Universidad Central “Marta Abreu” de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

Sistemas en Tiempo Real para Sistemas Embebidos

Autor: Javier Alejandro Ojeda Rodríguez

Tutor: Dr.C. René González Rodríguez

Santa Clara

2017

"Año 59 de la Revolución"

Universidad Central “Marta Abreu” de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

Sistemas en Tiempo Real para Sistemas Embebidos

Autor: Javier Alejandro Ojeda Rodríguez

Email: jojeda@uclv.cu

Tutor: René González Rodríguez

Email: voltusv@gmail.com

Consultante: Dr.C. Luis Hernández Santana

Santa Clara

2017

" Año 59 de la Revolución "

PENSAMIENTO

Mantenerse ocupado en algo que es completamente inútil cuando se puede hacer algo útil, entender lo más simple cuando se tiene la inteligencia para intentar lo difícil, es sencillamente quitarle todo el talento a su dignidad.

José Martí

DEDICATORIA

A mis padres Raúl Ojeda César y Nidia Rodríguez Romero.

A mi abuela Hilda Romero López.

AGRADECIMIENTOS

A mis tías Vilma y Neisy que siempre me apoyaron y depositaron su confianza.

A mi tutor René que siempre buscaba el tiempo para ayudarme a avanzar en el tema e inspirarme confianza y seguridad.

A mi hermano Jasiel, mi primo Lean y a los demás familiares que siempre me apoyaron.

A mis amigos y compañeros con los que compartí durante estos 5 años de la carrera.

A todos los profesores que formaron parte en mi preparación.

RESUMEN

Los sistemas operativos de tiempo real (RTOS, por sus siglas en inglés) son plataformas en los cuales la importancia no la tiene el usuario, sino los procesos. Generalmente estos sistemas suelen tener la misma arquitectura que un sistema operativo convencional, su diferencia radica en que proporcionan mayor prioridad a los elementos de control y procesamiento que son utilizados para ejecutar las tareas.

Con este trabajo se pretende investigar las tendencias de los RTOS que más influencia tienen en la actualidad, principalmente aquellos de código abierto basados en Linux. Para el desarrollo del trabajo se realizó un profundo análisis bibliográfico sobre los principales RTOS que poseen posiciones privilegiadas en el mercado y se mencionan algunos intentos de desarrollo e implementación de este tipo de sistemas en Cuba. También se hace un análisis comparativo de RTOS basados en Linux en cuanto a latencia que, aunque no es el único factor que influye en la selección de un RTOS si juega un papel importante puesto que permite conocer si el sistema es determinista.

En un último capítulo se muestra un algoritmo de control con exigencias de tiempo en sistemas embebidos que para el estudio en cuestión se utilizó un microcontrolador denominado Raspberry Pi.

TABLA DE CONTENIDOS

INTRODUCCIÓN

Los sistemas de tiempo real son sistemas informáticos que se encuentran en multitud de aplicaciones, desde la electrónica de consumo hasta el control de complejos procesos industriales. Están presentes en prácticamente todos los aspectos de la sociedad como, teléfonos móviles, automóviles, control de tráfico, ingenios espaciales, procesos automáticos de fabricación, producción de energía, aeronaves, etc. Además, el auge de los sistemas de tiempo real está en constante aumento, ya que cada vez más máquinas se fabrican incluyendo un número mayor de sistemas controlados por computador. Un ejemplo cercano es la industria del automóvil, ya que un vehículo en la actualidad incluye alrededor de una docena de estos automatismos (ABS[1], *airbag*, etc). Otro ejemplo cotidiano son los electrodomésticos de nueva generación, que incluyen sistemas de tiempo real para su control y temporización. Hoy día son tantas las aplicaciones de estos sistemas que su número duplica actualmente al de los sistemas informáticos "convencionales" o de propósito general. Las previsiones son que esta diferencia vaya en constante aumento, debido fundamentalmente al elevado crecimiento de la automatización. La mayoría de los sistemas de tiempo real son sistemas empotrados (Douglass, 2002) y suelen tener restricciones adicionales en cuanto al uso de recursos computacionales con respecto a otros tipos de sistemas informáticos. Además, suelen tener requisitos de seguridad y fiabilidad más severos, ya que si el sistema falla puede ocasionar pérdidas económicas o incluso humanas en dependencia de la aplicación. La rápida evolución de los componentes informáticos, su elevada velocidad y su reducido coste, cada

[1] *Antilock Brake System*: Sistema antibloqueo de ruedas, dispositivo diseñado para dar adherencia a los neumáticos durante el proceso de frenado.

día hace posible el desarrollo de aplicaciones de software que anteriormente solo eran abordables directamente con el hardware. Esto aumenta la flexibilidad y las posibilidades de estos sistemas, pero se hace necesaria la aplicación de una teoría que garantice el correcto funcionamiento de estos (Crespo and Alonso, 2006).

Varios proyectos que se están realizando en Cuba y específicamente en la Facultad de Ingeniería Eléctrica de la Universidad Central "Marta Abreu" de Las Villas en el Grupo de Automática Robótica y Percepción (GARP, por sus siglas) requieren de sistemas de tiempo real para su ejecución. Hoy el país no cuenta con una definición de una plataforma de desarrollo de estos sistemas y no se cuentan con los recursos materiales para adquirir sistemas propietarios por lo que esta carencia constituye el problema científico del presente trabajo que se puede definir como:

Necesidad de uso de sistemas operativos de tiempo real (RTOS, por sus siglas en inglés) para aplicaciones de control en el GARP con dispositivos de bajo coste y sostenibilidad tecnológica.

Hipótesis:

La aplicación de un parche de tiempo real al sistema operativo Linux compilado para plataformas ARM[2], permitirá desarrollar aplicaciones con altas exigencias de tiempo como dispositivos empotrados, de forma tal que se garantice el acceso a las entradas y salidas, la ejecución de tareas y los requerimientos de control con restricciones temporales.

Objetivo general

Seleccionar una plataforma Software/Hardware que permita el desarrollo de sistemas en tiempo real en sistemas embebidos, brindando una herramienta de desarrollo para sistemas de control con este tipo de exigencias.

[2] ARM es una arquitectura RISC (*Reduced Instruction Set Computer*, Ordenador con Conjunto Reducido de Instrucciones) para microprocesadores.

Objetivos específicos

- Proponer un sistema operativo de tiempo real basado en software libre.
- Seleccionar una plataforma de hardware libre basada en arquitectura ARM para aplicaciones de tiempo real.
- Obtener un RTOS compilado para plataformas ARM.
- Validar un algoritmo de control con exigencias de tiempo que corroboren el funcionamiento del RTOS y de la plataforma utilizada.

Para lograr el cumplimiento de los objetivos anteriormente expuestos se propone ejecutar las siguientes tareas:

- Revisión de la literatura especializada en RTOS y Hardware libre basados en ARM.
- Realización de un estudio sobre el estado actual de los RTOS en el mundo y en Cuba.
- Estudio de la programación en tiempo real.
- Selección del RTOS y de la plataforma a utilizar para el desarrollo de la aplicación de control.
- Instalación del RTOS en la plataforma.
- Realización de pruebas de estrés.
- Validación de la aplicación de control.

Organización del informe

La investigación incluye tres capítulos, además de las conclusiones, recomendaciones,

referencias bibliográficas y anexos correspondientes. Los temas que se abordan en cada capítulo se encuentran estructurados de la forma siguiente:

Capítulo I: En el primer capítulo se realiza un análisis de la literatura consultada, donde se exponen las principales tendencias de los sistemas operativos de tiempo real de corte comercial y de los de código abierto basados en Linux y se hace una breve descripción de los

que más tienden a usarse en la actualidad. Se abordan los conceptos, características y propiedades de los sistemas de tiempo real, así como los principales parámetros que definen a las tareas y las principales políticas de planificación. En un último epígrafe se definen algunas de las características y requisitos que deben cumplir los sistemas operativos de tiempo real y se realiza una conclusión parcial del capítulo.

Capítulo II: En este capítulo se realiza una comparación de tres sistemas de tiempo real de código abierto basados en Linux. Se presentan las principales características de cada uno y se hace una breve comparación de pruebas realizadas por la comunidad científica con cada uno de ellos en la cual se analiza entre otras cosas la latencia como una de las características a tener en cuenta. En un último epígrafe se exponen algunos criterios a la hora de seleccionar un RTOS para una aplicación en tiempo real como conclusión parcial del capítulo.

Capítulo III: Se realiza una breve descripción de la plataforma de hardware que se usa. Se modela e implementa una tarea de control de una estructura cinemática con dos articulaciones controladas por motores por pasos en un RTOS con las exigencias de tiempo requeridas.

CAPÍTULO 1. GENERALIDADES DE LOS SISTEMAS DE TIEMPO REAL

En este capítulo se realiza un análisis de la bibliografía consultada donde se tienen en cuenta las principales tendencias de los RTOS comerciales y los de código abierto basados en Linux, se abordan los principales conceptos de los sistemas de tiempo real como propiedades, características y clasificación de los mismos. Se definen los principales parámetros que caracterizan a las tareas, así como las principales políticas de planificación. Además, se realiza una breve descripción de los sistemas operativos de tiempo real y se mencionan las principales características y requisitos de los mismos. Por último, se exponen algunas consideraciones a la hora de elegir un RTOS para el desarrollo de alguna aplicación con restricciones temporales.

1.1 Sistemas operativos

En general, un sistema operativo es responsable de la gestión de los recursos de hardware de una computadora, así como el alojamiento de aplicaciones que se ejecutan en el equipo (Tanenbaum and Andrew, 1992). Un sistema operativo de tiempo real (RTOS) realiza estas actividades, pero también está especialmente diseñado para ejecutar aplicaciones con una sincronización muy precisa y un alto grado de habilidad (Insaurralde, 2014). Esto puede ser muy importante en los sistemas de medición y automatización en donde el tiempo de inactividad es costoso o un retraso de algún programa podría causar un riesgo de seguridad. Para ser considerado en tiempo real, un sistema operativo debe tener un tiempo máximo conocido para cada una de las operaciones críticas que realiza (o por lo menos ser capaz de garantizar el máximo la mayor parte del tiempo). Algunas de estas operaciones incluyen llamadas al sistema operativo e interrupciones. Los sistemas operativos que pueden

garantizar un tiempo máximo para estas operaciones se refieren comúnmente como "tiempo real duro" (Sprunt et al., 1989), mientras que los sistemas operativos que solo puede garantizar un máximo la mayoría de las veces son referidos como sistemas en “tiempo real suave” (Buttazzo et al., 2005). En la práctica, estas categorías estrictas tienen utilidad limitada, cada solución de RTOS demuestra características de rendimiento único. Algunas de las características esenciales que presentan estos tipos de sistemas son el determinismo, sensibilidad, tolerancia a fallos, fiabilidad y el control de usuario.

1.2 Antecedentes, desarrollo de los sistemas operativos en tiempo real

Existen un gran número de núcleos o sistemas operativos de tiempo real que cumplen los requisitos anteriormente citados. Dentro de esta amplia gama, se pueden distinguir aquellos que son comerciales y los que son de código abierto. Respecto a los RTOS comerciales, en (Timmerman, 2000) se puede encontrar una amplia lista y una evaluación completa de ellos. En una publicación (Melanson and Tafazoli, 2003) se realizó una evaluación de distintos aspectos: núcleo, planificación, modelo de tareas, interfaz de programación de aplicaciones (API, por sus siglas en inglés) etc, puntuando distintas características. Los resultados de esta evaluación muestran a QNX/Neutrino y OS-9 como los mejores, seguidos de OSE, Lynx OS y VxWorks. Se debe destacar los desarrollos de núcleos comerciales que se han mantenido desde sus inicios como es el caso de QNX y VxWorks por ser los más relevantes. QNX es un sistema operativo de tiempo real de tipo Unix que cumple con la norma POSIX[3], desarrollado por *QNX Software Systems*, una empresa canadiense que fue adquirida por BlackBerry en abril de 2010, convirtiéndose así en subsidiaria de esta última (Krten, 2010). QNX está basado en una arquitectura de *kernel* micro-núcleo que proporciona características

[3] POSIX es el acrónimo de *Portable Operating System Interface*, y X viene de UNIX como seña de identidad de la API. POSIX es una norma escrita por la IEEE. Dicha norma define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos (o *"shell"*), y programas de utilidades comunes para apoyar la portabilidad de las aplicaciones a nivel de código fuente. El término fue sugerido por Richard Stallman en la década de 1980.

de estabilidad avanzadas de memoria protegida frente a fallos de dispositivos y aplicaciones, está disponible para las arquitecturas x86, MIPS, PowerPC, SH4 ARM, StrongARM, xScale y BlackBerry Playbook. En el caso de VxWorks (Barabanov, 1997) inició a finales de 1980 como un conjunto de mejoras para un simple RTOS llamado VRTX vendidos por *Sistemas Ready*. *Wind River* adquirió los derechos para distribuir VRTX y añadir mejoras significativas, entre otras cosas, un sistema de archivos y un entorno de desarrollo integrado. En 1987, anticipándose a la terminación de su contrato de distribuidor de *Sistemas Ready, Wind River* desarrolló su propio núcleo para reemplazar VRTX dentro VxWorks. Entre las principales características distintivas destacan la compatibilidad con POSIX, el tratamiento de memoria y la característica de multiprocesador. Se usan generalmente en sistemas embebidos e incluye la familia de CPUs x86, MIPS, PowerPC, SH-4, ARM, StrongARM y xScale. LynxOS es otro de los sistemas operativos en tiempo real propietario de *Lynx Software Technologies* (anteriormente "LynuxWorks"), este presenta una interface de sistema operativo portable, y más recientemente, compatibilidad con Linux (Saavedra, 2015). Las primeras versiones de LynxOS fueron escritas en 1986 en Dallas, Texas, por Mitchell Bunnell y hecha a la medida para el procesador Motorola 68010. En 1988-1989, LynxOS fue portado a la arquitectura 80386 de Intel. Los componentes de LynxOS están diseñados para determinismo absoluto (rendimiento en tiempo real duro), lo que significa que responden dentro de un período de tiempo conocido. Los tiempos de respuesta predecibles están aseguradas incluso en presencia de E/S pesada debido al modelo de subprocesos de un solo núcleo, lo que permite a las rutinas de interrupción ser extremadamente cortas y rápidas. Entre las características de Lynx conocido también de esta manera destacan: nuevos controladores de seguridad al igual que nuevos compiladores y depuradores, soporte a POSIX mejorado y aumento del apoyo de memoria. En cuanto a los sistemas operativos de tiempo real basados en código abierto, son varios los que existen.

