# Breve introducción a OSEK-VDX

## Un sistema operativo de tiempo real estandarizado

Mariano Cerdeiro

*Correo-e:* mcerdeiro@gmail.com

**Resumen**

OSEK-VDX es un comité de estandarización, que entre otros, especifica un sistema operativo de tiempo real. El sistema operativo OSEK-OS es utilizado hoy en día en la mayoría de los controladores de autos. Este documento realiza una breve introdución al sistema operativo de tiempo real OSEK-OS. Para más información puede visitar http://www.osek-vdx.org.

**Palabras clave:** OSEK-VDX, OSEK, OSEK-OS, RTOS, ISO-17356, sistemas embebidos, sistemas embebidos de tiempo real, operating system, real time operating system.

# 1 Introducción

OSEK-VDX es comité de estandarización creado en 1994 por las automotrices europeas. OSEK-VDX incluye varios estándares que se utilizan en la industria automotriz, entre ellos los más relevantes:

- OSEK OS
- OSEK COM
- OSEK NM
- OSEK Implementation Language
- OSEK RTI
- OSEK Time Trigger Operating System

En este documento trataremos únicamente OSEK-OS y OSEK Implementation Language (OIL).

Algunas versiones especificas de estos documentos fueron estandarizados en la ISO17356.

Además de los estándares en si, OSEK-VDX a travéz del proyecto MODISTARC realizó las especificaciones de los tests. Estos documentos también se encuentran disponibles en la página de OSEK-VDX. Los mismos especifican paso a paso como testear un sistema para certificar de que el mismo sea conforme al estándard.

OSEK-VDX es un estándard libre y abierto, más información sobre este tema se puede encontrar en http://www.osek-vdx.org, el estándard y las especificaciones también se pueden bajar desde este link.

## 1.1 Por que OSEK-VDX

El estándard fue creado principalmente para poder reutilizar el SW de un proyecto a otro gracias a la definición de una interfaz estandarizada. Además al proveer un estándard se da la posibilidad a nuevas empresas de proveer SW compatible con el estándard y permitir la implementación de diversos sistemas OSEK compatibles. Esto último permite a la industria automotriz la posibilidad de elegir el proveedor dentro de una lista mayor de proveedores de SW ya que cualquiera puede implementar un OSEK-OS u otro estándard especificado por OSEK-VDX.

Hoy en día hay muchas implementaciones de OSEK-OS en el mercado. De ellas algunas son libres con diferentes licencias, así como también hay implementaciones cerradas. Sólo hace falta buscar en google OSEK OS para encontrar algunas de ellas.

En Wikipedia: http://en.wikipedia.org/wiki/OSEK se encuentran 12 implementanciones listadas, 8 de ellas son open source.

### 1.2 Futuro del estándard

OSEK-VDX es un estándard que esta terminado y no se realizan actualizaciones. La industria automotriz está migrando a un nuevo estándard llamado AUTOSAR. Este nuevo estándard esta provisto de un sistema operativo 100% compatible con OSEK-VDX. Información de AUTOSAR inclusive la especificación se pueden encontrar en http://www.autosar.org.

## 2 Dinámico vs. estático

OSEK-OS a diferencia de otros sistemas operativos como Linux y Windows es un sistema operativo estático. Esto significa que las tareas, sus prioridades, cantidad de memoria que utilizan y etc. es definido antes de compilar el código, en un proceso que se llama generación.

En OSEK-VDX no es posible crear una tarea de forma dinámica, no es posible cambiar la prioridad a una tareas como estamos acostumbrados en Windows y Linux. Por ende OSEK-VDX seria un sistema impensable para una computadora o un celular donde constantemente estamos cargando nuevos programas, corriéndolos, cerrándolos etc. OSEK-OS está pensado para un sistemas embebido que deben realizar una tarea especifica en tiempo real y donde no se necesita cargar nuevas tareas de forma dinámica.

El ser un sistema operativo estático trae grandes ventajas a su comportamiento en tiempo real. El sistema se comporta de forma totalmente determinística. No existe la posibilidad de que una terea no pueda ser cargada por que no hay más memoria disponible como podría pasar en Linux/Windows. Las tareas tienen una prioridad asignada de ante mano, por ende una tarea de gran prioridad por que realiza un control crítico de seguridad tendrá siempre esa misma prioridad. Esto es sobre todo importante en sistemas de control críticos con requerimientos SIL. Pero además para la funcionalidad en sistemas en donde los fallos no son aceptables o tienen un costo demasiado alto.

