UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

FACULTAD 5

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Pruebas de desempeño en los sistemas operativos de tiempo real para FreeRTOS sobre núcleo ARM.

Autor: Oniel R. Mato Ruiz.

Tutor: Ing. Antonio Cedeño Pozo.

Co-tutores: Ing. Roberto Alejandro Espi.

Ing. Alexander Moreno Limonte.

Habana junio, 2011

"Año 53 de la Revolución".

DECLARACIÓN DE AUTORÍA

Declaro ser autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

_____	_____
Firma del Autor	Firma del tutor
(Oniel Rafael Mato Ruiz)	(Antonio Cedeño Pozo)

DATOS DE CONTACTO

Ing. Antonio Cedeño Pozo.

Email: acedeno@uci.cu.

Graduado de Ingeniero Informático de la Universidad de las Ciencias Informáticas en el 2009. Cinco años de experiencia en el desarrollo de software.

AGRADECIMIENTOS

Agradezco a lo más grande que tengo a mi mamá y mi papá, por el amor incondicional que me brindan, por apoyarme en todos mis caprichos, por darme todo lo que tengo y gracias a ellos soy todo lo que soy.

A mi hermano por quererme mucho, porque cada vez que le pido o le quito algo no me protesta, por el amor que me tiene.

A mi abuela Gladys y mi Abuelo Betancourt a ellos agradezco por sacrificarse por mí, por darme todo lo que me dieron, por no retirarse hasta que me graduara para poder ayudar a mis padres en lo que me hiciera falta.

A mi abuela Estrella por rezar por mi cada vez que tenía pruebas, por querer y ayudarme en lo que podía.

Agradezco a mi tutor por ayudarme en todo lo que le pedí, y por sus consejos que me sirvieron de mucho.

A mis amigos en especial a Yasmany a Nestor a Yasniel a Osiel que llegó de último pero supo ganarse un lugar, a Emily por su preocupación, al Bu al Day a Niurka por ayudarme y siempre estar ahí para corregirme con las cosas que tenía mal.

DEDICATORIA

A mi mamá, mi papá, hermano, a mis Abuelos.

A mis amigos.

RESUMEN

En el presente trabajo se exponen las principales características y requerimientos de los sistemas operativos de tiempo real (RTOS), centrándose más en el sistema operativo de tiempo real FreeRTOS, el cuales es usado en el Centro de Informática Industrial (CEDIN), por la línea de sistemas empotrados.

Se analiza además las diferentes pruebas de desempeño que se le pueden realizar a un RTOS con el fin de saber su rendimiento, para su posterior uso en la línea. A partir de una serie de tareas desarrolladas se realiza una investigación de las pruebas hechas a otros RTOS para su posterior comparación, a los resultados alcanzados en las pruebas realizadas al FreeRTOS.

Se ofrece una visión sobre la valoración de los resultados obtenidos en las pruebas realizadas al FreeRTOS.

PALABRAS CLAVE

Sistemas empotrados, Pruebas de desempeño, requerimientos, rendimiento, FreeRTOS.

Tabla de contenido

INTRODUCCIÓN

Existen sistemas computacionales fuertemente relacionados con el entorno que les rodea y encontrándose en constante interacción con el mismo. Ejemplos de esto son los sistemas de adquisición de datos y los controladores industriales, los cuales están compuestos por diversos elementos (actuadores, sensores, unidades de cómputo, redes de interconexión) que deben trabajar de forma conjunta y coordinada. Debido a la naturaleza cambiante del entorno con el que interactúan, junto con la necesidad de coordinación entre sus componentes, los resultados obtenidos sólo podrán ser considerados válidos cuando, además de ser correctos desde el punto de vista lógico, hayan sido generados a tiempo. Resulta, por tanto, preciso imponer restricciones temporales, cuyo cumplimiento garantice el correcto funcionamiento del sistema. Este tipo de sistemas, capaces de realizar tareas y responder a eventos asíncronos externos dentro de unos plazos temporales determinados son los denominados "Sistemas de Tiempo Real". (Rivas, 2002) (Rubio, 2005)

Para que sea posible la predictibilidad temporal del sistema completo, todas las partes que le componen deberán de presentar un comportamiento predecible. En consecuencia, en el caso de utilizar un sistema operativo, los servicios que éste proporcione a las aplicaciones deberán presentar tiempos de respuesta acotados, de forma que así sea capaz de garantizar los requerimientos temporales de los procesos bajo su control. Mientras que en un sistema operativo de tiempo compartido, como Unix, lo importante es proporcionar a los usuarios unos buenos tiempos de respuesta promedios, la clave en los sistemas operativos de tiempo real será garantizar los requerimientos temporales; el tiempo de respuesta promedio pasa así a un segundo plano. (Rivas, 2002)

Como parte de la batalla de ideas y con el objetivo central de formar personal altamente calificado, se crea la Universidad de las Ciencias Informáticas (UCI). Dicho centro concibe como una de sus misiones fundamentales la vinculación de sus estudiantes, tanto en los fines docentes como en el trabajo productivo, como paso de un nuevo método de aprendizaje para el desarrollo en la educación superior. Con el fin de dar cumplimiento a este nuevo método, se crean los centros productivos, los cuales además deben ser

capaces de llevar a cabo el desarrollo de aplicaciones idóneas que respondan a las necesidades de la universidad, de nuestro país y otras naciones del mundo.

Específicamente en el centro productivo de Informática Industrial (CEDIN) en el cual existe una línea, además de diferentes proyectos que trabajan con sistemas operativos de tiempo real. Estos utilizan una tarjeta evaluadora llamada STM3210C la cual usa como sistema operativo el FreeRTOS y un núcleo ARM STM32. Los mismos no cuentan con herramientas que permitan determinar si el sistema operativo FreeRTOS y el núcleo ARM STM32 cumplen con los requerimientos para ser utilizados en sistemas de tiempo real A fin de buscar una vía para dar solución a esta problemática, se propone como **problema a resolver**: ¿Cómo medir el desempeño para tiempo real del sistema operativo FreeRTOS sobre un núcleo de arquitectura ARM?

Para un mejor entendimiento del tema, se define como **objeto de estudio**: Sistemas operativos de tiempo real; tomando como **campo de acción** para el desarrollo del trabajo: la aplicación de pruebas de desempeño al sistemas operativos de tiempo real FreeRTOS y el núcleo ARM STM32.

El **objetivo general** es: implementar un conjunto de pruebas sobre la tarjeta evaluadora STM3210C que permitan determinar si el sistema operativo FreeRTOS y el núcleo ARM STM32 cumplen con los requerimientos para ser utilizados en sistemas de tiempo real.

Evaluándose la situación existente planteada anteriormente, se precisa como **idea a defender** que: con la implementación de las pruebas de desempeño al FreeRTOS sobre núcleo de arquitectura ARM, se podrá determinar si los mismos cumplen con los requerimientos para ser utilizados en sistemas de tiempo real.

Hacia el cumplimento del objetivo específico, se toman como **tareas a desarrollar**:

- Estudio de FreeRTOS y otros sistemas operativos con soporte para tiempo real, así como todos los conceptos relacionados para la elaboración del marco teórico.
- Diseño de las pruebas para determinar si un sistema operativo cumple con requerimientos de tiempo real.

- Montaje del ambiente de trabajo para desarrollar sobre FreeRTOS en la tarjeta evaluadora STM3210C.
- Implementación de las pruebas utilizando el API de FreeRTOS.
- Evaluación de los resultados de las pruebas.

Métodos teóricos utilizados:

Analítico – Sintético: Aplicado al realizar un análisis sobre las pruebas de sistemas operativos de tiempo real, y como relacionar esas pruebas con el sistema operativo FreeRTOS sobre el núcleo ARM.

Histórico – Lógico: Aplicado para analizar las pruebas de sistemas operativos que existen en Cuba y en el mundo, así como para analizar las tecnologías, métodos y herramientas a utilizar en el desarrollo de dichas pruebas.

De este trabajo de diploma se espera:

- Obtener un documento con el diseño de las pruebas que se deber realizar sobre un sistema operativo para determinar si cumple con requerimientos de tiempo real.
- Se obtendrán además la implementación de dichas pruebas para el diagnóstico específico del sistema operativo FreeRTOS utilizado sobre arquitectura ARM y particularmente sobre microprocesadores STM32.

El contenido de este trabajo se distribuye en 3 capítulos. El primer capítulo contiene la fundamentación teórica del tema en cuestión. El capítulo 2 expone la descripción de las Pruebas de desempeño en los sistemas operativos de tiempo real para FreeRTOS sobre núcleo ARM.

