Práctica 2 Conociendo el entorno de un Sistema Operativo en Tiempo Real

# Definición de un sistema de tiempo real [Burns, Welling, 2009]

Todas las interpretaciones de la naturaleza exacta de un sistema de tiempo real tienen en común la noción de tiempo de respuesta – el tiempo que le toma a un sistema para generar una salida a partir de alguna entrada asociada.

Diccionario Oxford de Computación:

“Cualquier sistema en el que el tiempo en el cual se produce la salida es significativo. Esto es usualmente porque la entrada corresponde a algún movimiento en el mundo físico, y la salida tiene que relacionarse con ese mismo movimiento. El retardo entre el tiempo de aplicación de la entrada y el tiempo de aparición de la salida debe ser suficientemente pequeño para que la puntualidad sea aceptable.”

La palabra puntualidad debe ser interpretada en el contexto del sistema total. Por ejemplo, en un sistema de guía de misil, la salida es requerida dentro de unos cuantos milisegundos, mientras que, en una línea de ensamble de autos, la respuesta podría ser requerida solamente dentro de un segundo.

En un sentido general, las definiciones de sistema de tiempo real cubren un rango amplio de actividades de computadora. Los sistemas de tiempo real se distinguen de otros sistemas en los que la respuesta a tiempo es importante pero no crucial. Consecuentemente, el funcionamiento correcto de un sistema de tiempo real depende no solamente del resultado lógico de la computación, sino también del tiempo en el cual se producen los resultados. Los practicantes en el campo del diseño de sistemas de computadora de tiempo real frecuentemente distinguen entre sistemas de tiempo real **estrictos** y sistemas de tiempo real **suaves** (**hard** and **soft** **real-time systems**). Los sistemas de tiempo real estrictos son aquellos en donde es absolutamente imperativo que la respuesta ocurra dentro del tiempo límite (deadline). Los sistemas de tiempo real suaves son aquellos donde los tiempos de respuesta son importantes pero el sistema aún funcionará correctamente si ocasionalmente no se cumple con los plazos de finalización (deadline). Los sistemas suaves se pueden distinguir de los sistemas interactivos en los cuales no hay deadlines explícitos. Por ejemplo, el sistema de control de vuelo de una aeronave de combate es un sistema de tiempo real estricto porque un deadline incumplido podría conducir a una catástrofe, mientras que un sistema de adquisición de datos para una aplicación de control de proceso es suave, dado que podría estar definido para muestrear un sensor de entrada a intervalos regulares, pero para tolerar retardos intermitentes.

En un sistema de tiempo real (estricto o suave), usualmente la computadora tiene una interface directamente a algún equipo físico y está dedicada a monitorear o controlar la operación de ese equipo. Una característica de este tipo de aplicaciones es el rol de la computadora como un componente de procesamiento de información dentro de un sistema de ingeniería más grande. Esta es la razón por la que tales aplicaciones son conocidas como sistemas de computadoras inmersas.

Otra forma de clasificar el rol que tiene el tiempo en los sistemas de tiempo real es distinguir entre **reactive systems** y **time-aware systems**. Los time-aware systems hacen referencias explícitas al marco de tiempo del ambiente circundante –los valores de tiempo absoluto deben estar disponibles para el sistema. Un reactive system típicamente tiene que ver con tiempos relativos, por ejemplo, una salida tiene que ser producida a no más de 50 ms de una entrada asociada. Frecuentemente los sistemas reactivos también son sistemas de control y por lo tanto necesitan estar sincronizados con su ambiente.

Los reactive systems frecuentemente son estructurados para ser disparados por tiempo (**time-triggered**). Todas las actividades de computación son periódicas en el sentido de que tienen un tiempo de ciclo definido, por ejemplo, 50 ms, y su ejecución es liberada por un reloj interno. La alternativa a time-triggered es **event-triggered** en la cual el ambiente explícitamente controla (tal vez a través de una interrupción) la liberación de la ejecución de alguna actividad software. Estas actividades son denominadas **aperiódicas**, y son llamadas **esporádicas** si hay una cota sobre qué tan frecuentemente el evento liberador puede ocurrir en algún intervalo de tiempo. Muchos sistemas contendrán actividades periódicas y esporádicas. Sin embargo, algunos enfoques de diseño restringen la arquitectura del software de tal manera que hay solamente actividades time-triggered; y se utiliza una técnica denominada polling para gestionar eventos (esto es, examinados a través de una actividad periódica).

En un sentido amplio [Burns, Welling, 2009], los sistemas de tiempo real incluyen **reactive systems** y **time-aware systems** que pueden tener invocaciones de trabajo **time-triggered** y **event-triggered**. En un sistema de tiempo real se tienen actividades **periódicas**, **aperiódicas** y **esporádicas**.

# Sistemas Operativos de tiempo real

Como se mencionó antes, muchos sistemas de tiempo real están compuestos de tareas de tiempo real estricto y tareas de tiempo real suave. Si esas tareas tienen un propósito único, se pueden implementar como segmentos de programa semi-independientes. En tal caso, la lógica de utilización del procesador tiene que estar dentro de las tareas de aplicación. Las implementaciones de este tipo, típicamente toman la forma de un ciclo de control que continuamente revisa qué tarea ejecutar. Tales técnicas sufren de numerosos problemas y no representan una solución para aplicaciones de tiempo real regulares. Además, complican el mantenimiento y la reutilización del software.