## 3 Configuración

Al ser OSEK-OS un sistema estático es necesario configurarlo, indicar cuantas tareas hay definidas, que prioridad tienen estas tareas, etc. etc. Para ello OSEK-VDX definió otro estándard llamado OSEK Implementation Language comunmente llamado OIL.

Es un lenguaje textual muy fácil de interpretar similar al C donde uno indica las caracterísitcas del OS, Tareas, etc. Por ejemplo:

```
TASK InitTask {
  PRIORITY = 1;
  SCHEDULE = NON;
  ACTIVATION = 1;
  STACK = 128;
  TYPE = BASIC;
  AUTOSTART = TRUE {
     APPMODE = ApplicationMode1;
  }
}
```

Este ejemplo muestra la definición de una tarea llamada *InitTask* que tiene una prioridad 1, 128 bytes de stack y se auto inicia al comienzo. El resto de la información la iremos viendo en el transcurso del documento. Luego de configurar el OS debemos generar el sistema operativo. Una vez generado del OS y antes de compilar debemos crear el código C con nuestra tarea, por ejemplo un main.c con el siguiente código:

```
int main (void) {
  StartOs(ApplicationMode1)
}
TASK(InitTask) {
  /* perform some actions */
```

```
    TerminateTask();
}
```

Como se puede observar desde la main función se inicial el sistema operativo, el mismo correra automáticamente la tarea InitTask y finalmente terminara con la llamada *TerminateTask*.

# 4 Tareas

A diferencia de otros sistemas operativos donde las tareas corren por un tiempo indeterminado hasta ser terminadas, en OSEK-VDX y por lo general en sistemas de tiempo real las taras realizan su cometido y terminan. No se utilizan estructuras como *while(1)* para mantener la tarea corriendo, ni *sleeps* para dormir entre activaciónes. Sino que se inicial la tarea realiza su cometido y termina. Ya sea leer la temperatura de un sensor, recibir un paquete de comunicación, procesar datos de audio, sin importar lo que sea: se comienza, se procesa y se termina.

En los casos en donde una tarea realmente debe esperar y no puede terminar OSEK-OS provee la utilización de eventos, estos se verán más adelante.

Para el control de tareas OSEK-OS ofrece las siguientes interfaces:

- ActivateTask: activa una tarea
- ChainTask: realiza la combinación de *ActivateTask* seguido de *TerminateTask*.
- TerminateTask: termina una tarea

## 4.1 Estados

Cada tarea en OSEK-VDX se encuentra siempre en uno de los siguientes 4 estados:

- **running**: la tarea se encuentra corriendo, está utilizando los recursos del procesador en este mismo momento. En cada momento una única tarea puede encontrarse en este estado.
- **ready**: en este estado están todas las tareas que se encuentran esperando los recursos del procesador para poder correr. No se encuentran en el estado *running* por que una tarea de mayor prioridad se encuentra corriendo.
- **waiting**: la tarea se encontraba corriendo y decidio esperear la ocurriencia de un evento, hasta que el evento que espera ocurra la misma se encontrará en el estado *waiting*.
- **suspended**: es el estado por defecto de las tareas, la tarea esta momentaneamente desactivada.

La siguiente figura esquematiza los estados y sus transiciones:

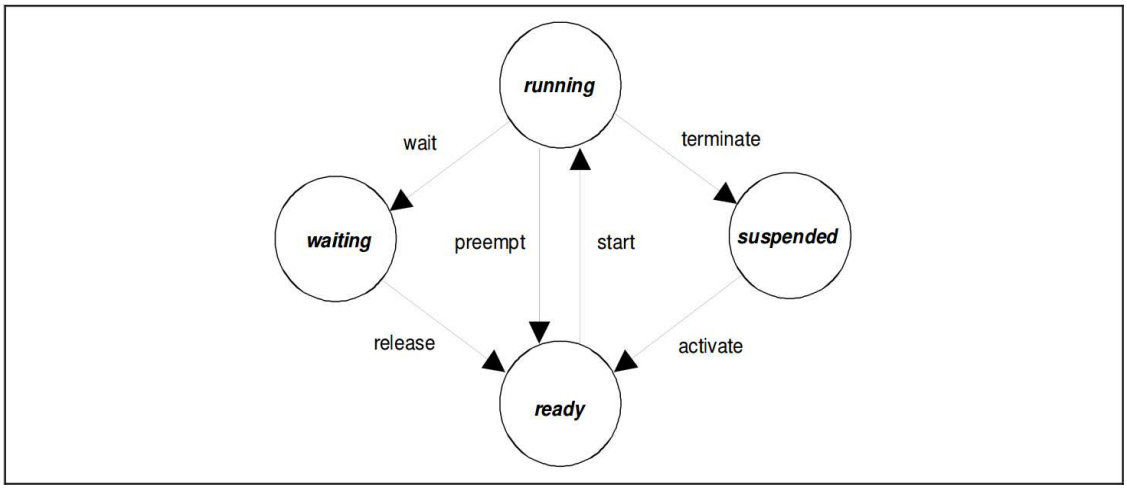


**Figura 1.** Posibles estados de una tarea (imagen de OSEK-VDX OS Standard http://www.osek-vdx.org)

La siguiente tabla explica las posibles transiciones:

| Transición | Estado anterior | Estado futuro | Descripción |
|---|---|---|---|
| activate | suspended | ready | Una nueva tarea es activada y puesta en la lista de ready para correr. Esta transición se puede realizar por ejemplo con las siguientes interfaces:<br>• ActivateTask<br>• ChainTask |
| start | ready | running | Una tarea es llevada al estado running de forma automática cuando es la tarea de mayor prioridad en la lista de *ready*. |
| wait | running | waiting | La tarea es llevada a este estado para esperar la ocurrencia de un evento. Esto se puede lograr con la siguiente interfaz:<br>• WaitEvent |
| realease | waiting | ready | Al ocurrir el evento que una tarea esperaba la misma es llevada de nuevo al estado *ready*. Esto se puede realizar con la siguiente interfaz:<br>• SetEvent |
| preempt | running | ready | Una tarea que estaba corriendo es desactivada, esto ocurre cuando una tarea de mayor prioridad se encuentra en la lista de *ready* y la tarea actual tiene una scheduling police FULL o llama a la siguiente interfaz:<br>• Schedule |
| terminate | running | suspended | La tarea termina su ejecución, esto lo puede llevar a cabo con las siguientes interfaces:<br>• ChainTask<br>• TerminateTask |

**Tabla 1.** Transiciones de estados de las tareas

## 4.2 Tipos de tareas

El sistema OSEK-VDX permite 2 tipos de tareas, **BASIC** y **EXTENDED**. Las tareas básicas no tienen eventos y por ende no pueden permanecer en el estado *waiting*. A diferencia las extendidas pueden tener uno o más eventos y esperar.

Los eventos deben ser definidos en OIL como sigue:

```
TASK(TaskA) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 5;
  STACK = 128;
  TYPE = EXTENDED;
  EVENT = Event1;
  EVENT = Event2;
}

TASK(TaskB) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 5;
  STACK = 128;
  TYPE = EXTENDED;
```

```
    EVENT = Event2;
}

EVENT Event1;

EVENT Event2;
```

En este ejemplo se ven 2 tareas *TaskA* y *TaskB*. Las 2 tareas son extendidas, la tarea *TaskA* tiene 2 Events: *Event1* y *Event2* mientras que la tarea *TaskB* tiene sólo un evento: *Event2*.

El sistema operativo ofrece las siguientes interfaces para manejar los eventos:

- WaitEvent
- SetEvent
- GetEvent
- ClearEvent

veamos un ejemplo:

```
TASK(TaskA) {
   EventMaskType Events;
   /* do something */
   WaitEvent(Event1 | Event2);
   GetEvent(TaskA, &Events);
   CleraEvent(Events);
   /* process events */
   TerminateTask();
}
TASK(TaskB) {
   /* set event */
   SetEvent(TaskA, Event1);
   TerminateTask();
}
```

El ejemplo ilustra dos tareas. *TaskA* corre y decide esperar uno o dos eventos, *Event1* y *Event2*. *TaskB* para sincronizar algún tipo de información con *TaskA* le envia un evento.

Al ocurrir el evento *TaskA* no sabe que evento ha ocurrido ya que ha llamado *WaitEvent* con 2 eventos. Por ello debe llamar a la función *GetEvent* para saber que eventos ocurrieron y por último a *ClearEvent* para borrar los eventos. Luego puede proceder a procesar la información relevante a uno o ambos eventos.