El resto del trabajo está centrado en la implementación de las Pruebas de desempeño en los sistemas operativos de tiempo real y en la valoración de los resultados obtenidos de dichas pruebas.

CAPÍTULO I. FUNDAMENTACIÓN TEÓRICA

1.1 Introducción.

En este capítulo se abordan diversos aspectos relacionados con los Sistemas Operativos de Tiempo Real. Se dan a conocer que es un Sistema Operativo de Tiempo Real, sus características fundamentales, los requerimientos de los Sistemas Operativos de Tiempo Real. Se describe el Sistema Operativo de Tiempo Real FreeRTOS, realizando un estudio del arte de los diferentes Sistemas Operativos de Tiempo Real existentes y por último se caracterizan las herramientas a usar en la implementación de las pruebas de desempeño.

1.2 Sistemas Operativos de Tiempo Real

Cuando un sistema computacional requiere dar respuesta a determinados dispositivos en tiempo real, aspectos como la administración eficiente de memoria y el aumento de la eficiencia de utilización del CPU se hacen secundarios, tal es el caso de las plataformas robóticas y las mallas de control industrial. Esta necesidad ya ha sido detectada por empresas diseñadoras de micro controladores usados en aplicaciones de tiempo real y ha sido superada mediante el desarrollo de sistemas operativos para estos dispositivos, los mismos son conocidos como sistemas operativos de tiempo real (*RTOS siglas en inglés*). (Rubio, 2005)

Los Sistemas Operativos de Tiempo Real son diseñados para determinar la respuesta a sucesos o eventos, de forma tal, que se respeten ciertos rangos de tiempos de respuesta, los cuales, en el caso de ser críticos para el sistema, requieren de atención inmediata. En todo caso el RTOS es determinista, en consecuencia permite que el usuario prediga los tiempos de respuesta que el RTOS proporcionara el sistema informático que administra. Estos eventos, normalmente se producen en el entorno del sistema; llegan como interrupciones al procesador, ya sea por software o hardware. Los RTOS deben atender ráfagas de miles de interrupciones que activan diferentes procesos, Para no perder ningún suceso recibido, el RTOS debe trabajar con la técnica multiproceso, la cual ejecuta los procesos independientes unos de otros, pero en ciertas aplicaciones se deben tener estrategias para asignar prioridad a los procesos y ejecutarlos según su importancia, la cual se logra con una planificación basada en prioridades. (Rubio, 2005)

1.2.1 Características:

Las características que los RTOS deben ofrecer son:

- Soporte para la planificación de procesos en tiempo real.
- Planificación por prioridad.
- Garantía de respuesta ante interrupciones.
- Comunicación interprocesos.
- Adquisición de datos a alta velocidad.
- Soporte de E/S.
- Control, por parte del usuario, de los recursos del sistema.

Soporte para la planificación de procesos en tiempo real:

Un RTOS debe proporcionar soporte para la creación, borrado y planificación de múltiples procesos, los cuales monitorizan y controlan parte de una aplicación. Típicamente, en un RTOS, es posible definir prioridades para procesos e interrupciones. En contraste, en un sistema de tiempo real compartido, solo el propio sistema operativo determina el orden en que se ejecutan los procesos. (real, 2008)

Planificación por prioridad:

Un RTOS debe asegurar que un proceso de alta prioridad, cuando esté listo para ejecutarse, pase por delante de un proceso de más baja prioridad. El Sistema Operativo (SO) deberá ser capaz de reconocer la condición (usualmente a través de una interrupción), pasar por delante del proceso que se está ejecutando y realizar un rápido cambio de contexto para permitir la ejecución de un proceso de más alta prioridad. (real, 2008)

Garantía de respuesta ante interrupciones:

Un RTOS debe reconocer muy rápidamente la aparición de una interrupción o un evento, y tomar una acción determinista (bien definida en términos funcionales y temporales) para atender a ese evento. Debe responder tanto a interrupciones de tipo *hardware* como *software.* El propio SO debe ser interrumpible y reentrante. Las interrupciones son una fuente introductoria de indeterminismo, imponen la aparición de latencias. (real, 2008)

Comunicación interprocesos:

Un RTOS debe ser capaz de soportar comunicaciones interprocesos de manera fiable y precisa, tales como semáforos, paso de mensajes y memoria compartida. Estas facilidades se emplean para sincronizar y coordinar la ejecución de los procesos, así como para la protección de datos y la compartición de recursos. (real, 2008)

Soporte de E/S:

Las aplicaciones típicamente incluyen cierto número de interfaces de entrada y salida (E/S). Un RTOS debe proporcionar herramientas para incorporar fácilmente dispositivos de E/S específicos (incluso a medidas). Además de soportar E/S asíncrona, donde un proceso puede iniciar una operación de E/S, y luego continuar con su ejecución mientras concurrentemente se está realizando la operación de E/S. En este aspecto cabe recordar la existencia de procesadores específicos (canales de E/S, controladores de DMA, entre otros.) dedicados a realizar operaciones de E/S sin el concurso de la CPU. (real, 2008)

Control, por parte del usuario, de los recursos del sistema:

Una característica clave de los RTOS es la capacidad de proporcionar a los usuarios del sistema, incluyendo la propia CPU, memoria y recursos de E/S. El control de la CPU se logra sobre la base de una planificación por prioridades en la cual los usuarios pueden establecer las prioridades de los procesos. Además, se dispone de temporizadores en tiempo real y de funciones para manejarlos, planificar eventos y períodos de espera. Un RTOS debe también facilitar el bloqueo de la memoria (locking), de esta forma se puede garantizar que un programa, o parte de él, permanece en la memoria, a fin de poder

realizar cambios de contexto de manera más rápida cuando ocurre una interrupción. Debe ser capaz de permitir al usuario la asignación de buffer y la posibilidad de bloquear o desbloquear archivos y dispositivos. (real, 2008)

1.2.2 Requisitos de los Sistemas Operativos de Tiempo Real

Los Sistemas Operativos de Tiempo Real, deben cumplir ciertas exigencias para poder utilizar el control de sistemas de tiempo real. Estas exigencias van encaminadas a garantizar la correcta ejecución temporal de las tareas, tanto en lo referente al momento de la activación, como en la finalización dentro de los plazos permitidos. Las características deseables de los Sistemas Operativos de Tiempo Real son:

Garantizar la correcta ejecución de todas las tareas críticas.

El sistema operativo debe ser capaz de cumplir los plazos de ejecución de las tareas críticas, aún en el caso de sobrecarga del sistema. Por tanto, la cantidad de tareas críticas admisibles necesitan estar limitadas.

Se requiere algún mecanismo o criterio que permita saber si la aplicación está dentro de este límite o lo sobrepasa.

El ser capaz de predecir en tiempo de ejecución que un conjunto de tareas se puedan planificar es un problema que no está resuelto. Lo normal en este sentido es que el sistema operativo permita la utilización de políticas de planificación que están bien estudiadas teóricamente y en esta teoría se ofrecen métodos para asegurar la planificación de estas tareas. (ReqSO,2004)

Administrar adecuadamente el uso de los recursos compartidos.

En una aplicación de tiempo real, no es común que se realice un conjunto de tareas, todas ellas independientes. Lo normal es que interactúen unas con otras o que utilicen recursos comunes y, por tanto, que se tengan que sincronizar. Normalmente esta sincronización la realiza el sistema operativo.

En la programación de tiempo real no basta una simple sincronización, sino que se debe evitar ciertas situaciones que no son aceptables, como por ejemplo los interbloqueos y la inversión de prioridad.

La sincronización entre bloqueos afecta a la planificación de un sistema, sobre todo si en ella intervienen tareas críticas. Estos efectos se deben tener en cuenta dentro de la política de planificación que permite utilizar el sistema operativo. (ReqSO,2004)

Buen tiempo de respuesta a las tareas que no tengan plazo de terminación.

Las tareas que no son urgentes y, por tanto, que no tienen plazo de finalización que cumplir, siempre se podrán ejecutar en el tiempo sobrante como tareas de fondo. Ahora bien, en algunos casos interesa ejecutarlas de forma más eficiente.

La forma de mejorar la ejecución del trabajo no urgente, es retrasando la ejecución del trabajo urgente pero sin que se llegue a sobrepasar sus límites de ejecución y, por tanto, disponer de tiempo para adelantar la ejecución del trabajo no urgente. Esto supone un cambio en el mecanismo de planificación que debe tenerse en cuenta dentro del sistema operativo que es quien realiza la planificación. (ReqSO, 2004)

Soportar los cambios de modo.

