ERIKA Enterprise Basic Manual

...multithreading in un click!

version: 1.1.0
May 27, 2009

About Evidence S.r.l.

Evidence is a spin-off company of the ReTiS Lab of the Scuola Superiore S. Anna, Pisa, Italy. We are experts in the domain of embedded and real-time systems with a deep knowledge of the design and specification of embedded SW. We keep providing significant advances in the state of the art of real-time analysis and multiprocessor scheduling. Our methodologies and tools aim at bringing innovative solutions for next-generation embedded systems architectures and designs, such as multiprocessor-on-a-chip, reconfigurable hardware, dynamic scheduling and much more!

Contact Info

Address:
Evidence Srl,
Via Carducci 64/A
Località Ghezzano
56010 S.Giuliano Terme
Pisa - Italy
Tel: +39 050 991 1122, +39 050 991 1224
Fax: +39 050 991 0812, +39 050 991 0855

For more information on Evidence Products, please send an e-mail to the following address: info@evidence.eu.com. Other informations about the Evidence product line can be found at the Evidence web site at: http://www.evidence.eu.com.

Contents

1 Introduzione all'API minimale

1.1 Erika Enterprise

Evidence presenta il sistema operativo real-time Erika Enterprise un sistema operativo minimale per microcontrollori che fornisce un semplice e compatto ambiente di esecuzione multithreading, con il supporto di algoritmi di scheduling real-time avanzati e il supporto allo stack sharing.

Il kernel Erika Enterprise e' stato sviluppato con l'intenzione di fornire un insieme minimale di primitive che possono essere utilizzate per implementare un sistema di ambiente multithreading. Le API minimali di Erika Enterprise costituiscono un insieme ridotto delle API OSEK/VDX, che forniscono il supporto per funzionalita' quali l'attivazione di thread, la mutua esclusione, gli allarmi e i counting semaphores.

Il consorzio OSEK/VDX ha messo a punto il linguaggio di specifica OIL (OSEK Implementation Language) come standard per la configurazione delle applicazioni. Esso viene utilizzato per la definizione statica di una serie di componenti e funzionalita' da istanziare in una applicazione. Erika Enterprise supporta appieno il linguaggio OIL per la configurazione di applicazioni real-time.

Per far fronte alla complessita' che deriva dalla manipolazione di un file di configurazione scritto in linguaggio OIL, Evidence ha sviluppato un tool di configurazione e profiling apposito: RT-Druid˙RT-Druid permette di configurare una applicazione in ogni sua componente, impostando i parametri in modo semplice attraverso una interfaccia visuale che automaticamente genera il codice di configurazione in linguaggio OIL.

Il tipico flusso di progetto e design di una applicazione include la definizione di un file di configurazione in linguaggio OIL, che definisce gli oggetti, i componenti che sono utilizzati all'interno dell'applicazione real-time. In questa fase, RT-Druid aiuta lo sviluppatore ad impostare i parametri dei singoli oggetti e provvede alla generazione del relativo file di configurazione in linguaggio OIL, nonche' alla generazione del codice sorgente e dei makefiles richiesti per compilare l'applicazione. L'ultimo step prevede l'effettiva compilazione del sorgente e la generazione del codice eseguibile sulla macchina target.

Alcune delle features fornite da Erika Enterprise allo sviluppatore, tipiche di un sistema real-time, sono le seguenti:

- Supporto per il multitasking sia di tipo preemptive che non-preemptive;
- Supporto per algoritmi di schedulazione a priorita' fisse;
- Supporto per tecniche di stack sharing e di modelli di task one-shot al fine di ridurre l'utilizzo di stack da parte del programma;
- Supporto per risorse condivise;

- Supporto per l'attivazione periodica di task per mezzo degli Alarms;

Lo scopo di questo documento e' quello di descrivere in dettaglio le funzionalita' e la API di Erika EnterprisePresso il sito web di Evidence e' disponibile ulteriore documentazione riguardo al porting di Erika Enterprise per varie architetture e microcontrollori.

