UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Máster Oficial en Sistemas Inteligentes
y
Aplicaciones Numéricas en Ingeniería

Trabajo Final de Master

Caracterización y estudio de los sistemas operativos de tiempo real para sistemas embebidos FreeRTOS y ChibiOS/RT sobre un procesador SAM3X8E

Guillermo Augusto Cedeño Rodríguez

Tutor: Jorge Cabrera Gámez

Diciembre 2014

AGRADECIMIENTOS

A Dios, familiares, compañeros del máster y amigos, agradezco el apoyo constante e incondicional que me han permitido culminar con éxitos este anhelo de titularme como máster, en tierras lejanas pero muy acogedoras.

A mi madre, el soporte fundamental de mi existencia, la que siempre estuvo a mi lado a pesar de que nos separaba todo un océano. Fiel amiga y consejera que si no fuera por su sacrificio, no hubiera podido alcanzar los logros profesionales de mi vida. Pero más que nada le agradezco su amor incondicional y eterno.

A mis hermanas, que han estado incondicionalmente en los buenos y malos momentos, que siempre han querido lo mejor para mí y fueron un gran apoyo a pesar de la distancia.

A mis profesores del máster, agradezco la formación y el conocimiento que me inculcaron, así como la acogida y apertura que me brindaron desde el primer día.

A D. Jorge Cabrera Gámez, tutor del presente trabajo de fin de máster. Por su apoyo, sus consejos invaluables, sus valiosas contribuciones y la tolerancia a lo largo de este tiempo, tanto en el ámbito académico como en la realización del presente trabajo. Pero sobre todo le agradezco su amistad, la acogida y la confianza que siempre me ha brindado.

A mi compañero del máster y gran amigo Ángel Ramos de Miguel, por haber sido ese amigo incondicional dentro y fuera de las aulas del máster.

A los extraordinarios hermanos y amigos de la comunidad ADSIS-Canarias, por haberme recibido con los brazos abiertos, como un integrante de la casa. Con ellos compartí el diario vivir y se constituyeron en mi familia, durante el tiempo que duraron mis estudios del máster, haciéndome sentir como en casa.

A Las Palmas de Gran Canaria, por haberme permitido aprender de su historia, cultura y tradiciones. Gracias por haberme acogido y hacerme sentir como en casa.

A la Pontificia Universidad Católica del Ecuador Sede Esmeraldas (PUCESE) por haberme permitido, mediante una beca, viajar a Las Palmas de Gran Canaria – España, para seguir mis estudios de máster, y luego volver, para replicar lo aprendido en beneficio de colectividad esmeraldeña y de la propia comunidad universitaria.

RESUMEN

El presente trabajo de fin de master expone diferentes aspectos y conceptos que constituyen los fundamentos de los sistemas operativos de tiempo real (RTOS). Se definen las características, más importantes, de algunos de los principales RTOS que se emplean en sistemas embebidos, incluyendo dos RTOS que se constituyen en objetos del presente trabajo que son FreeRTOS y ChibiOS/RT, que son libres, de código abierto y muy populares.

Luego, se describe los que son las placas de Arduino, se especifican las características de sus diferentes versiones, se presenta su entorno integrado de desarrollo (IDE) y se profundiza en la versión Due, describiendo las características y la estructura hardware de esta plataforma, así como las de su microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3.

Posteriormente, se detalla en qué consisten las comparativas de rendimiento Rhealstone que se aplicaron en los dos RTOS (FreeRTOS y ChibiOS/RT), que se ejecutaron en el microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3 del Arduino Due. Las mencionadas comparativas son: tiempo de conmutación de tarea, tiempo de expulsión, tiempo de espera de un semáforo, tiempo de ruptura de interbloqueo y latencia de paso de mensaje entre tareas. Se finaliza este apartado explicado lo que es y cómo se calcula el valor único de rendimiento Rhealstone.

Ulteriormente, se presenta la aplicación práctica de las cinco comparativas Rhealstone sobre los RTOS FreeRTOS y ChibiOS/RT, bajo la plataforma hardware del Arduino Due. Se explica en detalle, la configuración inicial que se realizó tanto para el IDE de Arduino como para los RTOS, y la forma en que se codificaron cada una de las comparativas en los mencionados RTOS, mostrando los resultados obtenidos en tablas individuales, para finalmente presentar una única tabla comparativa que recoge todos los resultados alcanzados. Basado en estos resultados de las diferentes comparativas, se calcula el valor único de rendimiento Rhealstone para cada RTOS. Y aunque no forma parte de las comparativas Rhealstone, se consideró interesante, debido a que generalmente en los sistemas embebidos se emplean microcontroladores con memoria de programa de pequeño tamaño, determinar la memoria Flash (espacio de programa) que ocupan las comparativas Rhealstone realizadas para los RTOS FreeRTOS y ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due; mostrándose los resultados alcanzados para este aspecto en una tabla comparativa.

Finalmente, se presentan las conclusiones del trabajo realizado, junto a líneas de trabajo futuro. Además se muestran las referencias consultadas y en los apéndices, el código fuente de las cinco comparativas Rhealstone, realizadas para los dos RTOS (FreeRTOS y ChibiOS/RT), que se ejecutaron en el microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3 del Arduino Due.

Palabras clave: Arduino Due, ChibiOS/RT, FreeRTOS, Pruebas de rendimiento, Sistema Embebido de Tiempo Real, Rhealstone, SAM3X8E.

ÍNDICE

RESUMEN **3**

INTRODUCCIÓN **8**

1 SISTEMAS OPERATIVOS DE TIEMPO REAL

1.1 Sistema de tiempo real 9

1.1.1 Tipos de sistemas de tiempo real 9

1.1.2 Estímulo – Respuesta 9

1.2 Sistema operativo de tiempo real (RTOS) 10

1.2.1 Características de un RTOS 10

1.2.2 Requisitos de un RTOS 11

1.2.3 Latencia y fluctuaciones 11

1.3 Sistemas embebidos aplicando un RTOS 11

1.3.1 Tarea 12

1.3.2 Planificador 12

1.3.3 Planificación tipo "Round Robin" 13

1.3.4 Interrupciones 13

1.3.5 Estructuración de prioridades 14

1.3.6 Semáforos 15

1.3.7 Mútex 15

1.3.8 Colas 15

1.4 Ejemplos de los principales RTOS empleados en sistemas embebidos 16

1.4.1 QNX Neutrino RTOS 16

1.4.2 VxWorks 17

1.4.3 ChorusOs 17

1.4.4 ECos 18

1.4.5 LynxOS 19

1.4.6 FreeRTOS 19

1.4.7 ChibiOS/RT 21

2 ARDUINO DUE

2.1 Arduino 25

2.2 Entorno integrado de desarrollo 26

2.2.1 Sketches 27

2.2.2 Librerías 28

2.2.3 Memoria Flash (espacio de programa) 28

2.2.4 Memoria SRAM (Static Random Access Memory o memoria estática de acceso aleatorio) .. 28
2.3 Arduino Due ... 28
2.3.1 Beneficios del núcleo ARM ... 29
2.3.2 Características ... 29
2.3.3 Diferencias del Arduino Due con otros modelos de Arduino ... 30
2.3.4 Alimentación ... 30
2.3.5 Memoria ... 31
2.3.6 Entradas y salidas ... 31
2.3.7 Comunicación ... 32
2.3.8 Programación ... 32
2.3.9 Protección de sobrecarga USB ... 33
2.3.10 Características físicas y compatibilidad con shields ... 33
2.4 Microcontrolador Atmel SAM3X8E ARM Cortex-M3 ... 33
2.4.1 Microcontrolador ... 33
2.4.2 Atmel Corporation ... 34
2.4.3 ARM ... 34
2.4.4 ATSAM3X8E ... 36
3 COMPARATIVAS DE RENDIMIENTO RHEALSTONE
3.1 Introducción ... 38
3.2 Métricas Rhealstone ... 38
3.2.1 Tiempo de conmutación de tarea (Task-Switching Time) ... 38
3.2.2 Tiempo de expulsión (Preemption Time) ... 38
3.2.3 Tiempo de espera de un semáforo (Semaphore Shuffle Time) ... 39
3.2.4 Tiempo de ruptura de interbloqueo (Deadlock Breaking Time) ... 39
3.2.5 Latencia de paso de mensaje entre tareas (Intertask Messaging Latency) ... 40
3.3 Valor único de rendimiento Rhealstone ... 40
4 APLICACIÓN DE LAS COMPARATIVAS RHEALSTONE SOBRE LOS SISTEMAS OPERATIVOS DE TIEMPO REAL FREERTOS Y CHIBIOS/RT BAJO LA PLATAFORMA HARDWARE DEL ARDUINO DUE
4.1 Introducción ... 42
4.2 Descripción de las comparativas ... 43
4.2.1 Tiempo de conmutación de tarea (Task-Switching Time) ... 43
4.2.2 Tiempo de expulsión (Preemption Time) ... 46
4.2.3 Tiempo de espera de un semáforo (Semaphore Shuffle Time) ... 50
4.2.4 Tiempo de ruptura de interbloqueo (Deadlock Breaking Time) ... 55
4.2.5 Latencia de paso de mensaje entre tareas (Intertask Messaging Latency) ... 60

4.3 Resultados de las comparativas Rhealstone 64
4.4 Cálculo del valor único de rendimiento Rhealstone 64
4.5 Memoria Flash (espacio de programa) que ocupan las comparativas Rhealstone realizadas para los RTOS FreeRTOS y ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due 65
5 CONCLUSIÓN Y TRABAJO FUTURO
5.1 Conclusión 68
5.2 Trabajo futuro 70
REFERENCIAS 71
APÉNDICES
Apéndice A: Código para determina el número de iteraciones que dura un tick de reloj (1 milisegundo) 73
Apéndice B: Código FreeRTOS del Rhealstone de tiempo de conmutación de tarea 73
Apéndice C: Código ChibiOS/RT del Rhealstone de tiempo de conmutación de tarea 74
Apéndice D: Código FreeRTOS del Rhealstone de tiempo de preferencia 75
Apéndice E: Código ChibiOS/RT del Rhealstone de tiempo de preferencia 76
Apéndice F: Código FreeRTOS del Rhealstone de tiempo de espera de un semáforo 78
Apéndice G: Código ChibiOS/RT del Rhealstone de tiempo de espera de un semáforo 79
Apéndice H: Código FreeRTOS del Rhealstone de tiempo de ruptura de interbloqueo 81
Apéndice I: Código ChibiOS/RT del Rhealstone de tiempo de ruptura de interbloqueo 83
Apéndice J: Código FreeRTOS del Rhealstone de latencia de paso de mensaje entre tareas 85
Apéndice K: Código ChibiOS/RT del Rhealstone de latencia de paso de mensaje entre tareas 86

LISTA DE FIGURAS

Figura 1: Funcionamiento de un RTOS 10
Figura 2: Estados de una tarea 12
Figura 3: Pantalla inicial de la versión 1.5.6-r2 del IDE de Arduino 27
Figura 4: Parte frontal de la placa Arduino Due 29
Figura 5: Parte posterior de la placa Arduino Due 29
Figura 6: Diagrama de bloques del ATSAM3X8E 35
Figura 7: Métrica Rhealstone - tiempo de conmutación de tarea 38
Figura 8: Métrica Rhealstone - tiempo de expulsión 39
Figura 9: Métrica Rhealstone - tiempo de espera de un semáforo 39
Figura 10: Métrica Rhealstone - tiempo de ruptura de interbloqueo 40
Figura 11: Métrica Rhealstone - latencia de paso de mensaje entre tareas 40
Figura 12: Memoria Flash que ocupa un sketch con la estructura mínima sin RTOS 66

LISTA DE TABLAS

Tabla 1: Especificaciones de las diferentes versiones de las placas de Arduino 26
Tabla 2: Resultado de la aplicación de la comparativa Rhealstone - tiempo de conmutación de tarea 44
Tabla 3: Resultado de la aplicación de la comparativa Rhealstone - tiempo de expulsión 47
Tabla 4: Resultado de la aplicación de la comparativa Rhealstone - tiempo de espera de un semáforo....... 51
Tabla 5: Resultado de la aplicación de la comparativa Rhealstone - tiempo de ruptura de interbloqueo..... 56
Tabla 6: Resultado de la aplicación de la comparativa Rhealstone – latencia de paso de mensaje entre tareas 61
Tabla 7: Recopilación de resultados de la aplicación de cada una de las comparativas Rhealstone............. 64
Tabla 8: Recopilación de resultados de las pruebas de memoria Flash necesaria para cargar las comparativas Rhealstone realizadas para los RTOS FreeRTOS y de ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due 67

INTRODUCCIÓN

Los sistemas embebidos, es decir, sistemas con capacidad computacional diseñados con un propósito específico son en la actualidad de uso muy extendido y cotidiano, los podemos encontrar en un taxímetro, un sistema de control de acceso, la electrónica que controla una máquina expendedora o el sistema de control de una fotocopiadora entre otras múltiples aplicaciones. Dentro de las más populares (por su hardware libre, shields y librerías disponibles) de las arquitecturas que soportan este tipo de sistemas se encuentra Arduino y este trabajo busca describir la versión Due.

En las aplicaciones de este tipo de arquitectura muchas veces se requiere de una rápida capacidad de reacción a eventos críticos (porque de lo contrario la aplicación falla) que con la programación tradicional con la que estos dispositivos se programan resulta complicado alcanzarla y es allí donde son útiles los sistemas operativos de tiempo real (RTOS) y el presente trabajo se enfoca en dos de ellos que son de uso muy extendidos y de licencia libre FreeRTOS y ChibiOS/RT, caracterizando su aplicación en la placa del Arduino Due que incluye un microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3.

De todo esto, se desprenden los principales objetivos de este trabajo fin de máster, los cuales son:

- Determinar los fundamentos y características de los sistemas operativos de tiempo real, con énfasis en los que se aplican para sistemas embebidos.
- Describir las características y la estructura hardware de la plataforma del Arduino Due, así como las de su microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3.
- Caracterizar la aplicación de FreeRTOS y ChibiOS/RT en la plataforma hardware del Arduino Due, mediante comparativas de rendimiento Rhealstone.

Cuando en un sistema embebido se requiere aplicar un RTOS, resulta de gran utilidad poder contar con información referente a estudios comparativos de rendimiento, basados en métricas establecidas como la Rhealstone, que se desarrolla en el presente trabajo; para conocer los tiempos de respuestas que al momento de elegir el RTOS adecuado, constituye un aspecto muy determinante.

Las métricas de evaluación comparativa Rhealstone para los dos RTOS (FreeRTOS y ChibiOS/RT) que se desarrollan en el presente trabajo, se basan en un trabajo realizado por Rabindra P. Kar [24] que es la aplicación de las comparativas Rhealstone para el sistema operativo de tiempo real iRMX (un RTOS diseñado específicamente para su uso con la familia de procesadores Intel 8080 e Intel 8086) el cual se lo ha trasladado a las instrucciones equivalentes en FreeRTOS y ChibiOS/RT, así como a las características de trabajo del Arduino Due.

Debido a que generalmente, en los sistemas embebidos se emplean mirocontroladores con pequeño tamaño de memoria de programa, se consideró importante la determinación de la memoria Flash (espacio de programa) que ocupan las comparativas Rhealstone realizadas, para los dos RTOS (FreeRTOS y ChibiOS/RT) en el microcontrolador ATSAM3X8E del Arduino Due.

En el presente trabajo se muestran tablas individuales con los resultados obtenidos para cada una de las métricas de evaluación comparativa Rhealstone, realizadas tanto para FreeRTOS como para ChibiOS/RT, posteriormente, se presenta una tabla general que recoge todas las métricas y se establecen comparaciones entre los resultados alcanzados, y finalmente, se calcula el valor único de rendimiento Rhealstone para cada RTOS, en base a estos resultados de las métricas. De igual forma, se procede con una tabla general para los resultados obtenidos del espacio de memoria Flash que ocupan las métricas realizadas en el microcontrolador ATSAM3X8E del Arduino Due.

CAPÍTULO I

SISTEMAS OPERATIVOS DE TIEMPO REAL

1.1 Sistema de tiempo real

Un sistema de tiempo real es aquel sistema informático en el que la corrección del sistema no sólo depende de los resultados lógicos de los algoritmos, sino que también depende del momento en el que estos se producen. En un sistema de tiempo real, la corrección semántica de la respuesta es responsabilidad del programador, y la corrección temporal depende del sistema operativo. Es el sistema operativo el que tiene que dar soporte y planificar la ejecución de todas las tareas, gestionar las interrupciones y encargarse de las comunicaciones (semáforos, colas, etc.).

Al contrario que sucede en los sistemas operativos convencionales, el objetivo de los de tiempo real es minimizar la complejidad para minimizar la incertidumbre (falta de predictibilidad). No se quiere un sistema operativo que haga muchas cosas, sino uno que las haga de forma predecible y rápida. Las características del sistema en tiempo real incluyen el cumplimiento de determinados plazos en el momento adecuado [1].

1.1.1 Tipos de sistemas de tiempo real

Los sistemas de tiempo real pueden ser de dos tipos, en función de su severidad en el tratamiento de los errores que puedan presentarse:

- Sistemas de tiempo real blandos: Estos sistemas pueden tolerar que una acción termine fuera de plazo de vez en cuando, siempre y cuando se garantice la ejecución a tiempo de las tareas críticas.
- Sistemas de tiempo real duros: Estos sistemas no pueden tolerar que una acción termine tarde ni una sola vez, ya que esta circunstancia podría tener consecuencias catastróficas. En ellos la respuesta fuera de término no tiene valor alguno y produce la falla del sistema.

1.1.2 Estímulo – Respuesta

De acuerdo a [2] una forma de ver un sistema de tiempo real es como un sistema de estímulo - respuesta. Dado un estímulo de entrada, el sistema debe producir la correspondiente salida. Se puede, por lo tanto, definir el comportamiento de un sistema de tiempo real haciendo una lista de estímulos recibidos por el sistema, las respuestas asociadas y el tiempo real en que dicha respuestas deben producirse. Los estímulos pueden pertenecer a dos clases:

- Estímulos periódicos: Ocurren a intervalos de tiempos predecibles, por ejemplo, el sistema debe leer un sensor cada 50 milisegundos y realizar una acción respuesta dependiendo del valor de ese sensor (estimulo).
- Estímulos aperiódicos: Ocurren de forma irregular, normalmente son provocados mediante interrupciones. Un ejemplo de dicho estimulo podría ser una interrupción para reaccionar ante el disparo de una alarma de incendio que se ha activado, disparando el correspondiente sistema de extinción. se ha activado y que se debe activar el sistema respectivo.

Los estímulos periódicos en un sistema de tiempo real son generados normalmente por sensores asociados al sistema. Estos proporcionan información sobre el estado del entorno al sistema, las respuestas son dirigidas a un conjunto de actuadores que controlan a un equipo. Los estimulo aperiódicos pueden generarse por actuadores o sensores.

Un sistema de tiempo real tiene que responder a estímulos que ocurren en diferentes instantes de tiempo por lo tanto se tiene que organizar su arquitectura para que, tan pronto como se reciba un estímulo, el control sea trasferido adecuadamente.

Los sistemas de tiempo real están diseñados como un conjunto de procesos que colaboran entre sí. La generalización de este estimulo - repuesta de un sistema de tiempo real conduce a un modelo arquitectónico genérico en el que hay dos tipos de procesos. Los procesos computacionales que calculan la respuesta requerida para el estímulo recibido del sistema y los procesos de control del funcionamiento del actuador. Este modelo permite recoger rápidamente los datos desde el sensor (antes de la siguiente entrada esté disponible) y permiten que su procesamiento y la respuesta asociada al actuador se realice.

1.2 Sistema operativo de tiempo real (RTOS)

Los sistemas operativos de tiempo real o RTOS (Real-Time Operating System), son sistemas los cuales manejan sucesos o eventos de gran importancia, por lo que deben cumplir con sus tareas bajo ciertas restricciones, es decir, este tipo de sistemas deben dar prioridad a los procesos según la importancia que se determina dependiendo del problema. A estos sistemas se les exige reacción en sus respuestas bajo ciertas restricciones de tiempo. Si no las respeta, se dirá que el sistema ha fallado. Se considera que un RTOS funciona correctamente si produce un resultado correcto dentro de los intervalos de tiempo estipulados. La figura 1 explica gráficamente su funcionamiento.

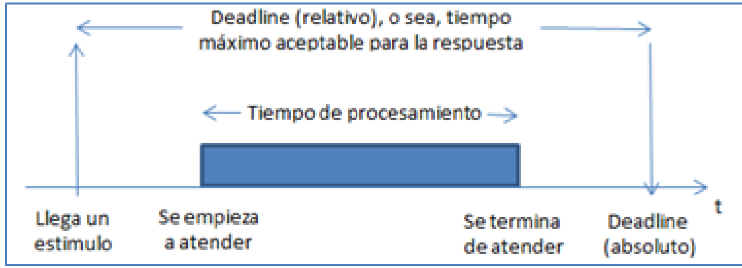

Figura 1: Funcionamiento de un RTOS

Un sistema operativo de tiempo real proporciona un conjunto de llamadas al sistema que permiten el desarrollado de un sistema en tiempo real [3]. Los sistemas operativos de tiempo real se presentan en entornos en donde deben ser aceptados y procesados una gran cantidad de sucesos, siendo la mayoría de estos sucesos externos al sistema computacional, con un tiempo de respuesta inmediato. De ahí que los RTOS se encuentran en aplicaciones como control de tráfico aéreo, bolsas de valores, control de refinerías, telecomunicaciones, control de trenes, control de edificios, etc. También están presente en el ramo automovilístico y de la electrónica de consumo.

1.2.1 Características de un RTOS

Los factores que normalmente caracterizan a un RTOS son:

- Tiempo de cambio de contexto o cuánto tarda el RTOS en desplazar una tarea por otra de mayor prioridad.
- Memoria consumida: Este factor determina si un RTOS es utilizable en sistema empotrables cuya memoria está fuertemente limitada.
- Capacidad de responder rápidamente a interrupciones externas, sin perder un solo suceso.
- Planificación preferente basadas en prioridades.
- Reducción de intervalos en los que están inhabilitadas las interrupciones.

- Primitivas para la gestión de hebras (creación, detención) y su sincronización (variables condición, mútexes, colas de mensajes, etc.).

1.2.2 Requisitos de un RTOS

Determinismo: Es la capacidad de determinar con una alta probabilidad, cuanto es el tiempo que tarda una tarea en iniciar (reconocer la interrupción), es decir, que los RTOS necesitan que ciertas tareas se comiencen a ejecutar antes que otras.

Sensibilidad: Se refiere a cuánto tiempo consume un sistema operativo en dar servicio a la interrupción después de reconocerla.

Control de usuario: Es mucho mayor en un sistema operativo de tiempo real que en un sistema operativo típico, donde el usuario no tiene control sobre la función de planificación del sistema operativo. En un sistema de tiempo real resulta esencial permitir al usuario un control preciso sobre la prioridad de las tareas. El usuario debe poder distinguir entre tareas rígidas y flexibles y especificar prioridades relativas dentro de cada clase.

Fiabilidad: Es normalmente mucho más importante en sistemas de tiempo real que en los que no lo son. Un fallo transitorio en un sistema que no sea de tiempo real puede resolverse simplemente volviendo a reiniciar el sistema. Pero como un sistema de tiempo real responde y controla sucesos de tiempo real, las pérdidas o degradaciones del rendimiento pueden tener consecuencias catastróficas, que pueden ir desde pérdidas financieras hasta daños en equipo e incluso pérdida de vidas humanas.

Estabilidad: Si en los casos en los que para el RTOS es imposible cumplir todos los plazos de ejecución de las tareas se cumplen al menos los de las más críticas y de mayor prioridad

1.2.3 Latencia y fluctuaciones

El objetivo de un sistema operativo de tiempo real es reducir la latencia de un evento y la fluctuación en las interrupciones, tanto internas como externas. Es decir, el aspecto fundamental en un sistema operativo de tiempo real que lo diferencia de un sistema operativo de propósito general, es el tiempo requerido para reaccionar ante interrupciones y otro tipo de eventos.

1.3 Sistemas embebidos aplicando un RTOS

Un sistema embebido o empotrado es un sistema electrónico que posee capacidad computacional y está diseñado con un propósito específico, al contrario de lo que ocurre con las computadoras de propósito general (como por ejemplo una computadora personal) que están diseñados para cubrir un amplio margen de necesidades. Estos sistemas realizan una o muy pocas tareas dentro de un entorno de características especiales y particulares, y su diseño está pensado para reducir costos y aumentar sus prestaciones. En un sistema embebido la mayoría de los componentes se encuentran incluidos en la placa base y muchas veces los dispositivos obtenidos no tienen la forma que se suele relacionar a una computadora. Algunos ejemplos de sistemas embebidos podrían ser dispositivos como un taxímetro, el sistema de seguridad de una vivienda, la electrónica que controla un cajero automático o el sistema de control de una fotocopiadora entre otras múltiples aplicaciones. En los RTOS [4] importan los procesos y no el usuario, es utilizado principalmente en entornos donde se procesan gran cantidad de eventos.

Un RTOS es diferente a un sistema operativo convencional de tiempo compartido como Mac OS X, Linux o Windows. Ya que estos últimos nada más arrancar el dispositivo computacional toma el control de este y luego ejecutan los programas de aplicación, mientras que un RTOS es una librería de funciones que se

enlaza con la aplicación, de forma que al arrancar el sistema embebido es la aplicación la que toma el control e inicializa al RTOS. Esto permite eliminar las partes del sistema operativo que no se usan en un sistema embebido para ahorrar memoria, que suele estar limitada en estos dispositivos.

En sistemas basados en microcontroladores, un RTOS no se protege frente a errores de las aplicaciones. Esto ocurre para simplificar el diseño y el hardware necesario, dado que los sistemas empotrados habitualmente sólo ejecutan una aplicación, si esta se detiene ya da igual que se detenga todo el sistema con ella, por lo que a la hora de diseñar los programas de tiempo real hay que ser muy cuidadosos.

1.3.1 Tarea

La tarea [5] es el bloque básico de un programa basado en RTOS, una tarea puede entenderse como una función que obviamente puede llamar a otras funciones. La única condición que ha de cumplir una función para convertirse en una tarea es que no termine nunca de realizar sus acciones. La tarea se inicializa mediante una llamada al RTOS, especificándose en dicha llamada la prioridad de la tarea. Los RTOS pueden ejecutar un número arbitrario de tareas, estando limitado dicho número sólo por la memoria disponible en el sistema. Una tarea puede estar en los siguientes tres estados (figura 2):

Ejecución: El procesador la está ejecutando. Sólo puede haber una tarea en ese estado.

Lista: La tarea tiene trabajo que hacer y está esperando a que el procesador esté disponible. Puede haber un número cualquiera en este estado.

Bloqueada: No tiene nada que hacer en este momento, está esperando algún suceso externo. Puede haber un número cualquiera de tareas en este estado.

