UNIVERSITÀ DI PADOVA FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA TRIENNALE

SCHEMATIZZAZIONE DI UNO SCHEDULER STATICO PER APPLICAZIONI EMBEDDED REAL-TIME IN AMBIENTE WINDOWS CE

Relatore: **Ch.mo Prof.Michele Moro**

Laureando: **Andrea Franceschini**

Anno Accademico 2006-2007

Se il drago rifiuta di battersi, forse è solo pigro.
Ma se ignora la zanzara, è davvero addormentato.

(A. Sisti)

Sommario

Viene qui discusso il problema di trovare una schedulazione in tempo reale ammissibile per un insieme di task periodici con precise scadenze temporali di cui è noto il tempo di esecuzione; si vuole inoltre implementare tale schedulazione su una piattaforma con scheduler nativo a priorità multilivello. La ricerca di una schedulazione statica dei task periodici si basa sul calcolo del flusso massimo per il grafo del sistema (v. Liu [5]). Si analizzano anche l'eventualità della presenza di task aperiodici e sporadici, elencando le problematiche ad essi legate e formulando ipotesi opportune per permetterne l'esecuzione nel sistema reale assieme ai task periodici che costituiscono il fulcro critico del lavoro.

Assieme a questo lavoro viene fornita una implementazione del problema di validità generale per la realizzazione di un sistema con requisiti pari o simili a quelli esposti. L'implementazione proposta viene divisa in due parti: la parte relativa alla costruzione della schedulazione ciclica statica è progettata in modo tale da essere facilmente portabile e non si richiede che possa essere eseguita sulla piattaforma embedded; la seconda parte viene implementata in linguaggio C++ nell'ambiente embedded Windows CE .NET 4.2 in esecuzione sulla scheda UNI-PD-PXA basata su Xscale PXA-255 e viene adeguatamente documentata per facilitarne la portabilità su altre piattaforme.

Indice

Elenco delle figure

Introduzione

Il problema dei sistemi in tempo reale è esteso a molte applicazioni, dal controllo di processi industriali critici come la gestione del nocciolo di una centrale nucleare, a situazioni meno pericolose ma altrettanto critiche come l'acquisizione e l'elaborazione di segnali. Si tratta di acquisire, elaborare e fornire dati entro intervalli temporali prestabiliti, superati i quali i dati potrebbero non essere più utili. In molti casi si trovano coinvolti in queste attività dei sistemi hardware dedicati ad un solo scopo ma non è infrequente che la scelta ricada su sistemi *general purpose*. È proprio nel caso di questi ultimi che è più difficile garantire prestazioni di tempo reale nei confronti dei processi da realizzare, poiché soggetti a variabili fuori dal controllo di chi realizza il software. I sistemi in tempo reale si distinguono sulla base del grado di tolleranza che offrono:

***soft real-time*:** la pericolosità della mancanza di una *deadline* aumenta con l'aumentare del ritardo del termine della computazione sulla *deadline*;

***hard real-time*:** la tolleranza alle *deadline* è nulla, se un dato viene fornito oltre il tempo massimo stabilito, esso diventa inutile e la sua mancanza può portare a conseguenze anche catastrofiche, a seconda della criticità dell'applicazione.

Per comprendere il significato di questo lavoro è bene prima di tutto capire di quali tipi di *task* ci si deve occupare. Possiamo suddividerli in tre tipologie: periodici, sporadici e aperiodici. Sul primo tipo si concentrerà la prima parte del lavoro, risolvendo il problema con il metodo del

flusso massimo di un grafo opportunamentre costruito per rappresentare il sistema di processi periodici (v. Liu [5]). Il secondo e terzo tipo richiedono un'attenzione particolare poiché la loro gestione non può avvenire a priori: per le loro caratteristiche intrinseche, infatti, non sappiamo niente su di loro fino al momento del rilascio nel sistema. Se per i processi aperiodici possiamo formulare alcune ipotesi in modo tale da renderli sostanzialmente inoffensivi, per i processi sporadici ci si dovrà concentrare sull'opportunità o meno di concederegli l'ingresso nel sistema, giacché essi necessitano di terminare i loro calcoli entro un tempo prestabilito e noto solo nell'istante di rilascio. Su questo principalmente si concentrerà la seconda parte.

Il primo capitolo tratta degli aspetti teorici della ricerca e costruzione di una schedulazione ammissibile, analizza i requisiti che l'insieme dei *task* deve rispettare ed espone il processo di soluzione del problema. Il secondo capitolo è centrato sull'implementazione delle varie parti del lavoro e fornisce dettagli riguardo l'effettiva realizzazione delle varie parti del sistema, presentando i particolari implementativi e, in particolare nello schema di simulazione in Windows CE, esponendo dettagli riguardo i costrutti di sincronizzazione usati.

Capitolo 1

Schedulabilità

1.1. Requisiti del sistema

Per capire come organizzare i processi critici nel sistema, dobbiamo prima capire che tipo di comportamento hanno i processi e che azioni dobbiamo intraprendere per garantire che tutte le richieste vengano soddisfatte. I requisiti dei processi si possono riassumere in questi tre tipi di comportamento:

aperiodico: il tipo più semplice, nel quale il *task* viene semplicemente accodato e gli viene assegnata la CPU quando nessun altro processo la richiede (*idle*). I *task* di questo tipo non hanno nessuna richiesta di terminare entro un determinato istante né particolari priorità, sono semplicemente presenti nel sistema e vengono fatti avanzare quando è possibile e subiscono *preemption* in favore dei processi critici;

sporadico: il tipo più critico poiché arrivano inaspettatamente come i *task* aperiodici e inoltre devono rispettare una *hard deadline*. Per la loro analogia con i *task* aperiodici e per il requisito addizionale di rispettare una *deadline*, possiamo assumere che siano più prioritari dei *task* aperiodici ma non sono soggetti a periodicità, vengono rilasciati nel sistema e esistono finché termina la loro computazione. Per garantire che un *task* sporadico che viene rilasciato possa effettivamente terminare prima della propria deadline, deve essere sottoposto ad un *test* di ammissibilità e, similmente ai *task* aperiodici, devono subire *preemption* quando necessario;

periodico: i *task* vanno eseguiti periodicamente e hanno delle *hard deadline* da rispettare. Possono essere schedulati *off-line* in maniera statica e la tabella di marcia può essere inserita nel dispositivo *embedded* dove girano i *task* di controllo. Per loro natura sono più prioritari sia dei *task* aperiodici che di quelli sporadici in quanto la loro computazione deve avvenire strettamente tra gli istanti prestabiliti.

Vediamo nel dettaglio come si gestisce la schedulazione di queste tre classi di *task*. L'approccio che si è scelto per risolvere il problema è di tipo *clock driven*, ossia le decisioni dello *scheduler* vengono prese ad istanti prefissati. Questo approccio progettuale identifica per sè tre strategie di gestione dei diversi comportamenti dei processi: la costruzione *off-line* di una tabella statica per i processi periodici, un *test* di ammissibilità *on-line* basato in parte su dati calcolati *off-line* per i processi sporadici e infine l'assegnazione della CPU nei momenti in cui non è richiesta da altri processi ai processi aperiodici.

1.2. Task periodici

Dobbiamo prima di tutto capire in che condizioni un sistema di processi periodici è schedulabile. Ammettiamo quindi di avere un insieme di n processi indipendenti e chiamiamo, con $0 \leq i \leq n$, p_i il periodo del processo i, e_i il suo tempo di esecuzione e d_i la sua *deadline*. Condizione necessaria perché il sistema sia in queste ipotesi schedulabile è:

$$\sum_{i=0}^{n} \frac{e_i}{p_i} \leq 1. \tag{1.1}$$

Ora, osserviamo che i periodi possono essere tra loro diversi e non necessariamente multipli l'uno dell'altro. Questo ci porta a pensare di dover costruire una tabella di lunghezza infinita, il che sembra impraticabile. Invece è possibile costruire una schedulazione ciclica della durata di un intervallo di tempo H chiamato iperperiodo calcolato come il minimo comune multiplo dei periodi di tutti i processi periodici. Dentro all'iperperiodo devono starci i processi periodici ed è immediato vedere che dovranno starci

da almeno una a più di una volta, a seconda del loro periodo. L'iperperiodo viene quindi suddiviso in *frame*, unità temporali che rappresentano la tabella di marcia del *dispatcher* e le unità di decisione su cui opera lo *scheduler on-line*. Questi *frame* debitamente calcolati garantiranno anche che tutti i processi possano venire eseguiti con cadenza periodica secondo i loro parametri.

Il passo successivo è quindi determinare l'ampiezza dei *frame*. La scelta della dimensione è vincolata da alcuni fattori che possono, in alcuni casi, rendere impossibile la scelta di una dimensione praticabile. Vediamo quali sono questi vincoli e cerchiamo di capire che benefici portano e perché potrebbero impedire di trovare una schedulazione.

1. $f \geq \max(e_i) \qquad \forall\, 1 \leq i \leq n$
   impedisce che venga effettuata la *preemption* dei *job*, ovvero ogni *job* poter essere interamente eseguito in un *frame*. Questo viene imposto per semplificare al massimo la gestione del sistema sul dispositivo *embedded*, però potrebbe condurre a non trovare del tutto una schedulazione possibile, anche quando una schedulazione *preemptive* esiste;

2. $[p_i]_f = 0 \qquad$ per almeno un i
   per mantenere il ciclo di schedulazione breve, f deve dividere l'iperperiodo e questo è vero se f divide almeno un p_i. Questo viene imposto per ridurre l'occupazione di memoria delle strutture di supporto ai processi di controllo nel dispositivo *embedded*. Di contro, potendo scegliere un ciclo di lunghezza diversa – tendenzialmente maggiore – potrebbe essere possibile trovare *frame* più grandi per rispettare il primo vincolo;

3. $2f - \gcd(p_i, f) \leq D_i \qquad \forall\, 1 \leq i \leq n$
   serve ad avere un intero frame tra l'istante di rilascio di un *job* e la relativa *deadline*, così che quel *job* possa essere inserito in quel *frame* e serve a garantire che, alla fine del *frame*, si possa controllare se il *job* sia stato effettivamente eseguito del tutto.

Nel caso in cui non si riesca a trovare almeno un valore che rispetti i vincoli, è possibile ripetere la ricerca rilassando il primo vincolo e consentendo quindi la *preemption* dei *job*. Questo rende in qualche modo più difficile l'effettiva gestione del sistema ma permette di trovare più facilmente schedulazioni ammissibili per molti più insiemi di processi. In questo lavoro si è scelto di cercare comunque una schedulazione *non preemptive* per mantenere chiaro il procedimento nelle sue linee generali; l'apertura ad una schedulazione *preemptive* è una questione implementativa che complica il sistema sia nel senso pratico di gestione e controllo sia in termini di maggiore intervento dei processi di controllo e che risulterebbe in un aumento dell'*overhead*.

A questo punto è possibile calcolare le informazioni necessarie alla costruzione della tabella. Il metodo seguito è quello di costruire un grafo diretto con capacità, in cui si associano i *job* e i *frame* nel modo introdotto da Liu [5] e descritto al §2.1.1. Del grafo costruito come visto si calcola il flusso massimo: se è diverso da 0 e se non è minore della somma del tempo di esecuzione richiesto dall'insieme dei *job* nell'iperperiodo, si costruisce la tabella con la seguente regola: il tempo concesso ad un dato *job* in un dato *frame* è pari al flusso sul ramo che li collega[1].