2 API reference

2.1 Introduzione

Il sistema operativo Erika Enterprise fornisce una interfaccia semplice per l'esecuzione di applicazioni concorrenti su sistemi a singolo processore.

L'intefraccia proposta e' adatta per piccoli microcontrollori da 8 a 32 bit. L'architettura prevede che i task possano essere eseguiti concorrentemente, con la possibilita' di scambiare informazioni attraverso il meccanismo della memoria condivisa.

I task, in Erika Enterprise, sono schedulati secondo priorita' fisse, e viene utilizzato il protocollo Immediate Priority Ceiling per la condivisione di risorse.

Gli interrupt hanno sempre priorita' maggiore rispetto ai task applicativi, in modo da favorire l'esecuzione le attivita' richieste dalle periferiche. Gli interrupt, o Interrupt Service Routines, possono essere di due tipi: *ISR Type 1* and *ISR Type 2*. La Sezione 2.7 e' dedicata esplicitamente alla descrizione della gestione di interruzioni in Erika Enterprise.

2.1.1 Classi di conformance

La API minimale di Erika Enterprise non supporta diversi tipi di classi di conformance. La sola configurazione supportata dal sistema e' FP (Fixed Priority), che ha un funzionamento simile a quello delle classi di conformance BCC1 e BCC2 di Erika Enterprise, in base al fatto che il kernel sia configurato in monostack oppure in multistack.

La classe di conformance FP e' supportata dal multithreading a priorita' fissa, con uno o piu' task per ciascuna priorita' e con una o piu' attivazione pendente per ciascun task.

2.1.2 Primitive disponibili

Erika Enterprise fornisce un insieme di primitive che possono essere chiamate in differenti contesti. La lista completa delle primitive e' presentata in Tabella 2.1, insieme alle indicazioni dei contesti nei quali e' possibile chiamare ciascuna funzione.

Service	Background Task	Task	ISR1	ISR2	Alarm Callback
ActivateTask	√	√		√	
Schedule		√			
GetResource		√		√	
ReleaseResource		√		√	
CounterTick		√		√	
GetAlarm	√	√		√	
SetRelAlarm	√	√		√	
SetAbsAlarm	√	√		√	
CancelAlarm	√	√		√	
InitSem	√	√		√	
WaitSem		√			
TryWaitSem	√	√		√	
PostSem	√	√		√	
GetValueSem	√	√		√	

Table 2.1: Questa tabella riporta la lista degli environment nei quali e' possibile richiamare le varie primitive.

2.2 Costanti

Di seguito e' riportata una lista delle costanti fornite in Erika Enterprise e disponibili dello sviluppatore per la scrittura delle applicazioni.

2.2.1 INVALID_TASK

Description

Questa costante rappresenta l'ID di task invalido.

2.2.2 EE_MAX_NACT

Description

Questa costante rappresenta il massimo numero di attivazioni pendenti che possono essere gestite per ciascun task. Il valore di default per questa costante e' il massimo valore disponibile per un unsigned integer dell'architettura specifica.

2.2.3 RES_SCHEDULER

Description

Questo e' l'ID della risorsa `RES_SCHEDULER`.

La risorsa e' disponibile solo quando `USE_RESSCHEDULER` e' impostato al valore `TRUE` all'interno del file di configurazione OIL. La costante `RES_SCHEDULER`, che identifica il ceiling di sistema, dipende dai task che sono stati creati creati, ed e' calcolata quando RT-Druid genera il codice di configurazione per Erika Enterprise.

