1 Hook ed ErrorHook

1.1 OSEK/VDX

La specifica OSEK/VDX definisce le *hook routines* come un mezzo per consentire azioni definite dall'utente riferite a processi interni del sistema operativo. Queste hanno alcune caratteristiche specifiche:

- sono chiamate dal sistema operativo, in un contesto speciale dipendente dal sistema operativo;
- con priorità maggiore di tutti i *task*;
- non possono essere interrotte da ISR2;
- con un'interfaccia standardizzata, ma con funzionalità non standardizzate (perchè definite dall'utente).

Esse possono essere utilizzate allo *startup* del sistema (*StartupHook*), al suo *shutdown (ShutdownHook)*, per *debugging (PreTaskHook/PostTaskHook/IdleHook)* o per *error handling (ErrorHook)*.

La maggior parte dei servizi del sistema operativo non è consentita per le *hook routines*. Questa restrizione è necessaria per ridurre la complessità del sistema.

La *routine ErrorHook* viene chiamata se un servizio del sistema operativo restituisce un valore di *StatusType* diverso da 'E_OK'. Lo scopo dell'*ErrorHook* è quindi di trattare in modo centralizzato alcuni errori del sistema operativo relativi all'esecuzione del sistema.

1.2 ERIKA³

Nello specifico dell'attivazione di un *task*, i possibili valori di *StatusType* sono 'E_OK', 'E_OS_LIMIT', 'E_OS_ID' (solo per *task* di tipo *extended*). Lo stato di interesse è 'E_OS_LIMIT', che identifica troppe attivazioni pendenti di un *task*. È importante notare come questo stato sia possibile solamente per primitive di tipo *ActivateTask* e *ChainTask* e attivazioni del *task* allo scattare di un allarme.

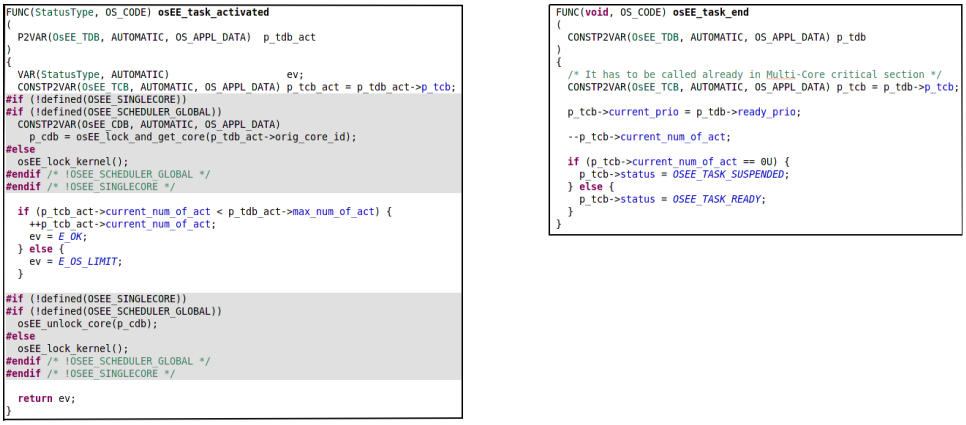
Un allarme (*Alarm*) è un meccanismo di notifica collegato a uno specifico *Counter* che, una volta raggiunto il *tick* stabilito, definito nel *file* OIL, può attivare un *task*, impostare un evento o invocare una *callback*. L'esecuzione della notifica relativa a un allarme avviene all'interno della funzione *IncrementCounter* che incrementa il *tick* del contatore collegato a quell'allarme. Questa funzione è atomica e al suo termine avviene il *rescheduling*, se questa è chiamata a livello di *task*, o, se chiamata da un ISR, il *rescheduling* avviene al termine dell'ISR nidificato più esterno.