1.2.1 Esempio

Consideriamo il seguente insieme di processi:

	period	work time	deadline
T0	4	1	4
T1	5	2	7

L'iperperiodo H vale 20, passiamo a cercare quindi gli f adatti. Applicando il primo vincolo otteniamo che dovrà risultare $f \geq 2$ e applicando il secondo calcoliamo che $f \in \{2, 4, 5\}$. Di queste tre possibilità trovate,

[1] Per i dettagli implementativi si veda pagina 13

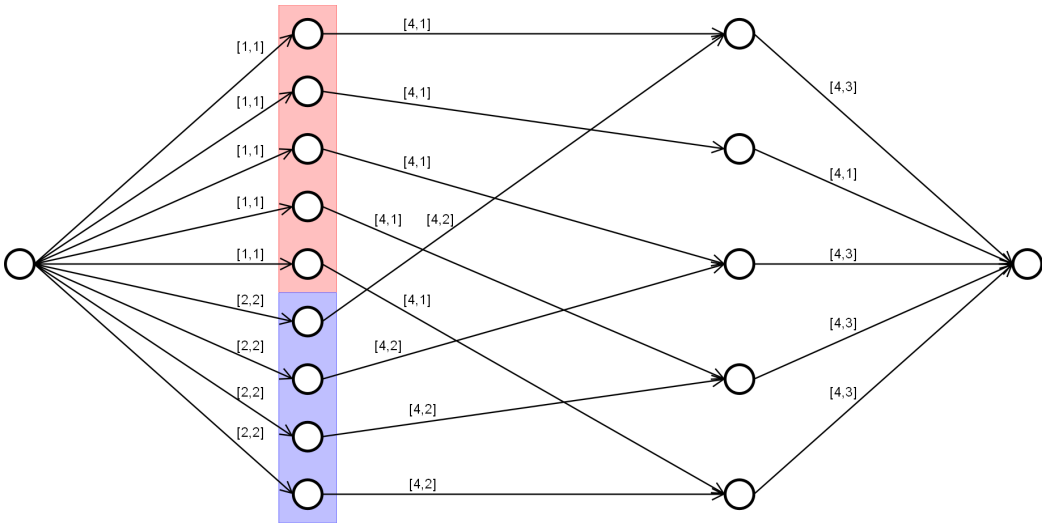

Figura 1.1: Grafo relativo all'esempio

vediamo quali possiamo tenere applicando anche il terzo vincolo:

$$
\begin{array}{lll}
f = 2 & 4 - 2 \leq 4 & OK \\
 & 4 - 1 \leq 7 & OK \\
f = 4 & 8 - 4 \leq 4 & OK \\
 & 8 - 1 \leq 7 & OK \\
f = 5 & 10 - 1 \leq 4 & NO
\end{array}
$$

Le scelte che ci rimangono infine sono 2 e 4, quindi possiamo cominciare a cercare una schedulazione partendo dalla dimensione maggiore disponibile, cioè 4, scendendo verso le inferiori se non siamo in grado di trovare una schedulazione ammissibile. Per visualizzare meglio l'esempio, nella figura 1.1 è rappresentato il grafo su cui si è calcolato il flusso massimo – le etichette sugli archi hanno la forma [capacità, flusso] – mentre nella figura 1.2 è rappresentata la schedulazione trovata con $f = 4$.

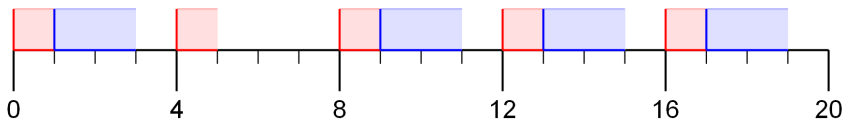

Figura 1.2: Timeline relativa all'esempio

1.3. Task sporadici

I *task* sporadici presentano una difficoltà di gestione rappresentata dal fatto che di essi sono noti a priori solo la *deadline* – assoluta rispetto alla *timeline* del sistema in esecuzione – e il tempo di lavoro, ma non è dato sapere quando saranno rilasciati né quanto sarà il loro tempo di esecuzione effettivo, considerando che devono cedere il passo ai *task* periodici quando richiesto. Per questo deve essere eseguito al volo un *test* di ammissibilità da parte di un apposito *thread* del *dispatcher* che decide se un *task* sporadico può essere inserito nel sistema oppure no.

1.3.1 Test di ammissibilità

Un algoritmo efficiente per determinare l'ammissibilità dei *job* sporadici nel sistema è il EDF[2] ciclico, che esegue le verifiche all'inizio di ogni *frame* e mantiene una lista dei *job* sporadici ordinata per *deadline* non decrescenti.

Preoccupiamoci dunque di determinare se un *task* sporadico è accettabile o no disponendo un *test* in due fasi che esamina il tempo di CPU disponibile per decidere se schedulare il processo $S(d, e)$:

1. si calcola la funzione $\sigma(t, l)$[3] e se risulta che $\sigma(t, l) < e$ il *task* è rifiutato;

2. se invece risulta che $\sigma(t, l) \geq e$ e l'accettazione di S non impedisce ad altri *task* sporadici di terminare in tempo, S viene accettato, altrimenti viene rifiutato.

[2] *Earliest Deadline First*

[3] Si chiama *slack time* e si definisce come il tempo di CPU disponibile tra il *frame* t ed il *frame* l con l l'ultimo frame precedente a d.

Per effettuare questi calcoli in modo efficiente è necessario avere alcune informazioni, di cui alcune possono essere precalcolate poiché dipendono dalle condizioni iniziali del sistema, altre devono essere determinate a *run-time*, essendo dipendenti dal sistema e dai *task* sporadici già accettati. Mantenendo opportune strutture nel sistema, possiamo ridurre il carico di CPU impiegato per portare a termine il *test* e quindi diminuire la latenza del *test* di ammissibilità. Vediamo quindi quali dati sono necessari alle due parti del *test* di accettazione.

Primo passo: in questo passo, il *thread* di schedulazione *on-line* deve sapere quanto tempo lasciano in ogni *frame* di un iperperiodo i *task* periodici, da ogni *frame i* ad ogni altro *frame h* con $i \leq h$. Questi valori possono essere calcolati al momento della costruzione della tabella statica e memorizzati in una ulteriore tabella di $(F+1)\frac{F}{2}$ elementi con F il numero dei *frame* in un iperperiodo. Da questa tabella è possibile calcolare il tempo residuo dal *frame i* di qualunque ciclo j al *frame h* di qualunque altro ciclo $j' > j$ con la seguente formula:

$$\sigma((j-1)F+i,(j'-1)F+h) = \sigma(i,F)+\sigma(1,h)+(j'-j-1)\sigma(1,F) \quad (1.2)$$

Secondo passo: in questo passo c'è bisogno di calcolare, per ogni *job* sporadico S_k già accettato, la parte del tempo di esecuzione (ξ_k) che è già stata completata all'inizio del *frame t* e la quantità di tempo disponibile per S dal *frame t* al *frame l* prima della sua *deadline*.

$$\sigma_c(t,l) = \sigma(t,l) - \sum_{d_k \leq d} (e_k - \xi_k) \quad (1.3)$$

dove $\sigma(t,l)$ è il tempo lasciato inizialmente dai *job* periodici. Se $\sigma_c(t,l) \geq e_i$, allora $S(d,e)$ è compatibile con gli altri *job* S_k con $d_k \leq d$ che sono già stati accettati. Se dunque $S(d,e)$ viene accettato, il tempo σ che lascia disponibile inizialmente prima della propria deadline è $\sigma = \sigma_c(t,l) - e$ (con $\sigma \geq 0$). Questo valore sarà utile in seguito.

Nel secondo passo c'è anche bisogno di valutare se l'ammissione di un nuovo *job* sporadico risultato compatibile ritardi l'esecuzione di altri

job sporadici S_l (con $d_l > d$) tanto da impedire loro di terminare entro le relative *deadline*. Se il nuovo *job* viene accettato, il tempo libero lasciato dal generico altro *job* S_l si riduce del tempo di esecuzione del nuovo arrivato. Dunque si può accettare il nuovo *job* solo se il tempo che lascerebbe libero per ogni altro *job* già accettato non risulta minore di zero.

Riassumento le idee, il secondo passo del *test* di ammissione richiede e calcola le seguenti informazioni:

- la tabella iniziale del tempo libero;
- il tempo di esecuzione parziale ξ_k completato all'inizio del *frame* corrente da ogni *job* S_k già accettato;
- il tempo libero attuale che ogni *job* sporadico lascia nel sistema.

1.3.2 Ottimalità dell'algoritmo EDF ciclico

Se confrontiamo questo algoritmo con l'intera classe di algoritmi che effettuano le verifiche agli istanti iniziali dei *frame*, sicuramente risulta l'algoritmo migliore finché i *job* sporadici sono schedulabili. Di contro, EDF non è ottimo nei confronti degli algoritmi che verificano l'ammissibilità all'istante di rilascio. Questo comportamento però rischia di introdurre ritardi – anche considerevoli – del tutto inaspettati che possono compromettere la solidità del sistema nei confronti delle *deadline*.

1.4. Task aperiodici

I *task* aperiodici sono il tipo di processo più semplice da gestire poiché non hanno periodo, *deadline* né altri requisiti di garanzia: semplicemente guadagnano il diritto all'uso della CPU solo quando non ci sono altri *task* – periodici o sporadici – pronti ad essere eseguiti e devono subire *preemption* ogni volta che questa è necessaria. La schedulazione dei *task* aperiodici può avvenire in tutte le classiche maniere già note in letteratura, dal più semplice FCFS di tipo *run–to–completion*, a sistemi più complessi basati su

livelli di priorità, requisiti di interattività, preferenza nell'uso dell'I/O o della CPU e così via.

In un contesto di sistema operativo *general purpose* con *scheduler* nativo a priorità com'è Windows CE possiamo assumere che ogni *task* del sistema che non fa parte dell'insieme dei *task* critici sia un *task* aperiodico, assunto che delega interamente allo scheduler nativo di Windows CE la gestione degli stessi, quando possibile.

Capitolo 2

Implementazione

2.1. Configurator

Configurator è lo *scheduler* statico, si occupa di prendere le informazioni sui *task* del sistema e tradurle nella tabella che Executor dovrà implementare.

L'interfaccia utente di Configurator, rappresentata in figura 2.1, è composta da una finestra di dialogo in cui si vanno inserite le informazioni relative ai *task* tramite i controlli nella parte superiore. Inserendo i vari processi, verranno aggiunte righe alla tabella immediatamente sotto l'area di immissione dati e l'indicatore di livello nella parte inferiore della finestra verrà riempito a seconda del carico della CPU calcolato di volta in volta. Ad inserimento completato, premendo il pulsante Ok nella zona inferiore destra della finestra di dialogo si avvia il processo che porterà, mediante una serie di iterazioni, alla costruzione della tabella statica e alla produzione del file di configurazione per Executor.