2.2.4 Stato di un task

Description

Di seguito e' riportata la lista degli stati che un task puo' assumere durante l'esecuzione del sistema:

```
#define EE_READY      1
#define EE_STACKED    2
#define EE_BLOCKED    4
#define EE_WASSTACKED 8
```

GLi stati di un task in Erika Enterprise sono tipicamente non visibili dal programmatore, in quanto sono altamente dipendenti dalla particolare configurazione del sistema. In particolare, quando viene usata una configurazione monostack, gli stati dei task sono rimossi dalla memoria per risparmiare RAM. Lo stato `EE_READY` caratterizza un task il quale e' pronto per l'esecuzione ma per il quale non e' ancora stato allocato lo stack. Lo stato `EE_STACKED` si riferisce a un task che puo' essere o il task in esecuzione oppure

il task che ha subito preemption sullo stack. Lo stato `EE_BLOCKED` identifica un task che e' stato eseguito e che attualmente e' bloccato da una primitiva di sincronizzazione (per esempio, dalla primitiva `WaitSem`). Un flag addizionale, `EE_WASSTACKED`, e' definito per utilizzi interni, allo scopo di mappare un task ready che e' stato risvegliato da una primitiva di sincronizzazione ma che si trova ancora nella coda dei task ready, in attesa di essere eseguito.

2.3 Tipi di dato

Questa sezione contiene la descrizione dei tipi dei dati utilizzati da Erika Enterprise. Quando la dimensione di un determinato tipo di dato e' indicata come pari alla dimensione dei registri macchina, si intende che tale tipo ha la dimensione dei registri general purpose della specifica CPU.

2.3.1 AlarmType

Description

Questo tipo di dati (signed) e' utilizzato per memorizzare gli ID degli allarmi. Ha la dimensione di un registro.

2.3.2 CounterType

Description

Questo tipo di dato (signed) e' usato per memorizzare gli ID dei contatori. Ha la dimensione di un registro.

2.3.3 ResourceType

Description

Questo tipo di dato (unsigned) e' utilizzato per memorizzare l'ID delle risorse. Ha la dimensione del registro.

2.3.4 SemType

Description

Questo tipo di dato e' una struttura che serve a memorizzare le informazioni relative a un semaforo di conteggio.

2.3.5 SemRefType

Description

Questo e' un puntatore a `SemType`.

2.3.6 TaskType

Description

Questo tipo di dati (signed) e' usato per memorizzare il Task ID, e ha la dimensione di un registro.

2.3.7 TickType

Description

Questo tipo di dato (unsigned) e' usato per memorizzare il valore dei Counter Ticks, e ha la dimensione di un registro.

2.3.8 TickRefType

Description

Questo e' un puntatore a un tipo di dati `TickType`.

2.4 Definizione degli oggetti

Le seguenti macro devono essere usate quando viene definito un Task.

2.4.1 TASK

Synopsis

```
TASK(Funcname) {...}
```

Description

La keyword TASK deve essere usata quando viene dichiarata una funzione TASK.

Conformance

BCC1, BCC2, ECC1, ECC2

2.5 Primitive per la gestione di Task

La API minimale di Erika Enterprise supporta la definizione di task che sono assimilabili ai Basic Task dello standard OSEK/VDX.

I task in Erika Enterprise sono tipicamente implementati nella forma di normali funzioni C, che eseguono il relativo codice e poi terminano. Ciascuna di tali esecuzioni e' chiamata *Istanza del task*. Al termine di un task basic, lo stack relativo al task viene liberato. I task Erika Enterprise tipicamente non si bloccano, ed in questo modo permettono allo sviluppatore di implementare la condivisione dello stack tra diversi task. La condivisione dello stack permette allo sviluppatore di ridurre l'utilizzo di memoria RAM da parte del programma.

Il supporto per primitive bloccanti come semafori di conteggio e' disponibile anche se il kernel e' configurato in modalita' multistack, nel caso di task ai quali e' stato assegnato uno stack privato. I task che utilizzano primitive bloccanti sono tipicamente implementati nella forma di task che senza terminazione, nei quali ciascuna istanza del task termina con una istruzione di sincronizzazione costituita, per esempio, da una chiamata alla wait su semaforo.

