



Microsoft®

SCOPRIAMO .NET Micro Framework

di Lorenzo Maiorfi e Gianluca Ruta
Innovactive Engineering

Tentiamo un primo approccio alla versione dedicata ai microcontrollori, dello strumento già creato da Microsoft per realizzare una Virtual Machine su Personal Computer. Vedremo le proprietà ed un primo esempio d'uso.

Se è vero che tutti conoscono Microsoft per la produzione di sistemi operativi, software e ambienti di sviluppo per i Personal Computer, è altrettanto vero che pochi sanno che la casa di Redmond ormai da qualche anno propone una piattaforma di sviluppo di applicazioni per dispositivi embedded, che si configura come la "sorella minore" del Microsoft *.NET Framework*. Nel corso che prende le mosse con questa puntata, vogliamo introdurre l'uso di questa piattaforma, denominata

.NET Micro Framework perché nasce per offrire le stesse qualità di *.NET Framework* al settore dei microcontrollori.

IN COSA CONSISTE IL .NET MICRO FRAMEWORK

Il Micro Framework è un'infrastruttura per microcontrollori a 32 bit i cui vantaggi principali consistono nel fornire una ricca API indipendente dall'hardware, un'efficiente controllo sulle periferiche del microcontrollore,

una runtime che gestisce aspetti che vanno dal bootstrap al debug del firmware, un ricco ambiente di sviluppo quale Visual Studio, il supporto all'emulazione del dispositivo oltre, naturalmente, a innumerevoli altri vantaggi intrinseci del Framework .NET, indipendentemente dal fatto che si parli della versione "Micro" (per microcontrollori), della versione "Compact" (per palmari e telefoni, ma utilizzato anche sulla XBOX) o della versione "maggiore" del framework, destinata allo sviluppo di software per Personal Computer.

Rilasciato nella sua prima versione nel 2002, il Framework .NET rappresenta l'interpretazione di Microsoft della tendenza -già consolidatasi negli anni precedenti con Java- alla "virtualizzazione" del microprocessore. L'obiettivo principale del progetto è infatti costituito dalla realizzazione di una "virtual machine" (così chiamata nel gergo Java) denominata "CLR" (Common Language Runtime), in grado di compilare o interpretare (a seconda delle implementazioni) codice scritto in un linguaggio indipendente dall'hardware, denominato IL (Intermediate Language), traducendolo "just-in-time" in istruzioni native per lo specifico microprocessore su cui si sta eseguendo l'applicazione.

I vantaggi della virtualizzazione sono numerosi ed evidenti, primi tra tutti quelli dell'indipendenza dall'hardware e della maggior possibilità di controllo su quanto viene eseguito, a fronte di un calo delle prestazioni assolutamente trascurabile.

A tale proposito, sono invece numerosi i casi in cui è stato dimostrato che un'applicazione

destinata ad essere eseguita dalla CLR (ossia "managed", come viene comunemente definita) può addirittura presentare performance superiori alla controparte "nativa", in quanto la possibilità da parte della runtime di ottimizzare l'esecuzione per lo specifico hardware su cui si sta eseguendo l'applicazione (tenendo conto ad esempio della versione del sistema operativo, del numero di core o processori e della RAM effettivamente a disposizione) permette di aumentare le prestazioni più di quanto esse non calino per gestire la compilazione "just-in-time" da parte dell'infrastruttura.

Nel mondo dei microcontrollori, in cui le architetture dominanti sono ben più numerose di quelle presenti nel mondo del Personal Computer (in cui sostanzialmente troviamo x86, x64 e Itanium) i vantaggi derivanti dall'adozione di un framework basato su una runtime che virtualizzi l'hardware sono ancora più evidenti. Grazie al .NET Micro Framework, possiamo infatti realizzare applicazioni che possono essere eseguite indistintamente su processori Atmel, NXP, Freescale, Analog Devices, ecc. Per meglio comprendere come si incastrino tutti i tasselli di hardware, funzionalità del framework e applicazioni managed, diamo un'occhiata all'architettura del Micro Framework, riassunta schematicamente nella Fig. 1.

IL CUORE DEL SISTEMA: LA HAL E LA PAL

In un sistema embedded di piccole dimensioni non troviamo quasi mai un sistema operativo, soprattutto in considerazione dell'impatto che questo potrebbe avere sulle risorse.

Per questo motivo il Micro Framework ne fa tipicamente a meno e, al contrario dei suoi fratelli maggiori (Compact Framework e Framework.NET), è in grado di prendere direttamente il controllo dell'hardware.

Il componente al livello più basso è detto HAL (Hardware Abstraction Layer) e si occupa di gestire le operazioni più a stretto contatto con il microprocessore, come la gestione dell'I/O, gli interrupt e i timer, nonché l'intero processo di bootstrap.

I componenti dell'HAL collaborano con i driver nativi che permettono di astrarre le funzionalità delle periferiche hardware.

A seconda del produttore hardware il .NET

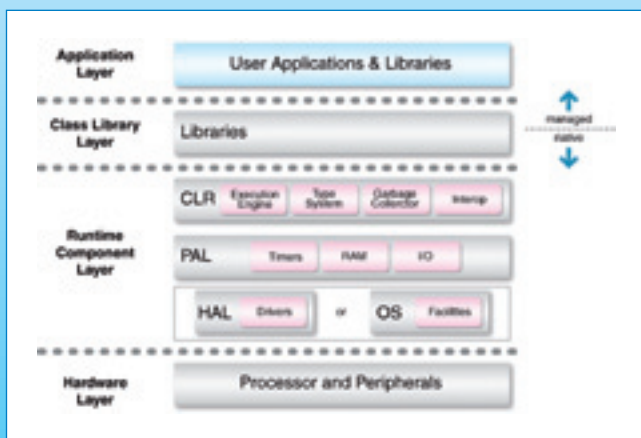


Fig. 1 - L'architettura del .NET Micro Framework.

Micro Framework porta con sé una serie di driver, come ad esempio quello per la gestione della seriale asincrona, delle comunicazioni sincrone (SPI e I²C), per pilotare i display LCD, per il monitoraggio dell'alimentazione, per la gestione delle porte di I/O e del modulo per la generazione PWM, per l'utilizzo di memorie Flash ed EEPROM e per molto altro ancora.

I driver nativi vengono tipicamente scritti dagli OEM che forniscono il "porting" del .NET Micro Framework specifico per la loro architettura.

Sopra la HAL siede un altro componente fondamentale, denominato PAL (Platform Abstraction Layer) che utilizza le funzionalità dello strato HAL per implementare nel modo più efficiente possibile caratteristiche tipiche del Micro Framework, quali il multi-threading ed il garbage-collector (il servizio in grado di liberare automaticamente blocchi di memoria non più utilizzati da parte del firmware).

Al di fuori di questi componenti (HAL, PAL e driver a basso livello) il resto del Micro Framework è indipendente dall'hardware.

Il porting del Micro Framework su un hardware differente si riduce perciò all'implementazione di entrambi i componenti o della sola PAL. Ad oggi, la famiglia di microcontrollori per i quali esiste il maggior numero di implementazioni in tal senso è senz'altro ARM (7 e 9).

Un altro compito importante della HAL è provvedere alla modalità di esecuzione del codice, che avviene su singolo thread o tramite ISR (Interrupt Service Routine). In sostanza Micro Framework rinuncia allo Scheduler dei thread in luogo di un multitasking di tipo cooperativo, in cui ciascun thread riceve una finestra temporale massima di 20 ms, che rappresenta quindi la "risoluzione" temporale minima per la realizzazio-

Listato 1

```
using System;

public class Impiegato
{
    DateTime _dataAssunzione;
}
```

ne di applicazioni che abbiano necessità di garantire un ritardo massimo nell'esecuzione di determinate operazioni di controllo.

IL MODELLO DI SVILUPPO

Il modello di sviluppo del Framework .NET prevede la generazione di uno o più file, denominati assembly, che agiscono come contenitori fisici (ed autodocumentanti, grazie all'esposizione di un piccolo database interno) delle entità logiche (tipicamente classi) che sono le principali responsabili del comportamento del software sviluppato.

Senza addentrarci in temi che possono essere facilmente approfonditi dalle moltissime fonti reperibili in argomento (primo fra tutti Internet), vale la pena di focalizzare la nostra attenzione sul fatto che qualsiasi applicazione .NET può essere vista come un insieme di moduli (principalmente classi e, in misura ridotta, interfacce e "tipi valore") legati da rapporti

Listato 2

```
using System;

public class Impiegato
{
    public void Licenzia()
    {
        // [...]
    }

    public bool MetodoConValoreDiRitorno()
    {
        // [...]
        return true;
    }

    public void MetodoConParametroIn(string par1, int par2)
    {
        // [...]
    }

    public void MetodoConParametriOutERef(out int par1, ref string par2)
    {
        // [...]

        par1 = 10;
        par2 = par2.ToUpper();

        // [...]

        return;
    }
}
```

di dipendenza del tipo chiamante/chiamato, oppure base/derivato, oppure contenitore/contenuto, in cui ciascuna entità è caratterizzata da un insieme di "membri" che possiamo riassumere in *field*, *metodi*, *proprietà* ed *eventi*.

I field sono membri che rappresentano un elemento di informazione costituente lo stato del tipo cui appartengono. Ad esempio, in un'applicazione per la gestione del personale di un'azienda, per un oggetto di tipo "Impiegato" è probabile che la data di assunzione sia rappresentata da un field, come illustrato nel frammento di codice contenuto nel **Listato 1**. I metodi rappresentano invece delle azioni che è possibile compiere sull'oggetto che li espone: ad esempio, per l'oggetto di tipo "Impiegato" è possibile prevedere un metodo denominato "Licenzia()".

Le parentesi accanto al nome del metodo, tipicamente rappresentano la lista dei parametri (vuota, in questo caso) che intendiamo passare al metodo.

Essi contribuiscono a creare un contesto per l'azione da svolgere. I metodi possono anche restituire un oggetto a titolo di valore di ritorno. Oltre a poter utilizzare tipi di oggetto differenti per i parametri del metodo, ciascun parametro è caratterizzato da una "direzione": ingresso (utilizzando la clausola "in", il default) uscita (con "out") o entrambi (con "ref"). Il frammento di codice nel **Listato 2** esemplifica quanto detto.

Le proprietà sono dei particolari metodi con semantica "get/set" che intervengono nel momento in cui il programma che le invoca accede, rispettivamente in lettura o in assegnazione, come ad esempio nel caso di una proprietà "Stipendio" esposta dalla classe "Impiegato", in cui un'eventuale assegnazione di un valore non valido (ad esempio un valore negativo) può essere intercettato grazie all'esecuzione del metodo "set" relativo alla proprietà. Come esempio, guardate il frammento di codice contenuto nel **Listato 3**.

Un evento è un ulteriore tipo di membro che rappresenta un metodo in cui chiamante e chiamato si invertono di ruolo; in altri termini, un oggetto che solleva un evento provoca l'esecuzione di una chiamata in uno o più "ascoltatori" (se presenti), nei confronti dei quali avviene una chiamata a metodo (il frammento di codice nel **Listato 4** presenta un esempio di evento).

Oltre alle possibilità messe a disposizione da codice "managed", .NET Micro Framework prevede un meccanismo di Interoperabilità che permette di scrivere una funzione direttamente in modalità nativa, tipicamente per quelle parti di applicazione che dovessero essere particolarmente critiche in termini di performance o controllo dell'hardware. Nella versione attuale di Micro Framework, la chiamata al codice nativo richiede l'uso del "Porting Kit", un SDK specifico tipicamente

rivolto a chi realizza nuove implementazioni della piattaforma. Ovviamente, pur risolvendo situazioni diversamente non affrontabili, l'uso dell'Interoperabilità complica notevolmente la portabilità del codice e può compromettere l'efficienza della CLR, motivo per cui questa soluzione deve essere utilizzata con grande attenzione.

LA GESTIONE DELLA MEMORIA

Uno dei pregi delle versioni del Framework

Listato 3

```
using System;

public class Impiegato
{
    decimal _stipendioBase;
    decimal _rimborsoSpese;

    public decimal Stipendio
    {
        get
        {
            return _stipendioBase + _rimborsoSpese;
        }

        set
        {
            if (value < 0) throw new ArgumentException("Valore non valido");

            _stipendioBase = value;
        }
    }
}
```

.NET è il *Garbage Collector*, che permette di mappare la memoria a livello di oggetto. Non appena un oggetto diventa orfano dei suoi reference diretti e indiretti, il Garbage Collector è libero di riusare la zona di memoria corrispondente.

Il compito del Garbage Collector è duplice: da una parte si occupa di rendere disponibile la memoria inutilizzata; dall'altra provvede a spostare i blocchi di memoria quando si libera lo spazio in mezzo. Quest'ultima procedura viene chiamata Garbage Collection e avviene sospendendo temporaneamente l'esecuzione dei thread e riaggiornando i reference ai blocchi di memoria spostati.

Il vantaggio principale della Garbage Collection è quello di evitare la frammentazione della memoria, fenomeno derivante dalla continua allocazione e deallocazione di blocchi di memoria di diversa misura che, alla lunga, porta a non avere più la disponibilità di blocchi contigui di memoria sufficientemente capienti. Quando si parla di CLR e della Garbage Collection, in molti si spaventano per il tempo di elaborazione necessario a svolgere questo processo; nel Micro Framework è stata implementata una versione di essa molto leggera, con un algoritmo di Mark+Sweep non incrementale in luogo del tradizionale sistema generazionale.

Il CLR prevede un meccanismo chiamato Weak Reference, che permette al Garbage Collector di reclamare memoria sotto condizioni di Memory Pressure, ossia quando si arriva vicini all'esaurimento di memoria. In aggiunta a questo prezioso meccanismo, il Micro Framework aggiunge l'Extended Weak Reference, che permette agli oggetti di essere persistiti e recuperati in memoria durante un reboot. Se consideriamo che il boot è molto sfruttato negli ambienti embedded, anche per assicurare la stabilità sul lungo termine, questo meccanismo si rivela assolutamente molto prezioso.

LA BASE CLASS LIBRARY (BCL)

La presenza di una runtime in grado di astrarre lo specifico hardware, consente, fra i tanti, l'indubbio vantaggio di poter definire delle "librerie" di classi relative allo svolgimento delle attività più comuni e frequenti. Chi sviluppa applicazioni con il .NET Micro

Listato 4

```
using System;

public class Impiegato
{
    int _livello;

    public event EventHandler Promosso;

    protected void OnPromosso()
    {
        if (Promosso != null)
        {
            Promosso(this, EventArgs.Empty);
        }
    }

    public void Promuovi()
    {
        _livello++;

        OnPromosso();
    }
}
```

Framework può infatti contare su una nutrita "Base Class Library" che, sfruttando lo strato HAL/PAL di cui sopra, permette di accedere a tutte le periferiche del microcontrollore utilizzando un modello ad oggetti coerente e funzionale.

Grazie poi alla struttura object oriented di questa piattaforma ed alla sua architettura, lo sviluppatore può estendere facilmente le classi esistenti o crearne di nuove che potranno essere facilmente riutilizzate anche su piattaforme hardware differenti.

Troviamo classi per la gestione delle porte I/O, delle porte seriali, delle porte I²C ed SPI, del file system, dei convertitori ADC e DAC built-in, dei moduli PWM e, per i produttori che forniscono un porting più completo, troviamo classi per la gestione dell'USB host/de-



Fig. 2 - Una schermata dell'ambiente Microsoft Visual Studio e dell'emulatore di hardware

Listato 5

```

using System;
using System.Collections;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace USBiziTest
{
    public class Program
    {
        static ArrayList al;

        public static class MyPins
        {
            public const Cpu.Pin GPIO39 = (Cpu.Pin)GHIElectronics.NETMF.Hardware.USBizi.Pin.IO39;
            public const Cpu.Pin GPIO1 = (Cpu.Pin)GHIElectronics.NETMF.Hardware.USBizi.Pin.IO1;
            public const Cpu.Pin GPIO2 = (Cpu.Pin)GHIElectronics.NETMF.Hardware.USBizi.Pin.IO2;
            public const Cpu.Pin GPIO3 = (Cpu.Pin)GHIElectronics.NETMF.Hardware.USBizi.Pin.IO3;
            public const Cpu.Pin GPIO5 = (Cpu.Pin)GHIElectronics.NETMF.Hardware.USBizi.Pin.IO5;
            public const Cpu.Pin GPIO7 = (Cpu.Pin)GHIElectronics.NETMF.Hardware.USBizi.Pin.IO7;
        }

        public static void Main()
        {
            //Inizializza la porta I/O 39 come porta di ingresso con resistenza di pull up interna
            InputPort ip = new InputPort(MyPins.GPIO39, true, Port.ResistorMode.PullUp);

            //Inizializza la porta I/O 2 come porta di uscita (è quella collegata al led presente sulla scheda USBizi)
            OutputPort op = new OutputPort(MyPins.GPIO2, false);

            //Inizializza le porte I/O 1, 3, 5 e 7 come porte tristate
            TristatePort tp1 = new TristatePort(MyPins.GPIO1, false, false, Port.ResistorMode.PullUp);
            TristatePort tp3 = new TristatePort(MyPins.GPIO3, false, false, Port.ResistorMode.PullUp);
            TristatePort tp5 = new TristatePort(MyPins.GPIO5, false, false, Port.ResistorMode.PullUp);
            TristatePort tp7 = new TristatePort(MyPins.GPIO7, false, false, Port.ResistorMode.PullUp);

            //Per realizzare l'accensione in sequenza si utilizza una collezione che includa le porte tristate
            al = new ArrayList();
            al.Add(tp1);
            al.Add(tp3);
            al.Add(tp5);
            al.Add(tp7);

            bool ipState;
            int step = 0;

            //Imposta la situazione di partenza: led acceso e primo relé acceso
            op.Write(true);
            StepUp(al, 0);

            //ciclo ripetuto all'infinito
            while (true)
            {
                //Legge lo stato della porta di ingresso
                ipState = ip.Read();

                //Se si legge un valore basso
                if (!ipState)
                {
                    //Cambia lo stato di accensione/spegnimento del led
                    op.Write(!op.Read());

                    //Conta un passo in più e di conseguenza accende/spegne i relé
                    //passando al metodo opportuno il valore del resto della divisione del passo attuale per 4
                    step++;
                    StepUp(al, step % 4);
                }

                Thread.Sleep(200);
            }

            //Questo metodo consente di realizzare l'accensione in sequenza dei relé utilizzando
            //la collezione delle porte tristate e l'indice dell'elemento nella collezione che va acceso

```

Listato 5 - (continuazione)

```

private static void StepUp(ArrayList al, int step)
{
    foreach (object tp in al)
    {
        if (al.IndexOf(tp) == step)
        {
            ((TristatePort)tp).Active = true;
            ((TristatePort)tp).Write(false);
        }
        else
        {
            if (((TristatePort)tp).Active == true)
            {
                ((TristatePort)tp).Active = false;
            }
        }
    }
}

```

vice, della rete, del CAN, della comunicazione 1-wire e molto, molto altro ancora.

Di grande interesse sono inoltre le numerose classi del namespace *Microsoft.SPOT*.

Presentation, che costituiscono un insieme di funzionalità molto ricche per la generazione di primitive grafiche come rettangoli, ellissi, linee, poligoni e immagini sui display per i cui controller esiste un driver nel porting che si sta utilizzando.

C'è anche un ampio supporto per il testo, il suo allineamento, il font e la formattazione, sebbene con alcune limitazioni sull'uso dei colori e il calcolo dello spazio occupato sullo schermo. Ed ancora, meritano di essere citati una buona scelta di controlli come *TextFlow*, *Border*, *Canvas* (un pannello), *Image*, *ListBox*, e *StackPanel*; quest'ultimo permette un allineamento a stack mirato ad evitare di dover specificare le coordinate in cui posizionare i controlli.

Grazie alle classi base è inoltre possibile costruire controlli personalizzati particolarmente attrattivi nel mondo embedded, specie se combinati, ad esempio, con un touch-screen, implementato nativamente dalla versione 4.0 del .NET Micro Framework.

L'AMBIENTE DI SVILUPPO E L'EMULATORE

Uno degli aspetti che in maniera più evidente caratterizza l'esperienza di sviluppo di firmware con .NET Micro Framework, è senz'altro quello inerente l'ambiente di sviluppo utilizzato.

A differenza di quanto avviene per molte case produttrici di sistemi per la programmazione di microcontrollori, Microsoft ha scelto di realizzare il kit di sviluppo per .NET Micro Framework come "plug-in" del noto Visual

Studio. In particolare, la versione più recente del .NET Micro Framework (la 4.0) prevede l'installazione come estensione di Visual Studio 2008. È inoltre imminente la release 4.1, che consentirà lo sviluppo all'interno del nuovo Visual Studio 2010. Vale la pena di sottolineare che il .NET Micro Framework SDK può essere installato all'interno di tutte le versioni di Visual Studio ma anche come plug-in di Visual C# Express Edition, il "fratello minore" gratuito di Visual Studio, dedicato allo sviluppo di applicazioni scritte con il linguaggio C#, scaricabile liberamente dalla pagina web <http://www.microsoft.com/express/>.

Gli elementi di Visual Studio con cui lo sviluppatore di applicazioni .NET Micro Framework si confronta, sono sommariamente l'editor, l'emulatore ed il debugger.

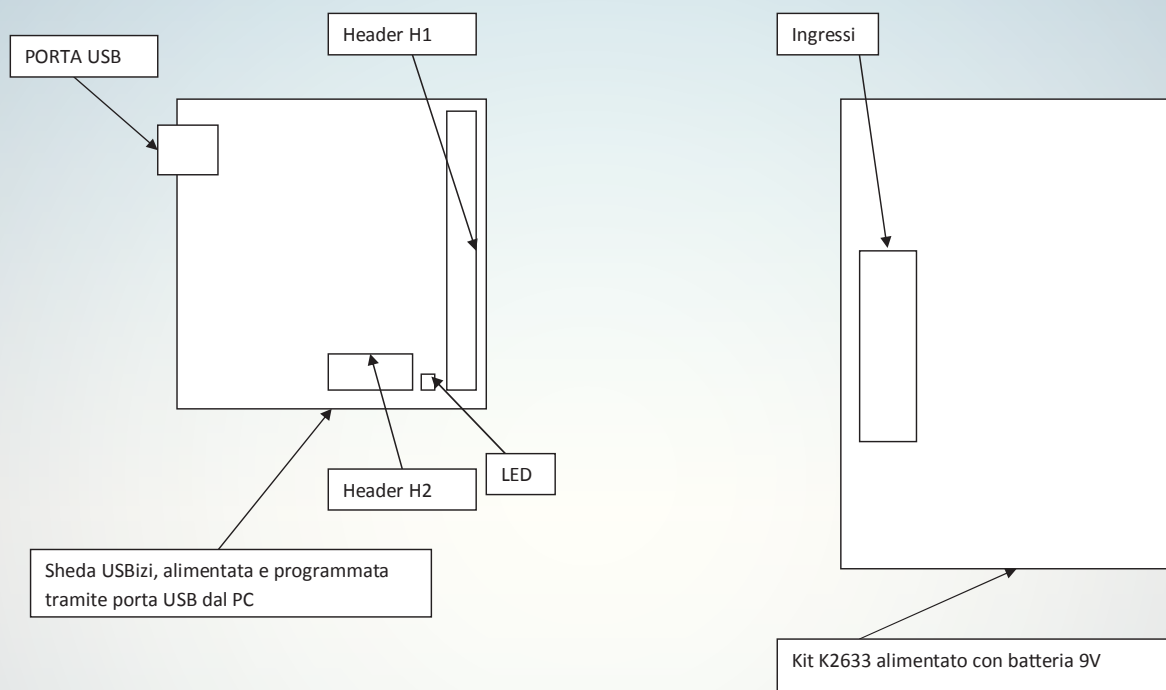
L'editor permette la scrittura di codice in linguaggio C# con un notevole supporto da parte dell'ambiente: indentazione automatica, autocompletamento, suggerimenti sulle modalità di invocazione, cross-reference di tutti gli oggetti definiti o utilizzati e molto altro ancora. L'emulatore integrato (si veda la Fig. 2) consente di verificare il funzionamento del proprio firmware pur senza disporre del dispositivo reale.