Quando si cerca di attivare un *task*, sia per la notifica di un allarme o una primitiva del sistema invocata dall'utente, il *kernel* di ERIKA³ effettua dei controlli sullo stato dei *task* del sistema e, in particolare, sullo stato della coda dei pronti. Se il *task* interessato dall'attivazione non ha *job* nella coda dei pronti (quindi inevitabilmente il numero di attivazioni di quel *task* è pari a 0) lo aggiunge a questa, mentre se ci sono già suoi *job* in attesa (in stato *'ready'* o *'waiting'*) o in esecuzione (stato *'running'*), viene valutato il numero di attivazioni pendenti di quel *task* in quel momento e confrontato con il numero di attivazioni massime impostato nel *file* OIL. Il valore di attivazioni pendenti a *runtime* viene quindi modificato in due momenti:

- All'attivazione del *task*, sia questa causata da un allarme o dalle primitive *ActivateTask/ChainTask*, viene invocata la funzione 'osEE_handle_action', la quale a sua volta chiama 'osEE_task_activated'. Questa, come si può vedere dal codice riportato nella Figura 1, verifica il valore del numero di attivazioni pendenti del *task* a cui fa riferimento il parametro passato nella chiamata e, se minore del numero massimo di attivazioni impostate per quel *task*, aumenta il numero di attivazioni pendenti e imposta la variabile che identifica lo *StatusType* a 'E_OK', in caso contrario imposta quest'ultima a 'E_OS_LIMIT'.
- Al termine dell'esecuzione del *task* viene invocata la funzione 'osEE_task_end' che, come si può vedere nella Figura 1 diminuisce il valore del numero di attivazioni pendenti del *task* a cui fa riferimento il parametro passato nella chiamata e modifica lo stato del *task* in base al numero di attivazioni pendenti rimanenti.

Durante l'attivazione del *task*, se il valore di *StatusType* definito dalla variabile 'ev' nel codice in Figura 1 è diverso da 'E_OK', la funzione 'osEE_handle_action' invoca la funzione 'osEE_call_error_hook' che a sua volta chiama la funzione 'ErrorHook' implementata dall'utente come definito nello standard OSEK/VDX.

In Figura 2 è riportato lo *stack* delle chiamate nel caso di invocazione dell'*ErrorHook*.



```
FUNC(StatusType, OS_CODE) osEE_task_activated
(
  P2VAR(OsEE_TDB, AUTOMATIC, OS_APPL_DATA)  p_tdb_act
)
{
  VAR(StatusType, AUTOMATIC)                  ev;
  CONSTP2VAR(OsEE_TCB, AUTOMATIC, OS_APPL_DATA) p_tcb_act = p_tdb_act->p_tcb;
#if (!defined(OSEE_SINGLECORE))
#if (!defined(OSEE_SCHEDULER_GLOBAL))
  CONSTP2VAR(OsEE_CDB, AUTOMATIC, OS_APPL_DATA)
    p_cdb = osEE_lock_and_get_core(p_tdb_act->orig_core_id);
#else
  osEE_lock_kernel();
#endif /* !OSEE_SCHEDULER_GLOBAL */
#endif /* !OSEE_SINGLECORE */

  if (p_tcb_act->current_num_of_act < p_tdb_act->max_num_of_act) {
    ++p_tcb_act->current_num_of_act;
    ev = E_OK;
  } else {
    ev = E_OS_LIMIT;
  }

#if (!defined(OSEE_SINGLECORE))
#if (!defined(OSEE_SCHEDULER_GLOBAL))
  osEE_unlock_core(p_cdb);
#else
  osEE_lock_kernel();
#endif /* !OSEE_SCHEDULER_GLOBAL */
#endif /* !OSEE_SINGLECORE */

  return ev;
}
```

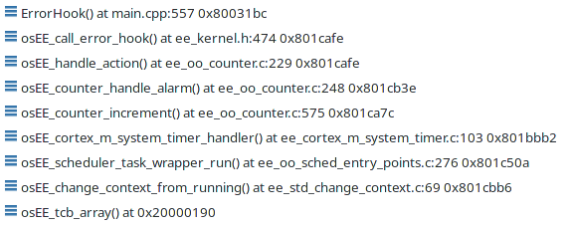
```
FUNC(void, OS_CODE) osEE_task_end
(
  CONSTP2VAR(OsEE_TDB, AUTOMATIC, OS_APPL_DATA) p_tdb
)
{
  /* It has to be called already in Multi-Core critical section */
  CONSTP2VAR(OsEE_TCB, AUTOMATIC, OS_APPL_DATA) p_tcb = p_tdb->p_tcb;

  p_tcb->current_prio = p_tdb->ready_prio;

  --p_tcb->current_num_of_act;

  if (p_tcb->current_num_of_act == 0U) {
    p_tcb->status = OSEE_TASK_SUSPENDED;
  } else {
    p_tcb->status = OSEE_TASK_READY;
  }
}
```