Si se llama WaitEvent con un único evento se puede simplificar el código como sigue:

```
TASK(TaskA) {
   WaitEvent(Event1);
   Clear(Event1);
   /* process Event1 */
   TerminateTask();
}
```

En este caso no es necesario llamar a la función *GetEvent* ya que solo el Evento1 puede haber ocurrido.

## 4.3 Prioridades

En OSEK OS las prioridades de las tareas se definen de forma estática en el OIL. La prioridad es un número entero entre 1 y 255. Mayor sea el número mayor es la prioridad. Si dos tareas tienen la misma prioridad son ejecutadas según su ordern de llegada FIFO. Una tarea que se encuentra corriendo nunca va a ser interrumpida por una de menor prioridad ni por una de la misma prioridad.

IMPORTANTE: A diferencia de otros sistemas operativos donde el tiempo de procesamiento es repartido de forma ponderada según la prioridad en OSEK-OS no se reparte el tiempo de ejecución, para pasar a correr una tarea de menor prioridad la tarea de mayor prioridad debe o terminar o pasar al estado *waiting*.

## 4.4 Scheduling

OSEK-OS provee 2 formas de scheduling que se pueden configurar a cada tarea:

- NON PREEMPTIVE
- PREEMPTIVE

Las tareas *NON PREEMPTIVE* no pueden ser interrumpidas nunca por otra tarea, sin importar que prioridad tengan, la tareas *NON PREEMPTIVE* se ejecutan sin interrumpción hasta que terminan, pasan al estado *waiting* esperando un evento o llaman a la función *Schedule*.

Para configurar una tarea como NON PREEMPTIVE o PREEMPTIVE se utiliza en OIL el parametro SCHEDULE. Por ejemplo:

```
TASK(TaskA) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 5;
  STACK = 128;
  TYPE = BASIC;
}

TASK(TaskB) {
  SCHEDULE = FULL;
  ACTIVATION = 1;
  PRIORITY = 8;
  STACK = 128;
  TYPE = BASIC;
}

TASK(TaskC) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 10;
  STACK = 128;
  TYPE = BASIC;
}
```

En este ejemplo la tarea *TaskA* es *NON PREEMPTIVE* mientras que la tarea *TaskB* es *PREEMPTIVE*. Si la tarea *TaskA* esta corriendo esta no será interrumpida por la *TaskB*. En cambio la tarea *TaskB* si puede ser interrumpida por la tarea *TaskC*, ya que *TaskB* es *PREEMPTIVE* y *TaskC* tiene una prioridad mayor.

La función Schedule es una alternativa para forzar al OS de verificar si hay tareas de mayor prioridad por correr.

Las tareas *NON PREEMPTIVE* se utilizan generalmente en dos situaciones.

### 4.4.1 Tarea de corta ejecución

Muchas veces una tarea de muy corta duración se configura como *NON PREEMPTIVE*, de esta forma se evita que sea interrumpida ya que el tiempo que lleva el cambio de tarea es similar al tiempo que dura la tarea en ejecutarse completamente. De esta forma se evita perdida de tiempo en el cambio de contexto.

### 4.4.2 Sistema determinístico

En muchos casos es recomendable usar tareas *NON PREEMPTIVE* para hacer el sistema determinístico. Para ello se define la tarea *NON PREEMPTIVE* y se llama al Schedule. Por ejemplo:

```
TASK(TaskA) {
   /* perform some actions */
   Schedule();
```

```
    /* perform more actions */
    Schedule();
    /* perform more actions */
    TerminateTask();
}
```

Un esquema como el de la tarea *TaskA* siendo *TaskA NON PREEMPTIVE* permite al programador estar seguro de en que momentos puede ser interrumpido su código. Solo cuando llama a la función Schedule y si cuando se llama hay una tarea de mayor prioridad se cambiara de tarea.

Llamar a la interfaz Schedule desde una tarea *PREEMPTIVE* se puede hacer pero carece de sentido. Como la tarea es interrumpida de todos modos, no es necesario llamar al Schedule.

# 5 Recursos

Los recursos en todo sistema sea o no de tiempo real son por lo general escasos. Por ello para acceder a ellos se utilizan en los sistemas convencionales semáforos o mutex para poder aceder a ellos desde varias tareas sin generar interferencias. La utilización de semáforos y mutex genera dos problemas conocidos: inversión de prioridades y deadlocks. OSEK-OS ofrece una solución para el acceso a recursos que evita estos dos problemas.