En (Ripoll et al., 2002) se puede encontrar una lista de criterios y una comparación de las características de los basados en Linux con los más relevantes comerciales. De entre los RTOS de código abierto basados en Linux destacan RTLinux-GPL y RTAI, que incluyen el núcleo como módulo en el sistema operativo Linux permitiendo que coexistan un entorno de ejecución de tiempo real y otro, Linux, de propósito general. Otros, como KURT y TimeSysLinux, añaden mejoras al núcleo estándar de Linux para reducir la latencia e

incrementar la resolución de los relojes y planificación de tiempo real. De los no basados en Linux destacan principalmente RTEMS y eCOS. Con relación a los sistemas operativos de código abierto basados en Linux, RTLinux y RTAI *(Real Time Application Interface)* han sido desde los inicios los más destacados. RTLinux es un sistema operativo de tiempo real multitarea que ejecuta Linux como un hilo de ejecución de menos prioridad que las tareas de tiempo real (Brito and Perdomo, 2011). Con este diseño, las tareas de tiempo real y los manejadores de interrupciones nunca se ven retrasados por operaciones que no son de tiempo real. RTLinux nació del trabajo de Michael Barabanov y Victor Yodaiken en New México Tech, que posteriormente fundaron FSM Labs ofreciendo soporte técnico. RTLinux se distribuye bajo *la "GNU Public License"* y las primeras versiones de RTLinux ofrecían un API muy reducido sin tener en cuenta ninguno de los estándares de tiempo real: *POSIX Real-Time extensions, PThreads, etc.* A partir de la versión 2.0 Victor Yodaiken decide reconvertir el API original a otro que fuera "compatible" con el API de POSIX *Threads*. Actualmente *Wind River* tras absorber *FSM Labs* en 2007 es la propietaria de RTLinux, recientemente ha liberado una nueva versión de *RTLinuxFree* con soporte para la serie 2.6 del *kernel* de Linux. *RTLinuxFree* proporciona el API de POSIX *threads*, manejadores de interrupciones, métodos de sincronización y mecanismos de comunicación entre tareas de tiempo real y tareas no críticas (Brito and Perdomo, 2011). Por otra parte, RTAI es una implementación de Linux con licencia GNU/GPL para tiempo real basada en RTLinux que añade un *kernel* en tiempo real al *kernel* de Linux y trata a éste como una tarea de menor prioridad (Brito and Perdomo, 2011). Añade además una serie de mecanismos de comunicación entre procesos y otros servicios de tiempo real. Aunque RTAI proporciona servicios en tiempo real, conserva las características y los servicios de Linux estándar, proporcionando además un módulo llamado LXRT que facilita el desarrollo de aplicaciones en tiempo real en el espacio de usuario. Algunas de las características de RTAI es que proporciona servicios de manejo de memoria y *threads* compatibles con POSIX. La característica más destacable de RTAI es que no se trata de un sistema operativo de tiempo real en sí, sino de un conjunto de módulos que se instalan en el *kernel* de Linux además de un desarrollador para implementar más módulos (Dozio and Mantegazza, 2003). Con ello se consigue dotar a un sistema operativo fiable y estable con herramientas no implementadas de tiempo real como es Linux de una serie de rutinas para que pueda tener capacidades de tiempo real.

En Cuba se reporta un sistema operativo de tiempo real en la década de los 90 denominado SOTRE que constituyó una innovación tecnológica en su tiempo con el cual se desarrollaron diversas aplicaciones en tiempo real. Este sistema corría sobre MS-DOS. También se tuvo la idea de poder hacer sistemas de tiempo real sobre Windows, desarrollando un dispositivo virtual que permitiera atender una interrupción generada por una tarjeta externa o atendiendo el reloj de la PC, reprogramando el 8253 (*Timer* programable) a la frecuencia deseada. Para ello se pensó en el trabajo con un VxD que no es más que un controlador de dispositivo virtual el cual tiene acceso a la memoria del *kernel* y a todos los procesos en ejecución, así como acceso directo al hardware (González et al.). Este sistema se ejecuta bajo los sistemas operativos Windows 3.x, Windows 95, Windows 98 y Windows Me. En el país se reportan varias instituciones que tienen aplicaciones de tiempo real sobre Windows como es el caso del Centro de Meteorología de Camagüey donde se diseñó un sistema para controlar los movimientos de un radar y monitorear las señales captadas. El Grupo de Investigaciones Mecatrónicas GIMAS(G+) de la Universidad Central de Las Villas reporta un software para la simulación y control de robots industriales desde Windows, un sistema de adquisición de datos para diagnosticar defectos en las soldaduras muestreando corriente y voltaje del arco eléctrico a alta frecuencia (González et al.).

1.3 Conceptos y características importantes de los sistemas de tiempo real

Aunque el término tiempo real se utiliza con excesiva frecuencia para un gran número de aplicaciones que tienen una respuesta rápida, la correcta definición de este tipo de sistemas se puede enunciar como sistemas informáticos en los que la respuesta de la aplicación ante estímulos externos debe realizarse dentro de un plazo de tiempo establecido (Crespo and Alonso, 2006).

No es suficiente que las acciones del sistema sean correctas lógicamente, sino que, además, es necesario que se produzcan en el instante adecuado (antes de un tiempo máximo o *deadline*) (Kopetz, 1997).

Unas características importantes de los sistemas en tiempo real son:

Determinismo: Conocer exactamente cómo se comporta el entorno del sistema. Tener seguridad que no van a aparecer situaciones a las que el sistema no pueda responder adecuadamente (Puente and Antonio, 2000).

Comportamiento predecible: Saber cómo se comporta el sistema de manera que no pueda aparecer ninguna situación que altere el comportamiento temporal. Esta característica permite conocer a priori cuál va a ser el comportamiento del sistema en las peores condiciones y poder analizar los tiempos de respuesta del sistema (Vivancos et al., 1997).

En los sistemas de tiempo real existen otras características comunes que conviene destacar:

Concurrencia: En general, un mismo sistema ha de responder a distintos estímulos realizando distintos procesos ligados entre sí o independientes. Se deben realizar procesos de control concurrentes, por lo que es necesario disponer de herramientas que permitan programación concurrente.

Mantenimiento: Un problema importante en los sistemas de tiempo real es la labor de mantenimiento. Cualquier cambio requiere una nueva verificación detallada para asegurar la validez del comportamiento tanto funcional como temporal, pues la modificación de un determinado proceso puede afectar al comportamiento temporal del resto.

La forma de ejecutar las tareas concurrentes en los sistemas de tiempo real debe asegurar que se cumplen algunas propiedades, distintas de las que se exigen en otros tipos de sistemas, como podrían ser los sistemas de tiempo compartido. Véase en el cuadro comparativo mostrado en la Tabla 1.1 estas propiedades respecto a algunos factores:

Tabla 1.1 Comparación de sistemas de tiempo real y sistemas de tiempo compartido.

Factor	Sistema de tiempo real	Sistema de tiempo compartido
Capacidad	Garantía de plazos	Flujo
Reactividad	Tiempo de respuesta	Tiempo de respuesta medio
Sobrecarga	Estabilidad	Equidad

Garantía de plazos: un sistema de tiempo real funciona correctamente cuando los plazos de todas las tareas están garantizados, es decir, todas las tareas ejecutan su actividad dentro del plazo cada vez que se activan. En un sistema de tiempo compartido lo más importante es

asegurar un flujo (número de activaciones por segundo) sea lo más elevado posible. Se habla de *deadline* en los STR o de (MIPS) en los STC (Sistemas de Tiempo Compartido).

Tiempo de respuesta máximo: en un sistema de tiempo real se trata de acotar el tiempo de respuesta en el peor de los casos de todas las tareas. En un sistema de tiempo compartido, se trata de conseguir que el tiempo de respuesta medio sea lo más corto posible.

Estabilidad: Si a causa de una sobrecarga del sistema no se pueden ejecutar todas las tareas dentro del plazo, se debe garantizar que, al menos, un subconjunto de tareas críticas cumpla los plazos. En un sistema de tiempo compartido, el criterio es asegurar la equidad en la ejecución de las tareas (por ejemplo, que ninguna se vea postergada indefinidamente).

1.3.1 Clasificación

Atendiendo al nivel de exigencia temporal, se pueden distinguir cuatro tipos de sistemas:

Críticos: *(hard real time systems)* son aquellos en los que el tiempo de respuesta debe

garantizarse obligatoriamente. Una respuesta tardía puede tener consecuencias fatales. Ej: Sistema de navegación de un avión.

Esenciales: *(soft real time systems)* son aquellos sistemas con restricciones de tiempo en las que una respuesta tardía no produce graves daños, pero sí un deterioro del funcionamiento global. Ej: Sistema de comunicaciones (no se debe producir sobrecarga en sistemas asíncronos, ni desbordamiento negativo en sistemas síncronos). Sistema de videoconferencia.

Incrementales: la calidad de la respuesta obtenida depende del tiempo disponible para su cálculo. Si se les da más tiempo la respuesta mejora. Ej: Algoritmo de cálculo iterativo. Programa de ajedrez.

No esenciales: corresponden con las tareas sin restricciones temporales.

Otros criterios de clasificación:

Atendiendo a la arquitectura hardware utilizada:

Propietarios: Desarrollados para un hardware específico y con un desarrollo particular para cada aplicación.

Abiertos: Basados en sistemas abiertos, incorporando estándares industriales: microprocesadores estándar, sistemas operativos estándar, protocolos de comunicaciones estándar y buses estándar.

Atendiendo a la arquitectura del sistema:

Centralizados: Los procesadores están localizados en un único nodo del sistema y la velocidad de comunicación entre procesadores es semejante a la velocidad del procesador.

Distribuidos: Los procesadores están situados en distintos puntos del sistema y la velocidad de comunicación entre procesadores es muy baja respecto a su velocidad de proceso.

Los sistemas operativos juegan un papel clave en la mayoría de los sistemas de tiempo real. En estos, el sistema operativo y la aplicación están fuertemente acoplados, mucho más que en los sistemas de tiempo compartido. Un sistema operativo en tiempo real debe ser capaz de responder a determinados eventos internos o externos en un margen de tiempo predecible. Esta es la diferencia de los sistemas de tiempo compartido, los cuales deben compartir los recursos del sistema (tales como CPU, memoria principal, y periféricos) igualmente entre un número de tareas en ejecución. En un sistema de tiempo compartido debidamente configurado, ninguna tarea debe quedar esperando indefinidamente un recurso del sistema. En un entorno de tiempo real, las tareas críticas deben recibir los recursos que necesiten en el momento que los soliciten, sin importarles el efecto que esto pueda tener sobre el resto de tareas en ejecución. Los dos elementos fundamentales en los sistemas operativos de tiempo real son:

- El planificador de tiempo real
- El *kernel* de tiempo real.

La función de un algoritmo de planificación es determinar, para un conjunto de tareas dado, como se deben planificar (es decir, la secuencia en que se van a activar y el tiempo que se le da a cada una) para que en la ejecución de las tareas se satisfagan las restricciones de tiempo, de precedencia y de uso de recursos que tienen definidas.

Los sistemas operativos multitarea de tiempo compartido como Linux funcionan repartiendo el tiempo en trozos que se suelen denominar rodajas. El planificador es una parte del sistema operativo que se activa al final de cada rodaja de tiempo (gracias a los impulsos de reloj

recibidos) y decide qué proceso de los que están activos pasa a ejecutarse en la siguiente rodaja (Iglesias, 1998).

Por lo general, se suele usar para repartir las rodajas de tiempo entre los procesos, una política llamada *round-robin*, en la que todos los procesos listos para ejecutar (es decir, que no están parados esperando el resultado de una operación de entrada/salida) se sitúan en una cola circular, de la que se van extrayendo y ejecutando cada uno (Corbet et al., 2005).

El otro elemento fundamental es el diseño del *kernel* de tiempo real. Un *kernel* de tiempo real debe ofrecer características y primitivas que permita una eficiente intercomunicación entre las tareas, sincronización, garantizar la respuesta a las interrupciones, un rápido y fiable sistema de ficheros, y un mecanismo eficiente de manejo de memoria. Además, las directivas deben tener un tiempo de ejecución predecible o ser reentrantes.

1.4 Tareas de un Sistema de Tiempo Real

Un sistema de tiempo real está caracterizado por varias tareas que se ejecutan sobre un procesador. Una tarea está caracterizada por los siguientes parámetros funcionales:

Período: Cada tarea es ejecutada de forma regular cada un intervalo de tiempo. Específicamente, cada período de una tarea Ti, denotado por Pi, es una secuencia de activaciones (a1, a2, a3, ..., an) cada una de ellas en un intervalo de tiempo. En concreto, la activación ak deberá ejecutarse dentro del intervalo denotado por $[(k-1)*Pi,\ k*Pi]$.

Plazo de entrega: El plazo de entrega de una tarea Ti, denotado por Di, es el intervalo de tiempo máximo que puede transcurrir entre el instante a partir del cual debe activarse y su finalización. Normalmente, el plazo de entrega de una tarea será igual al período. Esto quiere decir que cada activación de la tarea debe finalizar su ejecución antes del siguiente período. En algunos casos, dependiendo de las restricciones del sistema, el plazo de entrega será menor o mayor que el período.

Desfase Inicial: El desfase inicial de una tarea Ti denota el desfase que tiene el inicio de la primera activación de la tarea respecto al tiempo de inicio y se denota como ϕi.

Los siguientes parámetros dependen del procesador y del planificador.

Tiempo de cómputo: El tiempo de cómputo de una tarea Ti es el tiempo de CPU necesario, Ci, para completar su ejecución en cada una de las activaciones. Este tiempo depende de la complejidad del algoritmo de control y la velocidad del procesador. El tiempo de cómputo puede variar atendiendo a distintos aspectos: código con ejecución condicional que pueden consumir distinto tiempo o el subsistema de ejecución (memoria *cache y pipeline*). Esto hace que la ejecución de una activación cualquiera de la tarea Ti pueda variar entre dos límites $[e_i^-, e_i^+]$, que se corresponden con el mínimo y máximo tiempo de cómputo, respectivamente. El límite superior es el tiempo de peor caso (WCET) y es el que se usa para el análisis de la planificabilidad.

Retardo de inicio: Una actividad de una tarea Ti tiene un retardo de inicio que corresponde con el tiempo en el cual la actividad empieza a ser ejecutada. Este retraso viene determinado por el sistema de ejecución (granularidad del reloj para reconocer el inicio del período, la ejecución del planificador y la decisión del planificador sobre qué tarea debe ejecutarse en cada instante) y por las decisiones del planificador. Este retardo se puede modelar debido a los servicios del núcleo $r_{m_i n}$ y un intervalo $[r_i^-, r_i^+]$, además de la interferencia de otras tareas. Este término recibe en la literatura el nombre de *jitter* de entrada.

Tiempo de finalización: El tiempo de finalización es el tiempo en el cuál una actividad de una tarea Ti finaliza su ejecución. Este tiempo depende del instante de inicio de la ejecución de la tarea, el tiempo de cómputo necesario, los retrasos que la ejecución de la actividad ha podido sufrir al acceder a datos compartidos y la interferencia de otras tareas más prioritarias. El tiempo de finalización de una actividad se modela mediante un intervalo $[f_i^-, f_i^+]$, que delimita el retardo de finalización y se denomina *jitter* de salida (Crespo and Alonso, 2006).

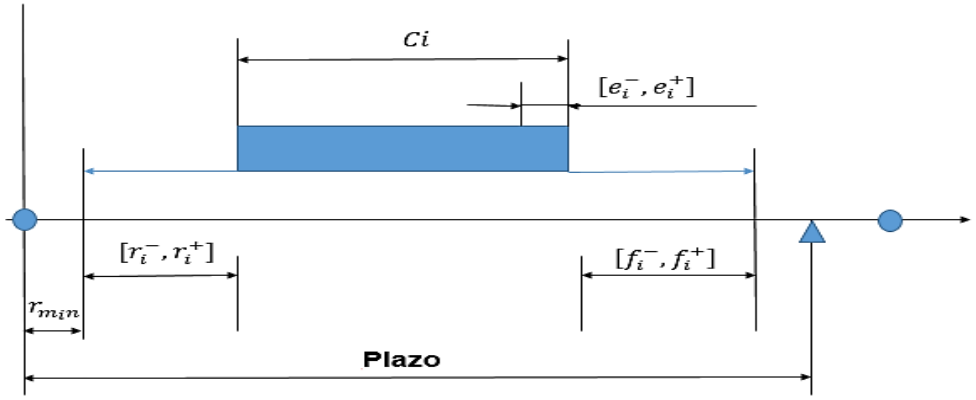


Figura 1.1 Parámetros de ejecución de una tarea.