El sistema debe facilitar el cambio ordenado del conjunto de tareas, permitiendo que durante un tiempo coexistan tareas de uno y otro modo, asegurando que no se viola ninguna restricción temporal. (ReqSO,2004)

Buen tiempo de respuesta a las interrupciones.

Las interrupciones juegan un papel muy importante en los sistemas de tiempo real.

Cuando ocurre un suceso externo un dispositivo hardware lo detecta y mueve una señal que le llega al procesador provocando una solicitud de interrupción justo en el instante de producirse el suceso. Si el sistema debe producir una respuesta acotada en el tiempo para esta interrupción, es importante el tiempo que tarda el sistema en reaccionar a la

interrupción, es decir, el tiempo que transcurre desde que se solicita la interrupción hasta que se activa su manejador.

Existen tres factores importantes que influyen en el tiempo de respuesta a las interrupciones:

1. **Niveles de interrupción.** La disponibilidad de distintos niveles de interrupción, es decir, la capacidad de que una interrupción de nivel superior interrumpa a un manejador de una interrupción de nivel inferior, permite organizar las interrupciones de forma que una interrupción importante no pueda ser bloqueada por otra de menos importancia, aunque esta segunda precise de un manejador de larga duración.

2. **Duración del tratamiento de las interrupciones**, es decir, el tiempo máximo que el sistema está tratando una interrupción. Durante el tratamiento de una interrupción, se inhiben el tratamiento de las interrupciones para evitar inconsistencias en el manejo del hardware asociado a la interrupción, al menos las de un determinado tipo o nivel. La duración máxima determinará la frecuencia más alta a la que pueden producirse las interrupciones.

3. **Tiempo de latencia de interrupción**. Es el tiempo que el sistema inhibe las interrupciones en zonas críticas que no puede ser interrumpidas. Los tiempos de latencia de interrupción provocarán retrasos en la activación de los manejadores produciendo, no solo el retraso de la respuesta a la interrupción sino que puede provocar que se pierdan interrupciones.

El primer factor viene dado por el hardware utilizado, aunque depende de la utilización que se dé a las posibilidades del hardware en el diseño del sistema. Los otros dos factores son exclusivamente software y dependen del diseño del sistema operativo. (ReqSO,2004)

Recuperación ante fallos software y hardware.

Es frecuente la utilización de los sistemas de tiempo real en aplicaciones de control. En estos sistemas no es aceptable una parada del sistema o un comportamiento incontrolado.

En la medida que el sistema operativo ofrezca posibilidades de controlar los fallos software y hardware facilitará la realización de aplicaciones fiables ante fallos.

Existe una dificultad adicional que consiste en el cumplimiento de las restricciones temporales aún en el caso de producirse un fallo. En este sentido, todo sistema de tiempo real estricto debe ser tolerante a fallos pues los límites temporales para las respuestas ante los eventos críticos no deben sobrepasarse aunque se produjera algún tipo de fallo. (ReqSO,2004)

Eficiencia en los cambios de contexto (procesos ligeros).

Los cambios de contexto se producen cada vez que se modifica la tarea en ejecución. Cuando corresponde ejecutarse una nueva tarea esta no comenzará inmediatamente, sino que su ejecución se verá retrasada por el tiempo que necesita el procesador para guardar el estado de la tarea que estaba en ejecución en ese momento y cargar el estado con el que debe continuar la nueva tarea.

Cuanto más tiempo le lleve al procesador el cambio de contexto, más se retrasará la ejecución de la tarea. El tiempo de cambio de contexto se puede considerar como tiempo de ejecución que tiene que añadir a la tarea que se activa pero, en cualquier caso, en los sistemas de tiempo real este debe ser lo más reducido posible, sobre todo en sistemas que por sus características temporales se realizan cambios de tarea muy frecuentemente.

Para facilitar el cambio de contexto se utilizan los procesos ligeros (llamados hilos). Muchos sistemas operativos para desarrollo de aplicaciones empotradas soportan en realidad un único proceso con múltiples hilos. Su funcionamiento es mucho más eficaz que los sistemas operativos que manejan procesos (pesados) y poseen las mismas prestaciones de concurrencia y sincronización. (ReqSO,2004)

1.2.3 Planificador.

El planificador es un conjunto de políticas y mecanismos incorporados al Sistema Operativo que gobierna el orden en que se ejecutan las tareas en un sistema multi programado. El objetivo primario de la planificación es optimizar el rendimiento del sistema de acuerdo con los criterios considerados más importantes por los diseñadores. Existen 3 tipos de planificadores los cuáles se clasifican según la frecuencia con la que es invocado por el procesador; estas son planificaciones a largo, mediano y corto plazo; este último es el único que se implementa en los RTOS.

El planificador a corto plazo, también conocido como distribuidor, es el que toma decisiones con una mayor frecuencia y detalle sobre la tarea que se ejecutará a continuación. Su principal objetivo es maximizar el rendimiento del sistema de acuerdo con un conjunto de criterios elegidos. El planificador a corto plazo debe ser ejecutado por lo tanto, cada vez que una tarea se detenga (bloqueo), o cada vez que un suceso (interno o externo) ocurra. Este es el tipo de planificador que soportan los RTOS, ya que el tiempo de ejecución es el criterio más importante a tener en cuenta. La suspensión, la terminación o aborto de una tarea en ejecución, son también sucesos que pueden necesitar la llamada al planificador a corto plazo. (Rubio, 2005)

1.2.3.1 Algoritmos de planificación.

Los algoritmos o mecanismos de planificación se pueden dividir en dos grandes grupos: expropiativa y no expropiativa. La no expropiación implica que cuando una tarea de mayor prioridad aparece, la tarea actual en ejecución no se ve obligada a ceder la propiedad del procesador sino hasta que este termine su ejecución. Con planificación expropiativa una tarea en ejecución puede ser sustituida por una tarea de mayor prioridad en cualquier instante, lo que implica entonces una mayor carga de trabajo para el planificador pero un menor tiempo de respuesta a los eventos críticos. Las características que deben cumplir los RTOS son apoyadas por el uso de planificación expropiativa.

El Round Robin (RR), es un planificador por turno rotatorio o también conocido como fracciones de tiempo, presenta una apropiación del reloj de modo que se genera periódicamente una interrupción indicando que la tarea que se encuentra en ejecución

tiene que ser colocada en la cola de listas y selecciona la siguiente, utilizando para esto el algoritmo First In First Out (FIFO). EL criterio de diseño es el tiempo de duración de los quantum que va a utilizar. Si el cuanto es muy pequeño, las tareas cortas tendrán que pasar varias veces por el sistema antes de ser finalizado, generando una sobre carga en la gestión de recursos. Si el cuanto es muy grande, RR se degenera en un simple FIFO donde las tareas cortas tienen que esperar mucho tiempo. (Rubio, 2005)

1.2.4 Estructuras de prioridad.

En un RTOS, el diseñador debe asignar prioridades a las tareas, otorgando mayor prioridad a las que tengan unas especificaciones de tiempo más severas y a las que resulten vitales para el correcto funcionamiento del sistema. A grandes rasgos, las tareas pueden ser divididas en tres niveles de prioridad.

- **Interrupciones:** En este nivel se encuentran las rutinas de servicio de interrupción para las tareas y dispositivos que precisan gran rapidez en la respuesta. Algunas de estas tareas son el reloj en tiempo real y el reloj que maneja el despertador de tareas.
- **Reloj:** Tareas que requieren un procesamiento repetitivo a intervalos de tiempo, tales como algoritmos de control que deben ejecutarse en cada período de muestreo.
- **Base:** Son tareas de baja prioridad y pueden soportar demoras en su ejecución por no tener unas fuertes especificaciones temporales o ser vitales para el sistema. (real, 2008)

1.2.5 Niveles de interrupción.

Una interrupción fuerza la replanificación del trabajo de la CPU y el sistema pierde el control sobre el tiempo de la misma. Por esta razón, es necesario que el código de la rutina de interrupción se ejecute rápidamente. Además es también preciso que cuente con un fragmento de código para el almacenamiento de información volátil, como el contenido

de los registros que va a utilizar, para que luego una rutina, que ocurra en un nivel de prioridad más bajo (nivel de reloj o base), trabaje con esa información salvada y restablezca la planificación. (real, 2008)

1.3 Diferentes Sistemas Operativos de Tiempo Real

A continuación se muestran algunos de los Sistemas Operativos de tiempo real que existen en el mundo. De los mismos se realizó un estudio del estado del arte, siendo más específicos en sus características principales.