La politica di scheduling utilizzata in Erika Enterprise e' la cosiddetta Fixed Priority Scheduling con Immediate Priority Ceiling and Preemption Thresholds. Questo permette di implementare la seguente tipologia di task:

Full Preemptive Task Un task Full Preemptive e' un task che puo' essere interrotto in qualsiasi istante dai task a priorita' piu' elevata.

Non Preemptive Task Un task di tipo Non Preemptive e' assimilabile ad un task di tipo Full Preemptive che esegue per tutta la sua durata bloccando una risorsa e avendo il relativo ceiling pari alla massima priorita' del sistema. Questo fatto fa' si che il task Non Preemptive non possa essere interrotto da nessun altro task, ma solo da eventuali interrupt.

Mixed Preemptive Task Un task di tipo Mixed Preemptive e' un task che viene eseguito ad una priorita' pari ad una priorita' maggiore della priorita' alla quale il task viene inserito nella ready queue (Questa tecnica e' chiamata *Preemption Thresholds*). In tal modo il numero di preemption puo' essere ridotto considerevolmente, cosa che permette di risparmiare una gran quantita' di memoria RAM.

I task vengono attivati per mezzo della primitiva `ActivateTask`. Attivare un task significa che il task che viene attivato puo' essere schedulato per l'esecuzione, e che quindi puo' eseguire una istanza. Una attivazione di task mentre il task e' gia' in esecuzione oppure mentre si trova nella ready queue, ovvero in attesa di essere eseguito, viene salvata come attivazione pendente. Il numero massimo di attivazioni pendenti e' dipendente dall'implementazione.

I task della API minimale di in Erika Enterprise sono leggermente diversi dai task utilizzati nelle conformance classes simili a OSEK. Le principali differenze sono le seguenti:

- La API minimale non supporta le primitive `TerminateTask` e `ChainTask`.

- Il numero di attivazioni pendenti non necessita di essere specificato all'interno di un file OIL, come invece avviene per le classi di conformance BCC2 and ECC2 in Erika Enterprise.

2.5.1 ActivateTask

Synopsis

```
void ActivateTask(TaskType TaskID);
```

Description

Questa primitiva attiva il task identificato da `TaskID`. Dopo l'attivazione il task puo' diventare il task running se esso ha la priorita' piu' elevata rispetto agli altri task presenti nella ready queue.

Una volta attivato, il task esegue una istanza, che inizia con la prima istruzione della funzione che implementa il codice del task. Se il task viene attivato quando una sua istanza e' gia' in esecuzione, o mentre si trova nella ready queue, l'attivazione viene salvata come attivazione pendente, che verra' gestita successivamente. Se la nuova attivazione pendente fa si che il numero di attivazioni pendenti diviene superiore al numero massimo impostato in fase di implementazione, la richiesta di attivazione viene ignorata.

La funzione puo' essere chiamata dal Background task, che tipicamente e' costituito dalla funzione `main()`.

Parameters

- `TaskID` Task reference.

Return Values

- `void` La funzione non ritorna alcun codice di errore.

Conformance

FP

2.5.2 Schedule

Synopsis

```
void Schedule(void)
```

Description

Questa primitiva puo' essere utilizzata come punto di ri-schedulazione per i task che utilizzano le Preemption Thresholds e per i task di tipo Non Preemptive.

Quando questa primitiva viene invocata, un task che utilizzi la preemption threshold imposta la propria priorita' al valore piu' basso usato per l'inserimento nella ready queue. A questo punto, il sistema controlla se esistono task a priorita' piu' elevata che hanno diritto di fare preemption, nel qual caso la preemption viene effettuata. Nel momento in cui la primitiva ritorna, i task che utilizzano la preemption threshold ritorneranno ad acquisire la loro priorita' di threshold.

La primitiva non ha effetto se il task chiamante non e' un task di tipo Non Preemptive ne' un task che usa la preemption thresholds.

Return Values

- `void` La funzione non ritorna alcun valore.

