*Un sistema "embedded" è un computer incorporato all'interno di un dispositivo di altro tipo - per esempio: una fotocopiatrice, una lavastoviglie, un'automobile - di cui deve gestire e controllare, normalmente in "real-time", alcune funzionalità. La programmazione dei sistemi embedded è caratterizzata dalla peculiare interfaccia utente, spesso limitata a pochi tasti funzionali integrati da un visore alfanumerico.*

# Introduzione alla programmazione dei sistemi embedded e real-time

*Giorgio Meini*

Ho terminato da pochi giorni, consegnandolo al committente per la fase di testing, il mio ultimo programma ("ultimo" in ordine temporale, da non intendersi come "ultimo" in senso definitivo!). Scritto quasi completamente in ANSI C e parte in linguaggio assembler, non contiene una sola riga di gestione dell'interfaccia utente e, tanto meno, supporta una GUI; inoltre è stato sviluppato utilizzando prevalentemente tool operanti in ambiente DOS. Molti lettori penseranno che sono un programmatore antidiluviano che sbarca il lunario sfruttando una nicchia di mercato incomprensibilmente sopravvissuta all'avvento del progresso, ma non è così: più semplicemente il mio lavoro non consiste nello sviluppare applicazioni per PC (le quali, giustamente, oggi non possono che "girare" sotto Windows 95 o NT) o per workstation UNIX, ma programmi per "sistemi embedded e real-time". In particolare, stando ai contenuti elencati nell'indice di un testo appositamente scritto per il programmatore di tali sistemi **[1]**, si è trattato di un programma assolutamente tipico in questo campo in quanto comprendente - in un unico prodotto console-less - la gestione di comunicazioni su linea seriale e su rete locale Ethernet, l'acquisizione con vincoli temporali di dati in forma analogica e digitale, la misurazione di intervalli di tempo intercorrenti tra eventi successivi e la reazione in tempo reale ad eventi esterni mediante il controllo di dispositivi aventi sia interfaccia digitale che analogica.

Il mio lavoro di programmazione consiste prevalentemente nello sviluppo di software per sistemi applicati al controllo di processo, all'acquisizione e pre-elaborazione di segnali analogici (incluse le immagini) o alla gestione di dispositivi di comunicazione, ma in letteratura sono considerati sistemi embedded soprattutto oggetti quali il dispositivo ABS delle automobili (ovviamente vincolato ad operare in stretto tempo reale!) ed il sintonizzatore digitale del televisore. Caratteristica comune a prodotti così diversi tra loro è la mancanza della tradizionale interfaccia con l'*utente* (schermo, tastiera e mouse), sostituita da una interfaccia con l'*ambiente* costituita da "sensori" che rilevano molteplici informazioni relative al mondo esterno ed "attuatori" che sul mondo esterno agiscono in vario modo. Il citato "manuale" di J. Labrosse **[1]** considera "codice di routine" - proposto nella forma di moduli *"building blocks"* standard ampiamente riutilizzabili - la gestione dei dispositivi sensori ed attuatori (sia analogici che digitali), mentre il processo di controllo in tempo reale del sistema esterno è il cuore di un *embedded system* e come tale il vero oggetto di progettazione e sviluppo in forme originali ("per non inventare ogni volta la ruota...").

Dato che una consistente porzione del codice di una applicazione per PC è oggi dedicato alla gestione delle interazioni con l'utente (indipendentemente dal fatto che sia direttamente scritto dallo sviluppatore, automaticamente generato con tool di tipo visuale o prevalentemente incapsulato in un framework di supporto) l'arte di programmare i sistemi embedded è divenuta nel tempo una cosa "a parte" nelle tecniche e negli strumenti di uso quotidiano. In molti casi lo smaliziato sviluppatore di applicazioni per ambiente GUI avrà l'impressione di un "mondo perduto", di un "parco giurassico" della programmazione dove ancora non si è estinto l'uso di C ed assembler; ma è bene ricordare che

molti argomenti che oggi sono all'ordine del giorno nella programmazione Windows come "multitasking di tipo preemptive", "sincronizzazione e cooperazione tra thread concorrenti" e "accesso protetto a dati condivisi tra processi in esecuzione contemporanea" sono da trent'anni il pane quotidiano dei programmatori di sistemi embedded e real-time, spesso senza il confortante supporto di un sistema operativo degno di tale nome.

**Embedded hardware e real-time software**

Il programmatore di sistemi embedded e real-time è in primo luogo un integratore di hardware ("embedded", cioè incorporato in un sistema ambiente da controllare) e software ("real-time" in quanto i processi di funzionamento del sistema ambiente sono normalmente determinati in tempo reale da eventi sia interni che esterni al sistema stesso). Il primo requisito per un *embedded system programmer* è la conoscenza congiunta di metodologie e strumenti tipici dello sviluppatore software e dei principi di funzionamento e di interfacciamento dei dispositivi hardware.
Normalmente un prodotto viene progettato contemporaneamente sia sotto l'aspetto hardware che software: spesso è il più flessibile software che si adatta alla vincolante rigidità hardware, ma in molti casi le componenti hardware devono essere scelte ed interconnesse in funzione del software che dovrà gestirle. Un semplice esempio: dovendo realizzare un tachimetro (misuratore di velocità) disponendo di un sensore che genera un impulso per ogni unità di spazio percorsa è possibile sia contare da programma gli impulsi generati in una unità di tempo che disporre un contatore esterno da interrogare ed azzerare una volta trascorsa la stessa unità di tempo; il criterio di progettazione in questo caso dipenderà sia da considerazioni tecniche (per esempio il periodo di tempo medio intercorrente tra due impulsi successivi) che, in particolare nel caso di produzione in grandi numeri, economiche (il costo del contatore hardware in confronto al costo zero della replicabilità del software). In ogni caso il programmatore non potrà prescindere da una precisa conoscenza del funzionamento dei dispositivi hardware (si veda a questo proposito anche il riquadro **"Chips programming"**).
Il programmatore di sistemi embedded esperto aggiungerà poi alla conoscenza integrata di hardware e software una vasta cultura "sistemica" relativa al contesto operativo dell'unità di controllo: se la specializzazione è in questo caso inevitabile - per esempio il mio lavoro si svolge principalmente nel campo dei sistemi automatici di ispezione visiva della qualità della produzione industriale - non si deve dimenticare che uno dei lati interessanti di questa particolare attività professionale del settore informatico consiste nella infinita varietà delle possibili e futuribili applicazioni. Inoltre la progettazione e realizzazione di un software di controllo anche solo di media complessità coinvolge necessariamente conoscenze di tipo ingegneristico, fisico, matematico ed informatico (teoria dei sistemi e del controllo, modelli differenziali continui e discreti, teoria dei segnali e dell'informazione, calcolo numerico approssimato in precisione finita, filtri digitali ed elaborazione statistica dei dati, elaborazione e compressione delle immagini, ...) che in altri campi di attività sono inesorabilmente relegate ai ricordi degli anni di università.
Ma che cosa programma lo sviluppatore di software per sistemi embedded? Generalmente un *microcontroller*: un microprocessore integrato *on chip* da vari dispositivi di I/O (seriale, digitale, analogico, ...) o di altro tipo (contatori, timer, gestori delle interruzioni esterne, ...). La varietà delle applicazioni - dal controllo del livello e della temperatura del liquido contenuto in un serbatoio all'analisi in tempo reale delle immagini riprese da una telecamera - ha come conseguenza una estrema variabilità della potenza computazionale impiegata e, se nel primo esempio è decisamente più che sufficiente un microcontroller basato su microprocessore Intel 386, nel secondo caso è assolutamente necessario un processore RISC ad elevate prestazioni integrato da una unità DSP.
Normalmente il software dei sistemi embedded deve essere eseguito in tempo reale: la risposta del sistema ad alcuni stimoli deve tassativamente iniziare e terminare entro limiti di tempo predefiniti. Tradizionalmente esistono due modalità organizzative per la stesura di un programma sottoposto a