1.3.1 MaRTE OS

Es un sistema operativo para aplicaciones empotradas cuya principal característica es que sigue el subconjunto mínimo de sistema de tiempo real definido en el estándar POSIX. Otras características importantes de MaRTE OS son:

- Pensado para aplicaciones principalmente embebidas.
- Presenta tiempos de respuesta acotados en todos sus servicios.
- Existe un sólo espacio de direcciones de memoria compartido por el núcleo y la aplicación.
- Permite ejecutar aplicaciones Ada y C.
- Portable a distintas arquitecturas.

1.3.2 VxWorks

VxWorks es un sistema operativo de tiempo real, basado en Unix, vendido y fabricado por Wind River Systems. Como la mayoría de los sistemas operativos de tiempo real, vxWorks incluye kernel multitarea con planificador preventivo (los procesos pueden tomar la CPU arbitrariamente), respuesta rápida a las interrupciones, comunicación entre procesos, sincronización y sistema de archivos.

VxWorks se usa generalmente en sistemas embebidos. Al contrario que en sistemas nativos como Unix, el desarrollo de vxWorks se realiza en un "host" que ejecuta Unix o Windows.

En la actualidad, vxWorks puede ejecutarse en prácticamente todas las CPU modernas del mercado de sistemas embebidos. Esto incluye la familia de CPU x86, MIPS, PowerPC, SH-4, ARM, StrongARM y xScale.

1.3.3 QNX

Es un sistema operativo de tiempo real basado en Unix que cumple con la norma POSIX. Es desarrollado principalmente para su uso en dispositivos empotrados. Desarrollado por QNX Software Systems empresa canadiense. Está disponible para las siguientes arquitecturas: x86, MIPS, PowerPC, ARM, StrongARM y xScale.

QNX está basado en una estructura de micro núcleo, que proporciona características de estabilidad avanzadas frente a fallos de dispositivos, aplicaciones, entre otros.

1.4 FreeRTOS.

Ha sido portado a 25 arquitecturas de hardware desde pequeños micro controladores de 8 byte a procesadores completos de 32 byte, incluyendo ARM7, ARM9, Cortex M3, MSP430, AVR y PIC. Su versatilidad en el port se debe a varios factores:

- El código base del FreeRTOS es pequeño. Está compuesto por un total de tres archivos para el núcleo y un archivo adicional para el port necesitado por el mismo kernel.
- FreeRTOS está en su mayoría escrito en C estándar. Solo unas pocas líneas de código ensamblador son necesarias para adaptarlo a una plataforma en particular.
- FreeRTOS posee una extensa documentación en el código fuente como así también en el sitio web oficial, con benchmark a nivel de aplicación. FreeRTOS es de código abierto. Esta licenciado bajo GPL modificado y puede ser usado en

aplicaciones comerciales bajo esta licencia. El código del FreeRTOS está disponible de manera gratuita en su sitio web, lo que hace al kernel fácil de estudiar y entender.

1.4.1 Características

- Posibilidad de realizar multitarea preventiva o cooperativa.
- Ejecución de tareas en orden de prioridad.
- Creación y destrucción dinámica de tareas.
- Comunicación entre tareas a través de colas coordinadas por el OS.
- Sincronización de tareas a través de semáforos.
- Protección de recursos compartidos utilizando exclusiones mutuas (Mutex).
- Gran cantidad de arquitecturas soportadas (25 hasta el momento).
- Escrito casi en su totalidad en C, con secciones de configuración en ensamblador.
- Libre disponibilidad del código fuente. (Xu, 2008)

1.4.2 Planificador

FreeRTOS dispone de Round Robin como algoritmo de planificación, el mismo es basado en prioridades. A cada tarea se le asigna una prioridad. Las que tienen la misma prioridad comparten el tiempo del CPU de manera Round Robin, es decir de acuerdo al quantum de tiempo que tienen asignado para su ejecución. El planificador del FreeRTOS se puede configurar como preventivo o de colaboración. El comportamiento de tiempo real del FreeRTOS requiere la programación preventiva.

Para sistemas más simples, se puede utilizar la programación de colaboración. El planificador tipo preventivo puede detener una tarea en ejecución durante su ejecución (anticiparse a la tarea) para dar los recursos del CPU a otra tarea que esté lista para ejecutarse. Esta función se utiliza para el intercambio del tiempo del CPU entre las tareas

con la misma prioridad. También se utiliza en caso de una interrupción que puede despertar una tarea que esté esperando una señal o de algunos datos. La tarea que esté esperando por un evento tiene que tener mayor prioridad que la tarea actual en ejecución para asignarle el tiempo de CPU directamente. (Xu, 2008)

1.4.3 Comunicación entre tareas

FreeRTOS proporciona varios métodos para la comunicación entre tareas, incluyendo mensaje en cola y semáforos binarios. En FreeRTOS el mecanismo de cola puede ser utilizado en la comunicación entre dos tareas y la comunicación entre tareas e interrupciones. Una cola es una estructura capaz de almacenar y recuperar datos. (Xu, 2008)

Los semáforos en FreeRTOS están implementados como un caso especial de las colas. (Xu, 2008) Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados secciones críticas) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados, permiten a más o menos procesos, utilizar el recurso de forma simultánea. Un tipo simple de semáforo es el binario, que puede tomar solamente los valores 0 y 1. Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Son esencialmente lo mismo que los mutex (mutual exclusión) o exclusión mutua. Cuando el recurso está disponible, un proceso accede y realiza un decremento al valor del semáforo con la operación P. El valor queda entonces en 0, lo que hace es que, si otro proceso hace un decremento a este valor, tenga que esperar. Cuando el proceso que decremento el semáforo realiza una operación V, algún proceso que estaba esperando puede despertar y seguir ejecutando. (Xu, 2008)

1.5 Métricas.

La CPU puede tener una eficiencia muy alta y escasas prestaciones para tiempo real. Por tanto se necesitan unas medidas específicas para los ordenadores dedicados a sistemas en tiempo real. Existen tres metodologías y sus métricas relacionadas para medir objetivamente el rendimiento de tiempo real de una CPU:

- Métrica Rhealstone.

- Tiempo de latencia en la activación de procesos. (ReqSO, 2004)

1.5.1 Métrica Rhealstone:

Consiste en obtener valores cuantitativos de seis medidas que influyen en el comportamiento de los sistemas de tiempo real.

Los factores a medir en un sistema operativo, según esta métrica son:

- **Tiempo de cambio de tarea:** tiempo medio que emplea el sistema en conmutar entre dos tareas independientes de la misma prioridad.
- **Tiempo de prioridad:** tiempo medio que emplea el sistema en conmutar de una tarea de menos prioridad a otra de más.
- **Tiempo de latencia de interrupción:** tiempo que transcurre desde que la CPU recibe una interrupción y activa la rutina que la sirve.
- **Tiempo de transición en un semáforo:** retraso introducido entre que una tarea libera un semáforo y otra tarea que estaba a la espera se activa.
- **Tiempo de ruptura:** Tiempo que necesita el sistema para suspender una tarea que utiliza un recurso usado por una tarea de menor prioridad y activar esta última.
- **Velocidad de flujo de datos en entrada/salida**: Es el flujo de información en Kbyte por segundo que una tarea puede enviar a otra utilizando primitivas del sistema operativo y sin utilizar mensajes situados en buffers de memoria compartida. (ReqSO, 2004)

Tiempo de latencia en la activación de procesos:

Esta métrica mide el intervalo de tiempo entre el instante en que, el sistema recibe una interrupción hasta que, se activa la tarea que va a responder a este estímulo. Los componentes más importantes de este tiempo son el tiempo de latencia de interrupciones y el tiempo de cambio de contexto, pero hay otros componentes de esta medida:

- **Tiempo de respuesta a la interrupción.**
  - Tiempo de retardo hardware en atender la interrupción.
  - Finalización de la instrucción actual.
  - Tiempo de latencia de la interrupción.
- **Rutina de tratamiento de la interrupción.**

- Pre-procesado.
- Rutina de servicio de interrupción.
- Post-procesado.

- **Decisión de cambio de contexto.**
- **Cambio de contexto**. (ReqSO, 2004)

1.6 Pruebas de desempeño.

Para poder determinar si un sistema operativo de tiempo real y un núcleo determinado, cumplen requerimiento de tiempo real, hace falta realizarle pruebas de desempeño. El aspecto principal a la hora de medir el desempeño de un sistema es establecer los objetivos que se pretenden alcanzar en tal medición. No existe métrica alguna aplicable a todas las posibles aplicaciones. La idea principal a la hora de confeccionar las pruebas será, lo que necesitamos que nuestro sistema realice.

