempistiche per l'intero stack;
- il componente MPFS, che serve per l'implementazione delle pagine web del server;
- la struttura dati AppConfig.

Vengono inoltre operate le inizializzazioni più generali dello stack; in particolare, l'inizializzazione della struttura dati AppConfig serve a caricare la maggior parte dei dati impostati tramite il tool di configurazione, come ad esempio i valori degli indirizzi, il NetBios Name, ecc...

Passiamo ora all'esame della parte operativa del task, il cui codice è riportato all'interno del **Listato 4**. Sono due i task chiamati in questo caso; il primo, ossia StackTasks, manda in esecuzione tutte le macchine a stati principali dello stack. Si tratta di una serie di servizi generali e temporizzazioni che sono necessari a qualsiasi task di livello applicazione dello stack. Invece il task StackApplications manda in esecuzione una serie di task di livello applicativo e dipende fortemente dalla configurazione dello stack, contenuta in *TCPIPConfig.h*. Nel caso specifico, ad esempio, sono eseguiti i task relativi al web server HTTP, al DHCP ed al NetBIOS Name Server.

Ora il nostro stack TCP/IP è correttamente inizializzato e sta girando, tuttavia se avviamo l'applicazione il nostro Web Server non permetterà di visualizzare nulla, dato che non abbiamo caricato nessuna pagina al suo interno. Per consentire la visualizzazione di una pagina web occorre creare un'immagine MPFS adatta ad essere ospitata sul Web Server http, come è stato descritto precedentemente. La pagina che intendiamo caricare è una normale pagina HTML, il cui codice è riportato nel **Listato 5**.

Analizziamo brevemente il codice HTML della pagina; la sezione iniziale comprende solo testo racchiuso tra differenti headings tag (h1, h2 e h3) e quindi è solo una porzione di HTML che visualizzerà del testo statico (in particolare la scritta "Welcome!" e alcune righe di descrizione). La sezione successiva fa uso del concetto di variabile dinamica per la visualizzazione del NetBios Name associato alla scheda (come abbiamo visto in precedenza, la variabile ~hname~ invocherà

### Listato 3

```

/*****
 * Function:      TcpIpInitStack
 * Input:         None
 * Output:        None
 * Author:        F.Ficili
 * Description:   Initialize the TCPIP stack.
 * Date:          31/05/15
 *****/
void TcpIpInitStack (void)
{
    /* Init tick counts */
    TickInit();
    /* Init MPFS */
    MPFSInit();
    /* Init AppConfig */
    InitAppConfig();
    /* Init stack */
    StackInit();
}

```

Listato 3 - Implementazione della funzione TcpIpInitStack.

### Listato 4

```

/*****
 * Function:      TcpIpStackTasks
 * Input:         None
 * Output:        None
 * Author:        F.Ficili
 * Description:   Manage the TCPIP necessary tasks.
 * Date:          31/05/15
 *****/
void TcpIpStackTasks (void)
{
    /* Perform Stack Tasks */
    StackTask();

    /* This tasks invokes each of the core stack application tasks */
    StackApplications();
}

```

Listato 4 - Implementazione della funzione TcpIpStackTasks.

### Listato 5

```

<h1>Welcome!</h1>
<h1>This is a test webpage!!!</h1>
<h3>This test webpage prove the capability _
of Microchip free TCP/IP stack</h3>

<div id="location">
    Machine NetBios Hostname: ~hname~
</div>

<br>

<form>

    <div id="led_control">
        LED Control:

    </div>

    <input type="radio" name="led"
        value="on"/> On
    <input type="radio" name="led"
        value="off"/> Off

    <br>
    <input type="submit" class="btn"
        value="Set"/>

</form>

```

Listato 5 - Codice HTML della pagina web da caricare sul Web Server.

## Listato 6

```

/*****
* Function:      HTTPPrint_hname
* Input:         None
* Output:        None
* Author:        F.Ficili
* Description:   Print hostname callback.
* Date:          31/05/15
*****/
void HTTPPrint_hname(void)
{
    TCPPutString(sktHTTP, AppConfig.NetBIOSName);
}

```

Listato 6 - Implementazione della callback HTTPPrint\_hname.

## Listato 7

```

/*****
* Function:      HTTPExecuteGet
* Input:         None
* Output:        None
* Author:        F.Ficili
* Description:   Execute get callback.
* Date:          31/05/15
*****/
HTTP_IO_RESULT HTTPExecuteGet(void)
{
    BYTE *DataPtr;

    /* Get data */
    DataPtr = HTTPGetROMArg(curHTTP.data, (ROM BYTE *) "led");

    /* Check if data is present inside the pointer */
    if (DataPtr)
    {
        /* If "on" data has been passed */
        if (strcmppgm2ram((char *) DataPtr, (ROM char *) "on")
        == 0)
        {
            /* Send on evt */
            GenerateEvt(&Led2On);
        }
        else
        {
            /* Send off evt */
            GenerateEvt(&Led2Off);
        }
    }
}

```

Listato 7 - Implementazione della callback HTTPExecuteGet.

la callback HTTPPrint\_hname, tramite la quale sarà possibile, usando una semplice funzione, far visualizzare al posto della variabile stessa il NetBios Name). Infine, la sezione racchiusa tra i tag <form> e </form> fa uso del metodo GET per generare un input per il web server che permetta di controllare un LED sulla scheda (per permettere di intercettare i dati passati dal metodo GET è necessario utilizzare la callback HTTPExecuteGet). Dal punto di vista grafico, questa sezione genera due radio button contrassegnati rispettivamente con On e Off ed un pulsante (Set). Prima di passare alla realizzazione delle callback, vediamo come generare il codice eseguibile della

pagina web, in maniera che possa essere integrato nel progetto MPLab X. Come abbiamo visto in precedenza, il codice della pagina si genera utilizzando il tool MPFS2. Possiamo sia utilizzare l'interfaccia grafica del tool, o, in alternativa, utilizzare uno script batch, come quello illustrato qui di seguito:

```

java -jar "..\Utilities\MPFS2.jar" /mpfs2 /C18_C32
"\WebPage" "." "MPFSImg2.c"

```

che invoca il tool MPFS2, passando i corretti parametri per la generazione del codice C corrispondente alla pagina.

Il risultato dell'esecuzione dello script è costituito dai seguenti due file:

- MPFSImg2.c = costituisce il codice C che implementa la pagina web da visualizzare;
- HTTPPrint.h = è l'header file che contiene i prototipi delle callback da includere nel progetto.