vincoli real-time in esecuzione **[1]**: la tecnica *foreback/foreground* ed il ricorso ad un kernel real-time per la gestione multitasking. Nel primo caso - tipico dei sistemi più semplici - il programma principale consiste essenzialmente in un ciclo infinito (*foreground superloop*) la cui ripetizione asincrona rispetto agli eventi esterni è continuamente intervallata dall'attivazione in *foreback* - sincrona rispetto agli stimoli - di singole funzioni ISR (*Interrupt Service Routine)* di gestione delle interruzioni generate dai dispositivi di I/O e di supporto (timer, ...). La comunicazione tra le funzioni invocate all'interno del ciclo infinito e le funzioni di gestione delle interruzioni avviene in questo caso mediante variabili globali condivise. Nel secondo caso - riservato alla realizzazione di sistemi più complessi - si dispone di un vero e proprio (micro) sistema operativo dedicato alla schedulazione temporale dei singoli task in esecuzione ed alla gestione delle primitive di comunicazione e di sincronizzazione tra task; in questo caso il programma consiste in un insieme di task (o thread) ciascuno dei quali può essere attivato da una diversa condizione: interruzione esterna, vincolo temporale, rilascio di una risorsa condivisa, sincronizzazione inter-task.

**Sistemi di sviluppo**

Quale linguaggio di programmazione impiega lo sviluppatore di software per sistemi embedded? Tramontata definitivamente, soprattutto a causa della crescente complessità delle applicazioni, l'era del "solo assembler", moltissimi programmi di controllo sono scritti in C, ma non mancano certo anche in questo campo - e sono anzi in rapida crescita - i fautori del ricorso alle funzionalità avanzate dei linguaggi orientati agli oggetti come C++ ed Ada. Se C è stato preferito fino ad oggi per l'efficienza del codice compilato e la possibilità di collegarlo facilmente a codice scritto direttamente in assembler, C++ è molto apprezzato per le sue funzionalità orientate agli oggetti che consentono sia una agevole modellizzazione delle componenti ambientali con le quali il sistema interagisce (non si dimentichi che C++ è nato come *front-end* C per applicazioni di simulazione) che una adeguata incapsulazione del software di gestione dei dispositivi sensori ed attuatori. Oltre ad avere queste stesse caratteristiche, Ada (un linguaggio progettato appositamente per le applicazioni embedded e recentemente rinnovato in veste *object-oriented*) fornisce costrutti di gestione multitasking completamente integrati nella struttura sintattica e semantica del linguaggio. Inoltre entrambi (C++ ed Ada) implementano meccanismi di *exception handling* che in questo contesto si rivelano particolarmente importanti (cfr. il riquadro **"Sicurezza e continuità di funzionamento delle applicazioni critiche"**).

Nel caso che il microcontroller da programmare sia basato su architettura Intel x86 è senz'altro possibile ricorrere ai tradizionali strumenti di sviluppo (compilatori, assemblatori, linker, librerie, debugger ...) per PC sia a 16 che a 32 bit, tenendo nella dovuta considerazione il fatto che il programma definitivo funzionerà senza il supporto del sistema operativo e, probabilmente, dello stesso BIOS (a meno di non ricorrere a versioni real-time e "ROMmabili" di DOS e BIOS quali i prodotti "Embedded DOS" e "Embedded BIOS" della General Software). In questa ottica utilizzando il C è quasi sempre necessario modificare il modulo *crt0.c* di inizializzazione dell'ambiente (si tratta di codice eseguito prima dell'invocazione della funzione *main()*) ed è sempre indispensabile porre molta attenzione all'implementazione delle funzioni di libreria perché spesso richiamano servizi di supporto - per esempio nel caso della gestione della memoria *heap* - che non sono disponibili. Inoltre, se si scrive una applicazione multitasking (ma lo stesso problema si ha anche nel caso di un programma *foreback/foreground* in cui le funzioni ISR siano scritte in C ed invochino a loro volta funzioni di libreria), occorre valutare la "rientranza" delle funzioni della libreria standard: solo in questo caso è possibile richiamarle con sicurezza in task concorrenti eseguiti "parallelamente". Infine non sempre è possibile sviluppare una applicazione completamente *stand-alone* e, di conseguenza, è necessario fornire il programma applicativo di un ambiente operativo minimo che implementi alcuni servizi di base, in particolare le funzionalità di gestione della memoria indispensabili per l'uso delle funzioni standard *malloc()* e *free()*.

Nonostante la recente invasione del campo embedded e real-time da parte di architetture Intel x86 compatibili, spesso il microprocessore target da programmare è di tipo radicalmente diverso (si passa, a seconda dell'applicazione, da microcontroller a 16 bit con pochi Kbyte di memoria RAM e ROM a processori RISC, spesso integrati da unità DSP, con potenze e velocità computazionali di tutto rispetto), ma la piattaforma di sviluppo è comunque quasi sempre un PC (DOS/Windows) o una workstation UNIX. In questo caso il "sistema di sviluppo" comprende, oltre agli strumenti software tradizionali (cross-compilatori, cross-assemblatori, linker, librerie, micro ambienti operativi, debugger, simulatori, cross-debugger, ...), una *evaluation board* che consente la "valutazione" ed il debugging in esecuzione reale (senz'altro più significativa della esecuzione simulata) del codice sviluppato prima di effettuarne il testing utilizzando un prototipo hardware del sistema definitivo. A questo scopo è sempre possibile "caricare" il programma in formato eseguibile direttamente nella RAM della scheda di valutazione (e non programmando una ROM: soluzione insensata in fase di debugging!) mediante un collegamento seriale o di tipo Ethernet che, se è gestito da un *monitor* adeguato, consente di utilizzare un cross-debugger avanzato (per esempio Microtec X-RAY) in esecuzione remota sulla piattaforma di sviluppo. Nel caso si presenti l'esigenza di un debug assolutamente "non invasivo" (soluzione costosa, ma estremamente desiderabile per sistemi che possono presentare malfunzionamenti strettamente dipendenti dal tempo di esecuzione e dalle interazioni con l'ambiente esterno) è possibile ricorrere a sofisticati *logic analyzer* che - correlando in tempo reale le informazioni ricavate dal monitoring hardware del flusso di segnali che transitano tra il processore, la memoria ed i dispositivi di I/O e di supporto con il codice in formato eseguibile e sorgente - forniscono un *tracing* assolutamente fedele dell'esecuzione in corso.

