3

Real-Time Kernel e VisualDSP++

In questo capitolo vengono illustrati l'ambiente di sviluppo utilizzato per la progettazione dell'algoritmo, VisualDSP++, e un componente software offerto dalla Analog Devices con il sistema di sviluppo, ovvero VDK (VisualDSP Kernel). Esso è un kernel real-time progettato per realizzare applicazioni in tempo reale sul DSP. Può essere paragonato ad un normale nucleo di un qualsiasi sistema operativo multitasking, ovviamente con una struttura estremamente semplificata e dedicata unicamente a facilitare la progettazione di software real-time e alla semplificazione della scrittura di interfacce software per l'I/O. Esistono in commercio numerosi altri kernel per il processore SHARC, ad esempio ThreadX fornito dalla Express Logic, oppure SharcOS fornito dalla JDC Electronic.

VisualDSP++ è un IDE (Integrated Development Environment) fornito dalla Analog Devices, il cui scopo è facilitare la programmazione e la verifica dei programmi scritti per processori SHARC. In generale, il VisualDSP++ supporta qualsiasi DSP prodotto dalla Analog Devices (TigerSharc, Blackfin, e 21xx). Tale ambiente può lavorare in modalità "simulazione", dove viene riprodotto il comportamento del DSP via software, in tutte le sue caratteristiche; tuttavia, in tale modalità, non è possibile testare le periferiche esterne al DSP, ovvero quelle presenti sulla board di sviluppo EZKIT. In modalità "emulazione", l'ambiente si collega, via USB, alla board di sviluppo EZKIT, ne verifica la funzionalità e

permette di eseguire il codice sul DSP reale, provvedendo unicamente a monitorare il codice e ad esplorare il contenuto della memoria.

La versione utilizzata nel presente lavoro è la 4.0, la quale gira su piattaforme Microsoft Windows®. Insieme con l'ambiente, viene fornita una nutrita libreria di funzioni matematiche; nel presente lavoro sono state utilizzate le funzioni trigonometriche coseno e arcotangente, nonché un'implementazione "Radix 2" della Fast Fourier Transform.

3.1 L'interfaccia VisualDSP++

L'interfaccia dell'ambiente VisualDSP++ è riportata in figura 3.1.

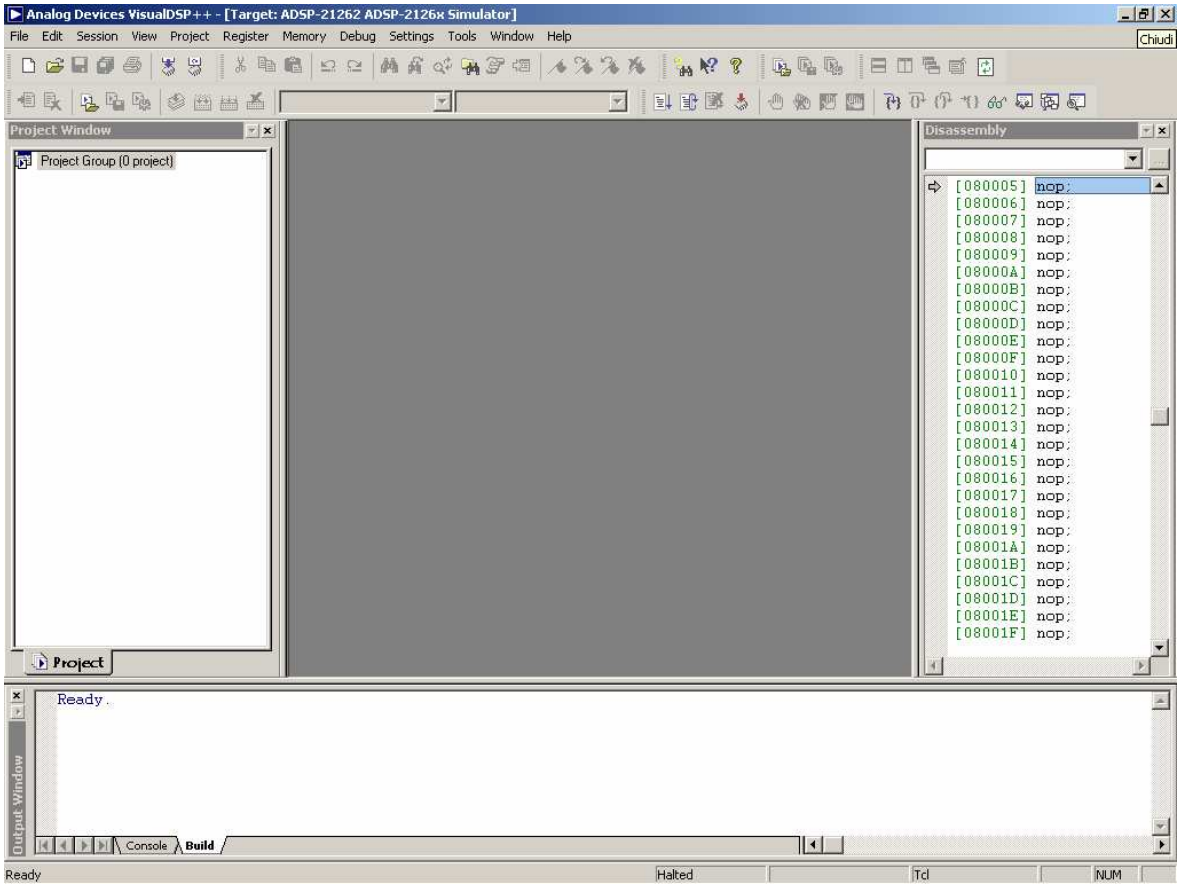

Fig. 3.1 – L'ambiente VisualDSP++

Oltre alla classica barra degli strumenti, che permette di usare tutte le caratteristiche dell'IDE, è possibile notare tre sezioni: "Project Window", la quale contiene la lista dei file contenuti nel progetto in realizzazione; "Disassembly", la quale contiene, in linguaggio macchina SHARC, le istruzioni che vengono eseguite; "Output Window", la quale contiene tutti i messaggi di stato e/o di errore prodotti durante la compilazione e la fase di linking.