Existen cuatro criterios a tener en cuenta al realizar las pruebas de desempeño:

- Rendimiento.
- Nivel de respuesta del sistema
- Determinismo
- Espacio en memoria.

En cada uno de esos aspectos se establecen determinados requisitos para el sistema.

Hoy en día existen sistemas operativos de tiempo real que han sido probados sobre otras arquitecturas, con el objetivo de ver si se cumplen dichos requerimientos, no así para el caso de FreeRTOS sobre núcleo de arquitectura ARM.

1.7 Herramientas a utilizar

Las Herramientas expuestas en este trabajo son las utilizadas en la línea de Sistemas empotrados por lo que en esta investigación solo se exponen sus principales características.

1.7.1 Eclipse como entorno de desarrollo

La plataforma Eclipse consiste en un Entorno de Desarrollo Integrado (IDE, Integrated Development Environment) abierto y extensible. Cuenta con numerosas herramientas de desarrollo de software. También da soporte a otros lenguajes de programación, como son C/C++, Cobol, Fortran, PHP y Python. A la plataforma base de Eclipse se le pueden añadir extensiones (plugins) para extender sus funcionalidades. El término Eclipse además identifica a la comunidad de software libre para el desarrollo de la plataforma Eclipse.

Características generales

- Muy completo
- Configurable mediante plugins
- Pensado para Java pero adaptable a otros lenguajes
- Versiones preparadas para C/C++
- Plugins para Subversion y Doxygen.(Melgarejo, 2009)

1.7.2 C como lenguaje de programación

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL.

Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos. (Ritchie, 1993)

1.8 Conclusiones

En este capítulo se realizo un análisis de los sistemas operativos de tiempo real, definiendo que es un RTOS, las características que lo hacen de tiempo real, como funcionan estos sistemas operativos, las partes que lo componen , así como los requerimientos que deben cumplir dichos sistemas operativos, también se hizo un estudio del arte de los diferentes sistemas operativos de tiempo real existentes en el mundo, y se dieron las características de las herramientas a usar para la implementación de la aplicación.

CAPITULO 2. PROPUESTA DE SOLUCIÓN.

2.1 Introducción

En este capítulo se describen las pruebas de desempeño de los sistemas operativos de tiempo real, cuáles son los criterios a medir en las pruebas de desempeño, los factores fundamentales de los RTOS, se explica además aspectos como el rendimiento del procesamiento, el nivel de respuesta del sistema y el determinismo entre otros.

2.2 Existen diversos criterios a medir en las pruebas de desempeño:

Uno clásico pudiera ser el rendimiento (throughput).

- ¿Cuán rápido se puede realizar la operación X?
- ¿Cuántas veces se puede realizar X en un segundo?

Otro de los criterios importantes es el nivel de respuesta del sistema.

- ¿Cuán rápido el sistema responde a una entrada?

Un criterio importante es el determinismo del sistema.

- ¿Varía el desempeño de un sistema ante variaciones de la carga?

Por último los recursos consumidos por el sistema pudieran ser importantes, como por ejemplo el espacio en memoria.

El primer criterio suele ser la prueba de desempeño más importante en los sistemas que no son de tiempo real. Si bien este criterio es también importante en los RTOS, basándose en la propia definición de estos sistemas, se puede concluir que el segundo criterio suele tener mayor peso. El tercero de los criterios suele ser uno inviolable cuando se habla de los sistemas de tiempo real críticos. Y el último tiene un mayor peso en los sistemas empotrados. (Vázquez, 2007)

2.3 Rendimiento del procesamiento (Throughput)

El rendimiento del procesamiento suele ser medido en número de operaciones por segundo: MIPS (Millions of Instructions Per Second), bytes por segundos y número de señales que se pueden enviar por segundo. Alternativamente se puede hablar en

términos del tiempo que toma ejecutar una operación: 10 micro segundos para una llamada a una función x (). (Vázquez, 2007)

2.4 Nivel de respuesta del sistema (Responsiveness)

Cuando se habla del nivel de respuesta del sistema, se refiera a la velocidad con la cual el sistema en cuestión responde a un evento determinado. Un ejemplo, en los sistemas de tiempo compartido, es el nivel de interactividad, respuesta del sistema al accionar de las teclas.

En los sistemas de tiempo real generalmente se habla de nivel de respuesta ante interrupciones, ya que todos los eventos asíncronos del sistema (E/S del disco, tención a dispositivos, entre otros) se implementan utilizando interrupciones.

Las mediciones más comunes incluyen:

- Procesamiento de mensajes.
- Procesamiento de semáforo.
- Latencia en la planificación.
- Tiempo de cambio de contexto: (Vázquez, 2007)

2.5 Determinismo

El determinismo, en contraposición con la incertidumbre, indica el optimismo a la hora de que el sistema responda con un nivel de respuesta deseado. En un sistema de tiempo real lo normal es encontrar varios procesos que compiten por determinados recursos, y es en este esquema, que se tiene que ser capaz de determinar una respuesta al peor de los casos, para cada proceso en análisis sin importar lo que esté sucediendo en el sistema en su totalidad. Es decir, una aplicación en concreto debe responder, a condiciones variables, en un tiempo acotado sin importar cuántos procesos se estén ejecutando ni qué código estén ejecutando dichos procesos. (Vázquez, 2007)

2.6 Espacio en memoria.

Los criterios analizados hasta el momento se refieren al desempeño con respecto a respuestas temporales del sistema. Otro de los criterios importantes es cuánto espacio en memoria se necesita.

En este aspecto las opciones muchas veces se reducen a estimar cuán fácil se adapta el sistema en estudio a las necesidades de la implementación. Por ejemplo realizar las siguientes acciones:

- Si las estructuras de datos son más grandes que lo necesario, estas pueden ser modificadas en orden de reducir su tamaño.

- Si existen secciones de código que no se utilizarán, estas pueden ser eliminadas, a través de algún menú de configuración.

Algunos sistemas son altamente configurables, permiten eliminar casi todas las funcionalidades que se puedan considerar innecesarias. En ese caso están las arquitecturas de micro núcleo como RTLinux, donde los servicios están organizados en módulos separados los cuales se cargan si son necesarios. Unas de las pruebas que se realiza es la de configurar el sistema más pequeño y el más grande que se pueda y de esta forma tener una idea de la flexibilidad del sistema. (Vázquez, 2007)

2.7 Factores fundamentales para los sistemas de tiempo real crítico

En los sistemas de tiempo real críticos un proceso al cual se le retrasa su ejecución pudiera no ejecutarse nunca, los criterios de determinismo y nivel de respuesta del sistema son fundamentales. Esto no significa que los demás criterios no sean importantes, si lo son, pero los dos primeros hablan de situaciones que puede poner en peligro real el funcionamiento del sistema. Los demás factores responden si con una plataforma dada, se es capaz o no de implementar el sistema. Por ejemplo si el procesador escogido es capaz de procesar n tareas, o la memora física existente es capaz de contener nuestra mayor aplicación sin hacer intercambios con la memoria virtual. (Vázquez, 2007)

2.8 Métricas para el análisis de sistemas de tiempo real.

La mayoría de las métricas existentes hablan sobre el rendimiento del sistema, a pesar de ello existen referencias a importantes métricas para los sistemas de tiempo real, como la latencia en la interrupción y la latencia en la planificación. No obstante se debe aclarar que el nivel de respuesta y el determinismo de un sistema en estudio no pueden ser

medidos en una forma estándar. Para ello se necesita realizar un análisis que tenga en cuenta el hardware sobre el cual se ejecuta el sistema, entre otras cosas.

Métricas importantes para los sistemas de tiempo real:

- Planificación
- Objetos de sincronismo
- Nivel de respuesta del sistema y determinismo (Vázquez, 2007)

2.9 Pruebas de desempeño y objetivos de las mismas.

2.9.1 Pruebas del servicio de planificación

2.9.1.1 Pruebas del parámetro latencia de la planificación

El objetivo de esta prueba es la de determinar el tiempo en el cual un hilo, que está listo para ejecutarse, toma el procesador.

2.9.1.2 Pruebas del parámetro prioridades

El objetivo de esta prueba es la de comprobar que se cumplen las prioridades de tiempo real en el sistema.