Figura 1: Funzioni 'osEE_task_activated' e 'osEE_task_end'



```
ErrorHook() at main.cpp:557 0x80031bc
osEE_call_error_hook() at ee_kernel.h:474 0x801cafe
osEE_handle_action() at ee_oo_counter.c:229 0x801cafe
osEE_counter_handle_alarm() at ee_oo_counter.c:248 0x801cb3e
osEE_counter_increment() at ee_oo_counter.c:575 0x801ca7c
osEE_cortex_m_system_timer_handler() at ee_cortex_m_system_timer.c:103 0x801bbb2
osEE_scheduler_task_wrapper_run() at ee_oo_sched_entry_points.c:276 0x801c50a
osEE_change_context_from_running() at ee_std_change_context.c:69 0x801cbb6
osEE_tcb_array() at 0x20000190
```

Figura 2: *Stack* delle chiamate a *ErrorHook*

1.3 UTILIZZO COME METRICA PER *DEADLINE-MISS*

Utilizzare il numero di errori di attivazione sfruttando *ErrorHook* ha certamente delle limitazioni in base alla configurazione del sistema, ed è utilizzabile solamente nel caso di *task* con *deadline* implicita in cui, se non si hanno *deadline miss*, l'attivazione di un *task* avviene quando questo non ha attivazioni pendenti. Nel mio progetto i *task* di interesse sono periodici e con *deadline* implicite, quindi utilizzare questo metodo come metrica per contare le *deadline miss* mi è sembrato di semplice attuazione e sufficiente per lo scopo.

P.S.: dopo aver effettuato l'approfondimento sottostante, ho realizzato che questo metodo soffre di una mancanza molto importante, che gli permette di identificare solamente alcune *deadline miss*. Ricordando che nell'esempio di prova i *task* possono avere al più un solo *job* nella coda dei pronti contemporaneamente, quando arriva una nuova richiesta di attivazione di un *task*, il sistema può essere in diverse situazioni:

- il *task* in questione non ha alcun *job* nella coda dei pronti, e dunque dopo l'attivazione un nuovo *job* viene inserito in questa nello stato *'ready'*;
- il *task* in questione ha un *job* nella coda dei pronti, sia questo in esecuzione o no: in questo caso il meccanismo che utilizza *ErrorHook* identifica la *deadline miss*, ma l'attivazione del nuovo *job* viene persa. In questo caso, quindi, quel *task* una volta terminato rimarrà in attesa di un nuovo allarme senza eseguire.

Quest'ultimo caso comporta un minor numero di attivazioni del *task*, e quindi anche meno *deadline miss* identificate. Questo comportamento era stato notato anche nella relazione in riferimento alla differenza di utilizzo % della CPU con vari carichi, ma non avevo pensato a questa importante implicazione.

Stimare le *deadline miss* effettive è piuttosto complesso e dipende dalle caratteristiche del sistema (priorità e tempo d'esecuzione dei *task*). È possibile però definire un *lower bound* al rapporto fra *deadline miss* rilevate e *deadline miss* effettive. Per la valutazione di questo ho mantenuto costante il tempo d'esecuzione dei *player*1 e 2, modificando quindi solamente il tempo d'esecuzione del *player*3.

Il caso peggiore si manifesta con la configurazione con priorità P3>P2>P1 (o P3>P1>P2), con *offset* di 1ms e tempo d'esecuzione del *player*3 maggiore di 10ms ma inferiore a 18ms. In questo caso le *deadline miss* rilevate riguardano tutte il *player*3 e sono circa 1/6 di quelle effettive, che invece interessano anche i *player*1 e 2, essendo questi a priorità inferiore e il *player*3 sempre in esecuzione (e avendo un *utilization*>1 non terminerà mai entro la sua *deadline*). Per

tempi d'esecuzione del *player*3 maggiori, il numero di *deadline miss* rilevate è sempre inferiore a 1/3 di quelle effettive, in quanto non vengono mai considerate le *deadline miss* sui *player*1 e 2, ma limitate comunque rispetto al caso precedente.