2.1.1 Ricerca di una schedulazione

Il procedimento inizia calcolando le informazioni inferibili dai dati inseriti, quali l'iperperiodo e le dimensioni dei *frame* che soddisfano le tre condizioni esposte a pagina 3. Ottenute queste informazioni è possibile procedere alla costruzione del grafo del sistema di processi come spiegato di seguito nel §2.1.1. Una volta risolto il problema di flusso massimo sul grafo e una volta verificato che la soluzione trovata consenta la costruzione di una schedulazione ammissibile, si può procedere alla costruzione della tabella

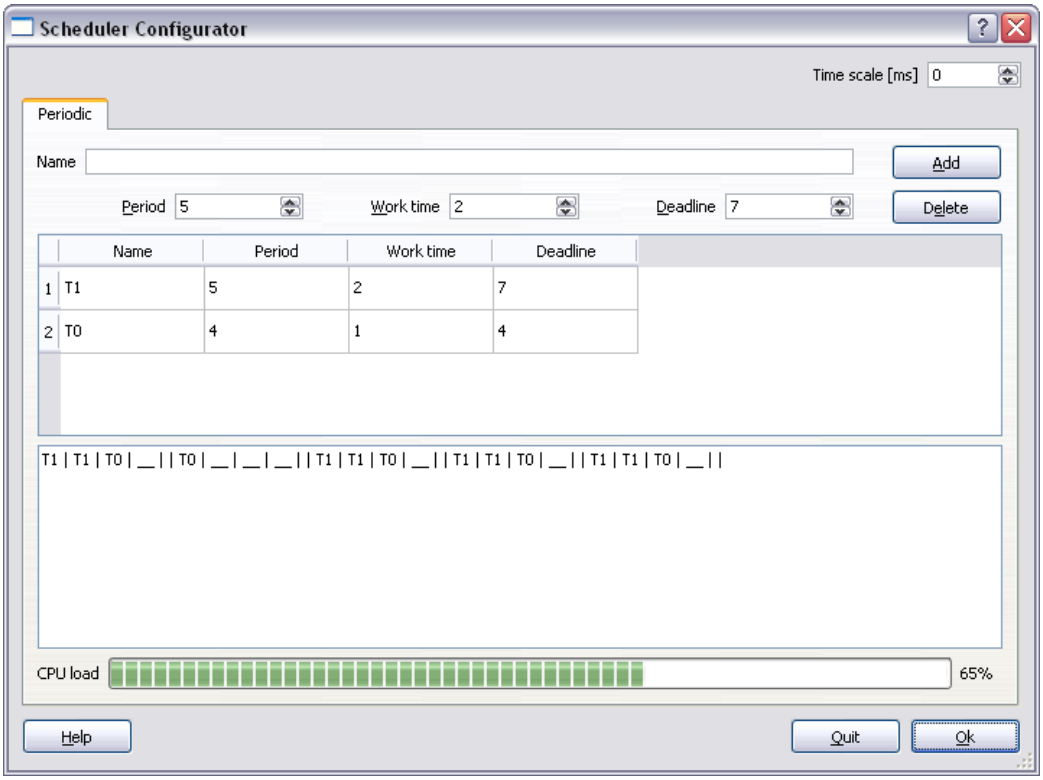

Figura 2.1: Finestra di dialogo principale dell'applicazione Configurator eseguita in Windows XP.

finale, al calcolo del tempo libero nei frame e alla produzione del *file* di configurazione per Executor.

Costruzione del grafo

Il grafo si compone di un certo numero di nodi che rappresentano i *job* e i *frame* più un nodo sorgente e uno destinazione. La costruzione comincia rappresentando i nodi che corrispondono agli *n frame* e connettendoli al nodo destinazione con archi di capacità pari all'ampiezza dei *frame*. Di seguito si rappresentano i nodi che corrispondono ai *job*; visto che l'insieme dei *frame* forma l'iperperiodo, un *job* può stare in più di un frame, a seconda del suo periodo. Si mantiene un vettore di interi che rappresenta il tempo disponibile in ogni *frame*: questo serve per aiutare nei passi successivi per la ricerca di una schedulazione *non preemptive*. Inoltre viene fatto un ordinamento dei *job* decrescente sul tempo di esecuzione. Per ogni *job* i si calcola quindi il rapporto $k = \frac{H}{p_i}$ – ovvero quante volte un dato *job* compare in un ciclo completo – che conduce a tre casi:

1. $k = n$
   se in tutti i *frame* è rimasto tempo disponibile maggiore o uguale al tempo di esecuzione del *job* che si sta gestendo, viene aggiunto un arco che collega le coppie corrispondenti di nodo *job* e nodo *frame*, altrimenti la costruzione termina;

2. $k < n \ \wedge \ e_i < f$
   viene prima fatto un controllo sul tempo rimanente nel *frame* e, se questo è sufficiente, si verifica che il *job* possa terminare entro la propria *deadline*. Se questo è il caso, l'arco viene aggiunto; se non è il caso, oppure non è stato possibile aggiungere tutti i k archi, la costruzione termina;

3. $k < n \ \wedge \ e_i = f$
   i nodi *frame* vengono divisi in k gruppi e i nodi *job* vengono collegati ai corrispondenti nodi *frame* se nei *frame* c'è tempo sufficiente e se è possibile che ogni ripetizione del *job* termini entro la propria

deadline. Se non è possibile, non c'è tempo a sufficienza o non è possibile inserire almeno k archi – i.e. uno per gruppo – la costruzione termina;

gli archi che collegano i *job* ai *frame* hanno capacità massima pari alla dimensione dei *frame* e il procedimento termina collegando la sorgente ai nodi *job* con archi di capacità pari al tempo di esecuzione del *job* a cui vengono associati. Si riveda il grafo di figura 1.1 per una rappresentazione pratica.

Se a questo punto è stato trovato un grafo, si può risolvere il problema di *max flow* associato e verificare l'ammissibilità della soluzione: essa è valida se il flusso che arriva alla destinazione è pari al massimo flusso che parte dalla sorgente[1]. Come ulteriore verifica si può controllare che il flusso trasportato da un nodo *job* ad un nodo *frame* sia pari al tempo di esecuzione del *job*, cosa che è verificata per costruzione del grafo con l'unica cautela che, per i *job* che rientrano nel terzo caso, se più di un arco per gruppo di *frame* viene riempito correttamente, è sufficiente assegnare il job ad uno solo dei *frame* associati.

Costruzione della tabella statica

Ottenute tutte le informazioni necessarie, Configurator è finalmente in grado di costruire il *file* di configurazione per Executor. Si tratta di un testo ASCII formattato in modo rigido tale da rendere facile e veloce la raccolta automatica di informazioni da esso. Vediamo di seguito un esempio completo in cui i due *task* T0 e T1 sono schedulati[2]:

```
1 1000:10:2
2 T0(2,5):
3 T1(1,4):_(1,0):
4 T1(1,4):_(1,0):
```

[1]Che non significa quanto gli archi di sorgente sono pieni ma quanto sarebbero potuti essere pieni: infatti ogni arco che esce dalla sorgente deve essere riempito per intero per condurre ad una potenziale soluzione ammissibile.

[2]Non si tratta degli stessi *task* di pagina 4

```
5  T0(2,5):
6  T1(1,4):_(1,0):
7  T0(2,5):
8  T1(1,4):_(1,0):
9  _(2,0):
10 T0(2,5):
11 T1(1,4):_(1,0):
12 0 1 2 2 3 3 4 6 6 7
13 1 2 2 3 3 4 6 6 7
14 1 1 2 2 3 5 5 6
15 0 1 1 2 4 4 5
16 1 1 2 4 4 5
17 0 1 3 3 4
18 1 3 3 4
19 2 2 3
20 0 1
21 1
```

Nella prima riga sono specificate le informazioni relative alla *timeline* nel formato `timeScale:frameCount:frameSize` mentre dalla seconda fino alla undicesima sono specificati, separati dal simbolo `:`, i *job* che riempiono ogni *frame*, un *frame* per riga. I *job* il cui nome è composto solo dal carattere *underscore* (_) sono in realtà indicazioni del tempo in cui la CPU si trova nello stato *idle*. Pertanto *underscore* non è un carattere valido con cui iniziare il nome di un *job*. Il formato per la specifica di un *job* è `name(worktime,deadline)` e per i *job idle* il parametro *deadline* viene ignorata – qui indicata con 0.

Infine, dalla alla riga 12 alla fine del *file*, è riportata la tabella del tempo residuo come richiesto dal passo 1 del *test* di ammissibilità a pagina 7 e calcolata con la formula 1.2.

2.2. Executor

Prima di analizzare nel dettaglio l'implementazione del *dispatcher*, è bene capire come è stata progettata e come funziona la piattaforma *embedded* su cui si è basata l'implementazione. Diamo una panoramica sul modello dei

processi di Windows CE e chiariamo a livello generale le linee guida che hanno regolato lo sviluppo di Executor.

2.2.1 Il modello dei processi

Windows CE, dalla versione 3.0, ha uno *scheduler* a priorità multilivello con 256 livelli di priorità a partire da 0, il più alto, fino a 255, il più basso. I processi, contrariamente a quanto avviene in Windows XP, non hanno una classe di priorità – per la precisione, in Windows CE non è nemmeno del tutto corretto parlare di "processi" come veri oggetti del sistema – e i *thread* possono assumere qualunque dei 256 livelli. Un *thread* nello stato *ready* viene eseguito solo se non esiste alcun *thread* nello stato *ready* di priorità superiore. All'interno dello stesso livello di priorità, i *thread ready* vengono schedulati in modo *round robin* con *quanto* configurabile (per *default* di 100 ms); tutto quanto detto vale a meno che non si presentino nel sistema *thread* di tipo *run–to–completion*, cui Windows CE fornisce supporto.

Un ambiente tipico di Windows CE lancia all'avvio un certo numero di processi con un certo numero di *thread* non noto a priori[3] che possiamo assimilare ai *task* aperiodici. Per questo motivo è necessario uno strumento di controllo che integri e piloti le decisioni dello *scheduler* nativo in modo da garantire il rispetto dei vincoli temporali per i processi critici. Questo strumento è il *dispatcher*, un processo ad alta priorità che gestisce i processi periodici e sporadici in base ad una tabella staticamente costruita dallo *scheduler off-line*. Il *dispatcher* è in grado, sulla base dei dati che gli sono forniti dallo *scheduler off-line* e da alcune semplici elaborazioni *on-line*, di decidere del destino dei processi dell'intero sistema agendo sui meccanismi nativi di Windows CE per ottenere il risultato voluto.

[3] In effetti è determinabile in base alle *feature* e ai *driver* inclusi nella piattaforma costruita con Platform Builder oppure *on-line* mediante apposite chiamate all'API di *debug* ToolHelp [7]

2.2.2 L'applicazione

In un sistema che dispone di uno *scheduler* nativo a priorità può non essere semplice garantire il rispetto dei vincoli temporali per un certo insieme di processi in ogni situazione, se lasciati liberi di girare nel sistema privi di controllo. In ogni articolo che riguarda Windows CE viene riportato il fatto che si tratta di un sistema operativo *hard real-time* sottointendendo una serie di ipotesi che poco c'entrano con l'effettiva definizione del termine. È vero che Windows CE è un sistema in cui fattori come la latenza sono considerati critici e molto sforzo si è profuso per adeguarli alle aspettative degli utenti di un sistema con caratteristiche *hard real-time*. È altresì vero che Windows CE offre la possibilità di assegnare a *run-time* priorità arbitrarie ai propri *thread* e a quelli di qualunque altro processo del sistema, come pure di intervenire sullo stato dei *thread*, sospendendoli o riattivandoli quando necessario o addirittura imporre che un dato thread non possa subire *preemption* da parte del resto del sistema. Tutto questo è molto utile per una certa classe di applicazioni per le quali si richiede un grado di determinismo piuttosto lasco e che basano le loro operazioni sulla speranza che nessun altro processo – o sufficientemente pochi altri – si comporti come loro. Questo panorama richiede l'impiego di uno strumento di controllo per portare ordine e garantire un livello di determinismo più stringente di quello fornito da Windows CE per sè.