2.9.1.3 Pruebas del parámetro cambio de contexto

El objetivo de esta prueba es medir el tiempo (del peor de los casos) al realizar un cambio de contexto entre dos hilos.

2.9.2 Pruebas del servicio de multitarea

2.9.2.1 Pruebas del parámetro implementación de hilos

El objetivo de esta prueba es constatar la estabilidad del sistema al construir N hilos.

2.9.2.2 Pruebas de los objetos de sincronismos (mutex y semáforos)

Esta es una prueba general que demuestra el funcionamiento de dichos objetos en el sistema.

2.10 Diseño de pruebas de los servicios de tiempo real.

2.10.1 Prueba del servicio de planificación

2.10.1.1 Pruebas del parámetro latencia de la planificación

Esta es una de las métricas fundamentales a tener en cuenta para el análisis de los servicios de tiempo real. Con ella se responde una de las preguntas principales de la planificación, cuánto es la demora que existe desde el momento que una tarea debe estar ejecutándose hasta el momento en que realmente se empieza a ejecutar. Para realizar esta medición existen varios escenarios:

Escenario 1: Se implementa un hilo periódico con la más alta prioridad del sistema, al implementarlo de esta forma se evita que se produzcan interferencias por parte de otros hilos del sistema en el experimento que se lleva a cabo. En el cuerpo del hilo en cuestión se invoca una función de espera, (xTaskDelay (), xTaskDelayUntil ()) de manera que el hilo realiza una espera por un tiempo determinado, entonces se determina de cuanto fue realmente esta espera la diferencia será la latencia buscada.

2.10.1.2 Pruebas el parámetro cambio de contexto

Cuando se habla de cambio de contexto se habla del tiempo que le toma al sistema operativo para cambiar de un proceso a otro. En algunas documentaciones se define el cambio de contexto por el tiempo sucedido entre la última instrucción de un proceso de usuario a la primera instrucción del próximo proceso de usuario. Esta medición de alguna manera también ofrece una idea de la latencia de la planificación del sistema.

- Llamada a xtaskYIELD ().

Esta llamada se realiza cuando una aplicación está en ejecución y es relevada del procesador de forma voluntaria, entrando en ejecución otro hilo de igual prioridad. El uso de esta llamada puede dar una imagen que toma un cambio de contexto. Se evidencia que este cambio de contexto, voluntario, es más rápido porque no hay tratamiento a interrupciones.

Escenario:

Se implementa un hilo con la máxima prioridad del sistema.

Se realiza la llamada xtaskYIELD () asegurando que no exista otro proceso con la misma prioridad. Como no habrá cambio de contexto la medición será sobrecarga pura.

- Llamada a xtaskYIELD (), caso con dos hilos.

Es una continuación de la prueba anterior

Escenario:

Se ejecutan dos hilos (procesos), con la máxima prioridad del sistema.

Se determina cuanto tiempo demora cada llamada a xtaskYIELD () cuando hay cambio de contexto.

Si se resta a este tiempo el encontrado en la prueba anterior y se obtiene el tiempo del cambio de contexto puro.

2.10.1.3 Pruebas del parámetro Prioridades

Para realizar la medición de esta métrica se tienen varios escenarios:

Escenario 1:

Un hilo de menor prioridad es relevado del procesador por otro de mayor prioridad.

Escenario 2:

Un hilo con la máxima prioridad del sistema ejecuta un cálculo intensivo, el sistema casi en su totalidad debe detener su funcionamiento hasta que el hilo termine su ejecución.

2.10.2 Prueba del servicio de Multitarea

A continuación se presentan los casos de prueba de las métricas correspondientes al servicio de multitarea.

2.10.2.1 Pruebas del parámetro implementación de hilos

La medición de este parámetro tiene los siguientes escenarios:

Escenario:

Se crean N hilos (un número relativamente grande) definidos por el usuario y se comprueba que la función xTaskCreate devuelve un ID válido.

2.10.2.2 Pruebas de los objetos de sincronismos (mutex y semáforos)

Semáforos:

Se prueba el tiempo entre una acción post sobre un semáforo y la ejecución de un hilo que está esperando en este semáforo.

Para esto se crea un hilo el cual hace la acción sem_wait sobre el semáforo, luego el segundo hilo se detiene a esperar a que se le permita continuar con su ejecución.

2.11 Conclusiones

En este capítulo se dio a conocer el objetivo de cada propuesta de prueba de desempeño, se realizó una descripción detallada de las mismas, dando a conocer cuáles son los criterios a medir en cada una y cuáles son los factores fundamentales de los sistemas operativos de tiempo real que se tuvieron en cuenta para el diseño de dichas pruebas.

CAPITULO 3. IMPLEMENTACIÓN Y RESULTADOS DE LAS PRUEBAS.

3.1 Introducción

En este capítulo se describe la implementación de las pruebas ilustrando su código y dando una breve explicación del mismo, de igual forma se dan a conocer los resultados obtenidos en las pruebas y una comparación de estos con pruebas que se han realizado a otros RTOS.

3.2 Especificaciones de hardware donde se ejecutaron las pruebas de Linux 2.6.18-rt7.

- Recursos físicos:

  PC; Intel Pentium 4 CPU: 3389. 401Mhz cache 2048 KB; 1GB RAM.

- Recursos lógicos:

  Núcleo 2.6.18. (Vazquez, 2007)

3.3 Especificaciones de hardware de ThreadX®RTOS.

- ARM Versatile Board, ARM926 Processor, 200MHz
- 10ms Timer
- No Caching
- ARM RealView Compiler NO-OPT
- Express Logic's ThreadX®RTOS.(Express Logic, 2008)

3.4 Prueba del servicio de planificación:

Escenario de la prueba:

La prueba se realizo en un período de tiempo de 30 segundos.

Para la implementación de esta prueba se crearon los siguientes métodos y tareas.

void carga (void* p). Esta tarea es creada para sobrecargar el sistema, con el fin de obtener una latencia, haciendo que dentro de esta se ejecuten ciclos con un gran número

de iteraciones, la misma cuenta con la prioridad más alta del sistema, se ejecutan 5 tareas de este tipo.

void latenciaPlanificador (void* p). Es en esta tarea, en la que se calcula la latencia del planificador y se muestra el resultado por el LCD (Liquid Crystal Display).

void pruebaLatenciaPlanificador(). Este método es el encargado de crear las tareas **void carga (void* p)**, **void latenciaPlanificador (void* p)**, y además de eliminar dichas tareas una vez que se tiene el resultado de la prueba, con el fin de no sobrecargar el sistema.

```
void pruebaLatenciaPlanificador(){

        xTaskHandle handle[6];

        xTaskCreate(carga, ( signed portCHAR * ) "carga", configMINIMAL_STACK_SIZE,
NULL, tskIDLE_PRIORITY+5, &handle[0]);

        xTaskCreate(carga, ( signed portCHAR * ) "carga1",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+5, &handle[1]);

        xTaskCreate(carga, ( signed portCHAR * ) "carga2",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+5, &handle[2]);

        xTaskCreate(carga, ( signed portCHAR * ) "carga3",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+5, &handle[3]);

        xTaskCreate(carga, ( signed portCHAR * ) "carga4",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+5, &handle[4]);

        xTaskCreate(latenciaPlanificador, ( signed portCHAR * )"latencia_planificador",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY, &handle[5]);

        portTickType time = xTaskGetTickCount();

        while(1){
```

```
                vTaskDelay(100);
            if( (xTaskGetTickCount() - time) > 140000 ){
                vTaskDelete(handle[0]);
                    vTaskDelete(handle[1]);
                    vTaskDelete(handle[2]);
                    vTaskDelete(handle[3]);
                    vTaskDelete(handle[4]);
                    vTaskDelete(handle[5]);
                    LCD_DisplayStringLine(205, "OK");
                    break;
                }
        }
}
void carga(void* p){
        unsigned long i,j = 0;
        while(1)        {
                for(j = 0; j < 900000000; j++){
                        vTaskDelay(1);
                        for(i = 0; i < 2000000;i++);
                        vTaskDelay(1);
```

```
                for(i = 0; i < 2000000;i++);
            }
            vTaskSuspend(NULL);
        }
}
void latenciaPlanificador(void* p){
        unsigned long i;
        portTickType tiempo = xTaskGetTickCount();
        portTickType aux = xTaskGetTickCount();
        Display(65, tiempo);
        for(;;){
            for (i = 0; i < 10000; ++i){
                vTaskDelayUntil(&aux, 10);
            }
            aux = xTaskGetTickCount();
            Display(85, aux);
            LCD_DisplayStringLine(125,"Latencia (NS) :");
            Display(145, ((aux - tiempo)-100000) * 100000);
            vTaskSuspend(NULL);
    }
```

}

Comparación del resultado obtenido

El resultado que se obtuvo en esta prueba fue de 300 000 nanosegundos, este resultado fue comparado por la prueba hecha a Linux con parche para tiempo real el cual fue de 11 453 nanosegundos en el mejor de los casos, hay que tener en cuenta las especificaciones de hardware donde se realizaron ambas pruebas para poder comparar estos resultados, valorando estas especificaciones, se puede determinar que la latencia obtenida en el planificador de FreeRTOS está de acorde con los planificadores de sistemas operativos de tiempo real ya probados.