Figura 2: Estados de una tarea

1.3.2 Planificador

En los sistemas de tiempo real [6] es fundamental hacer una buena planificación para aprovechar al máximo los recursos disponibles (fundamentalmente el procesador) sin llegar a sobrecargarlos, evitando así que los trabajos se completen fuera del plazo previsto. Así pues, debe haber un elemento denominado planificador que se encargue de asignar los recursos apropiados a los trabajos pendientes para que se realicen de manera eficiente.

A la hora de planificar un sistema concurrente se utiliza como unidad básica la tarea, que es una secuencia de operaciones que tiene restricciones de tiempo definidos a partir de una serie de parámetros (prácticamente equivalente a una hebra de ejecución). Una tarea se activa en un momento dado que se

llama activación y tiene que completar una serie de instrucciones que llevan asociado un cierto tiempo de ejecución antes de un instante que se denomina plazo de respuesta.

Mediante una política de planificación basada en prioridades fija se puede asegurar que en caso de sobrecarga, las primeras en incumplir su plazo de respuesta serían las tareas con un nivel de prioridad más bajo. Por tanto, si se considera asignar las prioridades en función de la importancia de las tareas, se podría afirmar que en caso de sobrecarga las primeras en perder su plazo serían aquellas tareas menos importantes.

El planificador [5] es la parte del RTOS que controla el estado de cada tarea y decide cuándo una tarea pasa al estado de ejecución. El principio es muy simple, exceptuando las tareas que están bloqueadas, la tarea con mayor prioridad es la que estará en el estado de ejecución.

Aunque haya tarea de mayor prioridad ejecutándose, los sistemas operativos de propósito general como Linux o Windows de vez en cuando asignan un poco de tiempo de CPU a las tareas de menor prioridad para que sus usuarios no se desesperen. Para ello sus planificadores ejecutan algoritmos más o menos elaborados. Por el contrario, los planificadores de los RTOS son más básicos, por lo que si una tarea de mayor prioridad se apropia del procesador durante un largo tiempo, las otras tareas tendrán que esperarse. Por lo que la elección de la prioridad es fundamental.

En la figura 2 se puede observar también las transiciones de las tareas realizadas por el planificador, aquí se muestra que una tarea sólo puede pasar a bloqueada cuando se está ejecutando, ya que sólo en este estado una tarea puede necesitar algún dato proporcionado por otra tarea o se identifica que debe esperar un tiempo determinado; para que una tarea pase de bloqueada a lista, otra tarea debe provocar las acciones que la tarea bloqueada espera y así activarla (estado "lista"), una vez que la tarea se activa su paso al estado de ejecución depende de que se convierta en la de mayor prioridad para el planificador.

1.3.3 Planificación tipo "Round Robin"

Cuando se tiene varias tareas con la misma prioridad que están en estado "lista" y con prioridad de ejecutarse, se emplea un algoritmo de planificación tipo "Round Robin" que consiste en que a cada tarea de igual prioridad que debe ejecutarse se le asigna un intervalo de tiempo llamado quantum, cada una de estas tareas se ejecuta durante ese quantum de tiempo. En la planificación tipo "Round Robin" pueden pasar dos cosas:

- La tarea tiene un tiempo de procesamiento menor que el quantum: entonces el proceso termina antes que el quantum y se da paso a la siguiente tarea.
- La tarea tiene un tiempo de procesamiento mayor que el quantum: entonces cuando se acaba el quantum, se expulsa a la tarea y se da paso a la siguiente tarea, colocándose la tarea saliente al final de la cola de tareas con estado "lista".

1.3.4 Interrupciones

Las interrupciones son cambios en el flujo de control, no ocasionados por el programa que se ejecuta, sino por algún otro suceso que necesita el servicio inmediato del procesador. La señal de petición de interrupción provoca que el procesador detenga el programa en curso, salve su estado (es decir, se guardan todos los contenidos de los registros de la CPU) y transfiera el control a una rutina de servicio de interrupción (ISR - del inglés "Interrupt Service Routine") la cual realiza alguna acción apropiada para darle servicio a la petición. Al terminar el servicio de la interrupción, se debe continuar con el código interrumpido exactamente en él mismo estado en que estaba cuando tuvo lugar la interrupción, lo cual se logra restaurando los registros internos al estado que tenían antes de la interrupción previamente salvado permitiendo continuar el flujo normal de procesamiento.

1.3.5 Estructuración de prioridades

En los RTOS el desarrollador debe asignar prioridades a las tareas. La prioridad que asigne dependerá de cómo de rápido deba responder una tarea a un evento particular. Los eventos pueden ser o bien una actividad que deba realizar una tarea o simplemente el finalizar el tiempo de espera que hubiera solicitado previamente una tarea. En general la mayoría de las tareas de un RTOS se pueden clasificar en alguno de los siguientes niveles de prioridad:

Nivel de interrupción: En este nivel se encuentran los manejadores de interrupción y las rutinas de servicio a las tareas y dispositivos que requieren la más rápida respuesta posible.

Nivel de reloj: En este nivel se encuentran las tareas que requieren un procesamiento repetitivo, como por ejemplo las tareas de muestreo y control o aquellas que requieren temporizaciones precisas.

Nivel de prioridad base: Las tareas que se encuentran en este nivel son las tareas de más baja prioridad, las que no tienen finalización exacta o aquellas tareas que tienen un amplio margen de error en las temporizaciones. Las tareas en este nivel pueden tener asignadas distintas prioridades o pueden ejecutarse con un único nivel de prioridad que podría ser el mismo que el nivel de prioridad base del planificador.

1.3.5.1 Nivel de interrupción

Una interrupción genera una nueva planificación fuera del control del sistema, por lo que las rutinas de manejo de interrupciones deben tener un mecanismo para grabar el contexto de la tarea interrumpida. Una vez ejecutada la rutina de interrupción se restaurará el contexto y como consecuencia se retornará a la tarea interrumpida. Dentro de este nivel de interrupción habrá diferentes prioridades y se tendrá que evitar que interrupciones de prioridad baja interrumpan a las que se estén ejecutando con una prioridad mayor.

1.3.5.2 Nivel de reloj

Cada interrupción de reloj es lo que se conoce como un tick y representa el intervalo de tiempo más pequeño conocido por el sistema. La función de la rutina de interrupción del reloj será actualizar el reloj-calendario y transferir el control al planificador. Posteriormente el planificador seleccionará la tarea que deberá arrancar en cada tick de reloj concreto. Las tareas de nivel de reloj se dividen en dos categorías:

- Cíclicas: Son las tareas que requieren una sincronización precisa con el mundo exterior.
- Delay (retardo): Son las tareas que requieren tener un retardo fijo entre sucesivas activaciones o que retardan sus actividades por un periodo de tiempo.

1.3.5.3 Nivel de prioridad base

Las tareas que se ejecutan en este nivel se ejecutan por petición más que por un intervalo determinado. Esta petición puede ser una entrada de un usuario desde un terminal, algún evento que espera un proceso, o algún requerimiento particular para un dato que se procesa.

1.3.6 Semáforos

Para que el RTOS no ejecute accesos simultáneos a un recurso es necesario implementar un mecanismo de control. Los semáforos son usados para sincronizar varias tareas dentro del sistema. Cuando una tarea quiera utilizar un recurso compartido protegido por un semáforo, pregunta por el estado del recurso que va a utilizar, si está ocupado la tarea quedará en espera hasta que se libere, de lo contrario toma y bloquea el semáforo realizando su operación y una vez terminado libera el semáforo permitiendo que otras tareas puedan hacer uso del recurso.

1.3.6.1 Múltiples semáforos:

Emplear múltiples semáforos es lo habitual ya que cada semáforo puede proteger un solo recurso compartido. Siendo la tarea del programador el usar el semáforo correspondiente antes de usar un recurso compartido, ya que el RTOS no puede saber qué semáforos están protegiendo cada uno de los distintos recursos compartidos.

1.3.6.2 Inconvenientes de los semáforos

Los semáforos son de mucha utilidad para proteger recursos compartidos. Sin embargo, al ser el programador el responsable de bloquear y desbloquear el semáforo, se pueden producir inconvenientes en su utilización, los más comunes son:

- Emplear el semáforo equivocado, por lo que también se accederá al recurso compartido sin considerar su semáforo.
- Olvidar liberar el semáforo, ocasionando que ninguna tarea pueda emplear el recurso protegido por el semáforo.
- Se puede aumentar la latencia de las demás tareas que usan el recurso protegido por un semáforo, si se lo tiene bloqueado por mucho tiempo.
- Se puede producir una inversión de prioridad, en la que una tarea de menor prioridad impida la ejecución de una tarea de mayor prioridad.

1.3.7 Mútex

Los mútex son semáforos binarios que incluyen un método de herencia de prioridad. Un mútex permite a las tareas asegurar la integridad de un recurso compartido al que tienen acceso, sus dos estados son bloqueado y desbloqueado. Cuando una tarea desea acceder al recurso primero debe bloquearlo y cuando haya terminado de utilizarlo debe desbloquearlo. Los mútex a diferencia de los semáforos emplean un método de herencia de prioridad, esto significa que si una tarea con mayor prioridad intenta obtener un recurso mientras está siendo usado por una tarea con menor prioridad, esta recibe la prioridad de la tarea que pretende obtener el recurso y cuando el recurso es liberado, la tarea recupera su prioridad real. De esta manera, cuando ocurre este fenómeno, la tarea de alta prioridad se mantiene bloqueada el menor tiempo posible.

1.3.8 Colas

Los RTOS disponen de colas para permitir comunicar varias tareas entre sí de forma fácil. Normalmente se usan:

- Cuando se necesita un almacenamiento temporal para soportar ráfagas de datos.

- Cuando existen varios generadores de datos y un sólo consumidor y no se quiere bloquear a los generadores.

Las colas son una forma primaria de comunicación entre tareas, pueden ser usadas para enviar mensajes entre tareas o interrupciones. Las colas son del tipo FIFO (primero en entrar, primero en salir).

1.4 Ejemplos de los principales RTOS empleados en sistemas embebidos

1.4.1 QNX Neutrino RTOS

El QNX Neutrino RTOS [7] es un sistema operativo de tiempo real con licencia comercial, desarrollado por QNX Software Systems empresa canadiense, que fue adquirida por BlackBerry. Está basado en Unix orientado para satisfacer las necesidades de recursos limitados de los sistemas embebidos de tiempo real. Su diseño de microkernel y arquitectura modular permite a los desarrolladores crear sistemas altamente optimizados y confiables con un bajo costo. Está disponible para arquitecturas: x86, MIPS, PowerPC, SH4, ARM, StrongARM, xScale y BlackBerry Playbook. QNX Neutrino RTOS proporciona:

- Manipular eventos múltiples dentro de restricciones fijas de tiempo.
- Multitarea.
- Planificación por prioridades con desalojo.
- Comunicación entre procesos basada en Mensajes.

QNX Neutrino RTOS adquiere eficacia, modularidad, y simplicidad a través de dos principios fundamentales:

- La arquitectura del microkernel.
- La comunicación entre procesos basada en mensajes.

1.4.1.1 El microKernel

El Microkernel de QNX Neutrino RTOS es responsable de lo siguiente:

- IPC (Comunicación Inter Procesos): El Microkernel supervisa el ruteo de mensajes; también maneja otras dos formas de IPC: proxies y señales.
- La comunicación de la red a bajo nivel: El Microkernel entrega todos los mensajes destinados a los procesos en otros nodos.
- Planificador de procesos: El planificador del Microkernel decide qué proceso se ejecutará luego.
- Manejo de interrupciones de primer nivel: Todas las interrupciones de hardware y las fallas se encaminan primero a través del Microkernel, luego se pasa al driver o al administrador del sistema.

1.4.1.2 El administrador de procesos

El administrador de procesos realiza la planificación sobre el microKernel y es el responsable de crear nuevos procesos en el sistema y manejar los recursos más fundamentales asociados con un proceso. Estos servicios son proporcionados vía mensajes. Por ejemplo, si un proceso corriente quiere crear un nuevo proceso, envía un mensaje que contiene los detalles del nuevo proceso a ser creado. Las primitivas de creación de procesos disponibles son:

- fork() crea un nuevo proceso que es una imagen exacta del proceso que lo invocó. El nuevo proceso comparte el mismo código que el proceso que invocó la primitiva y hereda una copia de los datos del mismo.

- exec() reemplaza la imagen del proceso que la invoca con una nueva imagen del mismo.
- spawn() crea un nuevo proceso como un hijo del proceso invocante. Puede evitar la necesidad de fork() y exec(), produciendo un medio más rápido y más eficaz para crear nuevos procesos. La primitiva spawn() puede crear procesos en cualquier nodo en la red.

1.4.1.3 Protección de memoria

Mientras muchos de los kernels de tiempo real proveen soporte para la protección de memoria en tiempo de desarrollo, pocos lo hacen para el tiempo de ejecución, debido a la perdida de desempeño como principal razón. La ventaja clave ganada por añadir memoria protegida, especialmente para sistemas de misión crítica, es la robustez.

Con protección de memoria, si un proceso que se está ejecutando en un ambiente multitarea intenta acceder a la memoria que no ha sido explícitamente declarada, la unidad de administración de memoria puede notificar al sistema operativo, para que luego este pueda abortar el proceso.

1.4.2 VxWorks

VxWorks [8] es un sistema operativo de tiempo real, basado en Unix, vendido y fabricado por Wind River Systems. VxWorks puede ejecutarse en prácticamente todos los procesadores del mercado de sistemas embebidos, esto incluye la familia de CPUs x86, MIPS, PowerPC, SH-4, ARM, StrongARM y xScale. Algunas de las características del sistema operativo actual son:

- Multitarea con respuesta de interrupción rápida.
- Sistema operativo de 32 bits y 64 bits nativo.
- Aplicaciones en modo de usuario aisladas de otras aplicaciones en modo de usuario, así como la del núcleo a través de mecanismos de protección de memoria.
- Marco de gestión de errores.
- Locales y distribuidas colas de mensajes

1.4.2.1 Mensajería

La mensajería es útil para pasar variables entre tareas asíncronas. VxWorks suministra siete funciones de mensajería: msgQCreate (), msgQDelete (), msgQSend (), msgQReceive (), msgQNumMsgs (), msgQShow (), msgQInfoGet (). Un mensaje puede ser colocado por delante de los mensajes anteriores mediante el envío del atributo MSG_PRI_URGENT.

1.4.2.2 Controladores de dispositivos

El controlador de dispositivo actúa como una interfaz entre el kernel y un dispositivo externo. Un dispositivo típico contiene un administrador para la interrupción del dispositivo. El administrador tendrá las funciones que reconocen la interrupción, inicializar y desactivar el dispositivo. Wind River suministra los controladores para (a) los conductores tradicionales y multimodo de serie, (b) los contadores de tiempo, (c) la memoria no volátil (NVRAM) y (d) controladores de interrupción.

1.4.3 ChorusOs

ChorusOs [9] es un sistema operativo de tiempo real, basado en Unix y desarrollado por la empresa Sun Microsystems. Actualmente está liberado bajo código abierto. Tiene un diseño de microkernel y es usado

en ambientes de aplicaciones distribuidas dedicadas de bajo costo, que necesitan un mínimo de funcionalidad y un mínimo uso de memoria. Soporta las plataformas x86/68k/PPC/SPARC/ARM/MIPS. Es un sistema operativo altamente escalable y de implementación confiable, tanto así que se ha establecido entre los proveedores superiores de telecomunicaciones.

1.4.3.1 Operaciones de comunicación

ChorusOs proporciona dos tipos de operaciones de comunicación: envío asíncrono y llamadas a procedimientos remoto (RPC). El envío asíncrono permite que un hilo sólo envíe un mensaje a un puerto. No existe garantía de que el mensaje llegue a su destino y no existe una notificación si algo sale mal.

La otra operación de comunicación es la RPC. Cuando un proceso ejecuta una operación de RPC, se bloquea en forma automática hasta que llega la respuesta o expira el cronómetro de la RPC, en cuyo momento se elimina el bloqueo del emisor. Se garantiza que el mensaje que elimina el bloqueo del emisor es la respuesta a la solicitud.

1.4.3.2 Abstracciones principales

- Actores: Un actor de ChorusOs es un entorno de ejecución equivalente a una tarea. Un actor puede tener uno o más hilos.
- Puertos: Un puerto es un canal de comunicación unidireccional con una cola de mensajes asociada. Los puertos se pueden migrar entre los actores.
- Mensajes: Un mensaje ChorusOs se compone de un cuerpo de longitud variable (limitado a 64 KBytes).
- Regiones, segmentos y cachés locales: El espacio de direcciones de un actor se divide en regiones. Una región puede ser asignada sobre una porción de un segmento, que es el equivalente de un objeto de memoria. Para cada segmento asignado el microkernel mantiene una caché local.

1.4.4 ECos

ECos [10] es un sistema operativo de tiempo real que funciona en dispositivos de varias arquitecturas, entre ellas ARM, CalmRISC, FR-V, Hitachi H8, IA-32, Motorola 68000, Matsushita AM3x, MIPS, NEC V8xx, Nios II, PowerPC, SPARC, SuperH.. Actualmente está gestionado por la FSF (Fundación del Software libre) aunque inicialmente pertenecía a una empresa dedicada al mundo de Linux como es Red Hat Software. Este RTOS es libre y gratuito.

ECos incluye las herramientas y funciones necesarias para satisfacer a las aplicaciones embebidas: basada en la prioridad en tiempo real del planificador y primitivas de sincronización, bibliotecas de soporte de idiomas, comunicaciones, controladores de dispositivos y soporte de depuración.

Diseñado para sistemas de tiempo real, eCos implementa una arquitectura multitarea clásica con un rico conjunto de primitivas de sincronización. Esto ofrece tiempos de respuesta deterministas, latencias de interrupción mínimos y bajos cambios de contexto fijos.

1.4.4.1 Capa de abstracción de hardware (HAL)

La capa de abstracción de hardware define la arquitectura, cada familia de procesadores soportado por eCos es una arquitectura diferente. El HAL contiene para cada arquitectura el código necesario para el arranque del procesador, la entrega de interrupción, el cambio de contexto, y otra funcionalidad específica de la arquitectura.

1.4.4.2 El microkernel

El microkernel de eCos consta de un planificador y de los mecanismos de sincronización de subprocesos, manejo de excepciones, manejo de interrupciones y temporizadores. El planificador es el corazón del microkernel y contiene dos modos de funcionamiento: de mapa de bits y la planificación de colas. El planificador de mapa de bits representa cada hilo, que debe tener una prioridad única, con un bit en un mapa de bits. La cola implementa los hilos según el número de prioridad. La sincronización de subprocesos se logra mediante el uso de mútex y semáforos. Estos se pueden combinar con las colas de mensajes para la comunicación entre hilos.

1.4.5 LynxOS

LynxOS [11] es un sistema operativo de tiempo real basado en Unix, con licencia comercial y desarrollado por LynuxWorks. LynxOS se utiliza en sistemas embebidos de tiempo real, en las aplicaciones de aviación, la industria aeroespacial, el control de procesos industriales militares y de telecomunicaciones. Soporta arquitecturas Motorola 68010, Intel 80386, ARM, PowerP.

LynxOS está diseñado para el determinismo absoluto, es decir, tiempo real duro. Esto significa que es absolutamente necesario responder en un plazo de tiempo conocido. Esta respuesta predecible está asegurada incluso en la presencia de operaciones de E/S complejas, ya que está diseñado para que las rutinas de interrupción se ejecuten rápidamente. LynxOS Soporta múltiples dispositivos de interrupción.

1.4.6 FreeRTOS

FreeRTOS [12] es un sistema operativo de tiempo real para dispositivos embebidos, desarrollado por Real Time Engineers ltd., tiene un diseño de microkernel y es soportado por 34 arquitecturas, entre ellas ARM (ARM7, ARM9, Cortex-M3, Cortex-M4, Cortex-A), Atmel AVR, AVR32, HCS12, MicroBlaze, Cortus (APS1, APS3, APS3R, APS5, FPF3, FPS6, FPS8), MSP430, PIC, Renesas H8/S, SuperH, RX, x86, 8052, Coldfire, V850, 78K0R, Fujitsu MB91460 series, Fujitsu MB96340 series, Nios II, Cortex-R4, TMS570, RM4x. Se distribuye bajo licencia GPL, con una excepción opcional, la misma que permite utilizar código propietario sin que este sea liberado, facilitando de este modo el uso de FreeRTOS en aplicaciones propietarias. Sus principales características son:

- Funcionamiento apropiable o cooperativo.
- Asignación de prioridades a las tareas de una forma flexible.
- Manejo de colas, utilizadas para el paso de mensajes entre tareas.
- Uso de semáforos y mútex.
- Funciones para distintos estados del planificador: tick del reloj, inactividad, chequeo de desbordamiento de pila.
- Macros para el monitoreo de parámetros de ejecución.
- Soporte y licenciamiento comercial opcional.

1.4.6.1 Implementación

FreeRTOS está diseñado para ser pequeño y sencillo. El microkernel en sí consta de sólo cuatro archivos de C. Para hacer que el código sea legible, fácil de portar, y fácil de mantener, está escrito principalmente en C, salvo en secciones específicas a cada arquitectura donde es necesario el uso de ensamblador.

FreeRTOS proporciona métodos para multitarea, exclusiones mutuas, semáforos, temporizadores y soporta tareas con prioridades, incluyendo colas de mensajes y semáforos binarios. El mecanismo de cola de FreeRTOS se puede utilizar en las comunicaciones entre dos tareas, así como para la comunicación entre las

tareas y la rutina de servicio de interrupción. Una cola es una estructura que es capaz de almacenar y restaurar los datos [13].

La descarga desde el Internet de FreeRTOS contiene configuraciones y demostraciones para todos los puertos y el compilador ya preparado, lo que permite el diseño rápido de aplicaciones. FreeRTOS también contiene una gran cantidad de documentación y tutoriales.

1.4.6.2 Gestión de planificación

El planificador de tareas tiene como objetivo decidir qué tarea en estado "listo" se debe ejecutar en un momento dado. FreeRTOS logra este objetivo con las prioridades dadas a las tareas mientras se crean, la prioridad de una tarea es el único elemento que el planificador tiene en cuenta para decidir qué tarea tiene que ejecutar. Cada tick de reloj hace que el planificador compruebe si debe desplazar la tarea actual por otra de mayor prioridad.

Las tareas creadas con la misma prioridad son tratadas por igual por el planificador, si dos de ellas están listas para funcionar, el planificador comparte el tiempo de ejecución entre ambas tareas, implementando una planificación tipo “Round Robin”, donde el quantum es el tiempo entre cada tick de reloj. En la gestión de planificación se considera:

- Un proceso no debe salir de su función de implementación nunca, es decir, nunca contendrán una instrucción return.
- Cada proceso es considerado como un programa independiente, siendo la primera función el punto de entrada que implemente el proceso.
- En un sistema con un solo procesador, como es el caso de la mayoría de sistemas embebidos, solamente una tarea podrá ejecutarse en un determinado instante de tiempo.

1.4.6.3 Gestión de memoria

El microkernel de freeRTOS deberá asignar memoria principal (RAM) cada vez que se crea una tarea (su contexto de ejecución), una cola o un semáforo. Para gestionar la memoria freeRTOS ofrece 3 esquemas a la hora de decidir cuál escoger.

1.4.6.3.1 Esquema 1

Es el esquema más simple de todos, no permite liberar una zona de memoria una vez se haya asignado.
El algoritmo tan sólo asigna memoria, este esquema:

- Es recomendable si la aplicación no elimina tareas o colas.
- Es determinista ya que siempre tarda el mismo tiempo en devolver un bloque de memoria.

Este esquema es adecuado para una gran cantidad de RTOS que cumplan con la única condición de que todas las tareas y las colas se creen antes de que el kernel sea iniciado.

1.4.6.3.2 Esquema 2

Este esquema utiliza un mejor algoritmo de ajuste y permite liberar bloques de memoria que fueron asignados. Sin embargo, este esquema no ofrece ninguna reordenación de memoria (no se combinan los bloques libres adyacentes en un solo bloque grande), por lo que puede producirse fragmentación en la RAM. Este esquema:

- Es recomendable para aplicaciones que reserven bloques de memoria RAM del mismo tamaño, es decir, tareas con el mismo tamaño de pila y colas de mensajes con el mismo tamaño.
- No es determinista.

Este esquema es adecuado para RTOS pequeños que requieran crear dinámicamente las tareas.

1.4.6.3.3 Esquema 3

Este esquema implementa un simple encapsulado de la biblioteca estándar de C malloc() y free() funciones que, en la mayoría de los casos, se suministra con el compilador elegido. El encapsulado, simplemente hace que el malloc() y free() funcionen. Este esquema:

- Requiere compilar la librería que proporcionar las implementaciones de malloc () y free ().
- No es determinista.
- Probablemente aumentará considerablemente el tamaño del código del kernel.

Este esquema es el menos restrictivo.

1.4.7 ChibiOS/RT

ChibiOS/RT [14] es un compacto y rápido sistema operativo de tiempo real para dispositivos embebidos, desarrollado por Giovanni Di Sirio, tiene un diseño de microkernel, es de código abierto y soporta un amplio conjunto de arquitecturas: Intel 80386, ARM7, ARM9, ARM Cortex-M0, ARM Cortex-M3, ARM Cortex-M4, PPC, e200z, Atmel AVR, TI MSP430, STM8, Freescale Coldfire, Renesas H8S.

ChibiOS/RT está diseñado para aplicaciones embebidas sobre microcontroladores de 8, 16 y 32 bits, el tamaño y la eficiencia en la ejecución son los principales objetivos de ChibiOS/RT. Como referencia, el tamaño del microkernel puede variar desde un mínimo de 1.2KB hasta un máximo de 5,5KB con todos los subsistemas activados. Las principales características de Chibios/RT son:

- Multitarea preferente.
- 128 niveles de prioridad.
- Planificador de ejecución tipo Round Robin.
- Temporizadores.
- Semáforos.
- Mútex.
- Variables condición.
- Mensajes síncronos y asíncronos.
- Banderas de eventos y controladores.
- Colas.
- Síncrona y asíncrona de E/S con capacidad de tiempo de espera.
- Capa de abstracción de hardware con soporte para ADC, CAN, GPT, EXT, I2C, UCI, MAC, MMC / SD, PAL, PWM, RTC, SDC, serie, SPI y drivers USB.

Todos los objetos del sistema, tales como hilos, semáforos, temporizadores, etc., se pueden crear y eliminar en tiempo de ejecución. No hay límite superior de excepción de la memoria disponible. Con el fin de aumentar la fiabilidad del sistema, la arquitectura del microkernel es completamente estática, no se requiere un asignador de memoria (pero está disponible como opción), y no existen estructuras de datos con límites superiores de tamaño como tablas o matrices.

ChibiOS/RT soporta múltiples tareas y ejecuta la que está en estado “lista” de acuerdo al nivel de prioridad que se les haya establecido cuando se las creó, ChibiOS/RT también soporta tareas con el mismo nivel de prioridad y si varias de estas tareas están en estado “lista” y con prioridad de ejecución, se realiza la planificación de estas empleando la estrategia tipo “Round Robin”.

1.4.7.1 Componentes

ChibiOS/RT tiene un diseño muy modular, internamente está dividido en varios componentes independientes, que a su vez se dividen en varios subsistemas.