En general, un sistema está formado por un conjunto de tareas: τ = [T1, T2, T3, T4, …Tn] donde n es la cardinalidad de τ. Cada tarea Ti, tiene las siguientes características:

Ti = (Ci, Di, Pi, ϕi) donde Ci es el tiempo de ejecución de la tarea, Di es el plazo de entrega, Pi es el período y ϕi es el desfase inicial.

1.4.1 Clasificación de las tareas

Las tareas suelen clasificarse en:

Periódicas*:* Se ejecutan en intervalos regulares de tiempo y se caracterizan por el tiempo de cómputo Ci, el período Pi, y el *deadline* Di.

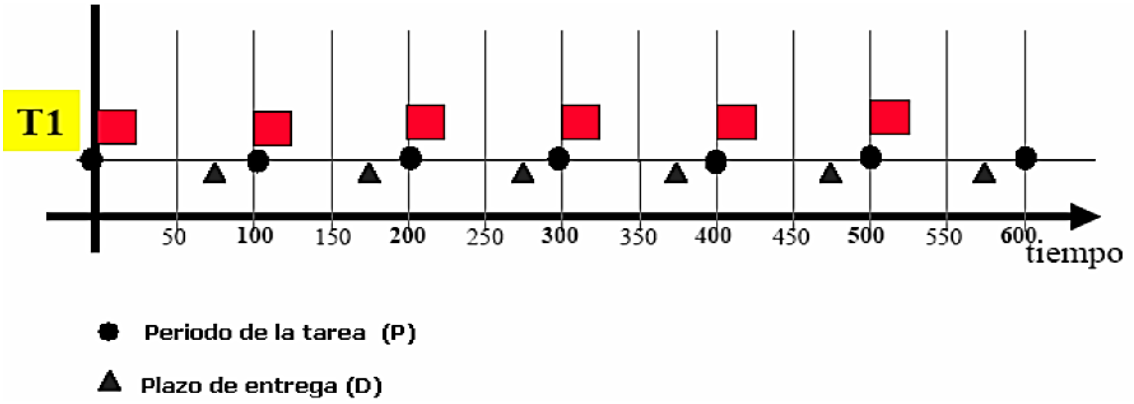


Figura 1.2 Esquema de tareas periódicas.

Aperiódicas: Se ejecutan en intervalos irregulares de tiempo y se caracterizan por el tiempo de cómputo Ci y el *deadline* Di.

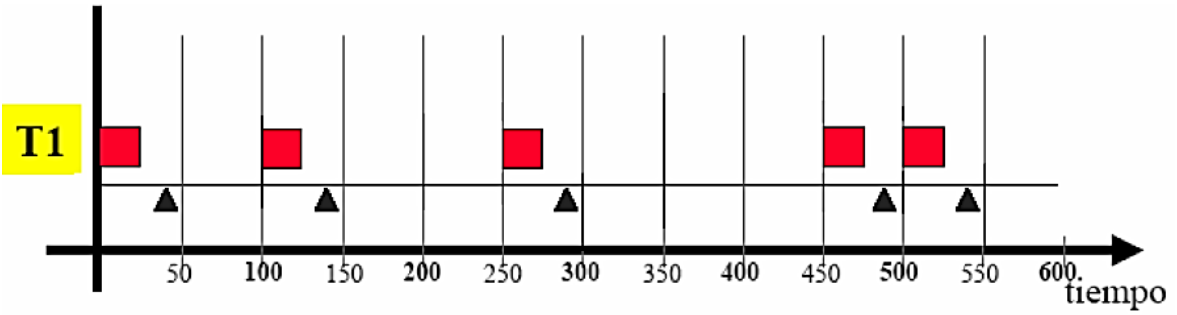


Figura 1.3 Esquema de tareas aperiódicas.

Esporádicas*:* se ejecutan ante la ocurrencia de estímulos externos o internos y se caracterizan por: evento, período mínimo entre dos eventos consecutivos, tiempo promedio y tiempo límite de ejecución.

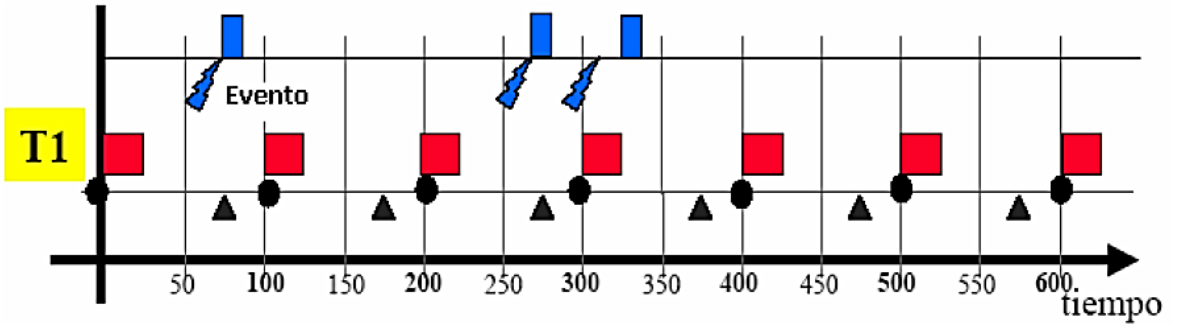


Figura 1.4 Esquema de tareas esporádicas.

1.5 Planificadores

Planificadores cíclicos: Una primera aproximación al diseño del sistema de tiempo real consiste en analizar la ejecución de las tareas considerando una ejecución estática decidida a priori. Este tipo de planificación se denomina cíclica (*Cyclic Executives*) y consiste en almacenar en una tabla el instante en el que debe ejecutarse cada una de las activaciones de las tareas o diseñar un secuenciador de las distintas actividades a realizar.

Este esquema resulta simple cuando el sistema de control tiene períodos iguales o múltiplos entre sí. En caso de períodos muy desiguales y gestión de eventos aperiódicos el cálculo de la tabla puede resultar extremadamente largo y complejo. En este mismo sentido, cuando pueden existir retrasos en la llegada de los datos del proceso, se produce una indeterminación que restringe el uso de este tipo de planificadores. En el caso más complejo, la búsqueda de

un plan requiere encontrar un intervalo submúltiplo del hiper-período (ciclo menor) que permita la ejecución de las tareas cumpliendo los plazos de estas. Este problema es computacionalmente complejo (*NP-Hard*) y requiere una serie de reglas que permitan reducir su complejidad. Un buen análisis del diseño de este tipo de planificadores y sus implicaciones se puede encontrar en (Baker and Shaw, 1989).

Para decidir una planificación cíclica hay que comenzar determinando la duración de los ciclos principal y secundario. Sea M la duración del ciclo principal y m la del ciclo secundario. Entonces, tenemos que la duración del ciclo principal debe ser el mínimo común múltiplo de los períodos de las tareas a planificar:

$$M = mcm(Pi) \tag{1.1}$$

En cuanto al ciclo secundario m debe cumplir las siguientes condiciones:

1. m debe ser menor o igual que el plazo de respuesta de cualquier tarea:

$$\forall i: m \leq Di \tag{1.2}$$

2. m debe ser mayor o igual que el mayor de los tiempos de computo:

$$\forall i: m \geq max(Ci) \tag{1.3}$$

3. m debe dividir a M:

$$\exists k: M = k \cdot m \tag{1.4}$$

4. m debe cumplir:

$$\forall i: m + \big(m - mcd(m \cdot Pi)\big) \leq Di \tag{1.5}$$

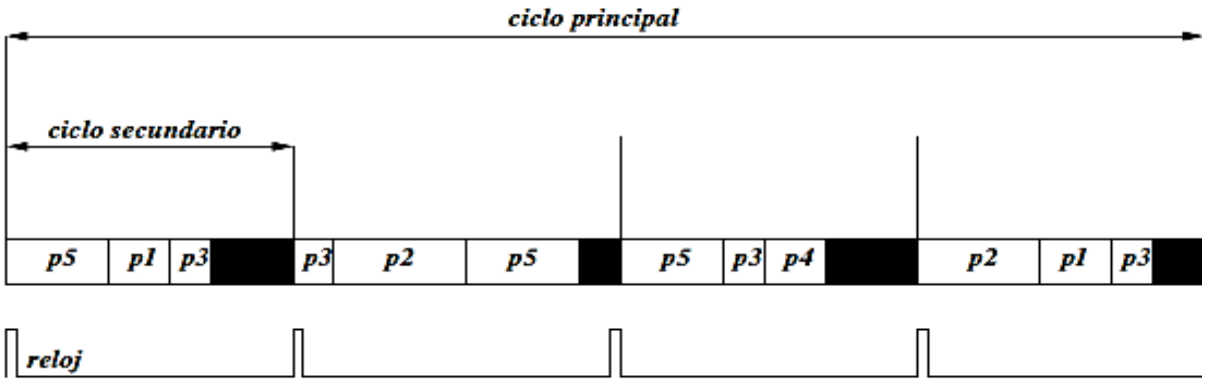


Figura 1.5 Planificación Cíclica.

Planificación basada en prioridades: Los algoritmos de planificación basados en prioridades seleccionan la tarea a ejecutar entre las disponibles atendiendo a un criterio básico (prioridad). Esta prioridad puede tener un carácter fijo (se fija por el diseñador atendiendo a algún criterio determinado: semántico, períodos o plazos) o dinámico (es determinada por el sistema atendiendo a algún criterio: la tarea más urgente), lo que determina el tipo de algoritmo de planificación basada en prioridades fijas (FPS) o planificación basada en prioridades dinámicas (EDF).

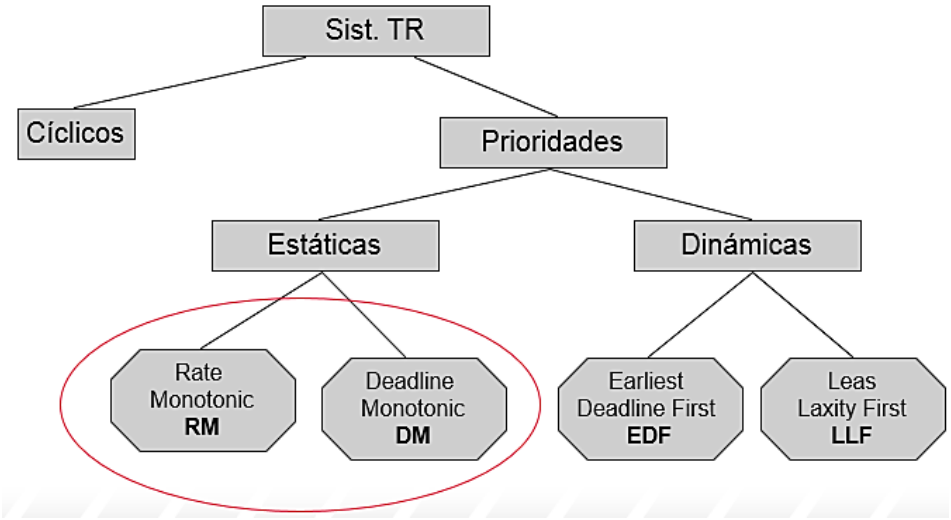


Figura 1.6 Esquema de planificación en Sistemas de Tiempo Real.

1.5.1 Planificación estática

En cuanto a las prioridades estáticas existe una forma fácil (lógica) de asignación para la cual hay desarrollada una teoría que permite determinar analíticamente si el sistema será panificable o no.

La teoría RMA (*Rate Monotonic Analysis*) establece:

$$U \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1.6}$$

Donde U es el factor de utilización para una tarea i-ésima y se calcula como el cociente entre el tiempo de cómputo (Ci) y el período asignado (Pi) a dicha tarea:

$$U = \frac{C_i}{P_i} \tag{1.7}$$

El factor de utilización representa el tiempo que utiliza la CPU para la realización de dicha tarea. La suma de las utilizaciones de todas las tareas se llama utilización del sistema y en un sistema monoprocesador debe ser menor que el 100%.

$$\sum_{i=1}^{n} \frac{c_i}{P_i} \leq 1 \tag{1.8}$$

El factor de utilización para un sistema se calcula como la sumatoria de los factores para cada una de las tareas por lo que la expresión general de la teoría RMA será:

$$U \leq U(n) \tag{1.9}$$

$$\sum_{i=1}^{n} \frac{c_i}{P_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1.10}$$

El test de RMA brinda la garantía de que el sistema será panificable si a cada tarea se le brinda una prioridad inversamente proporcional al período, es decir, a menor período mayor prioridad.

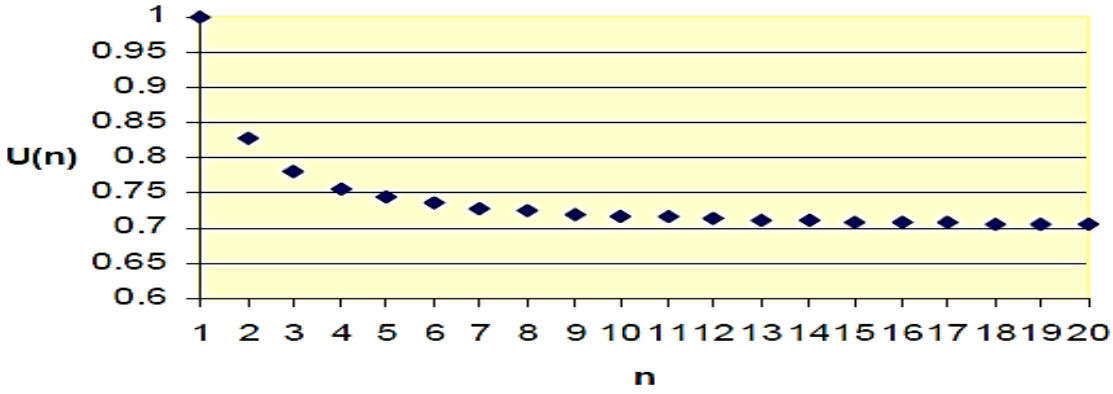


Figura 1.7 Utilización mínima garantizada para un sistema de n tareas.

Tabla 1.2 Utilización mínima garantizada para un sistema de n tareas.

n	1	2	3	4	……	∞
$n\left(2^{\frac{1}{n}}-1\right)$	1	0.828	0.780	0.756	……	0.693

El test RMA se conoce también como el test del Factor de Utilización y constituye la prueba necesaria pero no suficiente para conocer la planificación del sistema, por lo que un sistema de "n" tareas será panificable bajo cualquier asignación de prioridades si solo si cada tarea cumple su plazo de ejecución en el peor caso. El tiempo de finalización de peor caso de cada tarea ocurre cuando todas las tareas de prioridad superior se inician a la vez que ésta (Burns and Wellings, 1996).

Matemáticamente:

$$Wi \leq Di \forall 1 \leq i \leq n \tag{1.11}$$

Donde:

Wi: Representa el instante en el que finaliza la ejecución en el peor caso. Este valor se obtiene de la siguiente expresión:

$$Wi = Ci + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^0}{P_j} \right\rceil C_j \tag{1.12}$$

Donde:

$\forall j \in hp(i)$ es el conjunto de j-ésimas tareas con mayor prioridad que la tarea " i " y $\left\lceil \frac{w_i^0}{P_j} \right\rceil$ es la función techo que devuelve el entero por exceso de su argumento.

La solución es iterativa por lo que el test termina cuando dos valores consecutivos coinciden o se excede el plazo.

$$W_i^0 = Ci \tag{1.13}$$

$$W_i^1 = Ci + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^0}{P_j} \right\rceil C_j \tag{1.14}$$

$$W_i^2 = Ci + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^1}{P_j} \right\rceil C_j \tag{1.15}$$

$$W_i^k = W_i^{k+1} \tag{1.16}$$

Si en alguna iteración se obtiene un valor de W mayor que el plazo máximo de ejecución de la tarea, entonces esta tarea no será panificable y el sistema tampoco. Este test se conoce como el test de los tiempos de finalización (DM) y constituye la condición necesaria y suficiente para garantizar la planificación del sistema.