3.5 Prueba del parámetro Prioridad:

Escenario de la prueba:

Para la implementación de esta prueba se crearon los siguientes métodos y tareas.

void hiloIntensivo(void* p). Hilo con la mayor prioridad y es el primero que se ejecuta cediéndole el control del kernel al hilo de menor prioridad voluntariamente.

void hiloDebil(void* p). Hilo de menor prioridad.

void pruebaParametroPrioridad(). Este método es el que crea los hilos y espera un tiempo de 1 segundo después de haberse ejecutado cada hilo y los eliminar con el fin de no sobrecargar el sistema.

```
void pruebaParametroPrioridad(){

        xTaskHandle handle [2];

        xTaskCreate( hiloIntensivo, ( signed portCHAR * ) "hiloIntensivo",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 5, &handle[0] );

        xTaskCreate( hiloDebil, ( signed portCHAR * ) "hiloDebil",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY, &handle[1] );
```

```
        portTickType time = xTaskGetTickCount();
        while(1){
                if( (xTaskGetTickCount() - time) > 100000 ){
                        if(handle[0])
                                vTaskDelete(handle[0]);
                        if(handle[1])
                                vTaskDelete(handle[1]);
                        break;
                }
}
void hiloIntensivo(void* p){
        portTickType time = xTaskGetTickCount();
   while(1){
        if( (xTaskGetTickCount() - time) > 50000 ){
                vTaskDelay(10000);
        }
        LCD_ClearLine(65);
        LCD_DisplayStringLine(65, "Hilo mayor prioridad");
   }
}
```

```
void hiloDebil(void* p){
        portTickType time = xTaskGetTickCount();
        while(1)        {
        if( (xTaskGetTickCount() - time) > 50000 ){
                vTaskDelay(10000);
        }
        LCD_ClearLine(85);
        LCD_DisplayStringLine(85, "Hilo menor prioridad");
        }
}
```

Valoración del resultado.

El resultado de esta prueba fue satisfactorio, se pudo comprobar que los parámetros de prioridad se cumplen en los sistemas operativos, esto se muestra en el LCD, cuando el hilo de menos prioridad no se ejecuta o toma el control del kernel hasta que el hilo de mayor prioridad no le ceda voluntariamente el control del mismo.

3.6 Pruebas del parámetro implementación de hilos:

Escenario de la prueba:

Para la implementación de esta prueba se crearon los siguientes métodos y tareas.

Para esta prueba se implementó un método que es el encargado de crear las tareas y comprobar que los Id devueltos por xTaskCreate sean válidos, mostrando un mensaje por el LCD informando que todas las tareas fueron creadas satisfactoriamente.

```
void pruebaParametroImplementacionHilos(){
        xTaskHandle xHandle[8];
```

```
int cantidad_Tareas = 1;
int cantidad_Tareas_Creadas = 0;
char a[10];
while(cantidad_Tareas <= 8){
        memset(a, 0, 10);
        iToA(cantidad_Tareas, a);
        if (xTaskCreate( vTarea1, ( signed portCHAR * )a ,
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle[cantidad_Tareas_Creadas]) == pdPASS){
                if(xHandle)
                        cantidad_Tareas_Creadas++;
        }
        cantidad_Tareas++;
        Display(5 + 20 * cantidad_Tareas, cantidad_Tareas_Creadas);
}
if(cantidad_Tareas_Creadas == 8){
        LCD_DisplayStringLine(205, "Prueba OK 8 task.");
}
else{
        LCD_DisplayStringLine(205, "Prueba fallida.");
}
```

```
        int i;
        for (i = 0; i < cantidad_Tareas_Creadas; ++i){
                vTaskDelete(xHandle[i]);
        }
}
```

Valoración del resultado:

El resultado de esta prueba pudo constatar que el sistema es estable ante la creación de un número significativo de tareas ya que se crearon 8 tareas y todos los Id devueltos por la función xTaskCreate fueron válidos.

3.7 Prueba del parámetro cambio de contexto:

Escenario de la prueba:

La prueba se realizo en un período de tiempo de 30 segundos.

Esta prueba consta de 5 hilos cada uno con una prioridad única. Todos los hilos, excepto el de prioridad más baja se encuentran en estado suspendido, este hilo reanuda el hilo que le sigue y que a su vez, es de prioridad más alta, esto sucede con cada hilo que es reanudado, incrementando un mismo contador cuando es reanudado, esto sucede en un período de tiempo, con el objetivo de ver la cantidad de iteraciones que se hicieron en dicho período.

Para la implementación de esta prueba se crearon los siguientes métodos y tareas.

void taskPreemptive0(void *p). Hilo que se suspende y reanuda el próximo hilo de mayor prioridad y que a su vez cuando es reanudado incrementa un contador.

void pruebaPreemptiveCambioContexto(). Método que crea los 5 hilos y hace que se ejecute la prueba durante un período de 30 segundos, cuando expira este período elimina las tareas creadas, evitando sobrecarga del sistema.

```
xTaskHandle handlePreemptive[5];

int countPreemptive = 0;

void pruebaPreemptiveCambioContexto(){

        xTaskCreate(taskPreemptive0, ( signed portCHAR * )"Preemp1",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 0, &handlePreemptive[0]);

        xTaskCreate(taskPreemptive1, ( signed portCHAR * )"Preemp2",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, &handlePreemptive[1]);

        xTaskCreate(taskPreemptive2, ( signed portCHAR * )"Preemp3",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 2, &handlePreemptive[2]);

        xTaskCreate(taskPreemptive3, ( signed portCHAR * )"Preemp4",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 3, &handlePreemptive[3]);

        xTaskCreate(taskPreemptive4, ( signed portCHAR * )"Preemp5",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 4, &handlePreemptive[4]);

        vTaskResume(handlePreemptive[0]);

        portTickType time = xTaskGetTickCount();

        while(1){

                vTaskDelay(100);

                Display(185, countPreemptive);


        if( (xTaskGetTickCount() - time) > 300000 ){

                vTaskDelete(handlePreemptive[0]);

                vTaskDelete(handlePreemptive[1]);
```

```
                    vTaskDelete(handlePreemptive[2]);
                    vTaskDelete(handlePreemptive[3]);
                    vTaskDelete(handlePreemptive[4]);
                    break;
                }
            }
}
void taskPreemptive0(void *p){
            vTaskSuspend(NULL);
            unsigned long count = 0;
            for(;;){
                    countPreemptive++;
                    vTaskResume(handlePreemptive[1]);
                }
}
void taskPreemptive1(void *p){
            vTaskSuspend(NULL);
            unsigned long count = 0;
            for(;;){
                    count++;
```

```
                vTaskResume(handlePreemptive[2]);
                vTaskSuspend(NULL);
            }
}
void taskPreemptive2(void *p){
        vTaskSuspend(NULL);
        unsigned long count = 0;
        for(;;){
                count++;
                vTaskResume(handlePreemptive[3]);
                vTaskSuspend(NULL);
            }
}
void taskPreemptive3(void *p){
        vTaskSuspend(NULL);
        unsigned long count = 0;
        for(;;){
                count++;
                vTaskResume(handlePreemptive[4]);
                vTaskSuspend(NULL);
```

```
                }
}
void taskPreemptive4(void *p){
                vTaskSuspend(NULL);
                unsigned long count = 0;
                for(;;){
                                count++;
                                vTaskSuspend(NULL);
                }
}
```

Valoración del resultado

El resultado de esta prueba fue de 928 420 iteraciones, teniendo en cuenta las especificaciones de hardware en donde se realizaron las pruebas con la cual se va a comparar, se puede determinar que este se acerca al resultado obtenido por Express Logic's, Inc. Al ThreadX®RTOS.

3.8 Procesamiento de mensaje:

Escenario de la prueba:

La prueba se realizo en un período de tiempo de 30 segundos.