1.4.7.1.1 Capa de Puerto

Este componente es el responsable de la puesta en marcha del sistema, de la abstracción de interrupciones, primitivas de bloqueo/desbloqueo, el contexto y las estructuras relacionadas con el código. Este componente suele contener muy poco código, porque la mayor parte del sistema operativo es muy portátil, pero su calidad puede afectar en gran medida el rendimiento del sistema operativo. Esta es probablemente la parte más crítica de todo el sistema operativo.

1.4.7.1.2 Capa de abstracción del hardware (HAL)

Este componente contiene un conjunto de controladores de dispositivos abstractos que ofrecen una capacidad de comunicación entre los componentes de E/S común a la aplicación a través de todas las plataformas de apoyo. El HAL es totalmente portable a través de las diversas arquitecturas y compiladores. Los controladores se clasifican en varias categorías:

1.4.7.1.2.1 Controladores normales

Se compone de un controlador de alto nivel (HLD) y de un controlador específico de la plataforma de bajo nivel (LLD). Los controladores normales son lo suficientemente generales como para ser portado a varias plataformas con sólo escribir un LLD específico.

1.4.7.1.2.2 Controladores complejos

Esta clase de controlador es totalmente portátil y no tiene ninguna dependencia de hardware. Se basa en otros controladores para sus necesidades de E/S.

1.4.7.1.2.3 Controladores de la plataforma

Esta clase de controlador es específico de una plataforma y no está destinado a ser portátil.

1.4.7.1.3 Capa de Plataforma

Esta capa contiene un conjunto de implementaciones de controladores de dispositivos, que son generalmente dedicadas a toda una familia de productos en lugar de un modelo específico.

1.4.7.1.4 Varios

Se trata de una biblioteca de varias utilidades adicionales que no pertenecen a ningún componente en particular, pero que pueden hacer la vida más fácil, mientras se desarrolla una aplicación.

1.4.7.2 Arquitectura del microkernel

El microkernel en sí es muy modular y está compuesto por varios subsistemas, los cuales en su mayoría son opcionales.

1.4.7.2.1 Base de Servicios del microkernel

Esta categoría incluye los subsistemas del microkernel obligatorios:

- System: cerraduras de bajo nivel, la inicialización.
- Timers: temporizadores virtuales y comunicación entre los componentes de tiempo.
- Scheduler: todo el mecanismo de sincronización de nivel superior se implementan a través de este subsistema, es muy flexible.
- Threads: comunicación entre los hilos relacionados.

1.4.7.2.2 Sincronización

Esta categoría incluye los subsistemas relacionados con la sincronización, los cuales se pueden configurar fuera del microkernel:

- Semaphore: contador binario y semáforos del subsistema.
- Mútex: exclusiones mutuas del subsistema con soporte para el algoritmo de herencia de prioridad.
- Condvars: las variables condición son un mecanismo de sincronización destinado a ser utilizado dentro de una zona protegida por un mútex.
- Events: fuentes de eventos y banderas de eventos.
- Messages: mensajes síncronos ligeros.
- Mailboxes: colas de mensajes asíncronos.

1.4.7.2.3 Gestión de la memoria

Esta categoría incluye los subsistemas de la gestión de la memoria:

- Core Allocator: administrador de memoria de núcleo central. Este subsistema es utilizado por los demás asignadores para obtener fragmentos de memoria de una manera consistente.
- Memory Heaps: administrador de la pila central mediante una estrategia de ajuste, este subsistema también permite la creación de múltiples pilas con el fin de manejar las áreas de memoria no uniforme.
- Memory Pools: colección de buffers en memoria de tamaño fijo de asignación muy rápida.
- Dynamic Threads: por lo general los hilos son objetos estáticos en ChibiOS/RT pero existe la opción para el manejo dinámico de hilos.

1.4.7.2.4 Flujos y canales de E/S

Esta categoría incluye E/S y los subsistemas relacionados con el intercambio de datos:

- Data Streams: interfaz de flujo abstracto.
- I/O Channels: canales abstractos de E/S que hereda de la interfaz de flujo abstracto.
- I/O Queues: genérico, Byte de ancho, capacidad de comunicación entre los componentes de cola de E/S.

1.4.7.2.5 Depuración

Esta categoría incluye los subsistemas relacionados con la depuración:

- Assertions: comprobaciones de integridad en el tiempo.
- Parameter Checks: controles de parámetros en el tiempo.
- Stack Checks: comprobaciones de pila de tiempo de ejecución.
- Trace Buffer: búfer de seguimiento de cambio de contexto.
- Registry: el subsistema de registro puede ser visto como parte de la categoría de depuración, incluso si se emplea en contextos diferentes al de depuración.

CAPÍTULO II

ARDUINO DUE

2.1 Arduino

Arduino es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

Tradicionalmente el hardware se ha basado en una placa con un microcontrolador Atmel AVR y puertos de entrada/salida. Los microcontroladores más usados son el Atmega168, Atmega328, Atmega1280, ATmega8 por su sencillez y bajo coste que permiten el desarrollo de múltiples diseños. El lenguaje de programación usado en el IDE de Arduino es Processing, un dialecto de C/C++.

Desde octubre del 2012, Arduino se usa también con un microcontrolador Cortex-M3 de ARM de 32 bits, que opera a 3.3 voltios, a diferencia de la mayoría de las placas con AVR que usan mayormente 5 voltios. Sin embargo ya anteriormente se lanzaron placas Arduino con Atmel AVR a 3.3 Voltios como la Arduino Fio.

Arduino se puede utilizar para desarrollar sistemas autónomos o puede ser conectado a software de un ordenador. Las placas se pueden montar a mano o adquirirse. El entorno integrado de desarrollo (IDE) es libre y se puede descargar gratuitamente desde el Internet. Al ser Arduino hardware libre, tanto su diseño como su distribución son libres, es decir, puede utilizarse libremente para el desarrollo de cualquier tipo de proyecto sin haber adquirido ninguna licencia.

Las especificaciones de las diferentes versiones de las placas de Arduino se presentan a continuación en la tabla 1 [15]:

Placa	uC	Voltaje Ingreso	Voltaje Sistema	Velocidad Reloj	Digital E/S	Entradas Analógica	PWM	UART	Memoria Flash
Arduino Due	ATSAM3X8E	7-12V	3.3V	84MHz	54	12	12	4	512KB
Arduino Leonardo	ATmega32U4	7-12V	5V	16MHz	20	12	7	1	32KB
Arduino Uno - R3	ATmega328	7-12V	5V	16MHz	14	6	6	1	32KB
RedBoard	ATmega328	7-15V	5V	16MHz	14	6	6	1	32KB
Arduino Uno SMD (retired)	ATmega328	7-12V	5V	16MHz	14	6	6	1	32KB
Arduino Uno (retired)	ATmega328	7-12V	5V	16MHz	14	6	6	1	32KB
Arduino Duemilanove (retired)	ATmega328	7-12V	5V	16MHz	14	6	6	1	32KB
Arduino Bluetooth (retired)	ATmega328	1.2-5.5V	5V	16MHz	14	6	6	1	32KB
Arduino Pro 3.3V/8MHz	ATmega328	3.35 -12V	3.3V	8MHz	14	6	6	1	32KB

Arduino Pro 5V/16MHz	ATmega328	5 - 12V	5V	16MHz	14	6	6	1	32KB
Ethernet Pro (retired)	ATmega328	7-12V	5V	16MHz	14	6	6	1	32KB
Arduino Mega 2560 R3	ATmega2560	7-12V	5V	16MHz	54	16	14	4	256KB
Arduino Mega 2560 (retired)	ATmega2560	7-12V	5V	16MHz	54	16	14	4	256KB
Arduino Mega (retired)	ATmega1280	7-12V	5V	16MHz	54	16	14	4	128KB
Mega Pro 3.3V	ATmega2560	3.3-12V	3.3V	8MHz	54	16	14	4	256KB
Mega Pro 5V	ATmega2560	5-12V	5V	16MHz	54	16	14	4	256KB
Arduino Mini 04 (retired)	ATmega328	7-9V	5V	16MHz	14	6	8	1	32KB
Arduino Mini 05	ATmega328	7-9V	5V	16MHz	14	6	8	1	32KB
Arduino Pro Mini 3.3V/8MHz	ATmega328	3.35 -12V	3.3V	8MHz	14	6	6	1	32KB
Arduino Pro Mini 5V/16MHz	ATmega328	5 - 12V	5V	16MHz	14	6	6	1	32KB
Arduino Fio	ATmega328P	3.35 -12V	3.3V	8MHz	14	8	6	1	32KB
Mega Pro Mini 3.3V	ATmega2560	3.3-12V	3.3V	8MHz	54	16	14	4	256KB
Pro Micro 5V/16MHz	ATmega32U4	5 - 12V	5V	16MHz	12	4	5	1	32KB
Pro Micro 3.3V/8MHz	ATmega32U4	3.35 - 12V	3.3V	8MHz	12	4	5	1	32KB
LilyPad Arduino 328 Main Board	ATmega328	2.7-5.5V	3.3V	8MHz	14	6	6	1	32KB
LilyPad Arduino Simple Board	ATmega328	2.7-5.5V	3.3V	8MHz	9	4	5	0	32KB

Tabla 1: Especificaciones de las diferentes versiones de las placas de Arduino

2.2 Entorno integrado de desarrollo

El entorno integrado de desarrollo (IDE) de Arduino es de código abierto, se puede descargar gratuitamente desde el Internet siendo compatible con Windows, Mac OS X y Linux, contiene un editor de texto para escribir programas, un área de mensajes, una consola de texto, una barra de herramientas con botones con las funciones comunes, y una serie de menús. Se conecta con la placa de Arduino para cargar los programas y comunicarse con ella. La figura 3 muestra la pantalla inicial de la versión 1.5.6-r2 del IDE de Arduino.

Figura 3: Pantalla inicial de la versión 1.5.6-r2 del IDE de Arduino

2.2.1 Sketches

Los programas escritos usando el IDE de Arduino se llaman sketches, los cuales se escriben en el editor de texto y se guardan con la extensión de archivo ino. Para escribir sketches el IDE de Arduino proporciona características para cortar/copiar/pegar y buscar/reemplazar texto, se dispone de un área de mensajes que proporciona información al guardar y exportar además de los errores, además, existe una consola que muestra la salida de texto por el entorno Arduino incluyendo mensajes de error completos y demás información, en la esquina inferior derecha de la ventana del IDE se muestra la placa y el puerto serial actual, también existen los botones de la barra de herramientas que permiten comprobar, cargar, crear, abrir y guardar el sketch (programa) y abrir un monitor serial. A continuación se muestran estos botones de la barra de herramientas y su significado:

Verificar: comprueba si el código del programa (sketch) tiene errores.

Subir: compila el código del programa (sketch) y lo sube a la placa Arduino.

Nuevo: permite crear un nuevo sketch.

Abrir: permite abrir un sketch guardado anteriormente.

Salvar: permite guarda el sketch de la ventana actual.

Monitor serie: permite abrir en una ventana un monitor del puerto serie actual.

El IDE de Arduino proporciona otras opciones adicionales que facilitan la escritura de los sketches, las mismas que se encuentran dentro de los cinco menús: Archivo, Editar, Programa, Herramientas y Ayuda. Estas opciones son sensibles al contexto, lo que significa que sólo los elementos pertinentes al trabajo que se está llevando a cabo están disponibles.

2.2.2 Librerías

Las librerías proporcionan una funcionalidad adicional para ser usadas en los sketches, como por ejemplo, trabajar con un hardware o la manipulación de los datos. Es importante considerar que la librería que se carga en la placa Arduino conjuntamente con el sketch es una librería estática, por lo que se aumenta la cantidad de espacio que se ocupa con cada librería que se incluye, por lo que si no se necesita una librería, esta no debe ser incluida en el sketch. Algunas librerías se incluyen con la descarga del IDE de Arduino; otras se pueden descargar de una variedad de fuentes y también el usuario puede crear sus propias librerías e incluirlas en los sketches.

2.2.3 Memoria Flash (espacio de programa)

La memoria Flash (espacio del programa) es donde Arduino almacena el sketch. Un sketch es el nombre que usa Arduino para un programa y es la unidad de código que se sube y ejecuta en la placa Arduino. Esta memoria es no volátil, si Arduino deja de ser alimentado eléctricamente los datos que haya en esta memoria permanecerán.

El tamaño de la memoria Flash de Arduino puede variar dependiendo del microcontrolador, aunque no es muy grande (en el microcontrolador ATSAM3X8E del Arduino Due es de 512 KBytes). Por lo que se deben desarrollar los programas de forma muy optimizada, usando los tipos de variables que menos memoria requieran, en la medida de lo posible.

2.2.4 Memoria SRAM (Static Random Access Memory o memoria estática de acceso aleatorio)

La memoria SRAM (Static Random Access Memory o memoria estática de acceso aleatorio) es de tipo volátil, es el espacio donde los sketches (programas) almacenan y manipulan variables al ejecutarse. La información guardada en esta memoria será eliminada cuando Arduino pierda la alimentación. Esta memoria es de uso exclusivo para el programa en ejecución.

La memoria SRAM de Arduino es muy pequeña (en el microcontrolador ATSAM3X8E del Arduino Due es de 96 KBytes), por lo que se deben optimizar los programas al máximo y no abusar de tipos de datos muy grandes. Si la SRAM se queda sin espacio, el programa de Arduino fallará de forma imprevista, aunque se compile y se suba a Arduino correctamente la aplicación no se ejecutará o se ejecutara de manera extraña.

2.3 Arduino Due

Arduino Due [16] es una placa electrónica de la familia Arduino que se constituye en la primera tarjeta que utiliza el procesador con núcleo ARM de 32 bits Atmel SAM3X8E ARM Cortex-M3 MCU, lo cual mejora las capacidades del estándar de Arduino y añade nuevas características. Está disponible desde el 22 de octubre del 2012.

La tarjeta dispone de 54 entradas/salidas digitales (de las cuales 12 se pueden utilizar como salida PWM, con la posibilidad de elegir la resolución), 12 entradas analógicas con resolución de 12 bits, 4 UARTs (puertos serie de hardware), 2 salida DAC (convertidores de analógico a digital), velocidad del reloj de 84 MHz, 2 conectores USB, un conector de alimentación y un botón de reset (reinicio). La tensión máxima de los pines de Entrada/Salida es de 3.3 voltios. Por lo que no se puede utilizar un voltaje más alto, tales como 5 voltios en un pin de entrada, puesto que puede causar daños a la placa.

Uno de los dos conectores USB, el micro-USB es el nativo y puede funcionar como un host USB. Esto significa que se puede conectar otros dispositivos USB a la tarjeta, tal como ratones, teclados y teléfonos

inteligentes. El otro conector USB, es el programador que está conectado a un ATMEGA16U2, que proporciona un puerto COM virtual con el software en un ordenador conectado y se ha diseñado con fines de comunicación y programación de la placa. La figura 4 muestra la parte frontal de la placa y la figura 5 muestra la parte posterior de la misma.

Figura 4: Parte frontal de la placa Arduino Due

Figura 5: Parte posterior de la placa Arduino Due

2.3.1 Beneficios del núcleo ARM

El Arduino Due [16] tiene un núcleo ARM de 32 bits que supera a las tarjetas con microcontrolador de 8 bits. Las diferencias más significativas son:

- Un núcleo de 32 bits, que permite operaciones con ancho de datos de 4 Bytes en un solo ciclo de reloj.
- La frecuencia de operaciones es de 84Mhz lo que permite mayor capacidad de procesamiento.
- 96 KBytes de SRAM que facilita el desarrollo de aplicaciones más complejas.
- 512 KBytes de memoria flash para código de programas.
- Un controlador de DMA (acceso directo a memoria), que reduce el uso de CPU en operaciones intensivas de IO.

2.3.2 Características

El Arduino Due [16] tiene las siguientes características:

- Microcontrolador: ATSAM3X8E
- Voltaje operativo: 3.3V

- Voltaje de alimentación (recomendado): 7-12V
- Voltaje de alimentación (límite): 6-16V
- Entradas/Salidas digitales: 54 (12 de las cuales se pueden utilizar como salida PWM)
- Entradas analógicas: 12
- Salidas analógicas: 2 (DAC)
- Corriente (DC) máxima para todas las líneas de entradas y salidas: 130 mA
- Corriente (DC) para los pines 3.3 Voltios: 800 mA
- Corriente (DC) para los pines 5 Voltios: 800 mA
- SRAM: 96KB (dos bancos: 64KB + 32KB)
- Memoria flash: 512KB toda habilitada para las aplicaciones de ususario.
- Velocidad del reloj: 84 MHz

2.3.3 Diferencias del Arduino Due con otros modelos de Arduino

El poseer un procesador ARM de 32 bit a 84 Mhz, permite realizar cosas mucho más rápidamente, tener un procesador más rápido, y el uso del DMA, puede aumentar la estabilidad, capacidad de respuesta y la precisión de las aplicaciones sin mermar la capacidad de proceso.

El Arduino Due, ofrecer dos puertos USB, uno para programación y comunicación, y otro que actúa como cliente o Host que puede ser utilizado para conectar un ratón, teclado, etc. Posee entradas y salidas analógicas de 12-bit por lo que la tasa teórica de muestreo se ha visto multiplicada hasta en unos 1000ksps (kilomuestras por segundo) en comparación con Arduino uno, leonardo y mega 2560 que poseían una tasa de muestreo de 15ksps.

Arduino Due, es también el primer Arduino en traer integrado un conversor digital a analógico, dos de hecho. Una librería de audio se está desarrollando para añadir al Arduino Due la posibilidad de reproducir archivos WAV.

Arduino Due, continúa siendo compatible con la mayoría de los shields de Arduino (placas que pueden ser conectadas encima de la placa Arduino extendiendo sus capacidades) al seguir con la misma estructura. Ahora bien, Arduino Due opera a 3.3V, mientras que los Arduinos basados en AVR operan a 5V, así que ciertos shields o de terceros pueden no ser compatibles con las características del Due dependiendo de sus voltajes. Eso significa que si va a utilizar Arduino Due en alguna aplicación ya existente, se deberá asegurar y ajustar bien los voltajes para evitar el riesgo de dañar la placa. Los sketches escritos para las placas de Arduinos basadas en AVR, funcionan perfectamente con Due.

De igual manera que en Arduino uno, el Due utiliza un bootloader previamente grabado en el momento de su fabricación en conjunto con un chip ATMEGA16U que se encarga de la comunicación USB a serial.

2.3.4 Alimentación

El Arduino Due [16] puede ser alimentado a través del conector USB o con una fuente de alimentación externa que puede venir de un adaptador de AC-DC o de una batería. El adaptador se puede enchufar al conector de alimentación de la placa o los cables de la batería se pueden insertar en el pin GND y VIN del conector de alimentación.

La placa Due puede funcionar con un suministro externo de 6 a 16 voltios. Si se proporcionan menos de 7V, sin embargo, el pin de 5V puede suministrar menor voltaje y la placa puede ser inestable. Si se utiliza más de 12V, el regulador de voltaje se puede sobrecalentar y dañar la placa. El rango recomendado es de 7 a 12 voltios. Los pines de alimentación son los siguientes:

- VIN: Es el voltaje de entrada a la placa Arduino cuando se trata de utilizar una fuente de alimentación externa (en lugar de los 5 voltios de la conexión USB u otra fuente de alimentación regulada). Se puede suministrar voltaje a través de este pin.
- 5V: Este pin emite 5 voltios regulados desde el regulador en la placa. La placa puede ser alimentada ya sea desde la toma de la corriente continua (7V – 12V), el conector USB (5V) o el pin VIN de la placa (7V-12V). El suministro de voltaje a través de los pins de 5V o 3.3V no pasa por el regulador, y puede dañar la placa.
- 3.3V: Un suministro de 3.3 voltios generados por el regulador de la placa. El drenaje actual máximo es de 800 mA. Este regulador también proporciona la fuente de alimentación para el microcontrolador SAM3X.
- GND: Pines de tierra.
- IOREF: Este pin proporciona la referencia de voltaje con la que opera el microcontrolador. Un shield configurado apropiadamente puede leer el voltaje del pin IOREF y seleccionar la fuente de alimentación adecuada o habilitar convertidores de voltaje en las salidas para trabajar con los 5V o 3.3V.

2.3.5 Memoria

En el Arduino Due [16] el SAM3X tiene 512KB de memoria en dos bloques de 256KB para almacenar código. El bootloader viene precargado de ATMEL y está almacenado en la memoria dedicada ROM. La SRAM disponible es de 96KB en dos bloques continuos de 64KB y 32KB. Toda la memoria disponible (Flash, RAM y ROM) puede accederse directamente como una dirección de memoria plana.

Es posible borrar la memoria Flash de SAM3X con el botón integrado en la placa “erase”. Este nos permitirá eliminar el Skecth cargado. Para eliminarlo, deberemos mantener pulsado el botón “erase” durante unos segundos.

2.3.6 Entradas y salidas

En cuanto a los pines de entradas y salidas del Arduino Due [16] se tiene:

- E/S digitales (pines del 0 al 53): Cada uno de los pines de Arduino Due pueden ser usados como entradas o salidas, ellas trabajan a 3.3V. Cada pin puede suministrar una corriente de 3 mA o 15 mA dependiendo del pin, o recibir de 6 mA o 9 mA, dependiendo del pin. Estos pines también poseen una resistencia de Pull Down (desactivada por defecto) de 100 KOhm. Además, algunos de estos pines tienen funciones específicas.

  - Serial: 0 (RX) y 1 (TX)
  - Serial 1: 19 (RX) y 18 (TX)
  - Serial 2: 17 (RX) y 16 (TX)
  - Serial 3: 15 (RX) y 14 (TX)

  Usados para recibir (RX) y transmitir (TX) datos serie TTL (con niveles de 3.3 V). Los pines 0 y 1 están unidos con los correspondientes pines USB-TTL serie del chip ATmega16U2.
- PWM (pines del 2 al 13): Proporciona PWM de 8 bit de resolución.
- Conector SPI conector ICSP en otras placas Arduino): Estos pines soportan la comunicación SPI (Serial Peripheral Interface) usando la librería SPI. Los pines SPI están situados en el conector central de seis pines en el centro de la placa lo cual es físicamente compatible con Arduino uno, leonardo y mega2560. EL conector SPI puede usarse para comunicarse solo con otros dispositivos SPI, no para programar el SAM3X con la técnica de programación de circuito en serie.
- CAN (CANRX y CANTX): Estos pines soportan el protocolo de comunicación CAN (Controller Area Network).

- L (LED 13): Este es un LED conectado al Pin 13. Cuando el pin está en HIGH, el LED se enciende, cuando el pin está en LOW, el LED se apaga, el pin 13 también es una salida PWM, por lo se puede variar su intensidad.
- TWI 1: 20 (SDA) y 21 (SCL)
- TWI 2: SDA1 y SCL1.
- Entradas analógicas (pines de A0 a A11): El Due trae 12 entradas analógicas, cada una de las cuales proporcionan una resolución de 12 bit (4096 valores diferentes). Por defecto, la resolución de la lectura está establecida a 10 bit para que sea compatible con las aplicaciones diseñadas para otras placas Arduino, aunque es posible cambiar esta resolución. Las entradas analógicas del Due, miden desde tierra hasta un valor máximo de 3.3v, si se aplica más de 3.3v se puede dañar el chip SAM3X.
- DAC1 y DAC2: Estos pines proporcionan una salida analógica con una resolución de 12 bit (4096 niveles) y pueden usarse para crear una salida de audio.
- AREF: Es el voltaje de referencia para las entradas analógicas.
- Reset: Colocar esta línea LOW para resetear al microcontrolador. Normalmente se la utiliza para agregar un botón de reseteo para shields que bloquean la que está en la placa.

2.3.7 Comunicación

El Arduino Due [16] tiene muchas facilidades para su comunicación con un ordenador, otro Arduino u otros microcontroladores, y con diferentes dispositivos, como teléfonos, tabletas, cámaras, etc. El SAM3X ofrece un hardware UART y tres USARTs para TTL (3.3V) para la comunicación serial.

El puerto USB de programación está conectado a un ATMEGA16U2, que proporciona un puerto COM virtual con el software de un ordenador conectado. El 16U2 también está conectado al hardware UART del SAM3X. Los pines serie RX0 y TX0 proporcionan comunicación serie a USB para la programación de la placa a través del microcontrolador ATMEGA16U2. El software de Arduino incluye un monitor serie que permite mostrar los datos que son enviados hacia y desde la placa. Los LEDs RX y TX de la placa parpadearán cuando los datos se transmiten a través del chip ATMEGA16U2 y la conexión USB al ordenador.

El puerto USB nativo está conectado al SAM3X y permite la comunicación serie (CDC) a través de USB. Este proporciona una conexión serie con el monitor de serie u otras aplicaciones del ordenador. También puede actuar como un host USB para periféricos conectados, tales como ratones, teclados y teléfonos inteligentes.

2.3.8 Programación

El Arduino Due [16] se puede programar con el IDE que proporciona Arduino y que se puede descargar gratuitamente desde Internet. El procedimiento para cargar los sketches en el SAM3X es diferente al seguido con los microcontroladores AVR que se encuentran en otras placas de Arduino, porque la memoria flash debe ser borrada antes de ser re-programada. El proceso de carga en el chip es administrado por la ROM en el SAM3X, que se ejecuta sólo cuando la memoria flash del chip está vacía.

Cualquiera de los 2 puertos USB que proporciona el Arduino Due se puede utilizar para la programación de la placa, aunque se recomienda utilizar el puerto de programación debido a la forma en que se maneja el borrado del chip:

- Puerto de programación: Para utilizar este puerto, hay que seleccionar "Arduino Due (Puerto de Programación)" como placa de trabajo en el IDE de Arduino, el puerto de programación del Due es el más cercano a la toma de alimentación de la placa. Este puerto utiliza el 16U2 como un chip de USB a serie conectado a la primera UART del SAM3X (RX0 y TX0). El 16U2 tiene dos pines conectados a los pines Reset y Erase del SAM3X. Abrir y cerrar el puerto de programación conectado a 1200bps desencadena un procedimiento de "borrado duro" del chip SAM3X, activando los pines de Erase y

Reset en el SAM3X antes de comunicarse con la UART. Este es el puerto recomendado para la programación del Due ya que es más confiable que el "borrado suave" que proporciona el puerto nativo [16].

- Puerto nativo: Para utilizar este puerto, hay que seleccionar "Arduino Due (Puerto USB nativo)" como placa de trabajo en el IDE de Arduino. El puerto USB nativo está conectado directamente al SAM3X, este puerto en el Due es el más cercano al botón Reset de la placa. Abrir y cerrar el puerto nativo a 1200bps desencadena un procedimiento de "borrado suave" que hace que la memoria flash se borre y la placa se reanude con el gestor de arranque. Abrir y cerrar el puerto nativo a una velocidad de transmisión diferente, no se reseteará el SAM3X [16].