Questo IDE non si differenzia tanto dai comuni ambienti di sviluppo presenti in commercio (come gli IDE Borland® e Microsoft VisualStudio®), tuttavia presenta alcune caratteristiche interessanti, le quali sono finalizzate proprio allo sviluppo di codici per DSP. Nel momento in cui viene creato un progetto, viene offerta la possibilità di creare un'applicazione stand-alone, oppure di generare un eseguibile contenente il kernel VDK (figura 3.2).

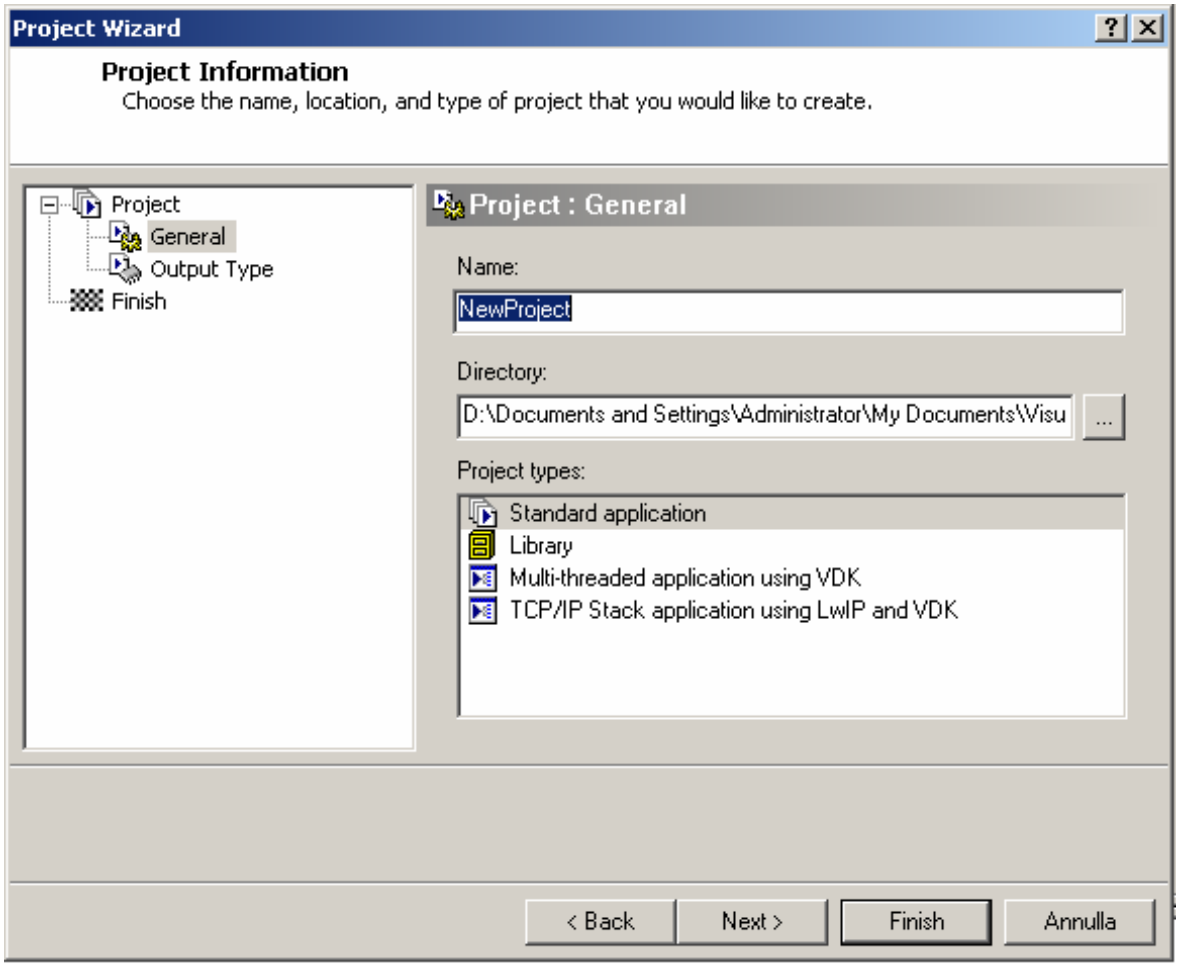

Fig. 3.2 – Creazione di un nuovo progetto

Utilizzando il kernel VDK, nella finestra di progetto è possibile configurare in modo visuale tutti i parametri del kernel, rendendo agevole la scrittura del codice e la sua integrazione all'interno del VDK.