Conformance

FP

2.6 Primitive di gestione delle risorse

Il termine risorsa viene utilizzato da Erika Enterprise per identificare i semafori binari che servono ad implementare le sezioni critiche condivise.

Le risorse sono implementate utilizzando il protocollo Immediate Priority Ceiling. Una risorsa viene bloccata (locked) utilizzando la primitiva `GetResource`, e sbloccata utilizzando `ReleaseResource`.

Esiste anche il supporto per una risorsa speciale chiamata `RES_SCHEDULER`. La risorsa `RES_SCHEDULER` ha un ceiling pari alla piu' elevata priorita' nel sistema. Un task che blocca la risorsa `RES_SCHEDULER` diviene non-preemptabile. Se necessaria, la risorsa `RES_SCHEDULER` deve essere impostata all'interno del file di configurazione OIL.

2.6.1 GetResource

Synopsis

```
void GetResource (ResourceType ResID)
```

Description

Questa primitiva puo' essere usata per implementare una sezione critica controllata dalla risorsa indicata da `ResID`. La sezione critica viene quindi terminata da una chiamata a `ReleaseResource`.

E' permesso l'annidamento di sezioni critiche controllate da diverse risorse.

La chiamata a `Schedule` non e' permessa all'interno di una sezione critica.

Il servizion puo' essere chiamato solo dal livello task.

Parameters

- `ResID` Riferimento a risorsa

Return Values

- `void` La funzione non ritorna alcun valore.

Conformance

FP

2.6.2 ReleaseResource

Synopsis

```
void ReleaseResource (ResourceType ResID)
```

Description

`ReleaseResource` e' utilizzata per rilasciare precedentemente bloccata da una chiamata `GetResource`, e che quindi chiude la sezione critica.

Per informazioni su sezioni critiche annidate, si veda `GetResource`. Il servizio puo' essere chiamato dal solo livello dei task.

Parameters

- `ResID` Identificatore di risorsa

Return Values

- `void` La funzione non ritorna alcun errore.

Conformance

FP

2.7 Primitive di gestione degli interrupt

Erika Enterprise supporta la gestione degli interrupt, che vengono modellizzati considerando il tipico design dei controllori degli interrupt dei microcontrollori, che utilizzano un assegnamento prioritario delle sorgenti di interrupt.

Per soddisfare i requisiti nella gestione di routine di interrupt veloci, Erika Enterprise supporta la definizione di cosiddetti fast interrupts handlers chiamati *ISR Type 1*. Questi permettono di gestire interruzioni nel modo piu' veloce possibile, ma per contro limitano la possibilita' di chiamare servizi di sistema all'interno del codice del fast handler.

Per questo motivo e' prevista la possibilita' di utilizzare i cosiddettid *ISR Type 2*, che possono essere usati - per esempio - per l'implementazione di timer hardware, dal momento che permettono la chiamata di primitive di sistema. Per contro, essi sono piu' lenti degli ISR Type 1 a causa dell'overhead introdotto dal sistema per l'implementazione della preemption.

Molti dettagli implementativi relativi alla gestione degli interrupt dipende fortemente dalla particolare architettura del microcontrollore sul quale Erika Enterprise viene utilizzato. Si faccia riferimento alla documentazione relativa al porting di Erika Enterprise su una specifica architettura per maggiori dettagli.

2.8 Primitive Counter e Alarms

Erika Enterprise supporta un meccanismo di notifica basato su *Counter*s e *Alarm*s.

Un Counter e' semplicemente un valore integer che viene incrementato di 1 "Tick" utilizzando la primitiva `CounterTick`.

Un Alarm e' una notifica che viene collegata ad uno specifico Counter. Il collegamento tra il Counter e l'Alarm viene specificato al momento della compilazione all'interno del file di configurazione OIL.

Un Alarm puo' essere impostato per essere generato ad uno specifico valore di tick utilizzando le primitive `SetRelAlarm` e `SetAbsAlarm`. Gli Alarms possono essere impostati per essere riattivati periodicamente. Gli Alarms possono essere cancellati per mezzo della primitiva `CancelAlarm`.