In alcuni casi le *deadline miss* rilevate sono invece uguali a quelle effettive: ne è un esempio l'esecuzione con priorità P3>P2>P1, tempo d'esecuzione del *player*3 pari a 9,5ms e senza alcun *offset* (ma anche con *offset* di 1ms la differenza è minima e si differenzia solamente per la prima esecuzione del *player*1). In questo caso il *task* che subisce interferenza e ha *deadline miss* è il *player*1, che non eseguirà mai e quindi non consumerà mai il *release event* iniziale, e quindi tutte le *deadline miss* sono rilevate.

Il caso peggiore con priorità P1>P2>P3 (o P2>P1>P3) si rileva con tempo d'esecuzione del *player*3 maggiore di 9ms ma inferiore a 18ms, in cui le *deadline miss* rilevate sono la metà delle *deadline miss* effettive, in quanto il *player*3 ha *deadline miss* ad ogni esecuzione, ma *release event* considerati esattamente la metà di quelli effettivi.

La varianza del rapporto tra *deadline miss* rilevate e *deadline miss* effettive è quindi elevata e stabilire un valore medio risulta complicato, soprattuto tenendo in considerazione che i *test* sono stati effettuati con un valore di carico sul *player*3 variabile nel tempo.

2 Offset

Il *critical instant* di un *task* T_i è l'evento di rilascio di un suo *job* j_i al quale corrisponde la massima estensione di interferenza, da cui consegue che, il *job* j_i rilasciato in quell'istante, ha il valore maggiore di response time fra tutti i *job* del *task* T_i.

Nel caso di *task* indipendenti fra loro, quindi senza risorse in comune o vincoli di precedenza, questo può avvenire se il *job* viene rilasciato nello stesso istante di tutti i *job* con priorità maggiore, e quindi deve aspettare prima di poter eseguire.

Rispetto al valore di *critical instant*, quanto più vicino a questo istante viene rilasciato j_i, più grande sarà il suo *response time*, con un limite superiore all'ampiezza di questo quando la fase di tutti i *job* è pari a 0 (*critical instant* al tempo 0). Da questo consegue che con *offset* pari a 0 per tutti i *task* si presenta lo scenario peggiore per lo *scheduling* del sistema [Liu, Layland: 1973].

Quando i *task* hanno priorità fissate *offline* (come nel caso di FPS), una situazione di *overloading* del sistema causata da un *job* di un *task* non può influenzare gli altri *task* a priorità maggiore. Allo stesso modo, però, una situazione di *overloading* costante del sistema può causare un blocco totale dei *task* a priorità inferiore del *task* bloccante [Mark K. Gardner, Jane W.S. Liu: *"Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload"*].

Nei risultati ottenuti dai *test* con carico riportati nella relazione, si vede infatti come la differenza fra utilizzo o meno dell'*offset*, nel caso di priorità dei *task player* rispettivamente P1>P2>P3 (e con prerilascio), sia pressochè inesistente. Questo a prova del fatto che i *task* a priorità maggiore non subiscono interferenze dai *job* in *overrun* che hanno priorità minore.

Quando i *task* però hanno priorità rispettivamente P3>P2>P1, e quindi il *task* i cui *job* causano l'*overload* del sistema ha priorità maggiore degli altri *task*, l'*offset* diventa essenziale per ridurre il numero di *deadline miss*, evitando che ci siano attivazioni dei *task* nello stesso istante.

Nel mio esempio di prova, il sistema risulta in uno stato di *overload* con *execution time* dei *job* del *player*3 compreso fra 9 e 10ms. Per quanto riguarda i *job* di *player*1 e *player*2, è ragionevole pensare che l'*execution time* di questi sia < <1ms, in quanto osservando il *log* del sistema a *runtime*, questi hanno sempre esecuzione <1ms e hanno lo stesso *timestamp*. Non avendo un valore preciso sul tempo di esecuzione di questi, considero in questo caso un *execution time* di 0,5ms.

Definiti questi tempi d'esecuzione e il periodo dei *task* di 10ms, i *task* rappresentanti i *player*1 e 2 hanno ampi margini per completare l'esecuzione, avendo un *utilization* di 0,05 ciascuno, mentre il *player*3, nell'intorno del punto di rottura, di 0,95. Risulta quindi di particolare importanza applicare un *offset* su quest'ultimo per ridurre le interferenze sui *player*1 e 2 quando questo sia possibile. Con il *player*3 che impiega 9,5ms per completare infatti, solamente il *player*2 avrà il tempo di eseguire prima della *deadline*, mentre il *player*1 non ci riuscirà mai. In questo caso l'intero sistema ha una *total utilization* > 1, che, con un'unica CPU disponibile come la *board* di prova, comporta l'impossibilità ad ottenere uno *schedule feasible*.