**Developing, debugging, testing & ... stressing**

Grazie alle "lezioni di *testing*" di Graziano Lo Russo tutti i lettori di Computer Programming sono consapevoli che il "debugging" di una applicazione non è il "testing": il fatto che un programma "giri" e che, almeno apparentemente, svolga i compiti per i quali è stato sviluppato non ne garantisce la consistenza con le specifiche di progetto. In apertura accennavo al destino del mio ultimo programma: consegnato al committente che, per clausola contrattuale dettata da motivazioni tecniche (la verifica finale di un *embedded system* deve necessariamente impiegare i dispositivi che il sistema stesso controlla e in questo caso - trattandosi di un sistema di navigazione automatica - essi non erano di comune disponibilità), si è incaricato di effettuare l'intera fase di testing. La fase di debugging è stata svolta sostituendo i dispositivi da controllare con simulatori realizzati utilizzando gli strumenti di un laboratorio di misure elettroniche (oscilloscopi per segnali analogici e digitali, generatori di segnali periodici e transitori): oggi molti di questi strumenti possono essere implementati in modo virtuale utilizzando un PC integrato da una scheda di I/O analogico e digitale gestita da uno specifico software applicativo. In ogni caso la separazione tra chi scrive il codice di una applicazione (e, eventualmente, effettua il debugging) e chi esegue il testing risponde a valide motivazioni, anche dal punto di vista psicologico **[2]**.

Il testing di un programma consiste nella sua esecuzione controllata al fine di individuare gli errori commessi in fase di sviluppo: nel caso di applicazioni embedded e real-time le tecniche usualmente impiegate per il testing di applicazioni convenzionali non sempre - a causa delle particolarità introdotte dalla presenza di interazioni tra i moduli che concorrono per l'accesso a risorse comuni e dalla eventuale dipendenza degli errori che si verificano in fase di esecuzione da contingenze temporali - si rivelano adeguate a garantire la necessaria sicurezza di funzionamento. **[3]** è un interessante articolo che presenta varie metodologie di testing impiegate nella validazione di un sistema operativo real-time (RTOS: Real-Time Operating System), ma che, in generale, risultano ampiamente applicabili alla verifica di una qualsiasi applicazione sottoposta a vincoli di tipo temporale in esecuzione. Gli autori propongono una prima fase di testing sia funzionale che

strutturale (*black-box* e *white-box* con realizzazione di *driver* e di *stub*) delle singole funzioni, seguita da una verifica esclusivamente funzionale dei moduli realizzati a partire dalle funzioni validate in precedenza. La verifica funzionale dell'applicazione complessiva viene in seguito effettuata integrando incrementalmente i moduli già "testati" secondo una logica *bottom-up*: dal livello più basso al più alto della gerarchia di controllo.
Per una applicazione critica sotto l'aspetto dei tempi, della continuità e della sicurezza di funzionamento il testing, sia pure condotto con metodologie adeguate, non è sufficiente: è sempre necessario verificare che l'applicazione è "robusta" anche in condizioni di *stress* operativo quali carico di lavoro sovradimensionato e tempi di risposta ridotti.

**Conclusioni**
Questa breve serie dedicata alla programmazione dei sistemi embedded e real-time proseguirà nei prossimi mesi con un primo articolo avente come argomento l'organizzazione software di tipo *foreground/foreback*, seguito da un secondo articolo relativo ai kernel real-time per la gestione di applicazioni multi-threading: in particolare verrà esaminato il sistema μC/OS il cui codice è distribuito in formato sorgente da J. Labrosse e R&D Books **[4]**. Gli strumenti e le tecniche di cui si tratterà si riferiscono ad un ambito applicativo tipico della produzione in piccoli numeri (sistemi di controllo e di visione industriali, sistemi di navigazione e di comunicazione, ...) e della relativa economia di scala in fase di progettazione, sviluppo e verifica: la produzione di sistemi embedded/real-time in grande serie (*automotive*, hi-fi, TV, ...) rappresenta infatti per alcune tecniche di ricerca e per l'impiego di sofisticati strumenti hardware un "mondo a parte" (anche in questo caso si nota la differenza con le applicazioni "standard", spesso sviluppate con gli stessi strumenti software che Microsoft impiega per produrre sistemi operativi e programmi applicativi).

**Riferimenti bibliografici**
[1] J. Labrosse, "Embedded Systems Building Blocks", R&D Books, 1995
[2] R. Pressman, "Software Engineering: a Practitioner's Approach", McGraw-Hill, 1992
[3] M. Tsoukarellas, V. Gerogiannis & K. Economides, "Systematically Testing a Real-Time Operating System", IEEE Micro, October 1995
[4] J. Labrosse, "μC/OS: the Real Time Kernel", R&D Books, 1992

# Riquadro 1 - Chips programming

Dal punto di vista del programmatore un sistema embedded è normalmente costituito da un microcontroller che interagisce con il mondo esterno per mezzo di sensori, attuatori e canali di comunicazione interfacciati mediante dispositivi hardware di vario genere quali ADC (*Analog to Digital Converter*), DAC (*Digital to Analog Converter*) e UART (*Universal Asinchronous Receiver and Trasmitter*). Dispositivi di questo tipo sono spesso realizzati come vere e proprie unità funzionali autonome che, una volta "programmate" per operare secondo una specifica modalità, interagiscono con il microcontroller solo al fine di consentire il necessario scambio di dati, direttamente o via DMA (*Direct Memory Access*). La complessità relativa alle possibili modalità operative di alcuni *chip* spesso trasforma la stesura del codice di inizializzazione in una e vera e propria attività di programmazione; molti microcontroller "vedono" i dispositivi di I/O per mezzo di registri corrispondenti ad indirizzi *memory mapped* sequenziali ai quali è possibile accedere direttamente da codice C:

```
typedef volatile struct DEVICE
{
 unsigned long int REG0;
 unsigned long int REG1;
 unsigned long int REG2;
} DEVICE_TYPE;

unsigned long int reg;
DEVICE_TYPE* device_pointer = (DEVICE_TYPE*)(0xffff0000);

/* device memory initial 32 bit address = 0xffff0000 */

device->REG0 = ...;
device->REG1 = ...;

reg = device->REG2;
```