2.2.3 I thread di Executor

Executor si compone di alcuni *thread* di supporto che vengono qui di seguito esposti:

Schedule ha la responsabilità di calcolare, all'inizio di ogni *frame*, la porzione di tempo libero lasciata dai *job* periodici e, se essa risulta non nulla, di passare il controllo ai processi aperiodici o, qualora ce ne fossero in attesa, ai processi sporadici. Sua è inoltre la responsabilità del rilascio dei *thread* sporadici nel sistema. All'approssimarsi dello scadere del tempo libero, il *thread* Schedule riprende il controllo e lo

cede al *thread* Executor ponendosi in attesa della sua segnalazione per l'esecuzione del *frame* seguente.

Executor esegue in un ciclo i *job* di ciascun *frame* sincronizzandosi con il *thread* Schedule per il frame seguente. Riceve da Schedule, attraverso una coda di messaggi, il numero del *frame* da eseguire e crea i relativi *thread* applicativi secondo la tabella statica, in ordine di esecuzione per priorità decrescente, così che sia lo *scheduler* nativo ad effettuare lo *switch* tra i *thread* minimizzando l'*overhead* dovuto all'azione di Executor. Contestualmente avvia anche un *watchdog timer* per ciascun *thread* applicativo per controllare che vengano rispettate le *deadline*: in caso contrario, effettua la segnalazione al *thread* Error.

Error si pone in attesa delle segnalazioni di errore che possono giungere dagli altri *thread* mediante una coda di messaggi. All'arrivo di una segnalazione d'errore viene intrapresa l'azione corrispondente per la gestione della situazione critica.

MCJ (i.e. *Mode Change Job*) è il *thread* incaricato di importare in Executor la configurazione dal *file* prodotto da Configurator. Viene eseguito la prima volta per caricare la configurazione iniziale e può esserne richiesto l'intervento in seguito per cambiarla a *run-time*. Agisce su un insieme di variabili condivise, condizione per cui, quando viene eseguito come *job* sporadico, possono venirsi a creare condizioni problematiche dovute alla mutua esclusione. Pertanto viene implementato in modo da essere eseguito più velocemente possibile ed avere il minore impatto sulle prestazioni generali del sistema.

Per un efficace controllo sull'esecuzione delle varie classi di *thread*, è necessario che i *thread* di controllo abbiano priorità più alta rispetto a quelli applicativi, per questo vengono usati i livelli più alti possibili per un sistema Windows CE, a partire da 0 per Executor e Schedule, a scendere verso i più bassi per Error, MCJ e tutti gli altri. Alcuni *thread* del sistema operativo occupano livelli di priorità piuttosto alti: è possibile rilevare la priorità

massima occupata con l'API ToolHelp [7] e organizzare di conseguenza lo spazio delle priorità.

2.2.4 Sincronizzazione in Windows CE

Le sincronizzazioni tra i *thread* di supporto e quelli applicativi sono ottenute utilizzando opportuni costrutti di Windows CE. In particolare per la comunicazione interprocesso vengono usati gli eventi e le code di messaggi punto–punto mentre per il controllo dei *thread* vengono sfruttate le chiamate allo *scheduler* nativo per la sospensione ed il risveglio, nonché per il cambio dei livelli di priorità. Vediamo per sommi capi in cosa consistono gli oggetti di sincronizzazione usati, poi diamo uno sguardo alle funzioni di sincronizzazione.

Oggetti di sincronizzazione

Event è un oggetto assimilabile ad un semaforo binario che può assumere i due stati *signaled* e *nonsignaled*. I *thread* possono effettuare una chiamata di tipo sospensiva in attesa che l'oggetto assuma lo stato *signaled*. A seconda del modo in cui viene costruito, un Event agisce in modo diverso nei confronti dei *thread* sospesi su di esso: se è un evento a *reset* manuale, al momento del *signal* esso rilascia tutti i *thread* su di esso sospesi ed è necessario resettare l'oggetto manualmente, mentre se è a *reset* automatico rilascia uno solo dei *thread* in attesa riportandosi nello stato *nonsignaled* se ci sono altri *thread* nella coda e rimanendo nello stato *signaled* se invece la coda è vuota. Non è chiaro dalla documentazione se la coda di attesa sia ordinata per istante di arrivo e quindi è lecito supporre che, in caso di *reset* automatico, venga liberato un *thread* scelto a caso.

p2p Message Queue è un sistema di comunicazione interprocesso leggero ed efficace per messaggi di modeste dimensioni. La documentazione riporta che per messaggi di una certa entità è preferibile usare un *file* o un *database* sebbene questi siano meccanismi ben più lenti.

Una coda di messaggi può essere creata in lettura o in scrittura ma per avere entrambi i diritti di accesso è necessario creare due volte la stessa coda con diverse impostazioni. I messaggi vengono accodati in ordine di arrivo e ad ogni arrivo di un nuovo messaggio, i *thread* lettori vengono avvertiti: solo uno di essi però avrà accesso al dato poiché un'operazione di lettura include anche la rimozione dalla coda del dato letto.

Tutti gli oggetti di sincronizzazione possono avere un nome che li identifica univocamente nel sistema e che viene assegnato dall'utente in fase di creazione. Se più *thread* vogliono usare il medesimo oggetto di sincronizzazione, possono creare un *handle* locale passando il nome globale dell'oggetto: se l'oggetto non esiste verrà creato, se invece esiste verrà restituito un *handle* all'oggetto già esistente. Lo spazio dei nomi è locale ad ogni classe di oggetto, quindi potremo avere un Event e una Message Queue con lo stesso nome, a differenza di quanto avviene nei Windows per *desktop* in cui lo spazio dei nomi è globale.

Funzioni di sincronizzazione

In Windows CE esistono due tipi di funzioni di sincronizzazione, le chiamate sospensive e le segnalazioni. Una chiamata sospensiva ha in genere l'effetto di mettere il *thread* chiamante in attesa di uno o più eventi segnalati su un oggetto di sincronizzazione. Le chiamate di tipo *wait* possono essere effettuate su oggetti di tipo *Event*, *Mutex*, *Semaphore*, *Process* e *Thread*; inoltre una chiamata *wait* può essere o meno dotata di *timeout*, specificato come parametro della chiamata stessa, scaduto il quale la chiamata ritorna restituendo un codice di errore. Le relative chiamate di tipo *signal* sono diverse per ogni tipo di oggetto su cui vengono effettuate e vanno dalla semplice segnalazione ad operazioni di scrittura ad esempio nel caso di una coda di messaggi. Per tutti gli ulteriori dettagli ed una trattazione aggiornata, si consiglia di consultare MSDN Kernel Synchronization Functions [6].

Conclusioni

Abbiamo qui discusso l'eventualità della ricerca di una schedulazione *real-time non preemptive* per un insieme di *task* periodici con caratteristiche note a priori; abbiamo inoltre considerato la possibilità di comparsa di *task* sporadici con caratteristiche note al momento del rilascio per cui sono richieste prestazioni di tipo *hard real-time* affiancando il tutto ad eventuali *task* aperiodici senza particolari requisiti, estranei al sistema dei *task* critici.

Ci sono alcuni buoni motivi per scegliere in questi casi l'approccio *clock driven* a schedulazione ciclica statica, tra cui:

semplicità concettuale: costruire *off-line* una tabella che rispetti i vincoli temporali è agevole e consente di sfruttare potenze di calcolo adeguate per implementare metodi eleganti e di una certa onerosità computazionale;

riconfigurazione: un approccio di questo tipo è facilmente riconfigurabile al volo, dato che si basa su un certo numero di informazioni acquisite *una tantum* per ogni configurazione: è infatti sufficiente cambiare la tabella nel dispositivo *embedded* per cambiare modalità operativa. Inoltre è fortemente favorito dai sistemi che vengono raramente modificati una volta configurati;

dipendenze complesse: sebbene si sia fatta l'ipotesi iniziale di un insieme di *task* indipendenti dal punto di vista delle risorse, in virtù della semplicità concettuale dell'approccio è possibile trovare soluzioni altrettanto eleganti per quanto complesse senza curarsi troppo delle

richieste di potenza di calcolo che non andranno certo a minare le prestazioni del dispositivo *embedded*;

sincronizzazioni non esplicite: proprio perché è possibile gestire le dipendenze al momento della creazione della tabella statica, non è necessario prevedere meccanismi di sincronizzazione tra i *thread* applicativi che sarebbero potenziale veicolo di fatali *deadlock*;

***overhead* limitato:** se i *frame* vengono scelti sufficientemente grandi si ha già una sensibile riduzione degli effetti del lavoro dei *thread* di controllo sui tempi dei *thread* applicativi. Inoltre i *thread* di controllo non devono lavorare molto poiché tutti i calcoli più gravosi sono già stati effettuati a priori, riducendo così ulteriormente l'impatto sul sistema di *job* critici;

validazione: è possibile predisporre adeguate simulazioni esaustive e *benchmark* pratici per verificare l'efficacia e funzionalità del metodo.

Alla luce di ciò, la vera difficoltà incontrata durante lo sviluppo delle applicazioni è stata riguardo la costruzione della tabella statica, in quanto ben poca documentazione – e per lo più imprecisa o incompleta – è reperibile, specialmente riguardo la costruzione del grafo del sistema. L'implementazione che viene allegata a questo lavoro è comunque stata testata con alcuni esempi di relativa facile soluzione con carta e penna e che fanno parecchio uso del tempo di CPU: tranne per i casi in cui una schedulazione non esiste o richiede parecchio lavoro aggiuntivo estraneo all'implementazione per essere trovata, tutte le prove sono state condotte a buon fine producendo configurazioni valide e pronte per essere caricate nel dispositivo *embedded*.

Alcune sequenze di simulazione hanno rivelato l'opportunità di mantenere basso l'uso della *cpu* per consentire il più possibile l'esecuzione dei *task* sporadici ed aperiodici. L'esistenza di un metodo per la costruzione di una schedulazione statica non deve quindi costituire un alibi per non ottimizzare i *thread* applicativi, specialmente in presenza di un sistema operativo di tipo *general purpose*, di cui Windows CE è un esempio, in

cui è necessario che anche ad altri *thread* – talvolta piuttosto importanti per la salute del sistema operativo stesso – sia garantito il diritto di esecuzione. Empiricamente possiamo considerare un utilizzo massimo del 70% del tempo di CPU per essere relativamente sicuri che tutto funzioni come deve senza grosse limitazioni.

Pensando alle possibili applicazioni di questo lavoro, dobbiamo tenere bene a mente la questione dell'*overhead*: non aspettiamoci di poter pilotare un aereo *jet* con Windows CE e questo lavoro né tantomento regolare la temperatura del nocciolo di un reattore nucleare, piuttosto applicazioni dai ritmi più blandi, rischio minore ma comunque per cui l'*hard real-time* è una necessità sono un buon banco di prova e piattaforma per sviluppi futuri. Allo stato di questo lavoro, il limite più importante è proprio l'approccio *user space* che ha il *dispatcher*: esiste la possibilità[4] di riprogettare il componente come *driver* per agganciarsi a più basso livello allo *scheduler* nativo di Windows CE attraverso la funzione *OEMInterruptHandler* che gestisce tutte le *interrupt request*. Lo *scheduler* nativo di Windows CE viene attivato ogni volta che sono trascorsi un numero configurabile di *tick* del *OSTimer* e, per far questo, la funzione *OEMInterruptHandler* restituisce al *kernel* l'*id* logico SYSINTR_RESCHED. L'idea, a grandi linee, è di creare una IST con un proprio *id* logico che gestisca le elaborazioni che ora vengono effettuate in *user space* e che venga attivata dalla funzione *OEMInterruptHandler* opportunamente modificata per restituire l'*id* logico della *routine* di nostro interesse. Per ulteriori dettagli si consulti Panizzo [8], §3.5.