Entrambi i file vanno inclusi nel progetto MPLab X. La pagina web, come da configurazione impostata tramite il tool TCPConfig, verrà compilata in modo da risiedere direttamente nella memoria Flash del PIC32. Abbiamo quasi finito: non ci resta che implementare le callbacks relative alla variabile dinamica hname ed al metodo GET utilizzato sulla pagina web e compilare il nostro progetto. Iniziamo con la callback per la visualizzazione del NetBios Name. Come abbiamo visto in precedenza, la presenza della variabile dinamica ~hname~ nella pagina web provocherà, al caricamento della pagina, l'invocazione della callback HTTPPrint\_hname(); non dobbiamo fare altro che implementare questa callback in modo che visualizzi il NetBios Name della scheda, una volta invocata. Il NetBios Name (il nome dell'host che abbiamo dato in fase di configurazione, nel nostro esempio EINBOARD) è facilmente recuperabile dalla struttura dati AppConfig ed il codice che implementa la callback è riportato nel Listato 6. In questo caso lo stack ci viene molto in aiuto, fornendoci la funzione TCPPutString, che scrive sul socket passato come primo parametro i dati presenti nell'array passato come secondo parametro; ci basta quindi passare alla funzione il socket http (sktHTTP) e i dati da visualizzare (AppConfig.NetBIOSName): molto semplice, come vedete...

Invece per quanto riguarda il controllo del LED, per intercettare il flusso di dati richiesti al server

tramite il metodo GET, lo stack ci fornisce la callback HTTPExecuteGet, che viene appunto invocata all'esecuzione di questo metodo. Inoltre i dati passati vengono automaticamente caricati sul buffer currHTTP.data. Dato che la callback è unica per tutte le istanze del metodo, occorrerà esaminare i dati e, tramite un'opportuna struttura di selezione, effettuare le azioni specifiche. Le implementazioni possibili, nel caso specifico, sono molteplici; noi abbiamo fornito quella illustrata nel **Listato 7**. Per prima cosa, attraverso la funzione HTTPGetROMArg, vengono recuperati i dati associati all'argomento passato come parametro (in questo caso la stringa "led"), ed il risultato viene passato ad un opportuno puntatore. Poi, per prima cosa si controlla se il puntatore ha dei dati validi, e quindi si verifica di fatto che l'operazione precedente abbia fornito un "match"; in caso affermativo, effettuando una comparazione tra la stringa dati e la stringa "on", si determina se il LED va acceso o spento. Si noti che l'effettiva accensione (o lo spegnimento) del LED non viene eseguita direttamente nella callback, ma l'operazione viene delegata al LedTask, tramite l'invio di un opportuno evento (**GenerateEvt(&Led2On)** e **GenerateEvt(&Led2Off)**). Lo snippet di codice del LedTask che intercetta l'evento è riportato all'interno del **Listato 8**. L'operazione potrebbe sembrare un pochino macchinosa, ma delega correttamente i compiti al task specifico, permettendoci di mantenere la corretta modularità nell'implementazione. Se un domani l'effetto del clic sul pulsante "Set" dovesse essere cambiato dal semplice on/off a un'operazione più complessa, come ad esempio un lampeggio, il LedTask sarebbe sicuramente il task adatto cui far compiere questa operazione, che non potrebbe essere implementata correttamente dalla HTTPExecuteGet, il che ci obbligherebbe a modificare lo snippet o aggiungere una funzione specifica.

La nostra applicazione a questo punto è completa; non ci resta che testarla, ad esempio, usando la demoboard a supporto del corso. Dopo aver effettuato il download del codice, aver dato potenza ed aver collegato il cavo ethernet al nostro PC o al nostro access point, dobbiamo semplicemente collegarci, con un qualsiasi browser, all'indirizzo `http://einboard`, e verrà visualizzata la pagina mostrata in **Fig. 19**. Come possiamo vedere, oltre alla stringa statica, la pagina fornisce il NetBIOS name associato alla scheda: EINBOARD. Per controllare invece il LED è sufficiente selezionare

## Listato 8

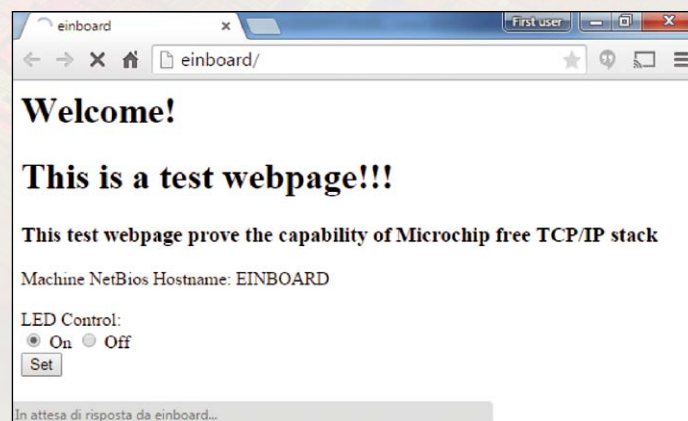
```
/* If a Led on evt i received */
if (ReceiveEvt(&Led2On))
{
    /* Turn on LED */
    LED_2 = LED_ON;
}
else if (ReceiveEvt(&Led2Off))
{
    /* Turn off LED */
    LED_2 = LED_OFF;
}
else
{
    /* Do nothing */
}
```

**Listato 8 - Snippet del LedTask che intercetta gli eventi di accensione/spegnimento del LED 2.**

il radio button corrispondente all'azione che si desidera eseguire (LED on o off) e fare clic sul pulsante "Set". Il LED si accenderà o si spegnerà di conseguenza. Ricordiamo che, avendo incluso il modulo DHCP non sarà necessario modificare gli attributi della scheda di rete.

## CONCLUSIONI

In questa ottava puntata abbiamo analizzato lo stack TCP/IP open source della Microchip, fornito a supporto dell'ambiente di sviluppo MPLab X, spingendoci con questo potente strumento ad una implementazione di una certa complessità: la realizzazione di un Web Server embedded. Questa applicazione evidenzia quanto potente e versatile sia l'ambiente di sviluppo, la piattaforma utilizzata (ossia i microcontrollori PIC32) e gli stack e le infrastrutture software fornite a supporto. A chi fosse interessato ad approfondire tematiche di questo tipo ricordiamo che a breve verrà realizzato un apposito corso da Futura Academy ([www.futura.academy](http://www.futura.academy)).



**Fig. 19 - Pagina web visualizzata accedendo al Web Server con chrome.**

# CORSO MPLABX



di FRANCESCO FICILI  
e VINCENZO GERMANO

**Ecco la DemoboardPIC32 annunciata durante il corso conclusosi nella scorsa puntata pensata, progettata e realizzata appositamente per mettere in pratica le esercitazioni proposte negli scorsi fascicoli.**

**U**na scheda di sviluppo di un microcontrollore, meglio nota "evaluation board" o "demoboard", è un circuito stampato che ospita un componente da testare più eventuali elementi a corredo per valutarne le caratteristiche e prototipare applicazioni. Nel caso dei microcontrollori, la demoboard contiene il micro e la logica di supporto necessaria per consentire all'utente di poter scoprire le sue funzionalità e imparare a programmarlo. Perciò la ragione principale per l'esistenza di una scheda di sviluppo è esclusivamente quella di fornire un sistema per imparare a usare un nuovo microcontrollore, infatti contiene poco o nessun hardware dedicato come interfaccia utente e tutto il superfluo viene eliminato, anche per contenere i costi. General-

mente non viene fornito alcun contenitore di protezione, per permettere all'utilizzatore di poter accedere in qualsiasi momento ad ogni sua parte; né tantomeno un alimentatore, perché molto spesso la si usa in un ambiente di laboratorio in cui sono presenti tutti gli strumenti per metterla in condizione di funzionare.