La problemática de los deadlocks es muy conocida y por ello no será tratada en esta breve introducción. En cambio el problema de inversión de prioridades no es tan conocido y por ello le dedicamos un apartado.

## 5.1 Inversión de prioridades

En el siguiente ejemplo hay 4 tareas: *T1*, *T2*, *T3* y *T4* donde *T1* tiene mayor prioridad que *T2*, *T2* tiene mayor prioridad que *T3* y *T3* tiene mayor prioridad que *T4*.

Todas estas tareas comparten la utilización de un mismo recurso. En el ejemplo de la figura *T4* se encuentra corriendo y utiliza un semáforo para reservar la utilización de un recurso. Mientras que tiene el recurso reservado se activan las otras tareas: *T1*, *T2* y *T3*. Como *T4* es la tarea con menor prioridad su ejecución es pausada y se procede a ejecutar las otras tareas.

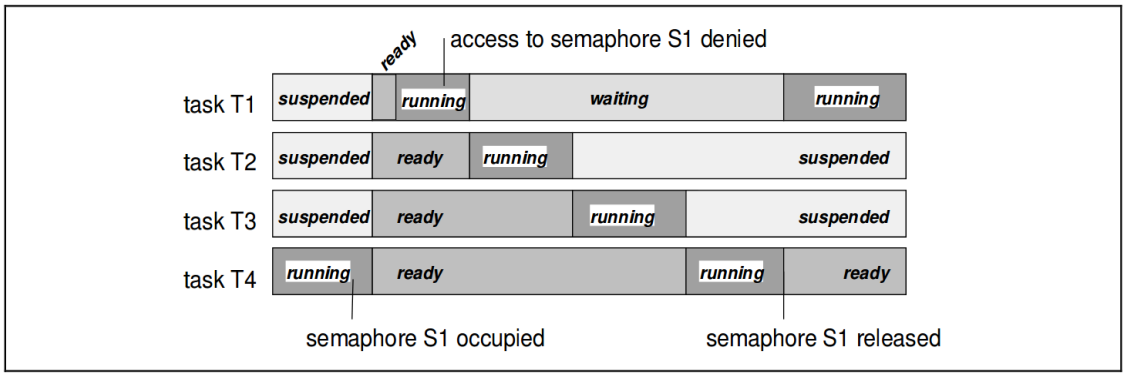


**Figura 2.** Inversión de prioridades (imagen de OSEK-VDX OS Standard http://www.osek-vdx.org)

La primera que se comienza a ejecutar es *T1*. Sin embargo *T1* requiere la utilización del recurso que momentaneamente es utilizado por *T4*. Como no puede acceder a este recurso espera a que este sea liberado. Como *T1* no puede correr más ya que se encuentra esperando el recurso el sistema ejecuta *T2* y *T3*. Por último cuando todas las otras tareas terminaron de ser ejecutadas corre *T4*. *T4* termina la utilización del recurso y lo libera. Recién al liberar *T4* el recurso se pasa a contunar con la ejecución de *T1*.

En este ejemplo podemos ver la **grave** situación en la que nos encontramos, esta situación es **totalmente inadmisible** en un sistema crítico. La ejecución de una tarea de prioridad alta *T1* es retrasada por la ejecución de otras 2 Tareas *T2* y *T3* de menor prioridad. A esto se le denomina inversión de prioridades.

## 5.2 OSEK Priotity Ceiling Protocol

OSEK-OS provee una solución mucho más adecuada para esta situación mediante la utilización de recursos. Veamos el mismo ejemplo con las tareas *T1*, *T2*, *T3* y *T4*. Como *T1* y *T4* tiene acceso al mismo recurso. Se le otorga al recurso en si una prioridad igual a la prioridad de la tarea más alta que lo utiliza. Como el recurso es utilizado por *T1* que tiene la mayor prioridad el recurso tiene una prioridad igual a la de *T1*.

