

Introducción a OSEK-OS

El Sistema Operativo del CIAA-Firmware

Programación de Sistemas Embebidos

1ra Edición, Noviembre de 2015

MSc. Ing. Mariano Cerdeiro

Introducción a OSEK-OS - El Sistema Operativo del CIAA-Firmware

Programación de Sistemas Embebidos

MSc. Ing. Mariano Cerdeiro

Introducción a OSEK-OS - El Sistema Operativo del CIAA-Firmware: Programación de Sistemas Embebidos

MSc. Ing. Mariano Cerdeiro
Copyright © 2015 Mariano Cerdeiro

Editorial ACSE.

El enlace oficial para bajar la última versión *PDF* de este documento es <http://soix.tk/downloads>. Este documento también se puede leer en linea en formato *HTML*.

El copyright sobre los íconos



es de VistaICO.com. Los mismos se distribuyen bajo la licencia <http://creativecommons.org/licenses/by/3.0/>.

El copyright sobre el archivo style.css utilizado en el formato *HTML* es de The FreeBSD Documentation Project y se distribuyen bajo la licencia BSD de 2 cláusulas.

El resto del trabajo se distribuye con la Licencia Creative Commons Atribución - No Comercial - Compartir Igual 4.0. <http://creativecommons.org/licenses/by-nc-sa/4.0/>



Cerdeiro, Mariano

Introducción a OSEK-OS. El sistema operativo del CIAA-Firmware : programación de sistemas embebidos / Mariano Cerdeiro. - 1a ed revisada. - Ciudad Autónoma de Buenos Aires : ACSE - Asociación Civil para la investigación, Promoción y Desarrollo de Sistemas Eléctricos Embebidos, 2015.

Libro digital, PDF

Archivo Digital: descarga
ISBN 978-987-45523-6-5

1. Ingeniería de Software. 2. Electrónica Digital. 3. Software. I. Título.
CDD 629.89

A mi mujer.

Agradecimientos

Antes que a nadie quiero agradecerle a mi esposa *Elay* y a mi hija *Emilia* por haberme dado el tiempo necesario para escribir este trabajo, cuya compilación demandó casi un año, durante el cual no pude compartir con ellas todo el tiempo que hubiese querido.

También quiero agradecer a todos los integrantes del Proyecto CIAA. Gracias al mismo y a la decisión de utilizar un OSEK-OS y específicamente la implementación FreeOSEK como RTOS del CIAA-Firmware es como FreeOSEK vuelve a nacer.

Dentro de los colaboradores del Proyecto CIAA quiero agradecer especialmente a *Pablo Ridolfi* que fue quien se puso a la tarea de portar el FreeOSEK a la primera versión de la CIAA, la CIAA-NXP basada en un LPC4337. A *Juan Ceconni* por tomarse la tarea de que el CIAA-Firmware y así el FreeOSEK pudiera ser depurado mediante OpenOCD. A *Gustavo Muro* por ser el primero en darle al FreeOSEK y al CIAA-Firmware un lugar en la industria, armando un tablero industrial con una CIAA-NXP corriendo FreeOSEK. A *Paola Pezoimburu* por las horas invertidas en la documentación del proyecto y sobre todo corrigiendo algunos de mis horrores ortográficos. A *Ariel Lutenberg* por hacer posible el proyecto, y por haber corregido versiones preliminares de este documento.

Al ACSE y especialmente a *Diego Brengi* por la tramitación del ISBN.

Por último quiero agradecer a *Alfredo Marega* por realizar el diseño de la portada, y siempre ayudarme con los diseños cuando necesito de uno.

Tabla de contenidos

Nota del autor	viii
Glosario	1
1. Introducción a OSEK-OS	2
1.1. ¿Por qué OSEK-OS?	2
1.2. Futuro del estándar	3
1.3. Dinámico vs. estático	3
1.4. ¿Porqué el Firmware de la CIAA utiliza OSEK-OS?	3
1.5. El estándar de OSEK-OS	4
2. Clases de Conformidad	5
2.1. Clases de conformidad en el CIAA-Firmware	6
3. Configuración y generación	7
3.1. Configuración	8
3.2. Generación	9
3.3. Ventajas y desventajas de un sistema operativo generado	13
3.3.1. Recursos	13
3.3.2. Scheduler y Tareas	14
4. Tareas	15
4.1. Sobre el while(1) y tareas infinitas	16
4.2. Estados	16
4.3. Tipos de Tareas	18
4.3.1. Múltiples Activaciones	19
4.4. Prioridades	21
4.4.1. Orden de ejecución	21
4.5. Tipos de Scheduling	22
4.5.1. Tareas de corta ejecución	23
4.5.2. Sistema determinista	23
4.5.3. Puntos de Scheduling	24
5. Recursos	25
5.1. Inversión de prioridades	25
5.2. Deadlock	26
5.3. OSEK Priority Ceiling Protocol	26
5.4. El Scheduler como recurso	29
5.5. Recursos Internos	29
5.6. Recursos Linkeados	31
5.7. Recursos en interrupciones	31
6. Alarmas	33
6.1. Offset en las Alarmas	34
7. Interrupciones	36
7.1. Anidamiento	38
8. Hookroutines	39
8.1. StartupHook	40
8.2. PreTaskHook	41
8.3. PostTaskHook	42
8.4. ErrorHook	43
8.5. ShutdownHook	44
9. Manejo de Errores y depuración	45
9.1. La función ErrorHook	46
10. Conclusiones y perspectivas	49
A. Manual de referencia de OSEK-OS	51
A.1. Tipos	52
A.1.1. StatusType	52
A.1.2. AlarmBaseType	53
A.1.3. TickType	53
A.1.4. TickRefType	53
A.2. ActivateTask	54

A.3. ChainTask	55
A.4. GetTaskID	57
A.5. GetTaskState	58
A.6. Schedule	59
A.7. TerminateTask	61
A.8. DisableAllInterrupts	62
A.9. EnableAllInterrupts	63
A.10. SuspendAllInterrupts	64
A.11. ResumeAllInterrupts	65
A.12. SuspendOSInterrupts	66
A.13. ResumeOSInterrupts	67
A.14. GetResource	68
A.15. ReleaseResource	69
A.16. SetEvent	70
A.17. GetEvent	71
A.18. WaitEvent	72
A.19. ClearEvent	73
A.20. GetAlarmBase	74
A.21. GetAlarm	75
A.22. SetAbsAlarm	76
A.23. SetRelAlarm	77
A.24. CancelAlarm	78
A.25. GetActiveApplicationMode	79
A.26. StartOS	80
A.27. ShutdownOS	81
Bibliografía	82
Índice	83

Lista de figuras

2.1. Clases de conformidad	5
3.1. Configuración, generación, compilación y lindeo del sistema	8
3.2. Captura de pantalla de <i>gobox</i>	9
4.1. Posibles estados de una tarea	17
4.2. Activación Múltiple	20
4.3. Scheduling en BCC1 y ECC1	21
4.4. Scheduling en BCC2 y ECC2	22
5.1. Inversión de prioridades	25
5.2. Deadlocks	26
5.3. OSEK Priority Ceiling Protocol	27
6.1. Desfase de tareas mediante offset	35
7.1. Planificación de Tareas e Interrupciones en OSEK-OS	37

Lista de tablas

2.1. Requerimientos por cada clase de conformidad	6
4.1. Transiciones entre los estados de las tareas	17
A.1. Interfaces de OSEK-OS y contexto del que pueden ser llamadas	51

Lista de ejemplos

3.1. Ejemplo de configuración mediante OIL	7
3.2. Extracto de la configuración de blinking del CIAA-Firmware	10
3.3. Salida del generador	10
3.4. Generación de Os_Internal_Cfg.c.php	11
4.1. Definiendo una tarea	15
4.2. Activación múltiple de una tarea	20
5.1. Definición de recursos en la configuración	28
5.2. Utilizando RES_SCHEDULER	29
5.3. Utilización de recursos internos	30
5.4. Utilización de recursos linkeados	31
5.5. Utilización de recursos en interrupciones	32
6.1. Configuración de alarmas	33
6.2. Utilización de alarmas	34
7.1. Utilización de interrupciones	37
8.1. Utilización de StartupHook	40
8.2. Utilización de PreTaskHook	41
8.3. Utilización de PostTaskHook	42
8.4. Utilización de ErrorHook	43
8.5. Utilización de ShutdownHook	44
9.1. Manejo de errores	45
9.2. Manejo de errores mediante ErrorHook	47
A.1. Utilización de ActivateTask	54
A.2. Utilización de ActivateTask	56
A.3. Utilización de GetTaskID	57
A.4. Utilización de GetTaskState	58
A.5. Utilización de Schedule	59
A.6. Utilización de TerminateTask	61
A.7. Utilización de DisableAllInterrupts	62
A.8. Utilización de EnableAllInterrupts	63
A.9. Utilización de SuspendAllInterrupts	64
A.10. Utilización de ResumeAllInterrupts	65
A.11. Utilización de SuspendOSInterrupts	66
A.12. Utilización de ResumeOSInterrupts	67
A.13. Utilización de GetResource	68
A.14. Utilización de ReleaseResource	69
A.15. Utilización de SetEvent	70
A.16. Utilización de GetEvent	71
A.17. Utilización de WaitEvent	72
A.18. Utilización de ClearEvent	73
A.19. Utilización de GetAlarmBase	74
A.20. Utilización de GetAlarm	75
A.21. Utilización de SetAbsAlarm	76
A.22. Utilización de SetRelAlarm	77
A.23. Utilización de CancelAlarm	78
A.24. Utilización de GetActiveApplicationMode	79
A.25. Utilización de StartOS	80
A.26. Utilización de ShutdownOS	81

Nota del autor

En abril del 2014 comencé a participar en el desarrollo del Proyecto CIAA, trabajando en el CIAA-Firmware. Desde el momento en que se decidió utilizar FreeOSEK como el sistema operativo del CIAA-Firmware, surgió la idea de escribir este documento. La idea inicial era escribir un libro que contenga diversos temas relacionados a los sistemas embebidos, como ser: procesos de desarrollo de software, trazabilidad de requerimientos, técnicas de validación y verificación, sistemas operativos de tiempo real, entre otros. Para quienes hayan asistido a las charlas que he dictado durante el SASE 2014 y 2015, la intención era incorporar los temas tratados en estos seminarios armando así un libro sobre sistemas embebidos. Lamentablemente el tiempo fue demostrando que tan solo la parte de sistemas operativos llevaría casi un año de recopilación. Tal vez en un futuro sea posible extender este trabajo a otros temas.

Hasta que una recopilación más completa de diversos temas de sistemas embebidos no me sea posible, espero que este documento en su estado actual sirva para facilitar la utilización del CIAA-Firmware. A su vez genere un mayor entendimiento sobre OSEK-OS específicamente, y en general sobre sistemas operativos de tiempo real.

Con la finalidad de que este documento llegue a la mayor cantidad de personas posible es que se ofrece de forma gratuita.

En lo personal espero se pueda apreciar la gran cantidad de horas invertidas en este documento, que de ninguna forma diría se encuentre terminado. Tengo la sensación de que un libro es como el software, nunca está terminado. Siempre habrá aspectos que mejorar. En mi caso sobre todo en la escritura, que como se darán cuenta durante la lectura no es mi fuerte. Sin embargo seguir retrasando la primera versión solo imposibilita a posibles lectores acceder a esta información. Por ello decidí publicar esta primera versión. Espero les sea de gran utilidad.

Glosario

ACSE	Asociación Civil para la Investigación, Promoción y Desarrollo de los Sistemas Embebidos (http://www.sase.com.ar/asociacion-civil-sistemas-embebidos/).
CIAA	Computadora Industrial Abierta Argentina (http://www.proyecto-ciaa.com.ar).
FreeOSEK	Nombre original del sistema operativo de tiempo real adoptado por el CIAA-Firmware.
ISR	Interrupt Service Routine.
OIL	OSEK Implementation Language.
SASE	Simposio Argentino de Sistemas Embebidos (http://www.sase.com.ar).

Capítulo 1. Introducción a OSEK-OS

OSEK-VDX es un comité de estandarización creado en 1994 por las automotrices europeas¹. OSEK-VDX incluye varios estándares que se utilizan en la industria automotriz, entre ellos los más relevantes:

- *OSEK OS*: especifica un sistema operativo basado en prioridades.
- *OSEK Implementation Language*: especifica un lenguaje de configuración utilizado por los otros estándares.
- *OSEK COM*: especifica entorno de comunicación.
- *OSEK NM*: ofrece una interfaz para la administración redes.
- *OSEK RTI*: estandariza una interfaz para poder comprender el manejo interno de datos de los estándares mediante herramientas como depuradores.
- *OSEK/VDX Time Trigger Operating System*: especifica un sistema operativo basado en ventanas de tiempo.

En este documento se tratan únicamente OSEK OS y OSEK Implementation Language (OIL).

Algunas versiones específicas de estos documentos fueron estandarizados en la ISO17356².

Además de los estándares en sí, OSEK-VDX a través del proyecto MODISTARC realizó las especificaciones de las pruebas³. Estos documentos también se encuentran disponibles en la página de OSEK-VDX [<http://portal.osek-vdx.org>]. Los mismos especifican paso a paso cómo verificar un sistema para certificar que el mismo sea conforme al estándar.

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

1.1. ¿Por qué OSEK-OS?

El estándar fue creado principalmente para poder reutilizar el software entre proyectos, gracias a la definición de una interfaz estandarizada. Además al proveer un estándar se da la posibilidad a las empresas de ofrecer software compatible con el estándar y permitir la implementación de una mayor cantidad de sistemas OSEK compatibles. Esto último permite a la industria automotriz y a quien quiera contar con la posibilidad de seleccionar un OSEK entre una mayor cantidad de proveedores.

Existen muchas implementaciones de OSEK-OS, algunas de ellas de código abierto. Un listado de alguna de ellas se pueden encontrar en Wikipedia [<http://en.wikipedia.org/wiki/OSEK>].

¹http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=4

²http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=17&Itemid=20

³http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=14&Itemid=17

1.2. Futuro del estándar

OSEK-VDX Operating System es un estándar de 2005 al cual no se le realizan más actualizaciones. La industria automotriz se encuentra migrando a un nuevo estándar llamado [AUTOSAR]. Este nuevo estándar está provisto de un sistema operativo compatible con OSEK-OS llamado AUTOSAR-OS⁴. Más información de AUTOSAR, inclusive la especificación, se puede encontrar en <http://www.autosar.org>.

1.3. Dinámico vs. estático

A diferencia de otros sistemas operativos como Linux y Windows, OSEK-OS es un sistema operativo estático. Las tareas, sus prioridades, la cantidad de memoria que utilizan, etc. son definidas antes de compilar el código, durante un proceso que se denomina *generación*.

En OSEK-OS no es posible crear una tarea de forma dinámica, hacer un `fork()` ni cambiar la prioridad a una tarea como estamos acostumbrados en Windows y Linux. OSEK-OS sería un sistema operativo impensable para una computadora de propósitos generales o un celular donde constantemente estamos cargando nuevos programas y aplicaciones. OSEK-OS está pensado para un sistema embebido con un propósito específico y no general como una computadora de escritorio o un celular. La finalidad es realizar una tarea específica en tiempo y forma, donde la tarea a realizar está definida con anterioridad y no es necesario 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 determinista. No existe la posibilidad de que una tarea no pueda ser cargada porque no hay suficiente memoria disponible. Las tareas tienen una prioridad que les fue asignada en tiempo de generación (antes de la compilación), por lo que una tarea de mayor prioridad que realiza un control crítico tendrá siempre esa misma prioridad asignada.

En Sección 3.3, "Ventajas y desventajas de un sistema operativo generado" se presenta este tema nuevamente con un poco más de detalle.

1.4. ¿Porqué el Firmware de la CIAA utiliza OSEK-OS?

Al comenzar el desarrollo del Firmware de la CIAA se discutió cual RTOS convenía utilizar. Se analizó la posibilidad de realizar una abstracción que permitiese la utilización de diversos sistemas operativos y se pensó en utilizar FreeRTOS⁵. Sin embargo finalmente se decidió basar el Firmware en una implementación de OSEK-OS.

⁴Requerimiento SWS_Os_00001 en [AUTOSAR-OS]

⁵Marca registrada de Real Time Engineers Ltd. <http://www.freertos.org/>

La utilización de una capa de abstracción fue descartada ya que hubiese beneficiado a algunos RTOS sobre otros. Por ejemplo el manejo de *Recursos* es muy distinto en OSEK-OS y en FreeRTOS⁵. Por ello fue descartada esta posibilidad.

De haberse utilizado FreeRTOS⁵ todos los usuarios del Firmware seríamos cautivos de un proveedor de sistema operativo. Una única empresa a la cual solicitar soporte, documentación, nuevos ports y versiones comerciales y/o para seguridad funcional. Además, dada su licencia todo producto utilizando FreeRTOS⁵ se encuentra obligado a indicarlo en la documentación del producto final⁶

La utilización de OSEK-OS como estándar de sistema operativo de tiempo real del Firmware de la CIAA trae además grandes ventajas. Por un lado se trata de un estándar que es abierto y no una implementación. Permitiendo de esta forma en el futuro intercambiar el RTOS por otro compatible. Además nos da la posibilidad de tener un sistema operativo propio incluyendo sus ensayos, ya que OSEK-OS especifica no solo un sistema operativo sino también como verificarlo. De esta forma es posible migrar el Firmware a la plataforma que se crea conveniente sin tener que pedir soporte a terceros.

El CIAA-Firmware implementa un OSEK-OS ofreciendo tanto el código como sus ensayos de código abierto, así se evita la dependencia de los usuarios del Firmware a una organización (empresa, institución, etc). Cualquiera con conocimiento en el tema puede modificar, mejorar, implementar y vender un OSEK-OS. Un usuario podría por ejemplo seleccionar entre diferentes proveedores de OSEK-OS según su prestación en situaciones específicas que le sean de interés dada una utilización específica.

1.5. El estándar de OSEK-OS

El estándar de OSEK-OS se puede encontrar en [OSEK-OS], es importante recordar que este documento no remplaza al estándar. Es un mero aporte para quienes prefieran leer una documentación en español. Algunos conceptos son explayados en más detalle que en el estándar y para cada interfaz del sistema operativo se adjunta un ejemplo de su utilización.

A aquellos que quieran implementar un sistema operativo basado en el estándar, o que tengan que verificar detalles muy específicos, se les recomienda utilizar [OSEK-OS] y este documento como complemento.

⁶En License Details [<http://www.freertos.org/a00114.html>] se indica que se debe informar que el producto utiliza FreeRTOS⁵, por ejemplo mediante un link.