Le demoboard sono anche molto utili nella fase prototipale di un progetto, perché danno la possibilità di testare l'hardware e il software prima che sia disponibile una board dedicata, riducendo al minimo l'utilizzo dei simulatori interni all'ambiente di sviluppo, che simulano l'hardware selezionato. Un esempio è riportato in Fig. 1, nella quale si può vedere una scheda di sviluppo della Microchip Explorer 16.

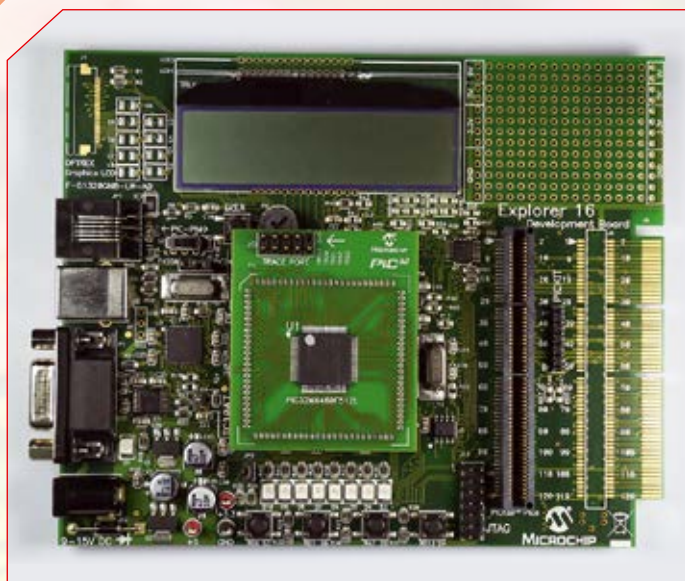


Fig. 1 - Esempio di Evaluation Board della Microchip: Explorer 16.

Lo sviluppo di una scheda elettronica parte dalla sua progettazione, che può essere più o meno complicata, ma in ogni caso deve prevedere l'uso di strumenti ECAD (Electronic Computer Aided Design) o EDA (Electronic Design Automation) più o meno sofisticati. Oltre a tools estremamente professionali e costosi, esistono anche diversi strumenti di progettazione open source, che si possono dimostrare perfettamente adeguati sia alle esigenze dell'hobbista che vuole cimentarsi nella progettazione elettronica, sia a quelle della piccola azienda con la necessità di progettazione di schede elettroniche per uso interno o per i suoi prodotti. Nella lista degli strumenti EDA open source spicca KiCad, una Electronic Design Automation Suite completamente open source e multi piattaforma, sviluppata e distribuita sotto licenza GNU e sfruttata per la progettazione della DemoboardPIC32 di questo corso.

Durante le precedenti puntate del corso, abbiamo avuto la possibilità di poter analizzare tutte le potenzialità del nuovo ambiente di sviluppo integrato prodotto e distribuito da Microchip Technology, MPLab X. Siamo passati dalla fase di download e d'installazione dell'applicazione, alla fase di presentazione e studio delle periferiche relative ai PIC32 mediante esempi pratici, fino all'analisi delle MLA (*Microchip Libraries for Applications*), molto utili anche per tutte le applicazioni più complesse come USB ed Ethernet. Avendo concluso con la precedente puntata la trattazione software, possiamo ad approfondire l'hardware presentando la

scheda di sviluppo utilizzata durante il corso: la *DemoboardPIC32*.

## IL PROGETTO

La "DemoboardPIC32" è stata pensata, progettata e realizzata in modo da poter essere utilizzata come starter kit e quindi come strumento di supporto hardware per questo corso, infatti prevede tutte le periferiche base dei microcontrollori PIC32, più alcune periferiche avanzate come ad esempio USB e Ethernet. Quando tale progetto è stato concepito, si è voluto realizzare una scheda elettronica che desse la possibilità agli utenti di avere un'unica board di sviluppo a supporto del corso presentato ma che al tempo stesso non fosse riduttiva nel suo utilizzo e potesse essere usata come punto di partenza di nuove applicazioni, tutte incentrate sui PIC32. Con tale filosofia si è pensato a una board con un diagramma a blocchi riportato in Fig. 2, in cui le connessioni in rosso indicano le varie alimentazioni dei blocchi, mentre le connessioni in nero, indicano protocolli di comunicazione e scambio dati. Come si vede in Fig. 2, il cuore della scheda è il microcontrollore della Microchip Technology, il PIC32MX795F512L, dotato di 512 k di Flash e 128 k di RAM (più 12 k di Auxiliary Flash); ad esso vengono collegate tutte le periferiche, da quelle base a quelle evolute come la connettività Ethernet e USB, in modo da poter essere sfruttata come piattaforma dimostrativa per questa tipologia di applicazioni. Come si evince, per quanto riguarda la connessione USB è sia di tipo Device (per poter supportare classi quali la HID – Human Interface Device e la CDC – Connected Device Class, oltre a poter essere utilizzata come bus per il Bootloading) che di tipo Host.

Dallo schema a blocchi di Fig. 2 si nota che nella scheda la MCU PIC32MX ha le seguenti periferiche:

- un termometro a stato solido TC72 e slot per memoria SD mediante protocollo SPI (*Serial Peripheral Interface*);
- una piccola rete I<sup>2</sup>C costituita da un Port Expander MCP23008 (collegato a sua volta ad 8 LED) e ad una memoria EEPROM I<sup>2</sup>C 24LC16B; grazie alla presenza di questa piccola rete I<sup>2</sup>C si possono sviluppare e verificare applicazioni relative a tale protocollo;
- un microcontrollore PIC12F1822, interfaccia-

to attraverso la porta UART, il quale a sua volta è collegato a un pulsante e due LED (che possono essere gestiti anche mediante l'utilizzo di PWM);

- due potenziometri connessi ad altrettanti ingressi analogici;
- due LED utente generici (User1 e User2), connessi direttamente a due pin del microcontrollore principale, utilizzabili anche mediante appositi PWM;
- un display da 16x2 caratteri per permettere la visualizzazione di tutte le informazioni d'interesse;
- un joystick mobile nelle quattro direzioni con switch interno.