Le varie periferiche del dispositivo vengono simulate attraverso l'utilizzo di specifici oggetti software che in molti casi possono velocizzare le fasi di sviluppo più slegate delle peculiarità del dispositivo hardware di destinazione, quali quelle relative alla generazione delle interfacce utenti o all'implementazione di logica di controllo o di elaborazione.

Il debugger, utilizzabile indifferentemente con l'emulatore o con il microcontrollore hardware, consente di eseguire il firmware sviluppato in una modalità in cui, eseguendo l'applicazione passo-passo o bloccandola nei punti desiderati tramite dei "breakpoint", è possibile generare dei log di "tracing", visualizzare il valore di variabili anche complesse, seguire il flusso dell'applicazione, nonché analizzare l'intero stack delle chiamate al fine di individuare eventuali percorsi "critici".

DEMO I/O (INPUTPORT, OUTPUTPORT E TRISTATEPORT)

A titolo di esempio, vediamo ora come sviluppare da zero un'applicazione per il .NET



COLLEGAMENTI

USBizi - Header H2 - Piedini 2 e 6 → microswitch normalmente aperto
 USBizi - Header H1 - Piedino 2 → K2633 ingresso 1
 USBizi - Header H1 - Piedino 4 → K2633 ingresso 2
 USBizi - Header H1 - Piedino 6 → K2633 ingresso 3
 USBizi - Header H1 - Piedino 8 → K2633 ingresso 4
 USBizi - Header H1 - Piedino 37 → K2633 GND

Fig. 3

Micro Framework; per iniziare, ci proponiamo di sperimentare con la gestione delle porte I/O digitali.

Dopo aver creato un nuovo progetto di tipo “.NET Micro Framework Console Application”, andiamo ad editare l'interno del metodo denominato “Main”. Esso rappresenta il cosiddetto “entry-point” del nostro firmware, ossia (ad eccezione di quanto concerne gli inizializzatori o i costruttori statici di cui parleremo in futuro) la porzione di codice eseguita all'avvio del dispositivo.

Per familiarizzare con i tre tipi di porta digitale definiti nelle classi base del framework, definiamo nel nostro firmware tre diversi oggetti di tipo, rispettivamente, InputPort, OutputPort e TristatePort, come illustrato nel frammento di codice illustrato nel Listato 5. Per verificare il funzionamento dell'applicazione potremo utilizzare il debugger integrato sia nel caso in cui utilizzeremo l'emulatore, sia se, invece, sceglieremo di “programmare” il firmware appena scritto nella flash di un microcontrollore reale. Per rendere più inte-

ressante la sperimentazione mostriamo il test di quanto scritto su una piattaforma hardware reale (e non emulata) costituita dalla scheda di sviluppo USBizi prodotta da GHI Electronics; per verificare l'utilizzo di una porta di uscita sfrutteremo un LED presente sulla scheda stessa; per l'uso di una porta di ingresso utilizzeremo, invece, un microswitch collegato ad un pin con resistenza di pull-up interna. Per verificare l'uso di una porta “tri-state” controlleremo gli ingressi open-collector di una scheda per il controllo di relé a quattro canali (costituita dal kit Velleman K2633). Il sistema di test è schematizzato nella Fig. 3. Una volta effettuata la “programmazione” della flash del microcontrollore, questo effettuerà il reboot ed il nostro firmware verrà avviato. Per come abbiamo implementato la logica di controllo, all'avvio dell'applicazione il LED verrà acceso così come verrà attivato il primo relé della scheda di controllo. Da quel momento in poi, ad ogni chiusura del microswitch il led invertirà il proprio stato e verrà attivato il successivo relé in sequenza (con “rollover”). ■



Microsoft®

CONOSCIAMO .NET Micro Framework

di Gianluca Ruta

Impariamo ad utilizzare le funzioni di gestione delle porte di comunicazione della scheda di sviluppo, per instaurare collegamenti sincroni ed asincroni via seriale o bus SPI ed I²C.

Nella scorsa puntata abbiamo illustrato le caratteristiche principali della piattaforma .NET Micro Framework (giunta da poco alla sua versione 4) conosciuto un po' l'ambiente di sviluppo e fatto qualche esperimento di gestione delle porte I/O di un microcontrollore utilizzando la scheda di sviluppo USBizi, prodotta da GHI Electronics. A partire da questa puntata inizieremo ad addentrarci nei dettagli più tecnici del Framework e lo faremo analizzando l'utilizzo delle periferiche a bordo del microcontrollore relative alla gestione dei canali di comunicazione verso dispositivi esterni: per le comunicazioni di tipo asincrono prenderemo in analisi le porte seriali/UART, mentre per quelle sincrone ci occuperemo di bus SPI ed

I²C. Questa carrellata non esaurisce tutti i possibili protocolli e canali di comunicazione, ma si limita ai più diffusi. Sappiate comunque che anche 1-wire, USB e CAN possono essere utilizzati tramite il .NET Micro Framework.

COMUNICAZIONE SERIALE ASINCRONA

I moduli UART presenti all'interno del microcontrollore sovrintendono alla gestione delle comunicazioni seriali asincrone, cioè quelle in cui non è previsto un segnale di clock separato dal segnale dati, ma in cui la sincronizzazione avviene seguendo un timing condiviso tra i due interlocutori. In moltissime applicazioni, questo tipo di comunicazione serve per coordinare l'azione di più dispositivi separati che interagiscono al fine di realiz-

zare una funzione generale più complessa e articolata. Non di rado, in una comunicazione seriale con il microcontrollore l'altro interlocutore è il PC, che, con un'interfaccia RS-232 implementata direttamente nel chipset della scheda madre o tramite un "bridge" USB (tipicamente basato su chip FTDI, Prolific o Silicon Labs) può diventare parte di una soluzione embedded.

Per familiarizzare con le classi del .NET Micro Framework dedicate alla comunicazione seriale/UART, abbiamo realizzato un firmware che implementa un protocollo personalizzato per la gestione di una porta I/O. Quale hardware di riferimento abbiamo utilizzato, come già fatto nella scorsa puntata, la scheda di sviluppo USBizi prodotta da GHIElectronics, ma, a parte la mappatura del pin relativo alla porta da controllare e la porta UART da utilizzare (in un microcontrollore ne è presente molto spesso più di una), il firmware è compatibile con tutte le piattaforme hardware che implementano il .NET Micro Framework.

Dal menu "Nuovo->Progetto" di Visual Studio, scegliamo di creare un nuovo progetto di tipo *.NET Micro Framework Console Application*. L'ambiente di sviluppo creerà per noi uno

Listato 1

```
namespace UARTTest
{
    public static class Program
    {
        public static void Main()
        {
            // Corpo del metodo Main()
        }
    }
}
```

"scheletro" di applicazione, rappresentato dalla dichiarazione di una classe -denominata *Program*- che definisce il contenitore logico all'interno del quale viene effettivamente eseguito il firmware che svilupperemo. La principale responsabilità della classe *Program* è senz'altro esporre il metodo "Main()" (statico, ossia non legato alle singole istanze di questa classe, ma alla classe stessa), che rappresenta il punto di ingresso nell'esecuzione del firmware. La struttura dell'applicazione "scheletro" è rappresentata dal **Listato 1** (in cui, per brevità, sono state omesse le dichiarazioni "using"). Le funzionalità relative alla gestione delle porte UART sono implementate dalla classe *SerialPort*, definita all'interno dell'assembly *Microsoft.SPOT.hardware.SerialPort*, che dovremo referenziare come riferimento esterno del nostro progetto attraverso la funzionalità "Aggiungi Riferimento" ("Add

Listato 2

```
static void _uart_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
    if(e.EventType!=SerialData.Chars) return;

    int buflen=_uart.BytesToRead;

    byte[] rxdata=new byte[buflen];

    _uart.Read(rxdata,0,buflen);

    byte[] buf = new byte[rxdata.Length];
    int i = 0;

    foreach (byte b in rxdata)
    {
        if (b >= 48 && b <= 57 || b >= 65 && b <= 90 || b >= 97 && b <= 122 || b==10 || b==13 || b==32)
        {
            buf[i++] = b;
        }
    }

    lock (_syncLock)
    {
        _inBuffer+=new string(Encoding.UTF8.GetChars(buf));

        parseCommands();
    }
}
```

Reference”, nella versione in inglese di Visual Studio). Una volta referenziato l’assembly in questione, possiamo dichiarare il riferimento ad una istanza di *SerialPort* mediante la definizione di un “field”, ossia una variabile incapsulata all’interno dell’oggetto contenitore che ne rappresenta parte dello stato (come visto nella prima puntata del corso), con la seguente frazione di codice:

```
static SerialPort _uart = new
SerialPort("COM1", 9600);
```

Il riferimento in questione viene valorizzato contestualmente alla dichiarazione, utilizzando l’operatore “new” del C#. L’esecuzione di tale operatore provoca due effetti: la runtime del framework dapprima alloca una quantità di memoria sufficiente per contenere l’oggetto da istanziare (quantità rappresentata dalla somma dei field in esso contenuti) e successivamente invoca un suo particolare metodo denominato “costruttore”, di cui possono essere definite più forme differenziate dal numero e dal tipo di parametri accettati. È facile evincere, poi, il significato dei due parametri passati al costruttore: il primo indica il nome della porta UART da utilizzare (tipicamente alla COM1 è associato l’UART0 del microcontrollore), mentre il secondo indica il baud-rate da utilizzare.

Una volta creato l’oggetto *SerialPort*, la UART “sottostante” non è ancora pronta per la comunicazione, fino a che non viene espressamente “aperta”, mediante il metodo *Open()*. Una volta aperta la porta, la nostra applicazione (rappresentata dalla classe *Program*) dovrà “isciversi” alle notifiche di ricezione dati provenienti da essa mediante la sottoscrizione all’evento denominato “DataReceived”, mediante il comando seguente.

```
_uart.DataReceived += new SerialDataReceivedE-
ventHandler(_uart_DataReceived);
```

L’identificativo “_uart_DataReceived” fa riferimento ad un metodo della classe *Program* che assumerà quindi la funzione di gestore di evento relativamente all’evento “DataReceived”. L’implementazione specifica di tale metodo nel nostro firmware di esempio è riportata nel **Listato 2**. Dietro l’apparente

Listato 3

```
private static void parseCommands()
{
    string[] tokens=_inBuffer.Split('\r');

    if (tokens.Length == 1) return;

    for (int i=0;i<tokens.Length-1;i++)
    {
        _commandQueue.Enqueue(tokens[i]);
    }

    _inBuffer=tokens[tokens.Length-1];
}
```

complessità del metodo è possibile individuare poche semplici operazioni svolte:

- con la prima istruzione ‘if’ se l’evento non si riferisce alla ricezione di caratteri (ma di fine flusso, ossia EOF) il metodo non svolge alcuna operazione, ignorando la notifica;
- viene predisposto un buffer, rappresentato da un array di variabili di tipo “byte”, avente la capacità sufficiente a contenere la quantità di dati effettivamente ricevuta, rappresentata dal valore della proprietà “BytesToRead” dell’oggetto *SerialPort* utilizzato;
- viene popolato il buffer allocato con i dati provenienti dalla porta UART;
- viene allocato un secondo array che conterrà

Listato 4

```
bool exit = false;

while (!exit)
{
    // Matching dei comandi riconosciuti
    switch (getLastCommandLocked())
    {
        case "led on":
            _ledPort.Write(true);
            writeToUart(OK);
            break;

        case "led off":
            _ledPort.Write(false);
            writeToUart(OK);
            break;

        case "led toggle":
            _ledPort.Write(!_ledPort.Read());
            writeToUart(OK);
            break;

        case "exit":
            exit = true;
            break;

        case null:
            break;

        default:
            writeToUart(ERR);
            break;
    }
}
```

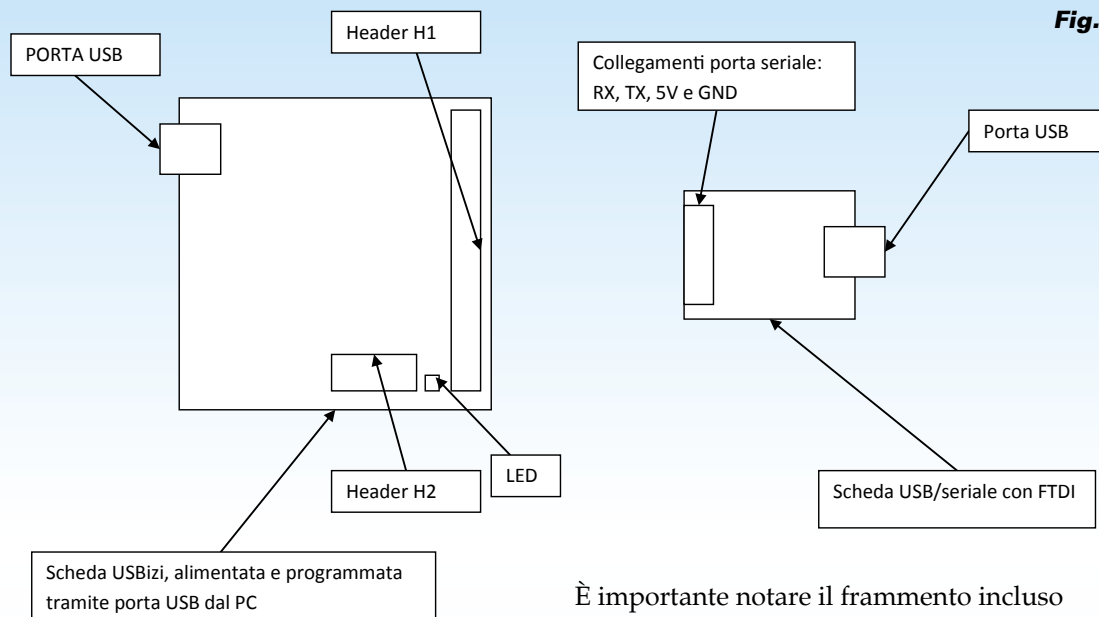


Fig. 1

COLLEGAMENTI

USB/seriale - Header H1 - Piedino 19 → FTDI RX
 USB/seriale - Header H1 - Piedino 21 → FTDI TX
 USB/seriale - Header H1 - Piedino 37 → FTDI GND
 USB/seriale - Header H1 - Piedino 39 → FTDI 5V

una copia "ripulita" del buffer precedente;

- viene scorso l'intero contenuto dell'array contenente i dati "grezzi" provenienti dall'UART, al fine di conservare solo quelli permessi dal protocollo che si vuole implementare; nel nostro caso conserveremo solo i caratteri alfanumerici, maiuscoli e minuscoli, gli spazi e i due caratteri di fine riga, ossia il "carriage return" ed il "line feed";
- i byte ricevuti e ripuliti vengono "trasformati" in una stringa, mediante la codifica UTF8 (che coincide con la ASCII per i caratteri standard) e concatenati alla stringa "_inBuffer" (definita come field della classe Program), che rappresenta il testo ancora da "interpretare" da parte del nostro processore di comandi, invocato contestualmente mediante il metodo "parseCommands()".

Listato 5

```

private static string getLastCommandLocked()
{
    lock (_syncLock)
    {
        if (_commandQueue.Count == 0) return null;

        return _commandQueue.Dequeue() as string;
    }
}

```

È importante notare il frammento incluso all'interno di una sezione "lock". Tale costrutto ha come scopo di impedire che più "thread" (ossia percorsi paralleli di esecuzione) possano eseguire delle operazioni critiche che devono essere necessariamente eseguite in maniera non concorrente. In questo caso, poiché il thread che esegue il gestore di evento "_uart_DataReceived" è diverso da quello che esegue il metodo Main() del nostro oggetto "applicazione" (ossia Program), il rischio di concorrenza di lettura/scrittura della stringa "_inBuffer" è molto concreto. Un'operazione di lettura che si dovesse sovrapporre con una di scrittura, potrebbe portare a problemi che vanno dalla mancata acquisizione di dati in ingresso alla porta UART, alla scrittura di memoria protetta; le conseguenze di ciò porterebbero molto probabilmente al crash del firmware.

L'interpretazione dei comandi implementati dal nostro protocollo avviene all'interno del metodo parseCommands(), riportato nel frammento di codice visibile nel Listato 3, dove si eseguono le seguenti operazioni:

- viene definito un array di stringhe attraverso la separazione del buffer ricevuto in "token", ossia in frasi separate dal carattere "\r" (carriage return), utilizzato come delimitatore di fine comando nel nostro protocollo;
- se non è stato individuato almeno un terminatore di comando, l'interpretazione non può essere effettuata, quindi il metodo esce senza fare altro;
- vengono accodate le stringhe complete all'interno di un oggetto di tipo "Queue", un contenitore che implementa il tipico

comportamento di una coda (ossia il cosiddetto paradigma "fifo": first-in-first-out);

- il buffer parziale viene aggiornato con l'ultimo comando non terminato.

I comandi accumulati nella coda vengono continuamente estratti ed eseguiti dal ciclo (descritto nel **Listato 4**) contenuto nel metodo `Main()`, in cui il firmware entra una volta esaurita la fase di inizializzazione. Nel **Listato 4** possiamo notare che:

- viene definita una variabile locale di tipo booleano (`true/false`) che contiene la condizione di fine ciclo (relativa al comando "exit");
- viene eseguita l'azione corrispondente al comando ricevuto, che può essere "led on", "led off", "led toggle", "exit" (il significato di questi è facilmente deducibile);
- vengono gestiti i due casi "anomali" relativi rispettivamente al comando "nullo" e "non previsto".

L'accesso al primo comando della coda viene effettuato in maniera sincronizzata ricorrendo al metodo riportato nel **Listato 5**. La scrittura sulla porta UART da parte del micro avviene tramite un accorgimento che utilizza il metodo `Write()` dell'oggetto `SerialPort`, riportato nelle seguenti righe di codice:

```
private static void writeToUart(string txt)
{
    _uart.Write(Encoding.UTF8.GetBytes(txt), 0,
    txt.Length);
}
```

Veniamo dunque al semplice circuito di test implementato (**Fig. 1**): come detto, utilizziamo la scheda USBizi prodotta da GHIElectronics, un convertitore USB/seriale con a bordo un chip FTDI (ne esistono molti in commercio, tutti praticamente equivalenti) ed un Personal Computer con 2 porte USB disponibili, usate una per programmare il chip NXP della scheda USBizi e l'altra per il dialogo seriale oggetto del test. Oltre all'ambiente di sviluppo, consigliamo di utilizzare TeraTerm, un programma di comunicazione in emulazione terminale gratuito e ben fatto: lo trovate all'indirizzo <http://www.tinyclr.com/dl>. Una volta realizzato il circuito e scritto il firmware nel

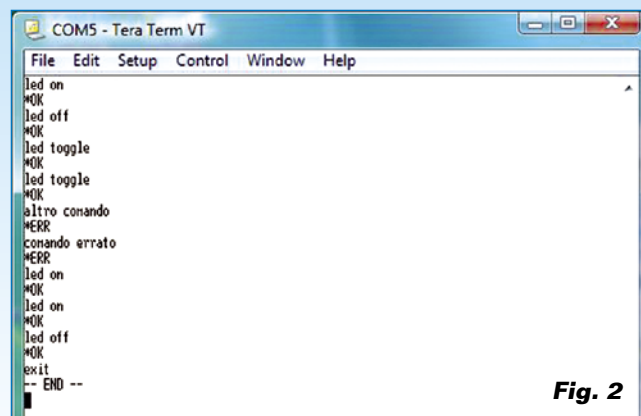


Fig. 2

chip della USBizi, potete far partire TeraTerm ed aprire una finestra di terminale sulla COM che è stata aggiunta al sistema operativo una volta connessa la porta USB al convertitore USB/seriale della FTDI. Il protocollo di comunicazione che utilizziamo è semplicissimo:

- 'led on' accende il LED
- 'led off' spegne il LED
- 'led toggle' cambia lo stato del LED
- 'exit' chiude la comunicazione

Alla ricezione di un comando riconosciuto, il microcontrollore risponde con 'ok', mentre ad ogni altro reagisce con 'err'. Una finestra TeraTerm di esempio è riportata nella **Fig. 2**.

COMUNICAZIONE SINCRONA VIA SPI

Lo standard SPI prevede una comunicazione sincrona punto-punto tra un dispositivo "master" ed uno "slave". La distinzione tra i due ruoli dipende da quale dei due interlocutori genera gli impulsi di clock. In una soluzione tipica, quale quella che prenderemo in considerazione nei prossimi esempi, il ruolo del master viene interpretato dal microcontrollore che, oltre ad avere la responsabilità della generazione del clock, produce su propria iniziativa le richieste da inviare al dispositivo slave. Dal punto di vista dei segnali, a differenza dell'UART in cui abbiamo solo il segnale RX ed il segnale TX, il protocollo SPI utilizza quattro "canali" digitali:

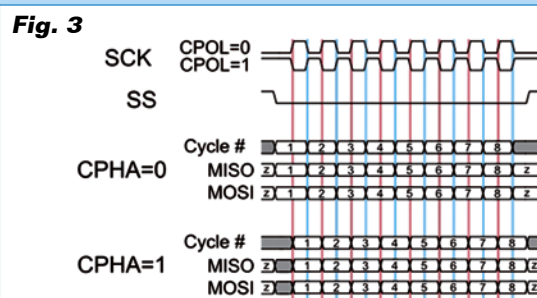


Fig. 3

Listato 6

```
public static class Program
{
    // parametri configurazione SPI
    static SPI.Configuration _spiConfig = new SPI.Configuration(
        USBizi.Pin.IO43,    // CS Pin
        false,              // CS attivo a LOW
        0,                  // CS setup time
        0,                  // CS hold time
        false,              // SCK idle a LOW
        true,               // campionamento MISO/MOSI su fronti di salita di SCK
        250,                // clock a 250 KHz
        SPI.SPI_module.SPI1);

    // istanza bus SPI
    static SPI _spi = new SPI(_spiConfig);
}
```

- **MOSI - Master Output Slave Input**, rappresenta il segnale che veicola l'informazione (sotto forma di bit, ovviamente) che va dal dispositivo master alla periferica slave;
- **MISO - Master Input Slave Output**, rappresenta il segnale che veicola l'informazione dalla periferica slave al dispositivo master;
- **SCK** - è il segnale di clock. Tipicamente il campionamento di MISO e MOSI avviene sul fronte di salita di questo segnale;
- **SSEL - Slave Select** (indicato spesso anche come CS, Chip Select), tipicamente attivo a livello logico basso, serve per attivare la comunicazione nei confronti di uno specifico slave qualora ce ne sia più di uno collegato sullo stesso bus SPI.