*T4* corre con una prioridad baja, sin embargo cuando le pide al sistema operativo acceso al recurso el sistema operativo le otorga mientras utilice el recurso una prioridad similar a la de *T1*, osea la prioridad asignada al recurso. Como en el ejemplo anterior mientras que *T4* utiliza el recurso se activan *T1*, *T2* y *T3*. Sin embargo ninguna de las otras tareas es ejecutada a pesar de tener mayor prioridad que *T4*. Esto se debe a que *T4* utiliza momentaneamente el recurso y hasta que no lo libere tiene una pioridad mayor.

Cuando *T4* libera el recurso vuelve a su prioridad normal que es baja. En ese momento *T4* es interrumpida y se procede con la ejecución de las otras tareas. Como *T1* es la de mayor prioridad y esta lista para ser ejecutada se procede a su ejecución.

En este ejemplo se puede ver claramente que la ejecución de *T1* solo se retrasa el tiempo que dura la utilización del recurso y no como en el ejemplo anterior donde se retrasaba innecesariamente por un tiempo mucho mas extendido.

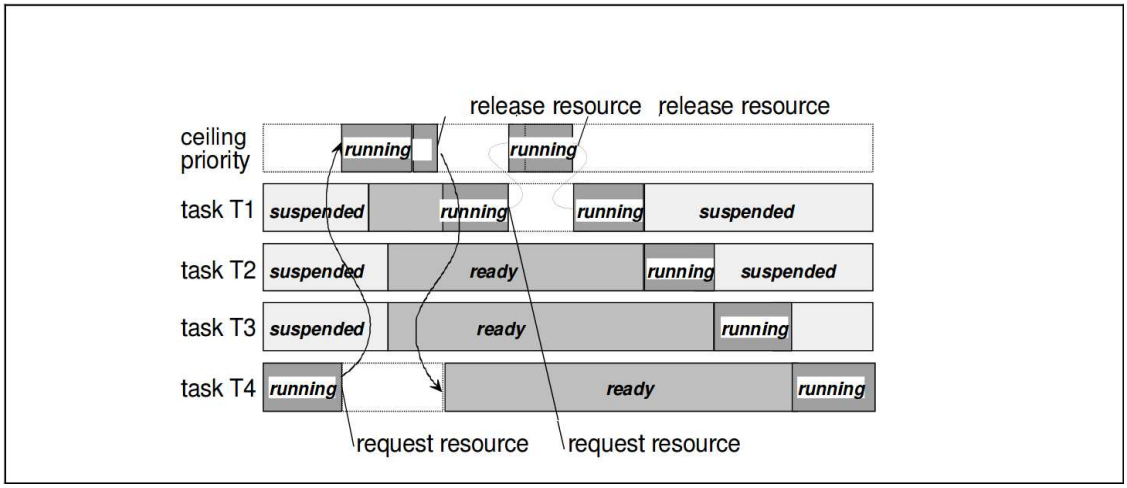


**Figura 3.** OSEK Priority Ceiling Protocol (imagen basada en OSEK-VDX Standard http://www.osek-vdx.org)

Para la utilización de recursos los mismos deben ser declarados en OIL e indicado que recursos van a ser utilizados por cada tarea, por ejemplo:

```
TASK(TaskA) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 5;
  STACK = 128;
  TYPE = BASIC;
  RESOURCE = Res1;
  RESOURCE = Res2;
}

TASK(TaskB) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 7;
  STACK = 128;
```

```
  TYPE = BASIC;
  RESOURCE = Res1;
}

TASK(TaskC) {
  SCHEDULE = NON;
  ACTIVATION = 1;
  PRIORITY = 3;
  STACK = 128;
  TYPE = BASIC;
  RESOURCE = Res2;
}
```

El ejemplo muestra la configuración de 3 tareas con 2 recursos: *Res1* y *Res2*. El recurso *Res1* va a tener una prioridad equivalente a 7 ya que es utilizado por *TaskA* y *TaskB*. Mientras que el recurso *Res2* tendrá una prioridad 5 ya que es utilizado por *TaskA* y *TaskC*. Desde el programa se puede acceder a los recursos mediante dos interfaces:

- GetResource
- ReleaseResource

A continuación un ejemplo en el que las tareas *TaskA* y *TaskC* usan los recursos *Res1* y *Res2* respectivamente.

```
TASK(TaskA) {
   /* some code */
   GetResource(Res1);

   /* perform actions using resource 1 */
   /* during this section TaskA has priority 7 */

   ReleaseResource(Res1);

   TerminateTask();
}

TASK(TaskC) {
   /* some code */
   GetResource(Res2);

   /* perform actions using resource 2 */
   /* during this section TaskA has priority 5 */

   ReleaseResource(Res2);

   TerminateTask();
}
```