Capítulo 2. Clases de Conformidad

Con la finalidad de permitir que OSEK-OS pueda ser utilizado en una gran variedad de sistemas con diferentes capacidades y demandas (por ejemplo memoria, capacidad de procesamiento, etc.) es que OSEK-OS define 4 clases de conformidad (CC).

Certificación

Las diferentes clases existen para poder comparar entre sistemas, agrupar las interfaces según la clase de conformidad y facilitar la certificación de conformidad.

Las clases de conformidad se determinan según los siguientes atributos:

- Múltiple activación de tareas básicas
- Tipos de tareas aceptadas: básicas y extendidas
- Cantidad de tareas por prioridad

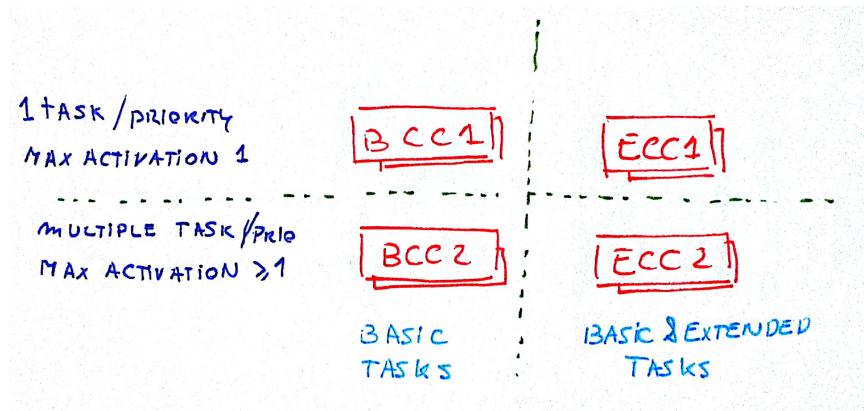


Figura 2.1. Clases de conformidad

De esta forma se definen las siguientes clases de conformidad:

- *BCC1*: que soporta únicamente tareas básicas con máximo una activación por tarea, y todas las tareas con prioridad diferente.
- *BCC2*: como BCC1 pero con más de una tarea por prioridad y las tareas soportan más de una activación.
- *ECC1*: como BCC1 pero con tareas extendidas
- *ECC2*: como ECC1 pero con más de una tarea por prioridad y las tareas soportan más de una activación.

A su vez se define una cantidad mínima de requerimientos por cada clase de conformidad como se indica en la Tabla 2.1, “Requerimientos por cada clase de conformidad”.

Característica	BCC1	BCC2	ECC1	ECC2
Más de una activación	no	sí	no	sí para tareas básicas
Cantidad de tareas en un estado distinto a suspended		>7		>15
Más de una tarea por prioridad	no	sí	no	sí
Cantidad de eventos por tarea	-			>7
Cantidad de prioridades		>7		>15
Recursos	RES_SCHEDULER	>7 (incluido RES_SCHEDULER)		
Recursos Internos		>1		
Alarmas		>0		
Modos de aplicación		>0		

Tabla 2.1. Requerimientos por cada clase de conformidad

Cada sistema operativo compatible documenta por lo general una tabla indicando qué clases de conformidad son soportadas y cuáles son los valores para los parámetros indicados en Tabla 2.1, “Requerimientos por cada clase de conformidad”, siendo los indicados en la tabla únicamente un mínimo y pudiendo una implementación soportar valores superiores.

2.1. Clases de conformidad en el CIAA-Firmware

El OSEK-OS de la CIAA soporta todas las conformidades. Dependiendo de esta configuración el sistema decidirá si se implementa una clase *BCC1* ó *ECC2*. Por lo que para el usuario final del CIAA-Firmware es prácticamente irrelevante conocer las clases. El Firmware decide automáticamente y configura la clase de conformidad correcta según los requerimientos del usuario. En el Capítulo 3, *Configuración y generación* se describe como se puede configurar el sistema operativo.



Lamentablemente el OSEK-OS del CIAA-Firmware todavía no cuenta con una tabla similar a la Tabla 2.1, “Requerimientos por cada clase de conformidad” que indique los límites máximos soportados por el sistema operativo. El fallo ya ha sido reportado en <https://github.com/ciaa/Firmware/issues/267>.

Capítulo 3. Configuración y generación

Al ser OSEK-OS un sistema estático es necesario configurarlo. La cantidad de tareas, que prioridad tienen las mismas, el tamaño de pila que utilizan, etc. Para ello OSEK-VDX definió otro estándar llamado OSEK Implementation Language comúnmente llamado "OIL".

OIL es un lenguaje textual con una sintaxis similar al lenguaje C donde se indican las características del sistema operativo como ser: tareas, prioridades, pila, interrupciones etc., como se ilustra en el Ejemplo 3.1, "Ejemplo de configuración mediante OIL".

Ejemplo 3.1. Ejemplo de configuración mediante OIL

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

El Ejemplo 3.1, "Ejemplo de configuración mediante OIL" muestra la definición de una tarea llamada `InitTask` con una prioridad de 1, 128 bytes de pila y la cual se auto inicia al comienzo. Los detalles de la configuración se irán estudiando en el transcurso de este documento.

Una vez configurado el sistema operativo mediante el archivo OIL se debe generar el mismo. Durante la generación el sistema operativo interpreta los requerimientos del usuario y en función de los mismos arma un sistema operativo a medida para esa aplicación. De esta forma elementos tales como la cantidad de tareas, sus nombres y prioridades serán únicamente las definidas dentro del archivo de configuración.

En la Figura 3.1, "Configuración, generación, compilación y lindeo del sistema" se ilustra la generación del sistema operativo. La generación se lleva a cabo en función a la configuración provista en el archivo de configuración *OIL*. Este archivo puede ser editado de forma manual mediante un editor de texto o mediante una herramienta gráfica. El generado lee la configuración del *OIL* y ejecuta los templates. Los templates son archivos con una doble extensión, por ejemplo: `.h.php`, `.c.php` ó similar. Estos archivos están programados en `php` y generan un archivo de salida con la primera extensión. En la Sección 3.2, "Generación" se analiza la generación en más detalle.

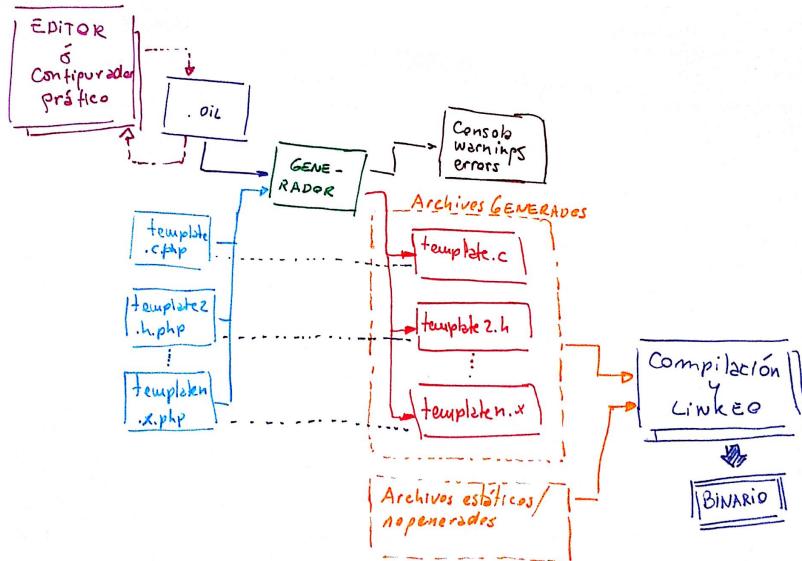


Figura 3.1. Configuración, generación, compilación y linkeo del sistema

Luego de haber generado el sistema operativo se podrá compilar y linkear el sistema operativo con la aplicación.

Cabe aclarar que OSEK-OS define en [OSEK-OIL] el formato de configuración, sin embargo no se define en el estándar como se crea el archivo .oil ni como se resuelve la generación en la Figura 3.1, "Configuración, generación, compilación y linkeo del sistema" marcado como *Editor o Configurador gráfico* y *Generador* respectivamente. En la CIAA el proceso de generación se resuelve mediante los templates programados en php, pero esto no es parte del estándar sino simplemente una implementación del mismo.

3.1. Configuración

En el CIAA-Firmware la configuración se debe realizar manualmente editando los archivos .oil. Sin embargo, existen herramientas gráficas que pueden ser utilizadas para simplificar la configuración. Generalmente las implementaciones comerciales de OSEK-OS incluyen un configurador gráfico. En un futuro la CIAA podría contar con uno también, por ejemplo si se implementase un plugin del IDE (que está basado en eclipse) para que los usuarios puedan configurar el sistema operativo y el Firmware en general de forma gráfica.

Otra opción posible sería utilizar algún otro producto ya sea o no de código abierto para generar los archivos OILs. Una opción podría ser gobx [<http://gobox.sourceforge.net/>]. Es importante tener en cuenta que *gobox* fue probado por última vez en conjunto con una versión preliminar del OSEK de la CIAA en 2008 y 2009, por lo que es probable que quien haga el intento tenga que corregir unos cuantos problemas antes de poder configurar el OSEK-OS de la CIAA mediante *gobox* de forma satisfactoria.



Figura 3.2. Captura de pantalla de gobx¹

En la Figura 3.2, “Captura de pantalla de gobx” se puede observar una captura de pantalla de gobx. En la misma se distingue la posibilidad de configurar entre otras cosas: Tareas, Interrupciones, Contadores, Alarmas, etc. del sistema operativo.

3.2. Generación

El proceso de generación no es parte del estándar y por ello cada implementación lo realiza de forma diferente. El OSEK-OS del CIAA-Firmware utiliza *php* para interpretar la configuración y generar el código.

En esta sección se ejemplifica la generación, para lo que se da por entendido que el lector cuenta con algunos mínimos conocimientos de *php*.

Si por ejemplo utilizamos el ejemplo *blinking* del CIAA-Firmware cuyo archivo de configuración OIL se encuentra en <https://github.com/ciaa/Firmware/tree/0.6.1/examples/blinking/etc/blinking.oil> para la versión 0.6.1.

¹Fuente: http://gobx.sourceforge.net/screenshots/ubuntu_snapshot_500_418.gif

Ejemplo 3.2. Extracto de la configuración de blinking del CIAA-Firmware

```

TASK InitTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    AUTOSTART = TRUE {
        APPMODE = AppModel;
    }
    STACK = 512;
    TYPE = EXTENDED;
    SCHEDULE = NON;
    RESOURCE = POSIXR;
    EVENT = POSIXE;
}

TASK PeriodicTask {
    PRIORITY = 2;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
    SCHEDULE = NON;
    RESOURCE = POSIXR;
    EVENT = POSIXE;
}

```

En el Ejemplo 3.2, “Extracto de la configuración de blinking del CIAA-Firmware” se puede observar que se definen dos tareas `InitTask` y `PeriodicTask` con 512 bytes de pila cada una. Si se llama al generador con esta configuración corriendo `make generate` el mismo retorna lo indicado en el Ejemplo 3.3, “Salida del generador”.



A quienes quieran investigar el `make` del CIAA-Firmware podrán usar el comando `make help` para obtener más detalles.

Ejemplo 3.3. Salida del generador

```

INFO: configuration file 1: examples/blinking/etc/blinking.oil1
INFO: list of files to be generated:
INFO: generated file 1: ./modules/rtos/gen/inc/Os_Internal_Cfg.h.php2
INFO: generated file 2: ./modules/rtos/gen/inc/Os_Cfg.h.php3
INFO: generated file 3: ./modules/rtos/gen/src/Os_Cfg.c.php4
INFO: generated file 4: ./modules/rtos/gen/src/Os_Internal_Cfg.c.php5
INFO: generated file 5: ./modules/rtos/gen/src/x86/Os_Internal_Arch_Cfg.c.php6
INFO: generated file 6: ./modules/rtos/gen/inc/x86/Os_Internal_Arch_Cfg.h.php7
INFO: output directory: ./out/gen
INFO: reading examples/blinking/etc/blinking.oil
INFO: generating ./modules/rtos/gen/inc/Os_Internal_Cfg.h.php to
    ./out/gen/inc/Os_Internal_Cfg.h8
INFO: generating ./modules/rtos/gen/inc/Os_Cfg.h.php to
    ./out/gen/inc/Os_Cfg.h9
INFO: generating ./modules/rtos/gen/src/Os_Cfg.c.php to
    ./out/gen/src/Os_Cfg.c10
INFO: generating ./modules/rtos/gen/src/Os_Internal_Cfg.c.php to
    ./out/gen/src/Os_Internal_Cfg.c11
INFO: generating ./modules/rtos/gen/src/x86/Os_Internal_Arch_Cfg.c.php to
    ./out/gen/src/x86/Os_Internal_Arch_Cfg.c12
INFO: generating ./modules/rtos/gen/inc/x86/Os_Internal_Arch_Cfg.h.php to
    ./out/gen/inc/x86/Os_Internal_Arch_Cfg.h13

```

En el Ejemplo 3.3, “Salida del generador” se puede observar que el archivo `examples/blinking/etc/blinking.oil` **1** es utilizado como entrada del generador, osea es el archivo que indica la configuración del sistema operativo. También se puede observar que los archivos de entrada a ser generados se denominan:

- `modules/rtos/gen/inc/Os_Internal_Cfg.h.php`**2**
- `modules/rtos/gen/inc/Os_Cfg.h.php`**3**

- modules/rtos/gen/src/Os_Cfg.c.php**4**
- modules/rtos/gen/src/Os_Internal_Cfg.c.php**5**
- modules/rtos/gen/src/x86/Os_Internal_Arch_Cfg.c.php**6**
- modules/rtos/gen/inc/x86/Os_Internal_Arch_Cfg.h.php**7**

y que los mismos generan los correspondientes archivos con el mismo nombre pero sin la extensión .php y en el directorio de salida (out/gen/...):

- out/gen/inc/Os_Internal_Cfg.h**8**
- out/gen/inc/Os_Cfg.h**9**
- out/gen/src/Os_Cfg.c**10**
- out/gen/src/Os_Internal_Cfg.c**11**
- out/gen/src/x86/Os_Internal_Arch_Cfg.c**12**
- out/gen/inc/x86/Os_Internal_Arch_Cfg.h**13**

Si por ejemplo se analiza el contenido de Os_Internal_Cfg.c.php y Os_Internal_Cfg.c que se puede ver en Ejemplo 3.4, “Generación de Os_Internal_Cfg.c.php”:

Ejemplo 3.4. Generación de Os_Internal_Cfg.c.php

```
$tasks = $config->getList("/OSEK", "TASK");
foreach ($tasks as $task)
{
    print "/** \brief $task stack */\n";
    print "#if ( x86 == ARCH )\n";
    print "uint8 StackTask" . $task . "[" . $config->getValue("/OSEK/" . $task,
        "STACK") . " + TASK_STACK_ADDITIONAL_SIZE];\n";
    print "#else\n";
    print "uint8 StackTask" . $task . "[" . $config->getValue("/OSEK/" . $task,
        "STACK") . "];\n";
    print "#endif\n";
}
```

```
/** \brief InitTask stack */
#if ( x86 == ARCH )
uint8 StackTaskInitTask[512 + TASK_STACK_ADDITIONAL_SIZE];
#else
uint8 StackTaskInitTask[512];
#endif
/** \brief PeriodicTask stack */
#if ( x86 == ARCH )
uint8 StackTaskPeriodicTask[512 + TASK_STACK_ADDITIONAL_SIZE];
#else
uint8 StackTaskPeriodicTask[512];
#endif
```

En el Ejemplo 3.4, “Generación de Os_Internal_Cfg.c.php” se puede observar cómo el código en php realiza un bucle sobre todas las tareas y genera la pila necesaria para cada una de ellas. En el código generado se puede observar que se generan 2 pilas, uno para la tarea InitTask llamada StackTaskInitTask y otra para la tarea PeriodicTask llamada StackTaskPeriodicTask.

Se puede observar que al ser un sistema operativo configurado y luego generado el mismo utiliza únicamente los recursos que son necesarios para el usuario en la configuración específica que le da en cada aplicación. Gracias a esto el consumo de RAM/ROM así como el tiempo de ejecución se pueden ver afectados por la configuración. El sistema solo generará las partes del código necesarias para la aplicación. El mismo Scheduler por ejemplo incluirá partes de su código dependiendo de que sean necesarias o no en cada aplicación.



Se puede observar que la cantidad de memoria reservada para la pila depende de la arquitectura del procesador. Para el entorno simulado x86 (ya sea Windows o Linux) se agrega una cierta cantidad de bytes de pila. Esto se debe a que para la simulación de interrupciones en Windows y Linux es necesaria una cantidad adicional. Más información al respecto se puede encontrar en <https://github.com/ciaa/Firmware/issues/165>.

Una vez generado el código y con una aplicación como la del ejemplo *blinking* que se ve a continuación que se puede encontrar en: <https://github.com/ciaa/Firmware/blob/0.6.1/examples/blinking/src/blinking.c>.

```
/* ... */
int main(void)
{
    /* Starts the operating system in the Application Mode 1 */
    /* This example has only one Application Mode */
    StartOS(AppModel);

    /* StartOs shall never returns, but to avoid compiler warnings or errors
     * 0 is returned */
    return 0;
}

/* ... */

TASK(InitTask)
{
    /* ... */

    /* terminate task */
    TerminateTask();
}

/* ... */

TASK(PeriodicTask)
{
    /* ... */

    /* terminate task */
    TerminateTask();
}
```

Como se puede observar desde `main` se inicia el sistema operativo, el mismo activa la tarea `InitTask` de forma automática. Por último la tarea termina llamando a `TerminateTask`. Mientras que la tarea `PeriodicTask` se seguirá ejecutando de forma periódica hasta que se detenga el sistema.

3.3. Ventajas y desventajas de un sistema operativo generado

En Sección 1.3, “Dinámico vs. estático” se trató el tema, sin embargo, luego de haber visto un poco más en detalle de qué se trata la configuración y generación del sistema operativo es interesante ver unos ejemplos.

Una buena implementación de OSEK-OS deberá realizar un extenso análisis sobre la configuración y realizar una generación que manteniendo la compatibilidad con OSEK-OS logre ofrecer al usuario el menor costo en tiempo de ejecución, RAM y ROM. De este modo una implementación de OSEK-OS puede ofrecer al usuario un sistema operativo con pocos kbytes de RAM y ROM pero manteniendo una gran versatilidad.

Sin embargo, es importante destacar que no todas son ventajas. Realizar la validación de un sistema operativo que se genera en función de su configuración, y existiendo una gran cantidad de posibles configuraciones, no es una tarea sencilla. No es posible verificar todas las configuraciones. Se deberá seleccionar un subconjunto y esperar que los resultados sean válidos para todas las configuraciones no ensayadas.

En las siguientes subsecciones se verán algunos ejemplos de posibles optimizaciones.