2.3.9 Protección de sobrecarga USB

El Arduino Due [16] tiene un fusible reseteable que protege a los puertos USB del ordenador de cortocircuitos y sobretensiones. Aunque la mayoría de los ordenadores proporcionan su propia protección interna, este fusible proporciona una capa adicional de protección. Si se aplica más de 500 mA al puerto USB, el fusible automáticamente rompería la conexión hasta que se elimine el cortocircuito o la sobrecarga.

2.3.10 Características físicas y compatibilidad con shields

La longitud máxima y la anchura del Arduino Due [16] es de 4 y 2,1 pulgadas respectivamente y existen tres orificios para los tornillos para que la placa pueda fijarse a una superficie.

El Arduino Due está diseñado para ser compatible con la mayoría de los shields diseñados para el Uno. Los pines digitales del 0 al 13 (y los adyacentes pines AREF y GND), las entradas analógicas del 0 al 5, el puerto de alimentación, y los "ICSP" (SPI) están todos en localizaciones equivalentes. Además, la UART principal (puerto serie) se encuentra en los mismos pines (0 y 1).

2.4 Microcontrolador Atmel SAM3X8E ARM Cortex-M3

2.4.1 Microcontrolador

Un microcontrolador es un circuito integrado programable, que ejecuta las órdenes almacenadas en su memoria. Un microcontrolador incluye normalmente en su interior los siguientes componentes, que cumplen tareas específicas: Procesador o CPU (Unidad Central de Proceso), memoria RAM para contener los datos, memoria para el programa tipo ROM/PROM/EPROM, líneas de E/S para comunicarse con el exterior, diversos módulos para el control de periféricos (temporizadores, puertas serie y paralelo, CAD: Conversores Analógico/Digital, CDA: Conversores Digital/Analógico, etc.) y un generador de impulsos de reloj que sincronizan el funcionamiento de todo el sistema.

Un microcontrolador es en definitiva un circuito integrado que incluye todos los componentes de un procesador y debido a su reducido tamaño es posible montarlo en el propio dispositivo al que gobierna. Aunque por supuesto sus prestaciones son limitadas si las comparamos con las de cualquier ordenador personal, además de dicha integración, su característica principal es su alto nivel de especialización.

Cuando es fabricado [17], el microcontrolador no contiene datos en la memoria ROM. Para que pueda controlar algún proceso es necesario generar o crear y luego grabar en la EEPROM o equivalente del microcontrolador algún programa, el cual puede ser escrito en lenguaje ensamblador u otro lenguaje para microcontroladores.

Los microcontroladores son diseñados para reducir el costo económico y el consumo de energía de un sistema en particular. Por eso el tamaño de la unidad central de procesamiento, la cantidad de memoria y los periféricos incluidos dependerán de la aplicación. Pueden encontrarse en casi cualquier dispositivo electrónico como automóviles, lavadoras, hornos microondas, teléfonos, etc.

Los productos que para su regulación incorporan un microcontrolador disponen de las siguientes ventajas: aumento de prestaciones (un mayor control sobre un determinado elemento representa una mejora considerable en el mismo), aumento de la fiabilidad (al reemplazar el microcontrolador a un elevado número de elementos disminuye el riesgo de averías y se precisan menos ajustes), reducción del tamaño en el producto acabado (la integración del microcontrolador en un chip disminuye el volumen y la mano de obra), mayor flexibilidad (las características de control están programadas por lo que su modificación sólo necesita cambios en el programa de instrucciones).

2.4.2 Atmel Corporation

Atmel Corporation es un fabricante de semiconductores fundado en 1984 con sede en San José-California-EEUU que se centra en el desarrollo de memorias EEPROM y Flash, microcontroladores (AVR y ARM), productos de radiofrecuencia, sensores y controladores táctiles y aplicaciones para dispositivos específicos. Atmel suministra productos de comunicaciones, redes de computadoras, industriales, automotriz, aplicaciones aeroespaciales, y para los sectores militares. Durante la última década, Atmel ha comenzado a centrarse cada vez más en su línea de productos de microcontroladores especialmente para sistemas embebidos, llegando a alcanzar un liderazgo en este mercado.

Atmel [18] fue fundada en 1984 por George Perlegos quien notó el potencial de mercado de los chips de memoria no volátil, mientras trabajaba para Intel durante los años 1970 y 1980. Perlegos adquirió una valiosa experiencia en la creación de una empresa como el co-fundador de Seeq Technology en 1981, que también era una empresa de fabricación de chips de memoria. La estancia de Perlegos en Seeq Technology fue corta, pero le permitió sentar las bases para Atmel.

2.4.3 ARM

ARM describe una familia de procesadores basados en RISC (Reduced Instruction Set Computer=Ordenador con Conjunto de Instrucciones Reducidas) de 32 bits desarrollada por ARM Holdings. Fue concebida originalmente en la década de los ochentas por Acorn Computers para su uso en ordenadores personales, y posteriormente se dividió como una empresa independiente llamada ARM Holdings. El nombre de ARM fue originalmente un acrónimo de Acorn RISC Machine y, posteriormente, después de la división el nombre pasó a ser acrónimo de Advanced RISC Machine. Los primeros productos basados en ARM eran los Acorn Archimedes, lanzados en 1987. Las familias de procesadores ARM desarrollados por ARM Holdings incluyen el ARM7, ARM9, ARM11 y Cortex.

El uso de un enfoque basado en el diseño RISC, hacen que los procesadores ARM requieran significativamente menos transistores que los procesadores que normalmente se encuentran en un ordenador tradicional. Los beneficios de este enfoque son los costos, el calor y el reducido uso de energía en comparación con diseños de chips más complejos, rasgos que son deseables para dispositivos ligeros, portátiles, baterías, como los teléfonos inteligentes y Tablet PC.

La complejidad reducida y el diseño más simple de ARM la hace ideal para aplicaciones de baja potencia. Como resultado, se han convertido en dominante en el mercado de la electrónica móvil e integrada, incorporados en microprocesadores y microcontroladores pequeños, de bajo consumo y relativamente bajo coste. Se utilizan ampliamente en la electrónica de consumo, incluyendo PDA, tabletas, Teléfono inteligente, teléfonos móviles, videoconsolas portátiles, calculadoras, reproductores digitales de música y

medios (fotos, vídeos, etc.), y periféricos de ordenador como discos duros y routers. Hoy en día, cerca del 75% de los procesadores de 32 bits poseen el chip ARM en su núcleo.

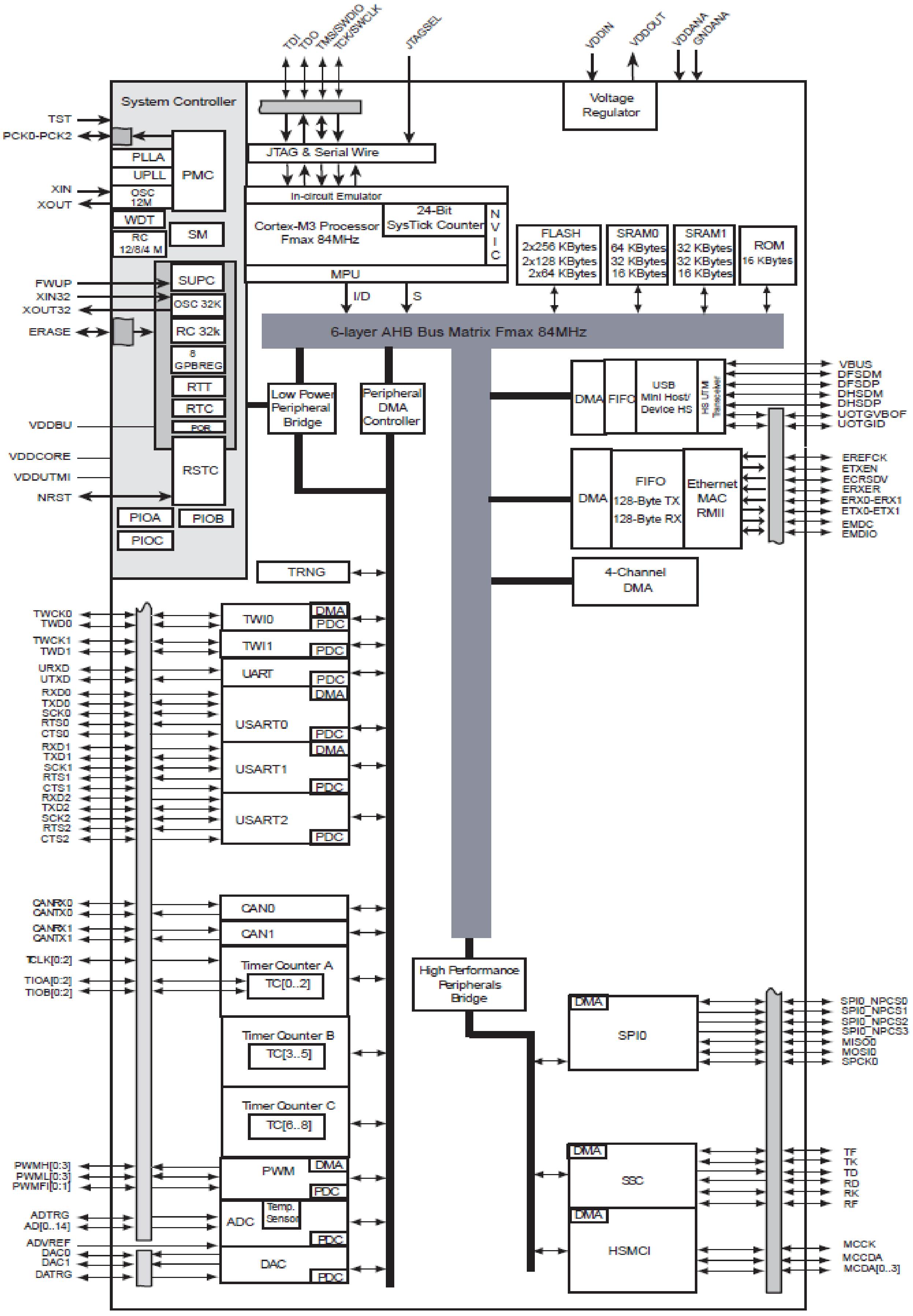

Figura 6: Diagrama de bloques del ATSAM3X8E

2.4.4 ATSAM3X8E

El ATSAM3X8E [19] es un miembro de la familia de microcontroladores flash de Atmel basado en el procesador ARM Cortex-M3 de tecnología RISC, el ATSAM3X8E funciona a 84MHz y se caracteriza por 512KB de flash en 2 x 256KB bancos y 96KB de SRAM en 64KB + 32KB bancos. Su conjunto de periféricos altamente integrado para la conectividad y la comunicación incluye Ethernet, dual CAN, USB MiniHost de alta velocidad, SD / SDIO / MMC y múltiples USART, SPI, TIWS.

El ATSAM3X8E también cuenta con ADC / DAC de 12 bits, sensor de temperatura, temporizadores de 32 bits, temporizador PWM y RTC. La interfaz de bus externo de 16 bits soporta SRAM, PSRAM, NOR y NAND flash con código de corrección de error. La librería Atmel QTouch está disponible para el SAM3X8E para una fácil implementación de los botones, deslizadores y ruedas.

La arquitectura del ATSAM3X8E está diseñada específicamente para soportar las transferencias de datos de alta velocidad, incluye una bus multicapa, así como múltiples bancos SRAM, PDC y canales DMA que permiten que se ejecuten tareas en paralelo y maximizar el rendimiento de datos El dispositivo funciona de 1.62V a 3.6V y sus principales características son:

- Flash (KBytes): 512
- Número de pines: 144
- Max. Frecuencia de operación: 84 MHz
- CPU: Cortex-M3
- N º de Canales táctiles: 32
- Max pines E/S: 103
- Ext. Interrupciones: 103
- Cuadratura Canales Decodificador: 2
- USB Transceptor: 1
- Velocidad USB: Hi-Speed
- Interfaz USB: Host de dispositivo
- SPI: 4
- TWI (I2C): 2
- UART: 5
- CAN: 2
- LIN: 3
- SSC: 1
- Ethernet: 1
- SD/eMMC: 1
- Canales de ADC: 16
- ADC Resolución (bits): 12
- ADC Velocidad (ksps): 1000
- Canales DAC: 2
- Resolución DAC (bits): 12
- Sensor de temperatura: Sí
- SRAM (KBytes): 96
- Interfaz de bus externo: 1
- NAND Interfaz: Sí
- Rango de temperatura (° C): -40 A 85
- Voltaje de funcionamiento (Vcc): 1,62-3,6
- Temporizadores: 9
- Canales de salida: 9
- Canales de entrada: 6

- Canales PWM: 8
- 32kHz RTC: Sí
- Calibrado oscilador RC: Sí

El diagrama de bloques del ATSAM3X8E se muestra en la figura 6 [20]:

CAPÍTULO III

COMPARATIVAS DE RENDIMIENTO RHEALSTONE

3.1 Introducción

El presente trabajo emplea el Arduino Due, como plataforma hardware común sobre la que se evalúan (mediante las métricas de evaluación comparativa Rhealstone) los RTOS libres y de código abierto FreeRTOS y ChibiOS/RT, ambos con soporte para el microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3 del Due.

3.2 Métricas Rhealstone

Las métricas Rhealstone fueron propuestas por Rabindra P. Kar y Kent Porter [21] en 1989 y permiten obtener valores cuantitativos de medidas que influyen crucialmente en el comportamiento de los sistemas operativos de tiempo real. Estas métricas ayudan principalmente a los desarrolladores de sistemas embebidos, que desean emplear un RTOS, a seleccionar el más apropiado para una aplicación específica. Los factores a medir en un RTOS, según esta métrica incluyen:

- Tiempo de conmutación de tarea (Task-Switching Time)
- Tiempo de expulsión (Preemption Time)
- Tiempo de espera de un semáforo (Semaphore Shuffle Time)
- Tiempo de ruptura de interbloqueo (Deadlock Breaking Time)
- Latencia de paso de mensaje entre tareas (Intertask Messaging Latency)

3.2.1 Tiempo de conmutación de tarea (Task-Switching Time)

El tiempo de conmutación de tareas es el tiempo promedio que el sistema necesita para cambiar entre dos tareas independientes y activas, no suspendida o durmiendo, de igual prioridad. Este tiempo está influenciado por la eficiencia del microcontrolador en guardar y recuperar su conjunto de registros y por su arquitectura y conjunto de instrucciones [22]. En la figura 7 se muestra lo antes mencionado, donde el tiempo de conmutación de tareas es equivalente a t1, luego a t2, etc., por lo que finalmente constituye el promedio de estos tiempos t.

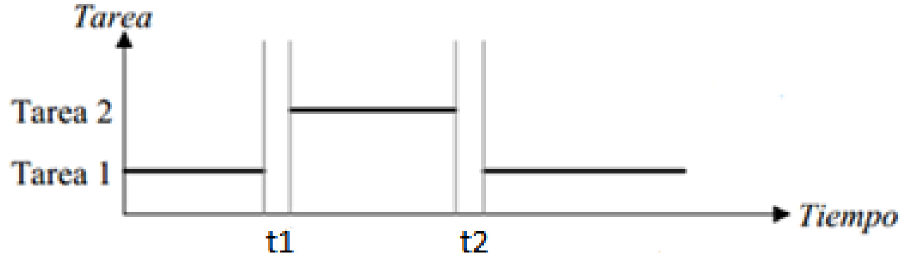

Figura 7: Métrica Rhealstone - tiempo de conmutación de tarea

3.2.2 Tiempo de expulsión (Preemption Time)

El tiempo de expulsión es el promedio de tiempo necesario para que una tarea de mayor prioridad tome el control del sistema a partir de una tarea en ejecución de menor prioridad. Este tiempo está influenciado

por la capacidad del sistema de reconocer la activación de una tarea de mayor prioridad que la actual [22]. La figura 8 representa el tiempo de expulsión, dado por t.

Figura 8: Métrica Rhealstone - tiempo de expulsión

3.2.3 Tiempo de espera de un semáforo (Semaphore Shuffle Time)

El tiempo de espera de un semáforo es el tiempo desde que una tarea que posee un semáforo libera el mismo y otra tarea que esperaba en el semáforo lo toma [22]. Este tiempo representa la sobrecarga de tiempo asociada con la exclusión mutua, que ocurre cuando múltiples tareas compiten por el mismo recurso. La figura 9 representa lo antes indicado, donde el tiempo de espera del semáforo está dado por t_{SS}.

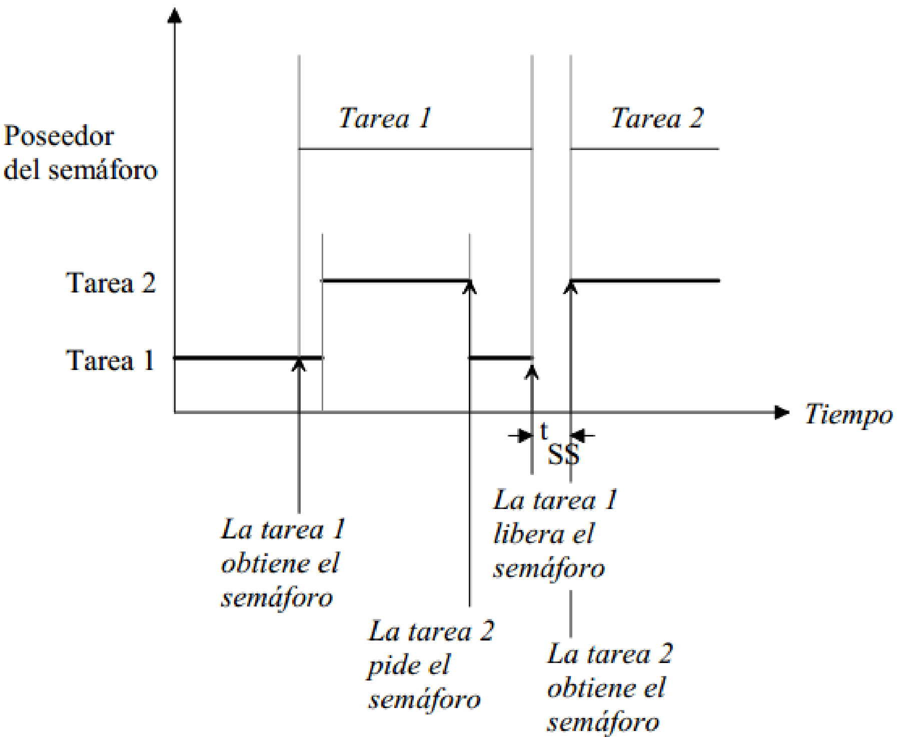

Figura 9: Métrica Rhealstone - tiempo de espera de un semáforo

3.2.4 Tiempo de ruptura de interbloqueo (Deadlock Breaking Time)

La ruptura de interbloqueo se produce cuando una tarea de mayor prioridad se antepone a una tarea de menor prioridad, que tiene un recurso que necesita la tarea de mayor prioridad, y existe una tarea de prioridad media que se activa y no permite el paso directo entre la tarea de mayor y menor prioridad para

liberar el recurso. Esta métrica mide el tiempo promedio que tarda el sistema en resolver el mencionado conflicto. La figura 10 representa lo mencionado anteriormente, donde el tiempo de ruptura de interbloqueo equivale de los tiempos $t_A + t_B$.

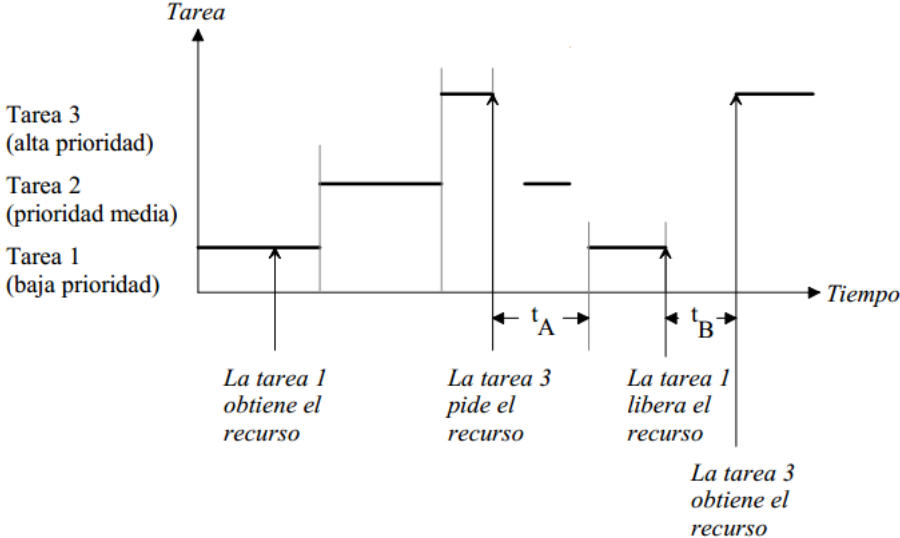

Figura 10: Métrica Rhealstone - tiempo de ruptura de interbloqueo

3.2.5 Latencia de paso de mensaje entre tareas (Intertask Messaging Latency)

La latencia de paso de mensajes entre tareas es el retraso en el sistema cuando se envía un mensaje de datos de longitud distinta de cero desde una tarea a otra que recibe el mensaje [23]. Para medir esto correctamente, la tarea emisora debe detener su ejecución inmediatamente después de enviar el mensaje y la tarea receptora debe estar suspendida a la espera del mensaje. La figura 11 representa lo antes indicado, donde la latencia de paso de mensaje entre tareas viene dado por t. Para la representación de la figura 11 se considera que tras la emisión, la tarea emisora se suspende y que la receptora está suspendida a la espera de la llegada del mensaje; ya que de lo contrario sería difícil hacer una representación debido a que el mensaje por una tarea y su recepción por otra, puede ocurrir en cualquier momento.

Figura 11: Métrica Rhealstone - latencia de paso de mensaje entre tareas

3.3 Valor único de rendimiento Rhealstone

Las diferentes métricas Rhealstone [24] generan tiempos de rendimientos de un RTOS importantes por sí mismos, sin embargo, en ocasiones es útil combinarlos en un sólo valor para poder hacer comparaciones generales con otros RTOS. El procedimiento de obtención de este valor único de rendimiento Rhealstone (VURR), es el siguiente:

- El resultado del tiempo que registraron cada una de las métricas Rhealstone deben ser expresados en segundos.
- Se calcula el promedio de los tiempos expresados en segundo.
- Finalmente, este promedio calculado se invierte para obtener el valor único de rendimiento Rhealstone expresado en Rhealstones/segundo.

En consecuencia, si se reemplaza con la variable T acompañado de un índice respectivo, a cada una de las métricas Rhealstone expresadas en segundos, se tendría que el tiempo de conmutación de tarea sería T1, el tiempo de expulsión sería T2, el tiempo de espera de un semáforo sería T3, el tiempo de ruptura de interbloqueo sería T4 y la latencia de paso de mensaje entre tareas sería T5. Por tanto, la fórmula del valor único de rendimiento Rhealstone expresado en Rhealstones/segundo, sería:

Valor único de rendimiento Rhealstone (VURR) = $\left(\frac{T1+T2+T3+T4+T5}{5}\right)^{-1}$ Rhealstones/segundo

CAPÍTULO IV

APLICACIÓN DE LAS COMPARATIVAS RHEALSTONE SOBRE LOS SISTEMAS OPERATIVOS DE TIEMPO REAL FREERTOS Y CHIBIOS/RT BAJO LA PLATAFORMA HARDWARE DEL ARDUINO DUE

4.1 Introducción

La evaluación comparativa (benchmark) para los sistemas operativos de tiempo real FreeRTOS y ChibiOS/RT que se presentan en este documento, se basa en un trabajo realizado por Rabindra P. Kar [24] que es la aplicación de las Rhealstone Benchmarking (métricas que permiten comparar las bondades de un RTOS) hecho para el sistema operativo de tiempo real iRMX (un RTOS diseñado específicamente para su uso con la familia de procesadores Intel 8080 e Intel 8086) el cual se ha trasladado a las instrucciones equivalentes en FreeRTOS y ChibiOS/RT, así como a las características de trabajo del Arduino Due.

Para la medición del tiempo en las comparativas Rhealstone se empleó la instrucción micros() que en Arduino mide el tiempo en microsegundos desde que la placa del Arduino empieza a ejecutar el programa cargado, en todos los aspectos necesarios se toman los tiempos y se opera con ellos con el fin de lograr medir la comparativa que se desea conocer.

Para el presente trabajo se empleó, en las diferentes comparativas Rhealstone, el mismo número de iteraciones que se proponen en los códigos fuentes del trabajo realizado en [24] aunque los resultados se probaron y son estables en todas las comparativas para un número de iteraciones diferentes a las planteadas.

Es importante recalcar, que para los diferentes códigos fuentes de las comparativas Rhealstone realizadas en el presente trabajo, se anuló la optimización (opción –O0) al compilar las estructuras presente en los mencionados códigos fuentes. Ya que la optimización por defecto (opción –Os) intervenía en los bucles por lo que se veían afectados los tiempos debido a que las comparativas se basan en iteraciones realizadas por bucles. Asimismo, esto no permitía calcular el número de iteraciones necesarias para que un bucle FOR incremente una variable contador con paso 1 en el tiempo que dura un tick de reloj (1 milisegundo), el código fuente de este cálculo se muestra en el apéndice A. Este tiempo con un ligero incremento, que para los efectos se lo ha denominado como “tick aumentado”, de acuerdo a [24] es el que permite generar un proceso controlado en tiempo de un poco más de lo que dura un tick de reloj (1 milisegundo) y esto es fundamental para que se produzcan las comparativas de tiempo de expulsión y de tiempo de ruptura de interbloqueo.

Para la compilación de los códigos fuentes de las comparativas benchmark realizadas, se empleó el entorno integrado de desarrollo (IDE) proporcionado por Arduino para el sistema operativo Windows, aplicación que para anular la optimización, antes indicada, se tuvo que configurar haciendo cambios en dos líneas contenidas en el archivo “platform.txt” que se encuentra en la ruta de la instalación del IDE de Arduino que para este estudio fue “C:\Program Files\Arduino\hardware\arduino\sam”. Las líneas de código contenidas en el archivo “platform.txt” que se modificaron fueron:

- “compiler.c.elf.flags=**-Os** -Wl,--gc-sections” se modificó por “compiler.c.elf.flags=**-O0** -Wl,--gc-sections”
- “compiler.cpp.flags=-c -g **-Os** -w -ffunction-sections -fdata-sections -nostdlib --param max-inline-insns-single=500 -fno-rtti -fno-exceptions -Dprintf=iprintf” se modificó por “compiler.cpp.flags=-c -g **-O0** -w -ffunction-sections -fdata-sections -nostdlib --param max-inline-insns-single=500 -fno-rtti -fno-exceptions -Dprintf=iprintf”

Para la realización de estas comparativas Rhealstone se emplearon la versión 1.5.6-r2 (32 bits) para el sistema operativo Windows del entorno integrado de desarrollo (IDE) proporcionado por Arduino y en cuanto a los RTOS se utilizaron la adaptación de la versión 2.6.5 de ChibiOS/RT al Arduino Due realizada por Bill Greiman; dicha versión fue publicada el 11 de agosto de 2014 y puede descargarse desde [25] y la adaptación de la versión 8.0.1 de FreeRTOS al Arduino Due realizada también por Bill Greiman, la misma que fue publicada del 14 de agosto del 2014 y puede descargarse desde [26]. Para que puedan ejecutarse, ambos RTOS deben integrarse a la carpeta de librerías del IDE de Arduino. Los mencionados programas se pueden descargar libremente en Internet.