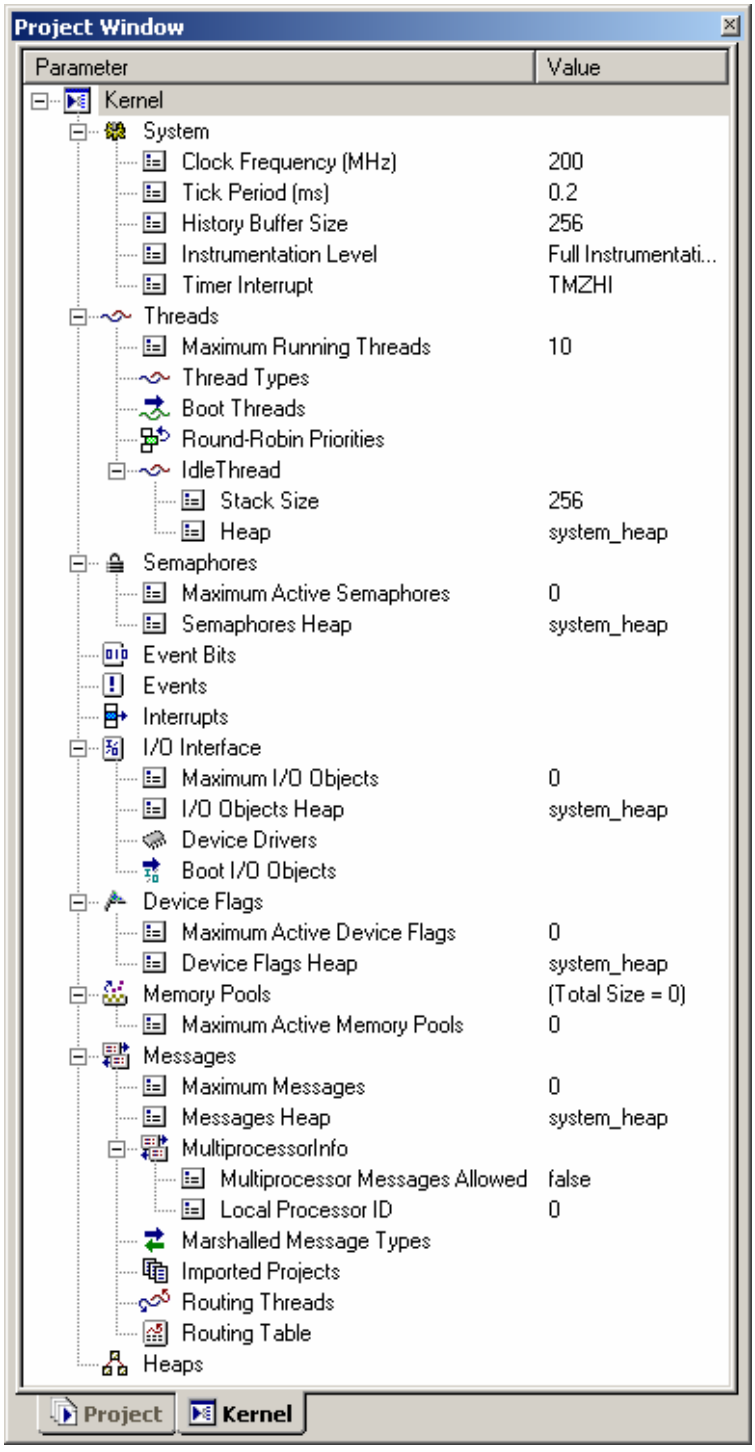

Fig. 3.3 – Parametri del kernel VDK

Come è possibile notare in figura 3.3, ogni aspetto del kernel VDK è configurabile tramite tale finestra, la creazione del codice per ogni thread contenente le sezioni di inizializzazione e uscita è automatica, così come la creazione delle strutture di sincronizzazione (semafori, messaggi, eventi). Questa flessibilità non solo rende agevole la programmazione, ma aiuta a produrre codice molto più robusto nei confronti di errori che si possono verificare al tempo di esecuzione (ad esempio, stack overflow, abort dei thread per errori di I/O, etc.)

Altra interessante caratteristica viene fornita dal "plot" dei risultati. Difatti, oltre alla normale visualizzazione del contenuto delle variabili, le cosidette "watch", il VisualDSP permette di visualizzare il contenuto di vettori in grafici 2D, e il contenuto di matrici in grafici 3D. Nel caso di elaborazioni numerica di segnali, tale opzione rende l'ambiente VisualDSP++ molto indicato per sviluppare algoritmi ad hoc.

Nella figura 3.4, viene riportato un passo dell'analisi del rumore di fase, con la visualizzazione di tutti i vettori coinvolti nell'elaborazione. E' possibile addirittura osservare l'evoluzione del codice nel tempo, poiché i plot offerti dal VisualDSP++ hanno la possibilità di interrompere l'esecuzione del codice ad intervalli di tempo regolari, in modo tale da permettere i refresh di tutti i plot. Su ogni grafico, è possibile effettuare cambiamenti di scala o trasformazioni elementari (tipo la conversione del valore in dB, come avviene in figura per la finestra FFT).

In simulazione, per la lettura dei dati del segnale campionato e la produzione dei risultati, sono stati usati degli strumenti di I/O offerti dall'IDE chiamati "stream", i quali consentono di associare ad una lettura/scrittura in memoria interna del DSP una lettura/scrittura su file.