Una volta programmato, un dispositivo hardware può interagire con il microcontroller generando interruzioni esterne che devono essere servite da una apposita ISR (*Interrupt Service Routine*) software di gestione, oppure rimanendo in attesa di essere interrogato (*polling*) su base periodica dal software di controllo (in questo caso i tempi di latenza sono spesso critici). Ogni microprocessore adotta una particolare politica di gestione delle interruzioni; una interessante panoramica comparativa che prende in esame i processori più diffusi (AMD29k, DEC Alpha, Intel 386/486, Intel Pentium, Motorola 680x0, Sun SPARC, ...) è riportata in:

- W. Walker & H. Cragon, "Interrupt Processing in Concurrent Processors", IEEE Computer, June 1995.

# Riquadro 2 - Sicurezza, correttezza e continuità di funzionamento delle applicazioni critiche

"Un sistema real-time deve rispondere a stimoli generati esternamente con un ritardo di tempo finito e specificabile a priori. I sistemi real-time sono spesso sistemi embedded che interagiscono direttamente con i dispositivi fisici che gestiscono e controllano. Esempi di questi sistemi sono i sistemi di controllo della navigazione e del traffico aereo e i sistemi di telecomunicazione. [...].
Il malfunzionamento di una applicazione critica come un sistema di telecomunicazione può causare una grave perdita economica, ma il malfunzionamento di un sistema di controllo della navigazione aerea può causare la perdita di vite umane." [W. Everett & S. Honiden, "Reliability and Safety of Real-Time Systems", IEEE Software, May 1995].
La progettazione delle componenti hardware di una applicazione *mission-critical* e/o *safety-critical* ha ormai raggiunto uno stato di relativa maturità tecnica, ma le metodologie per garantire la sicurezza, la correttezza e la continuità di funzionamento delle componenti software sono ancora ignote a molti sviluppatori. La limitatezza delle risorse disponibili per la computazione, ad iniziare dal tempo, e la possibile assenza di intervento umano in caso di malfunzionamento rendono radicalmente diversa l'analisi progettuale del software real-time per sistemi embedded rispetto ad altre tipologie di software: lo sviluppo di una applicazione standard è infatti centrato sulla gestione delle situazioni "normali", mentre lo sviluppo di una applicazione critica deve necessariamente prendere in considerazione anche la gestione delle situazioni "anormali".
Una possibile metodologia di analisi dei requisiti di sicurezza delle componenti software di un sistema per il controllo di processo fondata su tecniche formali (spesso osteggiate dagli sviluppatori privi delle necessarie conoscenze di matematica discreta e di logica matematica) è introdotta in:

- R. de Lemos, A. Saeed & T. Anderson, "Analyzing Safety Requirements for Process-Control Systems", IEEE Software, May 1995.

# Riquadro 3 - Sistemi fuzzy e reti neurali

Per quanto le più comuni tipologie di microcontroller per sistemi embedded siano tuttora convenzionali si stanno velocemente diffondendo alcune architetture innovative:

- processori RISC che integrano *on chip* funzionalità DSP (*Digital Signal Processing*);
- controller realizzati sulla base di un motore inferenziale per regole *fuzzy*;
- implementazioni hardware configurabili di reti neurali.

Se l'implementazione hardware delle funzioni DSP è ormai una tecnologia consolidata dell'era multimediale perché assolutamente indispensabile all'esecuzione in tempo reale degli algoritmi di compressione e di decompressione delle immagini video e delle informazioni sonore, i sistemi fuzzy e le reti neurali - dato anche il loro modello computazionale "approssimativo", per molti versi alternativo all'approccio classico - sono ancora considerati con sospetto dal progettista tradizionale. In realtà entrambe le tecnologie si stanno affermando con successo in alcuni campi applicativi specifici: le reti neurali sono validi strumenti se impiegate - ed è solo un esempio - nell'analisi di immagini finalizzata alla ricerca di *pattern* specifici, mentre i sistemi inferenziali fuzzy (i lettori di Computer Programming hanno avuto modo di approfondire l'argomento grazie alle "lezioni di logica e sistemi fuzzy" di Mario Motta) si stanno rivelando ottimi controllori di processo nel caso di sistemi non facilmente descrivibili per mezzo di un modello matematico lineare, sistemi ai quali non è possibile applicare la classica tecnica di controllo PID (Proporzionale, Integrativa e Derivativa). Un valido articolo introduttivo all'argomento "reti neurali" è:

- A. Jain, J. Mao & K. Mohiuddin, "Artificial Neural Networks: a Tutorial", IEEE Computer, March 1996.

*I sistemi embedded e real-time di tipo più semplice - basati spesso su architetture hardware di limitata complessità, caratteristica comune a molti microcontroller ad 8 e 16 bit - possono essere realizzati, per quanto riguarda la componente software, da un programma stand-alone eseguito senza il supporto run-time di un sistema operativo al quale devolvere i tradizionali compiti di gestione e controllo*

# Architetture software di tipo foreground/background

*Giorgio Meini*

Non volendo (o non potendo, ad esempio per motivi di *overhead* dei tempi di risposta e di computazione o per insufficienza dello spazio di memoria RAM e ROM) realizzare la componente software di un sistema embedded sulla base dei servizi resi disponibili a *run-time* da un sistema operativo real-time (RTOS: *Real Time Operating System*) è necessario scrivere un programma *stand-alone* che implementi, oltre alla "logica" richiesta per il controllo del sistema, anche alcuni dei tradizionali compiti di supporto normalmente gestiti dal sistema operativo.
Una organizzazione software frequentemente adottata in questi casi è schematizzata in **figura 1**: il programma principale è costituito da un ciclo infinito di controllo (*background*) delle operazioni non direttamente correlate agli eventi esterni, mentre le funzioni di gestione delle interruzioni (*foreground*) eseguono le operazioni esplicitamente richieste da specifici eventi esterni e la cui risposta deve rispettare precisi vincoli temporali **[1][2]**.