Si un conjunto de tareas con prioridades asignadas según el DM no es panificable, entonces ninguna otra forma de asignar prioridades lo hará panificable (Burns and Wellings, 1996).

El RM se convierte en un caso especial del DM (en el que los plazos son iguales al período).

Figura 1.8 Planificación estática

1.5.2 Planificación dinámica

Los planificadores dinámicos son, al igual que los estáticos, planificadores basados en prioridades, pero se caracterizan por que las prioridades no son estáticas durante toda la vida del sistema, sino que varían en el tiempo siguiendo un algoritmo que define el planificador (O'Boyle, 2014).

Básicamente se utilizan dos planificadores dinámicos, estos son:

EDF (*Earliest Deadline First*). Se asignan las prioridades dando preferencia a las tareas con plazo de finalización más próximo.

LLF (*Least Laxity First*). Se asignan las prioridades dando preferencia a las tareas con menor holgura.

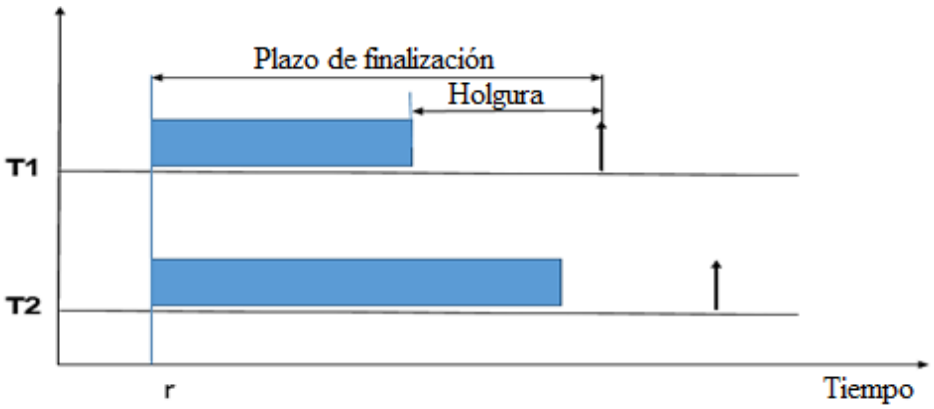


Figura 1.9 Esquema de planificación dinámica.

En el tiempo *t*, según el planificador EDF T1 tendría más prioridad que T2, pero según el LLF, T2 tendría más prioridad que T1. El inconveniente de los planificadores dinámicos es

que requieren un elevado tiempo de cálculo durante la ejecución del sistema. Por este motivo, el único que resulta realmente útil es el EDF.

Un planificador dinámico puede expresarse definiendo la tarea en ejecución Tx en cada momento de la siguiente forma:

$$\mathrm{T}x \text{ Ejecutándose} \Leftrightarrow \forall\, x,i \in \text{Activas}\ f\,(\mathrm{T}x,\, t) \geq f\,(\mathrm{T}i,\, t).$$

Si la función *f* no depende de *t*, se tratará un planificador estático. En el planificador EDF la función *f* es tal que da mayor prioridad a las tareas con distancia absoluta al plazo de ejecución más pequeña.

1.6 Sistemas operativos de tiempo real. Características y requisitos

Un sistema operativo de tiempo real es una clase de sistemas operativos que cumple unos determinados requisitos para dar soporte a aplicaciones de tiempo real. Un sistema operativo de tiempo real (RTOS) o Núcleo de tiempo real (RTK) tiene que cumplir las siguientes características:

- **Multiprogramación:** Tiene que dar soporte a varias tareas de aplicación.
- **Predictibilidad de los servicios:** Todos los servicios que ofrece deben ser predecibles y su coste debe ser conocido.
- **Garantía de ejecución:** Para poder garantizar la ejecución de las tareas, el RTOS tiene que poder seleccionar y ejecutar éstas atendiendo a criterios tales como la prioridad de una tarea.
- **Gestión del tiempo:** El sistema debe ser capaz de gestionar un reloj que sea siempre creciente (monotónico) y gestionar un número de temporizadores adecuado para permitir el control de los distintos requisitos temporales de las tareas.
- **Comunicación y sincronización:** El sistema operativo debe ofrecer los mecanismos básicos y los protocolos adecuados (basados en herencia de prioridad) para sincronizar y comunicar de forma segura y eficiente las tareas de la aplicación.
- **Gestión de memoria:** Por un lado, es importante que el espacio de direccionamiento del núcleo esté protegido respecto a la aplicación. Esto es fundamental para evitar que fallos en la aplicación afecten al propio sistema operativo. Por otro lado, el núcleo

tiene que suministrar mecanismos predecibles para la gestión de memoria dinámica que puedan utilizar las aplicaciones.

- **Interfaz de programación (API):** La interfaz de programación determina en gran medida la portabilidad de las aplicaciones.

Un RTOS debe ofrecer comportamiento determinista en todos los servicios del sistema. Son especialmente significativas las medidas de ciertas operaciones de bajo nivel que realiza muy frecuentemente el sistema y que requieren, además de estar acotadas, un coste temporal bajo. Algunas de estas medidas críticas son:

- Latencia de la gestión de interrupciones garantizada.
- Cambio de contexto acotado.
- Baja sobrecarga introducida por el planificador.
- Gestión eficiente del reloj y los temporizadores.
- Tiempo de respuesta del hardware.

Para seleccionar un RTOS para una aplicación determinada es necesario tener en cuenta no sólo sus características de tiempo real, sino que además se deben considerar otros factores tales como:

- Lenguajes soportados.
- Tipo de interfaz de llamadas al sistema (API) siguiendo estándares (POSIX, OSE, ARINC- 563, etc.) y, por tanto, portabilidad de las aplicaciones.
- Certificación.
- Estructura basada en componentes que permita la generación de sistemas. Seleccionando aquellos elementos necesarios para una aplicación determinada (sistemas empotrados).
- Middleware en el que se integra.
- Disponibilidad de amplia gama de manejadores de dispositivos.
- Otros aspectos: sistema de ficheros, pila de comunicaciones, etc.

1.7 Consideraciones del capítulo

En la actualidad los RTOS propietarios o de *kernel* cerrado como QNX y VxWorks se han mantenido entre los mejores sin embargo los basados en Linux y de código abierto como son RTLinux y RTAI han seguido extendiéndose entre los cuales destaca Xenomai de preferencia por muchos programadores.

Una opción viable para el desarrollo de los sistemas en tiempo real es el uso de software libre puesto que existe una gran comunidad de programadores dedicados al mantenimiento, mejora y distribución de estos sistemas operativos además de que poseen una gran sostenibilidad e independencia tecnológica. El uso de estos sistemas en plataformas embebidas de igual tipo, es decir, hardware libre, constituyen la vía adecuada para el desarrollo tecnológico. Algunas características que justifican el empleo de estos hardware son precisamente la fiabilidad, prestaciones y la relación costo/beneficio que estos brindan. De la revisión bibliográfica y los análisis reportados por la literatura se puede concluir que:

De los RTOS que existen en el mercado para aplicaciones de control con altas restricciones de tiempo se sugiere el uso de los basados en código abierto y dentro de estos los basados en Linux. En el próximo capítulo se hará mayor énfasis en dichas plataformas y se describirán las principales características de cada una de ellas.

CAPÍTULO 2. Pruebas de RTOS de código abierto basados en Linux

En este capítulo se presenta algunos experimentos con diferentes sistemas operativos que soportan las características de tiempo real como es el caso de RTAI, RTLinux y Xenomai. Dichos experimentos están destinados a medir y comparar la latencia de dichos sistemas. Si bien no es la única condición que debe cumplir un sistema operativo para ser considerado de tiempo real, es una condición necesaria. Los tiempos de latencia y su variabilidad darán los límites de requerimientos de restricciones temporales que podrán soportar. En los últimos epígrafes se exponen otros criterios que influyen en la selección de la plataforma escogida y se dan las conclusiones parciales del capítulo.

2.1 RTLinux

RTLinux es un sistema operativo de tiempo real multitarea que ejecuta Linux como un *thread* (hilo de ejecución) de menos prioridad que las tareas de tiempo real (Guire, 2007). Con este diseño, las tareas de tiempo real y los manejadores de interrupciones nunca se ven retrasados por operaciones que no son de tiempo real. RTLinux proporciona un entorno de ejecución bajo el *kernel* de Linux, como consecuencia de esto, las tareas de tiempo real no pueden usar los servicios de Linux (Yodaiken, 1999). Para disminuir este problema, el sistema de tiempo real se ha divido en dos partes: la capa de tiempo real estricto que ejecuta encima de RTLinux, y la capa de tiempo real flexible, que ejecuta como un proceso normal de Linux.

La propuesta de 2 capas es un método útil para proporcionar tiempo real estricto mientras se mantienen las características de escritorio de un sistema operativo. La separación del *kernel* de tiempo real del mecanismo del *kernel* de Linux de propósito general permite optimizar ambos sistemas de forma independiente (Proctor, 2002).

2.1.1 Planificación en RTLinux

Normalmente no se puede elegir la plantificación en un sistema operativo, no obstante, para aumentar la flexibilidad de RTLinux, Yodaiken y Barabanov optaron por incluir varias políticas de planificación para que los administradores del sistema eligieran la más adecuada a sus requerimientos de tiempo. La primera fue desarrollada por Victor Yodaiken y se trata de una planificación en la que una tarea lista para ejecución puede desalojar del procesador a otra de menor prioridad. En esta política se asigna una prioridad fija a cada tarea y se ejecutan según estén listas por orden de prioridad (Yodaiken, 1999). Además, se puede también implementar un gestor de interrupciones que se despierte cuando sea necesario. Con esta política se puede usar el algoritmo de planificación de tasa creciente *(Rate Monotonic)* y gestionar así todas las prioridades.

El otro planificador que se puede usar fue implementado por Ismael Ripoll y usa el algoritmo de *Earliest Deadline First* (Ripoll et al., 1997). Este algoritmo asigna las prioridades de forma dinámica de manera que se le da la máxima prioridad a la tarea cuyo plazo de finalización esté más cercano. El planificador por defecto que viene con RT-Linux es un planificador basado en prioridades estáticas y trata a la tarea Linux como la tarea de menor prioridad (Whilshire et al., 2000). Si las tareas de tiempo real consumen todo el tiempo del procesador, entonces la tarea Linux no recibe tiempo de procesador y da la impresión de que el sistema se ha "colgado". Con RT-Linux se tiene a la vez un sistema de tiempo real y un sistema operativo clásico.

2.1.2 Tareas de Tiempo Real en RTLinux

Para introducir las tareas de tiempo real en el espacio del núcleo se aprovecha la característica modular del *kernel* de Linux y se cargan las tareas de tiempo real en forma de módulos de manera que se ejecutaran todas en el espacio del núcleo. De hecho, como ya se ha mencionado el propio RTLinux se carga como un módulo (Flynn and M., 2010). Como consecuencia de esta arquitectura se debe realizar un gran esfuerzo de programación ya que si se comete algún error se puede bloquear todo el sistema. No obstante, desde un punto de vista práctico, un sistema en tiempo real en general tiene que lidiar con periféricos y en ellos es vital no cometer errores en la programación.

Otro de los peligros de la arquitectura de RTLinux está en un mal dimensionamiento de las tareas críticas. Uno de los errores más comunes consiste en que siempre haya una tarea crítica ejecutándose ya que el núcleo de Linux no conseguirá nunca el procesador y dará la impresión de que el sistema está bloqueado.

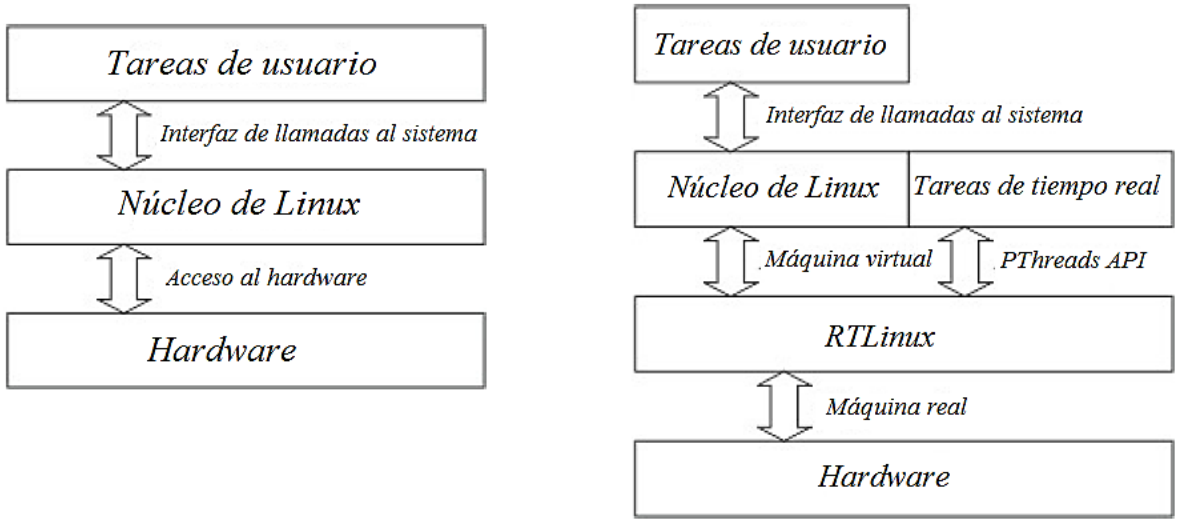


Figura 2.1 Arquitectura de Linux estándar y de RTlinux.

2.2 Real Time Application Interface (RTAI)

RTAI es una implementación de Linux con licencia GNU/GPL para tiempo real basada en RTLinux. Añade un *kernel* en tiempo real al *kernel* de Linux y trata al *kernel* de Linux como una tarea de menor prioridad. Añade además una serie de mecanismos de comunicación entre procesos y otros servicios de tiempo real (Beal, 2000).

Aunque RTAI proporciona servicios en tiempo real, conserva las características y los servicios de Linux estándar, proporcionando además un módulo que facilita el desarrollo de aplicaciones en tiempo real en el espacio de usuario.

2.2.1 Arquitectura RTAI

La arquitectura de RTAI es similar a RTLinux: El *kernel* de Linux es tratado como una tarea de baja prioridad que se ejecutará cuando no hay ninguna tarea de mayor prioridad en uso.

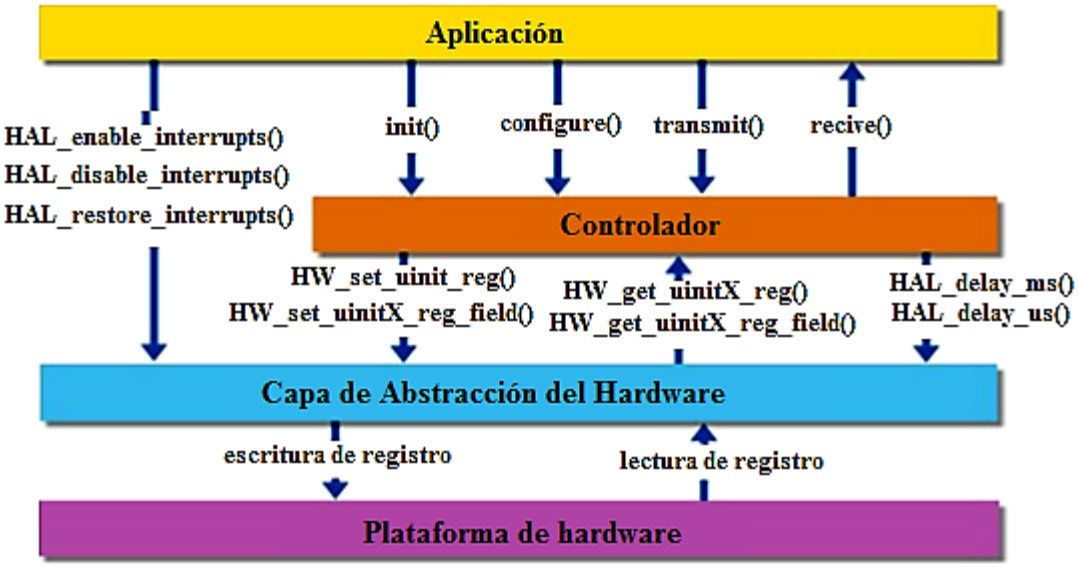


Figura 2.2 Arquitectura de RTAI.