Come esempio, abbiamo realizzato un firmware in grado di colloquiare su bus SPI

Listato 7

```
public class MCP3201_ADC
{
    // riferimento al bus SPI da utilizzare
    SPI _spi = null;

    // costruttore che accetta in ingresso un bus SPI
    pre-inizializzato public MCP3201_ADC(SPI spi)
    {
        _spi = spi;
    }

    // proprietà che restituisce la tensione misurata dall'adc
    public float Vin
    {
        get
        {
            ushort[] tx = new ushort[1] { 0 };
            ushort[] rx = new ushort[1] { 0 };

            _spi.WriteRead(tx, rx);

            return 3.3f * ((float)((rx[0] / 2) & 0x1fff)) / 4096.0f;
        }
    }
}
```

con due diversi dispositivi: un ADC ed un I/O Expander. Come ADC abbiamo usato un MCP3201 della Microchip, in grado di campionare segnali di ingresso alla frequenza di 100 kps (100.000 campioni al secondo, sufficienti per campionare un segnale con una massima occupazione di banda di 50 kHz), con una risoluzione di quantizzazione massima di 12 bit.

Come I/O Expander, invece, abbiamo utilizzato un MCP23S08, sempre prodotto da Microchip, caratterizzato dalla possibilità di gestire 8 GPIO, ciascuno configurabile come ingresso o uscita, insieme ad un sistema di interrupt relativi a notifiche di variazioni degli ingressi digitali. La scelta dei componenti non è casuale: entrambi sono installati a bordo di una scheda dimostrativa prodotta da Microchip e denominata "PICkit Serial SPI Demo Board", in cui, oltre ai dispositivi citati, sono presenti anche una EEPROM, un sensore di temperatura, un DAC, un potenziometro digitale ed un amplificatore a guadagno programmabile, tutti connessi al medesimo bus SPI.

La scheda in questione è progettata con il segnale di Slave Select condiviso fra tutte le periferiche; è previsto un jumper per veicolare detto segnale verso il dispositivo che si vuole utilizzare. Qualora si volessero impiegare più dispositivi all'interno di una stessa applicazione, sarebbe necessario collegare ai rispettivi pin di Slave Select altrettante uscite digitali del dispositivo master.

Analogamente a quanto visto per l'esempio di firmware relativo alla comunicazione UART, il progetto creato all'interno di Visual Studio è di tipo "Console Application" e include i riferimenti agli stessi assembly esterni già utilizzati nell'esempio precedente. L'utilizzo della comunicazione SPI nel .NET Micro Framework avviene tramite la classe "Microsoft.SPOT.

Listato 8

```

public class MCP23S08_IOEXP
{
    // Indici registri
    const byte REGISTER_IODIR_ADDR = 0x0;
    const byte REGISTER_GP_INT_EN_ADDR = 0x2;
    const byte REGISTER_INT_FLAG_ADDR = 0x7;
    const byte REGISTER_INT_CAP_ADDR = 0x8;
    const byte REGISTER_GPIO_ADDR = 0x9;

    // indirizzo logico sul bus
    const byte MCP23S08_A0 = 0x0;
    const byte MCP23S08_A1 = 0x0;

    // indirizzi logici di lettura e scrittura
    const byte MCP23S08_R_ADDRESS = ((1 << 6) | (MCP23S08_A1 << 2) | (MCP23S08_A0 << 1) | 0x1);
    const byte MCP23S08_W_ADDRESS = ((1 << 6) | (MCP23S08_A1 << 2) | (MCP23S08_A0 << 1) | 0x0);

    // riferimento al bus SPI da utilizzare
    SPI _spi = null;

    // costruttore che accetta in ingresso un bus SPI pre-inizializzato
    public MCP23S08_IOEXP(SPI spi)
    {
        _spi = spi;
    }

    // imposta i GPIO come ingressi (bit=1) o uscite (bit=0)
    public void SetIODirection(byte input_mask)
    {
        byte[] tx = new byte[3] { MCP23S08_W_ADDRESS, REGISTER_IODIR_ADDR, input_mask };

        _spi.Write(tx);
    }

    // rileva la direzione (ingresso o uscita) dei GPIO
    public byte GetIODirection()
    {
        byte[] tx = new byte[2] { MCP23S08_R_ADDRESS, REGISTER_IODIR_ADDR };
        byte[] rx = new byte[2] { 0, 0 };

        _spi.WriteRead(tx, rx);

        return rx[1];
    }

    // imposta lo stato dei GPIO (ha un effetto solo per quelli di uscita)
    public void WriteIO(byte output_mask)
    {
        byte[] tx = new byte[3] { MCP23S08_W_ADDRESS, REGISTER_GPIO_ADDR, output_mask };

        _spi.Write(tx);
    }

    // rileva lo stato di tutti i GPIO
    public byte ReadIO()
    {
        byte[] tx = new byte[3] { MCP23S08_R_ADDRESS, REGISTER_GPIO_ADDR, 0 };
        byte[] rx = new byte[3] { 0, 0, 0 };

        _spi.WriteRead(tx, rx);

        return rx[2];
    }
}

```

```

return 3.3f * ((float)((rx[0] / 2) & 0x1fff)) /
4096.0f;

```

dove viene ricavata la tensione di ingresso a partire dal valore a 12 bit restituito sul segnale MISO da parte del componente una volta che il master abbia generato 16 impulsi di clock.

Il calcolo effettuato tiene conto di una tensione di riferimento per l'ADC di 3,3 V e di una quantizzazione a 12 bit ($2^{12}=4096$).

Per incapsulare la logica della comunicazione all'interno di una classe che astragga le funzionalità del componente, è possibile realizzare una sorta di "driver" modulare, che permetta il riutilizzo della porzione di firmware responsabile della comunicazione e del protocollo. L'implementazione di tale driver è rappresentata dal **Listato 7**.

In questo modo, il programma principale potrà rilevare la tensione di ingresso all'ADC come illustrato nelle seguenti righe di codice:

```
float vin;
```

```

MCP3201_ADC adc
= new MCP3201_
ADC(_spi); // _spi
già inizializzato
vin=adc.Vin;
// in "vin" metto la
tensione misurata

```

Analogamente, è possibile implemen-

tare un componente software che astragga le funzionalità dell'I/O Expander nel modo illustrato dal **Listato 8**, che consigliamo di seguire consultando nel contempo il datasheet dell'MCP23S08, allo scopo di meglio comprendere il protocollo di comunicazione utilizzato da quest'ultimo.

Anche in questo caso, l'utilizzo da parte del programma principale è molto semplificato rispetto all'interfacciamento diretto con il bus SPI. Il **Listato 9** illustra una sessione di utilizzo del componente in questione, che, come vedete, è molto semplificata.

Veniamo al circuito di test e alle prove pratiche del firmware implementato: nella **Fig. 4** vedete il semplice circuito da realizzare, che prevede ancora la scheda USBizi ed il convertitore USB/seriale, oltre alla scheda di sviluppo SPI Demo Board di Microchip. Il firmware che abbiamo presentato consente la comunicazione attraverso porta seriale/USB del valore di tensione convertito dal componente ADC (la scheda Microchip a tal scopo ha un trimmer che consente di regolare la tensione in ingresso al convertitore per sperimentare la corretta lettura del dato), pertanto con il software TeraTerm potremo leggere il valore di tensione rilevato. Per i test con l'IO Expander, invece, la scheda di Microchip prevede un LED per ogni uscita parallela del MCP23S08 e il firmware fa accendere in sequenza i primi quattro LED mentre configura come ingressi digitali i restanti quattro; ancora con il software TeraTerm potremo vedere la conversione in numero decimale del numero binario a quattro bit che impostiamo su tali ingressi. Ad esempio collegando all'alimentazione positiva l'ingresso che viene interpretato come bit meno significativo e quello corrispondente al bit più significativo otterremo il numero 9.

COMUNICAZIONE SINCRONA VIA I²C

Il terzo ed ultimo modello di comunicazione con dispositivi esterni che prendiamo in esame è il protocollo I²C (acronimo di *Inter Integrated Circuit*), sviluppato dalla Philips nel 1982 e divulgato circa 10 anni dopo. Come nel caso dell'SPI, anche il protocollo I²C prevede l'utilizzo di un segnale di clock e anche in questo caso è possibile individuare i ruoli di master e di slave in base a quale dei dispositivi presenti sul cana-

Listato 9

```
MCP23S08_IOEXP ioexp = new MCP23S08_IOEXP(_spi);

ioexp.SetIODirection(0xF0);    // 4 ingressi e 4 uscite

Debug.Assert(_ioexp.GetIODirection() == 0xF0, "IOEXP SetIODirection() FALLITO!");

// impostazione tutte uscite ON
_ioexp.WriteIO(0x0F);

// lettura stati ingressi
byte stati_ingressi = _ioexp.ReadIO();
```

le genera effettivamente tale segnale ma, a differenza del caso SPI, la comunicazione I²C permette di utilizzare un bus costituito da due soli segnali, per di più condivisi da tutti i dispositivi presenti sul bus. Utilizzando infatti degli ingressi di tipo "open-drain" ed una coppia di resistenze di pull-up sulle due linee dati (SDA) e clock (SCL) è possibile realizzare dei canali di comunicazione multi-slave, in cui un master comunica con slave differenti o addirittura multi-master. La presenza di più dispositivi su un'unica linea dati comporta però ovviamente una maggiore complessità del protocollo e l'impossibilità di una comunicazione full-duplex: ciascun dispositivo slave è identificato univocamente sul bus da un indirizzo logico (tipicamente a 7 bit) e le "transazioni" di comunicazione avvengono secondo uno schema simplex in cui si avviano sul bus scritture e letture (viste dalla prospettiva del master).

Le principali classi responsabili della gestione della comunicazione I²C nel .NET Micro Framework sono "I2CDevice", che rappresenta il riferimento al bus I²C su cui si effettueranno le transazioni, "I2CDevice.Configuration", corrispondente ad un set di parametri di configurazione utilizzato per comunicare con un determinato slave (ossia indirizzo e frequenza del clock) e "I2CTransaction" che -attraverso

Listato 10

```
I2CDevice.Configuration config=new I2CDevice.Configuration(0x49, 100);
I2CDevice i2c=new I2CDevice(config);

I2CDevice.I2CTransaction[] tran = new I2CDevice.I2CTransaction[2];

byte[] tx = new byte[1] { 0 }; // tipicamente questo è l'indice di un registro
byte[] rx = new byte[2] { 0, 0 };

tran[0] = _i2c.CreateWriteTransaction(tx);
tran[1] = _i2c.CreateReadTransaction(rx);

int retVal = _i2c.Execute(tran, 1000); // Timeout a 1 secondo

/*ora possiamo leggere la risposta nel buffer rx[*]/
```

Listato 11

```

public class MCP9800_TemperatureSensor
{
    I2CDevice.Configuration _config;
    I2CDevice _i2c;

    public MCP9800_TemperatureSensor(I2CDevice I2CBus, ushort I2CAddress, int I2CClockFrequencyKHz)
    {
        // Memorizzo la configurazione passata come parametro
        _i2c = I2CBus;
        _config = new I2CDevice.Configuration(I2CAddress, I2CClockFrequencyKHz);
    }

    public MCP9800_TemperatureSensor(I2CDevice I2CBus)
    {
        // Uso la configurazione del bus passato come parametro
        _i2c = I2CBus;
        _config = I2CBus.Config;
    }

    public float Temperature
    {
        get
        {
            // Imposto la configurazione del bus
            _i2c.Config = _config;

            // Creo 2 transazioni, una di scrittura (indice registro da leggere)
            // ed una di lettura (2 byte, contenente la temperatura corrente)
            I2CDevice.I2CTransaction[] tran = new I2CDevice.I2CTransaction[2];

            byte[] tx = new byte[1] { 0 }; // indice registro temperatura=0
            byte[] rx = new byte[2] { 0, 0 };

            tran[0] = _i2c.CreateWriteTransaction(tx);
            tran[1] = _i2c.CreateReadTransaction(rx);

            int retval = _i2c.Execute(tran, 1000); // Timeout a 1 secondo

            if (retval != tx.Length + rx.Length) throw new ApplicationException("Transazione I2C FALLITA");

            // calcolo temperatura in °C
            if ((rx[0] & 0x80) != 0)
            {
                // se sotto 0°C applico il complemento a 2
                return (~rx[0] * 256 + ~rx[1]) / 256.0f;
            }
            else
            {
                return (rx[0] * 256 + rx[1]) / 256.0f;
            }
        }
    }
}

```

le due classi derivate "I2CWriteTransaction" e "I2CReadTransaction"- rappresenta un messaggio inviato rispettivamente dal master allo slave indirizzato o viceversa.

Una tipica sessione di comunicazione I²C avviene secondo quanto descritto nel **Listato 10**. Giunti a questo punto possiamo notare che:

- viene utilizzato un oggetto di tipo I2CDevice.Configuration inizializzato con l'indirizzo (a 7 bit) 0x49 e con la frequenza di clock di 100 kHz e con questo oggetto poi viene inizializzato il riferimento al bus I²C;
- viene definito un array di 2 transazioni I²C, un oggetto costituito da due messaggi distinti (uno di richiesta ed uno di risposta);

- viene valorizzato il buffer che costituirà l'effettiva richiesta e quello che ospiterà la transazione di risposta;
- l'array di transazioni viene popolato rispettivamente da una transazione di scrittura ed una di lettura
- viene effettuata la comunicazione vera e propria sul bus; il valore restituito rappresenta l'effettivo numero di byte veicolati dal bus durante la transazione.

Ci proponiamo ora di utilizzare il supporto al protocollo I²C del .NET Micro Framework per comunicare con due diversi dispositivi: il MCP9800, un sensore di temperatura a 12 bit,

Listato 14

```
// Configurazione default bus I2C (verrà sovrascritta da quella del dispositivo)
I2CDevice.Configuration configEmpty = new I2CDevice.Configuration(0x00, 10);

// Bus I2C
I2CDevice i2c = new I2CDevice(configEmpty);

// Sensore
TC1321_DAC dac = new TC1321_DAC(i2c, 0x48, 10);

// Imposta l'uscita a 1.5V
dac.SetOutVoltage(1.5f);
```

e il TC1321, un DAC (ossia un convertitore da digitale ad analogico) a 10 bit, entrambi prodotti da Microchip e riportati, insieme ad altri dispositivi, all'interno della scheda dimostrativa denominata "PICKit Serial I2C Demo Board". Analogamente a quanto fatto per l'esempio precedente, abbiamo realizzato due apposite classi "driver" che semplifichino l'utilizzo dei suddetti dispositivi da parte di un'applicazione client. Il data-sheet del sensore di temperatura MCP9800 riporta il protocollo relativo alla funzionalità di lettura della temperatura corrente come sequenza di due transazioni I²C: la prima, in scrittura, seleziona il registro da leggere nella successiva transazione, di lettura, in cui il dispositivo invia al microcontrollore due byte che esprimono in forma "complemento a 2" la temperatura rilevata, con una risoluzione di 0,0625 °C ed un'accuratezza di 0,5 °C. L'implementazione del protocollo descritto viene

che utilizzino le funzionalità del dispositivo) è illustrato nel **Listato 12**. Analogamente, è possibile realizzare un driver che piloti il dispositivo DAC di cui sopra attraverso una classe definita nel **Listato 13**.

Di nuovo, l'utilizzo di tale driver da parte del programma principale "nasconde" i dettagli della comunicazione sottostante, permettendo a chi sviluppa il firmware di concentrarsi solamente sulla logica applicativa, come illustrato nel **Listato 14**.

Ora veniamo ai test che si possono effettuare con il firmware da noi esaminato: nella **Fig. 5** viene presentato il sistema di test, che stavolta utilizza la scheda Tahoe II prodotta da Device Solutions, affiancata dalla scheda di Microchip I²C Demo Board, che ospita il componente DAC e il sensore di temperatura. Stavolta non utilizziamo una comunicazione seriale dei dati; potremo leggere la temperatura rilevata direttamente nell'ambiente di

sviluppo Visual Studio nella finestra 'Debug', mentre il segnale di tensione che impostiamo (variabile tra 0 e 2 volt) sul DAC dovremo ovviamente andarlo a rilevare con un voltmetro collegato ai piedini GND e DAC-out della Demo Board di Microchip.

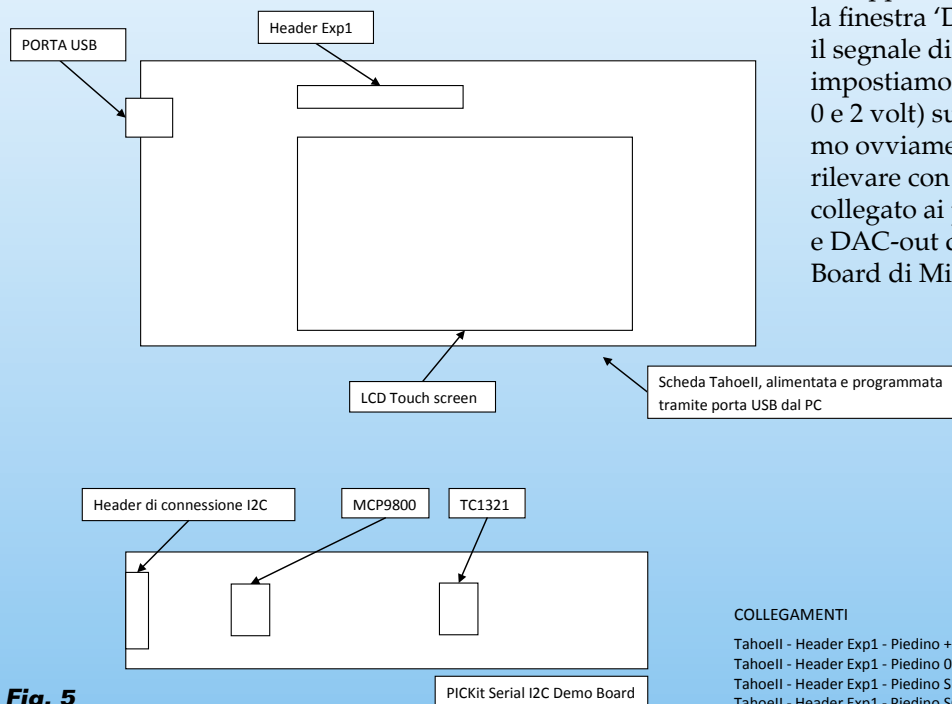


Fig. 5

COLLEGAMENTI

TahoeII - Header Exp1 - Piedino +5V → I2C Demo Board +V
 TahoeII - Header Exp1 - Piedino 0V → I2C Demo Board GND
 TahoeII - Header Exp1 - Piedino SDA → I2C Demo Board SDA
 TahoeII - Header Exp1 - Piedino SCL → I2C Demo Board SCL



Microsoft®

CONOSCIAMO .NET Micro Framework

di Lorenzo Maiorfi
e Gianluca Ruta

Innovactive Engineering

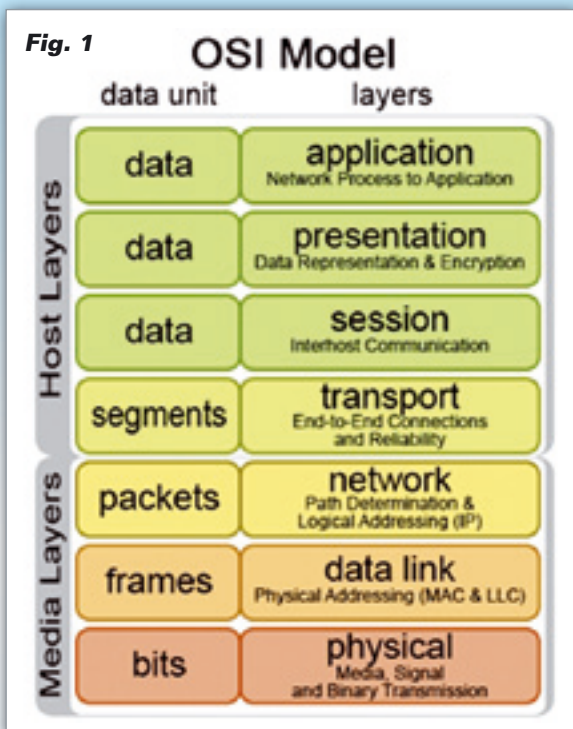
Impariamo ad usare le funzionalità di rete, testandole in pratica con la scheda di sviluppo Tahoe II e realizzando un'applicazione che effettua la ricerca di messaggi su Twitter ed un Server Web in grado di pubblicare informazioni su Internet.

Dopo aver analizzato la piattaforma per lo sviluppo di applicazioni per microcontrollori di Microsoft sotto l'aspetto sia delle caratteristiche generali, sia degli aspetti tecnici più particolari (inerenti l'utilizzo delle periferiche di comunicazione seriale di un microcontrollore), in questa puntata del corso su .NET Micro Framework ci occupiamo di networking in generale ed in particolare dell'utilizzo dello stack TCP/IP per comunicazioni su rete Ethernet. Per farlo, oltre ad una scheda di sviluppo con a bordo un microcontrollore che ospita le funzionalità del .NET Micro Framework, utilizziamo un Personal Computer, con opportuno software realizzato in linguaggio C# mediante il "fra-

tello maggiore" del Micro Framework, che è appunto Microsoft .NET Framework.