Un Sistema Operativo de Tiempo Real (SOTR) es un sistema operativo dedicado completamente diseñado para superar las restricciones de tiempo de un sistema de tiempo real. Un SOTR, como cualquier otro sistema operativo proporciona un ambiente en el cual se puede ejecutar programas de una manera conveniente y estructurada, pero sin el riesgo de incumplir con las restricciones de tiempo real.

## En general, los beneficios de usar un SOTR son:

-Un SOTR elimina la necesidad de asignación del procesador en el software de aplicación.

-Las modificaciones o adiciones de tareas completamente nuevas pueden ser hechas en el software de aplicación sin afectar requerimientos críticos de respuesta del sistema.

-Además de manejar la ejecución de tareas, la mayoría de los SOTR también proporcionan facilidades que incluyen comunicación entre tareas, sincronización de tareas, temporizadores, administración de memoria, etc.

-Un SOTR oculta detalles específicos del hardware subyacente al usuario ofreciendo un ambiente de tiempo de ejecución que es completamente independiente del procesador objetivo.

## Respuesta de tiempo real de un SOTR

Los SOTR deben ser expulsivos (preemtive). Un SOTR expulsivo garantiza que una tarea de prioridad más alta puede interrumpir una tarea de prioridad más baja. Por supuesto, hay estados en los cuales no se permite que esto pase. Por ejemplo, cuando se están manejando interrupciones, mientras el SOTR está manejando el scheduler, o mientras el SOTR está accediendo a sus secciones críticas internas. Esos estados son llamados ‘system locks’. En todos los otros casos, la expulsión debe ser permitida. Obviamente, siempre que el sistema sale de uno de esos system locks, una prueba debe determinar si la tarea actual debe ser expulsada por otra tarea con prioridad más alta. Fuera de un system lock, la expulsión sucede automáticamente como parte del servicio de interrupción que causa que una tarea de prioridad más alta se convierta en ejecutable (esto también es llamado ‘event-driven preemtion’).

A diferencia de los sistemas operativos de escritorio en los cuales el código de aplicación corre en modo usuario y el código de kernel corre en modo kernel (modo privilegiado), las tareas de tiempo real estricto siempre corren en modo kernel. De esta forma, las tareas de tiempo real estricto siempre pueden interceptar interrupciones (event-driven preemtion) y pueden invocar llamadas al sistema muy rápido porque no es necesario usar costosas interrupciones software y cambios de contexto para switchear entre el modo kernel y el modo usuario (y viceversa). Para una tarea de tiempo real estricto una llamada al sistema ya no es una trampa software sino una llamada a función.

## Compartición de memoria y otros recursos

Las tareas de tiempo real estricto tienen acceso privilegiado al espacio de memoria completo sin aislamiento de direcciones o protección de memoria. En sistemas de tiempo real estricto, se supone que todo el código de aplicación es de tiempo real estricto, lo cual efectivamente significa que todas las tareas y el sistema operativo comparten el mismo espacio de direcciones físicas. Conceptos usuales, tales como procesos (UNIX) o mecanismos unidad de administración de memoria (MMU), simplemente no existen en este contexto. O, equivalentemente, podríamos decir que, en sistemas de tiempo real estricto, todas las tareas comparten la información de estado de un solo proceso, y todas las tareas comparten la memoria y otros recursos directamente. El cambio de contexto entre tareas en un sistema de tiempo real estricto es mucho más rápido que el cambio de contexto entre procesos.

Entonces, una tarea de tiempo real podría fácilmente sobrescribir el espacio de direcciones de otra tarea y/o el sistema operativo. Durante la fase de desarrollo, siempre se debe estar al tanto del problema de corrupción de datos (desbordamientos de pila de hilo –thread stack--, o accesos de memoria basados en apuntador incorrecto). Estos tipos de errores de programación típicamente son extremadamente difíciles de detectar y provocarán fallas críticas una vez que el sistema completo ha sido desplegado en el campo.

## ¿Qué es POSIX?

Aunque se originó para referirse al original IEEE Std 1003.1-1998, el nombre POSIX más correctamente se refiere a la familia de estándares relacionados: IEEE Std 1003.n (donde n es un número). POSIX es desarrollado por el Portable Applications Standard Committe (PASC) de la IEEE Computer Society.

POSIX está basado en UNIX, una tecnología bien establecida que data de principios de los 1970s, y define una forma estándar en la que las aplicaciones hacen interface con el sistema operativo.

Entonces, POSIX es un conjunto de libros que especifican APIs, no es una pieza de código ni un sistema operativo. Los estándares POSIX no especifican como se deben implementar los servicios de un SOTR, solo especifican sus semánticas. Los implementadores pueden elegir su implementación siempre y cuando sigan la especificación de la interface.

Los estándares POSIX pueden ser agrupados en tres categorías:

* **Base standards**. Estos estándares definen las interfaces de sistema relacionadas a diferentes aspectos del sistema operativo. El estándar especifica la sintaxis y la semántica de las interfaces de sistema de tal manera que programas de aplicación pueden invocar directamente servicios del sistema operativo.

POSIX.1 es el System Interface, el estándar de referencia básico. La versión mas reciente de POSIX.1 es el IEEE Std 1003.1 2013 Edition.