Para ambos sistemas operativos de tiempo real FreeRTOS y ChibiOS/RT se determinó en las librerías de configuración correspondientes (FreeRTOS: FreeRTOSConfig.h -- ChibiOS/RT: chconf.h) que la frecuencia de un tick de reloj sea de 1000 Hz y ya que el método de planificación tipo "Round Robin" de FreRTOS emplea un quantum de un tick de reloj, se tuvo que cambiar el quantum de 20 ticks de reloj que trae originalmente ChibiOS/RT a un tick de reloj (archivo chconf.h -- `CH_TIME_QUANTUM 1`), estas configuraciones fueron necesarias para tener igualdad de condiciones en ambos RTOS para las comparativas.

A lo largo de la siguiente sección se describen cada una de las comparativas Rhealstone realizadas y el código fuente de las mismas, se proporcionan en los apéndices. Teniendo en cuenta los resultados de tiempos obtenidos en las diferentes comparativas Rhealstone, se calculó para cada RTOS el valor único de rendimiento Rhealstone expresado en Rhealstones/segundo. Finalmente, aunque no forma parte de las comparativas Rhealstone se realiza una comparación de la memoria Flash (espacio de programa) que ocupan las comparativas Rhealstone realizadas para los RTOS FreeRTOS y ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due.

4.2 Descripción de las comparativas

A continuación se detalla lo realizado en cada una de las comparativas Rhealstone, las cuales se basan en el trabajo hecho por Rabindra P. Kar [24] y que se han aplicado a los sistemas operativos de tiempo real FreeRTOS y ChibiOS/RT, ejecutándose en el microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3 incluido en el Arduino Due.

4.2.1 Tiempo de conmutación de tarea (Task-Switching Time)

El objetivo de esta comparativa Rhealstone es establecer el tiempo medio que se tarda en pasar de una tarea a otra. Para ello, se establecen dos tareas con la misma prioridad y se mide el tiempo de conmutación durante 500000 iteraciones para obtener un tiempo medio.

Para eliminar un tiempo que afecta al cálculo del tiempo de la conmutación de tareas, se midió y guardó en una variable (t) el tiempo que consume un bucle FOR sin realizar ninguna instrucción para las 500000 iteraciones, para luego restarlas al tiempo total; ya que el mencionado bucle es el que permite las iteraciones y no forma parte del tiempo a medir. Por eso hay que restar del tiempo total el tiempo que consume. Cabe indicar que se repite el mismo proceso dos veces ya que hay dos tareas que son las que conmutan. El código empleado para esta calibración se muestra a continuación:

```
ti=micros();
for(uint32_t x=1;x<=nt;x++){
  ;
}
for(uint32_t y=1;y<=nt;y++){
  ;
```

```
}
t=micros()-ti;
```

Una vez considerado este aspecto, se crean las dos tareas con la misma prioridad cuyo cometido no es más que cambiar de tareas 500000 iteraciones dadas por los bucles FOR. Una vez terminadas, las tareas se eliminan a sí mismas y se da paso al cálculo del tiempo, restándose los tiempos que afectan a lo que se quiere medir (conmutación de tareas). Cabe indicar que todo lo descrito en esta comparativa se aplicó diez veces para ambos RTOS, generando siempre el mismo resultado de tiempo.

Los resultados obtenidos, medidos en microsegundos (µs), son los que se presentan en la tabla 2:

Comparativa	FreeRTOS	ChibiOS/RT
Conmutación(500000)	Resultado (µs)	Resultado (µs)
	4,73	1,95

Tabla 2: Resultado de la aplicación de la comparativa Rhealstone - tiempo de conmutación de tarea

El cálculo de este tiempo de conmutación de tareas se lo utilizará también más adelante para ser restado al tiempo que se obtiene en otras comparativas, que se ven afectadas por la conmutación. Como cada RTOS tiene sus instrucciones y formas de trabajar distintas, se describe a continuación lo realizado específicamente para cada sistema en esta comparativa:

4.2.1.1 FreeRTOS

Se crean las dos tareas con la misma prioridad y una tercera tarea con prioridad más baja, que se activará terminadas las dos tareas. Esta tercera tarea se activa para calcular en ella el tiempo final de la comparativa. Como FreeRTOS permite crear las tareas y estas sólo se activan cuando se emplea la instrucción vTaskStartScheduler() se toma el tiempo inicial en dónde se empiezan a ejecutar las tareas (ti) y luego se lo considera en el cálculo del tiempo final. El código se muestra a continuación:

```
xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,2,&tarea1);
xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
ti=micros();
vTaskStartScheduler();
```

Las tareas de la misma prioridad que generan la conmutación, hacen el cambio de tareas mediante la instrucción taskYIELD() haciendo sólo este proceso 500000 iteraciones. Terminada todas las iteraciones de las dos tareas cada una se elimina a sí misma empleando la instrucción vTaskDelete(). El código se muestra a continuación:

```
static void Task1(void *pvParameters) {
 for(uint32_t x=1;x<=nt;x++){
  taskYIELD();
 }
 vTaskDelete(tarea1);
}
//-----------------------------------------------------------------------------
static void Task2(void *pvParameters) {
 for(uint32_t y=1;y<=nt;y++){
  taskYIELD();
 }
```

```
  vTaskDelete(tarea2);
}
```

Una vez auto-eliminadas las tareas se da paso a la tercera tarea para el cálculo del tiempo de conmutación de tareas. Es importante recalcar que este tiempo final es la división del tiempo neto para la multiplicación del número de iteraciones (500000) por dos, ya que son dos tareas que suman tiempo. Finalmente, esta tarea también se auto-elimina. El código se muestra a continuación:

```
static void Resultado(void *pvParameters) {
  t=micros()-ti-t;
  Serial.print((float)t/((float)nt*2.0));
  Serial.println(" Microsegundos por tarea");
  vTaskDelete(tarea3);
}
```

En el apéndice B se muestra el código FreeRTOS completo utilizado en Arduino Due para esta comparativa Rhealstone de tiempo de conmutación de tarea.

4.2.1.2 ChibiOS/RT

En la tarea principal o conductora de ChibiOS/RT, se crean las dos tareas con la misma prioridad. Como en ChibiOS/RT al momento de crear una tarea se ejecuta automáticamente (ChibiOS/RT no permite crear todas las tareas y al final activarlas mediante una instrucción), se ha empleado un mútex (exclusión mutua) para tomar el recurso hasta que ambas tareas queden definidas, así cuando se suelta el recurso empiezan a conmutar las dos tareas. Por lo que para eliminar este tiempo adicional provocado por el mútex que interfiere en lo que se quiere obtener (conmutación de tarea) se mide el tiempo de definir y de tomar y soltar un mútex en dos tareas auxiliares diferentes de igual prioridad (th), ya que en las dos tareas a conmutar se toma y suelta un mútex respectivamente; luego se toma el tiempo inicial en dónde se definen y se empiezan a ejecutar las tareas (ti) y estos dos tiempos se consideran en el tiempo final de la comparativa.

Una vez terminadas las tareas, estas se auto-eliminan y se vuelve a la tarea principal o conductora de ChibiOS/RT en donde se procede al cálculo del tiempo de conmutación de tarea. Como en el caso de FreeRTOS, este tiempo final es la división del tiempo neto para la multiplicación del número de iteraciones (500000) por dos, ya que son dos tareas que suman tiempo. De igual forma, al final esta tarea, que es la principal de ChibiOS/RT, se auto-elimina. El código se muestra a continuación:

```
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 2, Task1a, NULL);
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 2, Task2a, NULL);
  th=micros()-ti;
  chMtxLock(&mtx);
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO+1, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO+1, Task2, NULL);
  chMtxUnlock();
  t=micros()-ti-t-th;
  Serial.print((float)t/((float)nt*2.0));
  Serial.println(" Microsegundos por tarea");
  chThdExit(0);
}
```

El código de las tareas auxiliares del mútex, antes mencionadas, se muestra a continuación:

```
static WORKING_AREA(waTask1a, 64);
static msg_t Task1a(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
}
static WORKING_AREA(waTask2a, 64);
static msg_t Task2a(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
}
```

Las tareas de la misma prioridad que generan la conmutación, primero y por una sola vez, toman y sueltan un mútex (esto permite en ChibiOS/RT que al ejecutarse ambas tareas estén definidas) y luego ya empiezan hacer el cambio de tareas mediante la instrucción chThdYield() haciendo sólo este proceso 500000 iteraciones. Terminada todas las iteraciones de las dos tareas cada una se elimina a sí misma empleando la instrucción chThdExit(). El código se muestra a continuación:

```
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
 for(uint32_t x=1;x<=nt;x++){
  chThdYield();
 }
 chThdExit(0);
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
 for(uint32_t y=1;y<=nt;y++){
  chThdYield();
 }
 chThdExit(0);
}
```

En el apéndice C se muestra el código ChibiOS/RT completo utilizado en Arduino Due para esta comparativa Rhealstone de tiempo de conmutación de tarea.

4.2.2 Tiempo de expulsión (Preemption Time)

Esta comparativa Rhealstone busca determinar el tiempo medio que trascurre para dar paso a una tarea de más alta prioridad que se despierta cuando una de más baja prioridad está en ejecución. Este proceso se realiza 100000 iteraciones para obtener un tiempo medio. Para ello, se calcula el número de iteraciones necesarias para que un bucle FOR incremente una variable contador con paso 1 en el tiempo que dura un tick de reloj (1 milisegundo). Este valor resultó ser de 1551 para el Arduino Due y el código fuente que generó el mencionado valor se muestra en el apéndice A. Sin embargo, este valor de 1551 no fue el que se empleó para esta comparativa, sino que según [24] el valor a emplear es este valor ligeramente

aumentado, que para los efectos se ha llamado “tick aumentado”, representado por la variable tick_a y que se estableció en 1572. Este “tick aumentado” es el que permite generar un proceso controlado en tiempo de un poco más de lo que dura un tick de reloj (1 milisegundo) que es fundamental para esta comparativa de tiempo de expulsión.

Para esta comparativa se definen dos tareas de diferentes prioridades. La tarea de prioridad más alta se pausa por el lapso de 1 milisegundo (equivalente a un tick de reloj ya que 1000Hz equivale a 1 milisegundo), entrando inmediatamente en funcionamiento la tarea de prioridad más baja, que tendrá un bucle FOR que permitirá incrementar una variable contador con paso 1 por el lapso de un tick aumentado. Este procedimiento permite la expulsión de la tarea de menor prioridad ya que la pausa de un tick de reloj de la tarea de mayor prioridad se terminará antes del tick aumentado, este tick aumentado cumple la función de tener a la tarea de baja prioridad en proceso cuando la tarea de mayor prioridad se despierte y por tanto expulse a la mencionada tarea baja prioridad.

El procedimiento descrito en el párrafo anterior se repetirá 100000 iteraciones y una vez terminado las tareas se auto-eliminaran para dar paso al cálculo del tiempo, restándose los tiempos que puedan afectar a lo que se quiere medir (tiempo de expulsión).

Para eliminar un tiempo que afecta al cálculo del tiempo de expulsión, se midió y guardó en una variable (t) el tiempo que consume un bucle FOR a 100000 iteraciones y dentro de él otro bucle FOR a 1572 iteraciones (tick_a) que incrementa una variable contador con paso 1 (esto elimina del tiempo final, al tiempo de las operaciones de la tarea de más baja prioridad) y luego otro bucle FOR a 100000 iteraciones sin realizar ninguna instrucción (esto elimina del tiempo final, al tiempo que consume el bucle FOR de la tarea de más alta prioridad), para luego restarlas al tiempo total. Finalmente se vuelve a cero la variable contador, dejándola lista para ser incrementada por la tarea de más baja prioridad. El código se muestra a continuación:

```
ti=micros();
for(uint32_t x=1;x<=nt;x++)
  for(i=0;i<tick_a;i++)
    a++;
for(uint32_t y=1;y<=nt;y++)
  ;
t=micros()-ti;
a=0;
```

Al final, al tiempo total obtenido en esta comparativa se le resta el tiempo obtenido en la comparativa de conmutación de tarea para el respectivo RTOS, debido a que la conmutación de tarea influye en este procedimiento para obtener la comparativa. Cabe indicar que todo lo descrito en esta comparativa se aplicó diez veces para ambos RTOS, generando siempre el mismo resultado de tiempo, el cual se presenta en la tabla 3 expresado en microsegundos, donde se puede observar que se restó el valor de la conmutación de tareas (Cmt) a la prueba, mostrándose el valor final de la comparativa para cada RTOS en la columna Resultado respectivamente.

Comparativa	FreeRTOS			ChibiOS/RT		
Expulsión(100000)	Prueba (µs)	Cmt (µs)	Resultado (µs)	Prueba (µs)	Cmt (µs)	Resultado (µs)
	37,56	4,73	32,83	64,36	1,95	62,41

Tabla 3: Resultado de la aplicación de la comparativa Rhealstone - tiempo de expulsión

Como cada RTOS tiene sus instrucciones y formas de trabajar distintas, se describe a continuación lo realizado específicamente para cada sistema en esta comparativa:

4.2.2.1 FreeRTOS

Se crean las dos tareas con diferentes prioridades y una tercera tarea con prioridad más baja, que se activará terminadas las dos tareas. Esta tercera tarea se activa para calcular en ella el tiempo final de la comparativa. Como FreeRTOS permite crear las tareas y estas sólo se activan cuando se emplea la instrucción vTaskStartScheduler(), se toma el tiempo inicial en dónde se empiezan a ejecutar las tareas (ti) y luego se lo considera en el cálculo del tiempo de la comparativa. El código se muestra a continuación:

```
xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,3,&tarea1);
xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
ti=micros();
vTaskStartScheduler();
```

La tarea1 es la de prioridad más alta y contiene un bucle FOR que repite 100000 iteraciones la instrucción vTaskDelay(1) que permite a la tarea1 quedarse pausada por el lapso de 1 milisegundo (1 tick de reloj). Esto permite, en cada iteración, que se active la tarea2 que terminará su bucle FOR interno y continuará con la siguiente iteración del bucle FOR externo hasta que se active nuevamente la tarea1. La tarea2 contiene dos bucles FOR, uno externo que repite 100000 iteraciones un bucle FOR interno que va hasta un tick aumentado, incrementando una variable contador con paso 1.

Lo descripto anteriormente, permite que la tarea1 (prioridad más alta) siempre se pause y se despierte cuando la tarea2 (prioridad más baja) está ejecutándose, todo esto se realiza 100000 iteraciones por lo que para medir el tiempo de expulsión de una iteración se divide al final el tiempo neto para este valor y se le resta la conmutación de tarea. Terminada todas las iteraciones de las dos tareas, cada una se elimina a sí misma empleando la instrucción vTaskDelete(). El código se muestra a continuación:

```
static void Task2(void *pvParameters) {
  for(uint32_t x=1;x<=nt;x++)
    for(i=0;i<tick_a;i++)
      a++;
  vTaskDelete(tarea2);
}
//----------------------------------------------------------------------------
static void Task1(void *pvParameters) {
   for(uint32_t y=1;y<=nt;y++)
     vTaskDelay(1);
  vTaskDelete(tarea1);
}
```

Una vez auto-eliminadas las tareas se da paso a la tercera tarea para el cálculo del tiempo parcial de esta comparativa (se establece como tiempo parcial porque finalmente se debe restar el tiempo obtenido en la comparativa de conmutación de tarea). Es importante recalcar que este tiempo parcial es la división del tiempo neto para el número de iteraciones (100000) que hicieron ambas tareas. No se duplica ya que lo que se hace por iteración es que la tarea1 termine una iteración de la tarea2, por tanto, ambas tareas mezclan sus iteraciones. Tras completar este cálculo esta tarea también se auto-elimina. El código se muestra a continuación:

```
static void Resultado(void *pvParameters) {
  t=micros()-ti-t;
  Serial.print((float)t/((float)nt));
```

```
 Serial.println(" Microsegundos");
 vTaskDelete(tarea3);
}
```

En el apéndice D se muestra el código FreeRTOS completo utilizado en Arduino Due para esta comparativa Rhealstone de tiempo de expulsión entre dos tareas de distintas prioridades.

4.2.2.2 ChibiOS/RT

En la tarea principal o conductora de ChibiOS/RT, se crean las dos tareas con diferentes prioridades. Como en ChibiOS/RT al momento de crear una tarea se ejecuta automáticamente (ChibiOS/RT no permite crear todas las tareas y al final activarlas mediante una instrucción), se incluye el tiempo de definición de cada tarea como tal; por lo que para eliminar este aspecto, que interfiere en lo que se quiere obtener (tiempo de expulsión), se mide el tiempo de definir dos tareas auxiliares de diferente prioridad (th). Asimismo, se toma el tiempo inicial en dónde se definen y se empiezan a ejecutar las tareas (ti) y estos dos tiempos se consideran en el tiempo de la comparativa.

Una vez terminadas las tareas, estas se auto-eliminan y se vuelve a la tarea principal o conductora de ChibiOS/RT en donde se procede al cálculo del tiempo parcial de esta comparativa (se establece como tiempo parcial, porque finalmente se debe restar el tiempo obtenido en la comparativa de conmutación de tarea). Como en los test anteriores, este tiempo parcial es la división del tiempo neto para el número de iteraciones (100000) que hicieron ambas tareas. No se duplica ya que lo que se hace por iteración es que la tarea1 termine una iteración de la tarea2 y, por tanto ambas tareas mezclan sus iteraciones. De igual forma al final esta tarea que es la principal de ChibiOS/RT se auto-elimina. El código se muestra a continuación:

```
void chSetup() {
 ti=micros();
 chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 4, Task1a, NULL);
 chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 3, Task2a, NULL);
 th=micros()-ti;
 ti=micros();
 chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO + 2, Task1, NULL);
 chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO + 1, Task2, NULL);
 t=micros()-ti-t-th;
 Serial.print((float)t/((float)nt));
 Serial.println(" Microsegundos");
 chThdExit(0);
}
```

La tarea1 es la de prioridad más alta, esta contiene un bucle FOR que repite 100000 iteraciones la instrucción chThdSleep(1) que permite a la tarea1 quedarse pausada por el lapso de 1 milisegundo (1 tick de reloj) dando paso, en cada iteración, a que se active la tarea2. Esta terminará su bucle FOR interno y continuará con la siguiente iteración del bucle FOR externo hasta que se active nuevamente la tarea1. La tarea2 contiene dos bucles FOR, uno externo que repite 100000 iteraciones un bucle FOR interno que va hasta un tick aumentado, incrementando una variable contador con paso 1.

Lo descripto anteriormente, permite que la tarea1 (prioridad más alta) siempre se pause y se despierte cuando la tarea2 (prioridad más baja) está ejecutándose, todo esto se realiza 100000 iteraciones por lo que para medir el tiempo de expulsión de una iteración se divide al final el tiempo neto para este valor y se le resta la conmutación de tarea. Terminada todas las iteraciones de las dos tareas cada una se elimina a sí misma empleando la instrucción chThdExit(). El código se muestra a continuación:

```
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  for(uint32_t x=1;x<=nt;x++)
    for(i=0;i<tick_a;i++)
      a++;
  chThdExit(0);
}
//-----------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  for(uint32_t y=1;y<=nt;y++)
    chThdSleep(1);
  chThdExit(0);
}
```

En el apéndice E se muestra el código ChibiOS/RT completo utilizado en Arduino Due para esta comparativa Rhealstone de tiempo de expulsión entre dos tareas de distintas prioridades.

4.2.3 Tiempo de espera de un semáforo (Semaphore Shuffle Time)

Un semáforo es un método para restringir el acceso a un recurso compartido de un dispositivo. Esta comparativa Rhealstone busca determinar el tiempo medio que trascurre en tomar – soltar un semáforo binario por parte de una tarea que se ve bloqueada por el semáforo binario de otra tarea de igual prioridad, tras realizar 100000 iteraciones para obtener un tiempo medio. Para ello se definen un semáforo binario y dos tareas con la misma prioridad. Cada tarea tiene dos procesos que pueden tomar o soltar el semáforo y dar paso a la otra tarea después de ejecutar el proceso que corresponde.

Al ejecutarse el programa con valor para la variable de control sema_ejec =1 (con semáforo); la tarea1 comenzará tomando el semáforo y luego cederá el paso a la tarea2, que intentará tomar el semáforo, pero éste estará bloqueado por la tarea1 y deberá esperar a que el semáforo esté disponible, por lo que dará pasó a la tarea1, cuyo siguiente proceso es liberar el semáforo y dar paso nuevamente a la tarea2. Ahora la tarea2 ve que la tarea1 ha liberado el semáforo y luego lo toma y da paso a la tarea1. En ese momento la tarea1 intenta tomar el semáforo, pero no puede ya que la Tarea2 lo tiene, y deberá esperar a que esté disponible, por lo que dará paso a la tarea2 cuyo siguiente proceso es liberar el semáforo y dar paso a la tarea1 que toma el semáforo y luego cede el paso a la tarea2 y así se repite el mismo proceso descrito anteriormente para 100000 iteraciones controladas por un bucle FOR. Una vez terminadas estas 100000 iteraciones del bucle FOR, las tareas se auto-eliminan y se da paso al cálculo del tiempo de la prueba.

Para obtener el resultado final de la comparativa y eliminar los aspectos externos que afectan a la misma, se deben realizar dos pruebas (ejecuciones del programa), cambiando el valor de la variable de control (sema_ejec) para que en la primera prueba permita emplear el semáforo (sema_ejec=1) y en la segunda prueba no lo permita (sema_ejec=0) y así al final restar los tiempos finales conseguidos en ambas pruebas para determinar el tiempo final de la comparativa, ya que esta diferencia de tiempos resultante es lo que aporta la presencia del semáforo descrito anteriormente, eliminado el tiempo de ejecución de los dos bucles FOR a 100000 iteraciones, el tiempo de conmutación de tareas y todos los aspectos propios del programa (creación de tareas, creación del semáforo, declaración y asignación de variables, etc).

Cabe indicar que todo lo descrito en esta comparativa se aplicó diez veces para ambos RTOS, generando siempre el mismo resultado de tiempo, el cual se midió en microsegundos (μs) y se presenta en la tabla 4, donde se puede observar que se restaron entre ellas las pruebas que se hicieron sin y con la presencia del tiempo de espera del semáforo, mostrándose el valor final de la comparativa para cada RTOS en la columna Resultado respectivamente.

Comparativa	FreeRTOS			ChibiOS/RT		
	Sin (µs)	Con (µs)	Resultado (µs)	Sin (µs)	Con (µs)	Resultado (µs)
Espera_Sema(100000)	38,5	115,47	76,97	17,61	31,87	14,26

Tabla 4: Resultado de la aplicación de la comparativa Rhealstone - tiempo de espera de un semáforo

Como cada RTOS tiene sus instrucciones y formas de trabajar distintas, se describe a continuación lo realizado específicamente para cada sistema en esta comparativa:

4.2.3.1 FreeRTOS

Se asigna el valor a la variable de control (sema_ejec) dependiendo de la prueba con (1) o sin (0) semáforo que se quiera realizar. Si la prueba es con semáforo, se crea con la instrucción vSemaphoreCreateBinary() el semáforo binario cuyo nombre es sema (definido con la instrucción SemaphoreHandle_t). Luego, independientemente del tipo de prueba, se crean las dos tareas con la misma prioridad y una tercera tarea con prioridad más baja, que se activará terminadas las dos tareas. Esta tercera tarea se activa para calcular en ella el tiempo final de la prueba que corresponda (con o sin semáforo). Como FreeRTOS permite crear las tareas y estas sólo se activan cuando se emplea la instrucción vTaskStartScheduler() se toma el tiempo inicial en dónde se empiezan a ejecutar las tareas (ti) y luego se lo considera en el cálculo del tiempo final. El código se muestra a continuación:

```
sema_ejec = 1;
if (sema_ejec == 1)
  vSemaphoreCreateBinary(sema);
xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,2,&tarea1);
xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
ti=micros();
vTaskStartScheduler();
```

Las tareas de la misma prioridad que trabajan con el semáforo realizan 100000 iteraciones mediante un bucle FOR, en donde por cada iteración se verifica el valor que tiene la variable de control (sema_ejec). Si es 1 significa que la prueba debe trabajar con el semáforo y, por tanto, cuando corresponda cada tarea lo tomará con la instrucción xSemaphoreTake(sema, portMAX_DELAY) y cuando toque lo liberará con la instrucción xSemaphoreGive(sema). Entre tomar y soltar el semáforo cada tarea pasa el control a la otra mediante la instrucción taskYIELD() por lo que en total serían dos paso por cada iteración del bucle FOR, produciéndose lo descrito en el numeral anterior (Semaphore Shuffle). Si la variable de control es 0 significa que la prueba no debe trabajar con el semáforo y, por tanto, cada iteración lo que hace es pasar dos veces el control a la otra tarea mediante la instrucción taskYIELD(). Terminada todas las iteraciones de las dos tareas, cada una se elimina a sí misma empleando la instrucción vTaskDelete(). El código se muestra a continuación:

```
static void Task2(void *pvParameters) {
 for(uint32_t x=1;x<=nt;x++)
 {
  if (sema_ejec == 1)
  {
   xSemaphoreTake(sema,portMAX_DELAY);
  }
  taskYIELD();
  if (sema_ejec == 1)
```

```
    {
      xSemaphoreGive(sema);
    }
    taskYIELD();
  }
  vTaskDelete(tarea2);
}
//---------------------------------------------------------------------------
static void Task1(void *pvParameters) {
  for(uint32_t y=1;y<=nt;y++)
  {
    if (sema_ejec == 1)
    {
      xSemaphoreTake(sema,portMAX_DELAY);
    }
    taskYIELD();
    if (sema_ejec == 1)
    {
      xSemaphoreGive(sema);
    }
    taskYIELD();
  }
  vTaskDelete(tarea1);
}
```

Una vez auto-eliminadas las tareas se da paso a la tercera tarea para el cálculo del tiempo de la prueba que en ese momento se esté haciendo dependiendo del valor de la variable de control (sema_ejec). Es importante recalcar que este tiempo final de la prueba, es la división del tiempo neto para la multiplicación del número de iteraciones (100000) por dos, ya que son dos las tareas que suman tiempo. Esta tarea se auto-elimina cuando concluye ese cálculo de tiempos. El código se muestra a continuación:

```
static void Resultado(void *pvParameters) {
  t=micros()-ti;
  Serial.print((float)t/((float)nt)*2);
  if (sema_ejec == 0)
    Serial.println(" Microsegundos SIN semáforo");
  else
    Serial.println(" Microsegundos CON semáforo");
  Serial.println("\nIMPORTANTE: Para obtener el tiempo final reste, sema_ejec = 0; y sema_ejec = 1;");
  vTaskDelete(tarea3);
}
```

En el apéndice F se muestra el código FreeRTOS completo utilizado en Arduino Due para esta comparativa Rhealstone de Semaphore Shuffle Time.