Esta prueba consiste en un hilo que envía un mensaje de 16 bytes a una cola y este recibe el mismo mensaje de 16 bytes de la cola, después que ocurre la secuencia envío/ recibo, el hilo incrementa un contador, esta prueba se ejecuta en un período de tiempo, pasado el mismo el resultado será la cantidad de iteraciones que se hicieron en el período de tiempo.

Para la implementación de esta prueba se crearon los siguientes métodos y tareas.

void pruebaMensajeProcesamiento(). Este método es el que crea el hilo y se asegura que se ejecuta en un período de 30 segundos pasado este, muestra el mensaje por el LCD y elimina el hilo con el fin de dejar libre al sistema de carga.

void mensajeProcesamiento(void *p). Hilo que envía y recibe el mensaje e incrementa el contador.

```
xQueueHandle cola;

xTaskHandle handleMensaje;

unsigned long countMensaje = 0;

void pruebaMensajeProcesamiento(){

        cola = xQueueCreate( 32, 16);

        if(xTaskCreate(mensajeProcesamiento, ( signed portCHAR * )"semaphore",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY, &handleMensaje))

                LCD_DisplayStringLine(105, "OK");

        portTickType time = xTaskGetTickCount();

        while(1)        {

                vTaskDelay(100);

                displayEntero(185, countMensaje);

                if( (xTaskGetTickCount() - time) > 300000 ){

                        vTaskDelete(handleMensaje);

                        LCD_DisplayStringLine(205, "Prueba OK");

                        break;
```

```
            }
        }
}
void mensajeProcesamiento(void *p){
        char b[16];
        for(;;){
            xQueueSend(cola, "1234567890123456", portMAX_DELAY);
            xQueueReceive(cola, b, portMAX_DELAY);
            countMensaje++;
        }
}
```

Valoración del resultado:

El resultado obtenido en esta prueba es de 2 104 794 iteraciones, teniendo en cuenta las especificaciones de hardware del ThreadX®RTOS, y haciendo una comparación de las pruebas se puede concluir que el resultado obtenido es mejor al resultado arrojado por ThreadX®RTOS.

3.9 Procesamiento de semáforos:

Escenario de la prueba:

La prueba se realizo en un período de tiempo de 30 segundos.

Un semáforo es un recurso del sistema utilizado para garantizar a una tarea acceso exclusivo a una sección crítica y sincronizar las actividades asincrónicas. Esta prueba consiste en que el hilo tome un semáforo y lo ceda inmediatamente después,

esta secuencia la hace para un período de tiempo, después que se completa la secuencia de obtener el semáforo/cederlo, el hilo incrementa un contador

Para la implementación de esta prueba se crearon los siguientes métodos y tareas.

void pruebaSemaphoreProcesamiento(). Este método es el que crea el hilo y se asegura que se ejecuta en un período de 30 segundos pasado este tiempo muestra el mensaje por el LCD y elimina el hilo con el fin de dejar libre al sistema de carga.

void mensajeProcesamiento(void *p). Hilo que toma el semáforo y cede el mismo e incrementa el contador.

```
xTaskHandle handleSemaphore;

unsigned long countSemaphore = 0;

static xSemaphoreHandle xSemaphore;

void pruebaSemaphoreProcesamiento(){

	xSemaphore = xSemaphoreCreateMutex();

	if(xTaskCreate(semaphoreProcesamiento, ( signed portCHAR * )"semaphore",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY, &handleSemaphore))

		LCD_DisplayStringLine(105, "OK");

	portTickType time = xTaskGetTickCount();

	while(1)	{

		vTaskDelay(100);

		displayEntero(185, countSemaphore);


		if( (xTaskGetTickCount() - time) > 300000 ){
```

```
                vTaskDelete(handleSemaphore);
                LCD_DisplayStringLine(205, "Prueba OK");
                break;
            }
        }
}
void semaphoreProcesamiento(void *p){
        for(;;){
            if( xSemaphoreTake (xSemaphore , portMAX_DELAY) == pdTRUE ){
                xSemaphoreGive ( xSemaphore );
            }
            countSemaphore++;
        }
}
```

Valoración del resultado:

El resultado obtenido en esta prueba es de 2 104 794 iteraciones, teniendo en cuenta las especificaciones de hardware del ThreadX®RTOS, y haciendo una comparación de las pruebas se puede concluir que el resultado obtenido es mejor al resultado arrojado por ThreadX®RTOS.

3.10 Conclusiones

En este capítulo se detalló cada prueba de desempeño , realizando una comparación de los resultados obtenidos en dichas pruebas con los resultados alcanzados con las mismas a otros sistemas operativos de tiempo real, demostrando que el FreeRTOS tiene

resultados semejantes a los resultados lanzados por otros RTOS en núcleos de arquitectura ARM, Demostrando que cumple con requerimientos para ser utilizado en sistemas de tiempo real, los experimentos hechos al FreeRTOS fueron efectuados bajo las mismas condiciones que las pruebas realizadas al RTOS Linux con el parche de tiempo real y al ThreadX®RTOS.

CONCLUSIONES

Se realizó un estudio del sistema operativo de tiempo real FreeRTOS, así como algunos sistemas con soporte para tiempo real y los conceptos relacionados quedando conformado el marco teórico de esta investigación

Se hizo un estudio de que pruebas se podían realizar para determinar si un RTOS cumple con requerimientos de tiempo real, logrando el diseño de las mismas

Se implementarón las pruebas utilizando el API de FreeRTOS, se analizaron de forma crítica y valorativa los resultados alcanzados en dichas pruebas, comparándolos con resultados obtenidos por pruebas hechas a otros RTOS, quedando demostrado que el FreeRTOS está a la medida de otros sistemas operativos de tiempo real.

RECOMENDACIONES

Implementar más pruebas para con el objetivo de valorar aun más el desempeño del FreeRTOS.

Se recomienda utilizar como referencia el presente trabajo, en caso de implementar pruebas de desempeño a otros sistemas operativos de tiempo real.

BIBLIOGRAFÍA

Express Logic, Inc. 2008. *Thread-Metric Benchmark Suite.* Express Logic, Inc. 2008.

Melgarejo, Yailyn Fernández. 2009. *Servicios de Integración con Terceros para el intercambio de Alarmas y Eventos que ocurran en el proceso y el sistema SCADA Guardián del ALBA.* Universidad de las Ciencias Informáticas. 2009. Trabajo de Diploma.

real, Sistemas Operativos de tiempo. 2008. Universidad de Oviedo. *Universidad de Oviedo.* [Online] 2008. [Cited: 3 15, 2011.] http://isa.uniovi.es/docencia/TiempoReal/Recursos/temas/sotr.pdf.

ReqSO, Requerimientos de los Sistemas Operativos de tiempo real. 2004. Universidad de Valencia. [Online] 2004. [Cited: 2 8, 20011.] http://www.uv.es/gomis/Apuntes_SITR/Sistemas_Operativos.pdf.

Ritchie, Dennis M. 1993. Chistory. *Chistory.* [Online] 1993. [Cited: 3 6, 2011.] http://cm.bell-labs.com/cm/cs/who/dmr/chist.html.

Rivas, Mario Aldea. 2002. *Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas.* Universidad de Cantabria. 2002. Tesis Doctoral.

Rubio, David Magin Florez. 2005. *Arquitectura de Sistemas Multiagente para Micro controladores.* Facultad de Ingeniería Electrónica, Pontifica Universidad Javeriana . 2005. Trabajo de Diploma.

Vázquez, Fidel González. 2007. *Desarrollo SCADA Nacional Evaluación de servicios de tiempo real.* PDVSA- ALBET. 2007.

Xu, Tao. 2008. *Performance benchmarking of FreeRTOS and its Hardware Abstraction.* Technische Universiteit Eindhoven. 2008. Master's Thesis.

ANEXOS

Anexo1: Tabla de resultado de las pruebas.

	Linux parche de tiempo real	ThreadX®RTOS	FreeRTOS
Latencia Planificador	**11 453 ns**	**----**	**300 000 ns**
Cambio de Contexto	**----**	**579 190 iteraciones**	**928 420 iteraciones**
Parámetro Prioridad	**----**	**----**	**Se cumplen**
Implementación de Hilos	**----**	**----**	**Tiene estabilidad**
Procesamiento de Semóforo	**----**	**1 566 675 iteraciones**	**3 237 638 iteraciones**
Procesamiento de Mensajes	**----**	**1 501 724 iteraciones**	**2 104 794 iteraciones**