[4]Questo discorso è valido per Windows CE .NET 4.2 perché dalla versione 5.0 il sistema di gestione delle interruzioni è stato modificato.

Appendice A

Piattaforma hardware di sviluppo

In questa appendice si darà una descrizione generale dell'architettura della CPU impiegata nella scheda UNI-PD-PXA e poi una scorsa alle principali caratteristiche della scheda stessa.

A.1. ARM

Il primo RISC in assoluto è nato nel 1985 da ACORN Computer Group. Nel 1997 il RISC della ACORN debutta come primo processore RISC utilizzato in computer a basso costo. La caratteristica delle macchine RISC è che ogni istruzione viene decodificata per pilotare direttamente le risorse hardware e, più interessante, mediamente ogni istruzione viene eseguita in un solo ciclo di *clock*.

Nel novembre 1990 viene costituito il consorzio "Advanced RISC Machine" (ARM™) ad opera di Apple, ACORN e VLSI Technology. Nel 1991 ARM introduce il suo primo *core* RISC *embedded*: l'ARM6. Gli obiettivi di ARM sono le prestazioni elevate, l'alta integrazione ed il basso consumo energetico dei componenti. Il consorzio ARM progetta sistemi e microprocessori innovativi che poi fornisce alle maggiori compagnie mondiali del settore, che a loro volta le implementano nel proprio silicio. L'architettura ARM ha negli anni monopolizzato il mercato, giungendo nel 2001 a coprire oltre il 75% delle applicazioni RISC a 32 bit.

Nel 1998 Intel acquisisce Digital Equipment Corporation, ereditando così la tecnologia Strongarm™ ideata da Digital successivamente ad un

accordo con ARM, dalla quale aveva ottenuto il permesso di modificare il *core* ARM mantenendone la compatibilità *software*.

Il *core* XScale® è il primo RISC ARM completamente disegnato ed ottimizzato in casa Intel in tecnologia .18 μm. Basate su questo *core*, Intel ha ad oggi realizzato oltre quattordici famiglie di prodotti diversi, ognuna delle quali è stata pensata per una fascia diversa del mercato. Il componente che è alla base della piattaforma descritta in questa appendice è uno di questi quattordici ed è siglato PXA255.

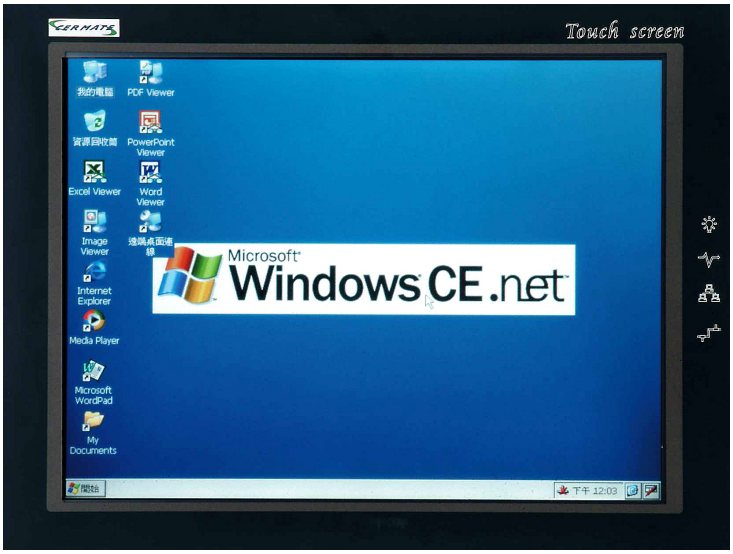

Figura A.1: Una piattaforma equipaggiata con Windows CE .NET, processore PXA255 da 400 MHz e display con risoluzione 800×600 a 24 bit.

A.1.1 Architettura ARM

Il set di istruzioni della architettura ARM è evoluto significativamente da quando è stato introdotto la prima volta e continuerà ad evolvere in futuro. Si sono succedute negli anni 5 versioni del *set* di istruzioni, denotate da un numero incrementale da 1 a 5. Oltre al numero della versione, a volte si evidenziano delle lettere aggiuntive che indicano la presenza di altre

istruzioni addizionali. Le 5 versioni del *set* di istruzioni della architettura ARM sono:

Versione 1: questa versione non è mai stata usata in prodotti commerciali. Essa contiene:

- istruzioni di base (non include la moltiplicazione);
- istruzioni di lettura e scrittura per byte, word e multi-word;
- istruzioni di salto;
- istruzione di interrupt software.

Inoltre la prima versione del *set* di istruzioni ARM aveva indirizzi da 26 bit.

Versione 2: con questa versione si sono avute le seguenti estensioni:

- istruzioni di moltiplicazione e di moltiplicazione con accumulo;
- supporto per coprocessore;
- due registri extra per il *fast interrupt mode*;
- istruzioni atomiche (indivisibili) di lettura-scrittura, chiamate SWP e SWPB.

Anche la versione 2 e 2a avevano indirizzi da 26 bit. Le versioni fino alla 2 sono oggi obsolete.

Versione 3: con questa versione gli indirizzi sono da 32 bit e, di conseguenza, lo spazio di indirizzamento è stato esteso a 4 GiB. Le informazioni di stato che precedentemente venivano memorizzate nel registro R15 sono state spostate su un nuovo registro, il *Current Program Status Register* (CPSR) e sui *Saved Program Status Register* (SPSR), che sono stati introdotti per preservare il contenuto di CPSR quando si verifica una eccezione. Ne consegue che con la versione 3 sono state introdotte due nuove istruzioni (MRS e MSR) per permettere l'accesso a CPSR e agli SPSR. La versione 3 include anche due nuovi

modi di operare, necessari per rendere possibile l'uso delle eccezioni di *data abort*, *prefetch abort* e *undefined instruction* direttamente dal sistema operativo.

Versione 4: con questa versione sono state create le seguenti estensioni:

- istruzioni di lettura e scrittura di (16 bit);
- istruzioni di lettura con estensione del segno, di byte e halfword;
- nella versione T è stata introdotta una istruzione necessaria al trasferimento in *thumb mode*;
- un nuovo modo privilegiato che usa i registri del modo *user*.

Versione 5: con questa versione sono state create le seguenti estensioni:

- è stato migliorato il passaggio al modo *thumb*;
- consente l'uso delle stesse tecniche di generazione del codice sia nella modalità *thumb* che nella modalità *non thumb*;
- è stata introdotta una istruzione di *breakpoint software*;
- sono state introdotte più opzioni per l'uso di coprocessori.

Oltre alle precedenti versioni, sono stati introdotte anche delle varianti, di volta in volta indicate con delle lettere.

Il *set* di istruzioni *thumb* (variante T) è un *subset* del *set* di istruzioni ARM. Le istruzioni *thumb* hanno una dimensione che è pari alla metà della dimensione delle istruzioni ARM (16 bit invece che 32). Il risultato che ne deriva è che normalmente può essere ottenuta una maggiore densità di codice utilizzando il *set thumb* invece che il *set* ARM. Utilizzando il *set* di istruzioni *thumb* si hanno due limitazioni:

- per la realizzazione della stessa funzione, il codice *thumb* usa più istruzioni del codice ARM e quindi il codice ARM è più indicato per massimizzare le prestazioni di un codice con di tempi di esecuzione critici;

- il *set* di istruzioni *thumb* non include alcune istruzioni necessarie per la gestione delle eccezioni.

La presenza del *set* di istruzioni *thumb* è denotata dalla lettera T (non si applica alle versioni antecedenti alla versione 4).

Il *set* di istruzioni di moltiplicazione con risultato a 64 bit (variante M) include 4 istruzioni che realizzano le operazioni $32 \times 32 \rightarrow 64$ e $32 \times 32 + 64 \rightarrow 64$ con accumulo. La presenza di queste istruzioni è denotata dalla lettera M. La prima introduzione di queste istruzioni si è vista a partire dalla versione 3 del *set* di istruzioni ARM.

Il *set* di istruzioni DSP (variante E) include un numero di istruzioni extra che migliorano le prestazioni del processore ARM in applicazioni di *digital signal processing*.

A.1.2 Architettura XScale

L'architettura Intel XScale è compatibile con il *core* ARM v5TE su cui si basano diverse famiglie di prodotti Intel, da computer palmari a navigatori GPS, da semplici telefoni cellulari a complessi dispositivi di comunicazione dotati di accesso *wireless* ad Internet, da soluzioni di *storage* con *controller* RAID a *router* ad alte prestazioni per *backbone*.

Il *core* XScale può operare in uno dei sette modi seguenti: *User*, *System*, *Supervisor*, *Abort*, *Undefined instruction*, *Fast Interrupt* e *Normal Interrupt*. Esso comprende 16 registri a 32 bit (R0 ÷ R15), di cui R13 è lo *stack pointer*, R14 il *link register* e R15 il *program counter*. Supporta sia la rappresentazione *big endian* che la rappresentazione *little endian* anche se nella versione del *core* integrata nel PXA255 è stata eliminata la rappresentazione *big endian*. Il passaggio dall'una all'altra modalità di funzionamento è determinato impostando il bit B del *control register* (coprocessore 15, registro 1, bit 7). Lo stato di *default* al *power on* è *little endian*.

Rispetto allo standard ARM v5TE, il *core* XScale implementa alcune ulteriori estensioni:

- è stato aggiunto un coprocessore DSP (CP0) che contiene un accumulatore a 40 bit e 8 nuove operazioni nello spazio coprocessore;
- sono state aggiunte delle funzionalità al coprocessore 15 ed è stato creato il coprocessore 14.

Ulteriori dettagli sulla architettura XScale, nonché sul suo *set* di istruzioni aggiunte, possono essere reperiti nel documento Intel 278525-001.pdf, che può essere scaricato direttamente dal sito Intel [4] o ordinato gratuitamente. Informazioni su come programmare il processore PXA255 si trovano invece nel documento [2].

A.2. La scheda UNI-PD-PXA

La scheda UNI-PD-PXA è stata ideata e progettata con lo scopo di rendere disponibile, presso i laboratori didattici del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, un *target* basato su un *core* RISC all'avanguardia e di grande diffusione nei sistemi *embedded*: il *core* ARM. Il componente scelto è il PXA255 che, oltre ad integrare uno tra i più diffusi *core* ARM oggi disponibili (XScale), presenta un elevato numero di interfacce interne per dispositivi periferici che lo rendono particolarmente adatto ad un ambiente didattico, perché consente di realizzare applicazioni sui più diffusi dispositivi digitali oggi disponibili: *controller* LCD, PCMCIA, AC97, USB, seriali sincrone, I2C ecc.

Al fine di consentirne un facile uso in laboratorio, la scheda è stata inserita in un *box* e munita di alcune interfacce e di alcuni dispositivi che possono essere utili allo studente nella realizzazione sia di piccoli esempi che di eventuali applicazioni più complesse: *display* LCD STN 320x240 con 16 bit di profondità di colore, 16 *microswitch* mappati in memoria e direttamente utilizzabili, 16 LED, 2 pulsanti direttamente connessi a dei pin *general purpose* I/O, un CODEC audio, una interfaccia per schede *CompactFlash*, 4 porte seriali, un USB *client*, due connettori a cui si può accedere rimovendo il coperchio in plexiglas presente sul *box* e ai quali afferiscono

tutti i segnali di *bus* del PXA255, nonché i segnali delle interfacce che non sono state rese direttamente disponibili all'esterno del *box*.