4.2.3.2 ChibiOS/RT

Se asigna el valor a la variable de control (sema_ejec) dependiendo de la prueba con (1) o sin (0) semáforo que se quiera realizar, si la prueba es con semáforo se crea con la instrucción chBSemInit() el semáforo binario cuyo nombre es sema (definido con la instrucción BSEMAPHORE_DECL()). Luego, independientemente del tipo de prueba, se crea un mútex (exclusión mutua) para el procedimiento auxiliar

que se explicará más adelante (chMtxInit()) y se llama a la tarea principal o conductora de ChibiOS/RT (chBegin(chSetup)). El código se muestra a continuación:

```
sema_ejec = 1;
if (sema_ejec == 1)
  chBSemInit(&sema, FALSE);
chMtxInit(&mtx);
chBegin(chSetup);
```

A continuación, en la tarea principal o conductora de ChibiOS/RT se crean las dos tareas con la misma prioridad. Como en ChibiOS/RT al momento de crear una tarea se ejecuta automáticamente (ChibiOS/RT no permite crear todas las tareas y al final activarlas mediante una instrucción), se ha empleado un mútex para tomar el recuso hasta que ambas tareas que trabajan con el semáforo queden definidas. Así cuando se libera el recurso empiezan las dos tareas a funcionar. Por lo que para eliminar este tiempo adicional provocado por el mútex, que interfiere en la medición del Semaphore Shuffle Time, se mide el tiempo de definición y de tomar y soltar un mútex en dos tareas auxiliares diferentes de igual prioridad (th), ya que en las dos tareas con las que se va a trabajar se toma y suelta un mútex respectivamente; luego se toma el tiempo inicial en dónde se definen y se empiezan a ejecutar las tareas (ti) y estos dos tiempos se consideran para la resta con el tiempo al terminar de ejecutar las mencionadas tareas (t). Este procedimiento se replica en ambos casos, es decir, se realice la prueba con semáforo o sin él.

Una vez terminadas las tareas que trabajan con el semáforo, estas se auto-eliminan y se vuelve a la tarea principal o conductora de ChibiOS/RT en donde se procede al cálculo del tiempo de la prueba que en ese momento se esté haciendo dependiendo del valor de la variable de control (sema_ejec). Es importante recalcar que este tiempo final de la prueba es la división del tiempo neto para la multiplicación del número de iteraciones (100000) por dos, ya que son dos tareas que suman tiempo. Como en anteriores test, esta tarea, la principal de ChibiOS/RT, se auto-elimina tras obtener el tiempo final. El código se muestra a continuación:

```
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 2, Task1a, NULL);
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 2, Task2a, NULL);
  th=micros()-ti;
  chMtxLock(&mtx);
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO + 1, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO + 1, Task2, NULL);
  chMtxUnlock();
  t=micros()-ti-th;
  Serial.print((float)t/((float)nt)*2);
  if (sema_ejec == 0)
    Serial.println(" Microsegundos SIN semáforo");
  else
    Serial.println(" Microsegundos CON semáforo");
  Serial.println("\nIMPORTANTE: Para obtener el tiempo final reste, sema_ejec = 0; y sema_ejec = 1;");
  chThdExit(0);
}
```

El código de las tareas auxiliares del mútex, antes mencionadas, se muestra a continuación:

```
static WORKING_AREA(waTask1a, 64);
static msg_t Task1a(void *arg) {
```

```
  chMtxLock(&mtx);
  chMtxUnlock();
}
static WORKING_AREA(waTask2a, 64);
static msg_t Task2a(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
}
```

Las tareas de la misma prioridad que trabajan con el semáforo, primero y por una sola vez, toman y sueltan un mútex (esto permite en ChibiOS/RT que al ejecutarse ambas tareas estén definidas) y luego poseen un bucle FOR a 100000 iteraciones, en donde por cada iteración se verifica el valor que tiene la variable de control (sema_ejec). Si es 1 significa que la prueba debe trabajar con el semáforo y por tanto, cuando corresponda cada tarea lo tomará con la instrucción chBSemWaitTimeout(&sema, TIME_INFINITE) y cuando toque lo liberará con la instrucción chBSemSignal(&sema). Entre tomar y soltar el semáforo cada tarea pasa el control a la otra mediante la instrucción chThdYield(), por lo que en total serían dos pasos por cada iteración del bucle FOR, produciéndose lo descrito en el numeral anterior (Semaphore Shuffle). Si la variable de control es 0 significa que la prueba no debe trabajar con el semáforo y, por tanto, cada iteración lo que hace es pasar dos veces el control a la otra tarea mediante la instrucción chThdYield(). Terminadas todas las iteraciones de las dos tareas, cada una se elimina a sí misma empleando la instrucción chThdExit(). El código se muestra a continuación:

```
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
  for(uint32_t x=1;x<=nt;x++)
  {
    if (sema_ejec == 1)
    {
      chBSemWaitTimeout (&sema,TIME_INFINITE);
    }
    chThdYield();
    if (sema_ejec == 1)
    {
      chBSemSignal(&sema);
    }
    chThdYield();
  }
  chThdExit(0);
}
//---------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
  for(uint32_t y=1;y<=nt;y++)
  {
    if (sema_ejec == 1)
    {
      chBSemWaitTimeout (&sema,TIME_INFINITE);
    }
    chThdYield();
```

```
    if (sema_ejec == 1)
    {
      chBSemSignal(&sema);
    }
    chThdYield();
  }
  chThdExit(0);
}
```

En el apéndice G se muestra el código ChibiOS/RT completo utilizado en Arduino Due para esta comparativa Rhealstone de Semaphore Shuffle Time.

4.2.4 Tiempo de ruptura de interbloqueo (Deadlock Breaking Time)

Esta comparativa Rhealstone busca determinar el tiempo medio que tarda en romperse el bloqueo producido por una tarea de más baja prioridad que solicita un mútex sobre otra de más alta prioridad que solicita el mismo mútex y en donde aparece una tarea con prioridad intermedia que no permite el paso inmediato entra las tareas bloqueadas. Este proceso se repite 10000 iteraciones para obtener un tiempo medio. Para ello, se calcula el número de iteraciones necesarias para que un bucle FOR incremente una variable contador con paso 1 en el tiempo que dura un tick de reloj (1 milisegundo), este valor fue de 1551 y el código fuente que generó el mencionado valor se muestra en el apéndice A. Sin embargo, este valor de 1551 no fue el que se empleó para esta comparativa, sino que según [24] el valor a emplear es este valor ligeramente aumentado, que para los efectos se ha llamado "tick aumentado", representado por la variable tick_a y que se estableció en 1572. Este "tick aumentado" es el que permite generar un proceso controlado en tiempo de un poco más de lo que dura un tick de reloj (1 milisegundo) que es fundamental para esta comparativa de tiempo de ruptura de interbloqueo.

Para esta comparativa se crea un mútex y tres tareas con diferentes prioridades. La tarea1 tiene la prioridad más alta, la tarea2 prioridad media y la tarea3 la prioridad más baja. Al ejecutarse el programa con valor para la variable de control ip=1 (con interbloqueo), la tarea1 se pausará 1 milisegundo y entrará a ejecutarse la tarea2 que luego de incrementar una variable contador con paso 1, que se repetirá por las iteraciones de un bucle FOR hasta un tick aumentado sobre 4 (como se indicó anteriormente este tick aumentado equivale a 1572 que dividido para 4 da un valor de 393), se pausa por 1 milisegundo; permitiendo que se ejecute la tarea3, ya que la tarea1 seguirá pausada debido a que la tarea2 sólo consumirá un poco más de 1/4 del milisegundo de la pausa de la tarea1.

La tarea3 toma el mútex y empieza incrementar la variable contador con paso 1 que se repetirá por las iteraciones de un bucle FOR hasta un tick aumentado. Sin embargo, este proceso se verá interrumpido ya que se terminará la pausa de la tarea1, que entrará en funcionamiento e intentará tomar el mútex, que al estar bloqueado por la tarea3, hará que la tarea1 se detenga y que entre en funcionamiento la tarea2, por lo que la tarea1 debe esperar a que la tarea2 dé paso a la tarea3 (produciéndose el interbloqueo). Al tener paso, la tarea3 liberará el mútex, activando inmediatamente a la tarea1 que esperaba ese evento, la misma que inmediatamente luego de tomarlo soltará el mútex. Y se repite por 10000 iteraciones todo el proceso que se ha descrito, desde la pausa de 1 milisegundo de la tarea1. Una vez terminadas estas 10000 iteraciones, las tareas se auto-eliminan y se da paso al cálculo del tiempo de la prueba.

Para obtener el resultado final de la comparativa y eliminar los aspectos externos que afectan a la misma, se deben realizar dos pruebas (ejecuciones del programa), cambiando el valor de la variable de control (ip) para que en la primera prueba permita emplear el mútex de la tarea1 que producirá el interbloqueo (ip=1) y en la segunda prueba no lo permita (ip=0) y así al final restar los tiempos finales conseguidos en ambas pruebas, determinado el tiempo final de la comparativa. Esta diferencia de tiempos resultante es lo que aporta la presencia del interbloqueo descrita anteriormente, eliminado el tiempo de ejecución de los bucles

FOR, las condicionales, las 10000 iteraciones, el tiempo de conmutación de tareas, tiempo de expulsión y todos los aspectos propios del programa (creación de tareas, creación mútex, declaración y asignación de variables, etc.). Cabe indicar que todo lo descrito en esta comparativa se aplicó diez veces para ambos RTOS, generando siempre el mismo resultado de tiempo, el cual se midió en microsegundos (µs) y se presenta en la tabla 5, donde se puede observar que se restaron entre ellas las pruebas que se hicieron sin y con la presencia del tiempo ruptura de interbloqueo, mostrándose el valor final de la comparativa para cada RTOS en la columna Resultado respectivamente.

Comparativa	FreeRTOS			ChibiOS/RT		
	Sin (µs)	Con (µs)	Resultado (µs)	Sin (µs)	Con (µs)	Resultado (µs)
Interbloqueo(10000)	1311,35	1343,13	31,78	1262,8	1267,65	4,85

Tabla 5: Resultado de la aplicación de la comparativa Rhealstone - tiempo de ruptura de interbloqueo

Como cada RTOS tiene sus instrucciones y formas de trabajar distintas, se describe a continuación lo realizado específicamente para cada sistema en esta comparativa:

4.2.4.1 FreeRTOS

Se asigna el valor a la variable de control (ip) dependiendo de la prueba que se quiera realizar, con o sin mútex (1 o 0), en la tarea1 para producir el interbloqueo. Independientemente del tipo de prueba, se crea el mútex con la instrucción xSemaphoreCreateMutex() y luego se crean las tres tareas con diferentes prioridades y una cuarta tarea con prioridad más baja, que se activará terminadas las tres tareas anteriores. Esta cuarta tarea se activa para calcular en ella el tiempo final de la prueba que corresponda (con o sin mútex en la tarea1). Como FreeRTOS permite crear las tareas y estas sólo se activan cuando se emplea la instrucción vTaskStartScheduler() se toma el tiempo inicial en dónde se empiezan a ejecutar las tareas (ti) y luego se lo considera en el cálculo del tiempo final. El código se muestra a continuación:

```
ip = 1;
mtx = xSemaphoreCreateMutex();
xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,4,&tarea1);
xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,3,&tarea2);
xTaskCreate(Task3,"Task3",configMINIMAL_STACK_SIZE,NULL,2,&tarea3);
xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea4);
ti=micros();
vTaskStartScheduler();
```

La tarea1 tiene la prioridad más alta, la tarea2 la prioridad intermedia y la tarea3 la prioridad más baja de las tres que se han definido para la comparativa. Las tres tareas están bajo un bucle while infinito que se rompe cuando la variable controladora de iteraciones, que es única para cada tarea y que se incrementa en cada iteración de la tarea respectiva, alcance el valor 10000, tras haber producido el mismo número de iteraciones. La terminación de este bucle while se verifica siempre al principio de cada tarea y se rompe auto-eliminando la tarea con la instrucción vTaskDelete().

La tarea1 empieza haciendo una pausa de 1 milisegundo con la instrucción vTaskDelay(1) lo que permite que funcione la tarea2, que luego de incrementar una variable contador con paso 1 en un bucle FOR hasta un tick aumentado dividido por 4, que equivale a 393 iteraciones, hace una pausa de 1 milisegundo con la instrucción vTaskDelay(1). En ese momento entra en funcionamiento la tarea3, ya que la tarea1 aún está en pausa al no cumplirse el milisegundo con las 393 iteraciones. La tarea3 toma el mútex de nombre mtx con la instrucción xSemaphoreTake(mtx, portMAX_DELAY), (definido con la instrucción SemaphoreHandle_t) y luego empieza un bucle FOR que incrementa la variable contador con paso 1 hasta un tick aumentado que se verá interrumpido, porque se terminará el milisegundo de pausa de la tarea1 que se activará para tomar

el mútex con la instrucción xSemaphoreTake(mtx, portMAX_DELAY) que al estar bloqueado por la tarea3, hace que la tarea1 se detenga y que entre en funcionamiento la tarea2, por lo que la tarea1 debe esperar a que la tarea2 dé paso a la tarea3 (produciéndose el interbloqueo). Al tener paso, la tarea3 soltará el mútex con la instrucción xSemaphoreGive(mtx), activando inmediatamente a la tarea1 que esperaba ese evento, la misma que inmediatamente luego de tomarlo, soltará el mútex. Durante el test, se repite por 10000 iteraciones todo el proceso que se ha descrito, desde la pausa de 1 milisegundo de la tarea1.

Cabe indicar que todo este proceso se realiza si el valor a la variable de control es 1 (ip=1). Esto se verifica únicamente en la tarea1, ya que cuando esta tarea toma el mútex es cuando se produce el interbloqueo. En el caso de que el valor de la variable sea 0 (ip=0) se hace el mismo proceso, pero sin tomar el mútex en la tarea1 por lo que el tiempo de ruptura del interbloqueo no aparece. Al final se restarán ambos tiempos con ip=1 y con ip=0 para obtener el resultado final de la comparativa. El código se muestra a continuación:

```
static void Task3(void *pvParameters) {
 while(1)
 {
  if (i3== nt)
  {
   vTaskDelete(tarea3);
  }
  xSemaphoreTake(mtx,portMAX_DELAY);
  for (x=0;x<tick_a;x++)
   a++;
  i3++;
  xSemaphoreGive(mtx);
 }
}
//-----------------------------------------------------------------------------
static void Task2(void *pvParameters) {
 while (1)
 {
  if (i2 == nt)
  {
   vTaskDelete(tarea2);
  }
  for (y=1;y<=tick_a/4;y++)
   a++;
  vTaskDelay(1);
  i2++;
 }
}
//-----------------------------------------------------------------------------
static void Task1(void *pvParameters) {
 while(1)
 {
  if (i1== nt)
  {
   vTaskDelete(tarea1);
  }
  vTaskDelay(1);
  if (ip == 1)
  {
   xSemaphoreTake(mtx,portMAX_DELAY);
```

```
    xSemaphoreGive(mtx);
   }
   i1++;
  }
}
```

Una vez auto-eliminadas las tareas se da paso a la cuarta tarea para el cálculo del tiempo de la prueba que en ese momento se esté haciendo, dependiendo del valor de la variable de control (ip). Es importante recalcar que el tiempo final de la prueba es la división del tiempo neto por el número de iteraciones (10000) que hicieron las tres tareas. No se triplica ya que las tres tareas mezclan sus iteraciones. Esta cuarta tarea se auto-elimina también al final del cálculo del tiempo. El código se muestra a continuación:

```
static void Resultado(void *pvParameters) {
  t=micros()-ti;
  Serial.print((float)t/((float)nt));
  if (ip == 0)
    Serial.println(" Microsegundos SIN interbloqueo");
  else
    Serial.println(" Microsegundos CON interbloqueo");
  Serial.println("\nIMPORTANTE: Para obtener el tiempo final reste, ip = 0; e ip = 1;");
  vTaskDelete(tarea4);
}
```

En el apéndice H se muestra el código FreeRTOS completo utilizado en Arduino Due para esta comparativa Rhealstone de Deadlock Breaking Time.

4.2.4.2 ChibiOS/RT

Se asigna el valor a la variable de control (ip) dependiendo de la prueba que se quiera realizar, con o sin mútex (1 o 0), en la tarea1 para producir el interbloqueo. Independientemente del tipo de prueba, se crea el mútex denominado mtx, declarado con la instrucción MUTEX_DECL(), con la instrucción chMtxInit() y se llama a la tarea principal o conductora de ChibiOS/RT, chBegin(chSetup). El código se muestra a continuación:

```
ip = 1;
chMtxInit(&mtx);
chBegin(chSetup);
```

A continuación, independientemente del tipo de prueba, se crean las tres tareas con distintas prioridades. La tarea1 tiene la prioridad más alta, la tarea2 la prioridad intermedia y la tarea3 la prioridad más baja de las tres que se han definido para la comparativa. Para medir el tiempo de la prueba se toma el tiempo inicial en donde se definen y se empiezan a ejecutar las tareas (ti) y se lo resta del tiempo cuando se terminan de ejecutar las mismas (t), ya sea para las pruebas con o sin mútex en la tarea1.

Una vez terminadas las tres tareas de la comparativa, estas se auto-eliminan y se vuelve a la tarea principal o conductora de ChibiOS/RT en donde se procede al cálculo del tiempo de la prueba que en ese momento se esté haciendo, dependiendo del valor de la variable de control (ip). Este tiempo final de la prueba es la división del tiempo neto para el número de iteraciones (10000) que hicieron las tres tareas. No se triplica ya que las tres tareas mezclan sus iteraciones. De igual forma, al final, esta tarea que es la principal de ChibiOS/RT se auto-elimina. El código se muestra a continuación:

```
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO + 3, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO + 2, Task2, NULL);
  chThdCreateStatic(waTask3, sizeof(waTask3),NORMALPRIO + 1, Task3, NULL);
  t=micros()-ti;
  Serial.print((float)t/((float)nt));
  if (ip == 0)
    Serial.println(" Microsegundos SIN interbloqueo");
  else
    Serial.println(" Microsegundos CON interbloqueo");
  Serial.println("\nIMPORTANTE: Para obtener el tiempo final reste, ip = 0; e ip = 1;");
  chThdExit(0);
}
```

Las tres tareas para la comparativa están bajo un bucle while infinito que se rompe cuando la variable controladora de iteraciones, que es única para cada tarea y que se incrementa en cada iteración de la tarea respectiva, alcance el valor 10000, tras haber producido el mismo número de iteraciones. La terminación de este bucle while se verifica siempre al principio de cada tarea y se rompe auto-eliminando la tarea con la instrucción chThdExit ().

La tarea1 empieza haciendo una pausa de 1 milisegundo con la instrucción chThdSleep (1) lo que permite que funcione la tarea2, que luego de incrementar una variable contador con paso 1 en un bucle FOR hasta un tick aumentado dividido por 4, que equivale a 393 iteraciones, hace una pausa de 1 milisegundo con la instrucción chThdSleep(1). En ese momento entra en funcionamiento la tarea3, ya que la tarea1 aún está en pausa al no cumplirse el milisegundo con las 393 iteraciones. La tarea3 toma el mútex de nombre mtx con la instrucción chMtxLock() y luego empieza un bucle FOR que incrementa la variable contador con paso 1 hasta un tick aumentado que se verá interrumpido, porque se terminará el milisegundo de pausa de la tarea1 que se activará para tomar el mútex con la instrucción chMtxLock() que al estar bloqueado por la tarea3, hace que la tarea1 se detenga y que entre en funcionamiento la tarea2 por lo que la tarea1 debe esperar a que la tarea2 dé paso a la tarea3 (produciéndose el interbloqueo). Al tener paso, la tarea3 soltará el mútex con la instrucción chMtxUnlock(), activando inmediatamente a la tarea1 que esperaba ese evento, la misma que inmediatamente luego de tomarlo, soltará el mútex. Durante el test, se repite por 10000 iteraciones todo el proceso que se ha descrito, desde la pausa de 1 milisegundo de la tarea1.

Cabe indicar que todo este proceso se realiza si el valor a la variable de control es 1 (ip=1). Esto se verifica únicamente en la tarea1, ya que cuando esta tarea toma el mútex es cuando se produce el interbloqueo. En el caso de que el valor de la variable sea 0 (ip=0) se hace el mismo proceso, pero sin tomar el mútex en la tarea1, por lo que el tiempo de ruptura del interbloqueo no aparece. Al final se restarán ambos tiempos con ip=1 y con ip=0 para obtener el resultado final de la comparativa. El código se muestra a continuación:

```
static WORKING_AREA(waTask3, 64);
static msg_t Task3(void *arg) {
  while(1)
  {
    if (i3== nt)
    {
      chThdExit(0);
    }
    chMtxLock(&mtx);
    for (x=0;x<tick_a;x++)
      a++;
    i3++;
```

```
    chMtxUnlock();
  }
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  while (1)
  {
    if (i2 == nt)
    {
      chThdExit(0);
    }
    for (y=1;y<=tick_a/4;y++)
      a++;
    chThdSleep(1);
    i2++;
  }
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  while(1)
  {
    if (i1== nt)
    {
      chThdExit(0);
    }
    chThdSleep(1);
    if (ip == 1)
    {
      chMtxLock(&mtx);
      chMtxUnlock();
    }
    i1++;
  }
}
```

En el apéndice I se muestra el código ChibiOS/RT completo utilizado en Arduino Due para esta comparativa Rhealstone de Deadlock Breaking Time.

4.2.5 Latencia de paso de mensaje entre tareas (Intertask Messaging Latency)

Para esta comparativa Rhealstone se entiende por latencia al retardo temporal producido por el envío y recepción de mensaje entre dos tareas de diferentes prioridades, por lo que en esta comparativa se mide lo antes mencionado. Para ello se define una cola para el mensaje que será un entero largo de 32 bits cuyo valor será 1234567 y luego se establecen dos tareas con diferente prioridad, estas tareas pasaran entre sí mensajes por 200000 iteraciones para obtener un tiempo medio. Para eliminar un tiempo que afecta al cálculo del tiempo de latencia de paso de mensaje, se midió y guardó en una variable (t) el tiempo que consume un bucle FOR sin realizar ninguna instrucción para las 200000 iteraciones, para luego restarlas al tiempo total; ya que el mencionado bucle es el que permite las iteraciones y no forma parte del tiempo a medir, por eso hay que restar del tiempo total, el tiempo que consume. Cabe indicar que se repite el mismo

proceso dos veces ya que hay dos tareas que son las que van a realizar la comparativa. El código se muestra a continuación:

```
ti=micros();
for(uint32_t x=1;x<=nt;x++)
  ;
for(uint32_t y=1;y<=nt;y++)
  ;
t=micros()-ti;
```

La tarea1, que tiene mayor prioridad que la tarea2, siempre recibe el mensaje mientras que la tarea2 siempre lo envía, al intentar recibir el mensaje la tarea1 se bloquea hasta recibirlo y eso da paso a la tarea2, la cual envía el mensaje y automáticamente entra a funcionar la tarea1, que recibe el mensaje que estaba esperando y da paso a una siguiente iteración que intenta recibir otro mensaje y otra vez se bloquea, entrando en funcionamiento una nueva iteración de la tarea2 que envía el mensaje y así se genera el mismo proceso descrito para 200000 iteraciones dadas por los bucles FOR. Una vez terminadas, las tareas se eliminan a sí mismas y se da paso al cálculo del tiempo, restándose los tiempos que afectan a lo que se quiere medir (Intertask Messaging Latency).

Al final al tiempo total obtenido en esta comparativa se le resta el tiempo obtenido en la comparativa de conmutación de tarea para el respectivo RTOS, debido a que la conmutación de tarea influye en este procedimiento para obtener la comparativa. Cabe indicar que todo lo descrito en esta comparativa se aplicó diez veces para ambos RTOS, generando siempre el mismo resultado de tiempo, el cual se midió en microsegundos (µs) y que se presenta en la tabla 6, donde se puede observar que se restó el valor de la conmutación de tareas (Cmt) a la prueba, mostrándose el valor final de la comparativa para cada RTOS en la columna Resultado respectivamente.

Comparativa	FreeRTOS			ChibiOS/RT		
	Prueba (µs)	Cmt (µs)	Resultado (µs)	Prueba (µs)	Cmt (µs)	Resultado (µs)
Latencia_msg(200000)	35,79	4,73	31,06	6,8	1,95	4,85

Tabla 6: Resultado de la aplicación de la comparativa Rhealstone – latencia de paso de mensaje entre tareas

Como cada RTOS tiene sus instrucciones y formas de trabajar distintas, se describe a continuación lo realizado específicamente para cada sistema en esta comparativa:

4.2.5.1 FreeRTOS

Se define una cola, con la instrucción xQueueCreate() cuyo nombre es cola (definido con la instrucción QueueHandle_t), para el mensaje que será el número 1234567 asignado en la variable mensaje_e y cuyo tamaño en Bytes se almacenará en la variable tc que es requerida por la instrucción xQueueCreate(). Luego se crean las dos tareas con diferente prioridad y una tercera tarea con prioridad más baja, que se activará terminadas las dos tareas, esta tercera tarea se activa para calcular en ella el tiempo final de la comparativa. Como FreeRTOS permite crear las tareas y estas sólo se activan cuando se emplea la instrucción vTaskStartScheduler() se toma el tiempo inicial en dónde se empiezan a ejecutar las tareas (ti) y luego se lo considera en el cálculo del tiempo final. El código se muestra a continuación:

```
tc=sizeof(mensaje_e);
cola = xQueueCreate(lc, tc);
xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,3,&tarea1);
xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
```

```
ti=micros();
vTaskStartScheduler();
```

La tarea1, que tiene mayor prioridad que la tarea2 y que siempre recibe el mensaje, lo hace con la instrucción xQueueReceive() mientras que la tarea2 que siempre envía el mensaje lo hace con la instrucción xQueueSendToBack(). Al intentar recibir el mensaje la tarea1 se bloquea hasta recibirlo y eso da paso a la tarea2, la cual envía el mensaje y automáticamente entra a funcionar la tarea1 que recibe el mensaje que estaba esperando y da paso a una siguiente iteración que intenta recibir otro mensaje y otra vez se bloquea, entrando en funcionamiento una nueva iteración de la tarea2 que envía el mensaje y así se genera el mismo proceso descrito para 200000 iteraciones dadas por los bucles FOR. Una vez terminadas, las tareas se eliminan a sí mismas con la instrucción vTaskDelete(). El código se muestra a continuación:

```
static void Task1(void *pvParameters) {
  for(uint32_t x=1;x<=nt;x++)
    xQueueReceive(cola,&mensaje_r,portMAX_DELAY);
  vTaskDelete(tarea1);
}
//---------------------------------------------------------------------------
static void Task2(void *pvParameters) {
  for(uint32_t y=1;y<=nt;y++)
    xQueueSendToBack(cola,&mensaje_e, portMAX_DELAY);
  vTaskDelete(tarea2);
}
```

Después de auto-eliminadas las tareas se da paso a la tercera tarea para el cálculo del tiempo parcial de esta comparativa (se establece como tiempo parcial porque finalmente se debe restar el tiempo obtenido en la comparativa de conmutación de tarea), es importante recalcar que este tiempo parcial es la división del tiempo neto para el número de iteraciones (200000) que hicieron ambas tareas. No se duplica ya que lo que se hace por iteración es que la tarea1 reciba un mensaje que envía la tarea2, por tanto ambas tareas mezclan sus iteraciones. Al final de esta tarea se elimina la cola y así mismo se auto-elimina la tarea. El código se muestra a continuación:

```
static void Resultado(void *pvParameters) {
  t=micros()-ti-t;
  Serial.print((float)t/((float)nt));
  Serial.println(" Microsegundos");
  vQueueDelete(cola);
  vTaskDelete(tarea3);
}
```

En el apéndice J se muestra el código FreeRTOS completo utilizado en Arduino Due para esta comparativa Rhealstone de latencia de paso de mensaje entre tareas.