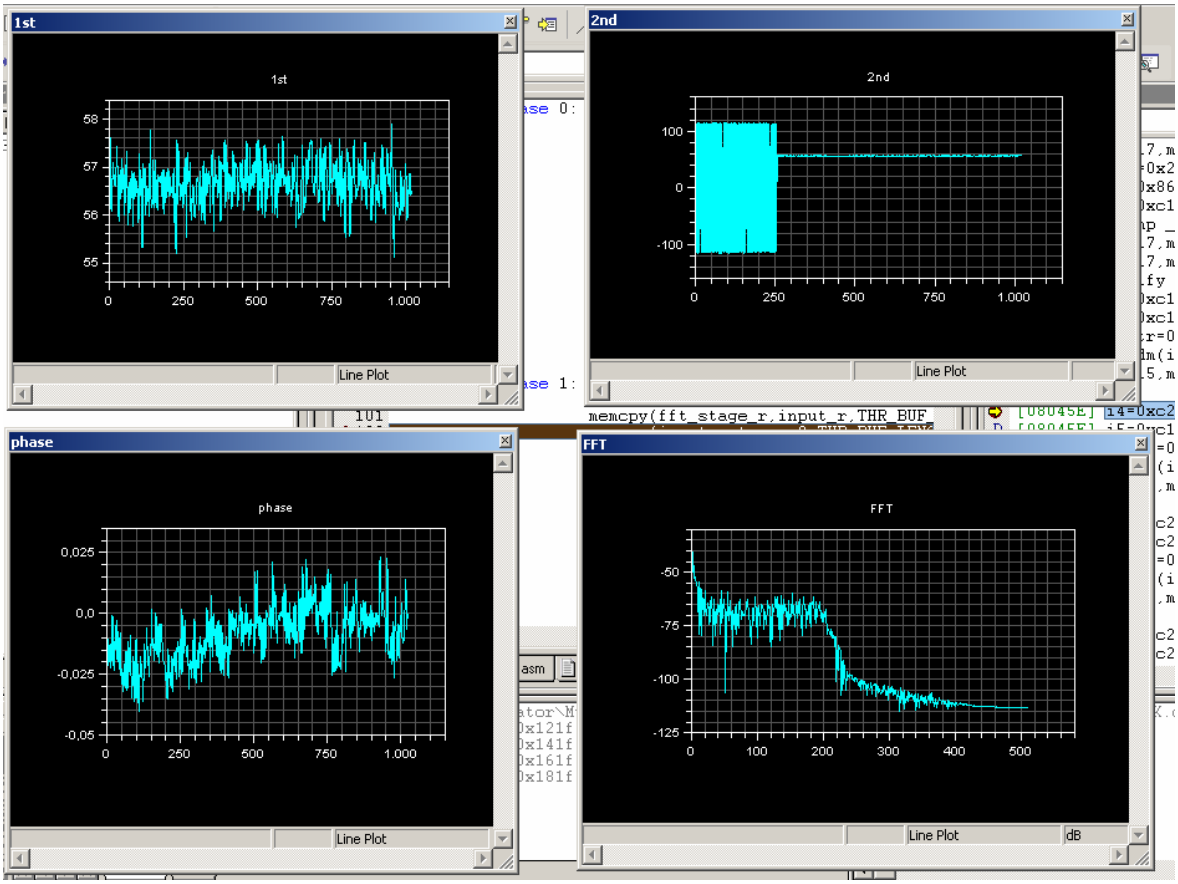

Fig. 3.4 – Visualizzazione mediante grafici dei vettori

3.2 Linear Profiling

Uno strumento utile per migliorare le prestazioni dell'algoritmo implementato è il "Linear Profiling", ovvero un tool software che misura, sul numero di cicli totali spesi per l'elaborazione, quanti di essi vengono impiegati per eseguire una particolare istruzione, e il risultato viene riportato in percentuale in una tabella ordinata per consumo di cicli in senso decrescente. In tal modo, è possibile individuare lo stadio più lento di tutto l'algoritmo e, laddove possibile, provare ad ottimizzare la sezione di codice che ha riportato il maggior consumo di cicli. La figura 3.5 mostra una misura effettuata durante una simulazione dell'algoritmo presentato in questa tesi.

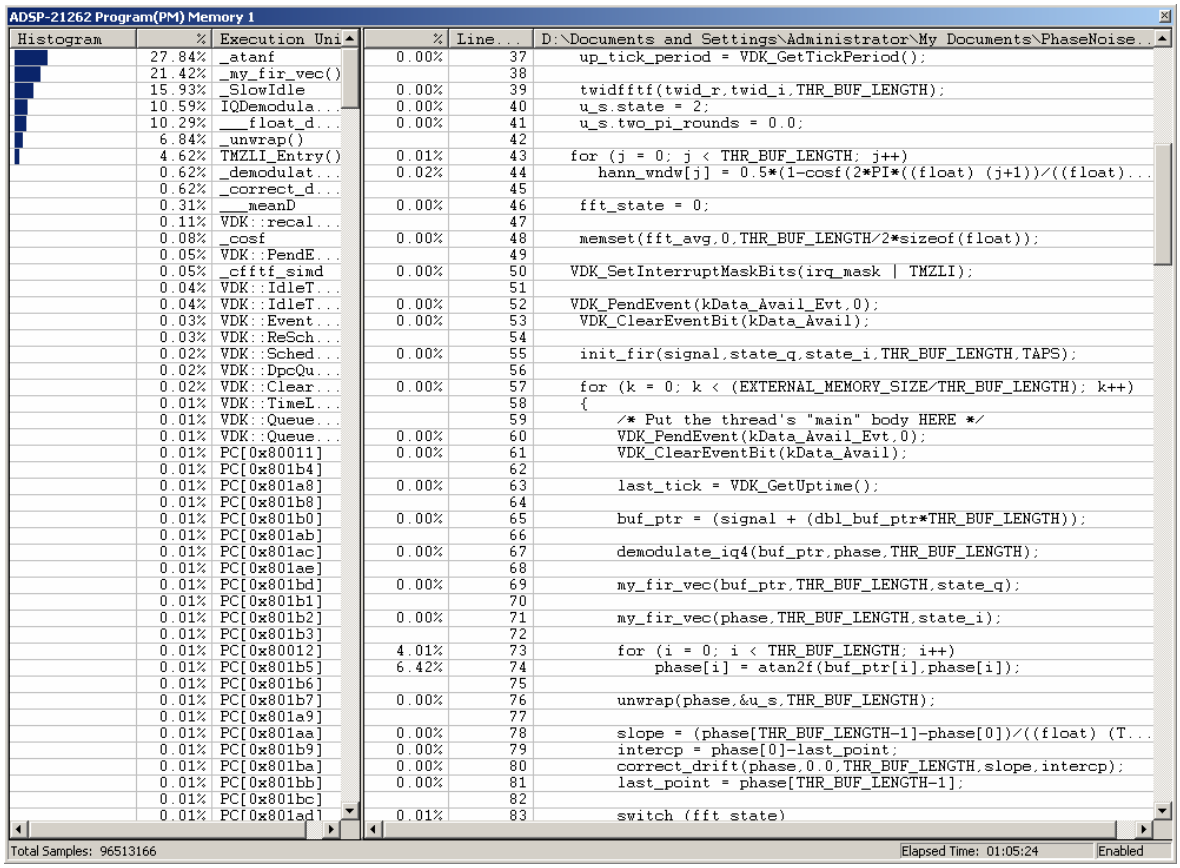

Fig. 3.5 – Profiling del codice in esecuzione

Come è possibile notare, lo stadio più lento riguarda il filtraggio FIR e il calcolo dell'arcotangente. Inoltre, è interessante osservare come un'operazione apparentemente elementare come una divisione impieghi, essendo realizzata via software, il 10.29% del tempo totale. Tale peso è dovuto anch'esso allo stadio arcotangente, il quale usa l'operazione di divisione per ogni punto del segnale da elaborare.