NETWORKING

Mettendo a confronto le possibilità offerte dalle diverse famiglie di microcontrollori disponibili sul mercato, si nota una divisione piuttosto netta nel supporto alle funzionalità di rete tra i microcontrollori a 8 bit e quelli a 16-32 bit; la principale motivazione di ciò risiede nella complessità dei protocolli che compongono il cosiddetto "stack" di rete. Ciascuno strato di tale "pila" (il modello utilizzato per descrivere la struttura "a layer" definita dallo standard "ISO/OSI") è responsabile del supporto allo strato sovrastante e può utilizzare solamente i



servizi forniti dallo strato sottostante. Facendo riferimento al diagramma "OSI Model" riportato in Fig. 1, è possibile, ad esempio, individuare un livello "fisico" in cui il protocollo gestisce direttamente i segnali digitali veicolati dal mezzo utilizzato (il cavo bipolare, ad esempio), un livello "data link", in cui il protocollo gestisce l'informazione

circa l'indirizzo dei nodi all'interno di una rete e così via, salendo di strato in strato, fino a raggiungere i layer relativi ai protocolli di alto livello, in grado di fornire servizi di comunicazione molto evoluti alle applicazioni che ne fanno uso. Sebbene il modello in questione possa rappresentare molti tipi diversi di "composizione" di protocolli, la sua implementazione più diffusa è senza dubbio quella relativa allo stack TCP/IP. Tale pila di protocolli definisce molte possibili varianti per ciascun livello, con l'indubbio vantaggio di svincolare il firmware dell'applicazione che stiamo sviluppando dal contesto richiesto da ciascuno specifico scenario. In particolare, il modello noto come OSI/IP prevede le seguenti combinazioni di layer, elencati dal livello 1 (fisico) al livello 7 (applicativo):

1. **Physical Layer:** RS-232, V.35-V.90 (utilizzato nei modem), DSL, 802.11 PHY (utilizzato dal Wi-Fi), USB, Bluetooth, Ethernet;
2. **Data Link Layer:** ARP (per la risoluzione degli indirizzi hardware dei nodi), PPP (utilizzato nel collegamento ad Internet via modem, tramite la mediazione di un provider), Framing Ethernet;
3. **Network Layer:** IP, ICMP (il protocollo reso famoso dal "ping", che utilizza a sua

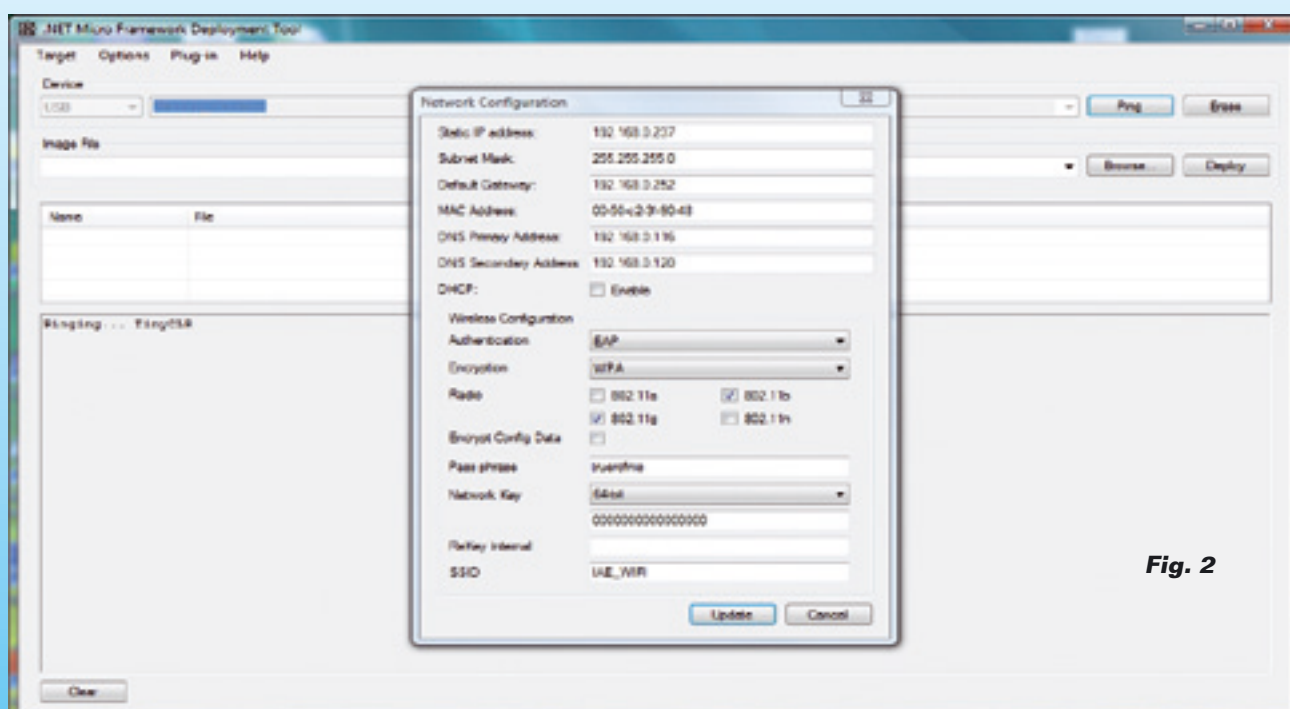


Fig. 2

Listato 1

```

namespace TCPSocketServer
{
    class Program
    {
        static void Main(string[] args)
        {
            TcpListener listener = new TcpListener(new IPEndPoint(IPAddress.Any, 1234));

            listener.Start();

            Console.WriteLine("Server started...");

            while (true)
            {
                TcpClient client = listener.AcceptTcpClient();

                Console.WriteLine(string.Format("{0} Connected!", client.Client.RemoteEndPoint));

                using (StreamReader sr = new StreamReader(client.GetStream()))
                {
                    string s = sr.ReadLine();

                    Console.WriteLine(string.Format("Received: {0}", s));

                    using (StreamWriter sw = new StreamWriter(client.GetStream()))
                    {
                        switch (s.ToLower())
                        {
                            case "exit":
                                listener.Stop();
                                break;

                            case "?":
                                sw.WriteLine("TCPSocketServer 1.0");
                                break;
                        }
                    }
                }
            }
        }
    }
}

```

volta il sotto-protocollo ECHO), AppleTalk, IPX (disponibile nelle vecchie reti Novell, oramai obsolete);

4. **Transport Layer:** TCP, UDP e simili;
5. **Session Layer:** Named Pipes, NetBIOS;
6. **Presentation Layer:** SSL, MIME, TLS;
7. **Application Layer:** HTTP, SMTP, Telnet, FTP, DNS, DHCP, e via di seguito.

Nella nostra sperimentazione con il .NET Micro Framework, faremo riferimento ad uno degli scenari più comuni, in cui il dispositivo che programmeremo utilizzerà una connessione TCP/IP su rete Ethernet. La configurazione dei parametri di rete del dispositivo può essere effettuata tramite l'utility MFDeploy, installata dal setup del .NET Micro Framework SDK, come visibile nella Fig. 2.

SOCKET

Il modello più utilizzato nella realizzazione di una comunicazione via rete tra dispositivi (o

tra più applicazioni all'interno dello stesso dispositivo) è senza dubbio il "Socket"; questo oggetto rappresenta letteralmente un "connettore" logico che permette ad un'applicazione di scambiare informazioni con un'altra e tipicamente funziona all'interno di un altro dispositivo presente sulla rete. Vale subito la pena di notare che l'uso dei Socket permette la comunicazione tra due qualsiasi "nodi" all'interno di una rete IP, sia essa una rete locale LAN, una rete geografica WAN o anche Internet.

In una comunicazione via Socket si individuano due ruoli: il "client" è rappresentato dal nodo che prende l'iniziativa nello stabilire una nuova connessione; il "server" utilizza invece un Socket che viene aperto "in ascolto", ossia in attesa di richieste di connessione da parte di un client. Ciascuno dei due estremi della comunicazione è in realtà rappresentato da una coppia di informazioni: un indirizzo IP, tipicamente espresso come quaterna di nu-

Listato 2

```

public static void Main()
{
    // Apertura connessione TCP al server "pip" su porta IP 1234
    using (Socket socket = Connect("pip.main.innovactive.it", 1234))
    {
        byte[] tx = Encoding.UTF8.GetBytes("?r\n");

        // Invio al socket la stringa "?+<invio>"
        socket.Send(tx);

        // Attendo fino a che non arriva risposta...
        while (socket.Available == 0)
        {
            Thread.Sleep(100);
        }

        string rxstring = string.Empty;

        // fino a che il socket è leggibile...
        while(socket.Poll(-1, SelectMode.SelectRead))
        {
            byte[] rx=new byte[1024];

            // viene popolato il buffer...
            socket.Receive(rx);

            // concateniamo alla stringa ricevuta fino a qui...
            rxstring += new string(Encoding.UTF8.GetChars(rx));

            // se non ci sono dati disponibili in ingresso esco dal ciclo
            if (socket.Available == 0) break;
        }

        // La stringa ricevuta finora è...
        Debug.Print(rxstring);
    }
}

static Socket Connect(string server, int port)
{
    // Risolvo il nome DNS del server
    IPHostEntry host = Dns.GetHostEntry(server);

    // Creo un socket TCP
    Socket sock = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

    // connetto il socket ad uno degli indirizzi del server (il primo),
    // alla porta IP richiesta
    sock.Connect(new IPEndPoint(host.AddressList[0], port));

    // una volta connesso, restituisco il socket aperto al chiamante...
    return sock;
}

```

meri 0÷255 separati da un punto (ad esempio 88.62.138.1) ed una porta IP, cioè un numero che insieme all'indirizzo IP identifica univocamente un'istanza di Socket attiva. Nel .NET Micro Framework tale coppia di informazioni prende il nome di "IPEndPoint"; è importante notare che, mentre nel Socket server la porta dell'IPEndPoint in ascolto deve essere nota anche al client affinché questo possa stabilire una connessione, la porta dell'IPEndPoint del client viene di solito scelta casualmente (tra quelle non riservate, tipicamente comprese nell'intervallo 0÷1024) in fase di connessione. Una volta stabilita la connessione tra i due Socket, lo scambio di informazioni avviene in maniera full-duplex, utilizzando i metodi Send() e Receive(). Nel primo esempio che vediamo, prendiamo in esame la situazione in cui un dispositivo embedded effettua una connessione ad un servizio di rete esposto da un'apposita applicazione Windows, realizzata come semplice "Console" (ossia in cui l'interfaccia utente è rappresentata da una semplice finestra terminale simile al prompt dei comandi di sistema) realizzata, in questo caso, con il Framework .NET "senior", ossia quello per PC. Iniziamo proprio dal codice di quest'ultimo, riportato nel **Listato 1**.

Da questo listato possiamo notare che:

- viene creata un'istanza della classe TcpListener, inizializzata con un IPEndPoint costituito dallo pseudo-indirizzo IP "Any" (il quale consente di ricevere connessioni

da uno qualsiasi degli indirizzi IP presenti sulla macchina) e dalla porta "1234"; le classi TcpListener e TcpClient semplificano lo sviluppo, rispettivamente di server TCP e client TCP, nelle applicazioni basate sul Framework .NET "senior" (tali classi fanno da involucro ai Socket sottostanti, esposti comunque da apposite proprietà);

- la chiamata **AcceptTcpClient()** è bloccante e ritorna solo nel momento in cui un client stabilisce effettivamente una connessione; l'indirizzo e la porta del client sono visualizzate nella finestra terminale della nostra applicazione;

- viene creato un oggetto di tipo *StreamReader*, inizializzato a partire dallo stream relativo al canale di comunicazione attivato; la classe *Stream* nel Framework.NET e tutte le sue classi derivate gestiscono il flusso di comunicazione nei vari contesti in cui questo è richiesto (ad esempio per la lettura/scrittura di file su disco) mentre le classi *StreamReader* e *StreamWriter* consentono, rispettivamente, l'estrazione e l'immissione di dati all'interno di un canale rappresentato da un determinato *Stream*;
- viene letta dal client una stringa costituita da tutti i caratteri ricevuti prima di un terminatore di riga;
- utilizzando uno *StreamWriter*, viene inviata al client la risposta relativa al comando ricevuto; in questo esempio il comando "exit" chiude il Socket server, mentre il comando "?" restituisce al client la stringa "TCPSocketServer 1.0".

Per sviluppare un firmware in grado di fare da client per il server desktop appena illustrato, abbiamo utilizzato la scheda *Tahoe II* prodotta da Device Solutions; tale unità integra infatti anche un transceiver Ethernet 10 Mbit/s (il noto ENC28J60), connesso internamente al modulo SPI1.

Il codice relativo al client realizzato con il .NET Micro Framework è simile alla controparte "server", sebbene in questo contesto non siano disponibili le classi di supporto *TcpListener* e *TcpClient*; tale codice è riportato nel **Listato 2**.

In questo secondo frammento di codice possiamo notare che:

- attraverso la chiamata ad un metodo "Connect" viene stabilita la connessione al listener TCP in ascolto sulla porta 1234 dell'host identificato dal nome DNS "pip.main.innovactive.it";
- viene inviata la stringa relativa all'unico comando riconosciuto dal server, ossia "?" + <invio>;
- dopo aver aspettato la disponibilità di dati dal socket, viene letta la risposta proveniente dal server, fino a che non ci sono più dati disponibili;
- la stringa ricevuta viene visualizzata nella finestra di debug di Visual Studio.

Riguardo al metodo *Connect()*, a parte notare le righe relative alla creazione e connessione di un Socket in maniera simile a quanto visto negli altri esempi, vale la pena di soffermarci sul fatto che, attraverso l'esecuzione del metodo statico **GetHostEntry()** della classe *Dns*, viene valorizzato un oggetto di tipo *IPHostEntry*. La classe *Dns* è responsabile della fruizione del servizio DNS così come configurato nel dispositivo. Il servizio DNS utilizza il protocollo omonimo, ossia uno dei protocolli applicativi dello stack IP/OSI, al fine di espletare il processo noto come "risoluzione dei nomi"; ciò perché all'interno di una rete IP che abbia molti nodi è frequente che per identificare un determinato nodo si preferisca fare riferimento ad una stringa più familiare piuttosto che ad una quaterna di numeri. Tale necessità è stata imposta dalla diffusione delle reti geografiche, prima fra tutte, ovviamente, Internet. Il servizio DNS si occupa -in sintesi- della traduzione di nomi in indirizzi (e viceversa) tramite la consultazione di un database interrogato attraverso il protocollo DNS stesso; tale database non è però gestito da un unico provider, ma è distribuito su tutti i server DNS presenti nel mondo, ciascuno con la propria "partizione" di competenza. La natura gerarchica dei nomi DNS ha proprio lo scopo di instradare una richiesta di risoluzione dei nomi verso il server DNS che la sappia gestire nel più breve tempo possibile. Quando il nostro firmware di esempio esegue la chiamata a **GetHostEntry("pip.main.innovactive.it")**, il sistema operativo (la runtime del .NET Micro Framework, in questo caso) effettua una richiesta al primo dei server DNS



Fig. 3

Listato 3

```

using System;
using Microsoft.SPOT;
using System.Net.Sockets;
using System.Net;
using System.Threading;
using System.Text;
using Microsoft.SPOT.Hardware;
using DeviceSolutions.SPOT.Hardware;

namespace TestTCPServer
{
    public static class Program
    {
        // Uscita digitale collegata al relé da attivare
        static OutputPort _relay = new OutputPort(Meridian.Pins.GPIO1, false);

        public static OutputPort Relay
        {
            get { return Program._relay; }
        }

        public static void Main()
        {
            // Creo il socket server, in ascolto sulla porta TCP 1234
            Socket serverSocket = new Socket(
                AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

            IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 1234);

            serverSocket.Bind(localEndPoint);

            serverSocket.Listen(int.MaxValue);

            while (true)
            {
                // Resto in attesa del collegamento di un client...
                Socket clientSocket = serverSocket.Accept();

                // Creo un oggetto che processi la richiesta...
                new ClientSocketProcessor(clientSocket, true);
            }
        }
    }

    internal class ClientSocketProcessor
    {
        Socket _clientSocket;

        // Il costruttore memorizza il socket client e lancia un worker thread se richiesto
        public ClientSocketProcessor(Socket ClientSocket, bool async)
        {
            _clientSocket = ClientSocket;

            if (async)
            {
                new Thread(processRequest).Start();
            }
            else
            {
                processRequest();
            }
        }

        void processRequest()
        {

```

impostati in fase di configurazione della rete; il server DNS contattato può contenere nel proprio database l'informazione richiesta, oppure no: nel primo caso la richiesta del client viene immediatamente esaudita, mentre nel secondo il server DNS diventa "client" del

server corrispondente al dominio di primo livello relativo al nome richiesto (".it", nel nostro esempio). Quest'ultimo viene quindi contattato in merito all'indirizzo del server DNS che risolve i nomi relativi al dominio di secondo livello ("innovactive.it", nel nostro

```

string rxstring = string.Empty;
// con il socket client...
using (_clientSocket)
{
    byte[] rx = new byte[1024];

    // fino a che è leggibile...
    while (_clientSocket.Poll(-1, SelectMode.SelectRead))
    {
        if (_clientSocket.Available == 0) break;

        int bytesread=_clientSocket.Receive(rx, _clientSocket.Available, SocketFlags.None);

        // concatenano le stringhe ricevute con quella corrente
        rxstring += new string(Encoding.UTF8.GetChars(rx));

        // se trovo il fine riga esco dal ciclo e processo il comando
        if (rxstring.Length > 2 &&
            rxstring[rxstring.Length - 2] == '\r' &&
            rxstring[rxstring.Length - 1] == '\n') break;
    }

    // processing dei comandi riconosciuti
    switch (rxstring.ToLower())
    {
        // comando "?"
        case "?\n\r\n":
        case "?\r\n":
            // mando al client la stringa di "benvenuto"
            _clientSocket.Send(Encoding.UTF8.GetBytes(".NET MF TCP Server 1.0\r\n"));
            break;

        case "on\n\r\n":
        case "on\r\n":
            // attivo il relé
            Program.Relay.Write(true);

            // mando al client la stringa di "feedback"
            _clientSocket.Send(Encoding.UTF8.GetBytes("Relay ON\r\n"));
            break;

        case "off\n\r\n":
        case "off\r\n":
            // disattivo il relé
            Program.Relay.Write(false);

            // mando al client la stringa di "feedback"
            _clientSocket.Send(Encoding.UTF8.GetBytes("Relay OFF\r\n"));
            break;

        case "toggle\n\r\n":
        case "toggle\r\n":
            // scambio lo stato del relé
            Program.Relay.Write(!Program.Relay.Read());

            // mando al client la stringa di "feedback"
            _clientSocket.Send(
                Encoding.UTF8.GetBytes("Relay " +
                    (Program.Relay.Read() ? "ON" : "OFF") + "\r\n"));
            break;
    }
}
}
}
}
}

```

esempio) e così via, fino a trovare il server DNS in grado di risolvere il nome dell'host completo.

Da un punto di vista pratico, realizzare questo test è molto semplice, anche se richiede ovviamente la presenza di una rete Ether-

net alla quale collegare il modulo Tahoe II (si veda la **Fig. 3**). Dopo aver correttamente impostato nel codice l'indirizzo al quale il dispositivo si deve connettere (quello che nel nostro esempio era "pip.main.innovactive.it") ed aver mandato in esecuzione il servizio

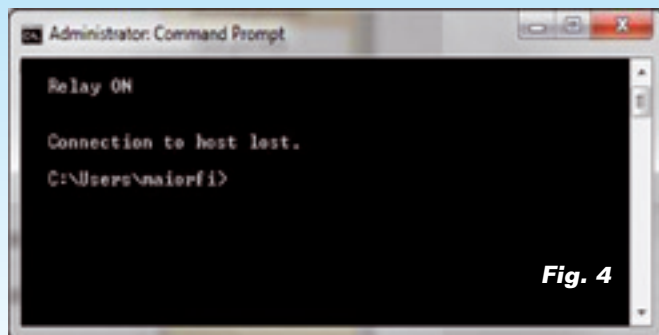


Fig. 4

TCPSocketServer, basta eseguire in modalità Debug il codice TCPClient ed osservare, nella finestra di Debug, il messaggio che il Server trasmette (nel nostro caso abbiamo semplicemente impostato la stringa 'TCPSocketServer 1.0'); contestualmente, nella finestra console dell'applicativo TCPsocketServer potremo leggere la diagnostica dell'avvenuto collegamento.

A questo punto possiamo provare ad implementare la soluzione inversa, in cui il server TCP è rappresentato dal nostro dispositivo embedded ed in cui il client è un'applicazione per PC. Per rendere ancora più generico il nostro firmware, ci proponiamo di supportare, come client TCP, l'applicazione Telnet, che di fatto è un'applicazione terminale simile a quella utilizzata per le comunicazioni seriali, ma che lavora su un canale di comunicazione TCP/IP. L'applicazione Telnet è presente nella maggior parte dei sistemi operativi (in Windows XP è possibile utilizzare anche HyperTerminal), anche se, come nel caso di Vista o Windows 7, potrebbe non essere presente nell'installazione predefinita. La mancanza della classe TcpListener complica un po' le cose, ma l'uso diretto della classe Socket in modalità "server" non è poi così complesso. Nel frammento di codice visibile nel **Listato 3** esaminiamo nel dettaglio l'applicazione firmware relativa ad un server embedded che, a fronte di uno specifico comando inviato via Telnet, esegua una determinata operazione (ad esempio l'attivazione di un relé) ovvero, dal punto di vista del microcontrollore, provveda a porre allo stato logico alto uno dei propri terminali di I/O. Descriviamo qui di seguito le parti più significative del server TCP appena descritto con il listato.

- Nelle righe precedenti il metodo **Main()** viene definita la porta digitale utilizzata per pilotare il relé ed una proprietà statica solo "get" per renderla visibile ad altre classi.
- Nel metodo **Main()** anzitutto viene creato e

configurato un nuovo Socket in ascolto, sulla porta 1234, di tutti gli indirizzi IP disponibili sul dispositivo, anche se di norma, a differenza di quanto avviene in un PC, in un dispositivo di questo tipo è presente un'unica interfaccia di rete, per la quale è definito un solo indirizzo IP, oltre ovviamente allo pseudo-indirizzo di loopback 127.0.0.1.

- Al momento della connessione di un nuovo client, viene creata una nuova istanza dell'oggetto *ClientSocketProcessor*, la cui funzione è quella di provvedere al processing delle richieste che arriveranno da parte di quel client.
- Al momento della creazione della nuova istanza della classe *ClientSocketProcessor*, il costruttore valuta il parametro booleano "async" per eseguire il metodo *processRequest()*, che effettivamente si fa carico della richiesta TCP, nello stesso thread che ha eseguito il metodo *Accept()* del socket oppure utilizzando un thread separato al fine di rendere disponibile il socket in ascolto per altri client ed aumentare di conseguenza la scalabilità del servizio. Infatti va detto che è possibile creare più istanze della classe Socket che siano in ascolto delle richieste provenienti allo stesso indirizzo e sulla stessa porta, purché al momento del collegamento di ogni nuovo client un thread stia eseguendo il metodo **Accept()** del socket server.
- Viene costruita la stringa relativa alla richiesta del client, attraverso la concatenazione dei vari frammenti in ingresso, fino a che non viene rilevata la presenza dei caratteri di fine-riga.
- Viene elaborato il comando ricevuto: a seguito del comando "on" viene attivata la porta che pilota il relé, la quale quando è ricevuto il comando "off" viene disattivata; al "toggle" viene invertito lo stato. In tutti i casi, il client riceverà dal server un feedback testuale.

Per fare una prova pratica bisogna, ancora una volta, collegare il modulo *Tahoe II* alla nostra rete locale, oltre che al PC tramite la porta USB; quest'ultima connessione servirà per la programmazione. L'azione che vedremo a seguito dei comandi impartiti tramite rete è in realtà il passaggio allo stato logico alto

o basso del piedino di uscita GPIO10, operazione effettuabile tramite un semplice voltmetro. Stavolta è il firmware implementato a fungere da server in ascolto sulla porta 1234 dell'indirizzo IP assegnato al Tahoe II. Tale servizio va interrogato tramite l'applicazione Telnet in modo estremamente semplice: basta impartire, dalla riga di comando (per sistemi Windows), "telnet 192.168.0.237 1234", dove assumiamo che l'indirizzo configurato per il dispositivo sia 192.168.0.237; poi dobbiamo digitare all'interno della finestra terminale il comando "on"<invio>. Se è tutto a posto, al momento dell'invio dovremo vedere la tensione al piedino GPIO10 rispetto al piedino GND salire a circa 3,3 volt, mentre la finestra terminale ci segnalerà l'avvenuta "accensione" e la successiva disconnessione, come mostrato nella Fig. 4. Analogamente il funzionamento per i comandi "off"<invio> e "toggle"<invio>, mentre al comando "?"<invio> il server implementato dal firmware risponderà con la stringa ".NET MF TCP Server 1.0".

HTTP

Il protocollo applicativo più utilizzato (non in termini di traffico, ma sicuramente di

EndPoint disponibili) su Internet è senza dubbio quello utilizzato dalle applicazioni "browser" per fruire dei contenuti pubblicati dai siti web: il protocollo HTTP. Per essere più precisi, questo (e le sue successive evoluzioni, quali il WebDAV) si occupa solo di descrivere il modello richiesta/risposta attraverso il quale un client può richiedere ed ottenere un qualche tipo di informazione veicolata da un canale di comunicazione basato su TCP/IP. Quando la risposta veicola, all'interno del corpo dei propri pacchetti HTTP, un testo codificato secondo un determinato linguaggio (come avviene quando si richiede una pagina HTML) il browser si occupa, oltre che dello "scaricamento" della pagina, anche della sua visualizzazione, attraverso l'utilizzo di un motore di "rendering" implementato all'interno dell'applicazione browser stessa.