Los módulos básicos de tiempo real se implementan dentro del *kernel* de Linux de manera que RTAI maneja las interrupciones de los periféricos que son atendidas por el *kernel* de Linux después de haberse ejecutado las acciones consecuentes de las rutinas de tiempo real (Racciu and Mantegazza, 2006). Con ello se consigue que las tareas en tiempo real sean prioritarias respecto a las tareas propias de Linux y en consecuencia se ejecutarán antes. Sólo cuando no haya tareas en tiempo real ejecutándose se ejecutarán las tareas propias del *kernel.*

2.2.2 Capa de Abstraccion del Hardware *(Hardware Abstraction Layer)*

La HAL *(Hardware Abstraction Layer)* es un elemento del sistema operativo que funciona como interfaz entre el software y el hardware (Luchetta et al., 2007). Su objetivo es evitar que las aplicaciones accedan directamente al hardware consiguiendo un alto grado de independencia entre ellos, abstrayendo el funcionamiento del hardware y dándole al software una forma de interactuar con los requerimientos específicos del hardware sobre el que va a correr. Los desarrolladores de RTAI introducen el concepto de *Real Time HAL* que se usa para capturar las interrupciones hardware y procesarlas después. RTHAL es una estructura instalada en el *kernel* de Linux que reúne los punteros a los datos internos del hardware relacionado en el *kernel* y las funciones necesarias por RTAI para operar. El objetivo de RTHAL es minimizar el número de cambios necesarios sobre el código del *kernel* y por tanto mejorar el mantenimiento de RTAI y del código del *kernel* de Linux.

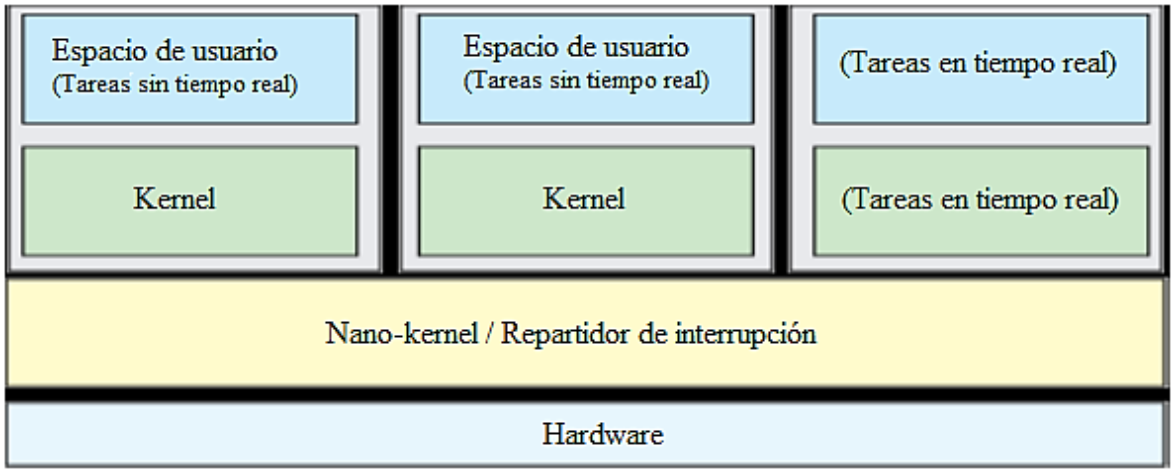


Figura 2.3 Capa de abstracción del hardware (*Hardware Abstraction Layer*).

2.2.3 Planificación en RTAI

La unidad de planificación de RTAI es la tarea. Siempre hay al menos una tarea que se ejecuta como la tarea de menor prioridad, y esa es el *kernel* de Linux (Zamarrón, 2004). El planificador proporciona servicios tales como:

- *Suspend:* Pausa la ejecución de una tarea.
- *Resume:* Vuelve a ejecutar una tarea pausada.
- *Yield:* Coloca la tarea preparada para su ejecución.
- *Make periodic:* Hace que una tarea se ejecute periódicamente.

El planificador es implementado como un módulo del *kernel* dedicado (contrario a RTLinux) lo que facilita la implementación de planificadores alternativos si es necesario.

2.2.4 LXRT Interfaz del espacio de usauario para RTAI

LXRT es un API para RTAI que hace posible el desarrollo de aplicaciones de tiempo real en el espacio de usuario sin tener que crear módulos para el *kernel.* Esto es útil por dos motivos:

1. El espacio de memoria destinado al *kernel* no está protegido de accesos inválidos. Consecuentemente el acceso indebido puede provocar la corrupción de datos y el mal funcionamiento del *kernel* de Linux.
2. En caso de que el *kernel* deba ser actualizado, los módulos que se necesitan se recompilarán y ello puede provocar incompatibilidades con la nueva versión.

Aunque los desarrolladores de RTAI ven bien el desarrollo de aplicaciones de tiempo real rígido usando LXRT, el tiempo de respuesta no es tan bueno como la ejecución de las tareas como módulos del *kernel*.

La característica más destacable de RTAI es que no se trata de un sistema operativo de tiempo real en sí, sino de un conjunto de módulos que se instalan en el *kernel* de Linux además de un desarrollador para implementar más módulos. Con ello se consigue dotar a un sistema operativo fiable y estable con herramientas no implementadas de tiempo real como es Linux de una serie de rutinas para que pueda tener capacidades de tiempo real. La principal ventaja de esta implementación es que permite que cada usuario adapte el sistema a sus necesidades, desarrollando más módulos, que una vez compilados y probados, si están libres de errores, pueden ser incorporados también al *kernel*.

2.3 Arquitectura Xenomai

Xenomai se basa en una arquitectura en la que se mantienen simultáneamente en ejecución dos núcleos, el de tiempo real, que ofrece los recursos propios de un sistema operativo de tiempo real y un núcleo de Linux estándar que mantiene todos sus recursos (Gerum, 2010). Entre ambos existen diferentes tipos de mecanismos de comunicación diseñados para que preserven la predictibilidad que requiere el tiempo real. En la Figura 2.4, se muestra los principales elementos de la arquitectura Xenomai. Ambos núcleos coexisten haciendo uso de una capa de virtualización denominada Adeos[4] que intercepta las interrupciones del hardware y las redirecciona hacia ambos núcleos. Adeos pasa los eventos a sus clientes de software en orden de prioridad; el sistema Xenomai tiene mayor prioridad que Linux. Por lo tanto, el *kernel* de Linux sólo recibe eventos de interrupción virtual, y aquellos sólo después de que el software de mayor prioridad (por ejemplo, la capa Xenomai) haya tenido la oportunidad de responder primero. Similarmente, cuando el *kernel* de Linux bloquea los manejadores de interrupción, lo hace sólo para sí mismo. Adeos está diseñado para que la transición entre la ejecución en los dos núcleos sea muy ligera y para que la latencia de respuesta a las

[4] ADEOS (Adaptive Domain Environment Operating Systems), activa múltiples núcleos, llamados dominios, que existen simultáneamente sobre el mismo hardware.

interrupciones hardware sea mínima. Los subprocesos Xenomai de alta prioridad recibirán sus eventos de la *I-pipe* según lo programado (Tuñón and Terriza, 2010).

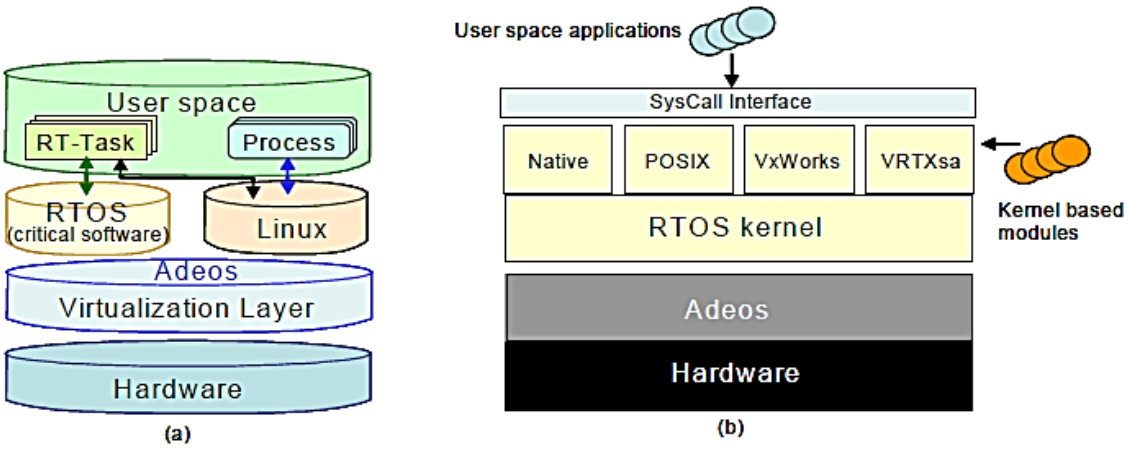


Figura 2.4 Arquitectura Xenomai. (b) Interfaces a RTOS legados.

2.3.1 API de Xenomai

La API nativa de Xenomai ofrece seis familias de servicios. Cada familia define un amplio conjunto de funciones, la mayoría de las cuales pueden ser invocadas tanto desde el espacio de usuario, como desde el núcleo:

• Gestión de tareas: Define un conjunto de servicios relacionados con el ciclo de vida de los *threads* y el control de sus parámetros de planificación.

• Servicios de temporización: Esta familia agrupa funciones relativas a la gestión de los diferentes tipos de temporizadores: *System timers*, *Watchdogs* y *Alarms.*

• Mecanismos de sincronización: Ofrece recursos para la sincronización de las tareas y el control de acceso a recursos que requieren exclusión mutua: *Counting semaphores*, *Mutexes*, *Condition variables* y *Event flag groups.*

• Servicios de comunicación y transmisión de mensajes: Ofrece servicios para implementar diferentes modos de intercambio de bloques de datos entre las tareas de tiempo real, y entre éstas y los procesos Linux del espacio de usuario: *Intertask synchronous message*, *Message queues*, *Memory heaps* y *Message pipes*.

• Manejadores de dispositivos. Como la gestión de entradas/salidas es uno de los aspectos más complejos de los RTOS, la interfaz nativa sólo ofrece mecanismos de gestión de las interrupciones, y de acceso a memoria de I/O desde el espacio de usuario.

• Soporte de registros: Es muy específica de la interfaz nativa, y ofrece funciones semejantes a las llamadas de sistema desde los diferentes espacios de ejecución.

2.3.2 Servicios de los hilos

Es un servicio del entorno de ejecución con el que los contenedores de los componentes crean los *threads* que invocan los puertos de activación. En el diagrama de clases del Anexo I se muestra la funcionalidad del servicio.

Para la activación de un componente basta con invocar, por cada puerto de activación, el método estático *execute()* del *ThreadingService*. Mediante esta invocación se crea un *thread* que activa el puerto del componente que se le pasa como parámetro.

• Por cada *thread* que se crea se retorna un objeto *ExecutionController* con el que el contenedor controla (*start(), stop(), resume(), finish()*) el ciclo de vida del *thread*, de acuerdo con el estado del componente.

• Utilizando Xenomai es posible implementar el servicio de forma muy simple ya que el API de las tareas de Xenomai ofrece funciones muy próximas a las que se necesitan en RT-CCM (*Real-Time Container Component Model)*.

• Una dificultad importante que se ha encontrado es la imposibilidad de extender la información asociada a un *thread*. Xenomai asocia a cada *thread* la estructura *RT_TASK_INFO* que es cerrada y no admite ser extendida. Por ello se ha optado por incluir en el nombre de la tarea un identificador con dos caracteres numéricos, que permiten, acceder de forma eficiente al correspondiente *ExecutionController*, y en él incluir los atributos que se asignan a la tarea. Esta estrategia se ha utilizado para incluir el atributo *StimId* que se utiliza para gestionar la planificación de la tarea.

2.3.3 Servicios de planificación

La tecnología RT-CCM (*Real-Time Container Component Model)* utiliza los interceptores como base para poder establecer los parámetros de planificación con los que se ejecuta cada invocación de los servicios de un componente. Un interceptor es un elemento que se

introduce en el contenedor por cada método de los puertos de un componente que se declaran de tiempo real, y que introduce un mecanismo para que cuando se invoque el método, se invoque previamente una operación del entorno (*receiveRequest()*), y cuando se obtenga el retorno del método, se ejecute una nueva operación del entorno (*sendReply()*).

En la tecnología RT-CCM se utilizan los interceptores para gestionar los parámetros de planificación con los que se ejecuta el servicio que representa el método al que se asocia. En xCCM el parámetro de planificación que se gestiona es la prioridad del *thread* que invoca la operación del servicio.

En el Anexo II se muestra la solicitud desde un componente cliente del método *oper()* del componente que se representa. La solicitud es procesada por el interceptor asociado al método *oper()*. Inicialmente solicita el método *nextStimId()* del *InterceptorSchedulingManager* asociado al interceptor, el cual de acuerdo con el estado de la transacción (*stimId* asociado al *thread*) evalúa el nuevo *stimId* con que se ejecutará el servicio y lo establece en el *thread* solicitante, y a través del método *setPriority()* del *SchedulingService* establece la prioridad del *thread* solicitante con la que se ejecutará la operación solicitada. Por último se solicita sobre el código del componente la operación *oper()* que era el objetivo de todo este proceso. Cuando la operación *oper()* termina y retorna los resultados, interviene de nuevo el interceptor. El cual solicita el método *getbackStimId()* del objeto *InterceptorSchedulingManager* que reestablece en el *stimId* del *thread* el valor que estaba establecido antes de la solicitud, y dentro de él se vuelve a solicitar el método *setPriority()* del *SchedulingService* con el que también se restaura la prioridad que tenía el *thread* antes de la solicitud.

2.4 Pruebas con RTAI y RTLinux

Las pruebas se realizaron sobre un sistema de cómputo (Romero et al., 2011), con las siguientes características:

- Arquitectura CPU: Intel x86
- Procesador: Intel Pentium III Coppermine 733 Mhz
- Frecuencia de bus externo: 133 MHz

El objetivo de las pruebas realizadas consistió en evaluar el desempeño en relación al tiempo de respuesta para los diferentes sistemas operativos. Para ello, se tomó como métrica principal la latencia en el peor caso.

Estas pruebas consisten en el inicio reiterado de tareas de alta prioridad, tomando la cuenta de ciclos de CPU para determinar el tiempo transcurrido entre que se produce el evento y el instante en el que comienza a ser atendida. Para cada caso, las pruebas se realizaron tanto bajo condiciones de CPU ociosa como en condiciones de carga.