Quando un Alarm viene attivato, ha luogo una notifica. La notifica puo' essere impostata per effettuare una delle seguenti azioni:

Task activation. In questo caso viene attivato un task quando il corrispondente Alarm viene generato.

Alarm callback. In questo caso viene chiamata una alarm callback (definita come `void f(void)`).

Le notifiche vengono eseguite all'interno della funzione `CounterTick`. E' compito dello sviluppatore posizionare il contatore in un punto signiticativo del programma (per esempio, all'interno di una routine di timer interrupt).

I Counters, gli Alarms e le relative azioni di notifica sono specificate all'interno del file di configurazione OIL.

Warning: Al momento non c'e' il supporto per Timers che sono automaticamente aggiornati dal sistema (per esempio, contatori che sono collegati ai timer hardware). Tutti i contatori devono essere definiti all'interno del file OIL di configurazione, e il programmatore deve esplicitamente richiamare la funzione `CounterTick` per incrementarli.

2.8.1 CounterTick

Synopsis

```
void CounterTick(CounterType c)
```

Description

Questa funzione riceve come parametro un identificatore di Counter e incrementa il relativo contatore di una unita'. Questa funzione viene tipicamente richiamata all'interno di una routine ISR type 2 oppure all'interno di un task.

La funzione implementa anche la notifica degli Alarms scaduti. Tale implementazione dipende dalla configurazione dell'allarme, e puo' essere una delle seguenti:

- una callback function collegata all'Alarm;
- l'attivazione di un task;

La funzione e' atomica, e non avviene alcun rescheduling dopo l'esecuzione di questa funzione. Quando viene chiamata dal livello di un task, per forzare il rescheduling l'applicazione deve chiamare la funzione `Schedule` dopo la chiamata di questa funzione. Quando la chiamata proviene dal livello di un ISR type 2 level, il rescheduling viene automaticamente effettuato alla fine della routine di interrupt.

Parameters

- `c` Il contatore che si vuole incrementare.

Return Values

- `void` La funzione non ritorna alcun errore.

Conformance

FP

2.8.2 GetAlarm

Synopsis

```
void GetAlarm (AlarmType AlarmID, TickRefType Tick)
```

Description

Il servizio di sistema GetAlarm ritorna il valore dei ticks che mancano allo scadere dell'allarme `AlarmID`. `AlarmID` *deve* essere un identificatore valido in uso nel sistema. La chiamata e' disponibile al livello di task, ISR, e in diverse routine di hook.

Parameters

- `AlarmID` Identificatore dell'Alarm
- `Tick` (out) Valore dei ticks mancanti allo scadere dell'allarme

Return Values

- `void` La funzione non ritorna alcun errore.

Conformance

FP

2.8.3 SetRelAlarm

Synopsis

```
void SetRelAlarm (AlarmType AlarmID, TickType increment, TickType cycle)
```

Description

Dopo che sono trascorsi `increment` ticks, viene eseguita la notifica `AlarmID`.

Se il valore relativo di `increment` e' molto piccolo, l'allarme potrebbe scadere, e la notifica puo' essere eseguita prima che il servizio ritorni. Se il valore di `cycle` e' diverso da zero, l'allarme viene loggato nuovamente immediatamente dopo che il relativo valore di `cycle` e' scaduto.

L'allarme `AlarmID` non deve essere rgia' in uso al sistema: prima di cambiare il valore di un allarme che e' gia' in uso, l'allarme deve essere cancellato. La chiamata e' valida dai livelli task e ISR.

Parameters

- `AlarmID` Riferimento all'allarme
- `increment` Valore relativo in ticks che rappresenta l'offset rispetto all'istante corrente della primo evento di expiration dell'allarme.
- `cycle` Periodo dell'allarme in caso di allarme ciclico. Nel caso in cui si tratti di un allarme singolo, questo parametro deve essere impostato a 0.