Quello che ci interessa in questo contesto è capire come poter effettuare delle richieste HTTP all'interno di un dispositivo embedded che utilizzi il .NET Micro Framework, al fine di utilizzare nel nostro firmware delle informazioni pubblicate all'interno di una Intranet o di Internet. È opportuno premettere da subito che non dovremo cedere alla tentazio-

Listato 4

```
public class Program
{
    public static void Main()
    {
        // Creazione di una nuova richiesta http
        HttpWebRequest request = HttpWebRequest.Create(
            "http://search.twitter.com/search.atom?q=netmf") as HttpWebRequest;

        // Invio della richiesta
        HttpWebResponse response = request.GetResponse() as HttpWebResponse;

        // Uso uno stream reader per leggere la risposta
        using (StreamReader sr = new StreamReader(response.GetResponseStream()))
        {
            // Se lo status code http è 200 (ok)
            if (response.StatusCode == HttpStatusCode.OK)
            {
                // Debug del contenuto della risposta
                Debug.Print(sr.ReadToEnd());
            }
            else
            {
                // Debug dello status code ottenuto (ad es. 404 Not Found)
                Debug.Print(response.StatusDescription);
            }
        }
    }
}
```

Listato 5

```

public class Program
{
    public static void Main()
    {
        // Creo un server in ascolto di richieste http
        HttpListener server = new HttpListener("http",8080);

        // lo avvio...
        server.Start();

        while (true)
        {
            HttpListenerResponse response = null;
            HttpListenerContext context = null;

            try
            {
                // Aspetto una richiesta http (questa chiamata è bloccante)
                context = server.GetContext();
                response = context.Response;

                HttpListenerRequest request = context.Request;

                // Verifico che la richiesta fatta sia del tipo "http://<indirizzo>:8080/status"
                if (request.RawUrl == "/status")
                {
                    // Mando lo status code 200 (OK)
                    response.StatusCode = (int)HttpStatusCode.OK;

                    // Comunico al browser che il contenuto va visualizzato come html
                    response.ContentType = "text/html";

                    // Compongo la risposta...
                    using(StreamWriter sw=new StreamWriter(response.OutputStream))
                    {
                        string html = "<html><head><title>.NET MF Test Page</title></head>";
                        html += "<body><h1>Questo &#232; un server web realizzato con";
                        html += ".NET Micro Framework</h1>";
                        html += "<h2>";
                        html += "Qui la temperatura &#232; di " +
                            TahoeII.Tsc2046.ReadTemperature().ToString("F") +
                            "&#186;C";
                        html += "</h2>";
                        html += "</body></html>";

                        sw.Write(html);
                    }
                }
            }
            catch
            {
                // Se ci sono stati problemi nella richiesta chiudo comunque il contesto...
                if (context != null) context.Close();
            }
        }
    }
}

```

ne di provare a realizzare un browser web all'interno del nostro firmware, dal momento che la complessità di un motore di rendering che permetta la visualizzazione di contenuti HTML è al di sopra delle possibilità di questo tipo di dispositivi, o almeno lo è nei dispositivi che non siano dotati di un vero e proprio sistema operativo. Ciò nonostante, la tendenza ad utilizzare servizi esposti su Internet mediante quella che prende il nome di API RESTful, rende questo approccio estremamente conve-

niente. Come esempio, vediamo cosa serve per realizzare un semplice firmware che effettui una ricerca su Twitter, il noto portale community costruito intorno alla pubblicazione di brevi messaggi di testo. In particolare, utilizzeremo l'API RESTful di Twitter per ottenere, in formato XML (più in dettaglio in formato AtomPub), una lista di messaggi pubblicati che riportano il termine "netmf". Il frammento di codice relativo all'esempio in questione è riportato nel **Listato 4**. A questo punto è

opportuno farvi notare che le funzionalità di client HTTP sono completamente incapsulate nelle classi *HttpRequest* e *HttpResponse*, le quali si occupano in maniera completamente trasparente sia della gestione dei socket sottostanti che delle necessarie richieste DNS. Anche in questo caso la lettura del contenuto della risposta proveniente dal server avviene tramite uno *StreamReader*. Per chi conosce il protocollo HTTP, vale inoltre la pena di notare che la gestione delle informazioni "fuori-banda" tipicamente veicolate dagli header è interamente implementata dalle classi in questione, che per gli header più importanti espongono proprietà specifiche -come ad esempio *StatusCode* di *HttpResponse*- che permette di rilevare l'esito di una richiesta HTTP. Il test pratico è davvero elementare: basta eseguire in debug il firmware sopra analizzato ed osservare la risposta alla ricerca su Twitter (in formato XML) nella finestra di Debug.

Anche nel caso delle comunicazioni HTTP, vediamo adesso come sia possibile implementare un server Web embedded con il .NET Micro Framework; in particolare, ci proponiamo di realizzare un semplice firmware in grado di pubblicare sul web (Internet o rete locale, dipendentemente dalla configurazione della rete in cui è inserito il dispositivo) informazioni relative all'ambiente fisico circostante.

Facendo riferimento ancora alla scheda Tahoe II di Device Solutions, utilizzeremo il controller del touch-screen built-in (denominato TSC2046) per rilevare la temperatura ambientale. Una volta avviato il dispositivo, non dovremo far altro che digitare nella barra degli indirizzi di un browser l'indirizzo `http://<indirizzo>:8080/status` (nel nostro esempio `http://192.168.0.237:8080/status`). Se è tutto ben configurato, otterremo un risultato simile a quello illustrato nella Fig. 5.

Il codice relativo al server web appena illustrato è riportato nel Listato 5. Come si può notare da esso, le funzionalità relative all'implementazione del protocollo HTTP sono -anche in questo caso- completamente demandate ad una classe del framework: la *HttpListener*. Il codice si commenta da sé e l'unica cosa che vale la pena sottolineare è l'utilizzo della classe statica relativa al driver per la gestione del controller del touch-screen, utilizzata in

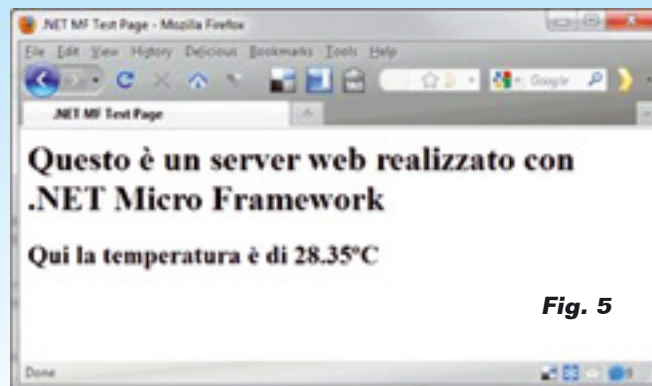


Fig. 5

questo contesto per rilevare la temperatura che farà parte del documento HTML restituito al browser da cui è provenuta la richiesta.

WEBSERVICES

L'uso delle comunicazioni HTTP al fine di veicolare informazioni che non esprimano necessariamente elementi di interfaccia utente, come frammenti di mark-up HTML ad esempio, è diventato negli ultimi anni estremamente importante e diffuso, al punto da meritare un nome specifico: Web Services. L'argomento è sicuramente troppo esteso per essere trattato in questa sede, in cui saremmo in ogni caso interessati a coglierne soprattutto gli aspetti inerenti allo sviluppo di applicazioni embedded, ma vale la pena di sottolineare che nel .NET Micro Framework è stato implementato un completo stack di protocolli dedicati all'esposizione ed al consumo di webservices, secondo quanto previsto dallo standard DPWS (Device Profile Web Services). Tale standard, oltre a supportare gran parte delle tecnologie e dei protocolli utilizzati nei Web Services non espressamente rivolti all'uso embedded, aggiunge una serie di funzionalità specifiche, finalizzate per lo più alla costruzione di reti di dispositivi con funzionalità "mesh" quali "discovery" (servizio grazie al quale un dispositivo può "annunciarsi" all'interno della rete o anche ricercarne altri compatibili con il tipo di interfaccia richiesto) o "eventing" (con cui si possono realizzare interfacce di comunicazione non più basate solo sul paradigma, tipico dell'HTTP, richiesta/risposta) ma più ricche, in cui sia possibile utilizzare anche il pattern full-duplex per lo scambio dei messaggi.

Se volete approfondire tale argomento, potete fare riferimento ai due progetti di esempio che illustrano l'utilizzo dello stack DPWS, denominati *SimpleService* e *SimpleServiceClient*, presenti all'interno del .NET Micro Framework SDK. ■



Microsoft®

.NET Micro Framework

di Lorenzo Maiorfi
e Gianluca Ruta
Innovactive Engineering

È giunto il momento di realizzare insieme un'interfaccia utente di tipo grafico per le nostre applicazioni basata su un display; scopriremo l'uso delle primitive, la gestione del touch-screen e tanto altro ancora. Quarta puntata.

Nelle puntate precedenti del corso sulla piattaforma per lo sviluppo di applicazioni per microcontrollori di casa Microsoft, abbiamo affrontato la comunicazione seriale dei dati e l'utilizzo dello stack TCP/IP per comunicazioni su rete Ethernet. In tutto il codice che abbiamo prodotto per realizzare test concreti inerenti agli argomenti trattati, abbiamo utilizzato spesso la finestra di debug dell'ambiente di sviluppo per avere una semplice diagnostica di quanto il microcontrollore operava, mentre in altri casi la nostra diagnostica si è basata sulla comunicazione seriale di dati ad un PC, con i messaggi visualizzati tramite una finestra di emulazione terminale. Per i semplici scopi didattici che ci siamo prefissati

questo poteva bastare, ma non c'è dubbio che la forma di comunicazione più efficace nella direzione che va dal sistema all'utente non può prescindere dall'uso di un display dedicato, possibilmente grafico e ormai sempre più spesso dotato di funzionalità "touch" per diventare periferica di input oltreché di output. L'argomento di questa puntata verte proprio sull'interfaccia utente delle nostre applicazioni per microcontrollori, realizzata per un display grafico.

GRAFICA ED INTERFACCIA UTENTE

Anche se parlare di interfaccia utente di un'applicazione non significa necessariamente fare riferimento ad argomenti legati alla

Listato 1

```

1: int w,h,d,r;
2:
3: HardwareProvider.HwProvider.GetLCDMetrics(out w,out h,out d,out r);
4:
5: Bitmap bmp = new Bitmap(w, h);
6:
7: bmp.DrawRectangle(
8:     Color.White, // colore bordo ininfluyente
9:     0,           // senza bordo
10:    0, 0, w, h,   // parte da 0,0 (in alto a sinistra) ed ha le stesse dimensioni del display
11:    0, 0,         // raggio degli angoli "stondati"
12:    Color.RoyalBlue, 0, 0, // colore e coordinate di partenza del gradient
13:    Color.RoyalBlue, 0, 0, // colore e coordinate di arrivo del gradient
14:    Bitmap.Opaque); // contenuto opaco (non trasparente)
15:
16: bmp.Flush();

```

visualizzazione di informazioni, è innegabile che il mezzo di gran lunga più efficace in tal senso è costituito dall'insieme delle funzionalità e dei dispositivi finalizzati alla rappresentazione grafica e testuale dei dati manipolati da un'applicazione. L'avvento dell'espressione "multimediale" ha in realtà fatto sì che l'attenzione degli utenti ricadesse anche su rappresentazioni diverse da quella visuale, come quella sonora (ad esempio), ma è indubbio che il ruolo giocato dai media non visuali è assolutamente marginale. Per questo motivo, gran parte dell'evoluzione subita dalle tecnologie che orbitano intorno al mondo dell'Information Technology, ivi compreso il mondo embedded, si è concentrata sul potenziamento delle capacità di visualizzazione delle informazioni e della gestione dei comandi impartiti dagli operatori.

Attento alle tendenze del panorama tecnologico e agli evidenti segnali che provengono dal mondo dell'elettronica consumer, il team dei designer del .NET Micro Framework ha speso molte energie nella realizzazione di

un sistema per la messa a punto di interfacce utente particolarmente ricche. Tale sistema, assolutamente inedito in questo settore, è caratterizzato dalla possibilità di esprimere gerarchie complesse di oggetti grafici attraverso un modello astratto ed indipendente dalla geometria del display utilizzato. Inoltre, il .NET Micro Framework supporta in modo nativo la gestione delle funzionalità "touch", supportate ovviamente dai dispositivi abilitati in tal senso, tra i quali figura la scheda di sviluppo "Tahoe II" di Device Solutions, utilizzata e descritta già nelle precedenti puntate.

BITMAP MODE

Addentrando nelle classi del .NET Micro Framework dedicate alla gestione degli oggetti grafici, scopriamo che esistono in realtà due diversi "strati" che separano il nostro firmware dal driver che di fatto pilota, tipicamente attraverso un'interfaccia parallela, il controller del display: un primo strato si occupa del trasferimento (detto "blitting") di un oggetto *Bitmap* sulla superficie del display, mentre lo strato superiore, denominato "Presentation", si occupa dell'astrazione degli oggetti grafici, ottenuta attraverso la definizione di una struttura gerarchica di contenitori ed elementi di interfaccia utente che alimentano il motore di layout e di rendering attraverso l'uso di un'apposita "pipeline" gestita dal framework stesso.

Iniziamo la nostra sperimentazione a partire dal layer di basso livello. Il paradigma utilizzato in questo modello di sviluppo prevede che all'interno del firmware venga definita un'istanza della classe *Bitmap* caratterizzata dalla stessa geometria del display disponibile sul dispositivo. Tale *Bitmap* agisce come un vero e proprio display "offline", sul quale

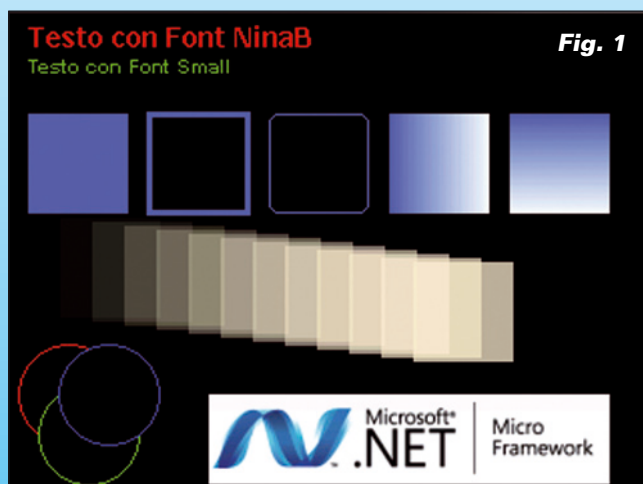


Fig. 1

vengono disegnate delle primitive grafiche attraverso appositi metodi della classe `Bitmap`, fino a quando, a seguito dell'esecuzione del metodo `Flush()`, i pixel che descrivono la superficie della bitmap offline vengono effettivamente disegnati sul display del dispositivo.

Se, ad esempio, vogliamo disegnare un rettangolo rosso che copra l'intera superficie del display, è sufficiente utilizzare la porzione di codice riportata nel **Listato 1**.

Analogamente, è possibile utilizzare le numerose primitive grafiche per disegnare ellissi, bitmap, testo e linee. Facendo riferimento alla **Fig. 1**, in cui è illustrato l'output del firmware che prenderemo a breve in esame, vale la pena di sottolineare che ciascuna primitiva comporta delle peculiarità di cui occorre essere al corrente per evitare sorprese al momento del suo utilizzo.

TESTO

Nella parte in alto a sinistra dell'immagine, vediamo due testi realizzati mediante la primitiva `"DrawText"`. Tale metodo può essere utilizzato secondo quanto illustrato nel **Listato 2**, in cui `"bmp"` rappresenta, come nel listato precedente, la `Bitmap` relativa al display "offline". Alle righe 2 e 7 è possibile notare come selezionare il font da utilizzare per il rendering del testo; infatti, a differenza di quanto avviene all'interno di Windows, nella runtime del .NET Micro Framework non sono presenti dei font residenti. Questo si riflette sull'impossibilità di istanziare la classe `Font` a partire, ad esempio, dal nome della famiglia del carattere e dalla dimensione. L'unica via prevista è quella in cui il font viene caricato attraverso le "risorse" definite all'interno dell'applicazione.

Una "risorsa" è definita dal framework come un oggetto (binario o testuale) incapsulato all'interno di un'apposita area dell'assembly prodotto come risultato della compilazione di un file sorgente con suffisso `".resx"`. Tale file, strutturato come un xml, riporta il riferimento ai file che verranno effettivamente inclusi come risorsa. In fase di editing del progetto, l'ambiente di sviluppo gestisce le risorse con

Listato 2

```
1: bmp.DrawText("Testo con Font NinaB",
2:   Resources.GetFont(Resources.FontResources.NinaB),
3:   ColorUtility.ColorFromRGB(255, 0, 0),
4:   10, 5);
5:
6: bmp.DrawText("Testo con Font Small",
7:   Resources.GetFont(Resources.FontResources.small),
8:   ColorUtility.ColorFromRGB(0, 255, 0),
9:   10, 20);
```

un apposito designer, dal quale è possibile includere file generici o di tipo specifico, quali immagini, font, testi, ecc. L'editor di risorse genera inoltre un file sorgente per ciascun file `.resx` presente nel progetto. Tale file, denominato con l'estensione `".Designer.cs"`, contiene la dichiarazione della classe parziale `"Resources"`, attraverso la quale è possibile accedere all'effettivo contenuto di una risorsa, utilizzando metodi specializzati per il tipo di risorsa richiesta. All'inclusione di un file di tipo `".tinyfont"`, che descrive un tipo di carattere tipografico utilizzabile nel .NET Micro Framework, viene, ad esempio, generato il metodo `GetFont()` che prevede un parametro di tipo enumerato denominato `FontResources`, anch'esso definito dal designer, in cui sono elencati tutti i font utilizzabili dall'applicazione. L'installazione del .NET Micro Framework SDK crea una cartella denominata `"Fonts"`, tipicamente contenuta in `"C:\Program Files\Microsoft .NET Micro Framework\v4.0"`, in cui sono presenti due file con suffisso `tinyfont`: `'NinaB'` e `'small'`, illustrati nella già citata **Fig. 1**, rispettivamente in rosso e in verde. Altri font possono essere aggiunti mediante l'utilità `TFCConvert`, contenuta nella cartella `"Tools"`. Tale utility consente infatti la conversione di un font TrueType in un `.tinyfont`, con l'unica limitazione rappresentata dal fatto che, essendo i font del .NET Micro Framework non scalabili, in fase di conversione occorre specificare la dimensione del font da convertire.

Listato 3

```
1: // Test trasparenza
2: for (ushort opacity = 0; opacity < 200; opacity += 16)
3: {
4:     bmp.DrawRectangle(Color.White,
5:       0,
6:       10 + opacity, 100 + opacity / 8, 50, 50,
7:       0, 0,
8:       ColorPapayaWhip, 0, 0,
9:       ColorPapayaWhip, 0, 0,
10:      opacity);
11: }
```

AREE**RETTANGOLARI**

La primitiva più sviluppata è senz'altro quella relativa alla produzione di aree rettangolari; ciò perché costituisce il principale strumento per la realizzazione di oggetti grafici complessi quali pulsanti, liste, pannelli, ecc. Analizzando i cinque rettangoli blu riportati nella **Fig. 1** è possibile notare come siano previste diverse varianti implementate dallo stesso metodo `DrawRectangle()`. Nell'applicazione demo sono disegnati, nell'ordine da sinistra verso destra, rispettivamente un rettangolo pieno con colore uniforme, uno vuoto con spessore maggiore di un pixel, uno vuoto con angoli arrotondati, un rettangolo colorato con riempimento 'gradient' (ossia sfumato tra due colori estremi) e in ultimo, ancora un rettangolo con riempimento 'gradient' ma con direzione verticale. La direzione del riempimento 'gradient' è liberamente impostabile specificando i due punti che delimitano gli estremi della interpolazione. Il rendering di aree rettangolari può inoltre avvenire utilizzando una modalità semitrasparente, in cui, attraverso l'ultimo parametro, è possibile specificare il livello di opacità tra 0 e 255. Un esempio di utilizzo di tale funzionalità è riportato nell'area centrale dell'immagine in **Fig. 1**, prodotta tramite i comandi riportati nel codice del **Listato 3**.

CERCHI ED ELLISSI

Attraverso la primitiva `DrawEllipse()` è possibile disegnare all'interno di una `Bitmap` cerchi ed ellissi, analogamente a quanto avviene per le aree rettangolari. Però, a causa di una limitazione nell'attuale implementazione del .NET Micro Framework, tale primitiva non può essere utilizzata per disegnare figure piene, ma solo in modalità "outline" (limitazione aggirabile attraverso l'uso di opportune risorse di tipo "Image"). L'uso della primitiva `DrawEllipse()` è riportato nel codice contenuto nel **Listato 4**, in cui viene riprodotta la parte in basso a sinistra dell'immagine in **Fig. 1**.

Listato 4

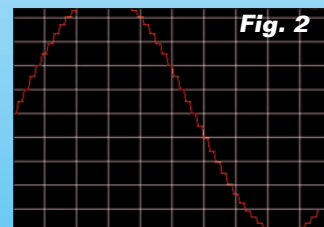
```
1: // Cerchi
2: bmp.DrawEllipse(ColorUtility.ColorFromRGB(255, 0, 0), 30, 190, 25, 25);
3: bmp.DrawEllipse(ColorUtility.ColorFromRGB(0, 255, 0), 40, 210, 25, 25);
4: bmp.DrawEllipse(ColorUtility.ColorFromRGB(0, 0, 255), 50, 190, 25, 25);
5:
6: // NON IMPLEMENTATO IN .NET MF 4.0
7: // bmp.DrawEllipse(
8: //     ColorYellow, 0, 90, 200, 25, 25,
9: //     ColorYellow, 90, 200, ColorMediumSpringGreen, 100, 210, Bitmap.OpacityOpaque);
```

Listato 5

```
1: // Immagini
2: Bitmap bmpIcon1 = Resources.GetBitmap(Resources.BitmapResources.netmf);
3: bmp.DrawImage(100, 190, bmpIcon1, 0, 0, bmpIcon1.Width, bmpIcon1.Height);
```

IMMAGINI BITMAPPED

L'ultima primitiva che prendiamo in considerazione è anche la più potente. Il metodo `DrawImage()` consente infatti di disegnare all'interno della superficie grafica "offline" delle immagini descritte attraverso istanze della classe "Bitmap". Oltre al rendering semplice, è possibile effettuare anche la scalatura, riportando l'immagine con dimensioni diverse da quelle originali, ed utilizzare diversi livelli di trasparenza, come per le aree rettangolari. Le immagini da disegnare possono essere definite attraverso il loro contenuto "binario", espresso in formato Bmp, Gif, Jpeg o TinyCLRbitmap (un formato personalizzato utilizzato all'interno del Framework), oppure, in alternativa, estratte dalle risorse dell'applicazione. Il codice di esempio riportato nel **Listato 5**, disegna sul display l'immagine raffigurante il logo del .NET Micro Framework, precedentemente aggiunto al progetto come risorsa. Come esempio conclusivo della parte riguardante l'approccio a basso livello per la generazione di interfacce utente, ci proponiamo di realizzare un semplicissimo oscilloscopio, attraverso lo sviluppo di un'applicazione che, utilizzando l'ADC presente a bordo della scheda di sviluppo TahoeII, permetta il monitoraggio grafico del livello dell'ingresso analogico "AUX", in una scala 0÷3,3V. L'immagine prodotta dal firmware in questione è riprodotta nella **Fig. 2**. Il codice relativo al sorgente del firmware è riportato nel **Listato 6**, in cui è possibile notare l'utilizzo della primitiva `DrawLine()`, non trattata in precedenza, per la tracciatura di linee



con spessore e colore parametrici.