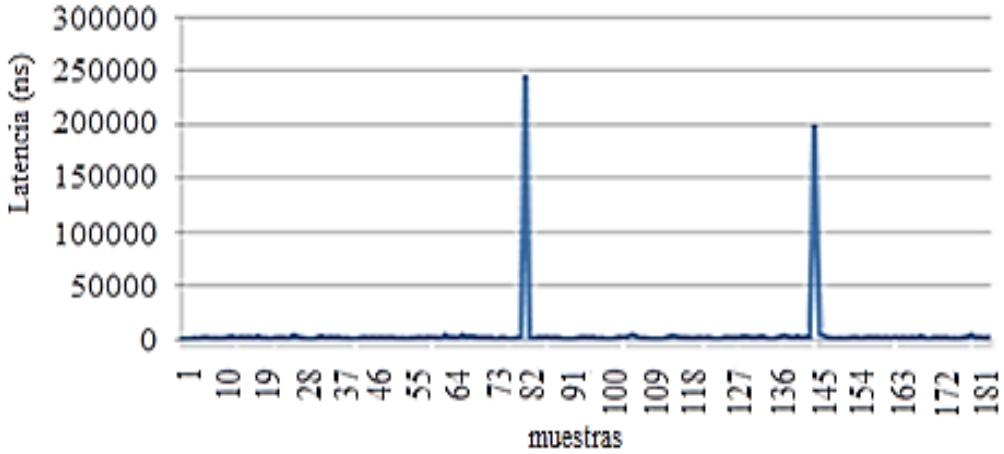


Figura 2.5 Latencia sin carga de RTAI (ns).

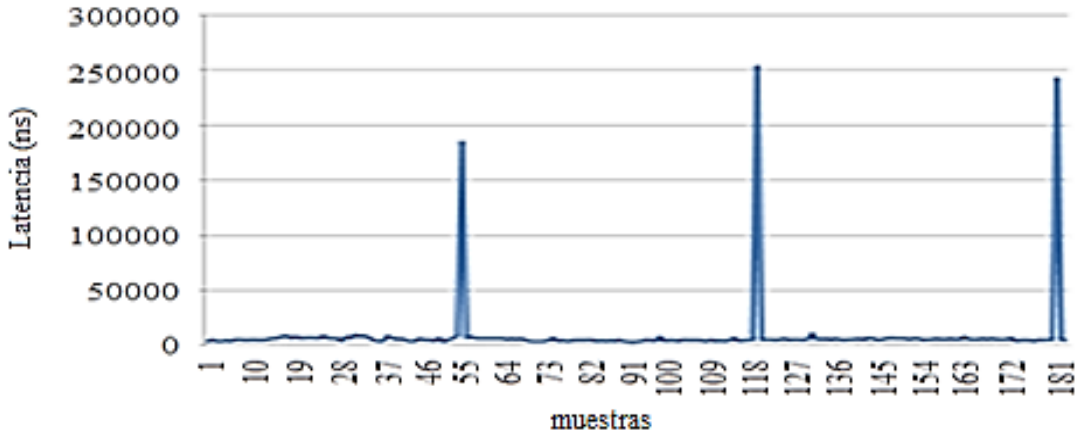


Figura 2.6 Latencia con carga en RTAI (ns).

Puede observarse que con la sobrecarga de CPU los valores de latencia máxima se han incrementado.

Tabla 2.1 Latencias en RTAI con carga y sin carga de CPU

Configuración	Experimento	Latencias mín	Latencias máx
RTAI	sin carga de CPU	2000 ns	4000 ns
RTAI	con carga de CPU	3800 ns	8000 ns

Los valores pico que pueden verse en los gráficos de latencias son causados por condiciones específicas del sistema, conocidas como condiciones "mata latencia" *("latency killers")* que suelen ser generadas por diversas causas, tales como el soporte del puerto USB, el sistema administrador de energía APM[5] o ACPI[6], o el inicio de interrupciones SMI[7] de modo de administración de sistema SMM[8], entre muchas otras. Para el caso de una implementación de tiempo real con restricciones de tiempo concretas, estos valores altos en la medición de latencia pueden no ser aceptables, y se suele cambiar la configuración en el BIOS o la configuración del *kernel* del sistema operativo para evitar dichas condiciones. Para el

[5] *Advanced Power Management.* Es un API desarrollado por Intel y Microsoft que permita que el BIOS administre la energía, tal como reducir la velocidad de la CPU, apagar el disco duro o apagar el monitor después de un período de inactividad para conservar corriente eléctrica, especialmente para las computadoras portátiles.

[6] Es el sucesor de APM.

[7] *Structure of Management Information:* es un subconjunto adaptado de ASN.1, funciona en el Protocolo de Manejo de Redes Simples para definir los módulos de objetos administrados relacionados en una Base de Información de Gestión.

[8] *System Management Mode:* modo de operación de microprocesadores x86 en el que se suspende la ejecución normal (incluyendo el sistema operativo), y se ejecuta un software especial separado en un modo de alto privilegio.

propósito de estas pruebas se considera que son tolerables, ya que solo ocurren ocasionalmente y son claramente identificables.

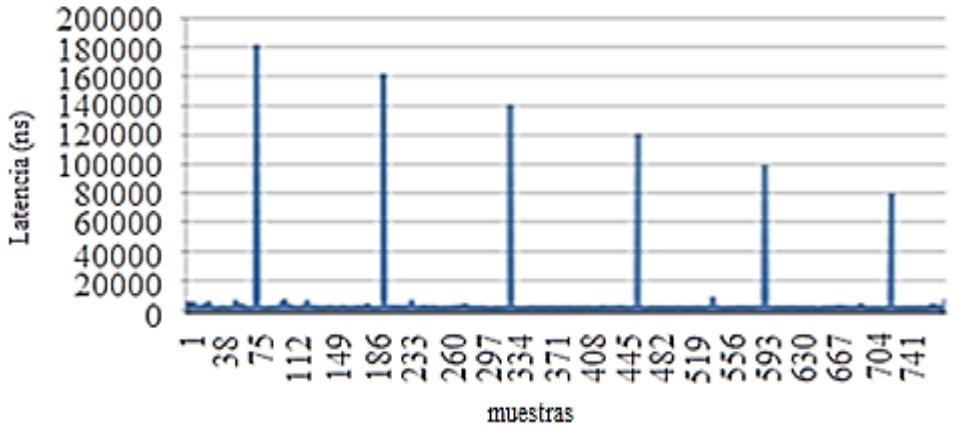


Figura 2.7 Latencia sin carga en RTLinux (ns).

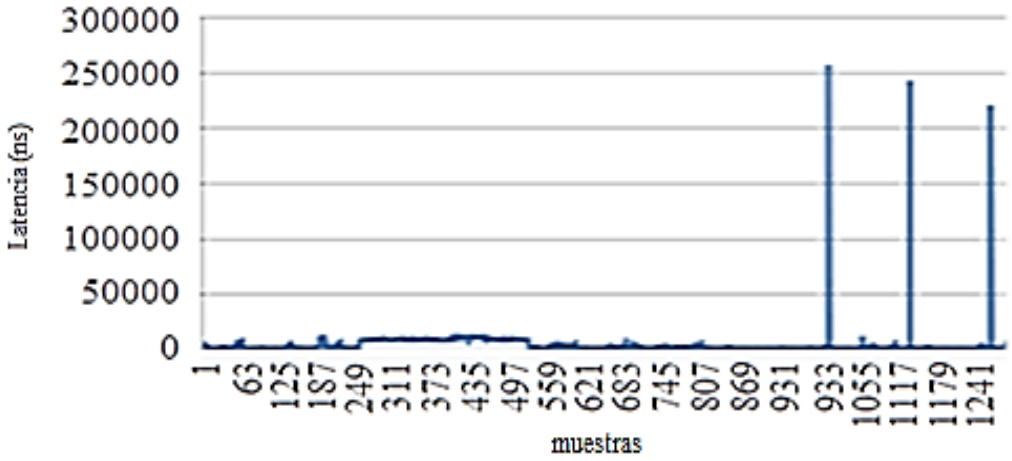


Figura 2.8 Latencia con carga RTLinux (ns).

Con carga de CPU, al igual que en el caso de RTAI, puede notarse en RTLinux un efecto mínimo de la carga de CPU sobre la latencia del planificador. Si bien una gran parte de las muestras tienen valores de latencia similares a aquellos obtenidos con la CPU en estado ocioso, también existe un pequeño conjunto de muestras con valores superiores entre los 7800 ns y 13200 ns.

Tabla 2.2 Latencias en RTLinux con carga y sin carga de CPU

Configuración	Experimento	Latencias mín	Latencias máx
RTLinux	sin carga de CPU	1200 ns	2400 ns
RTLinux	con carga de CPU	3800 ns	8000 ns

Aunque que para estas pruebas tanto en RTAI como en RTLinux las latencias máximas y latencias mínimas de CPU en condiciones de carga son iguales. Se puede notar en los gráficos que los picos de latencias generados por RTLinux ocurren con mayor periodicidad.

2.5 Pruebas con Xenomai y RTLinux

La plataforma utilizada para estas pruebas fue un popular sistema integrado llamado *BeagleBoard*. Se mide el desempeño de los procesos usando un sistema de medición externo (Brown and Martin, 2010).

Modo de respuesta: En el modo respuesta, el sistema de medición espera un intervalo aleatorio de 3 μs hasta un período máximo configurable, luego baja su pin de salida (es decir, la entrada al sistema de prueba) y mide cuánto tiempo ha pasado hasta que el sistema de prueba disminuya su pin de salida en respuesta. A continuación, inmediatamente levanta su pin de salida y espera a que el sistema de prueba haga lo mismo antes de comenzar otro ciclo. Las medidas se toman sólo en los bordes descendentes. Cuando se le ordena parar, el sistema de medición informa un histograma de latencias de respuesta.

Implementaciones de tareas de respuesta

En la tarea de respuesta de interrupción, el sistema de prueba debe esperar una interrupción desde un pin GPIO de entrada. Cuando esto sucede, el sistema debe configurar un pin GPIO de salida para que coincida con la señal de entrada. Las transiciones de entrada ocurren a intervalos aleatorios.

Resultados: La Tabla 2.3 muestra las estadísticas básicas de los experimentos. La Figura 2.9 presenta datos de rendimiento detallados para cada experimento, representados gráficamente en una escala lineal. Las líneas discontinuas indican la envoltura dentro de los

cuales se han producido el 95% de las muestras de medición. Los valores del 95% y 100% están en algunos casos separados por órdenes de magnitud. La Figura 2.10 representa estos valores en una escala logarítmica.

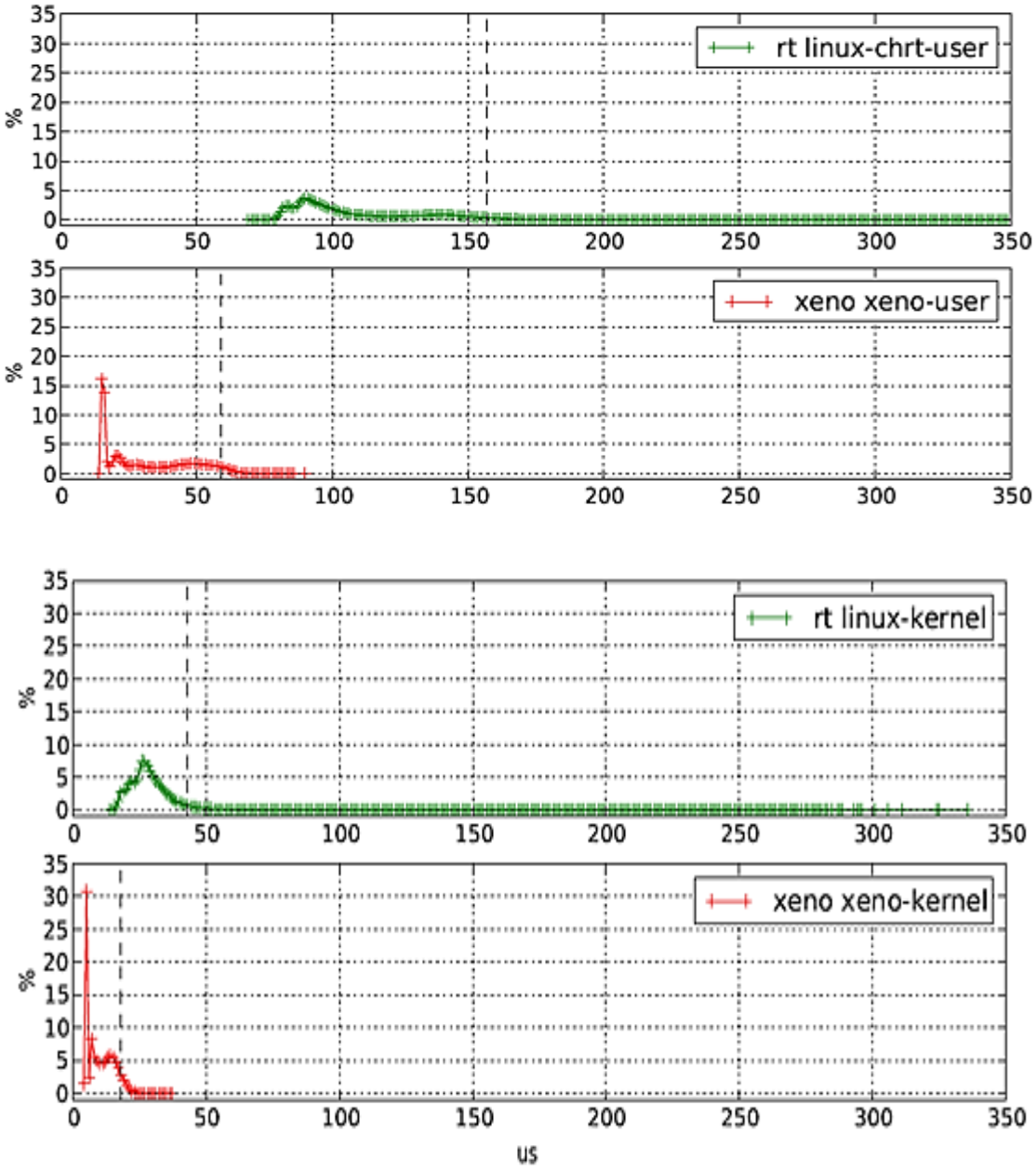


Figura 2.9 Experimentos de respuesta de configuración cruzada: tiempo de cambio del GPIO de entrada a cambio del GPIO de salida. La línea vertical discontinua indica la región que contiene 95% de las muestras para ese experimento.