3.3 Kernel Real-Time VDK

A prescindere dal fornitore e dalla struttura più interna del kernel, tutti i kernel real-time forniscono i seguenti componenti:

- uno scheduler per la gestione dei thread in esecuzione sul DSP;
- primitive e strutture software per la sincronizzazione dei thread;
- un livello di astrazione hardware per la realizzazione di device drivers dedicati alla gestione dell'I/O;

Nascondendo i dettagli hardware, il programmatore può concentrarsi sulla propria applicazione, partizionando le parti di calcolo da quelle dedicate dall'I/O, e utilizzando i tempi di latenza di queste ultime per effettuare elaborazioni, abbattendo notevolmente i tempi d'elaborazione.

Il kernel VDK è un kernel multitasking preemptive, ovvero ad un thread non viene sottratta la CPU fino a quando non esegue una system call oppure non si mette in attesa del completamento di un'operazione di I/O. Nel caso in cui il thread viene bloccato, esso viene inserito in una coda di processi gestita tramite priorità; i thread vengono serviti rispettando tale priorità, senza implementare tecniche presenti in sistemi operativi ordinari, come l'"aging". Quindi un non corretto uso dei thread può portare a "starvation", ovvero attesa indefinita della CPU.

Il VDK segue una politica rigorosamente "preemptive"; tuttavia esso stesso fornisce primitive per cambiare tale politica, adottandone una "cooperative" (ovvero il thread rilascia volontariamente la CPU a favore di un altro thread), oppure "time-sharing" (ad ogni thread è assegnata una slice temporale, entro la quale procede con il suo codice).

Durante la sua esecuzione, lo scheduler segue il diagramma di stato riportato in figura 3.6.

Le priorità da assegnare ad ogni thread possono essere assegnate fin dall'inizio, o cambiate in corso d'esecuzione. Sono previsti 30 livelli di priorità, dove il primo livello indica la priorità più alta. Oltre ai thread definiti dall'utente, il kernel definisce un thread "idle", il quale ha la priorità più bassa di tutti i thread definiti;

per default, tale thread non fa altro che portare il processore in idle, limitando il consumo di potenza. Tale codice può essere modificato per eseguire operazioni nelle fasi in cui non avviene la computazione oppure gestione I/O.