Return Values

- `void` La funzione non ritorna alcun errore.

Conformance

FP

2.8.4 SetAbsAlarm

Synopsis

```
void SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)
```

Description

La primitiva occupa l'elemento `AlarmID` del relativo all'allarme. Quando sono raggiunti `start` ticks, la notifica identificata da `AlarmID` viene eseguita.

Se il valore assoluto di `start` e' prossimo a quello del valore corrente del contatore, l'allarme potrebbe scadere, e il task potrebbe divenire ready o la callback associata all'allarme potrebbe essere richiamata prima che questo servizio di sistema sia terminato.

Se il valore assoluto di `start` e' stato gia' raggiunto prima della chiamata di sistema, l'allarme dovrebbe solo scadere quando il valore assoluto di `start` viene nuovamente raggiunto, per esempio quando avviene il successivo overrun del contatore.

Se `cycle` e' diverso da zero, l'elemento di allarme viene loggato immediatamente dopo che il relativo valore di `cycle` e' trascorso.

L'allarme `AlarmID` non dovrebbe esser gia' in uso nel sistema: prima di variare il valore di un allarme che e' gia' in uso, l'allarme dovrebbe essere cancellato. La chiamata puo' essere effettuata al livello di task o in un ISR.

Parameters

- `AlarmID` Riferimento all'allarme.
- `start` Valore assoluto in ticks che rappresenta il tempo al quale si desidera far scadere l'allarme per la prima volta.
- `cycle` Periodo dell'allarme in caso di allarme ciclico. Nel caso in cui si tratti di un allarme singolo, questo parametro deve essere impostato a 0.

Return Values

- `void` La funzione non ritorna alcun errore.

Conformance

FP

2.8.5 CancelAlarm

Synopsis

```
void CancelAlarm (AlarmType AlarmID)
```

Description

La primitiva cancella l'allarme identificato da `AlarmID`. La chiamata e' permessa al livello di task o di ISR.

Parameters

- `AlarmID` Riferimento all'allarme

Return Values

- `void` La funzione non ritorna alcun errore.

Conformance

FP

2.9 Counting Semaphores

Questa sezione descrive in dettaglio le primitive fornite da Erika Enterprise per il supporto dei counting semaphores come metodo per implementare la mutua esclusione e la sincronizzazione tra i task.

Un counting semaphore e' una astrazione tipica dei RTOS nella quale un contatore intero viene associato ad una coda di bloccaggio. In pratica, su di un semaforo sono possibili due operazioni: la *wait*, che consiste nel decremento del contatore se il contatore stesso ha un valore maggiore di 0, e che si traduce in un bloccaggio del task in esecuzione se il valore del contatore e' 0, e la *post*, che effettua un incremento del contatore se non ci sono task bloccati, oppure si traduce in uno sbloccaggio del task bloccato altrimenti.

In Erika Enterprise l'utilizzo dei counting semaphores avviene mediante una semplice interfaccia che copre alcune funzionalita' di base, come:

- L'inizializzazione del semaforo (`InitSem`);
- La wait sul semaforo in modo bloccante (`WaitSem`) o in modo non-bloccante (`TryWaitSem`;
- Il post di un semaforo (`PostSem`);
- La restituzione del valore del contatore relativo al semaforo (`GetValueSem`).

Dal momento che un semaforo puo' provocare il bloccaggio del task in esecuzione, la primitiva `WaitSem` dovrebbe essere richiamata solo se il task chiamante ha uno stack dedicato, che significa che Erika Enterprise e' stato configurato in modalita' multistack).

I semafori possono inoltre essere allocati staticamente come variabili globali. Questa opportunita' permette di evitare di inizializzare esplicitamente il semaforo chiamando `InitSem`.

La definizione di un semaforo non viene listata all'interno del file OIL di configurazione; le primitive di gestione dei semafori ricevono come parametro il puntatore al descrittore del semaforo.