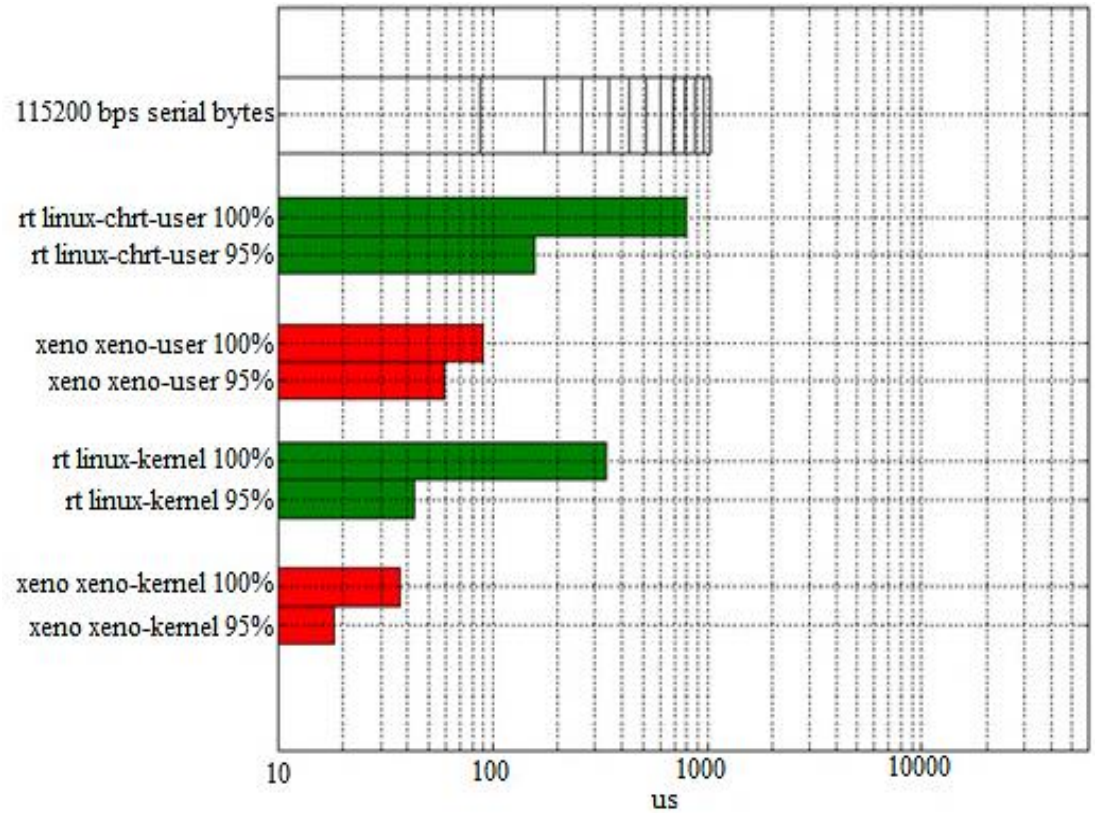


Figura 2.10 Experimentos de respuesta de configuración cruzada: máximo y 95% de tiempos de respuesta de envolvente trazados en una escala logarítmica. El tiempo de transmisión para 12 bytes serie a 115200bps (87µseach) se representa para la comparación.

Tabla 2.3 Experimentos de respuesta de configuración cruzada: latencia desde el cambio del GPIO de entrada al cambio del GPIO de salida correspondiente.

Configuración	Experimento	# Muestras	Media	95%	100%
rt	linux-chrt-user	1849438	99µs	157µs	796µs
xeno	xeno-user	1926157	26µs	59µs	90µs
rt	linux-kernel	1924955	28µs	43µs	336µs
xeno	xono-kernel	1943258	9µs	18µs	37µs

Tabla 2.4 Experimentos de respuesta de configuración cruzada: la frecuencia máxima aproximada posible para la cual la latencia no excede de 1/2 período, para casos del 95% y 100%.

Configuración	Experimento	95% período	100% período
rt	linux-chrt-user	3.18 kHz	0.63 kHz
xeno	xeno-user	8.47 kHz	5.56 kHz
rt	linux-kernel	11.63 kHz	1.46 kHz
xeno	xono-kernel	27.78 kHz	13.51 kHz

95% de rendimiento duro: El espacio de usuario de Xenomai es sustancialmente mejor que el espacio de usuario de RTLinux.

100% rendimiento duro: El espacio del *kernel* de Xenomai es el mejor intérprete. El espacio de usuario de Xenomai se ejecuta más lento que el espacio del *kernel*, pero de todas maneras supera todas las implementaciones que no son de Xenomai.

Para aplicaciones 100% duras, inmediatamente se sugiere un entorno Xenomai. Xenomai separa las rutas de manejo de interrupciones en tiempo real de las complejidades del *kernel* de Linux. Se cree que esto reducirá la probabilidad de eventos raros e irreproducibles que causen que los requisitos de tiempo sean violados. Los beneficios de esta separación se reflejan en la consistencia del rendimiento de Xenomai, es decir, en el intervalo relativamente pequeño entre los números de rendimiento mediano y 100%, en los números de Xenomai a través de las tareas periódicas y de respuesta, el espacio de usuarios y el espacio del *kernel.*

En el Anexo III se muestra el código del programa de un test de latencia de una tarea periódica en Xenomai. La plataforma utilizada para esta prueba fue una Raspberry Pi 3 Modelo B.

2.6 Otras características de Xenomai que influyen en la selección de un RTOS

Aunque la latencia constituye un factor importante para la selección de un RTOS este no es el único parámetro que se debe seguir. Mientras RTAI se enfoca en las latencias más bajas técnicamente factibles, Xenomai también considera la extensibilidad limpia (pieles de RTOS), portabilidad y capacidad de mantenimiento como objetivos muy importantes.

El camino de Xenomai hacia el apoyo de PREEMPT_RT de Ingo Molnár es otra diferencia principal comparado con los objetivos del RTAI.

Una gran diferencia de Xenomai frente al resto de soluciones es que permite que las tareas de tiempo real trabajen en el espacio de usuario. Además, tiene la capacidad de separar los procesos de tiempo real de las tareas normales que crea el API POSIX estándar de Linux, esto significa que los hilos de tiempo real en el *kernel* Xenomai heredan la capacidad de invocar las funciones de Linux cuando no se ejecutan en tiempo real. Una ventaja de esta separación de tareas es que las tareas de tiempo real no usan el mismo espacio de memoria como las otras tareas, por consiguiente, la posibilidad que ocurra una colisión debido a la existencia de errores en una tarea de tiempo real se reduce.

Xenomai presenta una abundante documentación con ejemplos incluidos en su sitio oficial en casos de que el usuario necesite ayuda.

Por estas razones se escoge Xenomai para validar un algoritmo de control con exigencias de tiempo.

2.7 Consideraciones del capítulo

Después del estudio comparativo entre los RTOS vistos anteriormente se decide usar Xenomai porque permite ejecutar *threads* en tiempo real, ya sea estrictamente en el espacio del *kernel*, o dentro del espacio de direcciones de un proceso Linux. La latencia de planificación de peor caso siempre está cerca de los límites de hardware y es predecible, ya que Xenomai no está obligado a sincronizarse con la actividad del *kernel* de Linux en dicho contexto y puede evitar cualquier actividad regular de Linux sin demora. Además, pueden existir algunos casos en los que se requiere ejecutar parte del código de tiempo real incorporado en los módulos del *kernel*, especialmente con sistemas heredados o plataformas de bajo nivel con hardware (MMU: unidad de manejo de memoria) de bajo rendimiento. Por esta razón, la API nativa de Xenomai proporciona el mismo conjunto de servicios en tiempo real de forma transparente a las aplicaciones, independientemente de su espacio de ejecución. Además, algunas aplicaciones pueden necesitar actividades en tiempo real en ambos espacios para cooperar, trabajando en el mismo conjunto de objetos API.

CAPÍTULO 3. MODELADO E IMPLEMENTACIÓN DE UNA APLICACIÓN DE CONTROL USANDO XENOMAI

En este capítulo se describe una breve historia del surgimiento del microcomputador Raspberry Pi, así como una descripción funcional del mismo. Se muestran los pasos a seguir para la instalación de Xenomai en dicha plataforma. Se presentan figuras y ecuaciones que describen el funcionamiento de una estructura cinemática (V-plotter) controlado por motores de pasos y se explica cómo realizar la tarea de control de dicha estructura usando Xenomai. Por último, se realiza un breve análisis económico y se exponen las consideraciones de capítulo.

3.1 Surgimiento del microcontrolador Rraspberry Pi

Raspberry Pi es una placa computadora de bajo coste, desarrollado en el Reino Unido por la Fundación Raspberry Pi (Luis and Tovar, 2015). El proyecto fue ideado en 2006 pero no fue lanzado al mercado hasta febrero de 2012. Ha sido desarrollado por un grupo de la Universidad de Cambridge y su misión es fomentar la enseñanza de las ciencias de la computación en las escuelas. Los primeros diseños de Raspberry Pi se basaban en el microcontrolador Atmel ATmega644 (Raspberry, 2015). El primer prototipo basado en ARM[9] fue montado en un paquete del tamaño de una memoria USB. Tenía un puerto USB en un extremo y un puerto HDMI en el otro (Perles, 2016). En agosto de 2011, se fabricaron 50 placas Alpha del modelo inicial, modelo A y en diciembre del mismo año 25 placas Beta

[9] Nombre comercial de una arquitectura *RISC* (*Reduced Instruction SetComputer)*, Computador con Set de Instrucciones Reducido, desarrollado por ARM Holdings.

del modelo B. El primer lote de 10.000 placas se fabricó en Taiwán y China, en vez de Reino Unido, con esto se conseguía un abaratamiento en los costes de producción y acortar el plazo de entrega del producto, ya que, los fabricantes chinos ofrecían un plazo de entrega de 4 semanas y en el Reino Unido de 12 semanas. Con este ahorro conseguido, la fundación podía invertir más dinero en investigación y desarrollo. El concepto es el de un ordenador desnudo de todos los accesorios que se pueden eliminar sin que afecte al funcionamiento básico. Está formada por una placa que soporta varios componentes necesarios en un ordenador común y es capaz de comportarse como tal (Raspberry, 2013).

3.2 Descripción funcional del microcomputador Raspberry Pi

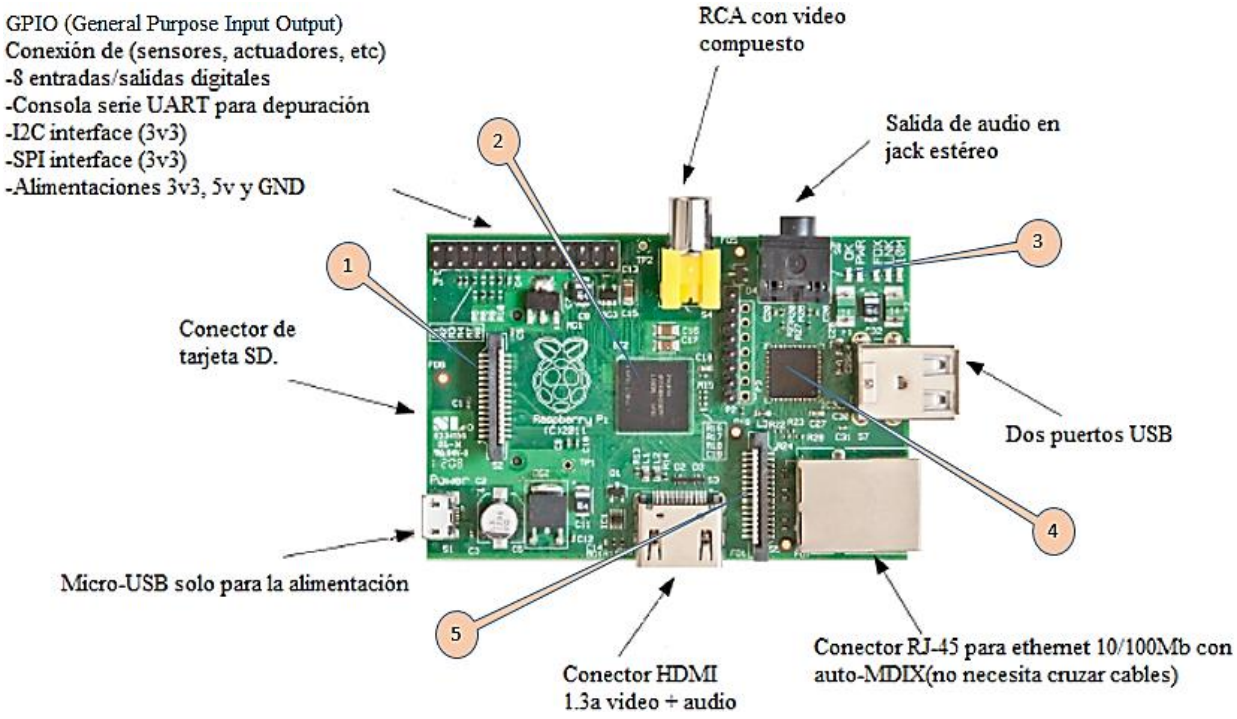


Figura 3.1 Esquema del minicomputador Raspberry Pi 2 Modelo B.

1. Conector DSI: Este conector tipo *Flat-Flex6* permitirá la conexión de pantallas tipo LCD que soporten el estándar DSI[10].

2. System on a Chip: El SoC[11] de la Raspberry Pi integra en un sólo encapsulado: un procesador de gráficos (GPU), microprocesador (CPU) de un solo núcleo, procesador digital de señales (DSP), un puerto USB y la memoria SDRAM de 256 MB (en la primera revisión) y 512 MB (en la segunda revisión). Debido a la naturaleza del ensamblaje del SoC, en ningún caso es posible expandir la memoria RAM del producto.

La memoria RAM es compartida entre el CPU y el GPU. De acuerdo a la versión del firmware cargado al dispositivo, es posible ajustar la cantidad disponible a cada subsistema con una granularidad de hasta 1 MB.

El SoC es un *BroadcomTM BCM2835*. Este fue seleccionado por la fundación por ofrecer buen rendimiento versus su costo, contiene un microprocesador ARM1176JZFS con unidad de punto flotante, corriendo a 700 MHz (nominal) y una unidad de procesamiento gráfico *Videocore 4*.

3. Leds de estado: Estos leds muestran el estado de operación actual del dispositivo, indicando el estado del puerto Ethernet, operación de lectura / escritura sobre el medio de almacenamiento masivo y la presencia de alimentación DC.

4. Controlador Ethernet: Este circuito integrado controla el puerto Ethernet en sí mismo, depende del puerto USB integrado internamente en el SoC.

---

[10] *Display Serial Interface,* Interfaz Serial para Visualizadores. Especificación creada por la organización sin fines de lucro *Mobile Industry Processor Interface Alliance (MIPI)*, para promover el desarrollo de tecnologías estándar de interfaces para terminales móviles y sus áreas de influencia.

[11] *System on a Chip,* Sistema en un Chip. Es un circuito integrado que integra todos los componentes de un computador u otro sistema electrónico en un solo chip, que puede contener una o más funciones de señal mixta, analógica, digital, memoria RAM o radiofrecuencia en un solo substrato de silicio.

5. Conector CSI-2: Este conector Flat-Flex permite la conexión al módulo de cámara propietaria de alta definición que ofrece la Fundación.

Sistema operativo recomendado para el minicomputador

La Raspberry Pi, está diseñada para ejecutar el sistema operativo GNU/Linux (Eben et al., 2014). El sistema operativo oficialmente recomendado y soportado por la Fundación es Raspbian, (una derivación de *Linux Debian Wheezy ARMhf*), aunque existen disponibles otros sistemas operativos compatibles con esta plataforma que han sido adaptados para usos específicos. Varias versiones de Linux (conocidas como distribuciones) han sido portadas al chip BCM2835 de la Raspberry Pi, incluyendo Debian, Fedora Remix y Arch Linux. Las distintas distribuciones atienden diferentes necesidades, pero todas ellas tienen algo en común: son de código abierto. Además, por lo general, todas son compatibles entre sí, esto quiere decir que es posible descargar el código fuente del sistema operativo por completo y hacer los cambios que uno desee. Nada es ocultado, y todos los cambios hechos están a la vista del público. Este espíritu de desarrollo de código abierto ha permitido a Linux rápidamente ser modificado para poder ejecutarse sobre la Raspberry Pi, un proceso conocido como portabilidad.