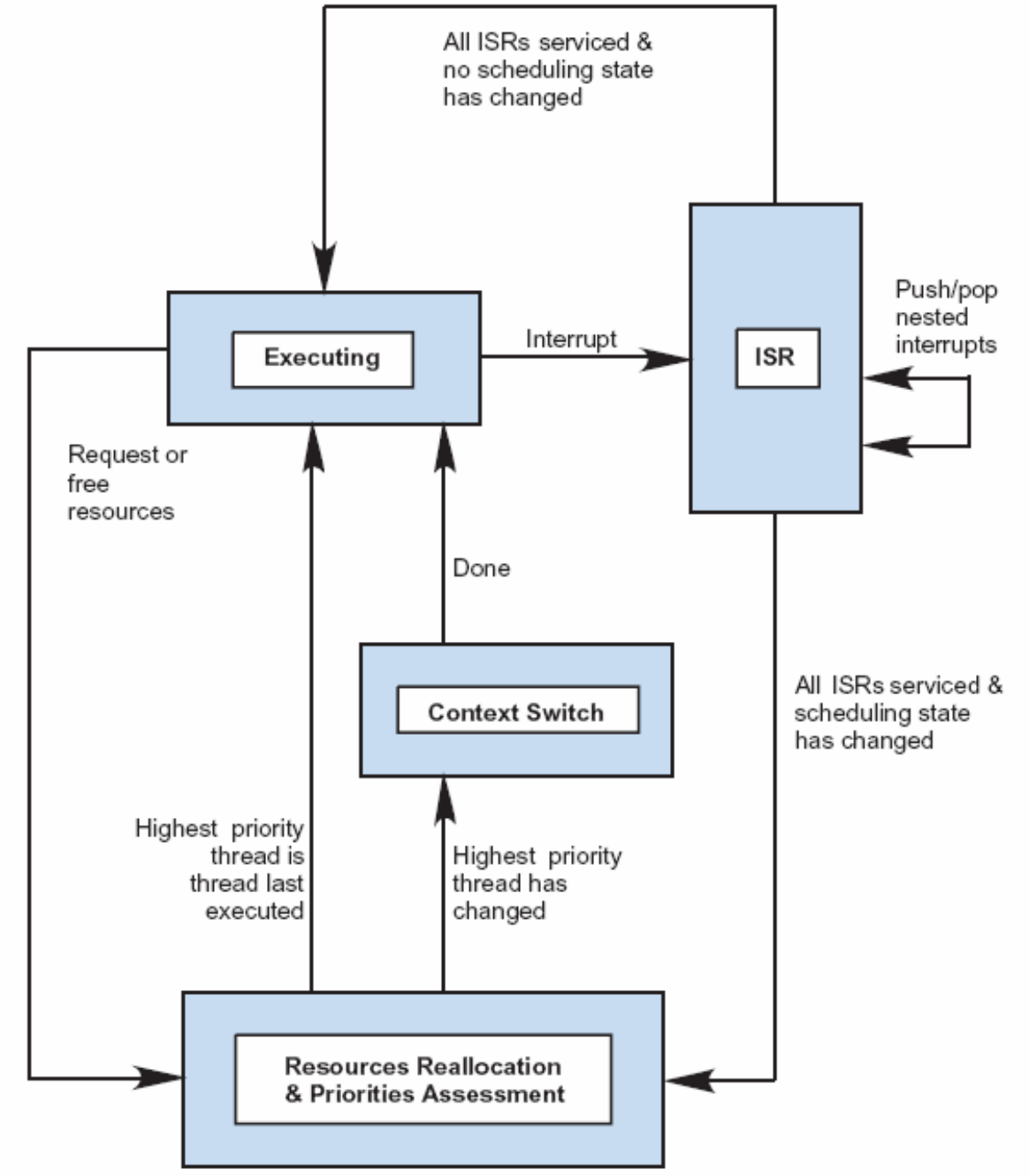

Fig. 3.6 – Diagramma di stato dello scheduler VDK

3.4 I thread

Un thread, nel senso classico del termine, può essere definito come una unità elaborativa indipendente che viene eseguita su un processore. Tale unità possiede un proprio codice e un'area dati che può condividere con altri thread. Lo scheduler decide di volta quale thread eseguire sul processore, preoccupandosi di salvarne lo stato d'esecuzione nel momento in cui viene sottratta la CPU al thread stesso.

La figura 3.7 riporta gli stati in cui può trovarsi un thread e le transizioni di stato che possono avvenire per opera dello scheduler.

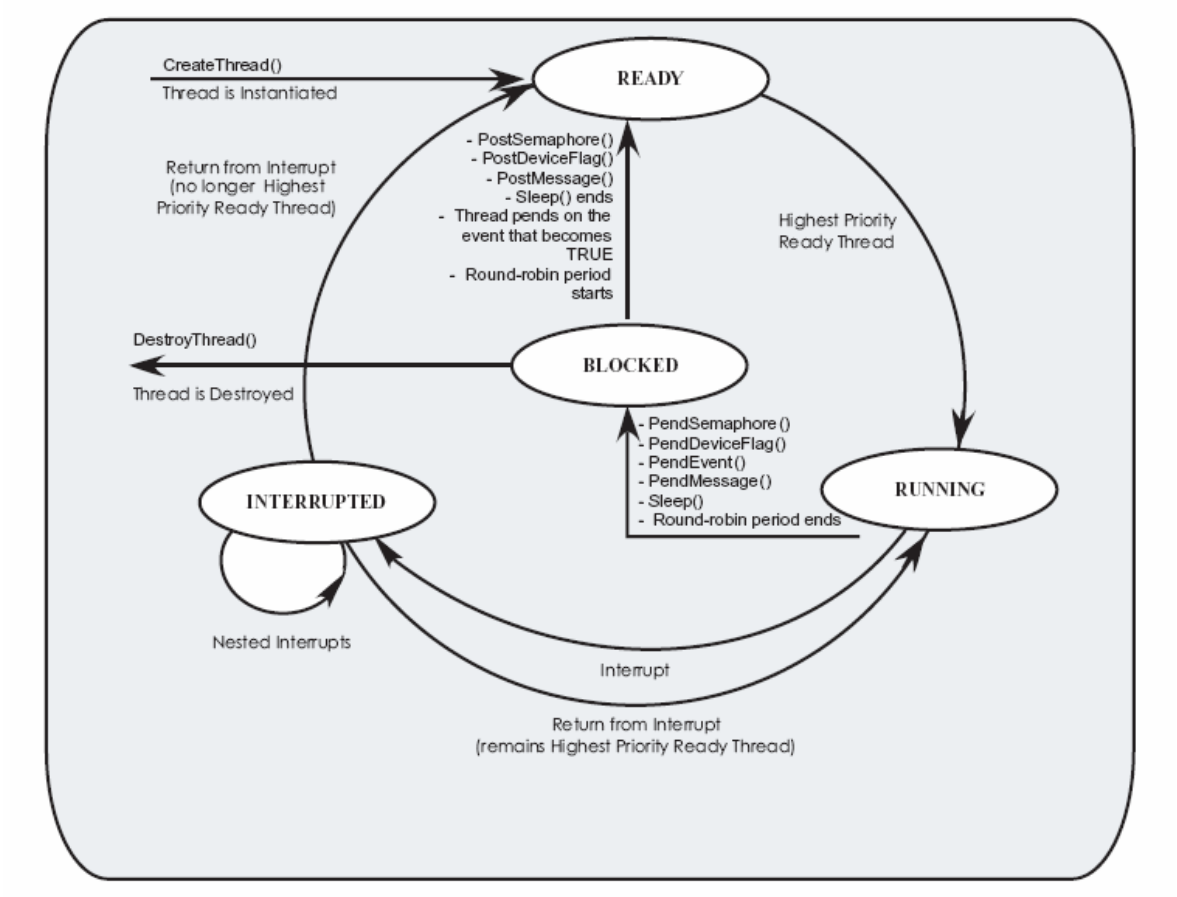

Fig. 3.7 – Diagramma di stato di un thread

Così come avviene nei normali sistemi operativi, un thread può indicare allo scheduler di non “essere schedulato”, ovvero pretendere che non venga sottratta la CPU senza preavviso (cosa che può avvenire a seguito di un interrupt), dichiarando una sezione del proprio codice “non schedulata”. Tale possibilità può tornar utile nel caso di operazioni time-critical.