* **Language bindings**. Estos estándares proporcionan las interfaces reales para diferentes lenguajes de programación: C, Ada, Fortran 77, Fortran 90, …
* **Operating system environment**. Estos estándares incluyen una guía para el ambiente POSIX y perfiles de aplicación (application profiles). Un perfil de aplicación es una lista de estándares POSIX que son requeridos para un cierto ambiente de aplicación, junto con las opciones y parámetros de esos estándares cuyo soporte es requerido para un ambiente de aplicación. Los perfiles de aplicación son medios muy importantes para conseguir un número pequeño de tipos bien definidos de implementaciones de sistema operativo apropiados para ambientes de aplicación particulares.

POSIX.13 es el estándar que describe los Perfiles de ambiente de Aplicación de Tiempo-Real (Real-Time Application Environment Profiles). Uno de los POSIX.13 más recientes es el POSIX.13 IEEE Std 1003.13-2003 edition.

## POSIX Real-time Profiles

El conjunto completo de servicios POSIX es útil para aplicaciones grandes, pero el conjunto es considerado demasiado grande para la mayoría de los sistemas inmersos. Los sistemas inmersos usualmente tienen requerimientos de memoria ajustados, podrían no tener capacidades de manejo de memoria, y podrían no tener una memoria para implementar el sistema de archivo UNIX.

Por estas razones, el estándar POSIX reconoce la necesidad de la creación de subconjuntos de los servicios de sistema operativo. Estos deben ser subconjuntos estándar que permitan la portabilidad de aplicaciones de una implementación a otra.

El estándar IEEE1003.13 (POSIX.13, 2003) describe cuatro Perfiles de Ambiente de Aplicación de tiempo-real (real-time Application Environment Profile) (AEPs) y sus requerimientos de hardware mínimos:

Minimal Real-time System AEP (PSE51)

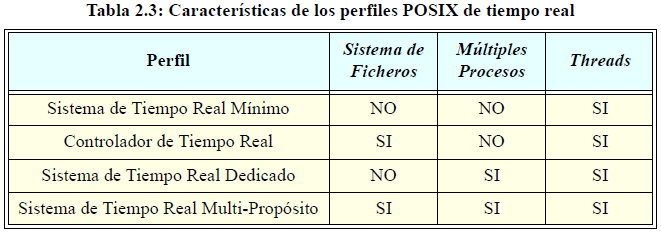
Real-time Controller System AEP (PSE52)

Dedicated Real-time System AEP (PSE53)

Multi-Purpose Real-time System AEP (PSE54)

Cada perfil es una extensión del perfil previo. Para más detalles véase el IEEE Std 1003.13 – 2003 IEEE Standard for InformationTechnology – Standardized Application Environment Profile (AEP) – POSIX Realtime and Embedded Application Support.

Las principales diferencias entre los citados perfiles radican en la existencia o no de múltiples procesos y de un sistema de archivos complejo y jerárquico. La tabla 2.3 muestra las diferencias de los cuatro perfiles de tiempo real respecto a las citadas características.



El menor de dichos perfiles, el denominado “Sistema de Tiempo Real Mínimo”, está pensado para aplicaciones inmersas pequeñas. Entre otras características, no requiere el soporte para múltiples procesos ni para un sistema de archivos jerárquico. Con estas dos simplificaciones se elimina la mayor parte de la complejidad de un sistema POSIX completo, permitiendo por tanto la implementación de un kernel de sistema operativo pequeño y eficiente, aplicado para pequeñas aplicaciones inmersas. Aplicaciones típicas de este perfil serían el control de electrodomésticos (como microondas o televisores), terminales de punto de venta, controladores de motores de automóviles, controladores de sensores industriales, etc.

Descripción del sistema de tiempo real mínimo

Como se mencionó anteriormente, el perfil de “Sistema de Tiempo Real Mínimo”, está pensado para aplicaciones inmersas pequeñas, por lo que en él se incluyen únicamente aquellos servicios POSIX que se consideran necesarios para las aplicaciones a las que está destinado. Con el reducido conjunto de servicios soportados, el estándar POSIX.13 pretende que un kernel de sistema operativo compatible con el “Sistema de Tiempo Real Mínimo” pueda ser de muy reducido tamaño y altamente eficiente. A continuación, procederemos a describir los principales servicios incluidos en este perfil:

**Soporte para ejecución concurrente.** El único mecanismo de ejecución concurrente facilitado son los denominados hilos de control o threads. Se incluyen servicios para su creación y terminación, así como para la gestión de sus atributos. También existe la posibilidad de asociar información específica con los threads. Otra funcionalidad incluida en este perfil permite a un thread esperar la terminación de otro.

**Planificación de threads.** La política de planificación de threads es expulsiva por prioridades fijas, pudiendo especificarse dos comportamientos diferentes entre threads de la misma prioridad: FIFO o “round robin”. Un thread de política FIFO (SCHED\_FIFO) se mantendrá en ejecución hasta que se bloquee o hasta que sea expulsado por otro thread de mayor prioridad. Los threads con política “round-robin” (SCHED\_RR) se comportan igual que los FIFO salvo por una limitación adicional: solo se pueden mantener en ejecución de forma continuada durante un intervalo máximo de tiempo definido por la implementación. Al finalizar dicho intervalo de tiempo, el thread deberá ceder la CPU a otros threads de su misma prioridad, solo en el caso de que no haya ninguno, el thread original retomará la CPU para comenzar otro intervalo.