Se il carico, e dunque il tempo d'esecuzione del *player3*, aumenta oltre il suo periodo, in riferimento alla configurazione del sistema di *test*, la nuova attivazione viene persa. Un esempio ti tale scenario è rappresentato da un carico sul *player*3 che ne aumenta il tempo d'esecuzione a 10,5ms. In Figura 3 ho riportato il confronto fra l'esecuzione del sistema

(considerando solamente i *task player*) senza *offset*, in cui si nota come si alternino le *deadline miss* del *player*3 e dei *player*1 e 2, e del sistema con *offset* uguale a 1ms, in cui le *deadline miss* interessano il *player3*. Per la *timeline* ho considerato un tempo d'esecuzione di 2 volte l'iperperiodo + il valore dell'*offset* [Leung, Merrill: "A note on preemptive scheduling of periodic, real-time tasks"]. Ho considerato un *execution time* per i *player*1 e 2 pari a 0,5ms, mentre per il *player*3 di 10,5ms.

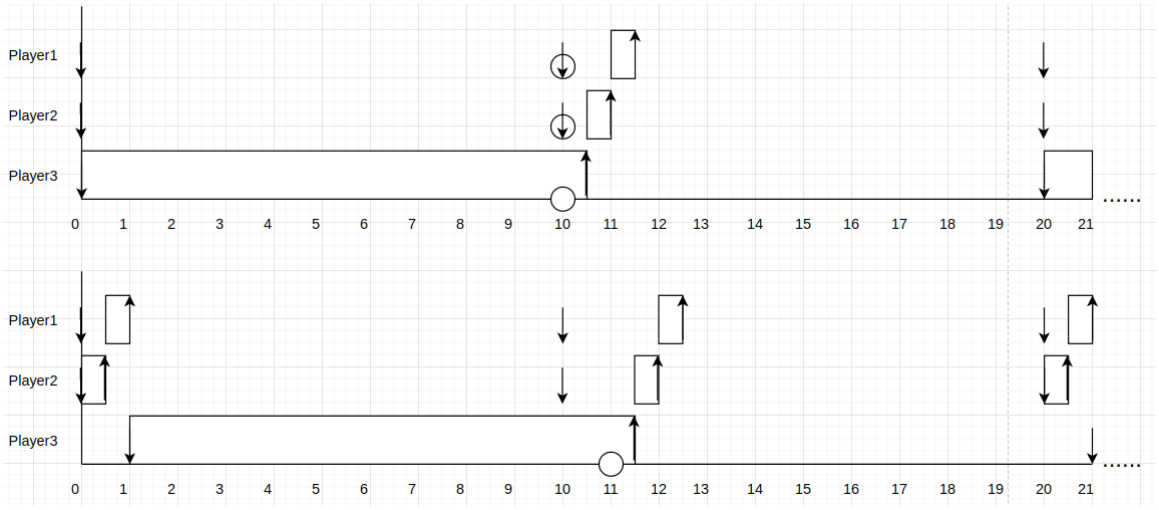


Figura 3: Confronto esecuzione senza e con *offset*

L'*offset* si rivela dunque estremamente utile in questi casi per ridurre eventuali interferenze causate da *task* a priorità maggiore, ma comporta una difficoltà maggiore nell'assegnazione di priorità ottimali ai *task* [Ken Tindell: *"Adding Time-Offsets to Schedulability Analysis"*]. A tal proposito, si può pensare al sistema presentato prima in cui il *player*3 ha un tempo di esecuzione di 9,5ms. In questo caso, se non si applica *offset*, il *player*1 non eseguirà mai, ma i *player*2 e 3 invece termineranno entro la loro *deadline* e l'utilizzo teorico della CPU sarà del 100%, mentre se si applica un *offset* pari a 1ms i *player*1 e 2 termineranno sempre, ma il *player*3 avrà una *deadline miss* ad ogni esecuzione, con conseguente perdita della nuova attivazione nel sistema di *test* proposto.