PRESENTATION

Lo strato di più alto livello per la realizzazione di interfacce utente prende, all'interno del .NET Micro Framework, il nome di "Presentation". In realtà esso è distribuito all'interno di un assembly specifico, denominato "Microsoft.SPOT.TinyCore", che definisce tutte le classi relative ai namespaces "Microsoft.SPOT.Presentation.*". A differenza di quanto visto per il layer di basso livello, lo strato "Presentation" si occupa anche della gestione dell'input dell'utente, per quanto riguarda l'uso sia di pulsanti, sia di display con funzionalità "touch". Prima di addentrarci nei meandri di "Presentation", vale la pena sottolineare che l'ambiente di sviluppo prevede un apposito template per la creazione di applicazioni che ne utilizzano le funzionalità. Tale template prende il nome di "Window Application" (nella Fig. 3 si

vede la finestra di dialogo di Visual Studio durante la creazione di un nuovo progetto con questo modello). L'applicazione così creata dall'ambiente di sviluppo ci dà modo di esaminare alcune delle classi più importanti ed in particolare la maniera in cui collaborano, nell'ambito di un'applicazione di questo tipo, gli oggetti di tipo "Window", "Program" e "GPIOButtonInputProvider".

Premesso che le funzionalità relative al firmware di esempio che verrà illustrato si limitano a riportare sul display una stringa caricata da risorse e ad intercettare la pressione dei pulsanti del dispositivo per riportarne l'identificativo all'interno della finestra di debug, vale la pena di "sezionare" le parti principali dell'applicazione per identificarne responsabilità e peculiarità. L'interfaccia dell'applicazione in questione è illustrata in Fig. 4. L'entry point dell'applicazione, stavolta derivata dalla classe "Microsoft.SPOT.Application", è ancora una volta il metodo statico "Main", nel quale vengono creati un oggetto Window ed un oggetto GPIOButtonInputPro-

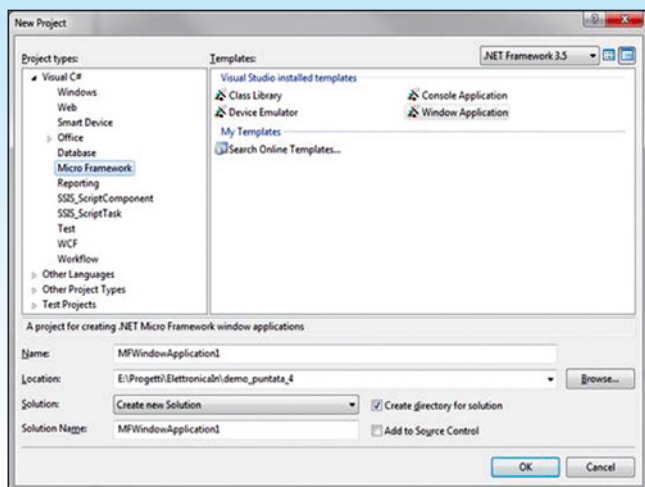
Listato 6

```

1: // Divisione del reticolo...
2: int xdivs = 10;
3: int ydivs = 10;
4: int oldx = 0, oldy = 0;
5:
6: while (true)
7: {
8:     oldx = -1;
9:     oldy = -1;
10:
11:     // Pulisco la bitmap
12:     bmp.Clear();
13:
14:     // Bordo...
15:     bmp.DrawRectangle(ColorGrey, 1, 0, 0, w, h,
16:         0, 0, Color.White, 0, 0, Color.White, 0, 0,
17:         Bitmap.OpaCityTransparent);
18:
19:
20:     // reticolo...
21:     for (int xi = 0; xi < xdivs; xi++)
22:     {
23:         bmp.DrawLine(ColorGrey, 1, xi * (w / xdivs), 0, xi * (w / xdivs), h);
24:     }
25:
26:     for (int yi = 0; yi < ydivs; yi++)
27:     {
28:         bmp.DrawLine(ColorGrey, 1, 0, yi * (h / ydivs), w, yi * (h / ydivs));
29:     }
30:
31:     bmp.DrawText("AUX",
32:         Resources.GetFont(Resources.FontResources.small),
33:         ColorUtility.ColorFromRGB(255, 0, 0), 294, 6);
34:
35:     bmp.Flush();
36:
37:     // Scansione e lettura ADC
38:     for (int xi = 0; xi < w; xi++)
39:     {
40:         float scaledtemp=(TahoeII.Tsc2046.ReadAux()) / 3.3f;
41:
42:         int yt = h-(int)(scaledtemp*h);
43:
44:         if (oldx != -1)
45:         {
46:             // Linea che unisce gli ultimi 2 punti rilevati...
47:             bmp.DrawLine(ColorUtility.ColorFromRGB(255, 0, 0), 1, oldx, oldy, xi, yt);
48:
49:             bmp.Flush();
50:         }
51:
52:         oldx = xi - 1;
53:         oldy = yt;
54:
55:         Thread.Sleep(10);
56:     }
57:
58: }

```

Fig. 3



vider, come illustrato nel **Listato 7**.

L'oggetto Window rappresenta un contenitore logico e visuale per la generazione dell'interfaccia utente. Durante il normale funzionamento dell'applicazione ne vengono spesso create più istanze, la cui sovrapposizione viene gestita dal motore grafico del framework. La creazione della prima (ed unica) finestra della nostra applicazione di esempio avviene grazie al codice riportato nel **Listato 8**.

Le sezioni degne di nota sono senz'altro quelle relative alle righe 11÷19, in cui viene creato e configurato un oggetto grafico di tipo "Text" (paragonabile ad un controllo grafico di tipo "etichetta") e alla riga 22, in cui vediamo come configurare il sistema di gestione dell'input in modo tale che gli eventi di tipo "ButtonUp" provenienti dal sottosistema "Buttons" vengano gestiti dal metodo denominato (arbitrariamente) OnButtonUp.

Tale metodo, implementato nella nostra applicazione come semplice funzione di debug che riporta il codice del pulsante premuto, è

Listato 7

```

1: public class Program : Microsoft.SPOT.Application
2: {
3:     public static void Main()
4:     {
5:         Program myApplication = new Program();
6:
7:         Window mainWindow = myApplication.CreateWindow();
8:
9:         // Crea l'oggetto che mappa i GPIO come ingressi di tipo "Button"
10:        GPIOButtonInputProvider inputProvider = new GPIOButtonInputProvider(null);
11:
12:        // Avvia l'applicazione a partire dalla finestra appena creata
13:        myApplication.Run(mainWindow);
14:    }
15:
16:    // [...]
17: }
```

Fig. 4



definito con il codice riportato nel **Listato 9**. Quello che non appare evidente da

questo listato è come sia possibile che gli eventi relativi alla pressione dei pulsanti presenti sul dispositivo target siano effettivamente legati agli ingressi digitali corrispondenti ai pulsanti stessi, o anche come siano coniugati lo stato logico dell'ingresso digitale con quello relativo alla pressione del pulsante, collegabile, quest'ultimo, con una resistenza di pull-up o di pull-down rispettivamente ai livelli di tensione alto e basso. La risposta alla nostra domanda viene dall'analisi di quanto avviene alla riga 10 del metodo Main() illustrato in precedenza: la classe "GPIOButtonInputProvider" si occupa della trasformazione degli interrupt di basso livello generati dalle variazioni di stato degli ingressi digitali collegati ai pulsanti in eventi instradati all'interno del sistema di gestione dell'input del layer "presentation" del Framework. Ciò avviene come illustrato nel **Listato 10**.

Ad una prima analisi, la complessità della struttura della classe GPIOButtonInputProvider può effettivamente spazzare, ma la cosa effettivamente importante da notare è l'effettiva responsabilità di ciascuna delle classi utilizzate, riassunta nell'elenco che segue.

- **PresentationSource** è una classe che agevola l'utilizzo di sistemi di rendering indipendenti dall'hardware, attraverso l'esposizione della proprietà RootUIElement, radice

logica di una sorgente di contenuti.

- **Dispatcher** è una classe che si occupa della gestione della coda di eventi di un thread.

- **InputManager** è una classe che coordina tutti i sistemi di input attivi nel sistema, trasformando gli eventi di basso livello in eventi consumabili dal sottosistema "Presentation".

- **HardwareProvider** fornisce informazioni sulla

mappatura delle funzionalità per uno specifico hardware. Nel caso appena visto ad esempio viene utilizzata per conoscere la mappatura tra GPIO di ingresso e codici pulsante.

- **ButtonDevice** rappresenta un generico dispositivo hardware dotato di pulsanti.
- **InputProviderSite** rappresenta una classe con funzioni di mediazione tra un input provider ed il InputManager.
- **RawButtonInputReport** rappresenta l'oggetto che veicola l'informazione sull'attivazione di un pulsante che viene passato da un input provider all'InputProviderSite e ancora all'InputManager.

PATTERN D'USO DEL PRESENTATION LAYER

Per approfondire la conoscenza sugli strumenti disponibili all'interno del vasto mondo del layer "Presentation", è opportuno prendere in esame alcuni esempi notevoli che illustrano soluzioni articolate per la rappresentazioni di informazioni anche complesse. Gran parte di quanto vedremo in questa sezione è riassunto nel sample denominato "SimpleWPFApplication", contenuto nel .NET Micro Framework SDK, di cui la **Fig. 5** rappresenta l'interfaccia utente principale. Per dare accesso a ciascun sotto-demo illustrato, l'applicazione in questione utilizza un menu scorrevole animato, azionabile attraverso i tasti "destra", "sinistra" e "centrale", mappati sulla scheda TahoeII che stiamo utilizzando, all'interno del tastierino direzionale riportato nella parte inferiore destra della scheda stessa. Ciascuna finestra relativa ad una parte del programma dimostrativo deriva da una classe comune, definita all'interno della medesima applicazione, denominata

Listato 8

```
1: public Window CreateWindow()
2: {
3:     Window wnd;
4:
5:     // Creazione di una finestra con le stesse dimensioni del display
6:     wnd = new Window();
7:     wnd.Height = SystemMetrics.ScreenHeight;
8:     wnd.Width = SystemMetrics.ScreenWidth;
9:
10:    // Creazione di un oggetto di tipo Text
11:    Text text = new Text();
12:
13:    text.Font = Resources.GetFont(Resources.FontResources.small);
14:    text.TextContent = Resources.GetString(Resources.StringResources.String1);
15:    text.HorizontalAlignment = Microsoft.SPOT.Presentation.HorizontalAlignment.Center;
16:    text.VerticalAlignment = Microsoft.SPOT.Presentation.VerticalAlignment.Center;
17:
18:    // Aggiunta dell'oggetto Text alla gerarchia della finestra
19:    wnd.Child = text;
20:
21:    // Aggiunta di un gestore di evento per l'evento ButtonUp
22:    wnd.AddHandler(Buttons.ButtonUpEvent, new ButtonEventHandler(OnButtonUp), false);
23:
24:    // La finestra viene effettivamente mostrata
25:    wnd.Visibility = Visibility.Visible;
26:
27:    // L'input di tipo "Button" viene ridirezionato su questa finestra
28:    Buttons.Focus(wnd);
29:
30:    return wnd;
31: }
```

'PresentationWindow'. La derivazione comune di più tipi di finestra da una stessa base permette, come avviene in generale per mezzo dell'ereditarietà dei linguaggi orientati ad oggetti, di condividere caratteristiche e comportamenti comuni, come il fatto di inicializzarsi con la massima dimensione disponibile sul display e supportare l'operazione comune relativa alla pressione di un pulsante, nel caso del codice dimostrativo in questione.

I vari sotto-demo illustrati mostrano ciascuno un aspetto peculiare dello sviluppo di interfacce utente attraverso gli strumenti del framework. A titolo di esempio, prendiamo in esame il sotto-demo relativo all'utilizzo del controllo 'Stack Panel', illustrato in **Fig 6**. Il codice corrispondente è nel **Listato 11**.

Vediamo sinteticamente gli aspetti salienti che costituiscono la soluzione appena illustrata.

- (riga 1) La finestra di questa sotto-demo eredita dalla 'PresentationWindow' illustrata in precedenza.
- (righe 3-17) Viene definita una classe an-

Listato 9

```
1: private void OnButtonUp(object sender, ButtonEventArgs e)
2: {
3:     // Visualizza il codice del pulsante nella finestra di debug
4:     Debug.Print(e.Button.ToString());
5: }
```

Listato 10

nidata che eredita dalla classe 'Shape'; l'ereditarietà consente alla nostra forma personalizzata di ereditare tutto ciò che è comune alle altre forme, pur mantenendo la flessibilità di ridefinire l'aspetto in cui è specializzata, effettuando quello che viene descritto come "overriding" del metodo OnRender(), in cui avviene la produzione del disegno vero e proprio.

- (righe 20-36) Nel costruttore della classe StackPanelDemo, viene creato uno StackPanel, ossia un contenitore grafico che implementa un layout secondo il quale gli oggetti figli vengono riportati uno sotto l'altro (in caso di orientamento verticale) o uno accanto all'altro (in caso di orientamento orizzontale), seguendo l'ordine di inserimento; il pannello viene poi aggiunto come primo ed unico figlio dell'oggetto Window corrente.

Nella seconda parte del costruttore viene poi popolato un array di oggetti di tipo Shape, in realtà costituito da forme di tipo diverso (un ellisse, una linea, un quadrato, un rettangolo ed un oggetto del nostro tipo personalizzato, ossia la croce). Come ultima operazione viene scorso l'array in questione per aggiungere ciascuna Shape al pannello. Lo StackPanel

```

1: public sealed class GPIOButtonInputProvider
2: {
3:     public readonly Dispatcher Dispatcher;
4:
5:     private ButtonPad[] buttons;
6:     private DispatcherOperationCallback callback;
7:     private InputProviderSite site;
8:     private PresentationSource source;
9:
10:    public GPIOButtonInputProvider(PresentationSource source)
11:    {
12:        // memorizziamo la sorgente dell'input
13:        this.source = source;
14:
15:        // registriamo questo oggetto come sorgente di "input" e abbiamo
16:        // indietro un "InputProviderSite", che inoltra le segnalazioni
17:        // di input all'InputManager, che a sua volta colloca l'input nell'
18:        // area di "staging"
19:        site = InputManager.CurrentInputManager.RegisterInputProvider(this);
20:
21:        // Creiamo un metodo di callback che, a seguito di un "report"
22:        // di segnalazione di basso livello proveniente da un interrupt,
23:        // inoltra la notifica all'InputProviderSite
24:        callback = new DispatcherOperationCallback(delegate(object report)
25:        {
26:            InputReportArgs args = (InputReportArgs)report;
27:            return site.ReportInput(args.Device, args.Report);
28:        });
29:        Dispatcher = Dispatcher.CurrentDispatcher;
30:
31:        // Creiamo un provider hardware
32:        HardwareProvider hwProvider = new HardwareProvider();
33:
34:        // definiamo i pin relativi ai GPIO collegati ai pulsanti
35:        // con il default dell'emulatore
36:        Cpu.Pin pinLeft = Cpu.Pin.GPIO_Pin0;
37:        Cpu.Pin pinRight = Cpu.Pin.GPIO_Pin1;
38:        Cpu.Pin pinUp = Cpu.Pin.GPIO_Pin2;
39:        Cpu.Pin pinSelect = Cpu.Pin.GPIO_Pin3;
40:        Cpu.Pin pinDown = Cpu.Pin.GPIO_Pin4;
41:
42:        // Se non siamo sull'emulatore, usiamo la mappatura del
43:        // HardwareProvider appena definito
44:        if ((pinLeft = hwProvider.GetButtonPins(Button.VK_LEFT)) ==
45:            Cpu.Pin.GPIO_NONE)
46:            pinLeft = Cpu.Pin.GPIO_Pin0;
47:        else
48:        {
49:            pinRight = hwProvider.GetButtonPins(Button.VK_RIGHT);
50:            pinUp = hwProvider.GetButtonPins(Button.VK_UP);
51:            pinSelect = hwProvider.GetButtonPins(Button.VK_SELECT);
52:            pinDown = hwProvider.GetButtonPins(Button.VK_DOWN);
53:        }
54:
55:        // Definiamo un insieme di oggetti ButtonPad come array
56:        // di oggetti che legano il GPIOButtonInputProvider, il codice "virtuale"
57:        // del pulsante e il pin associato
58:        ButtonPad[] buttons = new ButtonPad[]
59:        {
60:            // Associate the buttons to the pins as discovered or set above
61:            new ButtonPad(this, Button.VK_LEFT, pinLeft),
62:            new ButtonPad(this, Button.VK_RIGHT, pinRight),
63:            new ButtonPad(this, Button.VK_UP, pinUp),
64:            new ButtonPad(this, Button.VK_SELECT, pinSelect),
65:            new ButtonPad(this, Button.VK_DOWN, pinDown),
66:        };
67:
68:        this.buttons = buttons;

```

è solo uno dei contenitori inclusi nel framework e altri possono essere aggiunti anche attraverso la creazione di classi derivate da Panel; l'aspetto più rilevante da tenere a mente è che nell'ambito della gerarchia

```

69:     }
70:
71:     internal class ButtonPad : IDisposable
72:     {
73:         private Button button;           // Codice del pulsante
74:         private InterruptPort port;      // Porta che genera gli interrupt di notifica
75:         private GPIOButtonInputProvider sink;
76:         private ButtonDevice buttonDevice;
77:
78:         public ButtonPad(GPIOButtonInputProvider sink, Button button,
79:             Cpu.Pin pin)
80:         {
81:             this.sink = sink;
82:             this.button = button;
83:             this.buttonDevice = InputManager.CurrentInputManager.ButtonDevice;
84:
85:             // Se stiamo su un hardware vero e non emulato
86:             // creiamo una porta interrupt che notifica
87:             // di entrambi i tipi di variazione
88:             if (pin != Cpu.Pin.GPIO_NONE)
89:             {
90:                 // When this GPIO pin is true, call the Interrupt method.
91:                 port = new InterruptPort(pin, true,
92:                     Port.ResistorMode.PullUp,
93:                     Port.InterruptMode.InterruptEdgeBoth);
94:                 port.OnInterrupt += new NativeEventHandler(this.Interrupt);
95:             }
96:         }
97:
98:         protected virtual void Dispose(bool disposing)
99:         {
100:             if (disposing)
101:             {
102:                 if (port != null)
103:                 {
104:                     port.Dispose();
105:                     port = null;
106:                 }
107:             }
108:         }
109:
110:         public void Dispose()
111:         {
112:             Dispose(true);
113:             GC.SuppressFinalize(this);
114:         }
115:
116:         void Interrupt(uint data1, uint data2, DateTime time)
117:         {
118:             // Intercetto la variazione dell'ingresso digitale
119:             // e identifico il tipo di azione effettivamente svolta
120:             RawButtonActions action = (data2 != 0) ?
121:                 RawButtonActions.ButtonUp : RawButtonActions.ButtonDown;
122:
123:             // Creao un nuovo report di segnalazione
124:             RawButtonInputReport report = new RawButtonInputReport(
125:                 sink.source, time, button, action);
126:
127:             // accodo l'evento relativo all'interrupt verso il nostro InputProviderSite
128:             // utilizzando l'oggetto Dispatcher per poter effettuare
129:             // la chiamata della callback da un altro thread
130:             // (il thread dell'interrupt e quello che gestisce la
131:             // interfaccia utente sono diversi)
132:             sink.Dispatcher.BeginInvoke(sink.callback, new InputReportArgs(buttonDevice,
133:                 report));
134:         }
135:     }

```

del layer Presentation alcuni oggetti grafici non consentono l'inclusione di oggetti figli, altri consentono l'inclusione di un unico elemento figlio (come la Window) ed altri consentono invece l'inclusione di un nume-

ro arbitrario di figli (come per tutti gli oggetti derivati da Panel).

Il codice appena preso in esame viene utilizzato anche nel caso del sotto-demo "Horizontal Stack Panel", in cui gli oggetti Shape sono riportati in orizzontale anziché in verticale, sfruttando la proprietà dello StackPanel valorizzata alla riga 23 del codice appena esaminato.

Per un esempio di personalizzazione di un controllo contenitore di tipo "Panel", è possibile fare riferimento al sotto-demo "Scrollable Panel", la cui interfaccia utente è illustrata nella Fig 7. Tale applicazione illustra come creare una superficie grafica virtualmente molto più estesa del display fisico che la visualizza, tramite la definizione di un controllo personalizzato denominato "TextScrollView".

TOUCH-SCREEN

La gestione del touch-screen all'interno del sistema "Presentation" è

estremamente semplificato dal modello ad eventi di chiara ispirazione Windows. In particolare, come è nel demo denominato "StylusCapture", anch'esso installato dal setup del .NET micro Framework SDK, gli oggetti

Fig. 5

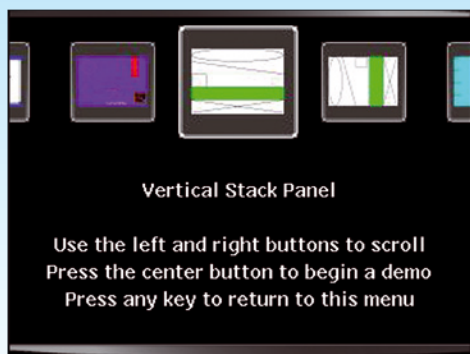


Fig. 6

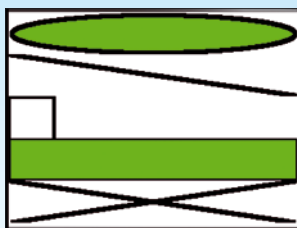
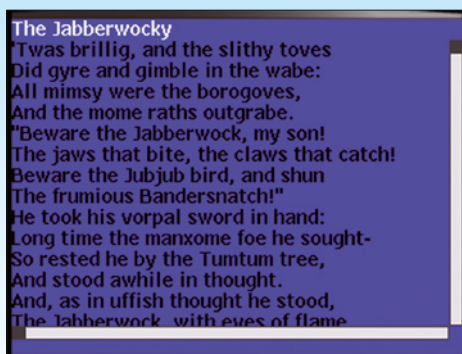


Fig. 7



Listato 11

```

1: internal sealed class StackPanelDemo : PresentationWindow
2: {
3:     private sealed class Cross : Shape
4:     {
5:         public Cross() { }
6:
7:         // Ridefiniamo il metodo OnRender della classe "UIElement"
8:         // ereditata tramite Window
9:         public override void OnRender(DrawingContext dc)
10:        {
11:            // Linea diagonale 1
12:            dc.DrawLine(base.Stroke, 0, 0, Width, Height);
13:
14:            // Linea diagonale 2
15:            dc.DrawLine(base.Stroke, Width, 0, 0, Height);
16:        }
17:    }
18:
19:    // Costruttore
20:    public StackPanelDemo(MySimpleWPFApplication program, Orientation orientation)
21:        : base(program)
22:    {
23:        StackPanel panel = new StackPanel(orientation);
24:        this.Child = panel;
25:        panel.Visibility = Visibility.Visible;
26:
27:        // Array delle forme geometriche da disegnare
28:        Shape[] shapes = new Shape[] {
29:            new Ellipse(0, 0,
30:            new Line(),
31:            // Quadrato
32:            new Polygon(new Int32[] { 0, 0, 50, 0, 50, 50, 0, 50 }),
33:            new Rectangle(),
34:            // Forma personalizzata (croce)
35:            new Cross()
36:        };
37:
38:        // Aggiungo le forme alla gerarchia visuale
39:        for (int x = 0; x < shapes.Length; x++)
40:        {
41:            Shape s = shapes[x];
42:            s.Fill = new SolidColorBrush(ColorUtility.ColorFromRGB(0, 255, 0));
43:            s.Stroke = new Pen(Color.Black, 2);
44:            s.Visibility = Visibility.Visible;
45:            s.HorizontalAlignment = HorizontalAlignment.Center;
46:            s.VerticalAlignment = VerticalAlignment.Center;
47:            s.Height = Height - 1;
48:            s.Width = Width - 1;
49:
50:            if (panel.Orientation == Orientation.Horizontal)
51:                s.Width /= shapes.Length;
52:            else
53:                s.Height /= shapes.Length;
54:
55:            panel.Children.Add(s);
56:        }
57:    }
58: }

```

grafici definiti all'interno della gerarchia visuale espongono una serie di eventi relativi alle singole azioni rilevabili, quali ad esempio "TouchDown" o "TouchUp", rispettivamente sollevati da un elemento visuale quando questo viene "toccato" o "rilasciato". Per sottoscrivere la nostra applicazione alla notifica di questo tipo di eventi, è sufficiente utilizzare i comandi riportati nel **Listato 12**. Una volta intercettato l'evento è possibile compiere qualsiasi tipo di azione, come cambiare lo stato dell'elemento visuale "toccato" o attivare delle specifiche funzioni hardware. Qualora fosse necessario che il controllo che rileva il primo evento di TouchDown continui a ricevere notifiche dal touchscreen anche quando l'area impegnata dal tocco non si sovrappone più con la superficie del controllo, è possibile utilizzare la cosiddetta funzionalità di "cattura", come illustrato nel **Listato 13**. In maniera analoga, è possibile intercettare gli eventi relativi al touch screen all'interno di un oggetto Window o derivato ridefinendo tramite "override" i metodi OnTouchUp() e OnTouchDown(), come illustrato nel **Listato 14**. L'assegnazione fatta a titolo di esempio a riga 7 consente di intervenire nel processo

di “bubbling” degli eventi gestito dal sistema “Presentation”. Per *bubbling* si intende la politica di gestione delle notifiche degli eventi relativi all’interazione con elementi visuali; poiché infatti è possibile che un evento venga rilevato da un elemento ma debba essere notificato anche al suo contenitore, o al contenitore ancora più esterno o all’oggetto Window che fa da involucro “root” della gerarchia, il framework utilizza il concetto di RoutedEvent. Quest’ultimo è un oggetto il cui compito principale è veicolare una notifica legata ad una qualche tipo di azione effettuata su un elemento visuale secondo una delle seguenti tre “strategie”:

- Bubble; in cui l’evento sale a galla della gerarchia, dall’elemento più interno al contenitore più esterno;
- Direct; in cui l’evento non viene effettivamente instradato ma arriva solo al target principale;
- Tunnel; in cui l’evento scende nella gerarchia, dall’elemento più esterno a quello più interno.