> **Warning:** I counting semaphores *NON* evitano il problema dell'Inversione di Priorita'. Nel caso in cui si voglia evitare l'inversione di priorita' devono essere utilizzare le Resources (vedi Sezione 2.6).

2.9.1 STATICSEM

Synopsis

```
SemType s = STATICSEM(value);
```

Description

Questa macro puo' essere utilizzata per inizializzare staticamente un semaforo. La macro deve essere usata all'interno della definizione globale di una variabile che rappresenta il semaforo. Riceve come parametro il valore iniziale da assegnare al semaforo.

Parameters

- `value` Il valore del contatore usato per inizializzare il semaforo.

Return Values

- `none` Questa funzione e' una macro e non ritorna alcun errore.

Conformance

FP

2.9.2 InitSem

Synopsis

```
void InitSem(SemType s, int value);
```

Description

Questa macro puo' essere utilizzata per inizializzare un semaforo a runtime. Essa riceve come parametro il valore iniziale da assegnare al semaforo.

Parameters

- `s` Il semaforo da inizializzare.
- `value` Il valore da assegnare al contatore associato al semaforo.

Return Values

- `void` Questa funzione e' una macro e non ritorna alcun errore.

Conformance

FP

2.9.3 WaitSem

Synopsis

```
void WaitSem(SemRefType s);
```

Description

Se il contatore associato al semaforo e' maggiore di 0, allora il contatore viene decrementato di 1. Se il contatore vale 0, allora il task chiamante (running) viene bloccato. Per il corretto funzionamento del sistema, deve essere allocato uno stack separato per ciascun task, in quanto il task puo' essere bloccato.

Parameters

- `s` Il riferimento al semaforo.

Return Values

- `void` La funzionee non ritorna alcun errore.

Conformance

FP

2.9.4 TryWaitSem

Synopsis

```
int TryWaitSem(SemRefType s);
```

Description

Questa e' la versione non bloccante di `SemWait`. Se il contatore del semaforo e' maggiore di 0, allora esso viene decrementato di 1 e la primitiva ritorna 0. Se il contatore ha valore pari a 0, allora il contatore viene decrementato e la primitiva ritorna 1.

Parameters

- `s` Il semaforo utilizzato dalla primitiva.

Return Values

- `int` 0 se il contatore e' stato decrementato, altrimenti 1.

Conformance

FP

2.9.5 PostSem

Synopsis

```
void PostSem(SemRefType s);
```

Description

Questa primitiva sblocca un task eventualmente bloccato dal semaforo. Se non ci sono task bloccati dal semaforo, allora il contatore del semaforo viene incrementato di una unita'.

Parameters

- `s` Il semaforo che deve essere utilizzato dalla primitiva.

Return Values

- `void` La funzionee non ritorna alcun errore.

Conformance

FP

2.9.6 GetValueSem

Synopsis

```
int GetValueSem(SemRefType s);
```

Description

Se non ci sono task bloccati dal semaforo, questa funzione restituisce `-1`; altrimenti, la primitiva ritorna il valore del contatore associato al semaforo.

Parameters

- `s` Il semaforo che deve essere utilizzato dalla primitiva.

Return Values

- `int -1` se non ci sono task bloccati sul semaforo, altrimenti il valore del contatore associato al semaforo.

Conformance

FP

2.10 Startup del sistema

Erika Enterprise non richiede una procedura di startup specifica. In particolare, il kernel diviene immediatamente attivo dopo la prima istruzione della funzione `main`.

Una tipica applicazione viene strutturata con una routine di inizializzazione, che dipende dall'applicazione stessa, posta all'interno della funzione `main`. A questo punto, i task vengono attivati con una chiamata a `ActivateTask`. Successivamente, si raggiunge il loop infinito posto alla fine della funzione `main`. La funzione `main` diviene cosi' il task di background.

3 Cronologia

Versione	Commento
1.0.0	Prima versione ufficiale di questo documento.
1.1.0	Typos. Cambio di nome per Erika Enterprise.

Index