Completano la dotazione della board due header di espansione con doppia fila a ventiquattro poli (12x2). Per una maggiore flessibilità sono state previste connessioni di programmazione sia mediante un header a sei poli per l'utilizzo di un PicKit e sia mediante un connettore RJ11 per la programmazione con un ICD3. In aggiunta, vista la presenza dei due microcontrollori (PIC32 e PIC12), grazie a un header sulla scheda è possibile selezionare quale dei due programmare.

Per meglio organizzare il progetto, in KiCad (il CAD elettronico cui abbiamo dedicato un corso nei fascicoli dal 184 al 189) è stata fatta una gestione gerarchica delle varie parti del circuito, cioè il diagramma a blocchi riportato in Fig. 2 è stato tradotto nel diagramma gerarchico di Fig. 3. Questo ha permesso di dividere il progetto in sezioni, ognuna dedicata a una funzionalità specifica, e infine di impostare i vari collegamenti tra le parti per garantire il corretto funzionamento e la comunicazione di tutti i blocchi. Come si evince dalla Fig. 3, si hanno seguenti blocchi:

- Voltage\_Regulator: al suo interno è compresa tutta la gestione delle alimentazioni stabilizzate, i 5V e i 3,3V;
- USB: comprende le connessioni USB sia Host che Device;
- Ethernet: al suo interno troviamo il driver Ethernet, il connettore e tutta la componentistica associata per un suo corretto funzionamento;
- PIC12F: comprende il PIC12F1822, la componentistica associata e i connettori di programmazione;

- PIC32MX: il cuore di tutto il progetto, con il PIC32, il Joystick e i connettori di espansione;
- Devices: infine il blocco contenente le periferiche aggiuntive, come sensore di temperatura, scheda miniSD, EEPROM e via dicendo.

## SCHEMA ELETTRICO

A questo punto analizziamo lo schema elettrico nei dettagli dei vari blocchi, per farvi comprendere le scelte progettuali e analizzare più da vicino le caratteristiche riguardo le varie sezioni.

### Schema elettrico

Partiamo dalla descrizione del blocco funzionale del Voltage Regulator, ossia la sezione dedicata all'alimentazione.

Come si può vedere, nel circuito abbiamo due alimentazioni generate all'interno della scheda, la 3,3V e la 5V, ricavate mediante dei regolatori lineari a tensione d'uscita fissa, rispettivamente LD1117DT-3.3 e LD1117DT-5.0. Questi due regolatori sono della ST e presentano un basso dropout, tipicamente dell'ordine di 1V, con correnti di uscita fino a 800 mA; è possibile trovarli sia in versione "Adjustable", cioè con

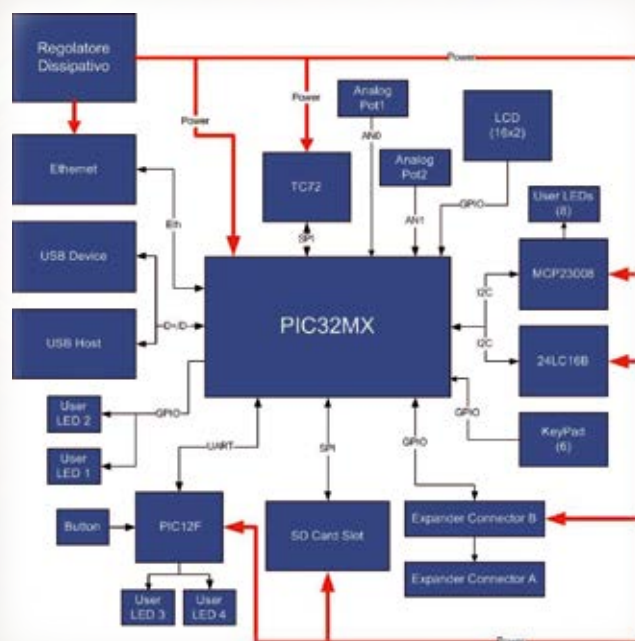
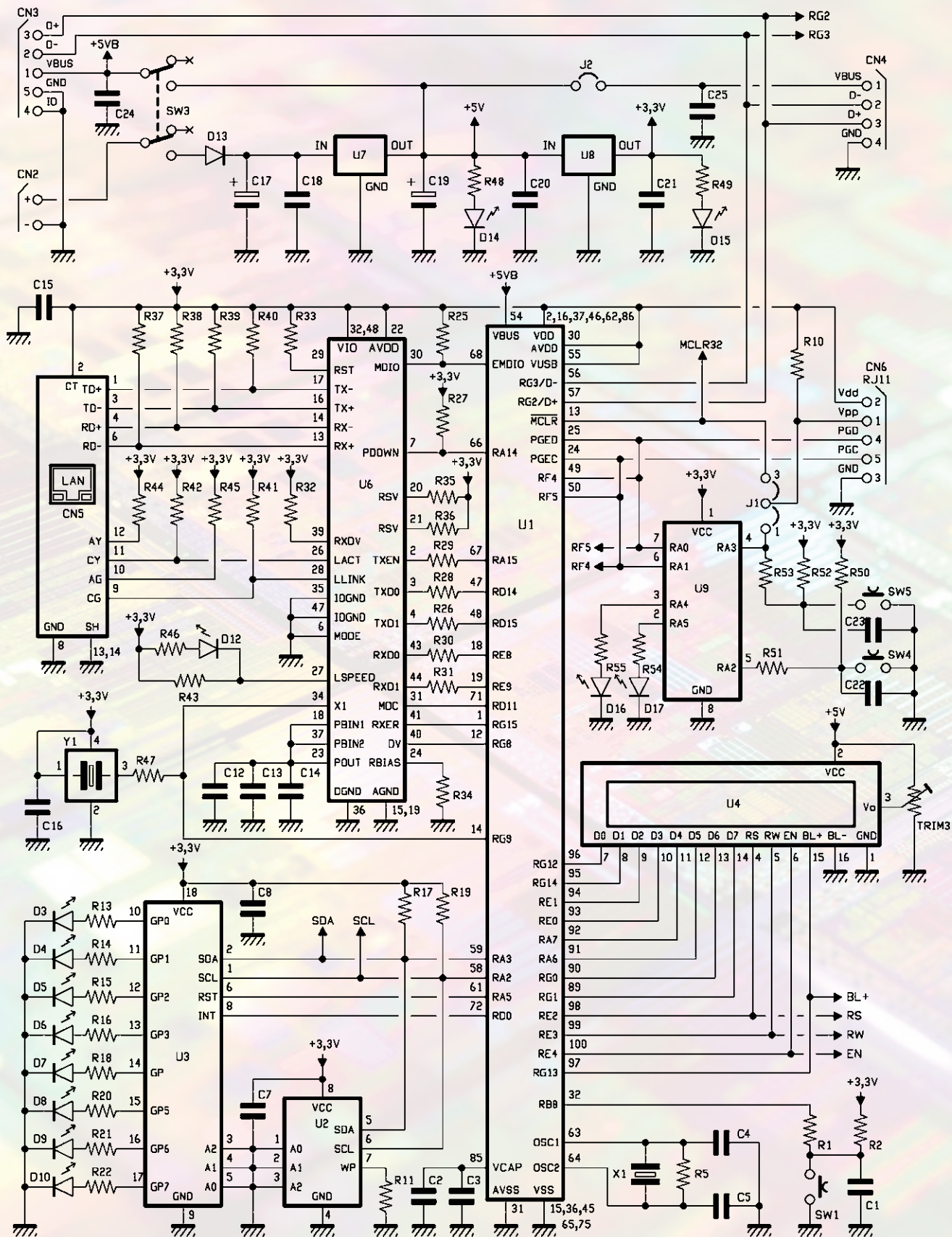
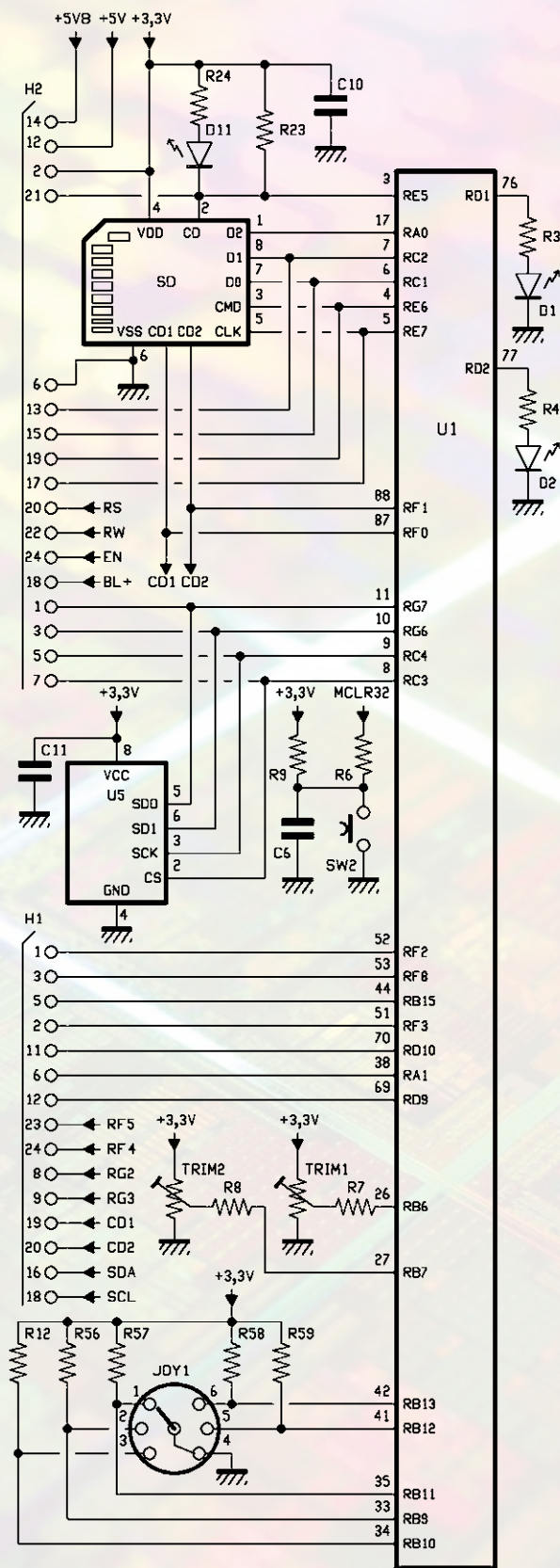