In figura A.2 viene mostrato il *box* con il pannello frontale e il *touch screen* in dotazione alla scheda.

Figura A.2: Vista frontale del box che contiene la scheda UNI-PD-PXA

A.2.1 L'architettura della scheda

Il cuore di tutta la scheda è il PXA255. Va notato che tutta l'elettronica esistente esternamente al PXA255 è passiva[1] ad eccezione del CODEC audio UCB1400.

Connessioni e preparazione della scheda al *power on*

Per alimentare la scheda, bisogna individuare il connettore di alimentazione posto sul retro del *box*. A questo connettore deve pervenire una tensione continua nel range 7,5 V ÷ 15 V e con il positivo interno. Qualora venissero utilizzati alimentatori diversi da quelli forniti in dotazione, si suggerisce di verificare che non abbiano un eccessivo *ripple* di tensione

[1] Ovvero non richiede alcuna configurazione.

(0.4 V massimo) e che siano in grado di erogare una corrente continua di circa 1.5 A.

Power on

Una volta predisposto quanto descritto al punto precedente, si è pronti ad accendere la scheda agendo sull'apposito interruttore. Si descrive ora sinteticamente quanto avviene subito dopo aver dato alimentazione:

- portando l'interruttore sulla posizione ON, partendo da una tensione in ingresso che può variare nel range 7,5 V ÷ 15 V, si alimenta la scheda secondo le temporizzazioni richieste dai dispositivi afferenti alle varie sezioni di alimentazione (5 V, 3,3 V e 1,4 V). L'accensione viene segnalata dal led verde posto sul pannello frontale;

- al *power on*, dopo il fronte di salita del segnale di *reset*, viene iniziata l'esecuzione della istruzione situata all'indirizzo 0 di memoria. A questo stesso indirizzo è mappato il segnale *chip select* CS0, per cui l'accesso all'indirizzo 0 da parte del processore (per il *fetch* della istruzione da eseguire) ha anche l'effetto di attivare il segnale CS0. Questo segnale gestisce le memorie *flash* e, tramite le impostazioni dei *jumper* JP2, JP3 e JP4, il PXA255, al *reset*, viene informato sul tipo di *flash* utilizzate (16 bit, 32 bit, asincrone, sincrone, ecc.). In questa fase non è ancora disponibile alcun tipo di RAM e quindi non si possono utilizzare istruzioni che ad esempio dovessero fare uso dello *stack pointer*. La scheda può essere riportata in questo stato o spegnendola o agendo sul pulsante di *reset*. Per evitare *reset* accidentali, questo pulsante è interno e va raggiunto con uno strumento sottile, come la punta di una matita;

- secondo quanto dettato dallo standard ARM, l'indirizzo zero contiene l'istruzione di salto all'*entry point* del programma di inizializzazione dei vari dispositivi interni ed esterni al PXA255, il quale deve provvedere poi a passare il controllo della CPU all'eventuale siste-

ma operativo. In questo caso il *boot loader* si occupa di caricare Windows CE.

La memoria residente *on board*

La memoria con cui è stata equipaggiata la scheda è costituita da 32 MiB di *strataflash* organizzata per 32 bit, 64 MiB di SDRAM organizzata per 32 bit e 512 KiB di SRAM organizzata per 32 bit. La RAM statica è stata gestita attraverso il *chip select* CS1 e può essere tenuta sotto tampone collegando il *jumper* J1 ad una batteria di tensione nel *range* 2,9 V ÷ 3,6 V. Al *power on* l'unica memoria disponibile è la *flash*, in quanto per poter utilizzare sia la SDRAM che la SRAM è necessario eseguire prima il codice che provvede ad effettuare tutte le impostazioni necessarie per definire le modalità (programmabili) di accesso del processore alle memorie e ai dispositivi interni al PXA255.

Il CODEC audio USB1400

L'UCB1400 è un CODEC audio stereo equipaggiato con un *touch screen*. L'interfaccia verso la CPU è standard AC97 Rev. 2.1. Il suo ingresso stereo può essere connesso direttamente ad un microfono o ad un lettore CD. Sulla scheda UNI-PD-PXA l'ingresso è stato direttamente collegato al connettore presente sul frontale. L'uscita può pilotare una cuffia ma nel nostro caso è stato interposto un amplificatore audio del tipo LM4881MM ed è stato inserito nel *box* scheda un piccolo altoparlante mono. Oltre al *touch screen* a 4 fili di tipo resistivo è anche presente un ADC a 10 bit del tipo ad approssimazioni successive. La versione attuale del *firmware* presente sulle schede UNI-PD-PXA non integra il *driver* di questo componente per cui non se ne può far uso per esercizi a meno di configurarlo preventivamente. Coloro che volessero approfondire le conoscenze di questo dispositivo o volessero maggiori informazioni su come programmarlo, possono far riferimento al documento numero 939775009611 (*order number* Philips) che può essere gratuitamente richiesto a Philips o scaricato dal sito.

Figura A.3: Il retro del box con le porte seriali accessibili dall'esterno.

La *CompactFlash*

Sulla scheda UNI-PD-PXA l'interfaccia *compact flash* è stata ricavata direttamente dal PXA255 con l'esclusivo ausilio di due *transceiver* secondo quanto previsto nel documento intel 278694-001.PDF, §2.6.6 nella modalità *single slot*. I segnali di controllo non previsti direttamente sul PXA255 sono stati ricavati utilizzando dei *pin* GPIO. Il connettore CF è presente sul lato destro del *box*. Il *firmware* di scheda non integra attualmente il *driver* CF. Per ulteriori dettagli su come gestire lo standard CF, si veda il documento "CF+ *and* CompactFlash *Specification Revision 1.4*".

La *Multimedia Card*

Il *controller* MMC presente nel PXA255 è compatibile con la specifica *multimedia card system* versione 2.1 con l'unica eccezione che il trasferimento a singolo byte ed a 3 byte non è supportato. Sulla scheda UNI-PD-PXA è presente un *socket* per SD *card* che però non presenta la relativa fessura sul bordo scheda. Qualora si volesse fare uso della interfaccia *multimedia card* si possono derivare i segnali necessari dai connettori di espansione di cui si parlerà più avanti. Il *firmware* di scheda non integra attualmente il driver MMC.

Le porte seriali

Le porte seriali asincrone presenti nel PXA255 sono 4; per 3 di esse i relativi segnali sono stati riportati su altrettanti connettori posti nella parete

posteriore del *box* contenente la scheda UNI-PD-PXA. La porta seriale *full modem* (FFUART) è stata connessa al connettore a vaschetta (CANON a 9 pin - RS232) posto sul retro del *box* in prossimità dell'interruttore di accensione. Fra i segnali del PXA255 ed il connettore è stato interposto un ICL3244ECAI al fine di adattare i livelli di tensione allo standard RS232. Attualmente la connessione con l'*host computer* avviene attraverso questa porta. La seconda porta è anche denominata BTUART dove BT sta per *bluetooth*, per indicare che il suo *transfer rate* può essere programmato esattamente uguale al *transfer rate* di una connessione *bluetooth*. Sulla scheda UNI-PD-PXA anche questa porta è stata connessa ad un secondo connettore a vaschetta posto sul retro. Mentre la FFUART presenta tutti i segnali di controllo necessari ad esempio per la connessione con un *modem*, la BTUART ha solo due segnali di controllo il CTS e il RTS. I connettori a vaschetta sono stati collegati secondo lo standard RS232 e quindi possono essere utilizzati dei cavi per connessioni seriali standard. Con la terza porta seriale del PXA255 è stata realizzata una connessione RS485 resa disponibile sul retro della scheda. La quarta porta seriale (HWUART) è stata diretta sui connettori di espansione della scheda.

I connettori di espansione J6 e J7

La scheda UNI-PD-PXA è stata munita di due connettori di espansione (J6 e J7) facilmente raggiungibili rimuovendo il coperchio trasparente in plexiglas. Attraverso questi connettori viene reso disponibile il *bus* del PXA255 opportunamente isolato e le periferiche non esternate sul *box*. Grazie a ciò è possibile realizzare delle schede *plug-in*, non previste inizialmente, idonee alle applicazioni più generiche e utilizzabili direttamente sulla scheda UNI-PD-PXA.

L'interfaccia LCD

La scheda UNI-PD-PXA presenta una interfaccia LCD *general purpose* che consente l'uso diretto di quasi tutti gli LCD disponibili sul mercato:

- STN *single panel*

- STN *dual panel*
- TFT
- Monocromatici
- Colori (fino a 16 bit di profondità)
- Risoluzione fino a 640x480

Nei box installati presso l'Università di Padova è stato utilizzato un LCD STN 320x240 con 65536 colori collegato al connettore JP5. Oltre al connettore JP5 utilizzato per l'LCD attuale, è presente il connettore J15 che rende disponibile l'interfaccia LCD nella sua completezza, per cui sarebbe semplice sostituire l'LCD attuale con altri tipi di LCD.

Appendice B

Ambiente software di sviluppo

In questo capitolo verranno descritti sommariamente gli strumenti che costituiscono l'ambiente di sviluppo usato nell'ambito del lavoro svolto. La versione di riferimento del sistema Windows CE è la 4.2 per la quale sono necessari alcuni accorgimenti circa eMbedded Visual C++.

B.1. Windows CE .NET 4.2

Windows CE .NET 4.2 è il sistema operativo a 32 bit scalabile e aperto, sviluppato appositamente per equipaggiare un gran numero di dispositivi elettronici, da controllori industriali ad *hub* per le comunicazioni, da fotocamere a terminali Internet passando per i *box* interattivi per le televisioni. Viene impiegato in tutte le piattaforme che richiedono requisiti particolari come connettività, sicurezza dei dati e caratteristiche di bassa latenza ed alta interattività.

Come per tutti i sistemi operativi Microsoft è a disposizione del programmatore un'estesa e complessa API adatta allo sviluppo di applicazioni grafiche e non di varia complessità.