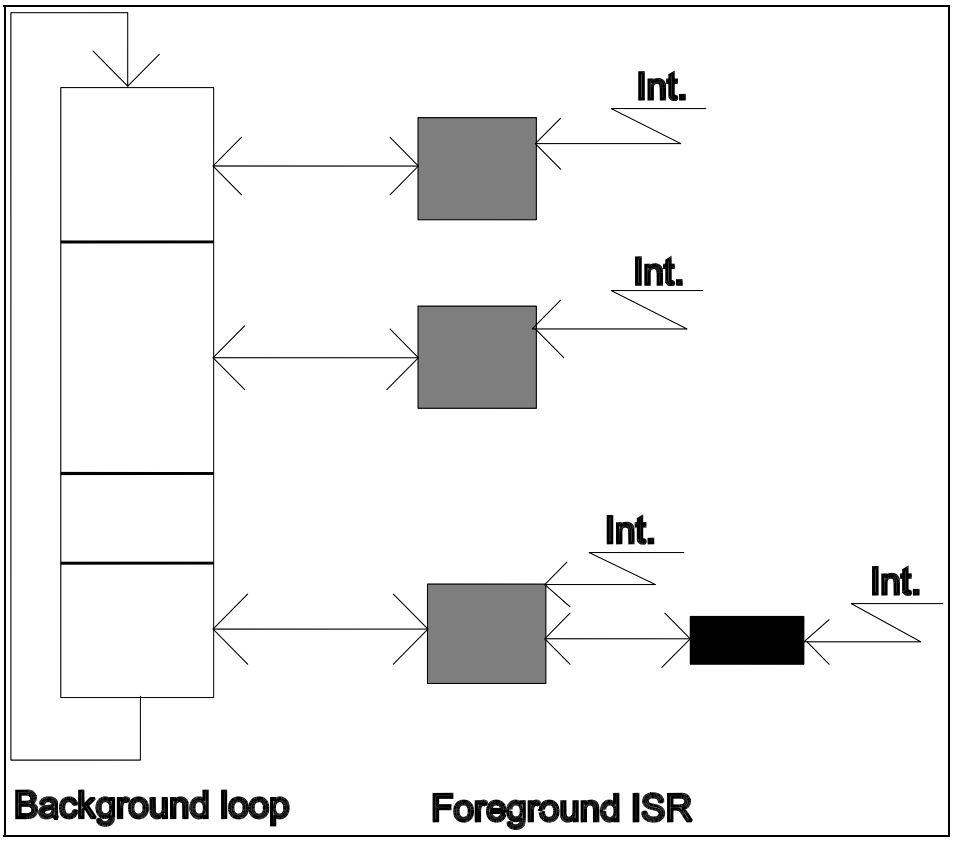


**Figura 1 - organizzazione foreground/background**

Come si vede la programmazione *event-driven* (*interrupt-driven* in questo caso) non è stata inventata per gestire le funzioni API di Windows, ma è un paradigma ampiamente utilizzato anche in altri contesti: non è raro infatti che in applicazioni di tipo embedded basate su microcontroller *low-power* il *background loop* consista in un ciclo completamente privo di operazioni (coincidente al limite con lo stato di *wait* del microprocessore) e che tutte le funzionalità del sistema siano gestite esclusivamente dalle ISR (*Interrupt Service Routine*).

**Foreground ISR e background loop**

Una architettura software di tipo foreground/background viene implementata mediante un programma "principale" che, dopo una prima fase di inizializzazione del sistema, realizza un ciclo infinito di operazioni eseguite concorrentemente all'attivazione asincrona di funzioni da parte di interruzioni che a loro volta sono generate da eventi esterni o interni al sistema stesso. Se gli eventi esterni da gestire, normalmente mediati da circuiti specializzati, possono essere relativi alle più diverse situazioni operative (ricezione o acquisizione di dati, variazioni di stato, condizioni di sincronizzazione, ...), un tipico evento interno è rappresentato dall'interruzione che viene periodicamente generata da un dispositivo *timer*. Molti microcontroller integrano un dispositivo programmabile per generare una interruzione ogni volta che un contatore a funzionamento ciclico degli impulsi del *clock* del processore si azzera: impostando il numero di impulsi da contare si ottiene un generatore periodico di interruzioni la cui frequenza (il numero di interruzioni generate per secondo) può essere stabilita dal programmatore con elevata precisione. Per esempio il timer interno del diffuso microcontroller AMD 29200 (architettura Advanced Micro Devices 29K) può essere facilmente programmato per generare una interruzione ogni millisecondo stabilendo come "costante di tempo" la frequenza di clock divisa per 1000:

```
const  gr112, 0x00004e20 ; clock 20 Mhz, timer 1ms
consth gr112, 0x00004e20 ; 0x4e20 = 20000
mtsr   TMC,   gr112
const  gr112, 0x01004e20 ; + IE bit
consth gr112, 0x01004e20
mtsr   TMR,   gr112
const  gr89,  0x00000000 ; mS counter
consth gr89,  0x00000000
```

La funzione di gestione dell'interruzione generata dal dispositivo timer deve ripristinare il corretto valore del registro contatore - a garanzia dell'esatta periodicità di ripetizione dell'evento - ed incrementare il numero di millisecondi trascorsi in modo da consentire l'aggiornamento di un eventuale clock di sistema:

```
mfsr    gr64, TMR        ; get TMR value
const   gr65, 0x10000000 ; reset IN bit in TMR
consth  gr65, 0x10000000 ; reset IN bit in TMR
andn    gr64, gr64, gr65 ; value in it0
mtsr    TMR,  gr64       ; set TMR
add     gr89, gr89, 1    ; increment mS counter
...
iret
```

All'interno della funzione di servizio delle interruzioni generate dal dispositivo timer può essere inserito il codice di aggiornamento del clock (da attivarsi per esempio ogni 100 millisecondi per un clock di sistema avente la risoluzione di un decimo di secondo), così come il codice relativo ad