Fig. 2 - Diagramma a blocchi della DemoboardPIC32.

# [schema **ELETTRICO**]





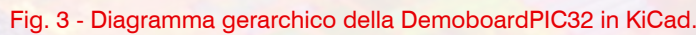
una tensione regolabile in uscita mediante una rete di resistenze esterne, che "Fixed", ossia con una tensione d'uscita fissa. Nel nostro caso avendo la necessità di due tensioni bene definite e stabili, si sono selezionati i due "part number" per i valori corrispondenti d'interesse. Per il loro corretto utilizzo necessitano soltanto di due condensatori per la stabilizzazione delle tensioni.

Nel pensare alla realizzazione di tale sezione, per comodità si è scelto di dare la possibilità all'utilizzatore di poter alimentare la scheda sia mediante un connettore jack standard (CN2 nello schema elettrico), per poter sfruttare alimentatori fino a un massimo di 15V, sia mediante connessione USB (VBUS). Tali alimentazioni esterne vengono poi riportate sul doppio deviatore SW3, grazie al quale è possibile accendere e spegnere l'intera board; molto comodo in svariate situazioni, perché evita di scollegare e ricollegare sempre uno dei due connettori d'alimentazione. Completano questo blocco, un diodo di protezione dall'inversione di polarità (D13) per evitare che inserendo un jack con alimentazione e massa invertiti, si possa danneggiare la scheda, un LED su ognuna delle due alimentazioni che fornisce un feedback visivo della corretta alimentazione della board e infine un jumper (J2) per scegliere se alimentare un dispositivo Host collegato direttamente alla scheda mediante connettore USB (sarà più chiaro in seguito analizzando il blocco funzionale USB).

## Blocco funzionale USB

Passiamo all'analisi del blocco funzionale USB, la sezione più semplice tra tutte perché contiene solo ed esclusivamente i connettori relativi all'USB. In esso sono presenti due connettori per la parte USB, uno per la funzionalità Device (CN3) e uno per la funzione Host (CN4); come generalmente accade il primo è un connettore "mini USB B", mentre il secondo un "USB A".

Visto che il microcontrollore PIC32 integra i vari moduli USB, da attivare e utilizzare, nello schema elettrico sono presenti esclusivamente le connessioni D+ e D- dai connettori direttamente verso le porte del micro. Quindi non c'è stata la necessità di considerare alcun altro driver per la gestione dell'USB. È doveroso precisare che condividendo le stesse linee dati,



ethernet, è stato previsto un connettore classico RJ-45, con LED (trasmissione/ricezione dati) già integrati nel case.

#### **Blocco funzionale PIC12F**

Questo blocco fa capo a un microcontrollore PIC12F1822, collegato al PIC32 e la sua componentistica di gestione/interfaccia, LED utente e switch utente serve per poter interagire con lo stesso. Infine in tale blocco è presente il connettore di programmazione in-circuit (ICSP) che è comune ai due microcontrollori; mediante il ponticello a tre poli J1 si sceglie quale dei due programmare: se si inserisce il jumper tra il pin 1 e 2 si programmerà il PIC12, mentre tra 2 e 3 si potrà programmare il PIC32. La scelta del microcontrollore da collegare al PIC32 è ricaduta sul PIC12F1822 perché tale integrato è uno dei più piccoli della Microchip ad avere una comunicazione EUSART (Enhanced Universal Synchronous Asynchronous Receiver Transmitter), infatti come riportato nello schema elettrico, i pin 6 e 7 vengono utilizzati per sfruttare tale protocollo di comunicazione. Nella DemoboardPIC32, questo microcontrollore può essere programmato e utilizzato per applicazioni molto semplici o anche per avvicinarsi per la prima volta al mondo dei prodotti Microchip, viste le sue caratteristiche non eccessivamente complesse. Infatti lo si può utilizzare per accendere e spegnere LED, leggere comandi utente mediante il tasto (SW4) oppure in collaborazione con il PIC32 mediante il protocollo UART per applicazioni molto più complesse. Per quanto riguarda il tasto SW5, è stato inserito per permettere all'utilizzatore di resettare il microcontrollore senza dover necessariamente spegnere l'intera scheda.