El número mínimo de niveles de prioridad diferentes requeridos por el estándar es 32. Con este número es posible lograr altos niveles de utilización del procesador incluso para números muy elevados de threads. El perfil mínimo incluye también servicios que permiten especificar los parámetros y políticas de planificación, para lo cual el estándar define los siguientes atributos:

Política de planificación: pudiéndose elegir entre las políticas FIFO y “round-robin”.

Parámetros de planificación: para las políticas anteriormente citadas el único parámetro definido es su prioridad.

Herencia de los atributos de planificación: permite indicar si los atributos serán heredado del thread padre o son los indicados explícitamente por su objeto de atributos.

**Sincronización entre threads.** Los servicios de sincronización incluidos en el perfil mínimo son los mutexes, las variables de condición y los semáforos contadores. Los mutexes son utilizados para asegurar la exclusión mutua entre threads accediendo a un mismo recurso. Para ellos se definen dos protocolos que permiten evitar la inversión de prioridad: la herencia básica de prioridad (PTHREAD\_PRIO\_INHERIT) y el techo protección inmediato (PTHREAD\_PRIO\_PROTECT). Por su parte las variables condicionales permiten implementar mecanismos de sincronización entre threads basados en señalización y espera. Los semáforos contadores constituyen un mecanismo de sincronización para exclusión mutua y señalización y espera menos evolucionado que los anteriores, adoleciendo de inversión de prioridad y resultando su utilización más compleja y más proclive a errores.

**Señales.** Las señales POSIX son utilizadas para notificar a la aplicación la ocurrencia de un evento. El perfil mínimo proporciona servicios para enviar, esperar y enmascarar señales, soportando también la instalación de manejadores de señal. Sin embargo, en sistemas en los que existe la concurrencia, la ejecución asíncrona de manejadores es preferiblemente reemplazada por operaciones de espera síncrona. La segunda estrategia tiene la ventaja de que los parámetros de planificación bajo los cuales es servida una señal están determinados por los atributos del thread encargado de su gestión.

**Acceso a dispositivos.** Aunque el perfil mínimo no requiere la existencia de un sistema de archivos completo, sí que se incluyen las operaciones básicas que permiten acceder a los dispositivos (open, close, write, read). La operación open está restringida a la apertura de archivos ya existentes, definidos durante la configuración del sistema, no pudiendo ser usada para crear nuevos archivos.

**Servicios de temporización.** Se incluye una operación de suspensión temporizada de tareas en la que el intervalo de suspensión puede ser especificado con alta precisión. También existen operaciones para la lectura de un reloj de alta resolución y para manejo de temporizadores. Los temporizadores sirven para informar a la aplicación de que un intervalo de tiempo dado ha transcurrido o de que se ha alcanzado un tiempo determinado. En ambos casos se puede especificar que se desea utilizar una señal como forma de notificación de la expiración del temporizador.

**Paso de mensajes.** Se encuentran definidas operaciones para utilizar el sistema de colas de mensajes. Los mensajes incluyen un campo de prioridad que es usado para indicar el orden de recepción. Por otra parte, en el caso de que se desee utilizar colas de mensajes, éstas pueden ser fácilmente implementadas por la aplicación utilizando mutexes y variables condicionales.

**Servicios de configuración.** Se proporcionan operaciones que permiten a la aplicación conocer información relativa a la configuración del sistema.

**Gestión de memoria dinámica.** Aunque el estándar POSIX no define ninguna interfaz para la gestión de memoria dinámica, si que requiere que estén soportadas las operaciones de memoria dinámica específicas de los lenguajes de programación, como malloc() para el lenguaje C o new para el Ada.

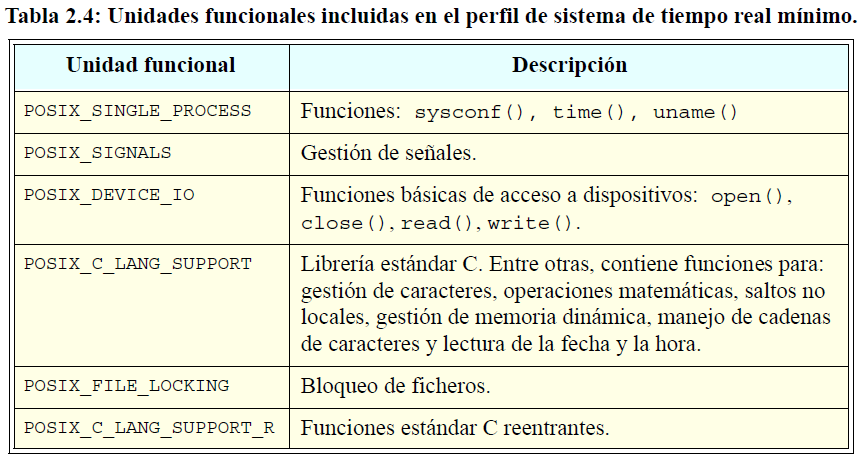
Además de los servicios anteriormente citados, el estándar POSIX.13 requiere que algunos otros servicios también sean soportados en el perfil mínimo por razones de compatibilidad con los demás perfiles:

**Bloqueo de memoria.** Estos servicios evitan los tiempos de acceso a memoria no acotados debidos a la utilización de memoria virtual, permitiendo mantener en memoria física el espacio de direcciones de los procesos de tiempo real. Aunque lo normal en las implementaciones de sistemas mínimos de tiempo real es que no soporten memoria virtual, la interfaz es incluida por razones de compatibilidad con el resto de los perfiles. Se recomienda a las aplicaciones la utilización de estos servicios incluso en sistemas en los que su implementación sea nula.