3.3.1. Recursos

Como se verá en Capítulo 5, *Recursos* OSEK-OS ofrece la utilización de recursos para el manejo de dispositivos o elementos únicos en el sistema. Otros sistemas ofrecen para los mismos fines mutexs y/o semáforos. A diferencia de otros sistemas operativos en OSEK-OS se debe indicar en la configuración qué recursos utilizan cada una de las tareas e interrupciones. Gracias a esto una buena implementación de OSEK-OS puede tener varios grados de optimización.

En el caso de que ninguna tarea tenga recursos, se puede directamente eliminar la función del núcleo (kernel) para el manejo de los mismos. Ya que siendo que el usuario no configuró ningún recurso sería un error cualquier llamado a estas interfaces.

Si por el contrario el usuario sí configura recursos, pero el generador detecta que ningún recurso es compartido entre tareas y/o interrupciones, las funciones serán necesarias para que el código compile, ya que el usuario las va a llamar. Sin embargo al utilizar cada tarea un recurso distinto las mismas se pueden implementar sin ninguna funcionalidad, ahorrando tiempo de ejecución y recursos. De esta forma la aplicación corre sobre un OSEK-OS y el sistema le ofrece una implementación que se adapta a sus necesidades y maximiza el desempeño.

En el caso de que un recurso sea compartido entre tareas e interrupciones, la implementación deberá ser la más completa, toda la funcionalidad deberá ser generada y compilada y se utilizarán más recursos del sistema.

Estos fueron tres ejemplos de optimización en el manejo de recursos, sin embargo no son todas las posibilidades existentes, se pueden realizar más y más complejas optimizaciones. El OSEK-OS de la CIAA ofrece algunas de estas optimizaciones, pero no todas.

3.3.2. Scheduler y Tareas

Otro buen ejemplo de como OSEK-OS puede optimizar los recursos gracias a que se trata de un sistema estático es su *Scheduler*. El funcionamiento exacto será discutido en el Capítulo 4, *Tareas*. Sin embargo hay algunos puntos que se pueden observar ya en la Ejemplo 3.2, “Extracto de la configuración de blinking del CIAA-Firmware”. En la configuración se indica por ejemplo *prioridad*, *tipo de scheduling*, *tamaño de la pila*, *cantidad de activaciones* así como otros parámetros.

Gracias a esto el Scheduler del sistema operativo podrá adaptarse a las necesidades del usuario.

Conociendo la cantidad de tareas y sus prioridades se podrán generar las FIFOs para el scheduling. El Scheduler de OSEK utiliza una FIFO por prioridad. El largo de cada FIFO será la suma de las tareas con una prioridad determinada².

Por otro lado en OSEK-OS existen tareas que pueden ser interrumpidas (*SCHEDULE = FULL*) y otras que no (*SCHEDULE = NON*). En la configuración se pueden dar tres opciones:

- que todas las tareas puedan ser interrumpidas,
- que ninguna de ellas ó
- una mezcla de ambos tipos de scheduling.

En función a esto el sistema operativo puede contener una parte del código, la otra o ser un scheduling con todas las posibilidades y que se decida en tiempo de ejecución cual tarea puede o no puede ser interrumpida.

En un sistema con ambos tipos de tareas (las que se pueden y las que no interrumpir) será necesario almacenar en alguna variable interna del sistema por cada tarea esta información para que el Scheduler lo pueda evaluar en tiempo de ejecución. Si el sistema tuviese sólo un tipo de tareas (todas pueden ser interrumpidas o ninguna) el sistema puede optimizar esta variable ahorrando RAM/ROM y tiempo de ejecución ya que no será necesario hacer ningún control en tiempo de ejecución.

²Para ser exactos se debe tener en cuenta también la cantidad de activaciones de una tarea, una tarea con n activaciones necesita hasta n espacios reservados en la FIFO

Capítulo 4. Tareas

En OSEK-OS las tareas deben ser definidas en la configuración, por lo que no es posible crear tareas de forma dinámica como en sistemas operativos de escritorio. En el momento de la generación, antes de la compilación se genera el código del sistema operativo y se definen la cantidad de tareas y sus características.

Cada tarea en OSEK-OS se define de forma similar a una función (que no recibe parámetros y retorna void) utilizando el macro `TASK(TaskName)`. Para utilizar este macro como cualquier tipo, macro o función de OSEK-OS se debe incluir `os.h`. A diferencia de una función, una tarea debe terminar utilizando la interfaz `ChainTask` ó `TerminateTask`. También podría una tarea no terminar nunca y quedar en un ciclo infinito, ver Sección 4.1, “Sobre el while(1) y tareas infinitas”.

Ejemplo 4.1. Definiendo una tarea

```
TASK(InitTask)
{
    /* ... */
    ChainTask(OtherTask);
}

TASK(OtherTask)
{
    /* ... */
    TerminateTask();
}
```

En Ejemplo 4.1, “Definiendo una tarea” se puede ver como se definen 2 tareas llamadas `InitTask` y `OtherTask` utilizando el macro `TASK`. El sistema operativo implementa este macro y crea una función por cada tarea. En el caso del CIAA-Firmware el macro `TASK` se define como `#define TASK(name) void OSEK_TASK_ ## name (void)`. Como se define este macro es específico de cada implementación y no es parte del estándar, sin embargo puede ser de interés en el momento de querer indicar un breakpoint en el código, en ese caso en este ejemplo se deben buscar los símbolos `OSEK_TASK_InitTask` y `OSEK_TASK_OtherTask`.



Todas las tareas configuradas deben ser definidas utilizando este mecanismo. Las tareas no deben ser declaradas, la declaración es realizada por el OSEK-OS.

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. Sobre el while(1) y tareas infinitas

En los sistemas operativos de escritorio las aplicaciones corren normalmente por un tiempo indeterminado hasta ser terminadas por los usuarios. En los sistemas de tiempo real, como así también en OSEK-VDX, es una buena práctica que las tareas sean activadas cuando tienen que realizar algún cometido (en base a eventos) y sean finalizadas al terminar el mismo. Tareas con un `while(1)` ó `while(exitFlag)` no son una práctica recomendable.

Cabe tener en cuenta que durante el tiempo que una tarea está siendo ejecutada la misma consume recursos que no podrán ser utilizados por otra tarea, por ejemplo la *pila*. Si en cambio la tarea es terminada los recursos podrían ser utilizados por otra tarea.

Además es importante destacar que mientras una tarea está siendo ejecutada OSEK-OS no ejecutará ninguna otra tarea con una prioridad igual o menor. Si además la tarea siendo ejecutada es *NONPREEMPTIVE* el sistema no ejecutará ninguna otra tarea. Por ello la utilización de `while`s por tiempo indeterminado tienen que ser evitadas o cuidadosamente analizados.

Si una tarea debe ser ejecutada constantemente la misma podría terminarse y volver a activarse, siendo esto una mejor opción que la utilización de `while`. Para ello se puede utilizar la interfaz `ChainTask`.

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.

4.2. Estados

Cada tarea en OSEK-OS se encuentra siempre en uno de los siguientes cuatro 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 decidió esperar la ocurrencia 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 momentáneamente desactivada.

La figura Figura 4.1, “Posibles estados de una tarea” esquematiza los estados recién mencionados y las respectivas transiciones.

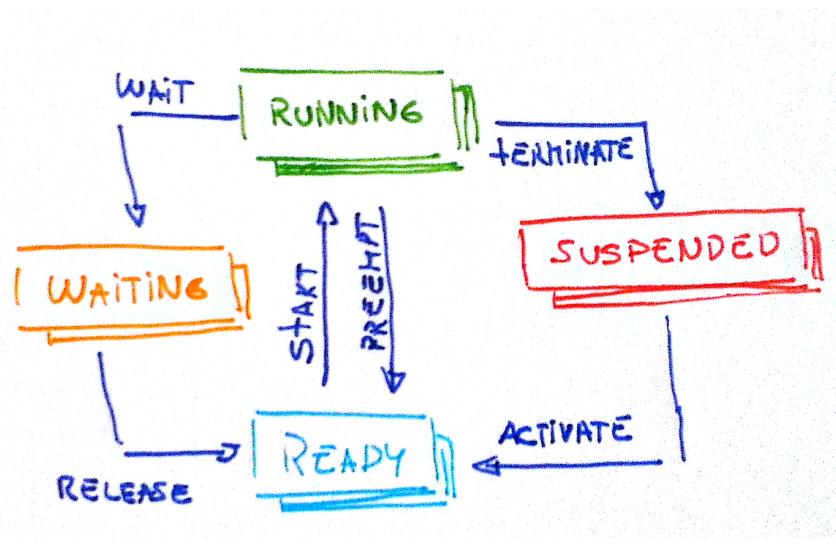


Figura 4.1. Posibles estados de una tarea

Transición	Estado anterior	Estado futuro	Descripción
<i>activate</i>	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: <ul style="list-style-type: none"> ActivateTask ChainTask
<i>start</i>	ready	running	Una tarea es llevada al estado running de forma automática cuando es la tarea de mayor prioridad en la lista de ready.
<i>wait</i>	running	waiting	La tarea es llevada a este estado para esperar la ocurrencia de un evento. Esto se puede lograr con la siguiente interfaz: <ul style="list-style-type: none"> WaitEvent
<i>release</i>	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: <ul style="list-style-type: none"> SetEvent
<i>preempt</i>	running	ready	Una tarea que se encontraba 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 una tarea con scheduling police NON llama a la siguiente interfaz: <ul style="list-style-type: none"> Schedule

Transición	Estado anterior	Estado futuro	Descripción
<i>terminate</i>	running	suspended	La tarea termina su ejecución, esto lo puede llevar a cabo con las siguientes interfaces: <ul style="list-style-type: none"> • ChainTask • TerminateTask

Tabla 4.1. Transiciones entre los estados de las tareas

4.3. Tipos de Tareas

El sistema operativo OSEK-OS existen dos tipos de tareas:

- *BASIC*: son aquellas tareas que no tengan eventos y por ende carezcan del estado waiting.
- *EXTENDED*: son aquellas tareas que tienen eventos y por ende pueden esperar hasta que uno o más eventos ocurran.

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;

```

Este ejemplo contiene dos tareas TaskA y TaskB. Ambas tareas son del tipo *extended*, la tarea TaskA tiene dos eventos: Event1 y Event2 mientras que la tarea TaskB tiene únicamente un evento: Event2.

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

- ClearEvent
- GetEvent
- SetEvent
- WaitEvent

Un ejemplo de código con estas dos tareas podría ser similar a:

```
TASK(TaskA) {
    EventMaskType Events;

    /* do something */

    WaitEvent(Event1 | Event2);
    GetEvent(TaskA, &Events);
    ClearEvent(Events);

    /* process events */

    TerminateTask();
}

TASK(TaskB) {
    /* perform some actions */

    /* inform TaskA by setting an event */
    SetEvent(TaskA, Event1);

    TerminateTask();
}
```

El ejemplo implementa dos tareas. TaskA corre y decide esperar hasta que Event1 y/o Event2 eventos ocurran. TaskB realiza alguna operación y luego informa a la tarea TaskA mediante el evento Event1.

Al ocurrir el evento TaskA no sabe que evento ha ocurrido ya que ha llamado `WaitEvent` con dos eventos. Por ello será necesario utilizar la función `GetEvent` para saber que evento/s ocurrieron y por último hacer `ClearEvent` para borrar los eventos ocurridos. A continuación falta implementar el código que procesa 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);
    ClearEvent(Event1);

    /* process Event1 */

    TerminateTask();
}
```

En este caso no es necesario llamar a la función `GetEvent` ya que sólo el `Event1` puede haber ocurrido.

4.3.1. Múltiples Activaciones

Las tareas básicas pueden ser configuradas para tener múltiples activaciones. Esto implica que las mismas pueden ser insertadas varias veces en la lista de tareas *ready*(a ser ejecutadas) y posteriormente ejecutadas según corresponda dependiendo del scheduler.

Por ejemplo si una tarea básica es utilizada para procesar un buffer de recepción de datos se podría generar el siguiente código.

Ejemplo 4.2. Activación múltiple de una tarea

```
ISR(RxIndication)
{
    /* copy data to
    intermediate buffer */

    ActivateTask(
        ProcessRxIndication); 1
}

TASK(ProcessRxIndication)
{
    /* process input data */
    TerminateTask();
}
```

```
TASK ProcessRxIndication {
    PRIORITY = 3;
    ACTIVATION = 10; 1
    STACK = 512;
    TYPE = BASIC; 2
    EVENT = Event1;
}
```

- 1** La tarea ProcessRxIndication podrá ser activada hasta 10 veces.
2 Únicamente las tareas *básicas* pueden tener un valor de activación mayor a 1.

- 1** En cada ocurrencia de la interrupción RxIndication la tarea ProcessRxIndication es activada.

Únicamente las tareas *básicas* pueden tener un valor de activación mayor a 1.



Cada vez que una tarea es activada se ingresa en una FIFO para ser procesada. El sistema operativo tiene una FIFO por prioridad.

Si tres tareas TaskA, TaskB y TaskC con las prioridades 5, 4 y 3 respectivamente, y siendo TaskB y TaskC tareas básicas con una ACTIVATION = 5 ; son activadas como en Figura 4.2, "Activación Múltiple".

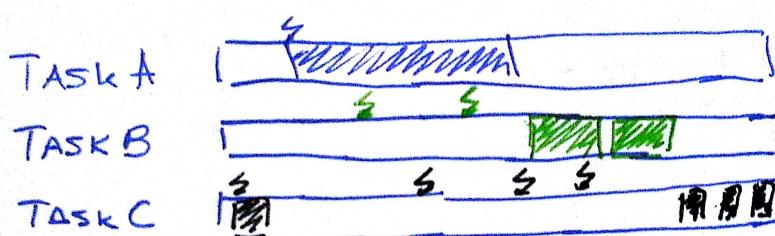


Figura 4.2. Activación Múltiple

La primera activación de TaskC es procesada inmediatamente, al igual que la primera activación de TaskA. Mientras que las activaciones de TaskB no pueden ser procesadas inmediatamente ya que TaskA se encuentra corriendo. Por ello las activaciones son almacenadas en una FIFO (interna del RTOS) para luego ser procesadas. Lo mismo sucede con la segunda, tercera y cuarta activación de TaskC.



No se debe superar la cantidad máxima de activaciones, en caso contrario el sistema retornará E_OS_LIMIT y la tarea no será activada.

En caso de existir dos tareas de la misma prioridad con una activación máxima mayor a 1 el sistema las ejecutará secuencialmente según su orden de activación, por ello se habla de una FIFO (first in first out).

4.4. Prioridades

En OSEK-OS las prioridades de las tareas se definen de forma estática en el OIL. La prioridad es un número entero entre 0 y 255. Mayor sea el número mayor es la prioridad. Si dos tareas tienen la misma prioridad son ejecutadas según su orden de activación. Una tarea que se encuentra corriendo nunca va a ser interrumpida por una de igual o menor prioridad.



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 terminar o pasar al estado waiting, hasta tanto se continuara con su ejecución.

4.4.1. Orden de ejecución

OSEK utiliza una FIFO por prioridad para almacenar las tareas en estado *ready* a ser ejecutadas. Supongamos que tenemos 1 sistema con 5 tareas todas de distinta prioridad (10, 8, 4, 3, 0) como el de la Figura 4.3, "Scheduling en BCC1 y ECC1".

Se puede observar que las FIFOs en la Figura 4.3, "Scheduling en BCC1 y ECC1" son de un único elemento dado que estamos en un sistema *BCC1, ECC1* (una tarea por prioridad y sin múltiple activación).

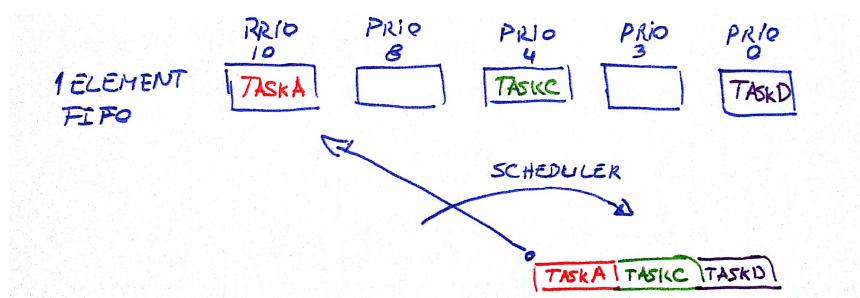


Figura 4.3. Scheduling en BCC1 y ECC1

Las tareas TaskA, TaskC y TaskD (de prioridades 10, 4 y 0 respectivamente) están en estado *ready* para ser ejecutadas. El scheduler irá ejecutando las tareas según su prioridad, pasando a la prioridad inferior cuando no existan más tareas en estado *ready* con una prioridad mayor. De no activarse mas tareas se ejecutarán en el siguiente orden: TaskA, TaskC y por último TaskD.

En un segundo ejemplo en la Figura 4.4, "Scheduling en BCC2 y ECC2" se trata de un sistema nuevamente con 5 tareas de prioridades (10, 7, 5, 2 y 1). Pero hay algunas tareas que comparten la prioridad y algunas también pueden ser activadas más de una vez.

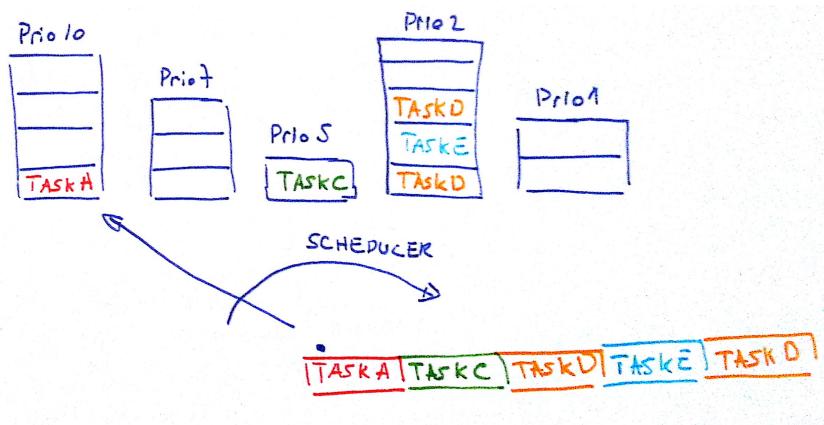


Figura 4.4. Scheduling en BCC2 y ECC2

Durante la generación y en base a la configuración (archivo OIL) el sistema puede calcular el largo necesario para cada FIFO. Simplemente se suman todas las activaciones por prioridad dando en este caso cuatro para la prioridad 10, tres para la prioridad 7, uno para la prioridad 5, cinco para la prioridad 2 y dos para la prioridad 1.