B.1.1 Architettura

La caratteristica di Windows CE è la sua composizione a strati orizzontali, separati logicamente l'uno dall'altro, che possono essere sviluppati in modo indipendente. Nello strato superiore sono presenti le applicazioni, in quello sottostante il sistema operativo e prima dell'*hardware* si trova lo

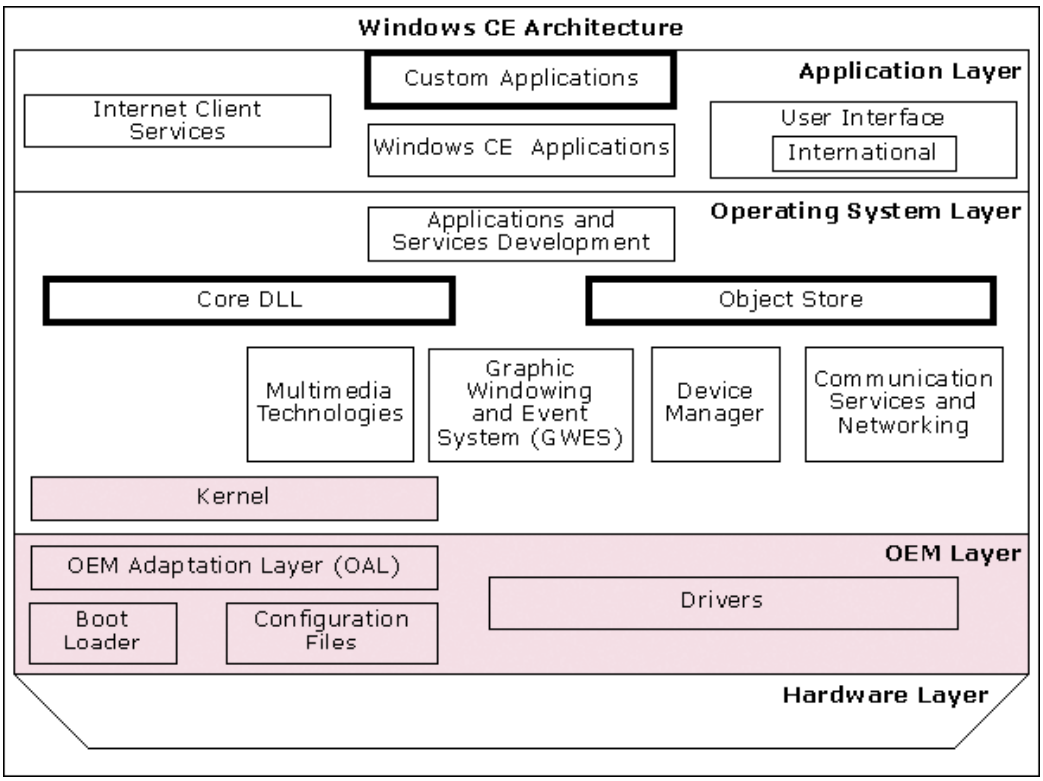

Figura B.1: Architettura di Windows CE.

strato OEM (interfaccia tra sistema operativo e *hardware*). Ogni strato accorpora più componenti che nell'insieme forniscono servizi comuni allo strato inferiore e superiore con cui è in contatto.

Di seguito si descrivono le parti più interessanti a riguardo di questo lavoro, mentre per una trattazione più completa si può vedere Zanconato [12].

Livello OEM

Si tratta dello strato che si trova a diretto contatto con l'*hardware* della scheda che comprende anche le periferiche che necessitano di apposito codice per funzionare: il *driver*.

OAL: strato di adattamento dell'OEM Il componente OAL è la parte di codice che logicamente risiede tra il *kernel* e l'*hardware* della scheda; pertanto, al momento della compilazione, il codice viene incluso nel file eseguibile del *kernel*. Il compito del OAL è di facilitare le comunicazioni tra sistema operativo e scheda gestendo le interruzioni, i temporizzatori, il risparmio energetico, il controllo delle periferiche e tutte le possibili segnalazioni provenienti dalla scheda stessa. Il OAL fornisce anche la base per lo sviluppo dei metodi di *debug*. Le *routine* contenute nel OAL servono per l'inizializzazione della piattaforma, l'esecuzione delle *routine* associate all'interruzione, per l'orologio *real-time*, per le temporizzazioni, per il *debugging*, per l'abilitazione e disabilitazione delle interruzioni e per il *profiling*.

Driver I *driver* sono componenti *software* che servono al sistema operativo per comunicare con le periferiche, solitamente fisiche, ma anche logiche, ossia programmi che il sistema operativo utilizza per il suo funzionamento: il *file system*, ad esempio, è una periferica logica. I *driver* sono composti da due interfacce: una per interagire con la periferica presente nella scheda, l'altra per mettere a disposizione del sistema operativo i servizi offerti dal *driver*.

Bootloader Il *bootloader* è un programma dalle caratteristiche minimali, risiedente nella memoria non volatile della scheda di sviluppo, a cui è affidato il compito di caricare nella memoria della piattaforma la *run-time image* e di avviarne l'esecuzione. Esistono diverse modalità di connessione tra dispositivo *host* e *target*, le più usate sono quelle Ethernet, seriali e USB. Il trasferimento dei file e dei comandi, ad esempio per la configurazione iniziale del *bootloader*, avviene utilizzando i più diffusi protocolli di comunicazione a seconda del mezzo di trasmissione scelto. Altra soluzione per l'immagazzinamento della *run-time image* è quello di utilizzare dispositivi per la memorizzazione locale dei dati (memorie *Compact Flash*, *hard disk*). Solitamente la memoria di destinazione della *run-time image* è la RAM o, tramite opportuna configurazione, direttamente la memoria non volatile della scheda. A sviluppo ultimato il *bootloader* può essere rimosso dalla scheda, anche se il suo utilizzo semplifica le operazioni di avvio della macchina. Windows CE mette a disposizione due configurazioni di *bootloader*; una utilizza la connessione seriale sia per l'impostazione sia per il trasporto dei dati, l'altra sfrutta il collegamento Ethernet per il trasferimento dei dati e il collegamento seriale per la configurazione.

Livello sistema operativo

In questo livello sono compresi gli elementi più importanti del sistema operativo, *kernel*, librerie di sistema, supporto per l'ambiente grafico, multimedia e protocolli di rete.

Kernel È l'elemento di congiunzione tra il livello Sistema Operativo e il livello OEM. Il *kernel*, costituito dal *file* `NK.exe`, è il cuore del sistema operativo poiché fornisce tutte le funzionalità base, gestione dei *thread* e della memoria, comuni a tutti i dispositivi basati su Windows CE. Se il sistema operativo è memorizzato nella ROM della scheda, il *kernel* si occupa di eseguire i moduli direttamente dalla memoria, se sono compressi provvede alla decompressione del modulo richiesto nella RAM per la sua successiva esecuzione. Il *kernel* fornisce anche caratteristiche *real-time* che

consistono nel garantire l'esecuzione di tutti i *thread* di priorità massima, nell'avere una latenza minima nell'esecuzione delle *routine* di servizio per le interruzioni ad alta priorità e un controllo preciso sullo *scheduler* e su come gestisce i *thread*.

L'ambiente grafico GWES Il componente GWES racchiude le principali caratteristiche dell'ambiente grafico di Windows CE, che comprende le API di Win32, le interfacce utente e le librerie grafiche del dispositivo. GWES fornisce l'interfaccia tra utente, applicazioni e sistema operativo.

Device Manager Il *device manager* effettua il monitoraggio del caricamento dei driver e delle loro interfacce. Il *device manager* è in continua esecuzione ed interagisce in modo stretto con il *kernel*, notificando quando l'interfaccia di un driver viene caricata, segnalando se le operazioni sulle periferiche sono valide o meno e gestendo il risparmio energetico sulle periferiche, alimentando solo quelle saranno utilizzate da lì a poco.

Object Store Solitamente nei dispositivi con Windows CE non è prevista la presenza di un disco fisso, ma per la memorizzazione permanente dei dati vengono utilizzati altri dispositivi come memorie non-volatile, NVRAM o *flash*, che vanno a formare il componente *object store*. Il compito dell'*object store* è di memorizzare in modo persistente il *file-system* e il registro di sistema. Nel caso che un dispositivo adotti una NVRAM, questa viene divisa in due sezioni: il magazzino degli oggetti (*object store*) e un'area per i programmi. Il magazzino degli oggetti è un *ramdisk* virtuale per la memorizzazione dei dati, i quali vengono conservati, anche se il dispositivo viene "addormentato" o resettato tramite comando *software*, grazie alla presenza di batterie tampone per il mantenimento dei dati (caratteristica tipica dei PDA).

Livello applicativo

Il livello più elevato è quello riservato all'esecuzione delle applicazioni. Windows CE fornisce una serie di programmi, tra i quali una versione di

Pocket Internet Explorer, *Windows Messenger* e *Windows Media Player*. Sono presenti anche numerose applicazioni di rete, sia lato *server*, sia lato *client*, da utilizzare a seconda del tipo di piattaforma che si sta sviluppando.

B.2. Configurator

L'applicazione preposta alla configurazione del *dispatcher* è dotata di una interfaccia grafica che permette l'inserimento dei processi periodici con tutte le relative caratteristiche. Non deve essere eseguita direttamente sulla piattaforma su cui agisce il *dispatcher* in quanto produce un file di configurazione che il *dispatcher* usa per gestire i processi. Va da sè che si tratta di un'applicazione grafica, multipiattaforma, facilmente portabile e scritta in C++.

B.2.1 Eclipse

Eclipse è un IDE completo per lo sviluppo, scritto in Java e con architettura a *plug-in*. Il *plug-in* per la gestione di codice C/C++ si chiama CDT ed è reperibile sul sito di Eclipse [3].

B.2.2 Qt

Qt (pr. ['kju:.ti]) è un *toolkit* multipiattaforma per lo sviluppo di programmi con interfaccia grafica basato su *widget*. Qt non è solo un *toolkit* grafico ma è anche una raccolta di classi di molti tipi, dai più comuni tipi di dato (`QList`) alle estensioni *multithread* (`QThread`) costituendo così a tutti gli effetti un *framework* completo.

Licenza

Qt è sviluppato dalla norvegese Trolltech che lo rende disponibile – nella versione 4 in uso in questo lavoro – in doppia licenza, GPL per applicazioni GPL e commerciale per applicazioni commerciali. Per lo sviluppo di

Configurator si fa uso della versione libera di Qt per compilare ed eseguire il programma ma è pienamente compatibile con la versione commerciale del *toolkit*. La versione *open source* per X11 è distribuita con licenza QPL. Tutte le versioni sono disponibili presso il sito di Trolltech [10].

B.2.3 Boost Graph Library

BGL è una libreria libera in C++ per lavorare sui grafi che fornisce un'implementazione generale dei grafi con supporto per implementazioni specializzate via *template*. Inoltre implementa alcuni tra i più noti algoritmi per lavorare sui grafi. Non è necessario compilare BGL in quanto fornisce un insieme di *header* (`.hpp`) che includono tutto il necessario per lavorare subito.

Ogni altra informazione e la relativa documentazione è reperibile sul sito del progetto [1].

B.3. Executor

B.3.1 Platform Builder 4.2

Platform Builder 4.2 include Windows CE .NET 4.2 e permette di creare la piattaforma di interfaccia col dispositivo *embedded* in uso. È inoltre possibile creare una piattaforma emulatore per CPU x86 al fine di agevolare lo sviluppo delle applicazioni. Nell'ambito di questo lavoro, è critica la fase di *test* sul dispositivo reale ma è stato comunque possibile sviluppare e verificare la correttezza del codice nell'emulatore.

Creazione di una *platform*

Si dà qui una breve panoramica indicativa dei passaggi da seguire per la costruzione della *platform*, per una descrizione più dettagliata si veda Zanconato [12]. Avviato Platform Builder, dal menu `File` si sceglie `New Platform` che avvia il New Platform Wizard, la creazione guidata di una

nuova piattaforma. Al termine del processo si possono fare gli ultimi ritocchi tramite la voce `Settings` del menu `Platform` e poi procedere alla compilazione della piattaforma, scegliendo `Batch Build` dal menu `Build`.

Creazione di un SDK

Una volta completata e verificata la platform, è necessario esportarla perché possa essere riferita da eVC. Dal menu `Platform` si sceglie la voce `Configure SDK`. La prima volta che una piattaforma viene configurata per l'esportazione, viene avviata una semplice procedura guidata, al termine della quale è necessario scegliere nuovamente `Configure SDK` per raffinare la configurazione dell'SDK che si vuole esportare. Una volta ultimata la configurazione, è sufficiente scegliere la voce `Build SDK` dal menu `Platform` e attendere il termine della compilazione. Verrà creato un file MSI con cui sarà possibile installare il SDK in modo da poterlo usare con eVC.

Molte fonti in rete consigliano di chiudere sia Platform Builder che eMbedded Visual C++ prima di installare il SDK appena creato, alcune consigliano anche di riavviare Windows prima di procedere.