operazioni che il sistema stesso deve eseguire rispettando precise condizioni temporali. Nel caso che l'esecuzione periodica di tali operazioni non sia vincolata ad una precisione assoluta è comunque preferibile "settare" una variabile *flag* che indichi al ciclo infinito eseguito in background di effettuare l'operazione stessa senza prolungare inutilmente il tempo di esecuzione della ISR del timer il cui codice viene normalmente eseguito con le interruzioni del processore disabilitate.
La condivisione di una variabile globale tra il ciclo infinito principale e le funzioni ISR è di fatto una tecnica comunemente adottata nelle architetture software foreground/background per implementare le primitive di comunicazione e sincronizzazione necessarie alla corretta esecuzione del codice concorrente. In particolare l'asimmetria in esecuzione tra le istruzioni del ciclo infinito di controllo e le istruzioni delle funzioni di gestione delle interruzioni - quest'ultime sono sempre eseguite interrompendo il normale flusso esecutivo delle prime, ma non avviene mai il viceversa - permette di regolare l'accesso ad eventuali risorse condivise (variabili globali, funzioni di libreria non rientranti, dispositivi hardware, ...) con modalità più semplici rispetto a quanto richiesto da un vero e proprio ambiente multitasking (semafori). Infatti in un contesto foreground/background, ferma restando la possibilità di disabilitare momentaneamente le interruzioni da parte del codice in background per impedire di essere interrotto in una fase "critica" della propria esecuzione da parte delle funzioni ISR, è spesso sufficiente impiegare una variabile "flag" globale per implementare un semaforo binario di regolamentazione dell'accesso ad un oggetto condiviso. Nell'ipotesi che una funzione ISR che accede ad una risorsa condivisa sia ricorsivamente interrotta solo da ISR diverse che non accedono alla medesima risorsa (ipotesi accettabile in quanto, normalmente, le funzioni di gestione delle interruzioni sono eseguite con le interruzioni disabilitate) e che rilasci in ogni caso ed in stato consistente la risorsa stessa prima di restituire il controllo alla normale esecuzione sequenziale interrotta è sufficiente che la variabile flag sia "settata" dal codice background e "testata" dal codice foreground. Non essendo in questo caso necessario che la scrittura o la lettura della variabile flag sia effettuata ricorrendo ad una particolare sequenza "atomica" di istruzioni non interrompibile è possibile scrivere il codice direttamente in C:

```
int free=1; /* risorsa condivisa libera */
void read_consistent_data_from_shared_memory();
void write_correlated_data_to_shared_memory();
struct SHARED
{
 int x;
 int y;
 int z;
} shared;

main()
{
 ...
 for(;;) /* ciclo infinito */
    {
     ...
     free = 0; /* risorsa condivisa impegnata */
     write_consistent_data_to_shared_memory();
     free = 1; /* risorsa condivisa libera */
     ...
    }
}
```

```
interrupt void int_handler()
{
 ...
 if (free)
   read_correlated_data_from_shared_memory();
 else
     {
      ...
     }
 ...
}
```

Nel caso si intenda evitare che la funzione di gestione dell'interruzione si trovi ad operare nella condizione di risorsa non disponibile è possibile sostituire le assegnazioni dei valori 0 ed 1 alla variabile flag rispettivamente con una istruzione di disabilitazione ed una di abilitazione delle interruzioni, eventualmente ricorrendo ad un inserimento in C mediante *in-line assembler*. In questo modo si posticipa la gestione di una eventuale interruzione generata contemporaneamente all'esecuzione del codice di "regione critica", ma - al tempo stesso - si contribuisce al degrado delle prestazioni complessive del sistema aumentandone il tempo di latenza.
Il tempo di latenza esprime il tempo massimo per il quale permangono disabilitate le interruzioni del processore e coincide con il tempo di massima attesa della risposta iniziale da parte di un evento esterno che genera una interruzione: ovviamente si tratta di un parametro fondamentale rispetto al quale valutare le prestazioni di un sistema real-time e la cui criticità deve essere attentamente valutata dal programmatore. Una tecnica alla quale ricorrere in caso di elevati tempi di latenza rispetto ad eventi esterni particolarmente critici sotto l'aspetto del tempo di risposta da parte del sistema consiste nel permettere ad alcuni segnali di interruzione specifici (molti processori consentono di selezionare le interruzioni attivabili in un dato contesto) l'interruzione "nidificata" (*nested*) di quelle funzioni ISR il cui codice richiede tempi di esecuzione significativi. Per limitare lo overhead imposto dalla nidificazione delle operazioni di salvataggio e ripristino del contesto del codice in esecuzione (compito trasparente - non per questo inesistente! - per i programmatori C che utilizzano compilatori con le comuni estensioni non standard quali la keyword *interrupt*, ma particolarmente gravoso su architetture hardware ad uso intensivo di registri globali come molti processori RISC) alcuni microcontroller consentono di gestire specifiche interruzioni in modalità *lightweight* utilizzando registri speciali e "congelando" momentaneamente lo stato del processo interrotto **[3]**. Generalmente è necessario codificare direttamente in assembler le routine di gestione relative ad interruzioni di questo tipo: dato che spesso il codice si limita a settare o incrementare una variabile flag questa limitazione non costituisce un problema.

**Un semplice esempio: un tachimetro software**
Disponendo di un dispositivo che genera un impulso per ogni unità di spazio percorsa non è difficile realizzare un tachimetro (misuratore di velocità): una soluzione completamente software prevede una funzione di gestione delle interruzioni corrispondenti agli impulsi il cui codice incrementa una variabile globale che, ad intervalli di tempo regolari (scanditi dalle interruzioni del timer), viene letta ed azzerata dal ciclo principale per calcolare la velocità come rapporto tra spazio percorso e tempo trascorso. Utilizzando il microcontroller AMD 29200 (architettura Advanced Micro Devices 29K) è possibile programmare un bit del dispositivo interno PIO (Parallel Input Output) al fine di generare una interruzione per ogni transizione di livello positiva del relativo segnale in ingresso:

```
#define PIO_InterruptRising(bit) \
        POCT &= ~(0x3 << (2* (bit))); \
        POCT |= (0x2 << (2* (bit))); \
        POCT &= ~(1 << (bit))

#define VECT_PIO(n) (220+8+(15-n))

unsigned int pulses = 0;

_settrap(VECT_PIO(10), ASM_PIO10int_handler);
PIO_InterruptRising(10);
```

La funzione di "sistema" *_settrap()* permette di associare una funzione di gestione della specifica interruzione al relativo "vettore": in seguito alla sua invocazione ogni interruzione generata dal segnale esternamente connesso al bit 10 del dispositivo PIO attiva la funzione *ASM_PIO10int_handler()*:

```
_ASM_PIO10int_handler:

const   gr64, ICT
consth  gr64, ICT
const   gr65, 0x00040000
consth  gr65, 0x00040000
store   0, 0, gr65, gr64 ; (re)set ICR bit 18 (IOPI PIO10)
const   gr66, pulses
consth  gr66, pulses
load    0, 0, gr67, gr66
add     gr67, gr67, 1
store   0, 0, gr67, gr66 ; pulses++;
iret
```

La funzione ISR *ASM_PIO10int_handler()* viene eseguita in modalità *freeze* **[4]** utilizzando esclusivamente i registri che le convenzioni di programmazione dell'architettura AMD 29K riservano per le routine di gestione delle interruzioni che non salvano il contesto di esecuzione del codice interrotto nello *stack* (*lightweight interrupt*); la funzione si limita ad incrementare il valore della variabile globale *pulses* e a comunicare l'avvenuta gestione dell'interruzione generata dal dispositivo interno PIO (*interrupt acknowledgement*). Nell'ipotesi che il dispositivo esterno generi una interruzione per ogni metro percorso e che la routine di gestione del timer "setti" la variabile globale *TimeFlag* ogni secondo trascorso, il ciclo infinito di controllo può periodicamente aggiornare la misura della velocità:

```
for (;;)
   {
    ...
    if (TimeFlag)
      {
       TimeFlag=0;
       DISABLE_INTS;
       Vel = pulses; /* m/s */
       pulses = 0;
       ENABLE_INTS;
      }
    ...
   }
```