En este caso si procede a ejecutar las tareas en estado *ready* resultará el siguiente orden: TaskA, TaskC, TaskD, TaskE y por último nuevamente TaskD.

Como se puede observar la tarea TaskD tiene una activación igual o mayor a 2 y se encontraba activada dos veces.



Como se puede observar en la Figura 4.3, "Scheduling en BCC1 y ECC1" y Figura 4.4, "Scheduling en BCC2 y ECC2" las prioridades son relativas. Carece de importancia si una la diferencia entre los valores absolutos de prioridad, dos tareas con prioridad 10 y una serán ejecutadas de la misma forma si tuvieran las prioridades 3 y 2.

4.5. Tipos de Scheduling

OSEK-OS ofrece dos tipos de scheduling:

1. *NON PREEMPTIVE*: son tareas que no interrumpidas por aquellas de mayor prioridad, salvo que la misma tarea llame a `Schedule`, pase al estado `waiting` llamando a `WaitEvent` o terminen.
2. *PREEMPTIVE*: son tareas que pueden ser interrumpida en cualquier momento cuando se encuentre una tarea de mayor prioridad en la lista `ready`.

Para configurar una tarea como *NON PREEMPTIVE* o *PREEMPTIVE* se utiliza el parámetro `SCHEDULE` del OIL. 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 *PREEACTIVE*. 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 *PREEACTIVE* 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.

4.5.1. Tareas de corta ejecución

Muchas veces una tarea de muy corta duración se configura como *NON PREEMPTIVE*, de esta forma se evita que sea interrumpida. El tiempo de ejecución llevaría el cambio de contexto sería de duración similar al tiempo que dura la tarea en ejecutarse completamente. Configurando este tipo de tareas como *NON PREEMPTIVE* se evita esta perdida de tiempo innecesaria.

4.5.2. Sistema determinista

En muchos casos es recomendable usar tareas *NON PREEMPTIVE* para hacer el sistema más determinista. 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 cambiará de tarea. Llamar a la interfaz `Schedule` desde una tarea *PREEMPTIVE* se puede hacer, pero carece de sentido. Como las tareas *PREEMPTIVE* son interrumpida de todas maneras, no es necesario llamar al `Schedule`.

4.5.3. Puntos de Scheduling

Puntos de scheduling son los puntos en que el sistema operativo analiza la lista de tareas y de haber una de mayor prioridad procede a ejecutarla. Estos puntos están claramente definidos en el estándar y es importante conocerlos.

El primer punto de Scheduling es al finalizar la inicialización del Sistema Operativo llamando a `StartOS`. Una vez iniciado el sistema operativo los puntos de Scheduling pueden ser diferenciados según la tarea que este corriendo ya sea la misma *NON-PREEMPTIVE* o *PREEMPTIVE*.

4.5.3.1. Puntos de Scheduling en una tarea NON-PREEMPTIVE

- Al llamar a la interfase `Schedule` cuando retorna `E_OK`.

4.5.3.2. Puntos de Scheduling en una tarea PREEMPTIVE

- Al finalizar un llamado a la interfase `ActivateTask` que retorna `E_OK`.
- Al finalizar un llamado a la interfase `ChainTask` que no retorna.
- Al finalizar un llamado a la interfase `TerminateTask` que no retorna.
- Al finalizar un llamado a la interfase `ReleaseResource` que retorna `E_OK`.
- Al finalizar un llamado a la interfase `SetEvent` que no retorna.
- Al expirar una alarma que activa una tarea.
- Al terminar la ejecución de una ISR de categoría 2

Capítulo 5. Recursos

Generalmente los recursos son escasos y/o únicos, independientemente e que sea un sistema de tiempo real o no. Si dos aplicaciones utilizan simultáneamente un recurso como ser un bus de datos o un puerto pero no se sincronizan entre sí, el resultado puede ser indeterminado. Por ello para acceder a los recursos se utilizan en los sistemas convencionales *semáforos* o *mutex* para poder accederlos desde varias tareas sin generar interferencias ni colisiones entre las mismas.

Sin embargo la utilización de semáforos y mutex tienen dos problemas:

- *inversión de prioridades*: se da cuando una tarea de prioridad relativamente alta es bloqueada al requerir el uso de un recurso que está siendo utilizado por una tarea de prioridad menor que fue interrumpida para ejecutar la tarea de mayor prioridad.
- *deadlocks*: se da cuando dos tareas utilizan dos recursos (o semáforos) en orden inverso bloqueándose mutuamente.

OSEK-OS ofrece una solución para el acceso a recursos que evita estos ambos problemas.



Al terminar este capítulo le recomendamos releer la Sección 3.3.1, "Recursos" con los nuevos conocimientos adquiridos.

5.1. Inversión de prioridades

En la Figura 5.1, "Inversión de prioridades" hay cuatro tareas: TaskA, TaskB, TaskC y TaskD donde TaskA tiene mayor prioridad que TaskB, TaskB tiene mayor prioridad que TaskC y TaskC tiene mayor prioridad que TaskD. Siendo los valores exactos de las prioridades irrelevantes, únicamente de importancia las prioridades relativas entre las tareas.

Todas estas tareas comparten la utilización de un mismo recurso. En el ejemplo de la figura TaskD 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: TaskA, TaskB y TaskC. Como TaskD es la tarea con menor prioridad su ejecución es pausada y se procede a ejecutar las otras tareas.

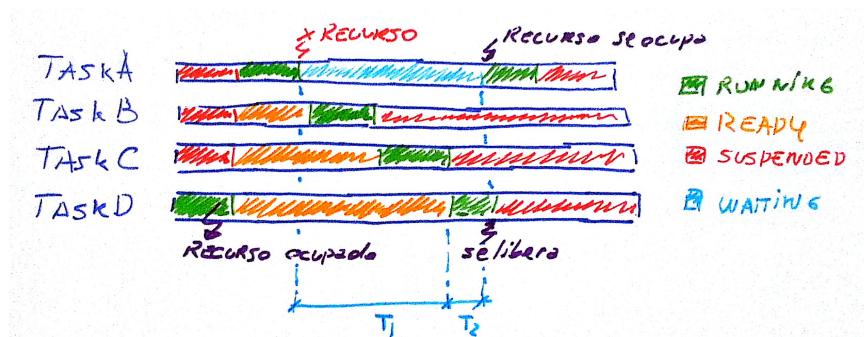


Figura 5.1. Inversión de prioridades

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

Este ejemplo muestra la grave situación en la que puede caer el sistema, sobre todo en sistemas críticos. La ejecución de una tarea de prioridad alta TaskA es retrasada por la ejecución de las tareas TaskB y TaskC ambas de menor prioridad dado que comparte un recurso con TaskA. A este escenario se lo denomina *inversión de prioridades*.

En la figura *T1* representa el tiempo durante el cual TaskA fue retrasada debido a la inversión de prioridades.

El tiempo *T2* no se puede computar conjunto con *T1* ya que en este caso el recurso en discusión se encuentra ocupado e independientemente de la inversión de prioridades TaskA no podría haberse ejecutado sin el retraso debido a *T2*.

5.2. Deadlock

Otro problema común durante la utilización de recursos son los deadlocks. En la Figura 5.2, "Deadlocks" se puede observar como una TaskA comienza la utilización de un *Recurso 1*, por alguna razón es interrumpida y se procede a la ejecución de una segunda tarea TaskB que comienza a utilizar el *Recurso 2*, cuando se continua la ejecución de TaskA el *Recurso 2* sigue ocupado por TaskB en esta situación cuando TaskA intenta acceder al *Recurso 2* es interrumpida a la espera del mismo que está siendo utilizado por TaskB. Se continuara la ejecución de TaskB. Si esta por último intenta acceder a *Recurso 1* se produce un *Deadlock*. Ambas tareas quedan bloqueadas a la espera de la liberación del recurso que está siendo utilizado por la otra tarea.

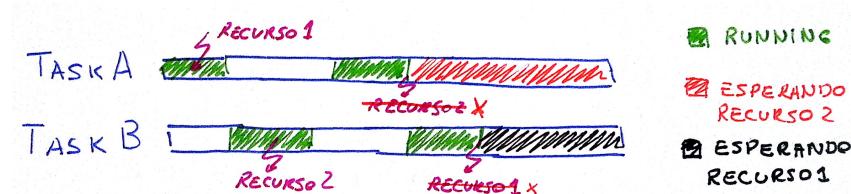


Figura 5.2. Deadlocks

5.3. OSEK Priority Ceiling Protocol

OSEK-OS provee una solución más adecuada para estas situaciones mediante la utilización de *recursos*. Veamos el mismo ejemplo de la Figura 5.1, "Inversión de prioridades" pero ahora utilizando los mecanismos de OSEK-OS, lo mismo se ejemplifica en la Figura 5.3, "OSEK Priority Ceiling Protocol". Dado que TaskA y TaskD comparten un mismo recurso, OSEK-OS

le otorga al *recurso* en sí mismo una prioridad. Esta prioridad será mayor a la prioridad de la tarea más alta que lo utiliza y menor que la prioridad de la siguiente tarea de mayor prioridad que no utiliza el recurso. Como el recurso es utilizado por TaskA que tiene la mayor prioridad el recurso tiene una prioridad aún mayor que TaskA.

En caso de existir una Task0 con mayor prioridad a TaskA y en caso de la misma no utilizar el recurso, el recurso tendría una prioridad mayor a TaskA y menor a Task0.

TaskD corre con una prioridad relativamente baja, sin embargo cuando le pide al sistema operativo acceso al *recurso* el sistema operativo le otorga momentáneamente y mientras utilice el *recurso* una prioridad superior a la de TaskA, es decir se le asigna la prioridad del *recurso*. Como en el ejemplo anterior mientras que TaskD utiliza el recurso se activan TaskA, TaskB y TaskC. Sin embargo ninguna de estas nuevas tareas es ejecutada a pesar de tener mayor prioridad que TaskD. Esto se debe a que TaskD utiliza momentáneamente el recurso y hasta que no lo libere tiene una prioridad mayor.

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

En la Figura 5.3, "OSEK Priority Ceiling Protocol" se observar claramente que la ejecución de TaskA solo se retrasa el tiempo que dura la utilización del recurso (T_2 en la figura anterior) y no como en el ejemplo anterior donde se retrasaba innecesariamente durante un tiempo T_1 mayor.

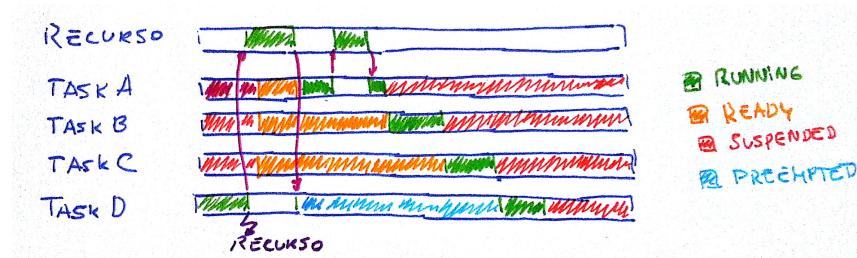


Figura 5.3. OSEK Priority Ceiling Protocol

Para la utilización de recursos los mismos deben ser declarados en OIL e indicado que recursos van a ser utilizados por cada tarea, como se muestra en el Ejemplo 5.1, "Definición de recursos en la configuración".

Ejemplo 5.1. Definición de recursos en la configuración

```

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;
}

RESOURCE Res1;

RESOURCE Res2;

```

El ejemplo muestra la configuración de tres tareas con dos recursos: Res1 y Res2. El recurso Res1 tendrá prioridad equivalente a 7,5 ya que es utilizado por TaskA y TaskB. Mientras que el recurso Res2 tendrá una prioridad 5,5 ya que es utilizado por TaskA y TaskC.

Desde el código se puede acceder a los recursos mediante las siguientes 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 7.5 */

    ReleaseResource(Res1);
    TerminateTask();
}

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

    /* perform actions using resource 2 */
    /* during this section TaskA has priority 5.7 */

    ReleaseResource(Res2);
    TerminateTask();
}

```

5.4. El Scheduler como recurso

Hay un recurso que siempre se encuentra disponible para las tareas y es el RES_SCHEDULER. Mediante la utilización de este recurso una tarea PREEMPTIVE (ver Sección 4.5, “Tipos de Scheduling”) puede bloquear a cualquier otra mientras se encuentra realizando un acción durante la cual no desea ser interrumpida. De esta forma es posible para una tarea PREEMPTIVE transformándose durante la utilización de este recurso en una tarea NON PREEMPTIVE.

Ejemplo 5.2. Utilizando RES_SCHEDULER

```
TASK(DemoTask)
{
    /* non critical code */

    GetResource(RES_SCHEDULER);

    /* critical code */

    ReleaseResource(RES_SCHEDULER);

    /* more no critical code */

    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    SCHEDULE = FULL;
}
```

En el Ejemplo 5.2, “Utilizando RES_SCHEDULER” se puede observar que la tarea DemoTask ha sido configurada como PREEMPTIVE (ver Sección 4.5, “Tipos de Scheduling”), y por lo tanto podrá utilizar el recurso RES_SCHEDULER. La utilización del recurso RES_SCHEDULER desde una tarea NON PREEMPTIVE carece de sentido, la misma de por sí no será interrumpida por ninguna otra independientemente de que utilice o no el recurso RES_SCHEDULER.



El OSEK-OS de la CIAA en la versión 0.6.1 no soporta el uso de RES_SCHEDULER.

5.5. Recursos Internos

Además de los recursos ya discutidos OSEK-OS ofrece la posibilidad de definir *recursos internos*. Estos recursos no se ocupan y liberan mediante la utilización de GetResource y ReleaseResource sino que son automáticamente reservados y liberados por el sistema al comenzar y finalizar una tarea.

Mediante la utilización de *recursos internos* se le permite al usuario crear grupos de tareas cooperativas las cuales utilizan el mismo *recurso interno*. De esta forma toda las tareas que utilicen un mismo recurso interno se comportan entre si como NON PREEMPTIVE.

Como se puede ver en el ejemplo Ejemplo 5.3, “Utilización de recursos internos” la tarea DemoTask1 durante su procesamiento no será nunca interrumpida por DemoTask5, a pesar de esta tener esta una prioridad 5 y esta ser mayor a la prioridad de DemoTask1 que es 1. Tanto DemoTask1 como DemoTask3 tendrán una prioridad de 5.5 al correr (ver Sección 5.3,

“OSEK Priority Ceiling Protocol”). Sin embargo DemoTask1 sí podrá ser interrumpida por DemoTask4 que tiene una prioridad de 6, ya que esta no utiliza el recurso interno.

Ejemplo 5.3. Utilización de recursos internos

```

TASK(DemoTask1)
{
    /* run some code which is
     * critical between DemoTask1
     * and DemoTask3 which both
     * have the same internal
     * resource.
    */

    TerminateTask();
}

TASK DemoTask1 {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    SCHEDULE = FULL;
    RESOURCE = ResInt1;
}

TASK DemoTask2 {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    SCHEDULE = FULL;
}

TASK DemoTask3 {
    PRIORITY = 5;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    SCHEDULE = FULL;
    RESOURCE = ResInt1;
}

TASK DemoTask4 {
    PRIORITY = 6;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    SCHEDULE = FULL;
    RESOURCE = ResInt1;
}

RESOURCE ResInt1 {
    RESOURCEPROPERTY = INTERNAL;
}

```

DemoTask3 sí podrá interrumpir DemoTask1 si esta llama a Schedule cuando DemoTask3 se encuentre en el estado ready (ver Sección 4.2, “Estados”).

De esta forma es posible armar grupos de tareas que no se interrumpan entre sí, o se interrumpan únicamente de forma explícita cuando el usuario llama a Schedule.



En el ejemplo Ejemplo 5.3, “Utilización de recursos internos” se puede ver una tarea DemoTask2 que no utiliza el recurso interno ResInt1. Esta tarea no podrá interrumpir nunca a ninguna de las tareas que utiliza ResInt1 (que tiene una prioridad de 5.5). Pero si la misma está en estado de ready simultáneamente con DemoTask1 se procederá a ejecutar DemoTask2 ya que el recurso interno es recién reservado cuando la tarea pasa a estado Running.



El OSEK-OS de la CIAA en la versión 0.6.1 no soporta el uso de recursos internos.

5.6. Recursos Linkeados

Al *linkearse* dos recursos se los une internamente de modo que para el sistema operativo tienen diferentes nombres pero son tratados como si fuesen el mismo recurso.

En el Ejemplo 5.4, “Utilización de recursos linkeados” se pueden ver dos tareas TaskA y TaskB que utilizan respectivamente Res1 y Res2. El sistema operativo trata ambos recursos como si fuesen el mismo.

Ejemplo 5.4. Utilización de recursos linkeados

```

TASK(TaskA)
{
    /* ... */
    GetResource(Res1);
    /* critical section */
    ReleaseResource(Res1);
    TerminateTask();
}

TASK(TaskB)
{
    /* ... */
    GetResource(Res2);
    /* critical section */
    ReleaseResource(Res2);
    TerminateTask();
}

TASK TaskA {
    /* ... */
    RESOURCE = Res1;
};

TASK TaskB {
    /* ... */
    RESOURCE = Res2;
}

RESOURCE Res1;

RESOURCE Res2 {
    RESOURCEPROPERTY = LINKED {
        LINKEDRESOURCE = Res1;
    }
};

```

La utilización de recursos linkeados puede ser de interés cuando se integran tareas que fueron inicialmente implementadas para diferentes sistemas y en un nuevo proyecto deben convivir. Así se puede modificar la configuración y mantener el código portable entre múltiples proyectos.



El OSEK-OS de la CIAA en la versión 0.6.1 no soporta el uso de recursos linkeados.

5.7. Recursos en interrupciones

En lo discutido sobre recursos hasta aquí, no se habló de la posibilidad de compartir un recurso entre una (o más) tarea(s) y una (o más) interrupción(s). Esto podría ser necesario si por ejemplo se desease procesar datos de una variable global o recursos de hardware desde una tarea y una interrupción. Para este fin y para evitar problemas de sincronización se podrían utilizar las interfaces: DisableAllInterrupts y EnableAllInterrupts ó SuspendAllInterrupts y ResumeAllInterrupts ó SuspendOSInterrupts y ResumeOSInterrupts sin embargo estas interfaces deshabilitarían y habilitarían todas las interrupciones del sistema o todas las de categoría 2 (ver Capítulo 7, *Interrupciones*). Esta solución es mucho

más invasiva de lo que el usuario realmente necesita, que es deshabilitar únicamente las interrupciones que pueden llegar a utilizar el recurso obtenido.

OSEK-OS describe una funcionalidad opcional que permite compartir recursos entre tareas e interrupciones. De esta forma si un recurso es compartido entre una combinación de tareas e interrupciones al momento de llamar a `GetResource` la interfaz se comporta de forma diferente si es llamada desde una interrupción o desde una tarea.

Si se llama `GetResource` desde una tarea, se modifica la prioridad según lo explicado en Sección 5.3, “OSEK Priority Ceiling Protocol”, pero además se deshabilitan todas las interrupciones que utilizan el recurso que se desea obtener. Si es llamada `GetResource` desde una interrupción, el sistema operativo deshabilita las interrupciones de mayor prioridad que utilizan el recurso que se desea obtener.

Al liberarse un recurso el sistema operativo realizaría la operación contraria a lo recién explicado.

Ejemplo 5.5. Utilización de recursos en interrupciones

```
TASK(TaskA)
{
    /* ... */
    GetResource(Res1);
    /* critical section */
    ReleaseResource(Res1);
    TerminateTask();
}

ISR(Isr1)
{
    /* ... */
    GetResource(Res1);
    /* critical section */
    ReleaseResource(Res1);
}
```

```
TASK TaskA {
    /* ... */
    RESOURCE = Res1;
};

ISR Isr1 {
    /* ... */
    RESOURCE = Res1;
}

RESOURCE Res1;
```

El Ejemplo 5.5, “Utilización de recursos en interrupciones” muestra como se puede utilizar y configurar un recurso que es compartido entre una tarea y una interrupción.



El CIAA-Firmware hasta su versión 0.6.1 no soporta la utilización de recursos en interrupciones.

Capítulo 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
- Establecer un evento de una tarea
- Llamar una callback (retrollamada) en C

Para la implementación de las alarmas, ya sean que se utiliza una ó más alarmas el sistema operativo necesitará un contador de hardware. Con un contador es posible configurar la cantidad de alarmas que sean necesarias. Cada alarma debe ser definida en el formato OIL así como la correspondiente acción. El Ejemplo 6.1, “Configuración de alarmas” muestra la configuración de dos alarmas, la primera alarma se llama `ActivateTaskC` 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 establece una alarma en tiempo relativo, la segunda en tiempo absoluto mientras la última cancela la alarma.

Ejemplo 6.1. Configuración de alarmas

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

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

En el Ejemplo 6.2, “Utilización de alarmas” 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.

Ejemplo 6.2. Utilización de alarmas

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

    TerminateTask();
}

TASK(TaskC) {
    static int counter = 0;

    /* increment counter */
    counter++;

    /* check if the task has been executed 10 times */
    if (counter > 10) {
        /* reset counter */
        counter = 0;
        /* stop the alarm */
        CancelAlarm(ActivateTaskC);
    }

    /* do something */

    TerminateTask();
}
```

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



Gracias al desfase inicial de 100 y 150 ticks entre ambas alarmas se puede lograr que las alarmas no expiren en el mismo momento sino con 50 ticks de diferencia. 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.