4.2.5.2 ChibiOS/RT

Se define una cola, con la instrucción MAILBOX_DECL () cuya variable es cola, para el mensaje que será el número 1234567 asignado en la variable mensaje_e, cuyo tamaño en Bytes también se calcula ya que es requerido por la instrucción MAILBOX_DECL(). El código se muestra a continuación:

```
MAILBOX_DECL(cola,buffer,sizeof(mensaje_e));
```

Luego, en la tarea principal o conductora de ChibiOS/RT, se crean las dos tareas con diferente prioridad. Como en ChibiOS/RT al momento de crear una tarea se ejecuta automáticamente (ChibiOS/RT no permite crear todas las tareas y al final activarlas mediante una instrucción), se incluye el tiempo de definición de cada tarea como tal; por lo que para eliminar este aspecto que interfiere en el tiempo de la comparativa, se mide el tiempo de definir dos tareas auxiliares de diferente prioridad (th). Asimismo, se toma el tiempo inicial en donde se define y empiezan a ejecutar las tareas que trabajan con el mensaje (ti) y estos dos tiempos se consideran en el tiempo de la comparativa.

Una vez terminadas las tareas, estas se auto-eliminan y se vuelve a la tarea principal o conductora de ChibiOS/RT en donde se procede al cálculo del tiempo parcial de esta comparativa. Se establece como tiempo parcial porque finalmente se debe restar el tiempo obtenido en la comparativa de conmutación de tarea. Es importante recalcar que este tiempo parcial es la división del tiempo neto para el número de iteraciones (200000) que hicieron ambas tareas. No se duplica ya que lo que se hace por iteración es que la tarea1 reciba un mensaje que envía la tarea2, por tanto ambas tareas mezclan sus iteraciones. De igual forma al final esta tarea que es la principal de ChibiOS/RT se auto-elimina. El código se muestra a continuación:

```
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 2, Task1a, NULL);
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 1, Task2a, NULL);
  th=micros()-ti;
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO+2, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO+1, Task2, NULL);
  t=micros()-ti-t-th;
  Serial.print((float)t/((float)nt));
  Serial.println(" Microsegundos");
  chThdExit(0);
}
```

La tarea1, que tiene mayor prioridad que la tarea2 y que siempre recibe el mensaje, lo hace con la instrucción chMBFetch(); mientras que la tarea2, que siempre envía el mensaje, lo hace con la instrucción chMBPost. Al intentar recibir el mensaje, la tarea1 se bloquea hasta recibirlo y eso da paso a la tarea2, la cual envía el mensaje y automáticamente entra a funcionar la tarea1 que recibe el mensaje que estaba esperando y da paso a una siguiente iteración que intenta recibir otro mensaje y otra vez se bloquea, entrando en funcionamiento una nueva iteración de la tarea2 que envía el mensaje. Este proceso se repite por 200000 iteraciones dadas por los bucles FOR. Una vez terminadas, las tareas se eliminan a sí mismas con la instrucción chThdExit(). El código se muestra a continuación:

```
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  for(uint32_t x=1;x<=nt;x++)
    chMBFetch(&cola,&mensaje_r,TIME_INFINITE);
  chThdExit(0);
}
//---------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  for(uint32_t y=1;y<=nt;y++)
    chMBPost(&cola,mensaje_e,TIME_INFINITE);
  chThdExit(0);
}
```

En el apéndice K se muestra el código ChibiOS/RT completo utilizado en Arduino Due para esta comparativa Rhealstone de latencia de paso de mensaje entre tareas.

4.3 Resultados de las comparativas Rhealstone

La recopilación de los resultados obtenidos para cada una de las comparativas Rhealstone descritas anteriormente y que se aplicaron sobre los sistemas operativos de tiempo real FreeRTOS y ChibiOS/RT ejecutándose sobre la placa del Arduino Due, se muestran en la tabla 7. En esta tabla aparece el nombre de la comparativa y entre paréntesis el número de iteraciones que se ejecutó en cada una de ellas. La tabla 7 también muestra los tiempos que se registraron para cada comparativa, medidos en microsegundos (µs). Además, se puede observar que se restó el valor de la conmutación de tareas en las pruebas que se veían afectadas por ella, así como se restaron entre ellas las pruebas que se hacían sin y con la presencia de lo que se quería medir en la comparativa, mostrándose el valor final de cada comparativa en la columna Resultados de la tabla para ambos RTOS.

En la tabla 7, además aparece la columna Diferencia que muestra la diferencia de tiempo entre FreeRTOS y ChibiOS/RT para cada comparativa. Asimismo se muestra la reducción en porcentajes que logró el RTOS que registró el menor tiempo para cada comparativa. Cabe indicar que cada una de estas comparativas se realizaron diez veces generando siempre los mismos tiempos. La tabla 7 que se ha descrito se muestra a continuación:

Comparativa	FreeRTOS			ChibiOS/RT			Diferencia	
Conmutación(500000)	Resultado (µs)			Resultado (µs)			(µs)	Reduce (%)
	4,73			1,95			2,78	58,77
Expulsión(100000)	Prueba (µs)	Cmt (µs)	Resultado (µs)	Prueba (µs)	Cmt (µs)	Resultado (µs)	(µs)	Reduce (%)
	37,56	4,73	32,83	64,36	1,95	62,41	29,58	47,4
Espera_Sema(100000)	Sin (µs)	Con (µs)	Resultado (µs)	Sin (µs)	Con (µs)	Resultado (µs)	(µs)	Reduce (%)
	38,5	115,47	76,97	17,61	31,87	14,26	62,71	81,47
Interbloqueo(10000)	Sin (µs)	Con (µs)	Resultado (µs)	Sin (µs)	Con (µs)	Resultado (µs)	(µs)	Reduce (%)
	1311,35	1343,13	31,78	1262,8	1267,65	4,85	26,93	84,74
Latencia_msg(200000)	Prueba (µs)	Cmt (µs)	Resultado (µs)	Prueba (µs)	Cmt (µs)	Resultado (µs)	(µs)	Reduce (%)
	35,79	4,73	31,06	6,8	1,95	4,85	26,21	84,39

Tabla 7: Recopilación de resultados de la aplicación de cada una de las comparativas Rhealstone

4.4 Cálculo del valor único de rendimiento Rhealstone

Basado en los resultados obtenidos para cada RTOS en las diferentes comparativas Rhealstone realizadas en el presente trabajo, que se recopilan en la tabla 7, se ha procedido al cálculo del valor único de rendimiento Rhealstone (VURR), a través de la fórmula respectiva:

$$VURR = \left(\frac{T1+T2+T3+T4+T5}{5}\right)^{-1} \text{ Rhealstones/segundo}$$

Al estar este valor único expresado en Rhealstones/segundo, hay que convertir el tiempo registrado en las diferentes comparativas a segundo.

Calculando el valor único de rendimiento Rhealstone (VURR) para FreeRTOS se tiene:

- Tiempo de conmutación de tarea (T1) = 4.73 µs = $4.73 * 10^{-6}$ s.
- Tiempo de expulsión (T2) =32.83 µs = $3.283 * 10^{-5}$ s.
- Tiempo de espera de un semáforo (T3) = 76.97 µs = $7.697 * 10^{-5}$ s.

- Tiempo de ruptura de interbloqueo (T4) = 31.78 µs = 3.178 * 10^{-5} s.
- Latencia de paso de mensaje entre tareas (T5)= 31.06 µs = 3.106 * 10^{-5} s.

VURR FreeRTOS = $\left(\frac{4.73 * 10^{-6} + 3.283 * 10^{-5} + 7.697 * 10^{-5} + 3.178 * 10^{-5} + 3.106 * 10^{-5}}{5}\right)^{-1}$ Rhealstones/segundo
VURR FreeRTOS = 28189.66 Rhealstones/segundo

Calculando el valor único de rendimiento Rhealstone (VURR) para ChibiOS/RT se tiene:

- Tiempo de conmutación de tarea (T1) = 1.95 µs = 1.95 * 10^{-6} s.
- Tiempo de expulsión (T2) = 62.41 µs = 6.241 * 10^{-5} s.
- Tiempo de espera de un semáforo (T3) = 14.26 µs = 1.426 * 10^{-5} s.
- Tiempo de ruptura de interbloqueo (T4) = 4.85 µs = 4.85 * 10^{-6} s.
- Latencia de paso de mensaje entre tareas (T5)= 4.85 µs = 4.85 * 10^{-6} s.

VURR ChibiOS/RT = $\left(\frac{1.95 * 10^{-6} + 6.241 * 10^{-5} + 1.426 * 10^{-5} + 4.85 * 10^{-6} + 4.85 * 10^{-6}}{5}\right)^{-1}$ Rhealstones/segundo
VURR ChibiOS/RT = 56612.32 Rhealstones/segundo

4.5 Memoria Flash (espacio de programa) que ocupan las comparativas Rhealstone realizadas para los RTOS FreeRTOS y ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due

Los procesadores comúnmente usados en sistemas embebidos son microcontroladores con memorias de datos y de programa de pequeño tamaño. Esto implica que los RTOS que se diseñan para estos microcontroladores han de proporcionar los servicios y recursos típicos de un sistema operativo de tiempo real (soporte para ejecución concurrente de múltiples hebras, mecanismos de sincronización como mútex, semáforos, etc.) sobre un núcleo de tamaño mínimo y con una demanda de memoria muy pequeña.

Por tanto, aunque no es parte de las métricas Rhealstone, podría resultar interesante conocer el espacio que para cada uno de estos dos RTOS (FreeRTOS y ChibiOS/RT) se requiere para cargar cada una de las comparativas Rhealstone realizadas, en la memoria Flash (espacio de programa) del microcontrolador que para objeto del presente trabajo es el ATSAM3X8E del Arduino Due. Para conseguir lo antes indicado, se desarrollaron seis pruebas.

La primera prueba fue la de control y consistió en cargar en el microcontrolador del Arduino Due un sketch (programa escrito para Arduino) que sólo tenga la estructura de los procedimientos obligatorios de Arduino (setup() y loop()) sin ninguna instrucción más, ni librería de RTOS alguno, código del sketch que se muestra a continuación:

```
void setup() {
}

void loop() {
}
```

Desde la segunda a la sexta prueba (que se realizaron dos veces, una para cada RTOS) se cargó respectivamente, en el microcontrolador del Arduino Due, cada sketch de las compartivas Rhealstone correspondiente, códigos fuentes que se pueden encontrar en los apéndices B, C, D, E, F, G, H, I, J, K de este documento.

En todas estas pruebas, la metodología que se aplicó para obtener el resultado, consistió en cargar el sketch correspondiente mediante el IDE de Arduino y registrar el mensaje que apareció en la consola de texto del mencionado IDE, referente a la cantidad de bytes de memoria Flash ocupada por el sketch que se había cargado. Cabe indicar que el total de memoria flash que el microcontrolador ATSAM3X8E del Arduino Due tiene disponible es de 512KB equivalente a 524288 Bytes. Un ejemplo de lo antes descrito se muestra en la figura 12 que fue lo que apareció en la consola de texto cuando se cargó el sketch de la primera prueba, aquí se puede observar que esta prueba ocupa 16096 bytes del total de memoria Flash disponible (524288 Bytes). De igual forma se procedió con las restantes pruebas.

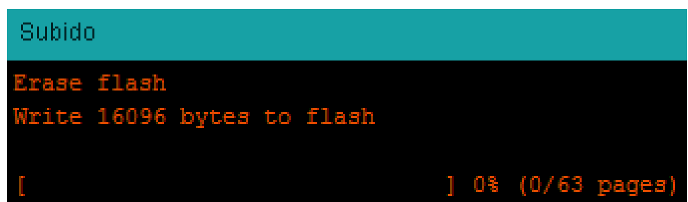

Figura 12: Memoria Flash que ocupa un sketch con la estructura mínima sin RTOS

La tabla 8 muestra todos los resultados obtenidos en las pruebas realizadas en cuanto a la memoria Flash necesaria para cargar las comparativas Rhealstone efectuadas para los RTOS FreeRTOS y ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due. En esta tabla aparece el nombre de la prueba que se identifica por P1, P2,…P6 para la prueba 1, prueba 2,… prueba 6, respectivamente; acompañada por el tipo de comparativa Rhealstone al que se refiere la prueba, a excepción de la prueba 1 que es la de control y por tanto sólo tiene la estructura de los procedimientos obligatorios de Arduino (setup() y loop()) sin ninguna instrucción más.

Luego se muestran los resultados que se han alcanzado en las mencionadas pruebas expresadas en Bytes para los dos RTOS, nótese que cuando se hace referencia a P1 + P2 significa que es el valor real que se obtuvo de la prueba que equivale a la suma de la prueba 1 (donde se sube la estructura de los procedimientos obligatorio setup() y loop() de Arduino) más la prueba 2 (dónde se carga la comparativa de tiempo de conmutación de tareas para el RTOS correspondiente) por lo que hay que restar el valor obtenido en la prueba 1 para sacar el valor neto que ocupa de memoria Flash la comparativa de tiempo de conmutación de tareas, de igual manera se procede para las demás pruebas cuando se hace referencia P1 + P3, P1 + P4 … P1 + P6, siendo ahora P3 equivalente a la comparativa tiempo de expulsión y así sucesivamente.

Para los dos RTOS, la tabla 8 también muestra en porcentaje (%) la cantidad de memoria Flash que ocuparon las pruebas desarrolladas en relación al total máximo disponible que en el microcontrolador ATSAM3X8E del Arduino Due es de 512KB equivalente a 524288 Bytes. Así mismo, cuando se indica P1 + P2 se refiere al porcentaje de memoria Flash que ocupan al mismo tiempo los procedimientos obligatorio setup() y loop() de Arduno más el código de la comparativa de tiempo de conmutación de tareas para el RTOS correspondiente, por lo que si se desea conocer el porcentaje que corresponde únicamente a la carga en memoria Flash de la mencionada comparativa hay que restar el porcentaje de la prueba 1, lo mismo se da para las demás pruebas cuando se hace referencia P1 + P3, P1 + P4 … P1 + P6, siendo ahora P3 equivalente a la comparativa tiempo de expulsión y así sucesivamente.

Finalmente, en la tabla 8 aparece la columna Diferencia que muestra, para cada comparativa, la reducción en porcentajes (%) que logró el RTOS que necesitó menor cantidad de memoria Flash. Cabe indicar que este valor porcentual se lo registró en una sola columna ya que es el mismo con o sin la presencia de la prueba 1. La tabla 8 que se ha descrito se muestra a continuación:

Prueba	FreeRTOS					ChibiOS/RT					Diferencia %
	Resultados			Flash que se ocupa		Resultados			Flash que se ocupa		Ocupación
P1: setup() + loop()	P1 (Bytes)			P1 (%)	P1 (%)	P1 (Bytes)			P1 (%)	P1 (%)	(%)
	16096			3,07	3,07	16096			3,07	3,07	0
P2: Conmutación	P1 + P2 (Bytes)	P1 (Bytes)	P2 (Bytes)	P1 + P2 (%)	P2 (%)	P1 + P2 (Bytes)	P1 (Bytes)	P2 (Bytes)	P1 + P2 (%)	P2 (%)	(%)
	30588	16096	14492	5,83	2,76	23668	16096	7572	4,51	1,44	1,32
P3: Expulsión	P1 + P3 (Bytes)	P1 (Bytes)	P3 (Bytes)	P1 + P3 (%)	P3 (%)	P1 + P3 (Bytes)	P1 (Bytes)	P3 (Bytes)	P1 + P3 (%)	P3 (%)	(%)
	31504	16096	15408	6,01	2,94	24312	16096	8216	4,64	1,57	1,37
P4: Espera_Sema	P1 + P4 (Bytes)	P1 (Bytes)	P4 (Bytes)	P1 + P4 (%)	P4 (%)	P1 + P4 (Bytes)	P1 (Bytes)	P4 (Bytes)	P1 + P4 (%)	P4 (%)	(%)
	31620	16096	15524	6,03	2,96	25032	16096	8936	4,77	1,7	1,26
P5: Interbloqueo	P1 + P5 (Bytes)	P1 (Bytes)	P5 (Bytes)	P1 + P5 (%)	P5 (%)	P1 + P5 (Bytes)	P1 (Bytes)	P5 (Bytes)	P1 + P5 (%)	P5 (%)	(%)
	31968	16096	15872	6,1	3,03	24736	16096	8640	4,72	1,65	1,38
P6: Latencia_msg	P1 + P6 (Bytes)	P1 (Bytes)	P6 (Bytes)	P1 + P6 (%)	P6 (%)	P1 + P6 (Bytes)	P1 (Bytes)	P6 (Bytes)	P1 + P6 (%)	P6 (%)	(%)
	31396	16096	15300	5,99	2,92	24432	16096	8336	4,66	1,59	1,33

Tabla 8: Recopilación de resultados de las pruebas de memoria Flash necesaria para cargar las comparativas Rhealstone realizadas para los RTOS FreeRTOS y de ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due

CAPÍTULO V

CONCLUSIÓN Y TRABAJO FUTURO

5.1 Conclusión

Los sistemas operativos de tiempo real, que se ejecutan en sistemas embebidos, son útiles en aplicaciones dónde existen tareas o sucesos prioritarios, es decir, que tienen mayor prioridad de ejecución que otros. La principal aplicación de estos sistemas, sobre todo si se ejecutan en sistemas embebidos, es el desarrollo de sistemas de control de cualquier tipo, pudiendo ser sistemas relativamente simples monoobjetivo hasta sistemas de control multiobjetivo. Con frecuencia, un aspecto crítico del desarrollo de estos sistemas de control es la elección del RTOS a emplear, por lo que la realización de estudios comparativos, basados en métricas establecidas como la Rhealstone, resultan relevantes y de elevado interés práctico.

Se probaron dos RTOS (FreeRTOS y ChibiOS/RT) que son libres, de código abierto y muy populares para usarse en sistemas embebidos. Ambos se ejecutaron sobre una placa Arduino Due que integra un microcontrolador de 32 bits Atmel SAM3X8E ARM Cortex-M3. Las comparativas Rhealstone realizadas han permitido medir el rendimiento de estos dos RTOS en el Arduino Due para proporcionar información importante o como un punto de partida para los interesados en desarrollar proyectos de sistemas embebidos que se basen en estas herramientas.

La implementación de las comparativas Rhealstone para los RTOS FreeRTOS y ChibiOS/RT sobre el Arduino Due, resultaron factibles y en igualdad de condiciones, ya que ambos RTOS proporcionan los mismos servicios y recursos necesarios para soportar completamente las mencionadas comparativas a las que fueron sometidos.

En cuanto a los resultados que se obtuvieron en las comparativas Rhealstone realizadas y que se plasman en la tabla 7 se puede determinar que ChibiOS/RT resultó tener tiempos de respuestas menores que FreeRTOS a excepción de la prueba de tiempo de expulsión. Analizando la reducción en porcentaje de los tiempos de respuestas que se obtuvieron por comparativa, se puede decir:

- Para la comparativa de tiempo de conmutación de tarea hay una reducción del 58.77% del tiempo conseguido para esta comparativa por ChibiOS/RT frente a FreeRTOS.
- Para la comparativa de tiempo de expulsión hay una reducción del 47.4% del tiempo conseguido para esta comparativa por FreeRTOS frente a ChibiOS/RT. Siendo esta la única comparativa en la que FreeRTOS consigue menor tiempo de respuesta que ChibiOS/RT.
- Para la comparativa de tiempo de espera de un semáforo hay una reducción del 81.47% del tiempo conseguido para esta comparativa por ChibiOS/RT frente a FreeRTOS.
- Para la comparativa de tiempo de ruptura de interbloqueo hay una reducción del 84.74% del tiempo conseguido para esta comparativa por ChibiOS/RT frente a FreeRTOS.
- Para la comparativa de latencia de paso de mensaje entre tareas hay una reducción del 84.39% del tiempo conseguido para esta comparativa por ChibiOS/RT frente a FreeRTOS.

En lo referente, al valor único de rendimiento Rhealstone que se expresa en Rhealstones/segundo y que se calcula en base al tiempo obtenido y expresado en segundo para las diferentes comparativas Rhealstone realizadas para cada RTOS, se tiene que FreeRTOS alcanzó 28189.66 Rhealstones/segundo y ChibiOS/RT obtuvo 56612.32 Rhealstones/segundo, y ya que para este valor único el mejor RTOS es el que obtiene más Rhealstones/segundo, se puede decir que ChibiOS/RT superó a FreeRTOS en el balance general de todas las comparativas realizadas, con una diferencia de 28422.66 Rhealstones/segundo que equivale a aproximadamente el 100.83% de mejora de ChibiOS/RT en relación al valor alcanzado por FreeRTOS.

Cabe indicar que para el cálculo de este valor único se trata a cada resultado de tiempo de las comparativas Rhealstone realizadas como parámetros con el mismo nivel de importancia. Por lo que el valor resultante es útil para evaluar el rendimiento del RTOS en general, más allá de aplicaciones específicas donde diferentes aspectos de la comparativa podrían pesarse de forma diferente. Por ejemplo, si en la aplicación las interrupciones ocurren 5 veces más que la conmutación de tarea, el peso correspondiente debe ser 5 veces más grande; del mismo modo, si una comparativa no aparece en absoluto, el peso respectivo se fija en cero, por lo que si no hay paso de mensajes entre tarea realizada por la aplicación, el peso se debe establecer en cero. Y es así que basado en pesos, que indican el nivel de influencia de cada comparativa Rhealstone sobre la aplicación, que se calcula el denominado valor de rendimiento Rhealstone de una aplicación específica.

Por otro lado, en los resultados obtenidos de las pruebas de memoria Flash (espacio de programa) necesaria para cargar las comparativas Rhealstone realizadas para los RTOS FreeRTOS y de ChibiOS/RT en el microcontrolador ATSAM3X8E del Arduino Due, que se muestran en la tabla 8 y que en plataformas hardware como las de Arduino con poca memoria de programa resulta ser un factor importante a considerar, se ha determinado que en todas las comparativas realizadas, ChibiOS/RT resultó tener una demanda de memoria Flash más pequeña que FreeRTOS. Analizando la reducción en porcentaje del requerimiento de ocupación de la mencionada memoria, en relación al máximo disponible (512KB equivalente a 524288 Bytes), por comparativa se puede decir:

- Para cargar la comparativa de tiempo de conmutación de tarea en el Arduino Due con el RTOS FreeRTOS se requiere del 5.83% y con el RTOS ChibiOS/RT se requiere el 4.51% de la totalidad de memoria Flash. Si sólo se tiene en cuenta el código puro de la comparativa, sin considerar la carga de la estructura de los procedimientos obligatorio setup() y loop() para Arduino, se tiene que de la totalidad de memoria Flash se emplea el 2.76% para FreeRTOS y el 1.44% ChibiOS/RT. De todo esto se desprende que existe una diferencia en porcentaje de reducción del espacio utilizado de memoria Flash para esta comparativa, con o sin la estructura de los procedimientos obligatorio setup() y loop() para Arduino, de aproximadamente el 1.32% habiendo 6920 Bytes menos de carga a favor de ChibiOS/RT.
- Para cargar la comparativa de tiempo de expulsión en el Arduino Due con el RTOS FreeRTOS se requiere del 6.01% y con el RTOS ChibiOS/RT se requiere el 4.64% de la totalidad de memoria Flash. Ahora bien, si sólo se tiene en cuenta el código puro de la comparativa, sin considerar la carga de la estructura de los procedimientos obligatorio setup() y loop() para Arduino, se tiene que de la totalidad de memoria Flash se emplea el 2.94% para FreeRTOS y el 1.57% ChibiOS/RT, de lo que se desprende que existe una diferencia en porcentaje de reducción del espacio utilizado de memoria Flash para esta comparativa, con o sin la estructura de los procedimientos obligatorio setup() y loop() para Arduino, de aproximadamente el 1.37% habiendo 7192 Bytes menos de carga a favor de ChibiOS/RT.
- Para cargar la comparativa de tiempo de espera de un semáforo en el Arduino Due con el RTOS FreeRTOS se requiere del 6.03% y con el RTOS ChibiOS/RT se requiere el 4.77% de la totalidad de memoria Flash. Si sólo se tiene en cuenta el código puro de la comparativa, sin considerar la carga de la estructura de los procedimientos obligatorio setup() y loop() para Arduino, se tiene que de la totalidad de memoria Flash se emplea el 2.96% para FreeRTOS y el 1.7% ChibiOS/RT. De lo anterior se desprende que existe una diferencia en porcentaje de reducción del espacio utilizado de memoria Flash para esta comparativa, con o sin la estructura de los procedimientos obligatorio setup() y loop() para Arduino, de aproximadamente el 1.26% habiendo 6588 Bytes menos de carga a favor de ChibiOS/RT.
- Para cargar la comparativa de tiempo de ruptura de interbloqueo en el Arduino Due con el RTOS FreeRTOS se requiere del 6.1% y con el RTOS ChibiOS/RT se requiere el 4.72% de la totalidad de memoria Flash. Si sólo se tiene en cuenta el código puro de la comparativa, sin considerar la carga de la estructura de los procedimientos obligatorio setup() y loop() para Arduino, se tiene que de la totalidad de memoria Flash se emplea el 3.03% para FreeRTOS y el 1.65% ChibiOS/RT, de todo esto se desprende que existe una diferencia en porcentaje de reducción del espacio utilizado de memoria Flash para esta comparativa, con o sin la estructura de los procedimientos obligatorio setup() y loop()

para Arduino, de aproximadamente el 1.38% habiendo 7232 Bytes menos de carga a favor de ChibiOS/RT.

- Finalmente, para cargar la comparativa de latencia de paso de mensaje entre tareas en el Arduino Due con el RTOS FreeRTOS se requiere del 5.99% y con el RTOS ChibiOS/RT se requiere el 4.66% de la totalidad de memoria Flash, y si sólo se tiene en cuenta el código puro de la comparativa, sin considerar la carga de la estructura de los procedimientos obligatorio setup() y loop() para Arduino, se tiene que de la totalidad de memoria Flash se emplea el 2.92% para FreeRTOS y el 1.59% ChibiOS/RT. Existe, por tanto, una diferencia en porcentaje de reducción del espacio utilizado de memoria Flash para esta comparativa, con o sin la estructura de los procedimientos obligatorio setup() y loop() para Arduino, de aproximadamente el 1.33% habiendo 6964 Bytes menos de carga a favor de ChibiOS/RT.