Le macro DISABLE_INTS ed ENABLE_INTS nascondono l'implementazione (*in-line* assembler) delle istruzioni di disabilitazione e di abilitazione delle interruzioni del processore: pur non essendo strettamente necessarie in questo caso si tratta di una tecnica comunemente adottata per regolamentare l'accesso a risorse hardware e software condivise da parte di codice eseguito concorrentemente.
Per limitare lo *overhead* del sistema è possibile adottare un contatore hardware esterno di impulsi da leggersi ed azzerarsi periodicamente mediante operazioni di I/O digitale eseguite dal ciclo principale del programma.

**Buffer ciclici e buffer doppi**
La comunicazione di un sistema embedded con un computer esterno avviene spesso per mezzo di un collegamento seriale (cfr. **riquadro 1**): in questo caso è necessario implementare e gestire un buffer ciclico (*ring buffer*) per memorizzare temporaneamente i dati da trasmettere o i dati ricevuti da elaborare. In un programma con architettura di tipo foreground/background il buffer ciclico è normalmente una struttura dati globale condivisa dalla funzione ISR di gestione del dispositivo fisico di comunicazione (per esempio un dispositivo UART: *Universal Asynchronous Receiver and Transmitter*) e dal codice del ciclo infinito che produce i dati da trasmettere o elabora i dati ricevuti:

```
struct RING_BUFFER
{
 char          buffer[1024];
 unsigned int BufferChars;
 unsigned int WritePointer;
 unsigned int ReadPointer;
} Rx, Tx;

Rx.BufferChars = Tx.BufferChars = 0;
Rx.WritePointer = Tx.WritePointer = 0;
Rx.ReadPointer = Tx.ReadPointer = 0;
```

Il campo *BufferChars* registra il numero di caratteri presenti nel buffer ed è l'unica variabile della struttura il cui aggiornamento è critico rispetto all'esecuzione concorrente del codice background e foreground. *WritePointer* e *ReadPointer* sono infatti variabili utilizzate esclusivamente dal codice "produttore" che inserisce nuovi dati nel buffer e dal codice "consumatore" che preleva i dati dal buffer; inoltre il codice consumatore ed il codice produttore accedono "contemporaneamente", in

condizioni operative normali, ad elementi distinti del vettore *buffer[]*. Nel caso della ricezione il codice produttore è implementato dalla funzione di gestione delle interruzioni *data ready* del dispositivo di comunicazione seriale:

```
interrupt void SerialReceive()
{
 unsigned int n,i;

 n = SerialDataToRead();
 for (i=0; i<n; i++)
    {
     Rx.buffer[Rx.WritePointer] = ReadSerialData();
     Rx.BufferChars++;
     Rx.WritePointer = (++(Rx.WritePointer)) % 1024;
    }
}
```

L'operazione di modulo (*%* è l'operatore C che calcola il resto della divisione intera) realizza la "ciclicità" del buffer trasformando l'incremento lineare della variabile *WritePointer* in una sequenza di puntatori che sovrascrive i dati precedenti - già letti ed elaborati dal processo consumatore - con i nuovi dati ricevuti: è sufficiente controllare che non si verifichi una condizione di *overflow* del buffer - situazione in cui il codice produttore sovrascrive dati non ancora "consumati" - per garantire la sincronizzazione tra i due processi che concorrono per l'accesso al buffer comune condiviso. Il codice consumatore presenta una struttura sostanzialmente simile, ma organizzata in modo simmetrico:

```
for (;;)
   {
    ...
    if (Rx.BufferChars>0)
      {
       unsigned int n,i;
       char string[1024];

       n = Rx.BufferChars;
       for (i=0; i<n; i++)
          {
           string[i] = Rx.buffer[Rx.ReadPointer];
           Rx.ReadPointer = (++(Rx.ReadPointer)) % 1024;
          }
       DISABLE_INTS;
       Rx.BufferChars = Rx.BufferChars - n;
       ENABLE_INTS;
       ...
      }
    ...
   }
```

L'esecuzione del codice consumatore - che, nel caso della ricezione, costituisce una parte del ciclo principale - può essere interrotta dal codice produttore contenuto nella funzione di gestione delle interruzioni del dispositivo di comunicazione: infatti date le limitate capacità di *hardware buffering* dei dispositivi UART questa scelta minimizza il tempo di latenza e rende praticamente nulla la

possibilità di "perdere" dati in ricezione. Se non si verifica una condizione di *overflow* la ISR scrive i nuovi dati ricevuti in elementi del vettore *buffer[]* diversi rispetto agli elementi cui accede in lettura il codice background ed incrementa la variabile *BufferChars*: l'operazione di aggiornamento di questa variabile da parte del codice inserito nel ciclo infinito deve quindi essere resa "atomica" (una operazione di aggiornamento normalmente prevede due accessi in memoria: il primo per leggere il valore attuale della variabile ed il secondo per memorizzare il nuovo valore) per impedire che una sua eventuale interruzione invalidi il valore stesso della variabile rendendolo non consistente con lo stato del buffer.
Una diversa tecnica di gestione si rivela utile in sistemi di acquisizione ed elaborazione in tempo reale di segnali analogici o di immagini. Il ricorso ad un "doppio" buffer (*double buffer*) permette al codice background di elaborare i dati acquisiti in precedenza mentre contemporaneamente il codice foreground memorizza in un buffer distinto nuove informazioni mediante ripetute attivazioni della funzione ISR: se il tempo di elaborazione non supera il periodo di acquisizione è sufficiente, al termine dell'operazione, "scambiare" il buffer su cui operano il ciclo infinito e la funzione di gestione delle interruzioni per assicurare un funzionamento continuativo ed in tempo reale del sistema (lo "scambio" può essere facilmente implementato scrivendo codice che accede al buffer per mezzo di puntatori anziché direttamente mediante il nome della variabile *array*).

**Conclusioni**

La tecnica di organizzazione software foreground/background si rivela senz'altro utile nella realizzazione di semplici sistemi embedded, ma la limitata possibilità di progettazione modulare del codice implicata dall'esistenza di un unico ciclo di controllo del funzionamento complessivo può essere superata solo adottando un sistema operativo multi-tasking e real-time: questo argomento rappresenta il tema del prossimo articolo che concluderà questa breve serie.