OSEK-OS define el tiempo dentro de los parámetros de las interfaces de las alarmas en Ticks. Esto es un problema ya que el código de configuración de los tiempos de las alarmas no será portable. En el CIAA-Firmware se intenta mantener la convención de que cada Tick equivale a un milisegundo.

6.1. Offset en las Alarms

Es muy común en sistemas de tiempo real que algunas tareas deban ser ejecutadas cíclicamente con diferentes periodos. En la Figura 6.1, “Desfase de tareas mediante offset” se pueden ver dos tareas cíclicas TaskA y TaskB con periodos de 10ms y 20ms.

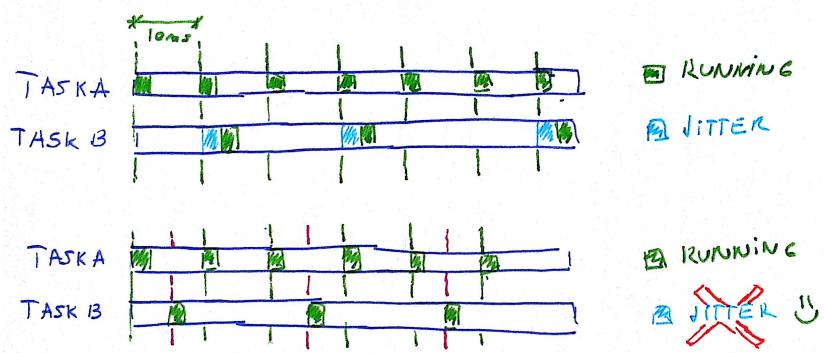


Figura 6.1. Desfase de tareas mediante offset

En la primera configuración se puede observar las dos tareas son activadas simultáneamente a los 10ms, 30ms, 50ms, etc. Al haber dos tareas activas el scheduler selecciona la de mayor prioridad y procede a ejecutarla. En este caso TaskA. Al finalizar se procede a ejecutar TaskB. Es probable que la duración de TaskA no siempre sea la misma y por ello la tarea TaskB no siempre será ejecutada cada 20ms, sino a veces un poco antes, a veces un poco más tarde.

La forma de evitar este *jitter* no deseado es mediante el desfase de las diferentes tareas. En el segundo ejemplo se observa que gracias a un desfase de 5ms la tarea TaskB es activada a los 5ms, 25ms, 45ms, etc. De esta forma se evita que ambas tareas sean activadas simultáneamente.

Siempre y cuando la tarea TaskA se ejecute en menos de 5ms la tarea TaskB será ejecutada sin *jitter*.

Para lograr estos desfases se puede utilizar el segundo parámetro de las interfaces `SetRelAlarm` y `SetAbsAlarm`.

Capítulo 7. Interrupciones

OSEK-OS define dos tipos de interrupciones que denomina:

- *ISR1*: las ISR (del inglés Interrupt Service Routine) Category 1 son transparentes al OSEK-OS y por ello no pueden utilizar casi ninguna interfaz del sistema operativo (ver la Tabla A.1, “Interfaces de OSEK-OS y contexto del que pueden ser llamadas”).
- *ISR2*: las ISR Category 2 tienen una mínima intervención del OS y por ello pueden utilizar algunas interfaces del sistema operativo (ver la Tabla A.1, “Interfaces de OSEK-OS y contexto del que pueden ser llamadas”).

Cualquier *ISR* ya sea Category 1 o 2 va a interrumpir a cualquier tarea independientemente de la prioridad de la misma. El scheduling de las tareas, como se ve en la Figura 7.1, “Planificación de Tareas e Interrupciones en OSEK-OS” es realizado por el kernel del sistema operativo mientras que las de las ISR son administradas por el hardware. Sin importar si se trata de una tarea *PREEMPTIVE* o *NON PREEMPTIVE* la misma va a ser interrumpida si se recibe una interrupción. En caso de querer evitar esto el sistema operativo provee al usuario las siguientes interfaces para desactivar las interrupciones:

- `DisableAllInterrupts`
- `EnableAllInterrupts`
- `SuspendAllInterrupts`
- `ResumeAllInterrupts`
- `SuspendOSInterrupts`
- `ResumeOSInterrupts`

No está permitido llamar a ninguna interfaz del sistema operativo mientras las interrupciones están deshabilitadas, salvo estas mismas funciones para habilitar y deshabilitar las interrupciones.

También existe la posibilidad de utilizar recursos para evitar interferencias entre interrupciones o entre interrupciones y tareas:

- `GetResource`¹
- `ReleaseResource`¹

El soporte de recursos dentro de las interrupciones es discutido en Sección 5.7, “Recursos en interrupciones”, la misma es una funcionalidad opcional de OSEK-OS que en la actualidad no es soportada por el OSEK-OS del CIAA-Firmware.

¹El soporte de recursos entre interrupciones y/o tareas e interrupciones es una funcionalidad opcional, la misma hoy no es soportada por el CIAA-Firmware, ver la Sección 5.7, “Recursos en interrupciones”.

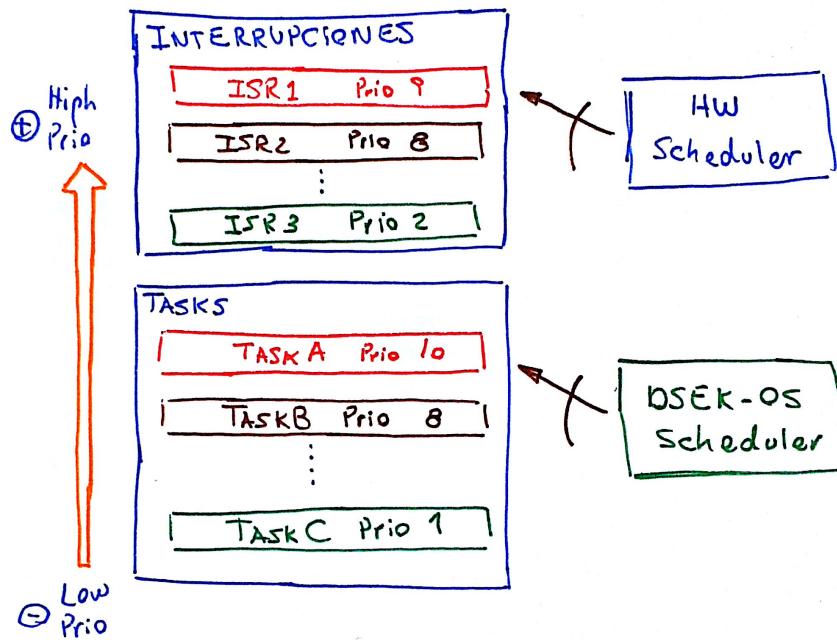


Figura 7.1. Planificación de Tareas e Interrupciones en OSEK-OS

Las interrupciones deben ser configuradas en el archivo .oil para ser utilizadas. Lo mismo se puede ver en el Ejemplo 7.1, "Utilización de interrupciones".

Ejemplo 7.1. Utilización de interrupciones

```
ISR(UART0_Handler){  
    /* do something uart  
       rx handling */  
}  
  
ISR(TIMER2_Interrupt){  
    /* do something */  
}
```

```
ISR_UART0_Handler {  
    CATEGORY = 2;  
    INTERRUPT = UART0;  
    PRIORITY = 0;  
}  
  
ISR_TIMER2_Interrupt {  
    CATEGORY = 1;  
    INTERRUPT = TIMER2;  
    PRIORITY = 0;  
}
```

- 1 Las interrupciones deben ser definidas utilizando el macro ISR y con el mismo nombre que fue definida en el archivo .oil de configuración.
- 2 Se puede observar que no hay diferencia en la definición de una ISR de categoría 1 o 2, desde el código no se puede reconocer la categoría, se debe editar el archivo de configuración oil. En este caso UART0_Handler es de categoría 2 mientras que TIMER2_Interrupt es de categoría 1^a.
- 3 A las interrupciones se les debe asignar un nombre el cual será luego utilizado en los archivo .c.
- 4 En este caso es de categoría 2. La interrupción es la UART0. El nombre de las interrupciones existentes es específica de cada implementación y cada plataforma.
- 5 A las interrupciones se les puede asignar una prioridad, sin embargo depende de la implementación del OSEK-OS y de la plataforma si este parámetro es evaluado y si es posible configurar la prioridad seleccionada. El OSEK-OS de la

CIAA-Firmware no soporta prioridades en las interrupciones.

- 5 En este ejemplo la interrupción `TIMER2_INTERRUPT` es de categoría 1.

^aEsto es específico del OSEK-OS del CIAA-Firmware, el estándar habla de la utilización del macro `ISR` únicamente en las interrupciones de categoría 2.

Los nombres de las interrupciones indicados en el OIL mediante el parámetro `INTERRUPT` no se encuentran definidos en el estándar. En este ejemplo `UART0` y `TIMER2` no son parte del estándar sino de la implementación en el CIAA-Firmware. Otras implementaciones utilizan otro parámetro y el mismo puede tomar otros valores. Por lo que al cambiar de una implementación a otra de OSEK-OS es probable sea necesario adaptar la configuración de las interrupciones.



En el CIAA-Firmware no existe todavía una documentación con un mapeo entre el nombre y la interrupción. Por lo general se utiliza el nombre como lo define la documentación del microcontrolador, también se puede analizar el código fuente del OSEK-OS del CIAA-Firmware. En el archivo `modules/rtos/gen/src/cortexM4/Os_Internal_Arch_Cfg.c.php` [https://github.com/ciaa/firmware.modules.rtos/blob/master/gen/src/cortexM4/Os_Internal_Arch_Cfg.c.php].

7.1. Anidamiento

Las interfaces `DisableAllInterrupts` y `EnableAllInterrupts` no pueden ser llamar recursivamente. No es posible llamar dos veces a `DisableAllInterrupts` y a continuación dos veces a `EnableAllInterrupts`.

Si se desea anidar los llamados se deben utilizar las interfaces `SuspendAllInterrupts`, `ResumeAllInterrupts`, `SuspendOSInterrupts`, `ResumeOSInterrupts`. Las primeras dos interfaces para deshabilitar y habilitar todas las interrupciones y las segundas dos para deshabilitar y habilitar las interrupciones de categoría 2.

Para manejar la recursividad el sistema operativo utilizará recursos adicionales del sistema, por lo que la implementación de las funciones (macros) recursivos puede ser algo más compleja que las interfaces que no aceptan recursividad.

Capítulo 8. Hookroutines

Las *hookroutines* (en español rutinas de enganche), son funciones implementadas por el usuario que el sistema operativo llamará en situaciones específicas. Las mismas son opcionales y se pueden utilizar para agregar algún manejo específico deseado por el usuario en las siguientes situaciones:

- `StartupHook`: es llamada durante la inicialización del sistema operativo, antes de ser completada.
- `ShutdownHook`: es llamada al finalizar el apagado del sistema operativo.
- `PreTaskHook`: es llamada antes de proceder a ejecutar una tarea.
- `PostTaskHook`: es llamada al finalizar la ejecución de una tarea.
- `ErrorHook`: es llamada en caso de que alguna de las interfaces del sistema operativo retorne un valor distinto a `E_OK`.

El usuario debe activar las rutinas que necesite en la configuración del sistema operativo y definir las funciones. El sistema operativo realiza la declaración de las mismas. Al incluir `os.h` se incluirán también las declaraciones de las rutinas activadas en la configuración.

Las siguientes cinco secciones son un manual de referencia de las hookroutines.

8.1. StartupHook

Declaración	void StartupHook (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	La función debe ser definida por el usuario y es llamada por el sistema al final de la inicialización del sistema operativo pero antes de iniciar el <i>Scheduler</i> . Puede ser utilizada para para la inicialización de drivers. Durante su ejecución las interrupciones se encuentran desactivadas.	

Ejemplo 8.1. Utilización de StartupHook

```
int main(void)①
{
    StartOS(AppMode1);②
    while(1);
}

void StartupHook(void)③
{
    /* do something */
}

TASK(InitTask)④
{
    /* do something */
    TerminateTask();
}
```

```
OS ExampleOS {
    STARTUPHOOK = TRUE;①
}

TASK InitTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    AUTOSTART = TRUE {②
        APPMODE = AppMode1;
    }
}
```

- ① Luego de completada la inicialización de C, la función `main` es llamada, la misma en este caso procede a iniciar la ejecución del sistema operativo en ②.
- ③ Una vez completada la inicialización del sistema operativo, el mismo llama a `StartupHook`.
- ④ Recién al retornar `StartupHook` se inicia el *Scheduler* y se procede a ejecutar las tareas.

- ① Para que el sistema operativo llame a la `StartupHook`, lo mismo se debe indicar estableciendo `STARTUPHOOK` a `TRUE`. Recién después de llamar a `StartupHook` el sistema operativo procederá a iniciar el *Scheduler*, por ello la tarea `InitTask` será ejecutada luego que `StartupHook`.

8.2. PreTaskHook

Declaración	void PreTaskHook (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	La función debe ser definida por el usuario y es llamada por el sistema operativo al comenzar la ejecución de una tarea.	

Ejemplo 8.2. Utilización de PreTaskHook

```
void PreTaskHook(void)
{
    /* some code */
}

TASK(TaskA)
{
    /* do something */
    TerminateTask();
}
```

```
OS ExampleOS {
    PRETASKHOOK = TRUE;❶
}

TASK TaskA {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```

- ❶ Para que el sistema operativo llame a la `PreTaskHook`, lo mismo se debe indicar configurando el parámetro `PRETASKHOOK` a `TRUE`.

8.3. PostTaskHook

Declaración	void PostTaskHook (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	La función debe ser definida por el usuario y es llamada por el sistema operativo al finalizar la ejecución de una tarea.	

Ejemplo 8.3. Utilización de PostTaskHook

```
void PostTaskHook(void)
{
    /* some code */
}

TASK(TaskA)
{
    /* do something */
    TerminateTask();
}
```

```
OS ExampleOS {
    POSTTASKHOOK = TRUE;❶
}

TASK TaskA {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```

- ❶ Para que el sistema operativo llame a la `PreTaskHook`, lo mismo se debe indicar configurando el parámetro `POSTTASKHOOK` a `TRUE`.

8.4. ErrorHook

Declaración	void ErrorHook (StatusType <i>Error</i>);	
Parámetros de entrada	Error	Error ocurrido
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	La función debe ser definida por el usuario y es llamada por el sistema operativo al finalizar la ejecución de cualquier servicio del sistema operativo cuando el valor a retornar sea diferente a <code>E_OK</code> . Para evitar una posible recursividad infinita, <code>ErrorHook</code> no es llamada si el servicio que retorna un valor diferente a <code>E_OK</code> fue llamado desde <code>ErrorHook</code> .	



En el Capítulo 9, *Manejo de Errores y depuración* y la Sección 9.1, “La función `ErrorHook`” se ven algunos ejemplos adicionales de como utilizar `ErrorHook` para la depuración de programas.

Ejemplo 8.4. Utilización de `ErrorHook`

```
void ErrorHook(StatusType error)
{
    /* some fault reporting code */
}

TASK(TaskA)
{
    /* force an error */
    /* the ErrorHook will be called
       indicating E_OS_LIMIT */
    ActivateTask(TaskA);

    TerminateTask();
}
```

```
OS_Exampleos {
    ERRORHOOK = TRUE;❶
}

TASK TaskA {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```

- ❶ Para que el sistema operativo llame a la `ErrorHook`, lo mismo se debe indicar configurando el parámetro `ERRORHOOK` a `TRUE`.

8.5. ShutdownHook

Declaración	<code>void ShutdownHook(StatusType Error);</code>	
Parámetros de entrada	Error	Error ocurrido
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	La función debe ser definida por el usuario y es llamada por el sistema durante o al finalizar la ejecución de ShutdownOS. El usuario puede utilizar esta función para implementar rutinas específicas para apagar el sistema.	

Ejemplo 8.5. Utilización de ShutdownHook

```
void ShutdownHook(StatusType Error)
{
    if (E_OK == Error)❶
    {
        /* normal shutdown */
        /* ... */
    } else {
        /* some error occurs */
        /* ... */
    }
}

TASK(TaskA)
{
    /* check if the system
       shall be turned off */
    if (ShutdownCondition)
    {
        ShutdownOS(E_OK);❷
    }
    /* ... */
    TerminateTask();
}
```

```
OS_Exampleos {
    SHUTDOWNHOOK = TRUE;❸
}

TASK TaskA {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```

- ❶ Para que el sistema operativo llame a la ShutdownHook, lo mismo se debe indicar configurando el parámetro `SHUTDOWNHOOK` a `TRUE`.