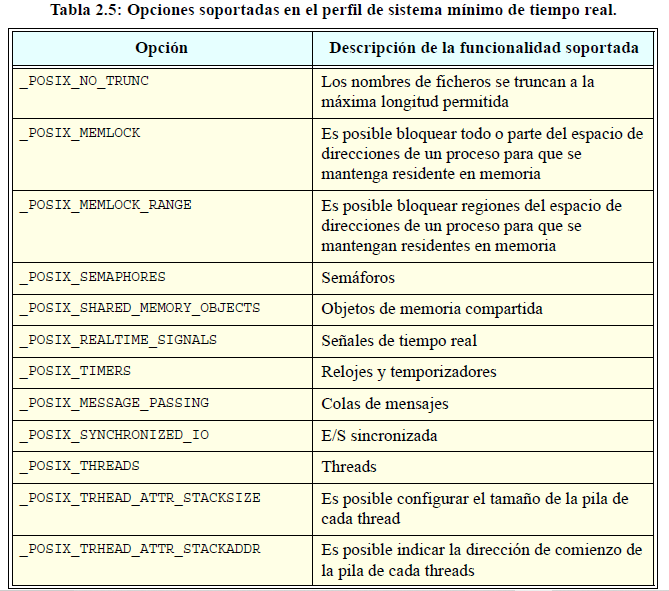
**Entrada/Salida sincronizada y no sincronizada.** Lo normal en sistemas inmersos pequeños será que la E/S se haga directamente sobre el dispositivo sin ningún tipo de almacenamiento intermedio, por tanto toda la E/S de este tipo de sistemas será sincronizada por omisión. Aún así, y al igual que en el cas anterior, se recomienta a las aplicaciones que expresamente requieran E/S sincronizada para mantener la compatibilidad con los restantes perfiles.

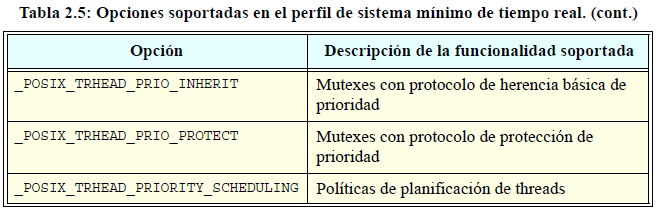
**Objetos de memoria compartida.** Pueden ser utilizados para implementar E/S mapeada en memoria, lo cual es muy común en muchas arquitecturas inmersas. Aunque la memoria pudiera ser directamente accesible por las aplicaciones, se recomienta realizar la E/S mapeada en memoria utilizando objetos de memoria compartida. De esta forma resultará posible la portabilidad de la aplicación a sistemas mayores en los que está prohibido el acceso directo a dispositivos mapeados en memoria. La implementación de los servicios descritos en sistemas sin protección, en los que toda la memoria es accesible, resulta extremadamente sencilla.

El estándar POSIX.13 define formalmente cada perfil de sistema de tiempo real en base a las unidades funcionales que incluye y las opciones que soporta. La tabla 2.4 muestra las unidades funcionales del estándar POSIX.1 que se encuentran incluidas en el perfil de sistema de tiempo real mínimo.



Las opciones que se encuentran soportadas en el perfil de sistema mínimo de tiempo real son las mostradas en la tabla 2.5.





Referencia. [Pag. 41/332 de Tesis de MAR].

## Diferencias entre un kernel de tiempo real y un kernel normal

Un sistema operativo (SO) de tiempo real [Dietrich y Walker, 2015] es un SO que se construye para aplicaciones de tiempo real. Las aplicaciones de tiempo real usualmente tienen dos requerimientos de tiempo muy estrictos: tiempo de respuesta rápido y plazo de finalización (deadline) garantizado. Específicamente, un SO de tiempo real debe ser capaz de

* Responder a peticiones de interrupción de eventos de tiempo real dentro de un tiempo límite muy corto.
* Completar un servicio pedido dentro de un cierto tiempo límite, conocido como plazo de finalización (deadline) de tarea.

Si un sistema siempre puede cumplir estos requerimientos críticos, entonces el sistema es llamado **sistema de tiempo real estricto**. Si un sistema solo puede cumplir estos requerimientos la mayor parte del tiempo, pero no siempre, entonces el sistema es llamado **sistema de tiempo real suave**. Para poder cumplir con esos requerimientos de tiempo críticos, un sistema de operativo de tiempo real usualmente es diseñado con las siguientes capacidades.