Mentre lo scopo delle prime due strategie è abbastanza semplice da intuire, la Tunnel ha una finalità più difficile da cogliere; essa viene utilizzata prevalentemente nelle notifiche di tipo “Preview” in cui a ciascun contenitore viene data l’opportunità di intercettare la notifica dell’evento del proprio “sottoalbero”. L’interazione con il processo di “bubbling” o di “tunnelling” avviene tramite la proprietà Handled, esposta dalla classe base che descrive il parametro che accompagna ogni RoutedEvent: *RoutedEventArgs*. Come ultimo esempio, vale la pena di soffermarci sulle immagini relative all’applicazione dimostrativa della più potente libreria disponibile per .NET Micro Framework in merito allo sviluppo di interfacce utente: MFRichMediaExtensions della Innobedded, società fondata da Jens Kühner, autore del miglior testo in circolazione in materia di .NET Micro Framework, giunto alla sua seconda edizione. Tale libreria, scaricabile all’indirizzo <http://www.innobedded.com/download.html>, è utilizzabile in modalità demo senza limiti di tempo, seppur con frequenti interruzioni da parte di una bitmap che agisce come “nag screen” raffigurante il logo della Innobedded. La licenza d’uso della

Listato 12

```
1: Text text1 = new Text();
2: text1.TouchDown += new TouchEventHandler(Text_TouchDown);
3: text1.TouchUp += new TouchEventHandler(Text_TouchUp);
4:
5: void Text_TouchDown(object sender, TouchEventArgs e)
6: {
7:     // ...
8: }
9:
10: void Text_TouchUp(object sender, TouchEventArgs e)
11: {
12:     // ...
13: }
14:
```

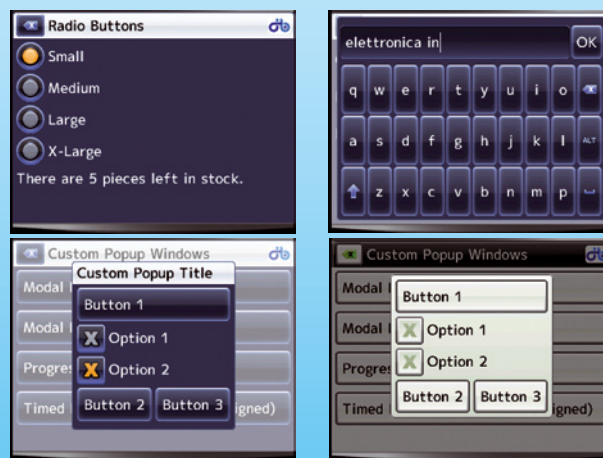
Listato 13

```
1: // Cattura...
2: TouchCapture.Capture(text);
3:
4: // Rilascio...
5: TouchCapture.Capture(text, CaptureMode.None);
```

Listato 14

```
1: protected override void OnTouchDown(TouchEventArgs e)
2: {
3:     base.OnTouchDown(e);
4:
5:     // ...
6:
7:     e.Handled = true;
8: }
9:
10: protected override void OnTouchUp(TouchEventArgs e)
11: {
12:     base.OnTouchDown(e);
13:
14:     // ...
15:
16: }
```

libreria ha un prezzo molto vantaggioso se utilizzata per scopi non commerciali ed in tutti i casi semplifica radicalmente lo sviluppo di un’interfaccia utente che sia al contempo funzionale ed esteticamente appetibile, di cui vedete un esempio nella Fig. 8.





Microsoft®

CONOSCIAMO .NET Micro Framework

di Lorenzo Maiorfi
e Gianluca Ruta

Innovactive Engineering

Esaminiamo le librerie e i componenti di terze parti che hanno arricchito le potenzialità del Framework, presentando alcuni esempi riguardanti la gestione del File System e delle comunicazioni wireless su ZigBee e rete GSM/GPRS. Quinta e ultima puntata.

Si conclude qui il nostro corso inerente alla piattaforma di sviluppo .NET Micro Framework; in questa puntata finale prenderemo in esame alcune delle più significative librerie ed alcuni dei più diffusi componenti software di terze parti che hanno arricchito le funzionalità del Micro Framework stesso, per quanto concerne sia l'interfacciamento con specifici dispositivi hardware, sia l'estensione di servizi presenti nella runtime. Tutto quello che prenderemo in esame in questa puntata è basato su materiale open-source reperibile nel portale Codeplex (<http://www.codeplex.com>). L'accesso a questo portale consente sia

di scaricare le varie versioni pubblicate, sia di collegarsi al sistema di controllo dei sorgenti per accedere anche agli aggiornamenti disponibili tra un rilascio ed il successivo o, se autorizzati dagli amministratori del progetto, di partecipare attivamente alla scrittura del codice di quest'ultimo.

GESTIONE DEL FILE SYSTEM

Uno dei sottosistemi di gran lunga più utilizzati nell'ambito dello sviluppo di applicazioni embedded è senza dubbio quello relativo all'immagazzinamento di dati non volatili. Ad esempio, in un'applicazione di data-logging è

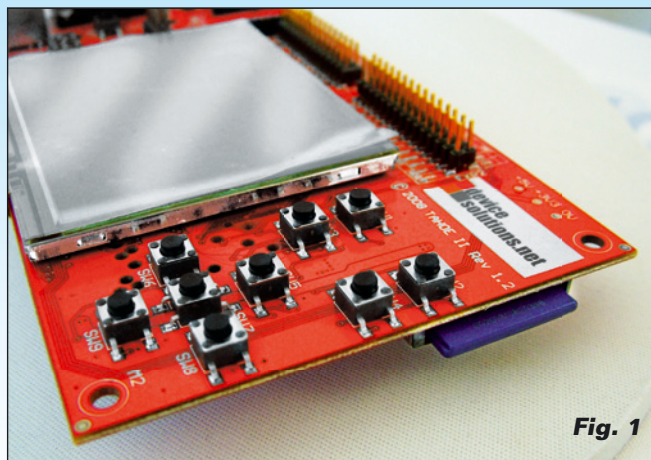


Fig. 1

imperativo che i dati raccolti siano conservati anche a seguito di un riavvio del dispositivo o della scarica completa delle batterie. Anche se in linea di principio è possibile utilizzare un qualsiasi tipo di memoria di massa, da un hard-disk ad una EEPROM, la tendenza attuale spinge verso l'utilizzo di memorie Flash, contenute in schede SD, MMC, PenDrive USB, ecc. A seconda del tipo di dispositivo utilizzato, il .NET Micro Framework rende disponibile l'accesso a memorie di massa (tipicamente memorie Flash), attraverso i servizi di un vero e proprio File System. Gli standard supportati sono al momento solo FAT e FAT32, anche se nuovi "porting" del Micro Framework (ovvero nuove implementazioni della runtime e delle librerie per una specifica architettura di microcontrollori) potrebbero consentire l'utilizzo di altri standard, quali NTFS o ext3, ad esempio.

Come per la stragrande maggioranza dei File System, il modello di interazione con i contenuti della memoria di massa avviene tramite l'accesso ad una struttura gerarchica fatta di file e cartelle. La radice di tale gerarchia prende il nome di "\", ed è effettivamente presente come primo carattere di ogni stringa che esprime un "percorso" all'interno dell'albero del File System. Nel caso della scheda TahoeII della DeviceSolutions (nella Fig. 1 si veda il particolare della scheda con lo slot SD), il percorso che fa capo allo slot SD presente sulla scheda stessa prende il nome di "\\SD". Tale nome potrebbe essere diverso su dispositivi differenti.

Le principali classi del Framework dedicate alla gestione del File System, contenute all'interno dell'assembly "System.IO" sono le seguenti:

- **Path**; espone molti metodi statici per la

gestione dei percorsi; consente ad esempio di estrarre le varie parti di un path gerarchico completo così come di comporre path complessi a partire dalle loro sottoparti;

- **Directory**; espone per lo più metodi statici per l'enumerazione di file e cartelle contenuti all'interno di un determinato percorso; non rappresenta un'istanza di directory vera e propria, per la quale è previsto l'uso della classe **DirectoryInfo**;
- **File**; espone metodi statici per la manipolazione di file (copia, ridenominazione, cancellazione, apertura, ecc.);
- **DirectoryInfo**; rappresenta una specifica istanza di directory; consente operazioni relative alla creazione di sub-directory, cancellazione, ecc.;
- **FileInfo**; rappresenta una specifica istanza di file; consente operazioni di creazione ed eliminazione, oltre ad esporre proprietà importanti quali la lunghezza espressa in byte.

L'accesso a tutte le operazioni di lettura e scrittura di dati all'interno di un determinato file, avviene tramite l'uso di un apposito Stream, denominato **FileStream**. Tramite le classi di supporto specializzate nella lettura e scrittura di Stream (rispettivamente *StreamReader* e *StreamWriter*) è possibile effettuare tutte le più comuni operazioni, come illustrato nel frammento di codice contenuto nel **Listato 1**; questo è chiaramente ispirato ad un data-logger, in cui vengono scorsi in maniera gerarchica tutte le cartelle e tutti i file presenti nel File System, alla ricerca di un ben preciso file ("datalog.txt", nel nostro esempio), al quale viene poi aggiunta una riga di testo composta da un progressivo di riga e dalla tensione misurata dal convertitore ADC all'ingresso "AUX" della scheda.

L'applicazione illustrata è costituita da due metodi principali: **enumerateDirectories()** e **updateDataLog()**. Il primo utilizza le chiamate a **Directory.GetFiles()** e **Directory.GetDirectories()** rispettivamente per scorrere tutti i file e le sotto-cartelle della directory corrente. Il secondo metodo apre dapprima in sola lettura il file "datalog.txt", ne conta le righe utilizzando una stringa intermedia come variabile di appoggio e successivamente lo chiude e riapre in modalità di scrittura per

Listato 1

```

1: public class Program
2: {
3:     // stringa per indentazione
4:     const string INDENT="  ";
5:
6:     public static void Main()
7:     {
8:         // chiama ricorsivamente l'enumerazione
9:
10:        enumerateDirectories(@"");
11:
12:        while (true)
13:        {
14:            Thread.Sleep(1000);
15:        }
16:    }
17:
18:    static string _tabs = string.Empty;
19:
20:
21:    static void enumerateDirectories(string path)
22:    {
23:        // stampa il nome della directory
24:        Debug.Print(_tabs + "[" + path + "]");
25:
26:        _tabs += INDENT;
27:
28:        // Scorre i file...
29:        foreach(string file in Directory.GetFiles(path))
30:        {
31:            Debug.Print(_tabs+Path.GetFileName(file));
32:
33:            // se viene trovato il file datalog.txt...
34:            if (Path.GetFileName(file).ToLower() == "datalog.txt")
35:            {
36:                // aggiungo una riga con la tensione misurata all'ingresso aux
37:                updateDataLog(file);
38:            }
39:        }
40:
41:        // scorro le directory figlie
42:        foreach (string dir in Directory.GetDirectories(path))
43:        {
44:            // ricorsivamente
45:            enumerateDirectories(dir);
46:        }
47:
48:        Debug.Print(string.Empty);
49:
50:        if (_tabs.Length >= INDENT.Length) _tabs = _tabs.Substring(0, _tabs.Length - INDENT.Length);
51:    }
52:
53:    private static void updateDataLog(string file)
54:    {
55:        int rownum = 0;
56:        string txt;
57:
58:        // Apro il file in lettura e conto le righe
59:        using (StreamReader sr = new StreamReader(File.Open(file, FileMode.
OpenOrCreate, FileAccess.Read)))
60:        {
61:            txt=sr.ReadToEnd();
62:        }
63:
64:        rownum = txt.Split('\n').Length - 1;
65:
66:        // riapro il file, stavolta in scrittura
67:        using (StreamWriter sw = new StreamWriter(File.Open(file, FileMode.Append, FileAccess.Write)))
68:        {
69:            // e aggiungo una riga con progressivo e aux
70:            txt = (rownum+1).ToString() + ";" + TahoeII.Tsc2046.ReadAux().ToString("G");
71:
72:            sw.WriteLine(txt);
73:        }
74:
75:    }
76:
77: }

```

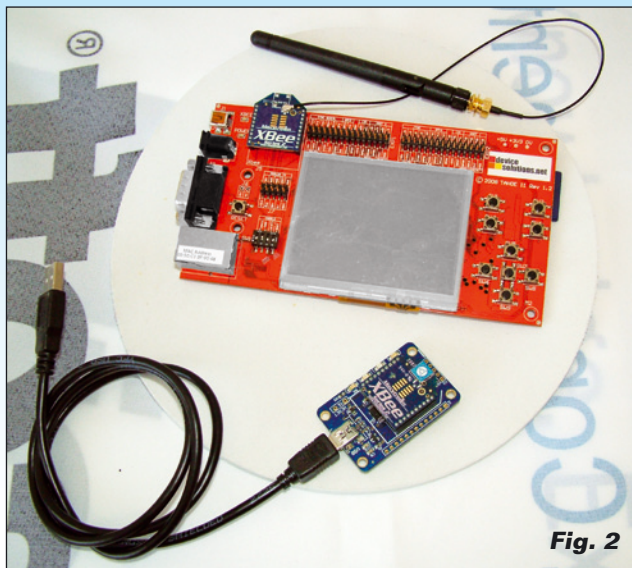


Fig. 2

aggiungere una riga di testo. È opportuno notare che, a riga 59, viene utilizzata l'opzione *FileMode.OpenOrCreate* per fare in modo che, qualora il file non esista (caso che non si applica al nostro demo in quanto il metodo viene eseguito solamente se il file viene effettivamente rilevato) esso venga creato in occasione della prima apertura. Merita un'ultima osservazione, inoltre, quanto presente alla riga 10, in cui il percorso radice del File System viene indicato con `@\"` anziché semplicemente con `\"`. Tale notazione non dipende dall'API relativa al file-system, bensì dal linguaggio C#, che utilizza `\"` come carattere speciale, utilizzato come prefisso di altri caratteri allo scopo di esprimere in forma testuale caratteri non stampabili: ad esempio `\\n`, messo ad indicare il carattere "newline". Qualora si volesse indicare all'interno di una stringa espressa come costante nel codice sorgente lo stesso carattere `\"`, è possibile utilizzare la sequenza `\\\"` oppure, come indicato nell'esempio appena illustrato, anteporre `@` alla stringa o al carattere costante, il quale, interpretato solo dal preprocessore del compilatore, disabilita la gestione del carattere speciale `\"` nella stringa che segue.

COMUNICAZIONI WIRELESS CON MODULI XBEE

Uno dei campi tecnologici in cui è più evidente il fermento da parte di chi realizza sistemi per soluzioni embedded, è senza dubbio quello che riguarda le comunicazioni via radio. Gli standard più utilizzati nei sistemi attuali sono Wi-Fi, Bluetooth e ZigBee, ciascuno con le proprie peculiarità in termini di funzionalità,

velocità, consumi, complessità, ecc.

In particolare, il protocollo che in questo momento rappresenta il compromesso più interessante nell'ambito dello sviluppo di soluzioni embedded è senza dubbio lo ZigBee (al quale abbiamo dedicato un corso nei fascicoli tra il 133 e il 136, cui rimandiamo per gli approfondimenti); quest'ultimo opera nelle frequenze radio assegnate per scopi industriali, scientifici e medici (ISM), distribuite nelle bande intorno ad 868 MHz in Europa, 915 MHz negli Stati Uniti e 2,4 GHz nella maggior parte del resto del mondo. Nel 2005 il costo stimato per il ricetrasmittitore di un nodo ZigBee era di circa 1,10 \$ per il produttore, contando grossi volumi. La maggior parte dei dispositivi ZigBee richiede però anche un microcontrollore, che fa alzare il costo totale. Per fare un confronto, sappiate che quando fu lanciato (nel 1998) il Bluetooth, per esso si prevedeva un costo di 4÷6 \$ per grandi volumi, mentre il prezzo attuale per la fascia consumer è oggi sotto i 3 dollari U.S.A.

I protocolli ZigBee sono progettati per l'uso in applicazioni embedded che richiedano un basso transfer-rate e bassi consumi. L'obiettivo attuale dello ZigBee è di definire una "wireless mesh network" economica e autogestita che possa essere utilizzata per scopi quali il controllo industriale, le reti di sensori, domotica, telecomunicazioni, ecc. La rete risultante avrà un consumo energetico talmente basso da poter funzionare per uno o due anni sfruttando la batteria incorporata nei singoli nodi. Ci sono tre differenti tipi di nodo ZigBee:

- **Coordinator (ZC):** è il dispositivo più "intelligente" tra quelli disponibili, costituisce la radice di una rete ZigBee e può operare da ponte tra più reti; ce ne può essere uno solo in ogni rete ed è in grado di memorizzare informazioni riguardo alla sua rete ed inoltre può agire come deposito per le chiavi di sicurezza;
- **Router (ZR):** questi dispositivi agiscono come router intermedi, passando i dati da e verso altri dispositivi;
- **End Device (ZED):** includono solo le funzionalità minime per dialogare con il suo nodo parente (Coordinator o Router), non possono trasmettere dati provenienti da altri dispositivi; sono i nodi che richiedono il minor quantitativo di memoria e quindi

risultano spesso più economici rispetto ai ZR o ai ZC.

I moduli ZigBee più diffusi sono senza dubbio gli Xbee, prodotti dalla società statunitense Digi. Tali moduli contengono al loro interno un transceiver in grado di operare sulle frequenze stabilite dal protocollo, con una potenza che a seconda dei modelli varia da 1 a circa 60 mW, un'antenna integrata o un connettore per antenne esterne ed un microcontrollore. I moduli Xbee possono essere collegati ad altri dispositivi embedded, tipicamente all'interno di una scheda a microcontrollore, tramite un'interfaccia UART CMOS con livelli a 3,3 V. Tali moduli possono inoltre agire come veri e propri sensori/attuatori remoti indipendenti grazie al fatto che le funzionalità di gestione degli I/O on-board sono pilotabili tramite comunicazione wireless. Questi I/O permettono ad esempio di rilevare lo stato di ingressi analogici o digitali, pilotare uscite digitali e addirittura generare segnali modulati tramite PWM, particolarmente utili ai fini del controllo di sistemi di potenza.

I moduli Xbee prevedono due modalità di comunicazione seriale con il sistema host (ossia tipicamente con il microcontrollore che fa capo alla scheda che li ospita): una detta "trasparente" ed una modalità "API".

Utilizzando la prima, due moduli possono essere messi in comunicazione diretta in modo tale che il contenuto del buffer di ingresso di un modulo si rifletta automaticamente nel buffer di uscita dell'altro, e viceversa. Il canale di comunicazione così inizializzato consente quindi di "remotizzare" una qualsiasi comunicazione seriale (con livelli a 3,3 V) semplicemente inserendo tra i due interlocutori due moduli wireless. Per far sì che un host possa interagire con il modulo locale sfruttando la medesima interfaccia seriale utilizzata per la trasmissione remota, viene sfruttata una particolare sequenza di caratteri (per impostazione predefinita sono "+++") che, eseguita entro un tempo limite, mette il modulo in modalità "comando". In questo stato è ad esempio possibile eseguire comandi AT di servizio, quali quelli che riguardano le funzionalità "mesh", come il discovery di altri nodi della stessa rete nei paraggi.

La modalità API consente invece di imple-

mentare una comunicazione a pacchetti in cui ciascun pacchetto sia descritto da una ben precisa struttura a frame specifica del tipo di operazione che si vuole compiere. Sebbene sia più complessa da gestire da parte di un'applicazione embedded, tale modalità consente un margine di flessibilità di gran lunga superiore a quello disponibile in modalità trasparente. Inoltre, alcune funzionalità interessanti, come la remotizzazione degli I/O, sono disponibili solamente in modalità API.

L'utilizzo dei moduli Xbee in modalità API all'interno di un firmware sviluppato con .NET Micro Framework è reso particolarmente semplice dalla libreria open-source "MFToolkit", disponibile all'indirizzo <http://mftoolkit.codeplex.com>.

Vediamo un esempio di applicazione divisa in due parti: un'applicazione console in esecuzione all'interno di un PC ed un firmware per la scheda di sviluppo TahoeII della DeviceSolutions, già utilizzata in precedenza. Per poter eseguire il demo completo, avremo bisogno di due moduli Xbee configurati con la stessa versione di firmware (ZIGBEE API per moduli di tipo XB24-ZB) in modo tale da appartenere alla stessa sottorete (PAN, nel gergo ZigBee). La Fig. 2 mostra la configurazione circuitale usata per questi test. La configurazione dei

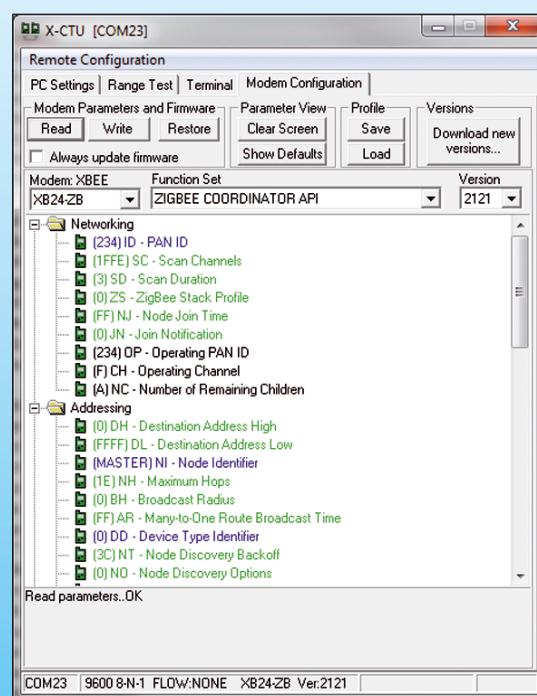


Fig. 3

moduli Xbee avviene per tramite di un tool denominato X-CTU; nella Fig. 3 è illustrata un'immagine di questo strumento relativa alla configurazione del dispositivo "coordinator". Nell'esempio in questione, utilizzeremo come modulo Xbee "coordinator" quello collegato al PC, mentre configureremo quello installato sulla scheda di sviluppo TahoeII come "router/end-device". Per collegare il modulo Xbee ad un PC è necessario utilizzare un'apposita daughter-board, che tipicamente incapsula un chip FTDI, Prolific o SiLabs per realizzare funzionalità USB-UART-Bridge.