- ❷ ShutdownHook es llamado desde ShutdownOS, por ello es necesario apagar el sistema para que la misma sea llamada.
- ❸ En caso de no haber ocurrido un error el parámetro `Error` tendrá el valor `E_OK`.

Capítulo 9. Manejo de Errores y depuración

OSEK-OS ofrece dos niveles de chequeo de errores que se puede configurar en el archivo *.oil*

- *extendido*: es el modo que se utiliza durante el desarrollo. Algunos chequeos extras son realizados y retornados en caso de error.
- *estándar*: es el modo que se utiliza en producción. Únicamente errores muy críticos son detectados y reportados.



Es importante tener en cuenta que el modo de manejo de errores extendido necesita de recursos adicionales de RAM/ROM y de procesamiento.

En el Ejemplo 9.1, “Manejo de errores” se utiliza la interfaz `ActivateTask` que puede retornar `E_OK`, `E_OS_LIMIT` ó `E_OS_ID` cuando el nivel de los chequeo de errores es establecido a *extendido* pero únicamente `E_OK` y `E_OS_LIMIT` cuando el nivel de chequeos de errores es establecido a *estándar*.

Ejemplo 9.1. Manejo de errores

```
TaskType taskToActivate = TaskB;  
  
TASK(TaskA)  
{  
    StatusType ret;  
  
    ret = ActivateTask(  
        taskToActivate);❶  
  
    if (E_OS_LIMIT == ret)❷  
    {  
        /* some error handling */  
    }  
    else if (E_OS_ID == ret)❸  
    {  
        /* some other error  
         handling */  
    }  
    else if (E_OK == ret)❹  
    {  
        /* E_OK has been ret. */  
        /* no error */  
    }  
    else  
    {  
        /* not possible */  
    }  
  
    TerminateTask();  
}
```

```
OS ExampleOS {  
    STATUS = EXTENDED;❶  
};
```

- ❶ El sistema operativo es configurado para retornar errores de forma extendida.

```
OS ExampleOS {  
    STATUS = STANDARD;❶  
};
```

- ❶ El sistema operativo es configurado para retornar únicamente los errores estándar.

- ❶ Se llama a la interfaz `ActivateTask`.
❷ La interfaz puede retornar `E_OS_LIMIT` si la tarea alcanza la mayor cantidad de activaciones configuradas.
❸ La interfaz puede retornar `E_OS_ID` si el sistema ha sido configurado para retornar errores extendidos. El error será

retornado si la tarea TaskB tiene un ID inválido.

- 4 En caso de no haber error la interfaz retorna E_OK

El `else if (E_OS_ID == ret)` es únicamente necesario si el sistema es configurado de forma extendida en caso contrario será código inalcanzable (unreachable code). Siendo este código por ello no deseable.

Por lo general se puede pensar que el chequeo extendido se utiliza durante el desarrollo y depuración y el modo estándar durante la producción. Lo mismo se puede tomar como cierto. Sin embargo hay que tener en cuenta algunos problemas asociados.

Modificar la configuración de este parámetro modifica el comportamiento del sistema. Por lo cual surge la pregunta de cuando utilizar y cuando no el chequeo extendido.

Si se ha realizado todo el desarrollo y ensayos de un sistema con la utilización de chequeos extendidos, será un riesgo a evaluar modificar este parámetro para la producción. Ya que el código ya fue probado, además de que podrán aparecer secciones de código inalcanzables. Por otro lado si se utiliza desde el comienzo del desarrollo los chequeos estándar se pierden las ventajas del extendido.

Lamentablemente no hay solución a este problema, siempre habrá tres estrategias:

- No utilizar nunca chequeo extendido.
- Utilizar siempre chequeo extendido.
- Comenzar el desarrollo con chequeo extendido y en algún momento pasar al chequeo estándar y realizar todas las verificaciones y ensayos que correspondan.

En caso de seguir esta última estrategia el equipo de desarrollo deberá decidir cual es el momento indicado para realizar este cambio:

- Si el cambio se realiza muy temprano en el desarrollo puede que algunos errores no sean detectados ya que se ha desactivado el uso extendido de errores
- Si el cambio se realiza muy tarde puede que no se llegue a encontrar otros errores debido a la eliminación de los chequeos extendidos



Existe una opción que no resuelve completamente el problema, pero evita que el código `else if (E_OS_ID == ret)` sea necesario. Mediante la utilización de una `ErrorHook` se puede indicar al sistema de en caso de que una interfaz retorne un error se llame a una función definida por el usuario. Para más información ver Sección 8.4, "ErrorHook".

9.1. La función ErrorHook

Si volvemos al Ejemplo 9.1, "Manejo de errores" se puede observar que controlar todos los posibles errores retornados por cada interfaz del sistema operativo puede ser algo engorro-

so, además de que el código pierde en legibilidad. OSEK-OS ofrece la posibilidad de que un usuario defina una función denominada `ErrorHook` que será llamada en caso de que cualquier interfaz del sistema operativo retorne un valor diferente a `E_OK`. La función se describe en la Sección 8.4, “`ErrorHook`”.

Utilizando `ErrorHook` el código del Ejemplo 9.1, “Manejo de errores” podría ser como el de Ejemplo 9.2, “Manejo de errores mediante `ErrorHook`”.

Ejemplo 9.2. Manejo de errores mediante `ErrorHook`

```
TaskType taskToActivate = TaskB;

TASK(TaskA)
{
    StatusType ret;

    (void)ActivateTask(
        taskToActivate);❶

    TerminateTask();
}

void ErrorHook(StatusType Error)
{
    TaskType task;
    TaskType taskID;

    /* get error information */
    GetTaskID(&task);❷

    switch(
        OS_ErrorGetServiceId());❸
    {
        case OSServiceId_ActivateTask:❹
            taskID =
                OS_Error_ActivateTask_TaskID();❺
            if (E_OS_ID == Error);❻
            {
                /* ... */
            }
            break;
        /* ... */
    }
}
```

```
OS ExampleOS {
    STATUS = EXTENDED;
    ERRORHOOK = TRUE;❻
};
```

- ❶ El sistema operativo es configurado para retornar errores de forma extendida y reportarlos a `ErrorHook`.

```
OS ExampleOS {
    STATUS = STANDARD;
    ERRORHOOK = TRUE;❻
};
```

- ❶ El sistema operativo es configurado para retornar únicamente los errores estándar y reportarlos a `ErrorHook`.

- ❶ Se llama a la interfaz `ActivateTask`. La interfaz puede retornar `E_OS_LIMIT`, `E_OS_ID` ó `E_OK`, sin embargo el valor retornaido se ignora.
- ❷ Mediante `GetTaskID` se detectar la tarea que causa el error.
- ❸ Mediante el macro `OS_ErrorGetServiceId` conocer la interfaz que retornó un error.
- ❹ Por cada interfaz el sistema operativo ofrece un macro, en este caso `OSServiceId_ActivateTask`.
- ❺ Se podrá utilizar un macro por parámetro, en este caso `ActivateTask` tiene un único parámetro `TaskID` y por ello el macro `OS_Error_ActivateTask_TaskID`.
- ❻ Por ultimo se puede utilizar el parámetro `Error` para conocer el error retornado.

Del Ejemplo 9.2, “Manejo de errores mediante `ErrorHook`” se pueden observar que OSEK-OS define los siguientes macros para poder depurar los errores.

- `OSErrorGetServiceId()` el cual retorna el servicio que genero el llamado a `ErrorHook`.
- `OSServiceId_xxx` donde `xxx` representa el nombre de las interfaces del sistema operativo como ser `OSServiceId_ActivateTask` o `OSServiceId_SetRelAlarm`.
- `OSError_xxx_yyy()` donde `xxx` representa el nombre de las interfaces del sistema operativo y `yyy` el nombre de los parámetros de la interfaz `xxx`. Por ejemplo `OSError_ActivateTask_TaskID()` o `OSError_SetRelAlarm_AlarmID()`.



El OSEK del CIAA-Firmware soporta parcialmente esta funcionalidad, los macros `OSError_xxx_yyy()` no se encuentran disponibles, sino `OSErrorGetParam1()`, `OSErrorGetParam2()` y `OSErrorGetParam3()`. Este error ya ha sido reportado en <https://github.com/ciaa/Firmware/issues/266>.

Capítulo 10. Conclusiones y perspectivas

OSEK-OS es un estándar estable, del mismo no se realizan más actualizaciones. Como tal tiene ventajas y desventajas. Por un lado, al no haber modificaciones ni actualizaciones, el mismo se mantiene compatible. Por el otro, si se tiene en cuenta que la versión 1.0 del estándar de OSEK-OS es del 1995 ([OSEK-OS] pág. 85) y la última versión la 2.2.3 del 2005 ([OSEK-OS]) es de entender la falta de soporte para funcionalidades como la protección de memoria o procesadores con múltiples núcleos. Ambas funcionalidades que serían de esperar en un sistema operativo de hoy en día.

Una forma de compensar estas falencias sería *extendiendo* OSEK-OS. En el proyecto CIAA se está trabajando en algunas extensiones y mejoras del sistema operativo. Por ejemplo en el soporte de procesadores con múltiples núcleos, mediante el cual se podrán asignar tareas a un núcleo específico y sincronizar las mismas mediante eventos y la utilización de recursos entre núcleos. También se piensan realizar algunas mejoras de rendimiento, por ejemplo en la adquisición y liberación de recursos. Otra extensión posible sería el control de desborde de pila. Es importante tener en cuenta que las *extensiones* no forman parte del estándar. Quien las utilice no podrá migrar a otra implementación de OSEK-OS sin antes adaptar las partes del código que utilizan estas extensiones. La decisión final será la del usuario, quien deberá decidir si valora más las extensiones o prefiere mantener la compatibilidad con el estándar.

La industria automotriz se encuentra migrando de OSEK-OS a un nuevo estándar denominado AUTOSAR (Sección 1.2, “Futuro del estándar”). AUTOSAR ofrece un sistema operativo de tiempo real compatible con OSEK-OS, pero que además soporta, entre otras funcionalidades, procesadores de múltiples núcleos¹, protección de memoria¹, protección de tiempo¹. Lamentablemente la licencia AUTOSAR no es completamente abierta como la de OSEK-OS y por ello su utilización es permitida únicamente a miembros la asociación.

La Tabla 2.1, “Requerimientos por cada clase de conformidad” indica los requerimientos mínimos para un OSEK-OS. La misma es de gran importancia para quien quiera asegurar la portabilidad entre sistemas operativos OSEK-OS. Sin embargo esta limitaría a muchas aplicaciones de hoy en día. Según la Tabla 2.1, “Requerimientos por cada clase de conformidad” para que una aplicación sea capaz de correr en *cualquier* OSEK-OS debe tener no más de dieciséis tareas, hasta 16 prioridades distintas y un máximo de una alarma. En caso contrario podría existir una implementación de OSEK-OS que siendo conforme al estándar no pueda correr nuestra aplicación. Obviamente casi todas las implementaciones de OSEK-OS soportan valores mayores a los indicados en dicha tabla. Por ejemplo la implementación de OSEK-OS ERIKA soporta >254² tareas, 31² alarmas (en un procesador de 32 bits), pero sin embargo sólo 16² prioridades como especificado en el estándar. El OSEK-OS del CIAA-Firmware soporta valores superiores a los de la Tabla 2.1, “Requerimientos por cada clase de conformidad”, pero los mismos todavía no fueron documentados³. A diferencia de ERIKA en el CIAA-Firmware

¹[AUTOSAR-OS]

²[ERIKA] pág. 10.

³<https://github.com/ciaa/Firmware/issues/267>

se pueden tener más de diecisésis prioridades, por lo que una aplicación que utilice esta funcionalidad adicional del CIAA-Firmware tendrá dificultades en el caso de ser portada a ERIKA.

El documento [FEPE] realiza un análisis profundo de OSEK-OS y OSEK-COM y crítica principalmente las limitaciones en la Tabla 2.1, “Requerimientos por cada clase de conformidad” como los más débiles del estándar OSEK-OS. Como contra punto hay que recordar que el estándar fue inicialmente escrito en 1995 y los procesadores de su momento no tenían la capacidad de computo ni de almacenamiento de hoy en día.

En el CIAA-Firmware no solo estamos trabajando en mejoras en la implementación, también se sigue trabajando en las pruebas del sistema operativo. OSEK-OS define además del estándar del sistema operativo en los documentos [OSEK-TM], [OSEK-TPlan] y [OSEK-TProc] como llevar a cabo la verificación del sistema operativo. Todas estas pruebas ya fueron implementadas y superadas por el CIAA-Firmware. Sin embargo siguen pendiente tareas como un análisis de code coverage, para verificar de que se hayan probado todas las líneas de código. Tampoco se han realizado pruebas de *brute-force* que no están especificadas en el estándar, pero seguro serían de gran valor agregado para el usuario final.

El sistema operativo del CIAA-Firmware, como así también el CIAA-Firmware, tienen un gran potencial. Todo dependerá de ustedes los usuarios. Cuantos más desarrolladores lo utilicen y más grande sea la comunidad que base sus trabajos en el CIAA-Firmware más prometedor será su futuro.

Apéndice A. Manual de referencia de OSEK-OS

Esta sección sirve de manual de referencia de los tipos e interfaces de OSEK-OS ([OSEK-OS]). Por cada interfaz se provee la siguiente información:

- Definición de la función
 - Parámetros de entrada, salida y entrada/salida
 - Valores que retorna de forma estándar [45] y de forma extendida [45]
 - Clases en las que la función se puede utilizar
 - Descripción de la función
 - Ejemplo de utilización de la función inclusive un archivo de configuración oil ([OSEK-OIL])

La Tabla A.1, "Interfaces de OSEK-OS y contexto del que pueden ser llamadas" incluye el listado completo de las interfaces indicando en qué contexto puede ser llamada cada una.

Grupo	Interfaz	Tarea	ISR1	ISR2	Error	PreTask	PostTask	Startup	Shutdown	callback
Alarmas	GetAlarmBase	X		X	X	X	X			
	GetAlarm	X		X	X	X	X			
	SetAbsAlarm	X		X						
	SetRelAlarm	X		X						
	CancelAlarm	X		X						
Otras	GetActiveApplication-Mode	X		X	X	X	X	X	X	
	StartOS ^b									
	ShutdownOS	X		X	X			X		

^aEl soporte de recursos en las interrupciones es una funcionalidad opcional de OSEK-OS. La funcionalidad no se encuentra implementada en el CIAA-Firmware 0.6.1.

^bStartOS no puede ser llamada desde el mismo sistema, lo más natural es llamarla desde la función main luego de realizar las inicializaciones necesarias.

Tabla A.1. Interfaces de OSEK-OS y contexto del que pueden ser llamadas

A.1. Tipos

A.1.1. StatusType

Es el tipo de datos retornado en todas las interfaces de OSEK-OS. El mismo puede tomar los siguientes valores:

- E_OK 0: es retornado cuando la interfaz se ha ejecutado sin errores
- E_OS_ACCESS 1: es retornado cuando
- E_OS_CALLEVEL 2: es retornado cuando
- E_OS_ID 3: es retornado cuando
- E_OS_LIMIT 4: es retornado cuando
- E_OS_NOFUNC 5: es retornado cuando
- E_OS_RESOURCE 6: es retornado cuando
- E_OS_STATE 7: es retornado cuando
- E_OS_VALUE 8: es retornado cuando

A.1.2. AlarmBaseType

Es el tipo utilizado para almacenar la información de un contador de una alarma. El tipo es una estructura que contiene:

- `maxallowedvalue`: el máximo valor de ticks que puede ser alcanzado.
- `ticksperbase`: la cantidad de ticks necesarios para alcanzar una unidad específica. La interpretación es específica de cada implementación. En el CIAA-Firmware este valor no es utilizado más que para ser retornado en la interfaz `GetAlarmBase`.
- `mincycle`: valor mínimo para de un ciclo que luego será indicado los parámetros de `SetRelAlarm` y `SetAbsAlarm`.

Todos los elementos son del tipo `TickType`.

A.1.3. TickType

Tipo de dato para representar el valor de un contador.

A.1.4. TickRefType

Referencia a `TickType`.

A.2. ActivateTask

Declaración	StatusType ActivateTask (TaskType TaskID);		
Parámetros de entrada	TaskID	Identificador de la tarea a activar	
Parámetros de salida	-		
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
		E_OS_LIMIT	La tarea ha llegado al límite de activations
	Extendida	E_OS_ID	El identificador de tarea es inválido
Clases	BCC1, BCC2, ECC1, ECC2,		
Descripción	La tarea es agregada en la lista de tareas <i>ready</i> a ser ejecutada. Si la tarea es extendida los eventos son establecidos a cero. Ver Sección 4.5.3, "Puntos de Scheduling"		

Ejemplo A.1. Utilización de ActivateTask

```

TASK(DemoTask)
{
    /* activate the task
       SomeOtherTask */
    Activate(SomeOtherTask);

    TerminateTask();
}

TASK(SomeOtherTask)
{
    /* task has been
       activated from
       DemoTask */

    TerminateTask();
}
  
```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
  
```

A.3. ChainTask

Declaración	StatusType ChianTask (TaskType TaskID);				
Parámetros de entrada	TaskID	Identificador de la tarea a activar			
Parámetros de salida	-				
Parámetros de entra- da/salida	-				
Retorna	Estándar	(no retorna)	Ningún error		
		E_OS_LIMIT	La tarea ha llegado al límite de activaciones		
	Extendida	E_OS_ID	El identificador de tarea es inválido		
		E_OS_RESOURCE	Uno (o más) recurso no ha sido retorna- do al sistema llamando a ReleaseRe- source.		
		E_OS_CALLEVEL	La interfaz no puede ser llamada desde una rutina de interrupción (ISR1/ISR2), únicamente desde tareas		
Clases	BCC1, BCC2, ECC1, ECC2,				
Descripción	La tarea actual es terminada, a continuación la tarea indicada en TaskID es agregada en la lista de tareas ready a ser ejecutada. Si la tarea indicada en TaskID es extendida los eventos son establecidos a cero.				



ChainTask puede ser utilizada por una tarea para reactivarse a si misma. La utilización de ChainTask no incrementa el número de activaciones y por ello también puede ser utilizado en tareas extendidas.



[OSEK-OS] especifica en la página 51:

If the succeeding task is identical with the current task, this does not result in multiple requests. The task is not transferred to the suspended state, but will immediately become ready again.

When an extended task is transferred from suspended state into ready state all its events are cleared.

Lo cual deja en claro que si una tarea extendida llama a ChainTask no pasa al estado suspended y por ello sus eventos no son establecidos a cero.

Ejemplo A.2. Utilización de ActivateTask

```
TASK(DemoTask)
{
    StatusType ret;
    ret = ChainTask(
        SomeOtherTask);
    /* something goes wrong */
    /* some error reaction */
    TerminateTask();
}

TASK(SomeOtherTask)
{
    /* task has been
       activated from
       DemoTask */
    TerminateTask();
}

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```

A.4. GetTaskID

Declaración	StatusType GetTaskID (TaskRefType TaskID);		
Parámetros de entrada	-		
Parámetros de salida	TaskID	Identificador de la tarea que se encuentra corriendo	
Parámetros de entra- da/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	-	
Clases	BCC1, BCC2, ECC1, ECC2,		
Descripción	Retorna el identificador de la tarea que se encuentra corriendo. Si en el momento de ser llamada la función no hay tarea corriendo la función escribe el valor INVALID_TASK en TaskID.		