Come già detto in precedenza, ogni thread può rilasciare volontariamente il possesso della CPU, o per realizzare una politica cooperative (con la primitiva Yield oppure Sleep), oppure perché in attesa di un evento esterno. In tal caso, il suo stato viene commutato in “blocked”, e verrà ripristinato in “ready” solo quando la sua attesa di un evento è terminata. Ad un thread può essere sottratta la CPU anche perché deve essere servita una ISR (Interrupt Service Routine). In ogni caso, lo stato del thread passa a “ready”, e viene posto in una coda FIFO che detiene la lista dei thread che sono (a seconda delle loro priorità) pronti ad impegnare la CPU. La figura 3.8 mostra lo schema di tale coda.

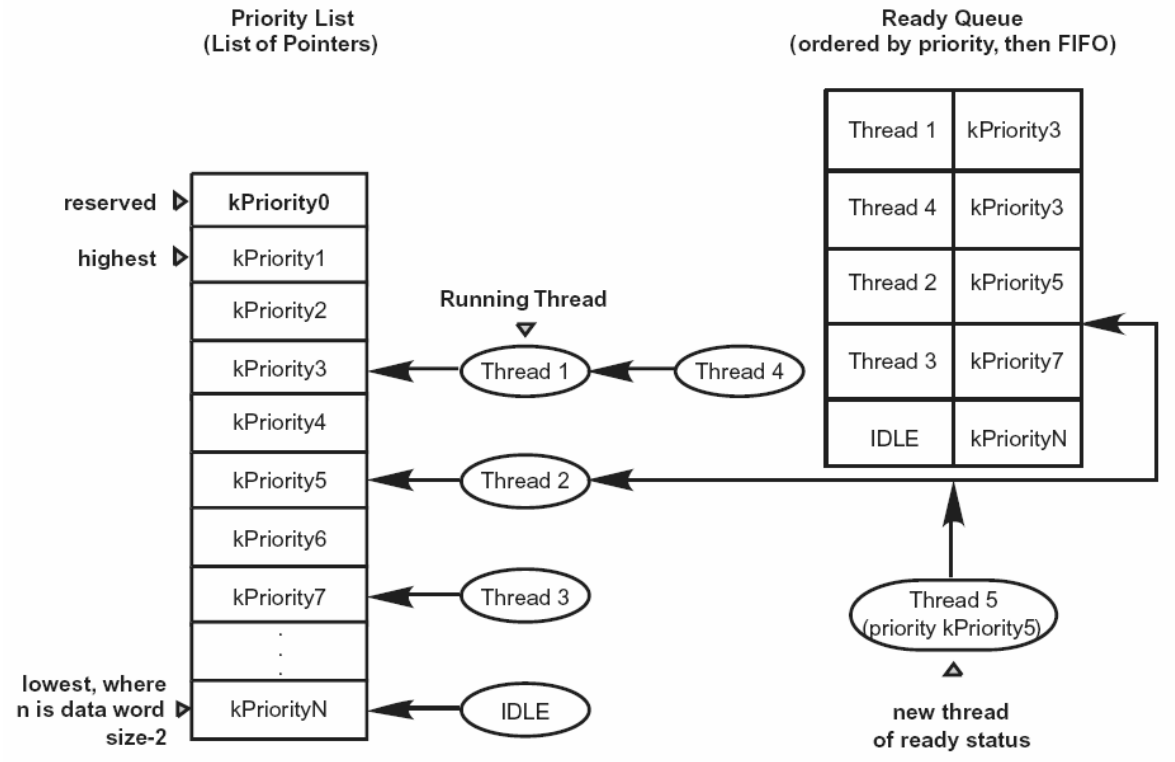

Fig. 3.8 – Coda FIFO dei processi “ready”

3.5 La sincronizzazione

Il kernel VDK fornisce molti metodi per sincronizzare i thread durante la loro esecuzione. Alcuni dei metodi offerti sono presenti in molti sistemi operativi moderni, tra i quali l'uso dei semafori e dei messaggi.

Un metodo invece del tutto nuovo presente nel VDK, e utilizzato nel presente metodo, è la maschera degli eventi, la cui implementazione passa attraverso la definizione dell'oggetto "evento", e la definizione della maschera di bit a partire dalla quale tale evento viene segnalato. Più che riportare definizioni e schemi, è utile fare riferimento ad un esempio. Supponiamo di avere quattro thread, chiamati T0, T1, T2 e T3. T0 si occupa della parte computazionale, mentre T1, T2 e T3 si occupano di realizzare operazioni I/O; T0, prima di poter procedere con il calcolo, ha bisogno di dati che devono essere letti da tre unità, la cui gestione è preposta a ciascuno dei tre thread restanti. All'avvio del sistema, viene lanciato T0, il quale crea un oggetto "evento", crea i thread rimanenti e si mette in attesa sull'evento oggetto creato. Su tale evento è stata definita una maschera di tre bit e allo scheduler è stato indicato, all'atto della creazione dell'oggetto "evento", di segnalare quando tutti i tre bit della maschera sono alti; tale segnalazione significa, per il thread T0, "dati disponibili", quindi possibilità di procedere al calcolo. T1, T2 e T3 (a seconda delle loro priorità) iniziano la gestione di ogni unità I/O, e alzano il bit corrispondente in modo asincrono, ovvero dipendente dalle velocità di I/O delle rispettive unità. Quando tutti i bit saranno alti, lo scheduler commuterà lo stato di T0 da "blocked" a "ready", e quest'ultimo potrà riprendere la sua esecuzione appena uscirà dalla coda FIFO dei processi pronti.

Dall'esempio è possibile capire che tale tecnica si adatta perfettamente sia all'algoritmo proposto, sia a futuri sviluppi, ovvero laddove il thread destinato alla computazione sia legato a più operazioni di I/O scorrelate tra loro. Ovviamente l'oggetto "evento" può essere segnalato o in base ad una AND sui bit della maschera collegata, o in base ad una OR, ed è possibile definire su parti della stessa

maschera eventi diversi, rendendo questo strumento di sincronizzazione altamente flessibile.