**Riferimenti bibliografici**

[1] J. Labrosse, "Embedded Systems Building Blocks", R&D Books, 1995
[2] G.Meini, "Introduzione alla programmazione dei sistemi embedded e real-time", Computer Programming n. ?, 1997
[3] W. Walker & H. Cragon, "Interrupt Processing in Concurrent Processors", IEEE Computer, June 1995
[4] D. Mann, "Evaluating and Programming the 29K RISC Family", 2nd ed., Advanced Micro Devices, 1995

# Riquadro 1 - Comunicazioni e protocolli

Il fatto che un sistema embedded sia normalmente privo delle tradizionali interfacce uomo-macchina (video, tastiera e mouse) non significa che l'interazione con l'ambiente esterno sia limitata ai dati acquisiti dai sensori ed agli effetti causati dall'azione degli attuatori: spesso infatti un sistema embedded comunica con altri computer per mezzo di canali digitali per la comunicazione di dati quali collegamenti seriali e reti locali.
La comunicazione seriale è quasi sempre di tipo asincrono e conforme allo standard RS-232 (comunicazione *point-to-point* tra due computer) o allo standard RS-485 (comunicazione *multi-drop*: fino a 32 computer diversi connessi ad un unica linea comune). Se per lo standard hardware RS-232 il protocollo software di comunicazione è sempre molto semplice (come è per esempio nel caso del protocollo XON/XOFF), lo standard RS-485 implica spesso, se non sempre, l'impiego di un protocollo che realizzi una organizzazione MASTER/SLAVE delle "transazioni" che avvengono sulla linea.
È necessario implementare un protocollo software/hardware di tipo MASTER/SLAVE anche impiegando il bus parallelo (ad 8 bit) di interfaccia IEEE488.
Negli ultimi anni è divenuto comune anche nel campo dei sistemi embedded e real-time l'impiego della rete locale di tipo Ethernet: in questo caso il protocollo utilizzato è UDP/IP, più veloce e maggiormente flessibile rispetto al protocollo TCP/IP normalmente impiegato per stabilire connessioni Internet.

# Riquadro 2 - Numeri *fixed-point* e matematica intera

Molti microcontroller non includono una unità *floating-point* ed i compilatori C implementano le operazioni aritmetiche che coinvolgono numeri non interi inserendo nel programma in formato eseguibile codice di emulazione. In fase di esecuzione la diversità tra implementazione hardware e software delle operazioni floating-point si riflette in una differenza di un ordine di grandezza dei tempi di computazione che spesso non è compatibile con i requisiti real-time del sistema.
Una soluzione praticabile consiste nell'impiegare la tradizionale matematica intera del processore adottando una rappresentazione *fixed-point* delle grandezze su cui operare: conoscendo a priori il *range* di variazione delle singole variabili è possibile "scalare" la rappresentazione binaria dei valori (moltiplicandoli per una opportuna potenza di 2) di un numero di bit sufficiente a garantire la precisione desiderata. Per esempio il valore 1.2345 può essere rappresentato su 16 bit come $40452_{10}$ = 0x9e94 (mantissa) scalato di 15 bit, infatti 40452 x $2^{-15}$ = 1.2344970...
Si noti che il numero di bit scalati (esponente) non è memorizzato nella rappresentazione e deve essere implementato in qualche forma nel codice che opera sul valore specifico: in particolare due numeri possono essere sommati solo se hanno lo stesso esponente e nel caso della moltiplicazione l'esponente del risultato è dato dalla somma degli esponenti dei fattori.
L'acquisizione di grandezze in forma analogica rappresenta un classico esempio di impiego della tecnica fixed-point, infatti il valore espresso nella specifica unità di misura si ottiene a partire dal valore fornito dal convertitore analogico-digitale applicando la seguente formula **[*]**:

$$U = \frac{\frac{ADC \cdot FSV}{2^n - 1} - V_{bias}}{T \cdot A_{gain}}$$

dove

U: grandezza fisica espressa in unità di misura specifiche;
ADC: valore fornito dal convertitore analogico-digitale (ADC);
FSV: tensione in Volt corrispondente al massimo valore fornito in uscita dal dispositivo ADC (fondo scala);
n: risoluzione in bit del dispositivo ADC;
T: sensibilità del trasduttore (sensore) espressa in Volt per unità di misura;
$V_{bias}$: *offset* applicato al valore di tensione prodotto dal trasduttore dopo lo stadio di amplificazione allo scopo di evitare input negativi al dispositivo ADC;
$A_{gain}$: guadagno dell'amplificatore operazionale di condizionamento del segnale elettrico prodotto dal trasduttore.

La formula precedente può riscritta come:

$$U = (ADC - Bias) \cdot Gain$$

dove

$$Bias = \frac{V_{bias} \cdot (2^n - 1)}{FSV}$$

e

$$\text{Gain} = \frac{\text{FSV}}{\text{T} \cdot \text{A}_{\text{gain}} \cdot (2^{\text{n}} - 1)}$$

Labrosse **[*]** applica queste formule alla misura espressa in gradi Fahrenheit di una temperatura compresa tra -50° e +300° effettuata per mezzo di un trasduttore di tipo LM34A avente una sensibilità di 0.1 V/°F e condizionato da un amplificatore operazionale con guadagno pari a 2.5 ed offset di tensione uguale a +1.25 V. Date queste condizioni il dispositivo ADC con 10 bit di risoluzione e tensione di fondo scala pari a 10 V riceve in ingresso una tensione compresa tra 0 e 8.75 V e si ha:

$$\text{Bias} = \frac{1.25 \cdot 1023}{10} = 127.875$$

$$\text{Gain} = \frac{10}{0.1 \cdot 2.5 \cdot 1023} = 0.391007$$

Il valore della costante *Bias* può essere scalato di 5 bit moltiplicando per $2^5$=32 ed ottenendo $4092_{10}$ (naturalmente il valore fornito dal dispositivo ADC dovrà essere scalato di 5 bit in modo analogo prima di effettuare la somma algebrica *ADC - Bias*). Il valore della costante *Gain* può invece essere rappresentato scalato di 16 bit moltiplicando per $2^{16}$=65536 in modo da ottenere $25625_{10}$: in questo modo il risultato della moltiplicazione *(ADC - Bias) x Gain* risulterà scalato di 16 + 5 = 21 bit. Per ottenere un valore di temperatura convenientemente scalato di 6 bit si applica il seguente codice C:

```
int Temp(int AdcVal)
{
 int counts,temp;

 counts = (AdcVal << 5) - 4092;
 temp = (int)(((long int)counts * (long int)25625) >> 15L);
 return(temp); /* 6 bit scaled result */
}
```

[*] J. Labrosse, “Embedded Systems Building Blocks”, R&D Books, 1995