GetTaskID es por lo general utilizada desde ISRs o desde librerías, para saber en qué entorno se encuentran corriendo momentáneamente.

Ejemplo A.3. Utilización de GetTaskID

```

void GenericFunction(void) {
    TaskType TaskID;

    /* this function only returns
     * E_OK */
    (void)GetTaskID(&TaskID);

    switch (TaskID) {
        case DemoTask:
            /* some special code
             * for DemoTask */
            break;

        default:
            /* some code for all
             * other tasks */
            break;
    }
}

TASK(DemoTask)
{
    GenericFunction();
    TerminateTask();
}

TASK(SomeOtherTask)
{
    GenericFunction();
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

```

A.5. GetTaskState

Declaración	StatusType GetTaskState (TaskType TaskID, TaskStateRefType State);		
Parámetros de entrada	TaskID	identificador de la tarea para la que se quiere obtener el estado	
Parámetros de salida	State	Estado de la tarea indicada en TaskID	
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	E_OS_ID	El identificador TaskID es invalido
Clases	BCC1, BCC2, ECC1, ECC2,		
Descripción	Retorna el estado de la tarea TaskID (running, ready, waiting, suspended) (Sección 4.2, "Estados").		

Ejemplo A.4. Utilización de GetTaskState

```

TASK(DemoTask)
{
    TaskState state;
    ActivateTask(SomeOtherTask);
    GetTaskState(SomeOtherTask,
                &state);
    if (WAITING == state)
    {
        SetEvent(SomeOtherTask,
                 Event1);
    }
    TerminateTask();
}
TASK(SomeOtherTask)
{
    WaitEvent(Event1);
    ClearEvent(Event1);
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    SCHEDLE = FULE
    TYPE = EXTENDED;
}

TASK SomeOtherTask {
    PRIORITY = 4;
    ACTIVATION = 1;
    STACK = 512;
    SCHEDULE = FULL;
    TYPE = EXTENDED;
    EVENT = Event1;
}

```

A.6. Schedule

Declaración	StatusType Schedule (void);		
Parámetros de entrada	-		
Parámetros de salida	-		
Parámetros de entra- da/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	E_OS_CALLEVEL	La interfaz no fue llamada desde el contexto de una tarea
		E_OS_RESOURCE	Al menos un recurso sigue siendo ocupado por la tarea (ver Sección A.15, "ReleaseResource")
Clases	BCC1, BCC2, ECC1, ECC2,		
Descripción	Si una tarea de mayor prioridad se encuentra en estado ready el contexto de la tarea actual es guardado y la misma es puesta en estado ready. La tarea de mayor prioridad pasa a ser ejecutada. En caso contrario la función retorna inmediatamente.		



La interfaz se puede utilizar desde cualquier tarea pero tiene efectos únicamente en tareas non preemptables o en tareas preemptive que tienen asignado al menos un recurso interno el cual es compartido con otras tareas.

Ejemplo A.5. Utilización de Schedule

```

TASK(DemoTask)
{
    ActivateTask(SomeOtherTask);①
    Schedule();②
    TerminateTask();
}

TASK(SomeOtherTask)
{
    /* do something */③
    TerminateTask();
}
  
```

```

TASK DemoTask {
    PRIORITY = 3;①
    ACTIVATION = 1;
    STACK = 512;
    SCHEDULE = NON;②
    TYPE = BASIC;
}

TASK SomeOtherTask {
    PRIORITY = 4;③
    ACTIVATION = 1;
    STACK = 512;
    SCHEDULE = NON;④
    TYPE = BASIC;
}
  
```

- ① Se activa la al tarea SomeOtherTask. Dado que la tarea DemoTask es non preemptive a pesar de tener menor prioridad que SomeOtherTask la ejecución de DemoTask continua.
- ② Para que el sistema evalúe la lista de tareas ready ready es necesario llamar a Schedule
- ③ Al llamar a Schedule el sistema evaluará la lista de tareas ready y pasará a ejecutar SomeOtherTask.

Se puede observar que DemoTask tiene prioridad 3(①) mientras que SomeOtherTask tiene una prioridad de 4(③) y ambas tareas son *non preemptive*(② y ④)

cutar la tarea `SomeOtherTask`, cuando la misma termina se continua la ejecución de `DemoTask`

A.7. TerminateTask

Declaración	StatusType TerminateTask (void);		
Parámetros de entrada	-		
Parámetros de salida	-		
Parámetros de entra- da/salida	-		
Retorna	Estándar	-	No retorna
	Extendida	E_OS_CALLEVEL	La interfaz no fue llamada desde el con- texto de una tarea
		E_OS_RESOURCE	Al menos un recurso sigue siendo ocu- pado por la tarea (ver Sección A.15, “Re- leaseResource”)
Clases	BCC1, BCC2, ECC1, ECC2,		
Descripción	Termina la ejecución de la tarea que llama a este servicio. La tarea es trans- ferida al estado <i>suspended</i> . De no haber errores, el servicio no retorna.		



Es obligatorio terminar todas las tareas utilizando ChainTask ó TerminateTask

Ejemplo A.6. Utilización de TerminateTask

```
TASK(DemoTask)
{
    /* perform task
     * actions */
    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    SCHEDULE = NON;
    TYPE = BASIC;
}
```

A.8. DisableAllInterrupts

Declaración	<code>void DisableAllInterrupts(void);</code>	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	Almacena el estado actual de las interrupciones (para ser restablecidas mediante <code>EnableAllInterrupts</code>) y deshabilita todas las interrupciones	

Ejemplo A.7. Utilización de DisableAllInterrupts

```
TASK(DemoTask)
{
    /* some un-critical code */

    DisableAllInterrupts();

    /* some critical code */

    EnableAllInterrupts();

    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```



Este servicio a diferencia de `SuspendAllInterrupts` `ResumeAllInterrupts` `SuspendOSInterrupts` `ResumeOSInterrupts` *no* soporta anidamiento.

La interfaz puede, por cuestiones de desempeño, ser implementada como macro por lo cual no es recomendable utilizar la dirección de la misma `ptr = (void*)DisableAllInterrupts;`.

Las interfaces `DisableAllInterrupts` y `EnableAllInterrupts` deshabilitan y habilitan todas las interrupciones, tanto las ISR2 como las ISR1.



Siempre y cuando sea posible es preferible utilizar `GetResource` y `ReleaseResource` antes que deshabilitar las interrupciones.

A.9. EnableAllInterrupts

Declaración	void EnableAllInterrupts (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	Restaura el estado de las interrupciones almacenado con la interfaz <code>Disab- leAllInterrupts</code>	

Ejemplo A.8. Utilización de `EnableAllInterrupts`

```
TASK(DemoTask)
{
    /* some un-critical code */
    DisableAllInterrupts();
    /* some critical code */
    EnableAllInterrupts();
    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```



Este servicio a diferencia de `SuspendAllInterrupts` `ResumeAllIn- terrupts` `SuspendOSInterrupts` `ResumeOSInterrupts` *no* soporta anidamiento.

La interfaz puede, por cuestiones de desempeño, ser implementada como macro por lo cual no es recomendable utilizar la dirección de la misma `ptr = (void*)EnableAllInterrupts;`.

Las interfaces `DisableAllInterrupts` y `EnableAllInterrupts` des- habilitan y habilitan todas las interrupciones, tanto las ISR2 como las ISR1.



Siempre y cuando sea posible es preferible utilizar `GetResource` y `Rele- seResource` antes que deshabilitar las interrupciones.

A.10. SuspendAllInterrupts

Declaración	void SuspendAllInterrupts (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	Almacena el estado de las interrupciones y las deshabilita. El estado de las interrupciones se podrá restablecer llamando a ResumeAllInterrupts .	

Ejemplo A.9. Utilización de SuspendAllInterrupts

```
TASK(DemoTask)
{
    /* some un-critical code */
    SuspendAllInterrupts();
    /* some critical code */
    ResumeAllInterrupts();
    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```



La interfaz puede, por cuestiones de desempeño, ser implementada como macro por lo cual no es recomendable utilizar la dirección de la misma `ptr = (void*)SuspendAllInterrupts;`.

Las interfaces `SuspendAllInterrupts` y `ResumeAllInterrupts` deshabilitan y habilitan todas las interrupciones, tanto las ISR2 como las ISR1.



Siempre y cuando sea posible es preferible utilizar `GetResource` y `Releas- seResource` antes que deshabilitar las interrupciones.

A.11. ResumeAllInterrupts

Declaración	void ResumeAllInterrupts (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	Restablece el estado de las interrupciones almacenados en la anterior llama- da a SuspendAllInterrupts.	

Ejemplo A.10. Utilización de **ResumeAllInterrupts**

```
TASK(DemoTask)
{
    /* some un-critical code */
    SuspendAllInterrupts();
    /* some critical code */
    ResumeAllInterrupts();
    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```



La interfaz puede, por cuestiones de desempeño, ser implementada como macro por lo cual no es recomendable utilizar la dirección de la misma `ptr = (void*)ResumeAllInterrupts;`.

Las interfaces `SuspendAllInterrupts` y `ResumeAllInterrupts` des- habilitan y habilitan únicamente todas las interrupciones.



Siempre y cuando sea posible es preferible utilizar `GetResource` y `Rele- seResource` antes que deshabilitar las interrupciones.

A.12. SuspendOSInterrupts

Declaración	void SuspendOSInterrupts (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	Almacena el estado de las interrupciones ISR2 y deshabilita las ISR2. El estado de las ISR2 se podrá restablecer llamando a ResumeOSInterrupts .	

Ejemplo A.11. Utilización de SuspendOSInterrupts

```
TASK(DemoTask)
{
    /* some un-critical code */
    SuspendOSInterrupts();
    /* some critical code */
    ResumeOSInterrupts();
    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```



La interfaz puede, por cuestiones de desempeño, ser implementada como macro por lo cual no es recomendable utilizar la dirección de la misma `ptr = (void*)SuspendOSInterrupts;`.

Las interfaces `SuspendOSInterrupts` y `ResumeOSInterrupts` deshabilitan y habilitan únicamente las ISR2.



Siempre y cuando sea posible es preferible utilizar `GetResource` y `ReleasResource` antes que deshabilitar las interrupciones.

A.13. ResumeOSInterrupts

Declaración	void ResumeOSInterrupts (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2,	
Descripción	Restablece el estado de las interrupciones almacenados en la anterior llamada a SuspendOSInterrupts.	

Ejemplo A.12. Utilización de ResumeOSInterrupts

```
TASK(DemoTask)
{
    /* some un-critical code */
    SuspendOSInterrupts();
    /* some critical code */
    ResumeOSInterrupts();
    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
```



La interfaz puede, por cuestiones de desempeño, ser implementada como macro por lo cual no es recomendable utilizar la dirección de la misma `ptr = (void*)ResumeOSInterrupts;`.

Las interfaces SuspendOSInterrupts y ResumeOSInterrupts deshabilitan y habilitan únicamente las ISR2.



Siempre y cuando sea posible es preferible utilizar GetResource y ReleaseResource antes que deshabilitar las interrupciones.

A.14. GetResource

Declaración	StatusType ReleaseResource(ResourceType ResID)				
Parámetros de entrada	ResID	Recurso a ser liberado			
Parámetros de salida	-				
Parámetros de entrada/salida	-				
Retorna	Estándar	E_OK	Ningún error		
	Extendida	E_OS_ID	El parámetro de entrada ResID es inválido		
		E_OS_NOFUNC	El recurso ResID no está siendo ocupado ú otro recurso debe ser liberado con anterioridad		
		E_OS_ACCESS	El recurso ResID tiene una prioridad inferior prioridad estática de la tarea		
Clases	BCC1, BCC2, ECC1, ECC2,				
Descripción	Es la contraparte de GetResource y sirve para terminar una sección de código crítica indicada mediante el recurso ResID				

Ejemplo A.13. Utilización de GetResource

```
TASK(DemoTask)
{
    /* some un-critical code */

    GetResource(DemoRes);

    /* some critical code */

    ReleaseResource(DemoRes);

    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    RESOURCE = DemoRes;
}

RESOURCE DemoRes;
```



Si la función ReleaseResource retorna E_OS_ACCESS el error se encuentra en el archivo .oil de configuración y no en el código.

Las interfaces GetResource y ReleaseResource pueden, para mejorar el desempeño, almacenar la información en la pila, por ello es recomendable llamar a ambas funciones desde la misma función.



Siempre y cuando sea posible es preferible utilizar GetResource y ReleaseResource, de no ser posible también existe la posibilidad de deshabilitar las interrupciones para acceder a un recurso de forma independiente. Ver: DisableAllInterrupts, EnableAllInterrupts, SuspendAllInterrupts, ResumeAllInterrupts, SuspendOSInterrupts ó ResumeOSInterrupts

A.15. ReleaseResource

Declaración	StatusType ReleaseResource(ResourceType ResID)			
Parámetros de entrada	ResID	Recurso a ser liberado		
Parámetros de salida	-			
Parámetros de entrada/salida	-			
Retorna	Estándar	E_OK	Ningún error	
	Extendida	E_OS_ID	El parámetro de entrada ResID es inválido	
		E_OS_NOFUNC	El recurso ResID no está siendo ocupado ú otro recurso debe ser liberado con anterioridad	
		E_OS_ACCESS	El recurso ResID tiene una prioridad inferior prioridad estática de la tarea	
Clases	BCC1, BCC2, ECC1, ECC2,			
Descripción	Es la contraparte de GetResource y sirve para terminar una sección de código crítica indicada mediante el recurso ResID			

Ejemplo A.14. Utilización de ReleaseResource

```
TASK(DemoTask)
{
    /* some un-critical code */

    GetResource(DemoRes);

    /* some critical code */

    ReleaseResource(DemoRes);

    TerminateTask();
}
```

```
TASK DemoTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
    RESOURCE = DemoRes;
}

RESOURCE DemoRes;
```



Si la función ReleaseResource retorna E_OS_ACCESS el error se encuentra en el archivo .oil de configuración y no en el código.

Las interfaces GetResource y ReleaseResource pueden, para mejorar el desempeño, almacenar la información en la pila, por ello es recomendable llamar a ambas funciones desde la misma función.



Siempre y cuando sea posible es preferible utilizar GetResource y ReleaseResource, de no ser posible también existe la posibilidad de deshabilitar las interrupciones para acceder a un recurso de forma independiente. Ver: DisableAllInterrupts, EnableAllInterrupts, SuspendAllInterrupts, ResumeAllInterrupts, SuspendOSInterrupts ó ResumeOSInterrupts

A.16. SetEvent

Declaración	StatusType SetEvent (TaskType TaskID, EventMaskType Mask);				
Parámetros de entrada	TaskID	Identificador de la tarea a la que se le establecerá el evento			
	Mask	Mascara del evento a establecer			
Parámetros de salida	-				
Parámetros de entrada/salida	-				
Retorna	Estándar	E_OK	Ningún error		
	Extendida	E_OS_ID	El identificador TaskID de tarea es inválido		
		E_OS_ACCESS	La tarea indicada en el parámetro TaskID no es extendida		
		E_OS_STATE	La tarea indicada en el parámetro TaskID se encuentra suspendida		
Clases	ECC1, ECC2				
Descripción	El o los eventos indicados serán establecidos en la tarea TaskID. Si la misma se encuentra en el estado <i>waiting</i> uno de los eventos indicados en Mask la tarea es transferida al estado <i>ready</i>				

Ejemplo A.15. Utilización de SetEvent

```

TASK(DemoTask)
{
    ActivateTask(SomeOtherTask);
    WaitEvent(Event1);
    TerminateTask();
}

TASK(SomeOtherTask)
{
    SetEvent(DemoTask, Event1 | Event2);❶
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
    EVENT = Event1;❷
    EVENT = Event2;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
EVENT Event1;❸
EVENT Event2;

```

- ❶ Se establecen los eventos Event1 y Event2 de DemoTask.

- ❷ Se debe indicar en la declaración de la tarea que eventos son utilizados.
❸ Es necesario definir los eventos.

A.17. GetEvent

Declaración	StatusType GetEvent (TaskType TaskID, EventMaskRefType Event);		
Parámetros de entrada	TaskID	Identificador de la tarea a obtener los eventos	
Parámetros de salida	Event	Eventos establecidos en la tarea TaskID	
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	E_OS_ID	El identificador TaskID de tarea es inválido
		E_OS_ACCESS	La tarea indicada en el parámetro TaskID no es extendida
		E_OS_STATE	La tarea indicada en el parámetro TaskID se encuentra suspendida
Clases	ECC1, ECC2		
Descripción	El estado de los eventos de la tarea TaskID son devueltos en Event		

Ejemplo A.16. Utilización de GetEvent

```

TASK(DemoTask)
{
    ActivateTask(SomeOtherTask);
    EventMaskType Events;

    WaitEvent(Event1 | Event2);
    GetEvent(DemoTask, &Events);①
    ClearEvent(Events);

    if (Events & Event1) {
        /* do something */
    }
    if (Events & Event2) {
        /* do something */
    }

    TerminateTask();
}

TASK(SomeOtherTask)
{
    SetEvent(DemoTask, Event1);
    TerminateTask();
}
  
```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
    EVENT = Event1;②
    EVENT = Event2;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

EVENT Event1;②
EVENT Event2;
  
```

- ① Se debe indicar en la declaración de la tarea que eventos son utilizados.
- ② Es necesario definir los eventos.

- ① Se obtiene el estado de los eventos de DemoTask.