* Latencia de respuesta de interrupción y tiempo de cambio de tarea (task switch) mínimos: un kernel de SO de tiempo real no debe enmascarar interrupciones por periodos de tiempo largos. El código usado para implementar regiones críticas y cambio de tareas (task switching) debe ser muy corto. Todas las regiones críticas deben ser tan cortas como sea posible.
* Un scheduler de tareas avanzado: el scheduler debe soportar planificación de tareas expulsiva con un algoritmo de planificación adecuado basado en el plazo de finalización (deadline), tal como el algoritmo Earliest Deadline First (EDF) [Liu y Layland, 1973]. La planificación (o scheduling) expulsiva permite que tareas de prioridad más alta expulsen tareas de prioridad más baja en cualquier momento. Este tipo de planificación es una condición necesaria, aunque no suficiente, para operaciones de tiempo real. Sin planificación expulsiva sería imposible cumplir con los deadlines de tareas.
* Para asegurar respuesta rápida, las tareas en un SO de tiempo real usualmente no tienen un modo de usuario separado. Todas las tareas corren en el mismo espacio de direcciones del kernel del SO.

Debido a sus requerimientos y objetivos únicos, el diseño e implementación de sistemas operativos de tiempo real difiere del de sistemas operativos de propósito general. Sin embargo, en ambos casos, también existen muchas cosas en común, especialmente en el área de sincronización de procesos, tales como, regiones críticas, protección de estructuras de datos para soportar ejecuciones concurrentes, prevención de deadlocks y de condiciones de carrera, etc.

# Micro C/OS-II (Port para Linux)

Para realizar la construcción de los ejemplos 0, 1, 2, 3, y 4 de un port para Linux del sistema operativo de tiempo real Micro C/OS-II, primero se deberá colocar los archivos fuentes del port y de los archivos independientes de la arquitectura en los directorios adecuados y crear un enlace simbólico. Los archivos fuente a los que se hará referencia en los siguientes párrafos se pueden descargar del siguiente enlace:

https://github.com/sotrteacher/dirtywork/tree/master/PRACTICA\_2\_Conociendo\_el\_Entorno\_de\_un\_SOTR/software.tar

Una vez descargado el archivo software.tar, colocarlo en el directorio de inicio de su usuario: /home/usuario/

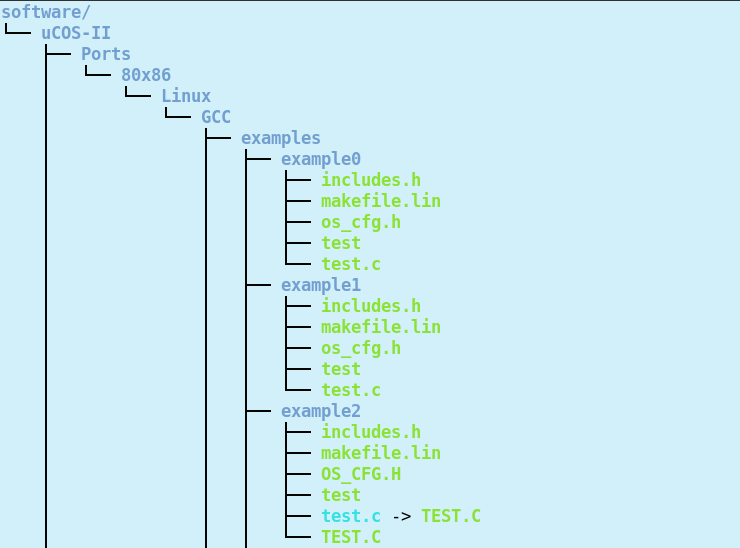
Extraer el contenido del archivo tar con el comando:

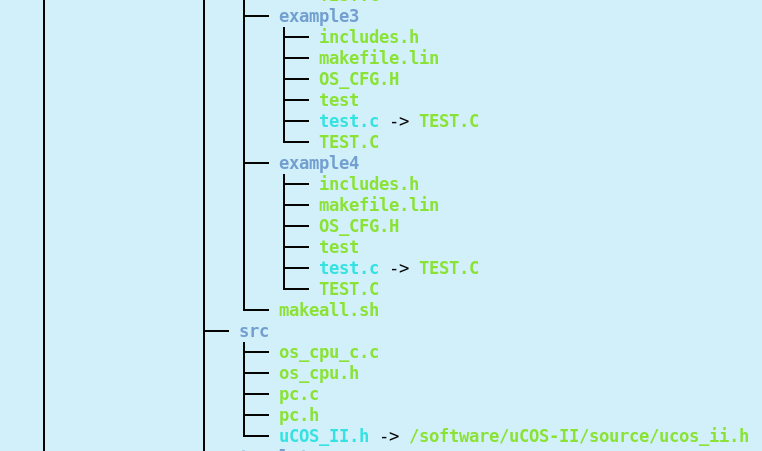
$ tar xvf software.tar

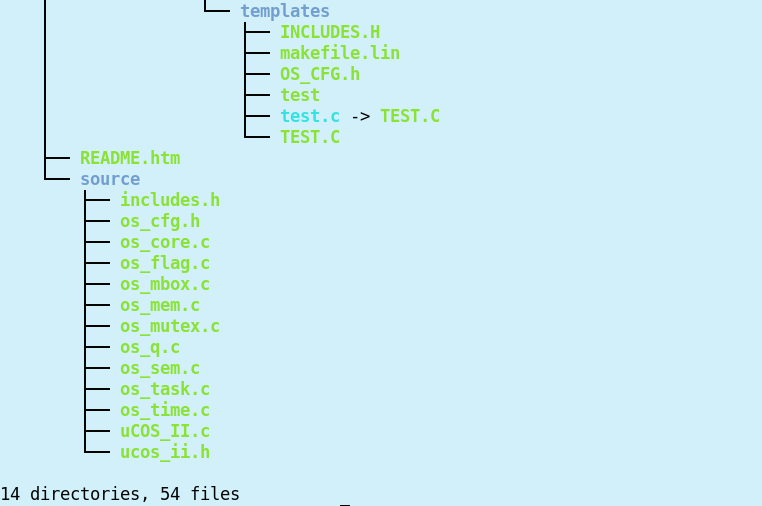
La salida del comando

$ tree software/

debe ser la siguiente:







Salida del comando tree software/

Después, deberemos crear un enlace simbólico en el directorio raíz del sistema de archivos con el comando:

$ sudo ln –s /home/usuario/software /software

/software -> /home/usuario/software

$ ls –hgo / |grep software



## Ejemplo #0

El ejemplo #0 está en el directorio

/software/uCOS-II/Ports/80x86/Linux/GCC/examples/example0/

$ cd /software/uCOS-II/ Ports/80x86/Linux/GCC/examples/example0/

Ya en el directorio del ejemplo 0, para construir el mismo solo se necesita ejecutar make como sigue:

$ make –f makefile.lin

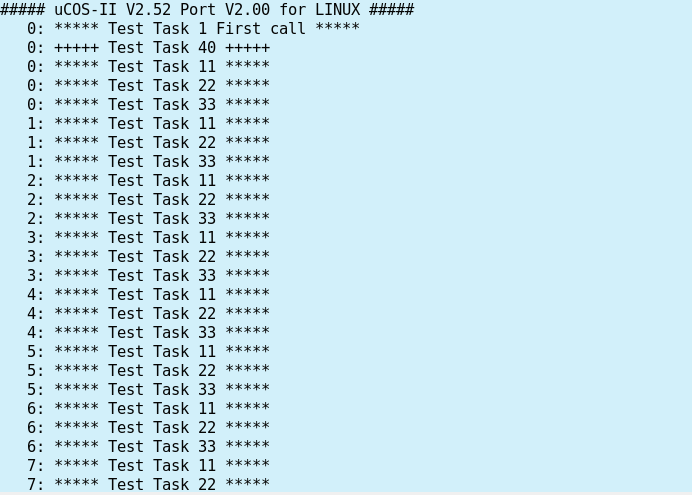
En respuesta a este comando se puede observar algo como lo siguiente:



Para correr el programa del ejemplo 0:

$ ./test

La salida del programa debe ser algo como lo que se muestra a continuación:



Para finalizar la ejecución del programa se presiona cualquier tecla.

El código de la función main en el archivo

/software/uCOS-II/examples/example0/test.c

es:

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* uC/OS-II

\* The Real-Time Kernel

\*

\* WIN32 PORT & LINUX PORT

\* (c) Copyright 2004, Werner.Zimmermann@fht-esslingen.de

\* All Rights Reserved

\*

\* EXAMPLE #0

\* Used to study the scheduling behaviour of the WIN32 and LINUX port during development

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#include "includes.h"

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CONSTANTS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#define TASK\_STK\_SIZE 1024 // Stack size, in bytes

//#define SUSPEND\_RESUME // Task 1 and 2 use suspend and resume for scheduling

// (if not defined: use time delays)

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* VARIABLES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

OS\_STK TestTaskStk1[TASK\_STK\_SIZE]; // Task stacks

OS\_STK TestTaskStk2[TASK\_STK\_SIZE];

OS\_STK TestTaskStk3[TASK\_STK\_SIZE];

OS\_STK TestTaskStk4[TASK\_STK\_SIZE];

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FUNCTION PROTOTYPES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void TestTask1(void \*pdata); // The 3 tasks

void TestTask2(void \*pdata);

void TestTask3(void \*pdata);

void TestTask4(void \*pdata);

void OurIsr1(void); // The interrupt service routine for IRQ 1

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* MAIN

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

int main(void)

{

// Display a banner.

#ifdef \_\_WIN32\_\_

printf("##### uCOS-II V%4.2f Port V%4.2f for WIN32 #####\n", ((FP32)OSVersion())/100, ((FP32)OSPortVersion())/100);

#endif

#ifdef \_\_LINUX\_\_

printf("##### uCOS-II V%4.2f Port V%4.2f for LINUX #####\n", ((FP32)OSVersion())/100, ((FP32)OSPortVersion())/100);

#endif

// Initialize uCOS-II.

OSInit(); //Calling sequence -->OSInitHookBegin-->OSTaskStkInit-->OSTCBInitHook-->OSTaskCreateHook-->OSInitHookEnd

// Create the first task

OSTaskCreate(TestTask1, (void \*) 11, &TestTaskStk1[TASK\_STK\_SIZE], 11); //Calling sequence -->OSTaskStkInit-->OSTCBInitHook-->OSTaskCreateHook

PC\_IntVectSet(1, OurIsr1); //Install an interrupt service routine for IRQ 1

// Start multitasking.