La figura 3.9 riporta lo schema di un thread che si pone in attesa della segnalazione di un evento.

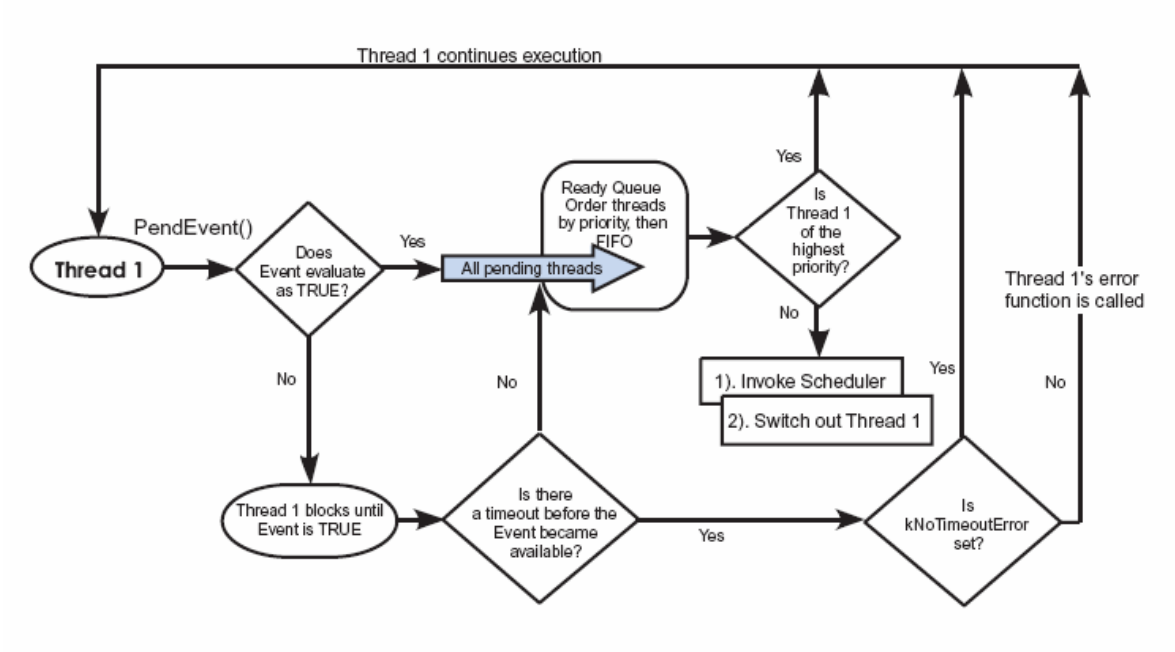

Fig. 3.9 – Attesa di un evento da parte di un thread

Sulla possibile segnalazione di un evento, un thread può attendere indefinitamente, oppure per un tempo prestabilito, chiamato “timeout”, con il quale è possibile notificare al thread che l’evento non si è presentato nell’arco temporale in cui era atteso.

Nel caso in esame, i singoli bit che compongono la maschera dell’evento vengono alzati dalle ISR che si occupano delle rispettive sezioni di I/O. La seguente figura riporta lo schema di principio di tale operazione.

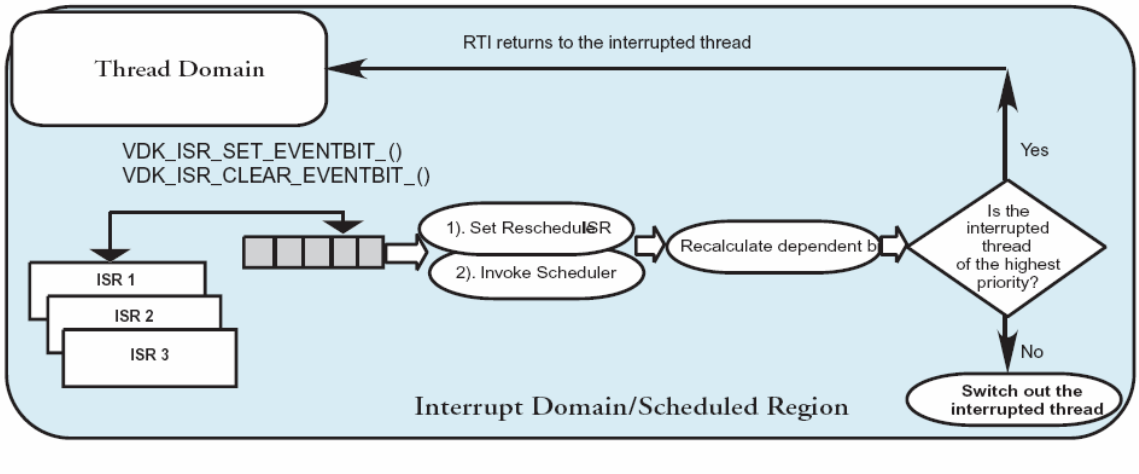

Fig. 3.10 – Modifica della maschera di bit di un evento in una ISR

3.6 Gestione degli interrupt

Il kernel VDK fornisce una interfaccia intuitiva per la scrittura di ISR, gestendo in automatico il vettore delle interruzioni e fornendo primitive idonee per il mascheramento e la disabilitazione/abilitazione degli interrupt. Le ISR vengono trattate come dei thread a priorità più alta di qualsiasi altro thread in esecuzione, quindi la CPU viene sottratta al thread in esecuzione appena si verifica un segnale d'interrupt. Essendo il DSP un processore dedicato ad applicazioni real-time, la durata di una ISR assume un aspetto molto critico, in quanto ISR troppo lunghe potrebbero comportare perdita di accessi I/O. Quindi le ISR devono essere unicamente utilizzate per segnalare il completamento di operazioni I/O, delegando ad opportuni thread la gestione dell'I/O, la quale può essere realizzata con priorità molto bassa.