#### **Blocco funzionale PIC32MX**

Il penultimo modulo riguarda quello relativo al PIC32MX, il cuore di tutta la scheda. La MCU utilizzata è un PIC32MX795F512L, dotata di 512k di FLASH e 128k di RAM (più 12k di auxiliary FLASH). Il package selezionato è il TQFP (100-Lead Plastic ThinQuadFlatpack - PT) con 12x12x1mm di Body. Visto che nelle puntate precedenti di questo corso abbiamo analizzato ampiamente molte delle caratteristiche di questo dispositivo Microchip, eviteremo di approfondirle ulteriormente ma ci limiteremo

**Tabella 1 - Caratteristiche principali del microcontrollore PIC32MX795F512L.**

Caratteristica	Valore
MaxSpeed (MHz)	80
Program Memory Size (KB)	512
RAM (KB)	128
Auxiliary Flash (KB)	12
Temperature Range (C)	-40 to 105
Operating Voltage Range (V)	2.3 to 3.6
DMA Channels	8
SPITM	4
I2CTM Compatible	5
USB	FS Host/OTG
USB (Channels, Speed, Compliance)	1,FS Host/OTG,USB 2.0 OTG
A/D channels	16
Max A/D Resolution	10
Max A/D Sample Rate (KSPS)	1000
Input Capture	5
Output Compare/Std. PWM	5
16-bit Digital Timers	5
Parallel Port	PMP16
Comparators	2
Internal Oscillator	8 MHz, 32 kHz
I/O Pins	83
Pin Count	100

mo esclusivamente a una tabella riassuntiva che riporta quelle di maggiore interesse, si veda **Tabella 1**.

Si noti come in tale blocco siano state inserite anche altre parti oltre il PIC32 come l'oscillatore a 8 MHz, infatti si può facilmente distinguere il Joystick (JOY1), il tasto utente (SW1) e due potenziometri analogici (TRIM1 e TRIM2), molto utili per applicazioni in cui si ha la necessità di una diretta interazione con la DemoboardPIC32. Mentre come feedback visivo sono stati considerati due LED utente (D1 e D2).

Come già accennato, per consentirvi di gestire e sfruttare anche altri segnali provenienti dal microcontrollore, sono stati inseriti due Header da 24 pin (H1 e H2), che riportano all'esterno vari segnali "general purpose" oltre a segnali d'interfacce utilizzate e utili in fase di debug.

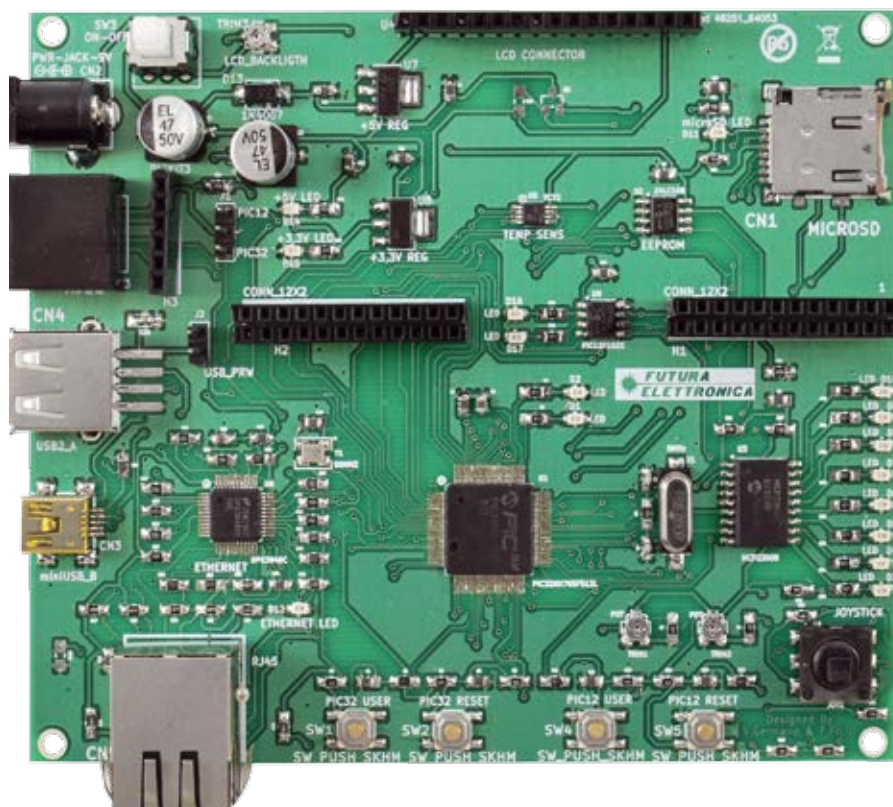
#### **Blocco funzionale Devices**

Concludiamo con l'analisi dell'ultimo blocco funzionale riguardante le interfacce aggiuntive che espandono e completano le funzionalità di questa evaluation board. In **Fig. 4** è riportata la sezione gerarchica dei Device, dove appare che tali integrati sono stati divisi per protocollo di comunicazione: SPI (Serial Peripheral Interface) e I<sup>2</sup>C (Inter Integrated Circuit).

In essa possiamo rapidamente distinguere:

## Elenco Componenti:

R1: 470 ohm (0805)  
 R2: 10 kohm (0805)  
 R3, R4: 470 ohm (0805)  
 R5: 1 Mohm (0805)  
 R6÷R8: 470 ohm (0805)  
 R9: 10 kohm (0805)  
 R10: 4,7 kohm (0805)  
 R11: 4,7 kohm (0805)  
 R12: 10 kohm (0805)  
 R13÷R16: 470 ohm (0805)  
 R17: 4,7 kohm (0805)  
 R18: 470 ohm (0805)  
 R19: 4,7 kohm (0805)  
 R20÷R22: 470 ohm (0805)  
 R23: 10 kohm (0805)  
 R24: 4,7 kohm (0805)  
 R25: 1,5 kohm (0805)  
 R26: 33 ohm (0805)  
 R27: 1,5 kohm (0805)  
 R28÷R31: 33 ohm (0805)  
 R32: 2,2 kohm (0805)  
 R33: 1,5 kohm (0805)  
 R34: 4,87 kohm 1% (0805)  
 R35, R36: 2,2 kohm (0805)  
 R37÷R40: 49,9 ohm 1% (0805)  
 R41÷R43: 2,2 kohm (0805)  
 R44, R45: 249 ohm 1% (0805)  
 R46: 510 ohm (0805)  
 R47: 33 ohm (0805)  
 R48: 470 ohm (0805)  
 R49: 330 ohm (0805)  
 R50: 10 kohm (0805)  
 R51: 470 ohm (0805)  
 R52: 10 kohm (0805)