En consecuencia, ChibiOS/RT genera programas más compactos y con menor demanda de memoria Flash que FreeRTOS.

5.2 Trabajo futuro

Una línea de trabajo futuro podría ser aplicar estas comparativas Rhealstone a otros microcontroladores incluidos en placas importantes que se usan actualmente en sistemas embebidos, con el objetivo de poder comparar, ya no sólo RTOS, sino también otros microcontroladores. Así mismo se podrían extender las comparativas a otros RTOS populares. Todo esto permitirá tener información de prestaciones referente a diferentes fabricantes de microcontroladores y/o RTOS que pueden ser determinantes para tomar las mejores decisiones a la hora de elegir, de acuerdo a lo que vaya a manejar, la mejor combinación RTOS - microcontrolador para el sistema embebido que se pretenda desarrollar.

Otra línea de trabajo futuro constituiría un estudio para determinar los tiempos de respuesta, soporte y recursos que tendrían FreeRTOS y/o ChibiOS/RT ejecutando en el microcontrolador del Arduino Due para aplicaciones de sistemas embebidos multiobjetivos. Ya que un RTOS al manejar tareas con prioridades puede facilitar el desarrollo de este tipo de sistemas que buscan encontrar un equilibrio óptimo para sus funciones objetivos en conflicto.

Por último, la caracterización de FreeRTOS y/o ChibiOS/RT ejecutándose en el microcontrolador del Arduino Due, para sistemas embebidos de alta confiabilidad basados en técnicas de redundancia, podría constituir otra línea de trabajo a futuro interesante. En este contexto sería relevante conocer tiempos de respuestas y el soporte que brindan estos RTOS cuando tienen que recuperar al sistema embebido de un estado de error mediante técnicas de recuperación basadas en redundancia.

REFERENCIAS

[1] T. N. Bao Anh y S.-L. Tan, *Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers,* 2009.

[2] Cartmanese, «Diseño y arquitectura de productos de software,» [En línea]. Available: http://cartmanese.wordpress.com/diseo-y-arquitectura-de-productos-de-software/. [Último acceso: 2014].

[3] O. ÖRNVALL, *Benchmarking Real-time Operating Systems for use in Radio Base Station applications,* Suecia, 2012.

[4] A. Cañón, «Sistemas Operativos de Tiempo Real o Embebidos,» 2014. [En línea]. Available: http://es.slideshare.net/alejandrpcaon/sx-embebidos. [Último acceso: 2014].

[5] J. D. Muñoz Frías, Sistemas Empotrados en Tiempo Real. Una introducción basada en FreeRTOS y en el microcontrolador ColdFire MCF5282, 2009.

[6] . M. Soloriio, «Diseño de software de arquitectura de tiempo real,» 09 05 2013. [En línea]. Available: http://metodoz.blogspot.com/2013_05_01_archive.html. [Último acceso: 2014].

[7] «QNX Neutrino RTOS,» [En línea]. Available: http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html. [Último acceso: 2014].

[8] River Wind, «VxWorks,» [En línea]. Available: http://www.windriver.com/products/vxworks.html. [Último acceso: 2014].

[9] Oracle Corporation, «ChorusOS 5.0,» [En línea]. Available: http://docs.oracle.com/cd/E19048-01/chorus5/index.html. [Último acceso: 2014].

[10] «eCos Home,» [En línea]. Available: http://ecos.sourceware.org/. [Último acceso: 2014].

[11] LYNX Software Technologies, «Real-Time Operating Systems and Virtualization for Safety and Security Applications,» [En línea]. Available: www.lynuxworks.com/rtos. [Último acceso: 2014].

[12] Real Time Engineers Ltd., «FreeRTOS,» [En línea]. Available: www.freertos.org. [Último acceso: 2014].

[13] T. Xu, *Performance benchmarking of FreeRTOS and its Hardware Abstraction,* 2008.

[14] G. Di Sirio, «ChibiOS/RT Homepage,» [En línea]. Available: www.chibios.org. [Último acceso: 2014].

[15] Wikipedia, «Arduino,» [En línea]. Available: http://es.wikipedia.org/wiki/Arduino. [Último acceso: 2014].

[16] «Arduino due,» [En línea]. Available: http://arduino.cc/en/Main/arduinoBoardDue. [Último acceso: 2014].

[17] E. Gómez Rangel, «Sistema de ventilación automatizado para transformadores en aceite tipo subestación,» 04 2014. [En línea]. Available: http://www.uteq.edu.mx/tesis/ITA/0201.pdf. [Último acceso: 2014].

[18] International Directory of Company Histories, «Atmel Corporation History,» [En línea]. Available: http://www.fundinguniverse.com/company-histories/atmel-corporation-history. [Último acceso: 2014].

[19] Atmel Corporation , «ATSAM3X8E Cortex-M3 MCU,» [En línea]. Available: http://www.atmel.com/devices/sam3x8e.aspx?tab=overview. [Último acceso: 2014].

[20] Atmel Corporation, «SAM3X/SAM3A Series Summary,» [En línea]. Available: http://www.atmel.com/Images/doc11057s.pdf. [Último acceso: 2014].

[21] K. Rabindra y K. Porter, «Rhealstone--A Real Time Benchmarking Proposal,» *Dr.Dobb's Journal,* 1989.

[22] J. A. Fernández Madrigal, «Introducción a los Sistemas en Tiempo Real,» 2002. [En línea]. Available: http://www.isa.uma.es/personal/jafma/docencia/str20012002/STR_introduccion.pdf. [Último acceso: 2014].

[23] A. Heursch, A. Horstkotte y H. Rzehak, «Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux 2.4,» de *Real-Time & Embedded Computing Conference*, Milán, 2001.

[24] K. Rabindra P., «Implementing the rhealstone real-time benchmark,» [En línea]. Available: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1990/9004/9004d/9004d.htm. [Último acceso: 2014].

[25] B. Greiman, «ChibiOS/RT 2.6.5 for Arduino AVR, Due, Teensy 3.0 and 3.1,» 11 08 2014. [En línea]. Available: https://github.com/greiman/ChibiOS-Arduino. [Último acceso: 2014].

[26] B. Greiman, «FreeRTOS 8.0.1 Arduino Libraries,» 14 08 2014. [En línea]. Available: https://github.com/greiman/FreeRTOS-Arduino. [Último acceso: 2014].

APÉNDICES

Apéndice A

Código que determina el número de iteraciones necesarias para que un bucle FOR incremente una variable contador con paso 1 en el tiempo que dura un tick de reloj (1 milisegundo)

```
uint32_t t,x,dif,v=1551,a=0;
void setup() {
  Serial.begin(9600);
  delay(1000);
}
void loop() {
  t = micros();
  for (x=0;x<v;x++)
    a++;
  dif = micros() - t;
  Serial.println(dif);
  delay(2000);
}
```

Apéndice B

Código FreeRTOS de la comparativa Rhealstone de tiempo de conmutación de tarea (Task-Switching Time)

```
#include <FreeRTOS_ARM.h>
//---------------------------------------------------------------------------
uint32_t t,ti,nt=500000;
TaskHandle_t tarea1,tarea2,tarea3;
//---------------------------------------------------------------------------
static void Resultado(void *pvParameters) {
  t=micros()-ti-t;
  Serial.print((float)t/((float)nt*2.0));
  Serial.println(" Microsegundos por tarea");
  vTaskDelete(tarea3);
}
//---------------------------------------------------------------------------
static void Task1(void *pvParameters) {
  for(uint32_t x=1;x<=nt;x++){
    taskYIELD();
  }
  vTaskDelete(tarea1);
}
//---------------------------------------------------------------------------
static void Task2(void *pvParameters) {
  for(uint32_t y=1;y<=nt;y++){
    taskYIELD();
  }
  vTaskDelete(tarea2);
}
```

```
//---------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  ti=micros();
  for(uint32_t x=1;x<=nt;x++){
    ;
  }
  for(uint32_t y=1;y<=nt;y++){
    ;
  }
  t=micros()-ti;
  xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,2,&tarea1);
  xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
  xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
  ti=micros();
  vTaskStartScheduler();
}
//---------------------------------------------------------------------------
void loop() {
}
```

Apéndice C

Código ChibiOS de la comparativa Rhealstone de tiempo de conmutación de tarea (Task-Switching Time)

```
#include <ChibiOS_ARM.h>
//---------------------------------------------------------------------------
uint32_t ti,t,th,nt=500000;
MUTEX_DECL (mtx);
//---------------------------------------------------------------------------
static WORKING_AREA(waTask1a, 64);
static msg_t Task1a(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
}
static WORKING_AREA(waTask2a, 64);
static msg_t Task2a(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
}
//---------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
  for(uint32_t x=1;x<=nt;x++){
    chThdYield();
  }
  chThdExit(0);
}
//---------------------------------------------------------------------------
```

```
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
  for(uint32_t y=1;y<=nt;y++){
    chThdYield();
  }
  chThdExit(0);
}
//------------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  ti=micros();
  for(uint32_t x=1;x<=nt;x++){
    ;
  }
  for(uint32_t y=1;y<=nt;y++){
    ;
  }
  t=micros()-ti;
  chMtxInit(&mtx);
  chBegin(chSetup);
}
//------------------------------------------------------------------------------
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 2, Task1a, NULL);
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 2, Task2a, NULL);
  th=micros()-ti;
  chMtxLock(&mtx);
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO+1, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO+1, Task2, NULL);
  chMtxUnlock();
  t=micros()-ti-t-th;
  Serial.print((float)t/((float)nt*2.0));
  Serial.println(" Microsegundos por tarea");
  chThdExit(0);
}
//------------------------------------------------------------------------------
void loop() {
}
```

Apéndice D

Código FreeRTOS de la comparativa Rhealstone de tiempo de expulsión (Preemption Time)

```
#include <FreeRTOS_ARM.h>
//------------------------------------------------------------------------------
uint32_t i,nt=100000,tick_a=1572,ti,t,a=0;
TaskHandle_t tarea1,tarea2,tarea3;
```

```
//----------------------------------------------------------------------------
static void Resultado(void *pvParameters) {
  t=micros()-ti-t;
  Serial.print((float)t/((float)nt));
  Serial.println(" Microsegundos");
  vTaskDelete(tarea3);
}
//----------------------------------------------------------------------------
static void Task2(void *pvParameters) {
  for(uint32_t x=1;x<=nt;x++)
    for(i=0;i<tick_a;i++)
      a++;
  vTaskDelete(tarea2);
}
//----------------------------------------------------------------------------
static void Task1(void *pvParameters) {
   for(uint32_t y=1;y<=nt;y++)
     vTaskDelay(1);
  vTaskDelete(tarea1);
}
//----------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  ti=micros();
  for(uint32_t x=1;x<=nt;x++)
    for(i=0;i<tick_a;i++)
      a++;
  for(uint32_t y=1;y<=nt;y++)
    ;
  t=micros()-ti;
  a=0;
  xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,3,&tarea1);
  xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
  xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
  ti=micros();
  vTaskStartScheduler();
}
//----------------------------------------------------------------------------
void loop() {
}
```

Apéndice E

Código ChibiOS de la comparativa Rhealstone de tiempo de expulsión (Preemption Time)

```
#include <ChibiOS_ARM.h>
//----------------------------------------------------------------------------
uint32_t i,nt=100000,tick_a=1572,ti,t,th,a=0;
//----------------------------------------------------------------------------
static WORKING_AREA(waTask1a, 64);
static msg_t Task1a(void *arg) {
```

```
}
static WORKING_AREA(waTask2a, 64);
static msg_t Task2a(void *arg) {
}
//--------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  for(uint32_t x=1;x<=nt;x++)
    for(i=0;i<tick_a;i++)
      a++;
  chThdExit(0);
}
//--------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  for(uint32_t y=1;y<=nt;y++)
    chThdSleep(1);
  chThdExit(0);
}
//--------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  ti=micros();
  for(uint32_t x=1;x<=nt;x++)
    for(i=0;i<tick_a;i++)
      a++;
  for(uint32_t y=1;y<=nt;y++)
    ;
  t=micros()-ti;
  a=0;
  chBegin(chSetup);
}
//--------------------------------------------------------------------------
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 4, Task1a, NULL);
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 3, Task2a, NULL);
  th=micros()-ti;
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO + 2, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO + 1, Task2, NULL);
  t=micros()-ti-t-th;
  Serial.print((float)t/((float)nt));
  Serial.println(" Microsegundos");
  chThdExit(0);
}
//--------------------------------------------------------------------------
void loop() {
}
```

Apéndice F

Código FreeRTOS de la comparativa Rhealstone de tiempo de espera de un semáforo (Semaphore Shuffle Time)

```
#include <FreeRTOS_ARM.h>
//---------------------------------------------------------------------------
uint32_t nt=100000,sema_ejec,ti,t;
TaskHandle_t tarea1,tarea2,tarea3;
SemaphoreHandle_t sema;
//---------------------------------------------------------------------------
static void Resultado(void *pvParameters) {
 t=micros()-ti;
 Serial.print((float)t/((float)nt)*2);
 if (sema_ejec == 0)
  Serial.println(" Microsegundos SIN semáforo");
 else
  Serial.println(" Microsegundos CON semáforo");
 Serial.println("\nIMPORTANTE: Para obtener en Microsegundos el tiempo de shuffle por semáforo debe
restar los resultados de las 2 pruebas, sema_ejec = 0; y sema_ejec = 1;");
 vTaskDelete(tarea3);
}
//---------------------------------------------------------------------------
static void Task2(void *pvParameters) {
 for(uint32_t x=1;x<=nt;x++)
 {
  if (sema_ejec == 1)
  {
   xSemaphoreTake(sema,portMAX_DELAY);
  }
  taskYIELD();
  if (sema_ejec == 1)
  {
   xSemaphoreGive(sema);
  }
  taskYIELD();
 }
 vTaskDelete(tarea2);
}
//---------------------------------------------------------------------------
static void Task1(void *pvParameters) {
 for(uint32_t y=1;y<=nt;y++)
 {
  if (sema_ejec == 1)
  {
   xSemaphoreTake(sema,portMAX_DELAY);
  }
  taskYIELD();
  if (sema_ejec == 1)
  {
   xSemaphoreGive(sema);
  }
  taskYIELD();
```

```
 }
 vTaskDelete(tarea1);
}
//------------------------------------------------------------------------------
void setup() {
 Serial.begin(9600);
 delay(1000);
 /***********************************************************************
  * Hay que ejecutar el programa 2 veces una con sema_ejec = 0 (sin semáforo)
  * y la otra con sema_ejec = 1 (con semáforo) y restar los tiempos obtenidos en
  * ambas pruebas que será el resultado final de este BENHCMARK
  ***********************************************************************/
 sema_ejec = 1;
 if (sema_ejec == 1)
  vSemaphoreCreateBinary(sema);
 xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,2,&tarea1);
 xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
 xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
 ti=micros();
 vTaskStartScheduler();
}
//------------------------------------------------------------------------------
void loop() {
}
```

Apéndice G

Código ChibiOS de la comparativa Rhealstone de tiempo de espera de un semáforo (Semaphore Shuffle Time)

```
#include <ChibiOS_ARM.h>
//------------------------------------------------------------------------------
uint32_t nt=100000,sema_ejec,ti,t,th;
BSEMAPHORE_DECL (sema,FALSE);
MUTEX_DECL (mtx);
//------------------------------------------------------------------------------
static WORKING_AREA(waTask1a, 64);
static msg_t Task1a(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
}
static WORKING_AREA(waTask2a, 64);
static msg_t Task2a(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
 chMtxLock(&mtx);
 chMtxUnlock();
 for(uint32_t x=1;x<=nt;x++)
```

```
  {
    if (sema_ejec == 1)
    {
      chBSemWaitTimeout (&sema,TIME_INFINITE);
    }
    chThdYield();
    if (sema_ejec == 1)
    {
      chBSemSignal(&sema);
    }
    chThdYield();
  }
  chThdExit(0);
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  chMtxLock(&mtx);
  chMtxUnlock();
  for(uint32_t y=1;y<=nt;y++)
  {
    if (sema_ejec == 1)
    {
      chBSemWaitTimeout (&sema,TIME_INFINITE);
    }
    chThdYield();
    if (sema_ejec == 1)
    {
      chBSemSignal(&sema);
    }
    chThdYield();
  }
  chThdExit(0);
}
//------------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  /*************************************************************************
   * Hay que ejecutar el programa 2 veces una con sema_ejec = 0 (sin semáforo)
   * y la otra con sema_ejec = 1 (con semáforo) y restar los tiempos obtenidos en
   * ambas pruebas que será el resultado final de este BENHCMARK
   *************************************************************************/
  sema_ejec = 1;
  if (sema_ejec == 1)
    chBSemInit(&sema, FALSE);
  chMtxInit(&mtx);
  chBegin(chSetup);
}
//------------------------------------------------------------------------------
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 2, Task1a, NULL);
```

```
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 2, Task2a, NULL);
  th=micros()-ti;
  chMtxLock(&mtx);
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO + 1, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO + 1, Task2, NULL);
  chMtxUnlock();
  t=micros()-ti-th;
  Serial.print((float)t/((float)nt)*2);
  if (sema_ejec == 0)
    Serial.println(" Microsegundos SIN semáforo");
  else
    Serial.println(" Microsegundos CON semáforo");
  Serial.println("\nIMPORTANTE: Para obtener en Microsegundos el tiempo de shuffle por semáforo debe restar los resultados de las 2 pruebas, sema_ejec = 0; y sema_ejec = 1;");
  chThdExit(0);
}
//------------------------------------------------------------------------------
void loop() {
}
```

Apéndice H

Código FreeRTOS de la comparativa Rhealstone de tiempo de ruptura de interbloqueo (Deadlock Breaking Time)

```
#include <FreeRTOS_ARM.h>
//------------------------------------------------------------------------------
uint32_t nt=10000,ip,x,y,tick_a=1572,ti,t,i1=0,i2=0,i3=0,a=0;
TaskHandle_t tarea1,tarea2,tarea3,tarea4;
SemaphoreHandle_t mtx;
//------------------------------------------------------------------------------
static void Resultado(void *pvParameters) {
  t=micros()-ti;
  Serial.print((float)t/((float)nt));
  if (ip == 0)
    Serial.println(" Microsegundos SIN interbloqueo");
  else
    Serial.println(" Microsegundos CON interbloqueo");
  Serial.println("\nIMPORTANTE: Para obtener en Microsegundos el tiempo de ruptura de interbloqueo debe restar los resultados de las 2 pruebas, ip = 0; e ip = 1;");
  vTaskDelete(tarea4);
}
//------------------------------------------------------------------------------
static void Task3(void *pvParameters) {
  while(1)
  {
    if (i3== nt)
    {
      vTaskDelete(tarea3);
    }
    xSemaphoreTake(mtx,portMAX_DELAY);
```

```
    for (x=0;x<tick_a;x++)
      a++;
    i3++;
    xSemaphoreGive(mtx);
  }
}
//--------------------------------------------------------------------------
static void Task2(void *pvParameters) {
  while (1)
  {
    if (i2 == nt)
    {
      vTaskDelete(tarea2);
    }
    for (y=1;y<=tick_a/4;y++)
      a++;
    vTaskDelay(1);
    i2++;
  }
}
//--------------------------------------------------------------------------
static void Task1(void *pvParameters) {
  while(1)
  {
    if (i1== nt)
    {
      vTaskDelete(tarea1);
    }
    vTaskDelay(1);
    if (ip == 1)
    {
      xSemaphoreTake(mtx,portMAX_DELAY);
      xSemaphoreGive(mtx);
    }
    i1++;
  }
}
//--------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);

/**********************************************************************************
   * Hay que ejecutar el programa 2 veces una con ip = 0 (sin interbloqueo)
   * y la otra con ip = 1 (con interbloqueo) y restar los tiempos obtenidos en
   * ambas pruebas que será el resultado final de este BENHCMARK

**********************************************************************************/
  ip = 1;
  mtx = xSemaphoreCreateMutex();
  xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,4,&tarea1);
  xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,3,&tarea2);
  xTaskCreate(Task3,"Task3",configMINIMAL_STACK_SIZE,NULL,2,&tarea3);
```

```
  xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea4);
  ti=micros();
  vTaskStartScheduler();
}
//------------------------------------------------------------------------------
void loop() {
}
```

Apéndice I

Código ChibiOS de la comparativa Rhealstone de tiempo de ruptura de interbloqueo (Deadlock Breaking Time)

```
#include <ChibiOS_ARM.h>
//------------------------------------------------------------------------------
uint32_t ti,t,nt=10000,ip,x,y,tick_a=1572,i1=0,i2=0,i3=0,a=0;
MUTEX_DECL (mtx);
//------------------------------------------------------------------------------
static WORKING_AREA(waTask3, 64);
static msg_t Task3(void *arg) {
  while(1)
  {
    if (i3== nt)
    {
      chThdExit(0);
    }
    chMtxLock(&mtx);
    for (x=0;x<tick_a;x++)
      a++;
    i3++;
    chMtxUnlock();
  }
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  while (1)
  {
    if (i2 == nt)
    {
      chThdExit(0);
    }
    for (y=1;y<=tick_a/4;y++)
      a++;
    chThdSleep(1);
    i2++;
  }
}
//------------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  while(1)
```

```
 {
  if (i1== nt)
  {
   chThdExit(0);
  }
  chThdSleep(1);
  if (ip == 1)
  {
   chMtxLock(&mtx);
   chMtxUnlock();
  }
  i1++;
 }
}
//------------------------------------------------------------------------------
void setup() {
 Serial.begin(9600);
 delay(1000);

/***************************************************************************************
  * Hay que ejecutar el programa 2 veces una con ip = 0 (sin interbloqueo)
  * y la otra con ip = 1 (con interbloqueo) y restar los tiempos obtenidos en
  * ambas pruebas que será el resultado final de este BENHCMARK

***************************************************************************************
*/
 ip = 1;
 chMtxInit(&mtx);
 chBegin(chSetup);
}
//------------------------------------------------------------------------------
void chSetup() {
 ti=micros();
 chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO + 3, Task1, NULL);
 chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO + 2, Task2, NULL);
 chThdCreateStatic(waTask3, sizeof(waTask3),NORMALPRIO + 1, Task3, NULL);
 t=micros()-ti;
 Serial.print((float)t/((float)nt));
 if (ip == 0)
  Serial.println(" Microsegundos SIN interbloqueo");
 else
  Serial.println(" Microsegundos CON interbloqueo");
 Serial.println("\nIMPORTANTE: Para obtener en Microsegundos el tiempo de ruptura de interbloqueo debe restar los resultados de las 2 pruebas, ip = 0; e ip = 1;");
 chThdExit(0);
}
//------------------------------------------------------------------------------
void loop() {
}
```

Apéndice J

Código FreeRTOS de la comparativa Rhealstone de latencia de paso de mensaje entre tareas (Intertask Messaging Latency)

```
#include <FreeRTOS_ARM.h>
//-----------------------------------------------------------------------------
uint32_t nt=200000,lc=10,tc,ti,t;
int32_t mensaje_r, mensaje_e=1234567;
TaskHandle_t tarea1,tarea2,tarea3;
QueueHandle_t cola;
//-----------------------------------------------------------------------------
static void Resultado(void *pvParameters) {
  t=micros()-ti-t;
  Serial.print((float)t/((float)nt));
  Serial.println(" Microsegundos");
  vQueueDelete(cola);
  vTaskDelete(tarea3);
}
//-----------------------------------------------------------------------------
static void Task1(void *pvParameters) {
  for(uint32_t x=1;x<=nt;x++)
    xQueueReceive(cola,&mensaje_r,portMAX_DELAY);
  vTaskDelete(tarea1);
}
//-----------------------------------------------------------------------------
static void Task2(void *pvParameters) {
  for(uint32_t y=1;y<=nt;y++)
    xQueueSendToBack(cola,&mensaje_e, portMAX_DELAY);
  vTaskDelete(tarea2);
}
//-----------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  ti=micros();
  for(uint32_t x=1;x<=nt;x++)
    ;
  for(uint32_t y=1;y<=nt;y++)
    ;
  t=micros()-ti;
  tc=sizeof(mensaje_e);
  cola = xQueueCreate(lc, tc);
  xTaskCreate(Task1,"Task1",configMINIMAL_STACK_SIZE,NULL,3,&tarea1);
  xTaskCreate(Task2,"Task2",configMINIMAL_STACK_SIZE,NULL,2,&tarea2);
  xTaskCreate(Resultado,"Resultado",configMINIMAL_STACK_SIZE,NULL,1,&tarea3);
  ti=micros();
  vTaskStartScheduler();
}
//-----------------------------------------------------------------------------
void loop() {
}
```

Apéndice K

Código ChibiOS de la comparativa Rhealstone de latencia de paso de mensaje entre tareas (Intertask Messaging Latency)

```
#include <ChibiOS_ARM.h>
//--------------------------------------------------------------------------------------
uint32_t nt=200000,ti,t,th;
int32_t mensaje_r, mensaje_e=1234567, buffer[10];
MAILBOX_DECL(cola,buffer,sizeof(mensaje_e));
//--------------------------------------------------------------------------------------
static WORKING_AREA(waTask1a, 64);
static msg_t Task1a(void *arg) {
}
static WORKING_AREA(waTask2a, 64);
static msg_t Task2a(void *arg) {
}
//--------------------------------------------------------------------------------------
static WORKING_AREA(waTask1, 64);
static msg_t Task1(void *arg) {
  for(uint32_t x=1;x<=nt;x++)
    chMBFetch(&cola,&mensaje_r,TIME_INFINITE);
  chThdExit(0);
}
//--------------------------------------------------------------------------------------
static WORKING_AREA(waTask2, 64);
static msg_t Task2(void *arg) {
  for(uint32_t y=1;y<=nt;y++)
    chMBPost(&cola,mensaje_e,TIME_INFINITE);
  chThdExit(0);
}
//--------------------------------------------------------------------------------------
void setup() {
  Serial.begin(9600);
  delay(1000);
  ti=micros();
  for(uint32_t x=1;x<=nt;x++)
    ;
  for(uint32_t y=1;y<=nt;y++)
    ;
  t=micros()-ti;
  chBegin(chSetup);
}
//--------------------------------------------------------------------------------------
void chSetup() {
  ti=micros();
  chThdCreateStatic(waTask1a, sizeof(waTask1a),NORMALPRIO + 2, Task1a, NULL);
  chThdCreateStatic(waTask2a, sizeof(waTask2a),NORMALPRIO + 1, Task2a, NULL);
  th=micros()-ti;
  ti=micros();
  chThdCreateStatic(waTask1, sizeof(waTask1),NORMALPRIO+2, Task1, NULL);
  chThdCreateStatic(waTask2, sizeof(waTask2),NORMALPRIO+1, Task2, NULL);
  t=micros()-ti-t-th;
```

```
  Serial.print((float)t/((float)nt));
  Serial.println(" Microsegundos");
  chThdExit(0);
}
//------------------------------------------------------------------------------
void loop() {
}
```