B.3.2 eMbedded Visual C++ 4.0

eMbedded Visual C++ 4.0 è un IDE completo per la gestione di progetti per piattaforme Windows CE in linguaggio C/C++. È richiesto un solo accorgimento per farlo lavorare con Windows CE .NET 4.2 ovvero l'installazio del Service Pack 4. Il SP4 per eVC è un set di aggiornamenti cumulativi che ne estende la compatibilità fino a Windows CE .NET 5.0.

Assieme al IDE, vengono forniti molti utili strumenti per interagire con la piattaforma *embedded* ed eseguire operazioni di ogni tipo, dal semplice trasferimento di *file* (attraverso il Remote File Viewer) a complesse operazioni di *debug* o di *benchmark* (attraverso il Kernel Tracker).

L'unica vera e sostanziale mancanza che si sente nel programmare Win-

dows CE in C++ è l'assenza della STL[1] e quindi l'impossibilità di attuare molte pratiche comuni che questa libreria solitamente permette.

[1]Standard Template Library

Appendice C

Documentazione del codice

C.1. Common

Common è una classe che include metodi statici di utilità generale che non trovano altra collocazione. Farebbe eccezione il metodo **maxFlow** ma si è scelto di inserirlo qui per comodità e per non doverlo modificare oltre il necessario, giacché proviene dall'originale incluso nella BGL (v. [1] e §B.2.3).

```
#include <Common.h>
```

C.1.1 Metodi

static int gcd(int *a*, int *b*)

Questo metodo calcola il *greatest common divider* dei due parametri interi *a* e *b*.

static int lcm(int *a*, int *b*)

Questo metodo calcola il *lowest common multiple* dei due parametri interi *a* e *b*.

static solution* maxFlow(stringstream* *problemStream*)

Questo metodo risolve iterativamente un problema di flusso massimo ricevendolo nella forma DIMACS passatogli attraverso uno **stringtream***. Utilizza l'algoritmo **push_relabel_maxflow** della libreria BGL.

C.2. Common::solution

solution è una sottoclasse di **Common** che serve a rappresentare la soluzione del problema di flusso massimo risolto. La variabile membro **maxflow** registra il valore del flusso massimo trovato mentre la variabile membro **arcs** è un vettore degli archi che collegano i nodi dei *job* a quelli dei *frame* con il relativo flusso. Gli archi sono rappresentati dalla sottoclasse **arc** di **solution**.

C.3. Common::solution::arc

Sottoclasse di **Common::solution** che rappresenta un arco diretto, mantiene informazioni sul nodo di partenza, quello di arrivo e sul flusso che lo attraversa. Viene utilizzata per estrarre e rappresentare la soluzione di un problema di flusso massimo in modo da rendere più agevole la sua analisi ed utilizzo nel contesto della costruzione di una schedulazione ciclica statica.

C.4. INF

Classe di supporto per l'algoritmo *Iterative Network Flow*, fornisce un metodo per la costruzione del grafo del sistema e per calcolare la grandezza dei *frame* nell'iperperiodo. Fornisce inoltre i descrittori dei *task* del sistema.

```
#include <INF.h>
```

C.4.1 Metodi

bool addTask(int *p*, int *e*, int *d*, QString *n*)

Aggiunge un descrittore alla tabella dei *task*. Un descrittore è formato da periodo, tempo di esecuzione, *deadline* e una stringa che identifica il processo. Il metodo restituisce **false** se si tenta di inserire più metodi di

quanti non se ne siano specificati nella tabella dell'interfaccia ma uesto non dovrebbe mai succedere in quanto l'inserimento viene fatto in modo automatico al termine della specifica di tutti i processi.

string buildGraph(int *n*)

Costruisce un problema di flusso massimo nel formato DIMACS e lo restituisce come stringa ASCII. La stringa ottenuta viene trasformata in uno **stringstream*** che viene passato come parametro a **Common::maxFlow()**.

bool findFrames()

Cerca le dimensioni dei *frame* che rispettano i tre vincoli di pagina 3. Si è scelto di rispettare strettamente il primo vincolo per evitare la ricerca di una schedulazione *non preemptive* e quindi semplificare di molto l'implementazione sul dispositivo *embedded*. Il metodo restituisce **false** in caso non si trovino dimensioni adeguate, in questo caso il processo di schedulazione viene interrotto.

C.5. MainDlg

Questa classe implementa i metodi per l'utilizzo della GUI. La forma speciale **on_widgetname_signalname** dei nomi dei metodi – che, nel *framework* Qt, sono chiamati *slot* – è stata scelta per consentire a Qt di attuare la connessione automatica tra *signal* e *slot* che altrimenti sarebbe onere dello sviluppatore.

C.5.1 Metodi

void on_periodicAddButton_clicked()

Aggiunge un descrittore di *task* periodico alla tabella dei descrittori e aggiorna di conseguenza l'indicatore di carico della CPU.

void on_periodicAddButton_released()

Riporta il *focus* sul primo campo del modulo per l'inserimento dei valori. Si è preferito tenere separate le azioni **clicked** e **released** perché la prima esegue un'operazione riguardante lo scopo del *software* mentre la seconda esegue un'operazione sull'interfaccia utente.

void on_periodicDelButton_clicked()

Rimuove le righe selezionate dalla tabella nella finestra di dialogo e aggiorna di conseguenza l'indicatore di carico della CPU.

void on_okButton_clicked()

Avvia il processo per la ricerca e la costruzione di un schedulazione ciclica ammissibile in base ai dati inseriti nella tabella.

void on_helpDialog_clicked()

Mostra una semplice finestra di dialogo con qualche riga su come usare l'applicazione.

Appendice D

Glossario

API significa *Application Interface Programming* ed è l'insieme di funzioni di libreria che permettono di programmare in un certo ambiente o per ottenere particolari scopi.

ARM (precedentemente nota come Advanced RISC Machine e prima ancora come Acorn RISC Machine) è un'architettura per processori RISC a 32 bit largamente usata per dispositivi *embedded* che richiedono bassi consumi e tuttavia alta efficienza.

Deadline è un istante temporale in relazione col periodo entro il quale il processo deve terminare l'esecuzione. Può essere minore, uguale o maggiore del periodo ma mai minore del tempo di esecuzione.

Deadlock è una situazione critica in cui un certo insieme di *thread* rimane in stallo incapace di proseguire. Solitamente avviene per motivi legati all'attesa di risorse occupate che non vengono rilasciate o all'attesa ciclica della terminazione degli stessi *thread* coinvolti nel *deadlock* come per molti altri motivi.

Dispatcher è spesso chiamato anche *scheduler* di basso livello poiché è il componente deputato a scegliere il prossimo *task* a cui assegnare la CPU e materialmente compiere tutte le operazioni necessarie all'assegnazione effettiva. La scelta del prossimo *task* può essere presa autonomamente – nel caso di *scheduler* molto semplici – oppure ricevuta da uno *scheduler* più complesso che agisce ad alto livello. Talvolta può anche completare il lavoro dello *scheduler* compiendo operazioni semplici che altrimenti lo appesantirebbero inutilmente.

FCFS è un acronimo che significa First Come, First Served: le richieste vengono servite in ordine cronologico di arrivo.

IDE significa *Integrated Development Environment* ed è uno strumento *software* che solitamente integra in una singola applicazione l'intero *toolchain* necessario allo sviluppo in un certo ambito, da un *editor* per i sorgenti ad un compilatore ad un *debugger* e così via.

Latenza è l'intervallo temporale che intercorre tra l'istante in cui un'azione viene richiesta al sistema operativo (ad esempio l'arrivo di una richiesta di interruzione) e l'attivazione da parte del sistema operativo delle procedure richieste (ad esempio l'avvio della routine di servizio all'interruzione). Di particolare criticità per un sistema *real-time* è la latenza poiché valori troppo alti possono compromettere in varia misura le prestazioni del sistema.

Periodo è l'intervallo temporale tra due inizi successivi dello stesso processo.

Processo (anche *task*) è un'istanza applicativa che richiede, usa e rilascia risorse del sistema.

RISC significa Rationalized Instruction Set Computer (precedentemente noto come Reduced Instruction Set Computer) ed è un modello e una filosofia di progetto di CPU che preferisce l'impiego di un insieme semplice di istruzioni che impieghino circa tutte lo stesso tempo per essere eseguite.

Scheduler è il componente che decide qual'è il *task* più opportuno a cui assegnare la CPU in ogni istante. Di solito è un componente che gira ad un livello di astrazione piuttosto alta e pilota la decisione del *dispatcher*.

Task (v. Processo)

Bibliografia

[1] Boost Graph Library (2006). http://www.boost.org/libs/graph/.

[2] Descrizione del sistema UNI–PD–PXA ed esercizi (2004).

[3] Eclipse SDK (2006). http://www.eclipse.org/.

[4] Intel XScale® Technology (2006). http://www.intel.com/design/intelxscale/.

[5] Liu J. (2000). *Real-time systems*. Prentice Hall.

[6] MSDN Kernel Synchronization Functions (2006). https://msdn.microsoft.com/library/en-us/wcemain4/html/cmconKernelSynchronizationFunctions.asp.

[7] MSDN ToolHelp (2006). http://msdn2.microsoft.com/en-us/library/aa450775.aspx.

[8] Panizzo A. (2006). Benchmarking di Windows CE .NET 4.2 su piattaforma Intel XScale PXA255.

[9] Rate-monotonic scheduling (2006). http://en.wikipedia.org/wiki/Rate-monotonic_scheduling.

[10] Trolltech (2006). http://www.trolltech.com/.

[11] Wikipedia (2006). http://www.wikipedia.org/.

[12] Zanconato S. (2005). Sistemi Embedded: Windows CE per scheda UNI–PD–PXA255.

Indice analitico

Ringraziamenti e scuse

Tanto per mettere le mani avanti, mi scuso con l'eventuale a me ignoto che dovrà prendere in mano il mio lavoro se questo lavoro sarà base per sviluppi futuri: lo so che mi odierai e mi maledirai ma non l'ho fatto apposta. Capirai queste mie parole più avanti.

Questo lavoro è stato redatto con LaTeX2e, per questo vorrei ringraziare sia Donald Knuth che Leslie Lamport per avermi risparmiato tutta quella fatica di fare le cose a mano. Ringrazio anche chi mi ha riempito di dritte sull'uso di LaTeX altrimenti sarei ancora intento a strapparmi i capelli. In effetti sarebbe ingeneroso non ringraziare tutti coloro che fanno molto per l'*open source* e le libertà digitali, tranne Richard Stallman che è solo un fanatico trombone anche se tutti gli dobbiamo molto. Ecco, lo volevo dire e l'ho detto. Vai Eric, stendilo!

Passando alle cose serie, ricordo a tutti che Windows è un marchio registrato di Microsoft Corporation negli Stati Uniti d'America e in altre nazioni. Così tutte le altre applicazioni di Microsoft che ho usato e citato in questo lavoro. Quelle che non sono di Microsoft, sono in qualche modo di proprietà dei relativi sviluppatori e ad esse si applicano le varie licenze del caso.

No, così, perché Microsoft ha una pagina grossa così su queste cose, mi pareva prudente riportarlo anche qui, non si sa mai...

Questo lavoro è disponibile in formato elettronico nella sua revisione più aggiornata nel sito *web* `http://www.morpheu5.net/`

E poi, in fondo, ringrazio tutti coloro che non mi hanno negato un sorriso.

P.S. Più grossi sono, più rumore fanno quando cadono. Sì, anche nella foresta dove nessuno li sente.