R53÷R55: 470 ohm (0805)  
 R56÷R59: 10 kohm (0805)  
 R60: 0 ohm (0805)  
 C1, C2: 100 nF ceramico (0805)  
 C3: 4,7 µF ceramico (0805)  
 C4: 22 pF ceramico (0603)

C5: 22 pF ceramico (0603)  
 C6÷C12: 100 nF ceramico (0805)  
 C13: 4,7 µF ceramico (0805)  
 C14÷C16: 100 nF ceramico (0805)  
 C17: 47 µF 16VL elettrolitico (Æ4 mm)  
 C18: 100 nF ceramico (0805)

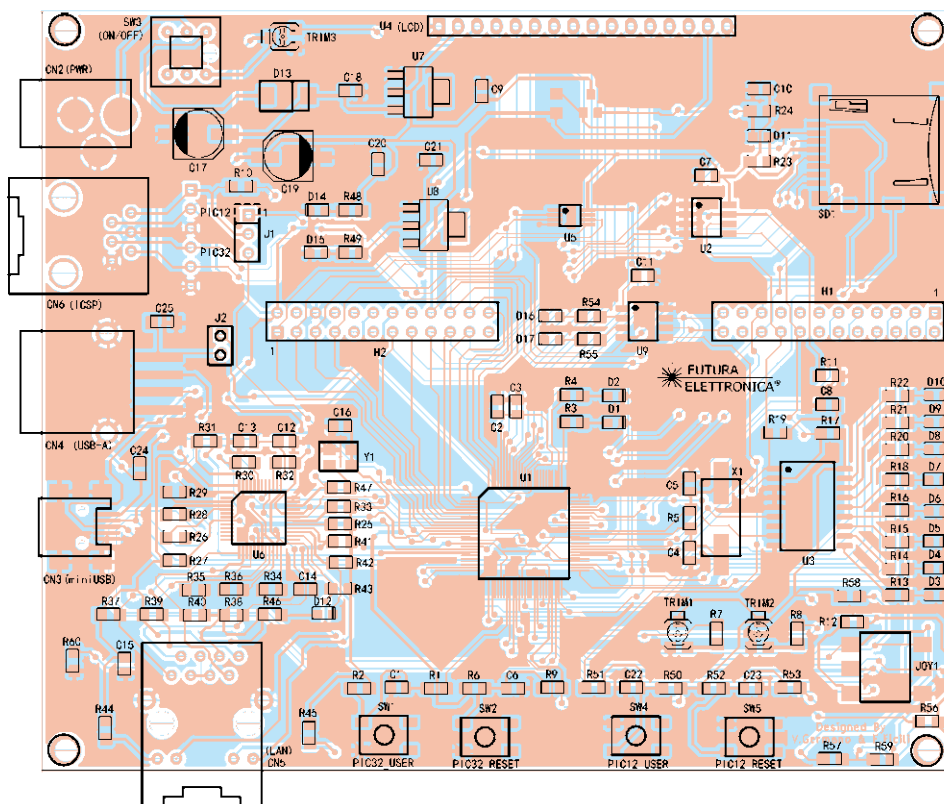
- un "I/O expander" usato come driver LED, che è l'integrato MCP23008 (U3);
- una EEPROM, 24LC16B (U2), per la memorizzazione di dati;
- un display LCD (U4), per la visualizzazione di dati utente;
- un sensore di temperatura TC72 (U5), per il rilevamento della temperatura;
- uno slot SD Card (CN1), per lo storage di dati all'interno di una miniSD.

L'integrato MCP23X08 a 8 bit è un driver di uso generale, ed è possibile reperirlo in due differenti versioni, dipendentemente dal protocollo di comunicazione utilizzato e quindi dai pin usati:

- MCP23008 - interfaccia I<sup>2</sup>C;

- MCP23S08 - interfaccia SPI.

Come visto precedentemente, il part number utilizzato è quello relativo all'I<sup>2</sup>C-Bus. Riguardo la sua funzionalità principale, molto semplicemente, riceve tutte le informazioni di come pilotare le sue otto uscite (GP0-GP7) mediante il protocollo I<sup>2</sup>C, abilitandole e/o disabilitandole. Perciò nella DemoboardPIC32 è stato selezionato per pilotare 8 differenti LED. Durante la fase progettuale della scheda, si voleva dare la possibilità all'utilizzatore di poter testare e famigliarizzare con il protocollo I<sup>2</sup>C, perciò oltre al I/O expander precedente è stato previsto un altro componente in I<sup>2</sup>C, la EEPROM 24LC16B. Tale memoria, sempre della Microchip, è una 16 kbit PROM cancella-



C19: 47  $\mu$ F 16VL elettrolitico (AE4 mm)  
 C20: 4,7  $\mu$ F ceramico (0805)  
 C21: 4,7  $\mu$ F ceramico (0805)  
 C22÷C24: 100 nF ceramico (0805)  
 D1÷D11: LED rosso (0805)  
 D12: LED giallo (0805)

D13: GF1M  
 D14: LED verde (0805)  
 D15: LED verde (0805)  
 D16: LED rosso (0805)  
 D17: LED rosso (0805)  
 U1: PIC32MX795F512L-80I/PT

(MF1224)  
 U2: 24LC16B-I/SN  
 U3: MCP23008-E/SO  
 U4: Display LCD 16x2  
 (MC21605C6W-SPR)  
 U5: TC72-3.3MUA  
 U6: DP83848CVVX/NOPB  
 U7: LD1117S50TR  
 U8: LD1117S33TR  
 U9: PIC12F1822-I/SN  
 SW1, SW2: Pulsante SMD  
 basso profilo  
 SW3: Doppio deviatore DPDT  
 SW4: Pulsante SMD basso profilo  
 SW5: Pulsante SMD basso profilo  
 TRIM1: Trimmer 10 kohm (TC33X)  
 TRIM2: Trimmer 10 kohm (TC33X)  
 TRIM3: Trimmer 10 kohm (TC33X)  
 X1: Quarzo 8 MHz  
 Y1: Oscillatore 50 MHz  
 JOY1: Joystick SKQUDBE010

#### Varie:

- Plug alimentazione
- Connettore mini-USB
- Connettore USB-A Femmina
- Connettore micro-SD
- Connettore RJ11 da CS
- Connettore LAN-RJ45
- Strip maschio 3 vie
- Strip maschio 6 vie
- Strip maschio 16 vie
- Strip femmina 16 vie
- Strip femmina 2x12 vie (2 pz.)
- Jumper (2 pz.)
- Circuito stampato S1224