OSStart(); //Calling sequence -->OSTaskSwHook-->OSStartHighRdy

/\* NEVER EXECUTED \*/

printf("main(): We should never execute this line\n");

return 0;

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* First Task (startup task)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void TestTask1(void \*pdata)

{ printf("%4u: \*\*\*\*\* Test Task 1 First call \*\*\*\*\*\n", OSTime);

#if OS\_TASK\_STAT\_EN > 0

OSStatInit(); //Initialize the statistics task

#endif

OSTaskCreate(TestTask2, (void \*) 22, &TestTaskStk2[TASK\_STK\_SIZE], 22); //Create 3 other tasks

OSTaskCreate(TestTask3, (void \*) 33, &TestTaskStk3[TASK\_STK\_SIZE], 33);

OSTaskCreate(TestTask4, (void \*) 10, &TestTaskStk3[TASK\_STK\_SIZE], 10);

while (1)

{ printf("%4u: \*\*\*\*\* Test Task 11 \*\*\*\*\*\n", OSTime);

#ifdef SUSPEND\_RESUME

OSTaskSuspend(OS\_PRIO\_SELF); //Calling sequence -->OSTaskSwHook-->OSCtxSw

#else

OSTimeDly(1); //Calling sequence -->OSTaskSwHook-->OSCtxSw

#endif

}

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Second Task

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void TestTask2(void \*pdata)

{ while (1)

{ printf("%4u: \*\*\*\*\* Test Task 22 \*\*\*\*\*\n", OSTime);

#ifdef SUSPEND\_RESUME

OSTaskSuspend(OS\_PRIO\_SELF);

#else

OSTimeDly(1);

#endif

}

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Third Task

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void TestTask3(void \*pdata)

{ while (1)

{

#ifdef SUSPEND\_RESUME

printf("%4u: \*\*\*\*\* Test Task 33 \*\*\*\*\*\n", OSTime);

OSTaskResume(11);

printf("%4u: \*\*\*\*\* Test Task 33 \*\*\*\*\*\n", OSTime);

OSTaskResume(22);

#endif

printf("%4u: \*\*\*\*\* Test Task 33 \*\*\*\*\*\n", OSTime);

OSTimeDly(1);

if (kbhit())

exit(0);

}

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Forth Task

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void TestTask4(void \*pdata)

{ while (1)

{

printf("%4u: +++++ Test Task 40 +++++\n", OSTime);

OSTaskSuspend(10); //Suspend yourself

}

}

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Interrupt service routine for IRQ 1

\* (use "irqGenerator 1" to trigger this interrupt)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void OurIsr1(void)

{ OSIntEnter();

printf("##### Interrupt service routine for IRQ 1 #####\n");

OSTaskResume(10); //Trigger task 4

OSIntExit();

}

Archivo test.c del ejemplo 0.

OSTime

Archivo /software/uCOS-II/source/ucos\_ii.h

#if OS\_TIME\_GET\_SET\_EN > 0

OS\_EXT volatile INT32U OSTime; /\* Current value of system time (in ticks) \*/

#endif

OS\_TIME\_GET\_SET\_EN

/software/uCOS-II/examples/example0/os\_cfg.h:#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet() \*/

/software/uCOS-II/examples/example1/os\_cfg.h:#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet() \*/

/software/uCOS-II/examples/example2/OS\_CFG.H:#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet() \*/

/software/uCOS-II/examples/example3/OS\_CFG.H:#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet() \*/

/software/uCOS-II/examples/example4/OS\_CFG.H:#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet() \*/

/software/uCOS-II/templates/OS\_CFG.h:#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet()

Para determinar de cuál de esos headers se está usando la definición, se edita el archivo /software/uCOS-II/examples/example0/os\_cfg.h para colocar después de la línea

#define OS\_TIME\_GET\_SET\_EN 1 /\* Include code for OSTimeGet() and OSTimeSet() \*/

la directiva

#error OS\_TIME\_GET\_SET\_EN is defined HERE !!!!

y reconstruimos (desde el directorio /software/uCOS-II/examples/example0/) con

make –f makefile.lin

Y efectivamente al obtener la salida de error

os\_cfg.h: 112:2 error: #error OS\_TIME\_GET\_SET\_EN is defined HERE !!!!

lo cual nos indica que en este proceso de construcción, la macro OS\_TIME\_SET\_SET\_EN se está definiendo en el archivo

/software/uCOS-II/examples/example0/os\_cfg.h

Función OSTaskCreate()

/software/uCOS-II/source/ucos\_ii.h

#if OS\_TASK\_CREATE\_EN > 0

INT8U OSTaskCreate(void (\*task)(void \*pd), void \*pdata, OS\_STK \*ptos, INT8U prio);

#endif

OS\_VERSION

/software/uCOS-II/source/ucos\_ii.h

#define OS\_VERSION 252

OS\_STK

/software/uCOS-II/src/os\_cpu.h

typedef INT32U OS\_STK

REFERENCIAS

[13720667.pdf] Altium TASKING POSIX Implementation.

[Altium, BV, 2015] TASKING POSIX Implementation.

[Burns, Welling, 2009] Alan Burns, Andy Welling, Real- Time Systems and Programming Languages, Pearson, 2009.

[Dietrich y Walker] Dietrich, S., y Walker, D., “The evolution of Real-Time Linux”, <http://www.cse.nd.edu/courses/cse60463/www/amatta2.pdf>, 2015.

[Labrosse, Jean J] Micro C/OS-II The Real Time Kernel (2/a edición). CRC Press, 2002.

[Liu, Layland] Liu, C.L., Layland, J.W., “Scheduling Algorithm for Multi-programming in a Hard Real-Time Environment,” J. ACM, Vol. 20, pp. 40-61, 1973.

[Wang] K.C. Wang, Design and Implementation of the MTX Operating System. Springer International Publishing, 2015.