A.18. WaitEvent

Declaración	StatusType WaitEvent (EventMaskType Mask);		
Parámetros de entrada	Mask	Mascara de los eventos a esperar	
Parámetros de salida	-		
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	E_OS_ACCESS	La interfaz debe ser llamar desde una tarea extendida
		E_OS_RESOURCE	La tarea debe retornar todos los recursos antes de llamar esta interfaz
		E_OS_CALLEVEL	La interfaz debe ser llamad desde una tarea
Clases	ECC1, ECC2		
Descripción	La tarea es pasada al estado <i>waiting</i> , hasta que ocurra uno o más de los eventos indicados en Mask. Si uno o más de los eventos indicados se encuentran establecidos al momento de llamar esta interfaz, la misma retorna inmediatamente.		

Ejemplo A.17. Utilización de WaitEvent

```

TASK(DemoTask)
{
    ActivateTask(SomeOtherTask);

    WaitEvent(Event1);①
    TerminateTask();
}

TASK(SomeOtherTask)
{
    SetEvent(DemoTask, Event1);

    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
    EVENT = Event1;②
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

EVENT Event1;③

```

① Se espera la ocurrencia de Event.

① Se debe indicar en la declaración de la tarea que eventos son utilizados.

② Es necesario definir los eventos.

A.19. ClearEvent

Declaración	StatusType ClearEvent (EventMaskType Mask);		
Parámetros de entrada	Mask	Mascara de los eventos a restablecer	
Parámetros de salida	-		
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	E_OS_ACCESS	La interfaz debe ser llamar desde una tarea extendida
		E_OS_CALLEVEL	La interfaz no debe ser llamada desde interrupciones
Clases	ECC1, ECC2		
Descripción	Los eventos son restablecidos acorde al parámetro Mask.		

Ejemplo A.18. Utilización de ClearEvent

```

TASK(DemoTask)
{
    ActivateTask(SomeOtherTask);
    WaitEvent(Event1);①
    TerminateTask();
}
TASK(SomeOtherTask)
{
    SetEvent(DemoTask, Event1);
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
    EVENT = Event1;①
}
TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}
EVENT Event1;②

```

① Se espera la ocurrencia de Event.

① Se debe indicar en la declaración de la tarea que eventos son utilizados.

② Es necesario definir los eventos.

A.20. GetAlarmBase

Declaración	StatusType GetAlarmBase (AlarmType AlarmID, AlarmBase-RefType Info);		
Parámetros de entrada	AlarmID	Identificador de la alarma de la cual se desea leer la base	
Parámetros de salida	Info	Referencia a una estructura donde se almacenara la base de la alarma	
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
	Extendida	E_OS_ID	El parámetro AlarmID es invalido
Clases	BCC1, BCC2, ECC1, ECC2		
Descripción	La información del contador relacionado a la alarma AlarmID es almacenada en Info.		

Ejemplo A.19. Utilización de GetAlarmBase

```
TASK(DemoTask)
{
    AlarmBaseType info;
    GetAlarmBase(Alarm1, &info);①
    TerminateTask();
}
```

- 1 Se leen los parámetros del contador relacionado a la alarma Alarm1. Los elementos de la estructura info.maxallowedvalue, info.ticksperbase y info.mincycle contendrán los valores 1000, 10 y 1 respectivamente.

```
TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
}

TASK PeriodicTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

ALARM Alarm1 {
    COUNTER = HardwareCounter;
    ACTION = ACTIVATETASK {
        TASK = PeriodicTask;
    }
}

COUNTER HardwareCounter {
    MAXALLOWEDVALUE = 1000;①
    TICKSPERBASE = 10;②
    MINCYCLE = 1;③
    TYPE = HARDWARE;
    COUNTER = HWCOUNTER0;
};
```

- 1 Máximo valor que tomará el contador. Será retornado en info.maxallowedvalue.
- 2 Cuantidad de ticks por base, es un valor dependiente de la implementación. Será retornado en info.ticksperbase.
- 3 Mínimo valor en el que se puede asignar una alarma relacionada a este contador. Será retornado en info.mincycle.

A.21. GetAlarm

Declaración	StatusType GetAlarm (AlarmType <i>AlarmID</i> , TickRefType <i>Tick</i>);		
Parámetros de entrada	<i>AlarmID</i>	Identificador de la alarma a leer	
Parámetros de salida	<i>Tick</i>	Ticks restantes para que la alarma expire	
Parámetros de entrada/salida	-		
Retorna	Estándar	<i>E_OK</i>	Ningún error
		<i>E_OS_NOFUNC</i>	La alarma <i>AlarmID</i> no esta siendo utilizada
	Extendida	<i>E_OS_ID</i>	El parámetro <i>AlarmID</i> es invalido
Clases	BCC1, BCC2, ECC1, ECC2		
Descripción	Retorna la cantidad de Ticks restantes hasta expirar la alarma <i>AlarmID</i> .		

Ejemplo A.20. Utilización de GetAlarm

```

TASK(DemoTask)
{
    TickType Ticks;
    StatusType ret;

    ret = GetAlarm(Alarm1,
                  &Ticks);❶
    if (E_OK == ret)
    {
        if (1000 < Ticks)
        {
            CancelAlarm(Alarm1);
            SetRelAlarm(Alarm1,
                        Ticks-500, 0);❷
        }
        else if (E_OS_NOFUNC ==
                 ret)
        {
            SetRelAlarm(Alarm1,
                        500, 0);❸
        }
    }
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

ALARM Alarm1 {
    COUNTER = SoftwareCounter;
    ACTION = ACTIVATETASK {
        TASK = SomeOtherTask ;
    }
}

```

- ❶ Se obtiene la cantidad restante de Ticks para que la alarma expire.
- ❷ Si la alarma se encuentra activa y le restan más de 1000 ticks se reactiva restando 500 Ticks.
- ❸ Si la alarma no se encuentra activa, se la activa para que expire en 500 ticks.

A.22. SetAbsAlarm

Declaración	StatusType SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle);		
Parámetros de entrada	AlarmID	Identificador de la alarma a establecer	
	start	Valor del tick en el que expirará la alarma por primera vez	
	cycle	Valor en ticks para las siguientes expiraciones, en caso de ser 0 la alarma expirará un única vez	
Parámetros de salida	-		
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
		E_OS_STATE	La alarma AlarmID esta siendo utilizada
	Extendida	E_OS_ID	El parámetro AlarmID es invalido
		E_OS_VALUE	El valor start ó cycle se encuentra/n fuera del rango. El rango válido de start es 0 a maxallowedvalue. El rango válido de cycle es 0 ó entre mincycle y maxallowedvalue
Clases	BCC1, BCC2, ECC1, ECC2		
Descripción	Establece la alarma AlarmID para expirar en un tick absoluto indicado (start) y luego cada cycle ticks.		

Ejemplo A.21. Utilización de **SetAbsAlarm**

```

TASK(DemoTask)
{
    if (E_OK != SetAbsAlarm(
        ActivateSomeOtherTask,
        2000,
        500))❶
    {
        /* some error reaction */
    }
    TerminateTask();
}

```

- ❶ Se establece la alarma ActivateSomeOtherTask a expirar en el Tick 2000 y luego cíclicamente cada 500 Ticks.

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

ALARM ActivateSomeOtherTask {
    COUNTER = SoftwareCounter;
    ACTION = ACTIVATETASK {
        TASK = SomeOtherTask ;
    }
}

```

A.23. SetRelAlarm

Declaración	StatusType SetRelAlarm (AlarmType AlarmID, TickType increment, TickType cycle);		
Parámetros de entrada	AlarmID	Identificador de la alarma a establecer	
	start	Valor relativo en ticks en el que expirará la alarma por primera vez	
	cycle	Valor en ticks para las siguientes expiraciones, en caso de ser 0 la alarma expirará una única vez	
Parámetros de salida	-		
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
		E_OS_STATE	La alarma AlarmID esta siendo utilizada
	Extendida	E_OS_ID	El parámetro AlarmID es invalido
		E_OS_VALUE	El valor increment ó cycle se encuentra/n fuera del rango. El rango válido de increment es 0 a maxallowedValue. El rango válido de cycle es 0 ó entre mincycle y maxallowedValue
Clases	BCC1, BCC2, ECC1, ECC2		
Descripción	Establece la alarma AlarmID para expirar en un tick relativo indicado (start) y luego cada cycle ticks.		

Ejemplo A.22. Utilización de SetRelAlarm

```

TASK(DemoTask)
{
    if (E_OK != SetRelAlarm(
        ActivateSomeOtherTask,
        2000,
        500))❶
    {
        /* some error reaction */
    }
    TerminateTask();
}

```

- ❶ Se establece la alarma ActivateSomeOtherTask a expirar en 2000 Ticks y luego cíclicamente cada 500 Ticks.

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
}

TASK SomeOtherTask {
    PRIORITY = 1;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = BASIC;
}

ALARM ActivateSomeOtherTask {
    COUNTER = SoftwareCounter;
    ACTION = ACTIVATETASK {
        TASK = SomeOtherTask ;
    }
}

```

A.24. CancelAlarm

Declaración	StatusType CancelAlarm (AlarmType <i>AlarmID</i>);		
Parámetros de entrada	AlarmID	Identificador de la alarma a cancelar	
Parámetros de salida	-		
Parámetros de entrada/salida	-		
Retorna	Estándar	E_OK	Ningún error
		E_OS_NOFUNC	La alarma <i>AlarmID</i> no esta siendo utilizada
	Extendida	E_OS_ID	El parámetro <i>AlarmID</i> es invalido
Clases	BCC1, BCC2, ECC1, ECC2		
Descripción	Cancela la alarma <i>AlarmID</i> .		

Ejemplo A.23. Utilización de CancelAlarm

```

TASK(DemoTask)
{
    if (E_OK != CancelAlarm(
        ActivateSomeOtherTask))①
    {
        /* some error reaction */
    }
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
}

ALARM ActivateSomeOtherTask {
    COUNTER = SoftwareCounter;
    ACTION = ACTIVATETASK {
        TASK = DemoTask;
    }
}

```

- ① Se lee el modo en el cual se encuentra corriendo el sistema. El valor retornado es el mismo que se utilizo al llamar StartOS.

A.25. GetActiveApplicationMode

Declaración	AppModeType GetActiveApplicationMode (void);	
Parámetros de entrada	-	
Parámetros de salida	-	
Parámetros de entra- da/salida	-	
Retorna	Estándar	Retorna el application mode con el que fue iniciado el sistema operativo.
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	Retorna el application mode con el que fue iniciado el sistema operativo mediante la función StartOS.	

Ejemplo A.24. Utilización de GetActiveApplicationMode

```

TASK(DemoTask)
{
    AppModeType appMode;
    appMode =
        GetActiveApplicationMode();1

    switch(appMode) {
        case AppMode1:
            /* do something */
            break;

        default:
        case AppMode2:
            /* do something else */
            break;
    }
    TerminateTask();
}

```

```

TASK DemoTask {
    PRIORITY = 3;
    ACTIVATION = 1;
    STACK = 512;
    TYPE = EXTENDED;
}

APPMODE AppMode1;

APPMODE AppMode2;

```

- 1** Se cancela la alarma ActivateSomeOtherTask.

A.26. StartOS

Declaración	void startOS (AppModeType Mode);	
Parámetros de entrada	Mode	Modo de aplicación para iniciar el sistema operativo
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	Inicia el sistema operativo en el modo indicado.	

Ejemplo A.25. Utilización de startOS

```
int main(void)
{
    StartOS(AppMode1);1
    /* start os may return */
    while(1) {2
        /* add some halt
           command */
    }
    return 0;
}
```

APPMODE AppMode1;

- 1** Se inicia el sistema operativo en el modo AppMode1.
- 2** StartOS puede o no retornar. En la CIAA-Firmware la función no retorna nunca, pero dado que es específico de cada implementación es conveniente implementar el código en caso de que si retorne.

A.27. ShutdownOS

Declaración	void shutdownOS (StatusType <i>Error</i>);	
Parámetros de entrada	Error	Error ocurrido
Parámetros de salida	-	
Parámetros de entrada/salida	-	
Retorna	Estándar	-
	Extendida	-
Clases	BCC1, BCC2, ECC1, ECC2	
Descripción	El usuario o el mismo sistema pueden llamar a esta interfaz para apagar el sistema. En caso de estar configurada la <code>ShutdownHook</code> la misma es llamada desde el contexto de esta interfaz.	

Ejemplo A.26. Utilización de `shutdownOS`

```
TASK(MyTask)
{
    if (shutdownCondition)
    {
        ShutdownOS(E_OK);❶
    }
    /* do something */
    TerminateTask();
}
```

- ❶ Se apaga el sistema.

Bibliografía

- [AUTOSAR] AUTOSAR Webpage <http://www.autosar.org/>. AUTOSAR Webpage Copyright © 2014 AUTOSAR. *AUTOSAR (AUTomotive Open System ARchitecture)*.
- [AUTOSAR-OS] AUTOSAR-OS http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf. AUTOSAR-OS Copyright © 2014 AUTOSAR. *Specification of Operating System*.
- [CIAA-Firmware] CIAA Firmware en GitHub <https://github.com/ciaa/Firmware>. CIAA Firmware en GitHub Copyright © 2014-2015 CIAA. *Computadora Industrial Abierta Argentina*.
- [ERIKA] Evidence S.r.l.<http://www.evidence.eu.com/>. Copyright © 2005-2012 Evidence S.r.l.. Diciembre de 2012. *ERIKA Enterprise Manual*. Real-time made easy http://download.tuxfamily.org/erika/webdownload/manuals_pdf/ee_refman_1_4_5.pdf. Version 1.4.5.
- [FEPE] Software Engineering Institute - Carnegie Mellon University <http://www.sei.cmu.edu/>. Copyright © 2003 Carnegie Mellon University. Noviembre de 2003. Peter H. Feiler. *Real-Time Application Development with OSEK A Review of the OSEK Standards*. Performance-Critical Systems Initiative <http://www.sei.cmu.edu/reports/03tn004.pdf>. CMU/SEI-2003-TN-004.
- [gobx] GOBX WebPage <http://gobx.sourceforge.net/>. gobx Copyright © 2008-2009 Sam Wang. *The free OSEK configuration builder*.
- [OSEK-OIL] OSEK-VDX <http://www.osek-vdx.org>. Copyright © 2004 OSEK. *System Generation*. OIL: OSEK Implementation Lanaguage. Version 2.5.
- [OSEK-OS] OSEK-VDX <http://www.osek-vdx.org>. Copyright © 2005 OSEK. *Operating System Specification*. Version 2.2.3.
- [OSEK-TM] OSEK-VDX <http://www.osek-vdx.org>. Copyright © 1999 OSEK. 16/04/1999. *OSEK/VDX Conformance Testing Methodology*. Version 2.0.
- [OSEK-TPlan] OSEK-VDX <http://www.osek-vdx.org>. Copyright © 1999 OSEK. 16/04/1999. *OSEK/VDX OS Test Plan*. Version 2.0.
- [OSEK-TProc] OSEK-VDX <http://www.osek-vdx.org>. Copyright © 1999 OSEK. 16/04/1999. *OSEK/VDX OS Test Procedure*. Version 2.0.

Índice

A

activate, 17
ActivateTask, 54
Alarmas, 33
 jitter, 35
 offset, 34
AlarmBaseType, 53
AUTOSAR, 3
AUTOSAR-OS, 3

B

basic, 18
BCC1, 5
BCC2, 5

C

CancelAlarm, 78
CC (ver Clases de Conformidad)
Ceiling Protocol, 26
ChainTask, 55
Chequeo de errores, 45
 estándar, 45
 extendido, 45
CIAA-Firmware, 4
Clases de Conformidad, 5
 CIAA-Firmware, 6
ClearEvent, 73
Configuración, 8

D

deadlocks, 25
DisableAllInterrupts, 62

E

ECC1, 5
ECC2, 5
EnableAllInterrupts, 63
ErrorHook, 43, 46
estático, 3
extended, 18

F

fork, 3
FreeOSEK, 8
FreeRTOS, 4

G

generación, 9
GetActiveApplicationMode, 79
GetAlarm, 75
GetAlarmBase, 74

GetEvent, 71
GetResource, 28, 68
GetTaskID, 57
GetTaskState, 58
gobox, 9

I

inversión de prioridades, 25, 25
ISR1, 36
ISR2, 36

L

Linux, 3

M

make, 10
maxallowedvalue, 53
mincycle, 53

N

NON PREEMPTIVE, 22, 29

O

OIL, 7
OSEK-OS, 2, 4

P

php, 8, 9
PostTaskHook, 42
preempt, 17
PREEMPTIVE, 22, 29
PreTaskHook, 41

R

ready, 16, 17, 17, 17
recursos, 13, 29, 31
 en interrupciones, 32
 internos, 29
 linkeados, 31
release, 17
ReleaseResource, 28, 69
Requerimientos clase de conformidad, 5
ResumeAllInterrupts, 65
ResumeOSInterrupts, 67
running, 16, 17, 17

S

Schedule, 59
Scheduler, 14
scheduling
 FULL, 14
 NON, 14
Scheduling points, 24
SetAbsAlarm, 76
SetEvent, 70
SetRelAlarm, 77

ShutdownHook, 44
ShutdownOS, 81
Sistema Operativo generado, 13
start, 17
StartOS, 80
StartupHook, 39, 40
StatusType, 52
SuspendAllInterrupts, 64
suspended, 16, 17
SuspendOSInterrupts, 66

T

Tareas, 14
templates, 8
terminate, 18
TerminateTask, 61
TickRefType, 53
ticksperbase, 53
TickType, 53
Tipos de tareas, 5

W

wait, 17
WaitEvent, 72
waiting, 16, 17, 17, 18
while(1), 16
Windows, 3

Una introducción a OSEK-OS, el estándar del sistema operativo de tiempo real utilizado en el CIAA-Firmware. En el libro se tratan los conceptos de este estándar, que a diferencia de otros, especifica un sistema operativo estático. Al ser estático ofrece al usuario grandes ventajas a un costo mínimo o nulo, siempre y cuando sea utilizado en sistemas embebidos de usos específicos. Además se incluyen una gran cantidad de ejemplos basados en FreeOSEK, la implementación de OSEK-OS del CIAA-Firmware.



Mariano Cerdeiro es egresado en Ingeniería en Electrónica de la Universidad Tecnológica Nacional, Facultad Regional Buenos Aires del 2004. Luego realizó una Maestría en Tecnología de la Información en la Universidad de Ciencias Aplicadas de Mannheim, Alemania. Desde entonces se dedica al desarrollo de Software Embebido en el área de la industria automotriz. En abril de 2014 comenzó a participar en el desarrollo del CIAA-Firmware.

ISBN 978-987-45523-6-5

A standard linear barcode representing the ISBN number 978-987-45523-6-5.

9 789874 552365