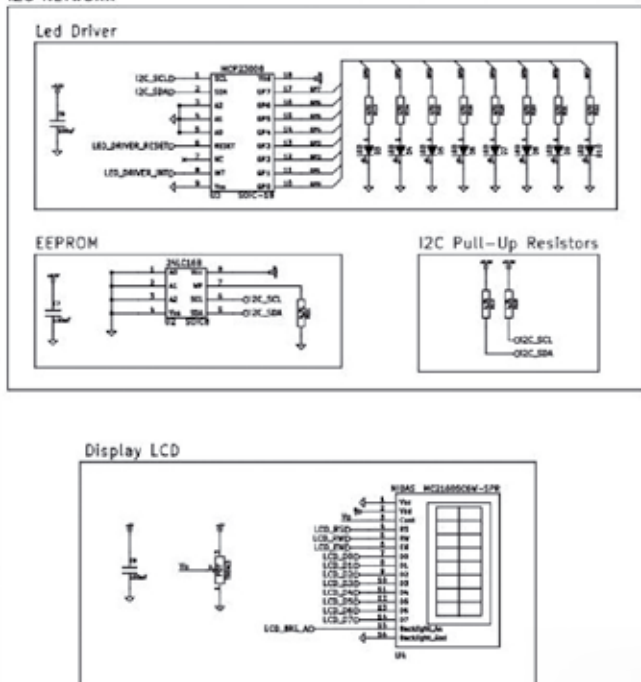
bile elettricamente; organizzata in otto blocchi di memoria da 256 x 8-bit. Consente il funzionamento fino a 2,5 V con uno standby e con correnti attive di solo 1  $\mu$ A. Offre anche una capacità di scrittura pagina fino a un massimo di 16 byte di dati. In una configurazione di questo tipo, l'utente ha la possibilità di scrivere applicazioni che sfruttano tali integrati famigliarizzando con il protocollo I<sup>2</sup>C-Bus.

Oltre al protocollo UART e I<sup>2</sup>C, mediante la board è possibile utilizzare e testare il protocollo SPI, grazie al TC72 e allo slot per le miniSD: il primo è un sensore di temperatura digitale in grado di effettuare una lettura in temperatura da -55 °C a + 125 °C; dotato di un'interfaccia seriale che permette la comunicazione con un controller host o altre periferiche, non richiede

ulteriori componenti esterni per un corretto funzionamento. Tuttavia, abbiamo preferito dotarlo di un condensatore di disaccoppiamento da 100 nF tra i pin di alimentazione e massa. Il TC72 può essere utilizzato sia in modalità di conversione continua della temperatura, che in modalità di conversione "One-Shot"; essendo un integrato ad alta precisione e semplice da utilizzare, è la soluzione ideale per implementare la gestione termica in un'ampia varietà di sistemi.

Per quanto riguarda lo slot per le miniSD, è stato collegato in SPI verso il PIC32, prevedendo un LED di feedback per l'utilizzatore in caso tale slot venga usato. Può essere utile per memorizzare grandi quantità di dati e quindi soprattutto in applicazioni di "data logging".

## I2C NETWORK



## SPI DEVICES

### Temp Sensor



### SD Card Slot

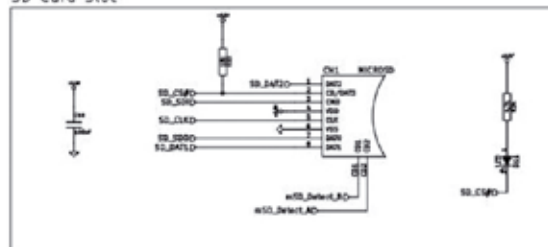
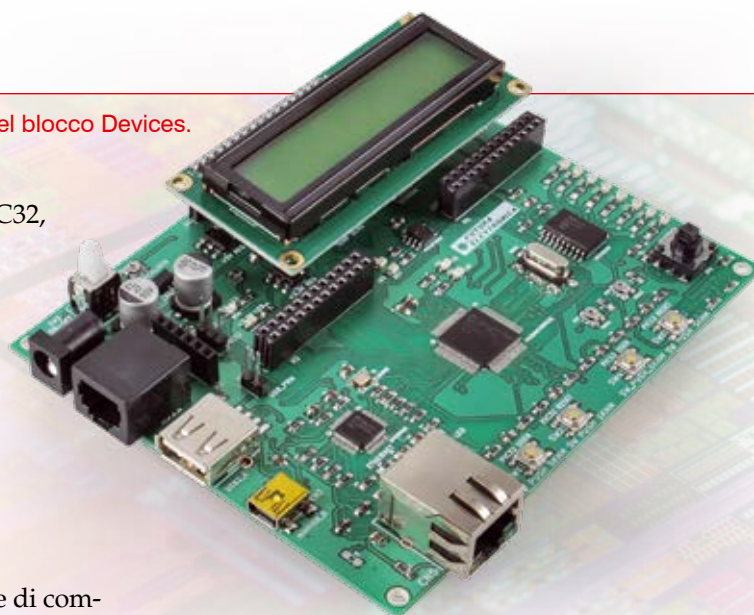


Fig. 4 - Diagramma gerarchico in KiCad del blocco Devices.

A completamento della DemoboardPIC32, è stato previsto un display (U4) per visualizzare tutte le informazioni d'interesse. Il display è un 16x2 segmenti del quale il microcontrollore governa la retroilluminazione e le linee dati e di controllo; mediante un potenziometro è possibile variarne il contrasto.

## REALIZZAZIONE PRATICA

Vediamo adesso come si costruisce la demoboard, che fa uso per buona parte di componenti SMD, quindi è richiesto l'uso di fusante, filo di stagno sottilissimo e saldatore da 20 W con punta finissima, oltre che di pinzette e di una lente d'ingrandimento per posizionare gli elementi prima di saldarli. Per il posizionamento dei componenti fate riferimento al piano di montaggio visibile nelle pagine precedenti; i primi componenti da montare sono gli integrati SMD, che bisogna centrare nelle rispettive piazzole, quindi fermare stagnando un pin e procedendo poi un lato dalla volta. Per il display prevedete appositi pin-strip a passo 2,54 mm. Completate il montaggio con i pin-strip per i jumper, i pulsanti e il deviatore, i connettori per USB, alimentazione, ethernet ecc. ■



## per il MATERIALE

La DemoboardPIC32 (cod. FT1224M) viene fornita montata, collaudata e completa di display LCD. Può essere acquistata presso Futura Elettronica al prezzo di Euro 98,00. Il prezzo si intende IVA compresa.

Il materiale va richiesto a:  
Futura Elettronica, Via Adige 11, 21013 Gallarate (VA)  
Tel: 0331-799775 - Fax: 0331-792287  
<http://www.futurashop.it>