Vale la pena di sottolineare, infine, che il sorgente della libreria per la gestione della comunicazione con i moduli Xbee in modalità API è condivisa tra i due progetti rispettivamente destinati al .NET Micro Framework e al .NET Framework "senior".

Il sorgente relativo al Main() dell'applicazione desktop è riportato nel **frammento di codice visibile nel Listato 2**; nel metodo viene semplicemente avviato un thread di background che ha come unico scopo quello di eseguire il metodo "RunMasterModule".

Il metodo eseguito dal thread si occupa dell'inizializzazione del modulo (collegato alla porta seriale COM23 del PC nel nostro

esempio) e dell'esecuzione di un comando di tipo "Node Discovery", il cui scopo è quello di effettuare la ricerca, all'interno di uno spazio delimitato dalla portata del ricetrasmittitore, di altri nodi iscritti alla propria PAN (nel nostro caso il PAN ID è 234). Il sorgente del metodo in questione è riportato nel **Listato 3**. Ogni comunicazione che il modulo Xbee effettua nei confronti dell'host viene gestita da un evento denominato "FrameReceived". All'interno del gestore di evento è possibile eseguire tutte le operazioni di decodifica necessarie, come ad esempio quelle relative all'individuazione dell'identificativo della richiesta cui la risposta si riferisce o al riconoscimento del tipo di risposta ricevuta. Nel nostro esempio gestiamo tre tipi di risposta: la prima è quella relativa alla scoperta di nodi all'interno della stessa PAN (per la quale il modulo ci invia più istanze, una per ciascun nodo), la seconda è quella relativa all'esito di una trasmissione (rappresentata dalla stringa "Hello") che il demo effettua nei confronti di un ben preciso nodo (identificato dal nome

"SLAVE", secondo quanto previsto dal parametro di configurazione "NI" stabilito dal protocollo) e la terza riguarda la comunicazione effettuata da parte del firmware in esecuzione sulla scheda TahoeII, come verrà illustrato a breve. Il codice relativo al gestore di evento in questione è riportato nel **Listato 4**. L'interfaccia utente dell'applicazione in esecuzione sul PC è quella visibile nella Fig. 4. Il codice relativo alla parte embedded della soluzione descritta (si veda il frammento di codice contenuto nel **Listato 5**) è ovviamente molto simile

Listato 2

```
1: class Program
2: {
3:     static void Main(string[] args)
4:     {
5:         // Lancio un thread che gestisca il modulo Xbee
6:         Thread thd1 = new Thread(new ThreadStart(RunMasterModule));
7:         thd1.IsBackground = true;
8:         thd1.Start();
9:
10:        // Con il thread principale resto in attesa...
11:        Console.ReadLine();
12:    }
13:
14:    // ...
15: }
```

Listato 3

```
1: static void RunMasterModule()
2: {
3:     using (XBee xbee = new XBee("COM23", 9600, ApiType.Enabled))
4:     {
5:         xbee.FrameReceived += new FrameReceivedEventHandler(xbee_
6:             FrameReceived);
7:         xbee.Open();
8:
9:         // esecuzione del comando "Node Discovery"
10:        AtCommand at = new NodeDiscoverCommand();
11:        xbee.Execute(at);
12:
13:        // lascio attivo il modulo Xbee solo 10 secondi
14:        Thread.Sleep(10 * 1000);
15:
16:        // ...poi lo spengo
17:        xbee.StopReceiveData();
18:        Console.WriteLine("stopped master");
19:    }
20: }
```

Listato 4

```

1: static void xbee_FrameReceived(object sender, FrameReceivedEventArgs e)
2: {
3:     Console.WriteLine("Ricevuta la seguente risposta: " + e.Response);
4:
5:     // La risposta riguarda il comando di Node Discovery?
6:     if (e.Response is AtCommandResponse &&
7:         (e.Response as AtCommandResponse).Command == new NodeDiscoverCommand().Command)
8:     {
9:         // interpreto il pacchetto ricevuto (ne arriva uno per nodo)
10:        NodeDiscover nd = NodeDiscover.Parse((e.Response as AtCommandResponse));
11:
12:        // Se il nodo rilevato si chiama "SLAVE"...
13:        if (nd != null && nd.ShortAddress != null)
14:        {
15:            Console.WriteLine(nd);
16:
17:            if (nd.NodeIdentifier == "SLAVE")
18:            {
19:                Console.WriteLine("Mando la stringa \"Hello\" a SLAVE...");
20:                (sender as XBee).ExecuteNonQuery(
21:                    new ZNetTxRequest(nd.SerialNumber, nd.ShortAddress, Encoding.ASCII.GetBytes("Hello")));
22:            }
23:        }
24:    }
25:
26:    // La risposta riguarda l'esito della trasmissione di "Hello"?
27:    if (e.Response is ZNetTxStatusResponse)
28:    {
29:        Console.WriteLine((e.Response as ZNetTxStatusResponse).DeliveryStatus.ToString());
30:    }
31:
32:    // La risposta riguarda una trasmissione da parte di un altro nodo?
33:    if (e.Response is ZNetRxResponse)
34:    {
35:        Console.WriteLine("Ho ricevuto la stringa: ");
36:        Console.WriteLine(Encoding.ASCII.GetString((e.Response as ZNetRxResponse).Value));
37:    }
38:
39: }

```

a quello appena visto, ad eccezione del fatto che il nodo "SLAVE" di fatto non effettua alcuna operazione di "discovery", limitandosi a rispondere con la stringa "Hello by MF" a ciascun nodo che invii un qualche tipo di comunicazione.

In questo caso, l'unico output dell'applicazione è visibile solo eseguendo la stessa in modalità di debug all'interno dell'ambiente di sviluppo, come illustrato nella Fig. 5.

UTILIZZO DI MODULI GPS/GSM

Una delle funzionalità sicuramente più interessanti nell'ambito dello sviluppo di soluzione embedded, riguarda la possibilità da parte di un dispositivo di utilizzare un modulo GPS per conoscere la propria posizione geografica, particolarmente utile in applicazioni fruibili

"in mobilità". Tale famiglia di applicazioni beneficia spesso della possibilità di effettuare comunicazioni remote tramite la rete GSM/GPRS/UMTS, finalizzate alla fruizione di servizi di rete che richiedono un collegamento ad Internet.

Sul mercato è possibile reperire molti tipi di

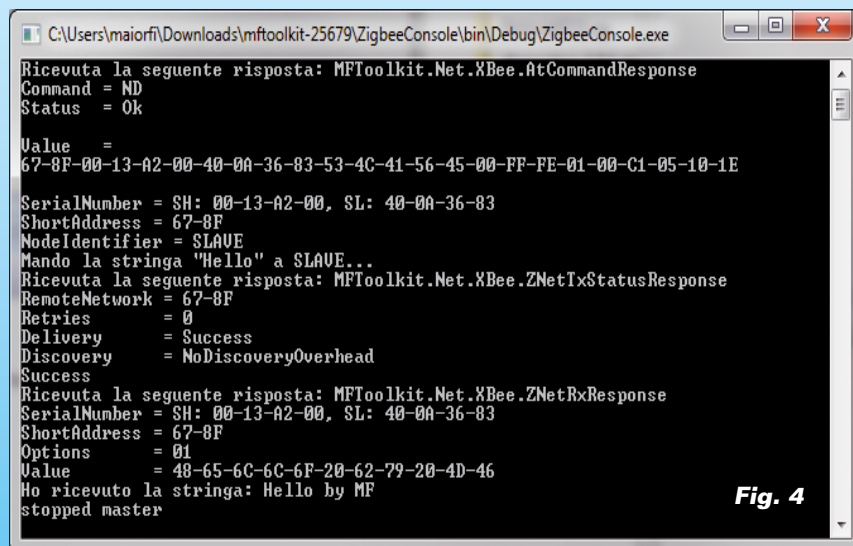


Fig. 4

Listato 5

```

1: public class Program
2: {
3:     public static void Main()
4:     {
5:         using (XBee xbee = new XBee("COM2", 9600, ApiType.Enabled))
6:         {
7:             xbee.FrameReceived += new FrameReceivedEventHandler(xbee_OnPacketReceived);
8:             xbee.Open();
9:
10:            // Attivo il modulo Xbee solo per 60 secondi
11:            Thread.Sleep(10 * 60 * 1000);
12:
13:            xbee.StopReceiveData();
14:        }
15:    }
16:
17:    static void xbee_OnPacketReceived(object sender, FrameReceivedEventArgs e)
18:    {
19:        XBeeResponse response = e.Response;
20:
21:        // Debug risposta arrivata
22:        Debug.Print(response.ToString());
23:
24:        // La risposta riguarda una trasmissione da parte di un altro nodo?
25:        if (e.Response is ZNetRxResponse)
26:        {
27:            Debug.Print("Received : " +
28:                new string(System.Text.Encoding.UTF8.GetChars((e.Response as ZNetRxResponse).Value)));
29:
30:            // Rispondo al nodo chiamante con "Hello by MF"
31:            (sender as XBee).ExecuteNonQuery(
32:                new ZNetTxRequest(
33:                    (e.Response as ZNetRxResponse).SerialNumber,
34:                    (e.Response as ZNetRxResponse).ShortAddress,
35:                    Encoding.UTF8.GetBytes("Hello by MF")));
36:        }
37:    }
38: }

```

ricevitori GPS, quasi sempre basati su chipset SiRF, così come molto varia è anche l'offerta di moduli GSM/GPRS. Una buona soluzione è rappresentata da moduli che integrano entrambe le funzionalità, come ad esempio il Telit GM862-GPS, utilizzato in numerosi progetti per la sua grande programmabilità. Il modulo in questione incorpora infatti al proprio interno un microcontrollore che implementa funzionalità complesse quali l'esecuzione di applicazioni scritte in Python, un linguaggio di alto livello molto utilizzato nelle comunità degli sviluppatori di tutto il mondo, e la gestione di un vero e proprio stack di rete TCP/IP, con il quale il modulo permette al proprio host di essere collegato ad Internet via GPRS. L'interfaccia di comunicazione tra scheda a

microcontrollore host e modulo GSM/GPS è seriale ed in particolare basata su comandi "AT", secondo quanto previsto dallo standard Hayes del 1977.

Per utilizzare tale interfaccia all'interno di un'applicazione sviluppata con il .NET Micro Framework, è pertanto sufficiente allestire una comunicazione seriale asincrona ed utilizzare il set dei comandi riconosciuti dal

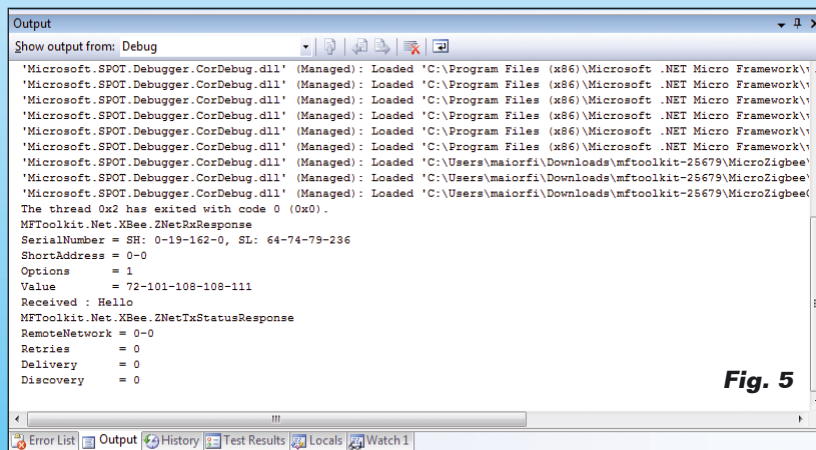


Fig. 5

modulo. Per i nostri esempi abbiamo utilizzato il modulo in questione in una configurazione hardware che prevede una daughter board, prodotta dalla Sparkfun, la quale, oltre a disporre di un connettore con pin a passo 0,1" per collegarvi il modulo, si occupa di alimentare quest'ultimo e gestire i due LED di diagnostica (si veda la **Fig. 6**).

Il dispositivo prevede inoltre due connettori coassiali MMCX per l'utilizzo di due antenne esterne distinte per i due sotto-moduli GPS e GSM/GPRS. Da un punto di vista software, lo sviluppo viene enormemente semplificato dalla disponibilità di una libreria open-source denominata "MicroGM862", inclusa recentemente, come per la libreria per la gestione dei moduli Xbee, all'interno del progetto MFToolkit, disponibile come già detto sul sito Codeplex all'indirizzo <http://mftoolkit.codeplex.com>. La libreria mette a disposizione una classe denominata GM862GPS, che espone tutte le funzionalità relative alla registrazione all'interno della rete GSM e GPRS, alla gestione degli SMS e del GPS, fino alla gestione dello stack di rete, con il quale è ad esempio possibile effettuare richieste HTTP via GPRS.

L'inizializzazione dell'oggetto GM862GPS avviene tramite un costruttore che prevede il passaggio di un'istanza della classe AT_Interface, anch'essa definita all'interno della libreria, responsabile della gestione della comunicazione seriale con l'host, come illustrato nella seguente riga di codice:

```
GM862GPS GM862 = new GM862GPS(new AT_Interface("COM1", 115200));
```

Una volta creato l'oggetto "driver", è possibile effettuare le operazioni di registrazione all'interno della rete GSM e GPRS come illustrato nel frammento di codice contenuto nel **Listato 6**. Analogamente, è possibile inizializzare il modulo per la gestione dei sotto-sistemi relativi rispettivamente a SMS e GPS, mediante il frammento di codice contenuto nel **Listato 7**. Una volta completata la procedura di inizializzazione, eventualmente preceduta da un reset hardware del modulo attivabile mediante l'utilizzo di un'apposita OutputPort collegata al pin "RESET" del modulo, è possibile accedere alle funzionalità di alto livello relative ai vari sotto-sistemi, il più semplice dei quali è senza dubbio quello

Listato 6

```
1: // Selezione banda GSM-DCS
2: if (!GM862.GSM.SelectNetworkBand _
   (GSM.NetworkBands GSM900_DCS1800))
3: throw new Exception("Selezione banda fallita!");
4:
5: // Sottoscrizione evento OnPinRequest
6: GM862.GSM.OnPinRequest = new _
   GSM.PinRequestHandler(delegate(String PINType)
7: {
8:     if (PINType == "SIM PIN")
9:         return "0000";
10:
11:     if (PINType == "SIM PUK")
12:         return "05969374, 0000";
13:
14: throw new Exception("Richiesta PIN sconosciuta");
15: });
16:
17: // Sottoscrizione evento ricezione chiamata
18: GM862.GSM.OnReceivingCall = new _
   GSM.ReceivingCallHandler(delegate()
19: {
20:     Debug.Print("Driiiii!");
21:     Thread.Sleep(500);
22: });
23:
24: // Roaming GSM
25: GM862.GSM.AllowRoaming = true;
26:
27: // Inizializzazione GSM
28: GM862.GSM.Initialize();
29:
30: // Roaming GPRS
31: GM862.GPRS.AllowRoaming = true;
32:
33: // Inizializzazione GPRS
34: GM862.GPRS.Initialize();
```

relativo al GPS, utilizzabile mediante il codice scritto nel **Listato 8**, che illustra come estrarre il dato relativo rispettivamente a qualità del rilevamento GPS, numero di satelliti visibili,

Listato 7

```
1: // Inizializzazione text messaging
2: GM862.TextMessaging.Initialize();
3:
4: // Inizializzazione GPS
5: GM862.GPS.Initialize();
```

Listato 8

```
1: // Parametri GPS
2: byte gpsFix;
3: byte gpsNoSat;
4: double gpsLat;
5: double gpsLon;
6: double gpsSpeed;
7: DateTime gpsDateTime;
8:
9: GM862.GPS.ReadGPSData(out gpsFix, out gpsNoSat,
   out gpsLat, out gpsLon, out gpsSpeed, out gpsDate
   Time);
10: Debug.Print((gpsFix > 0) ? "GPS FIX OK" : "NO GPS FIX");
```

Listato 9

```

1: GM862.TextMessaging.OnRecievedTextMessage =
2:   new TextMessaging.RecievedTextMessageHandler(delegate(String Memory, int Location)
3:   {
4:       TextMessaging.TextMessage TextMessage;
5:
6:       Debug.Print("SMS ricevuto!");
7:
8:       // Lettura contenuto messaggio
9:       if (GM862.TextMessaging.ReadTextMessage(Memory, Location, out TextMessage))
10:      {
11:          Debug.Print("Da: " + TextMessage.Originator);
12:          Debug.Print("Messaggio: " + TextMessage.Message);
13:
14:          if (Memory.IndexOf("SM") != -1)
15:              Debug.Print("Eliminazione messaggio: " +
16:                  (GM862.TextMessaging.DeleteMessage("SM", TextMessage.Location) ? "OK" : "ERROR"));
17:
18:          Debug.Print("Rispondo...");
19:          GM862.TextMessaging.SendTextMessage(TextMessage.Originator, "Roger");
20:
21:      }
22:
23:  })

```

latitudine, longitudine, velocità e data/ora del rilevamento (in formato UTC).

Un po' più complessa, ma non di molto, è la gestione degli SMS. Le funzionalità principali esposte dalla libreria consentono di enumerare i messaggi contenuti nella SIM o nella memoria del modulo, leggere il contenuto

di un messaggio, eliminare un messaggio, inviare un nuovo messaggio e ricevere una notifica a seguito della ricezione di un nuovo messaggio.

Nel frammento di codice contenuto nel **Listato 9**, ad esempio, viene registrato un gestore di evento relativo alla ricezione di un nuovo

Listato 10

```

1: // Rimaniamo in attesa della disponibilità della rete GSM/GPRS
2: while (true)
3: {
4:     if (!GM862.GSM.RegistratedOnGSMNetwork()) continue;
5:     if (!GM862.GPRS.RegistratedOnGPRSNetwork()) continue;
6:     break;
7: }
8:
9: Debug.Print("GSM/GPRS pronti");
10:
11: // Configurazione parametri accesso GPRS
12: if (!GM862.GPRS.SetContextConfiguration(1, "web.omnitel.it", "", "", "0.0.0.0"))
13:     throw new Exception("Configurazione GPRS fallita");
14:
15: // Attivazione del ptimo contesto GPRS
16: if (!GM862.GPRS.ActivateContext(1))
17:     throw new Exception("Attivazione GPRS fallita");
18:
19: // Configurazione socket
20: if (!GM862.GPRS.SetSocketConfig(1, 1, 0, 90, 600, 50))
21:     throw new Exception("Configurazione socket fallita (1/2)");
22:
23: if (!GM862.GPRS.SetSocketExtendedConfig(1, 0, 0, 0))
24:     throw new Exception("Configurazione socket fallita (2/2)");
25:
26: if (GM862.Networking.WebRequest(1, "http://search.twitter.com/trends/weekly.json",
27:     String.Empty, out response, "GET", String.Empty, String.Empty))
28: {
29:     Debug.Print(new string(System.Text.Encoding.UTF8.GetChars(response)));
30: }
31: else
32: {
33:     Debug.Print("Chiamata HTTP fallita");
34: }

```

SMS; tale gestore si occupa dapprima della lettura del messaggio (righe 9÷12) e successivamente elimina il messaggio ricevuto (se presente nella memoria di tipo "SM", ossia nella SIM, unica possibilità prevista nell'attuale versione del firmware). Una volta eliminato il messaggio, viene poi composto ed inviato (a riga 19) un nuovo messaggio di risposta al mittente.

L'ultimo esempio che prendiamo in esame si riferisce invece all'esplorazione delle possibilità relative al networking, ed in particolare del supporto al protocollo HTTP che consente ad un'applicazione host di utilizzare il modulo GM862GPS per effettuare una richiesta web via Internet. Nel caso illustrato di seguito, una volta verificata l'avvenuta registrazione alla rete GSM e GPRS (righe 4-5) ed impostati i parametri di configurazione relativi all'accesso GPRS (riga 12) e allo stack di rete (righe 20÷24), l'applicazione effettua una chiamata HTTP di tipo GET verso il servizio API del noto portale di social networking Twitter; lo scopo della chiamata è, nel nostro caso, rileva-

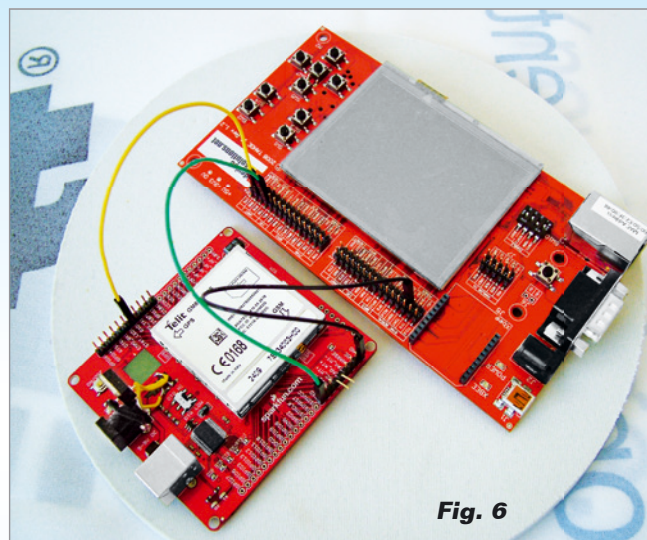


Fig. 6

re le "tendenze" della settimana, riportando, all'interno della finestra di debug dell'ambiente di sviluppo, la risposta ottenuta.

Le istruzioni relative a quest'ultimo esempio sono riportate nel frammento di codice visibile nel **Listato 10**.



Combinatore telefonico GSM

... Mai più spiacevoli sorprese!



cod. FR425

€ 139,00

Dedicato al settore della sicurezza e dell'automazione a distanza, dispone di 4 ingressi d'allarme (con messaggi personalizzabili) e di 4 uscite programmabili con toni DTMF, chiamata telefonica (menu vocale guidato) e comandi SMS. Le uscite possono essere associate ad eventi di allarme, oppure essere gestite da remoto per controllare apparecchi elettrici di vario genere. Possibilità di memorizzare 8 numeri telefonici per l'invio di avvisi d'allarme tramite chiamata vocale, SMS o entrambi. Alimentazione batteria 12 ÷ 14,5Vdc/250mA. SIM e batteria non inclusi.

Batteria di alimentazione consigliata NP08-12 - € 19,50



Completo di contenitore

FUTURA ELETTRONICA

Via Adige, 11 - 21013 GALLARATE (VA)
Tel. 0331/799775 - Fax 0331/778112
www.futurashop.it

Maggiori informazioni e caratteristiche tecniche sono disponibili sul sito www.futurashop.it tramite il quale è anche possibile effettuare acquisti on-line.

Tutti i prezzi si intendono IVA inclusa.