# 6 Alarmas

Las alarmas son utilizadas para realizar una acción luego de determinado tiempo. Las alarmas de OSEK-OS pueden realizar tres tipos de acciones:

- Activar una tarea
- Setear un evento de una tarea
- Llamar una callbac en C

Para la implementación de las alarmas, ya sea 1 o n el sistema operativo necesita un contador de HW. Con un contador es posible configurar la cantidad de alarmas que sea necesario. Cada alarma debe ser definida en el formato OIL así como la correspondiente acción:

```
ALARM ActivateTaskC {
  COUNTER = SoftwareCounter;
  ACTION = ACTIVATETASK {
    TASK = TaskC;
  }
  AUTOSTART = FALSE;
}

ALARM SetEvent1TaskA {
  COUNTER = SoftwareCounter;
  ACTION = SETVENT {
    TASK = TaskA;
    EVENT = Event1;
  }
  AUTOSTART = FALSE;
}
```

El ejemplo muestra dos alarmas, la primera alarma se llama *AcitvateTaskC* y al expirar activa la tarea *TaskC*. La segunda alarma se llama *SetEvent1TaskA* y al expirar activa el evento *Event1* de la tarea *TaskA*. En este ejemplo las alarmas no se activan automáticamente, por lo que la activación debe ser realizada desde el código, para ello se utilizan las siguientes interfaces:

- SetRelAlarm
- SetAbsAlarm
- CancelAlarm

La primera interfaz setea una alarma con el tiempo relativo al actual, la segunda indica un tiempo absoluto y la última para la alarma. Veamos un ejemplo en el código:

```
TASK(TaskB) {
   /* some code */
   SetRelAlarm(ActivateTaskC, 100, 100);
   SetRelAlarm(SetEvent1TaskA, 150, 200);

   TerminateTask();
}

TASK(TaskC) {
   static int counter = 0;

   if (counter ++ > 10) {
      CancelAlarm(ActivateTaskC);
   }

   /* do something */

   TerminateTask();

}
```

En el ejemplo la tarea *TaskB* activa 2 alarmas. La alarma *ActivateTaskC* es activada para expirar por primera vez luego de 100 ticks y luego reiteradamente cada 100 ticks. La alarma *SetEvent1TaskA* es activada para expirar por primera vez luego de 150 ticks y luego reiteradamente cada 200 ticks.

Por último la tarea *TaskC* corre 10 veces y luego desactiva la alarma que la estaba activando.

NOTA: Gracias al desfasaje inicial de 100 y 150 ticks se puede lograr que las alarmas no expiren en el mismo momento sino desfasadazas 50 ticks. Esto es bueno para evitar jitter, ya que sino el OS debe procesar las alarmas en el mismo momento. Este recurso se utiliza mucho en OSEK-OS y en sistemas de tiempo real.

# 7 Hay más

Este documento, como el título lo indica, es una breve introducción a OSEK-OS. Los lineamentos generales del sistema se encuentran explicados, pero no se mencionan algunos temas importantes como ser: interrupciones, manejo de errores, escalabilidad entre otros. Más informacón se puede encontrar en el estándar que se puede bajar de http://www.osek-vdx.org.

# 8 Conclusión

OSEK-OS es un sistema operativo de tiempo real especialmente diseñado para sistemas embebidos críticos. Donde la ejecución determinística del sistema es un punto esencial. Esto se aplica especificamente en sistemas de control con un nivel de seguridad crítico por ejemplo SIL o ASIL. En automoviles se utiliza en el 90% de los controladores, como por ejemplo control del motor, caja de cambios automáticas, abs, control de tracción, etc.

También se aplica a sistemas no críticos desde el punto de seguridad sino desde el punto de vista de accesibilidad. Donde la perdida del sistema causa un costo muy grande a la empresa o al usuario. Nadie aceptaría un auto donde cada tanto sea necesario reiniciarlo por que la memoria del sistema se encuentra demasiado fragmentada y *malloc* no otorga más bloques de memoria consecutivos. Para ello OSEK-OS, y gracias a que es un sistema estático, define todos los recursos que necesita durante la generación y no utiliza nunca funciones no determinísticas como por ejemplo *malloc*.