3.3 Instalación de Xenomai en Raspberry Pi

Se debe empezar con una tarjeta SD grabada con una imagen oficial de Raspbian descargándola de *http://www.raspberrypi.org/downloads*. En un ordenador con Linux se deben obtener las fuentes de Xenomai y del *kernel* de linux y la *toolchain* para la compilación cruzada. Para ello, se abre una terminal, se navega hasta la carpeta donde se quieran almacenar los archivos y se ejecutan los siguientes comandos:

```
mkdir XenoPi
```

```
cd XenoPi
```

Una vez en la carpeta *XenoPi* (puede ser otro nombre) se ejecuta el siguiente comando para obtener el *kernel* 3.8.13 para Raspberry Pi. A fecha de hoy la última versión de Xenomai no es compatible con *kernel* posteriores.

```
git clone -b rpi-3.8.y --depth 1 git://github.com/raspberrypi/linux.git linux-rpi-3.8.y
```

Esto clonará la rama necesaria del repositorio en la carpeta *linux-rpi-3.8.y* que se creará automáticamente. La versión 2.6.3 de Xenomai, la última estable a día de hoy, se puede descargar con los siguientes comandos:

```
wget http://download.gna.org/xenomai/stable/xenomai-2.6.3.tar.bz2

tar xvjf xenomai-2.6.3.tar.bz2
```

Esto descargará las fuentes comprimidas de Xenomai y las descomprimirá en la carpeta xenomai-2.6.3. Por último, para descargar la *toolchain* cruzada "oficial" de Raspberry Pi hay que ejecutar los siguientes comandos:

```
wget https://github.com/raspberrypi/tools/archive/master.tar.gz

tar xzf master.tar.gz
```

Luego en la carpeta *XenoPi* debería haber tres carpetas, *tools-master* con la *toolchain*, *linux-rpi-3.8.y* con las fuentes del *kernel* y xenomai-2.6.3 con las fuentes de Xenomai, además del archivo *rpi_xenomai_config*

Se deben aplicar unos parches al *kernel* de Linux para incluir entre otras cosas el planificador específico de Xenomai. Se utilizarán los parches oficiales de Xenomai 2.6.3 para el *kernel* 3.8.13. Los parches se deben aplicar utilizando los siguientes comandos:

```
cd linux-rpi-3.8.y

patch -p1 < ../xenomai-2.6.3/ksrc/arch/arm/patches/raspberry/ipipe-core-3.8.13-raspberry-pre-2.patch

patch -p1 < ../xenomai-2.6.3/ksrc/arch/arm/patches/ipipe-core-3.8.13-arm-3.patch

patch -p1 < ../xenomai-2.6.3/ksrc/arch/arm/patches/raspberry/ipipe-core-3.8.13-raspberry-post-2.patch
```

```
cd ..

xenomai-2.6.3/scripts/prepare-kernel.sh --arch=arm --linux=linux-rpi-3.8.y
```

En el comando *patch –p1* (es un uno, no una ele). Con esto el *kernel* quedaría parcheado y listo para configurar.

Antes de compilar se debe configurar el *kernel*. Se puede optar por emplear el fichero de configuración ya preparado o partir de una configuración mínima añadir los módulos y configuraciones necesarias manualmente. Para ello hay que ejecutar los siguientes comandos:

```
cp rpi_xenomai_config linux-rpi-3.8.y/.config

cd linux-rpi-3.8.y

make ARCH=arm menuconfig
```

Con este comando se abrirá un menú donde se configurarán los módulos necesarios.

Para la compilación del *kernel* se ejecuta el comando:

```
make ARCH=arm CROSS_COMPILE=/home/<USUARIO>/XenoPi/tools-master/arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi/bin/arm-bcm2708hardfp-linux-gnueabi- zImage modules
```

Para compilar e instalar las librerías de Xenomai lo más fácil es compilarlas desde la propia Raspberry Pi. Desde el PC en el que se está compilando todo se copia en la carpeta que contiene las fuentes de Xenomai a cualquier sitio de la Raspberry Pi. Para ello se hace lo siguiente suponiendo que se viene del paso anterior.

```
cd ../../../..

cp -r xenomai-2.6.3 /media/fc254b57-8fff-4f96-9609-ea202d871acf/home/pi
```

Cuando termine se extrae la SD del PC de forma segura para asegurarse de que se ha escrito todo y se introduce en la Raspberry Pi, esperando a que arranque. Ya sea vía SSH o con teclado y pantalla se ejecutan los siguientes comandos en la Raspberry Pi:

```
sudo su

cd /home/pi/xenomai-2.6.3/

./configure --host=arm-linux CFLAGS='-march=armv6' LDFLAGS='-march=armv6'

make

make DESTDIR=/ install

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/xenomai/lib/

PATH=$PATH:/usr/xenomai/bin/
```

Cuando termine se puede comprobar la correcta instalación haciendo:

```
cd /usr/xenomai/bin

./latency -p 100
```

3.4 Modelado e implementación de una tarea de control de una estructura cinemática usando Xenomai

Para realizar la tarea de tiempo real en Xenomai se utiliza un fragmento de código de un proyecto V-plotter el cual consiste en encontrar a partir de unas coordenadas (X, Y) el ángulo y la longitud de la línea.

Dicha aplicación hace uso de dos motores de paso 11-SHBX-51KT/H347 los cuales están constituidos por 4 bobinas donde una de ellas es un terminal común.

Para el funcionamiento de los motores se emplea la secuencia de paso completo que consiste en activar cada una de las bobinas de forma independiente, lo que provoca que el eje del motor se oriente hacia la bobina activa.

La Figura 3.2 muestra una masa m suspendida por dos líneas. Cada línea puede (y normalmente tendrá) un ángulo diferente al eje X.

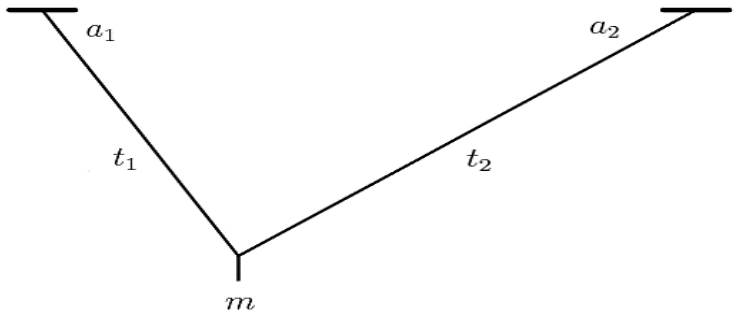


Figura 3.2 Representación geométrica.

Para describir las fuerzas horizontales a lo largo del eje X (en equilibrio), se escribe:

$$t1 \cdot \cos \alpha 1 = t2 \cdot \cos \alpha 2 \tag{3.1}$$

Para describir la fuerza m a lo largo del eje Y, causada por el peso del conjunto del cabezal del plotter, se escribe:

$$t1 \cdot \sin \alpha 1 + t2 \cdot \sin \alpha 2 = m \tag{3.2}$$

Resolviendo estas dos ecuaciones en términos de tensión, se obtiene:

$$t1 = \frac{m \cdot \cos \alpha 2}{\cos \alpha 1 \cdot \sin \alpha 2 + \sin \alpha 1 \cdot \cos \alpha 2} \tag{3.3}$$

$$t2 = \frac{m \cdot \cos \alpha 1}{\cos \alpha 1 \cdot \sin \alpha 2 + \sin \alpha 1 \cdot \cos \alpha 2} \tag{3.4}$$

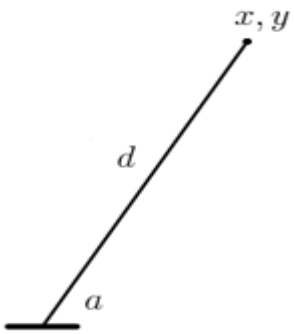


Figura 3.3 Conversión cartesiana.

Desde un ángulo y una longitud (basados en el origen), se desea encontrar una coordenada (X, Y). O partir de un par de coordenadas (X, Y) se desea encontrar el ángulo y la longitud (al origen).

$$x, y = d \cdot \cos \alpha \ , d \cdot \sin a \tag{3.5}$$

$$d, a = \sqrt{x^2 + y^2} \, , \tan^{-1}(x/y) \tag{3.6}$$

El ajuste de estas fórmulas para ubicaciones que no son de origen implica un simple ajuste de suma o resta.

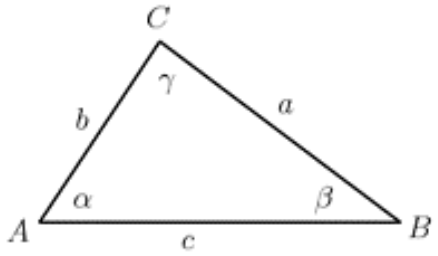


Figura 3.4 Elementos de un triángulo necesarios para aplicar la ley de los cosenos.

La ley de los cosenos permite calcular la posición del cabezal de impresión cuando una línea cambia de longitud. La forma básica de la ley es:

$$a^2 = b^2 + c^2 - 2 \cdot b \cdot \cos \alpha \tag{3.6}$$

Despejando α:

$$\alpha = \cos^{-1}\frac{b^2 + c^2 - a^2}{2 \cdot b \cdot c} \tag{3.7}$$

En ciertos puntos, un pequeño cambio en la longitud de línea resultará en un cambio demasiado grande en el sistema de coordenadas (X, Y).
Para comprobar la resolución, se ajusta a o b, luego se usa la ley de cosenos para encontrar α (alfa). Conociendo α (alfa), la longitud b y la posición A, se puede encontrar la posición C.

El Anexo VI muestra el esquema geométrico y las ecuaciones que describen el movimiento de un plotter. El Anexo VII muestra el esquema de conexión de la Raspberry Pi con los motores de paso.

Para realizar la tarea de la cinemática en Xenomai se utiliza la función *rt_task_create()*, la cual recibe como argumento la dirección de un descriptor de tarea, el nombre de la tarea, el tamaño de la pila en bytes de la tarea, la prioridad de la tarea y el modo de creación de la tarea.

*int rt_task_create (RT_TASK *task, const char *name, int stksize, int prio, int mode).*

Dicha función retorna 0 si la tarea se creó con éxito. De otra manera:

-*ENOMEM:* es devuelto si el sistema no puede obtener bastante memoria dinámica del tiempo real global para crear o registrar la tarea.

-*EXIST:* es devuelto si el nombre está ya en uso por algún objeto registrado.

-*EPERM:* es devuelto si este servicio estaba designado de un contexto asincrónico.

Una vez creada la tarea esta se queda inactiva hasta que se invoca a la función *rt_task_start(),* la cual da inicio a la tarea creada. Esta función recibe como argumento la dirección del descriptor de tarea que había sido previamente creada, la dirección de rutina del cuerpo de la tarea y un argumento de entrada que recibirá un usuario definido.

*int rt_task_start (RT_TASK *task, void(*entry)(void *arg), void *arg).*

Dicha función retorna 0 si la tarea se inició con éxito. De otra manera:

- *EINVAL* es devuelto si la tarea no es un descriptor de tarea.

- *EIDRM* es devuelto si la tarea es un descriptor de tarea eliminado.

- *EPERM* es devuelto si este servicio estaba designado de un contexto asincrónico.

Una tarea existe en el sistema desde que se llama a la función *rt_task_create()* para crearla. Para terminar con una tarea de tiempo real en Xenomai se llama a la función *rt_task_delete(),* la cual termina una tarea y libera todos los recursos del *kernel* sujetos a ella. Esta función recibe como argumento la dirección del descriptor de la tarea previamente creada para eliminarla.

*int rt_task_delete (RT_TASK *task)*

Dicha función retorna 0 si se eliminó con éxito. De otra manera:

- *EINVAL* es devuelto si la tarea no es un descriptor de tarea.

- *EPERM* es devuelto si la tarea es NULA pero no designada de un contexto de tarea, o este servicio estaba designado de un contexto asincrónico.

- *EIDRM* es devuelto si la tarea es un descriptor de tarea eliminado.

Las funciones destinadas a trabajar con los tiempos de las tareas del en el programa son *rt_task_set_periodic()* y *rt_task_wait_period().*

rt_task_set_periodic(), recibe como argumento la dirección del descriptor de tarea, la fecha inicial y el período de la tarea. A continuación, se muestra la estructura.

*int rt_task_set_periodic (RT_TASK *task, RTIME idate, RTIME period)*

Dicha función retorna 0 si la acción fue realizada con éxito. De otra manera:

-*EINVAL* es devuelto si la tarea no es un descriptor de tarea.

-*EIDRM* es devuelto si la tarea es un descriptor de tarea eliminado.

-*ETIMEDOUT* es devuelto si el *idate* es diferente a *TM_INFINITE* y representa una fecha en el pasado.

-*EWOULDBLOCK* es devuelto si el cronometrador de sistema no está activo.

-*EPERM* es devuelto si la tarea es NULA pero no designada de un contexto de tarea.

rt_task_wait_period(), espera el siguiente punto periódico de liberación. Atrasa la tarea actual hasta que el siguiente punto periódico de liberación sea alcanzado. Esta función recibe como argumento un puntero a la localización de memoria la cual será escrita con las cuentas pendientes de sobrecorrida.

*int rt_task_wait_period (unsigned long *overruns_r)*

Dicha función retorna 0 si la acción fue realizada con éxito. Si *overruns_r* no es nulo 0 es escrito en la posición de memoria apuntada. De otra manera:

-*EWOULDBLOCK:* es retornado si *rt_task_set_periodic()* no es llamado por la tarea actual.

-*ETIMEDOUT:* es retornado si ocurrió una sobrecorrida del *timer.*

-*EPERM:* es retornado si este servicio fue llamado de un contexto inválido.

Todas las funciones Xenomai mencionadas anteriormente se encuentran dentro de la biblioteca *#include <native/task.h>*.

En Anexo IV se muestra el código del programa de la estructura cinemática (V-plotter) usando de Xenomai.

3.5 Análisis económico

El software propietario es distribuido normalmente en formato binario, sin acceso al código fuente, en el cual el autor no transmite ninguno de los derechos, sino que establece las condiciones en que el software puede ser utilizado, limitando normalmente los derechos de ejecución, copia, modificación, cesión o redistribución y especifica que el propietario, sea bien aquél que lo ha desarrollado o bien quien lo distribuye, sólo vende derechos restringidos de uso del mismo, con lo que el usuario no adquiere sino que más bien alquila; es decir, el producto pertenece al propietario, desarrollador o proveedor, que concede al usuario el privilegio de utilizarlo (Caraballo, 2011).

La licencia pública general (GPL) de GNU es la licencia más ampliamente usada en el mundo del software y garantiza a los usuarios finales (personas, organizaciones, compañías) la libertad de usar, estudiar, compartir, copiar y modificar el software. Su propósito es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios (Caraballo, 2011).

Tabla 3.1 Precios de las licencias de software y hardware.

Software Propietario	**QNX**	**Lynx OS**	**VxWorks**
Licencia	$ 500 a $ 1500	$ 500 a $ 1500	Hasta $20000
Raspberry Pi			**Precio**
Incluye Pi Box y fuente de alimentación de 5V y 2.5 A			$69.99

Los precios de los softwares propietarios varían según el tipo de licencia.

Licencia de desarrollo: Se paga por las herramientas de desarrollo y el derecho a usarlas.

Licencias por explotación: Se paga por cada máquina instalada con el S.O. que se usará para la aplicación en explotación.

Contratos de mantenimiento/soporte: Se llama a la compañía si hay problemas, si se ha pagado previamente.

Xenomai se distribuye gratuitamente lo cual no requiere licencias para su uso. El costo total de un proyecto usando Xenomai y Raspberry Pi es de 69.99 dólares.

3.6 Consideraciones del capítulo

Después de lo observado en el capítulo se puede concluir que Xenomai cumple con los requerimientos temporales necesarios para realizar operaciones de tiempo real críticas en plataformas embebidas basados en arquitectura ARM como es el caso del microcontrolador Raspberry Pi. Además, constituye una plataforma con la cual el usuario puede familiarizarse rápidamente y explotar sus recursos de manera fácil debido a la documentación que existe en el sitio oficial de Xenomai. Otras de las razones por la cual Xenomai es una opción viable es que al ser software libre de código abierto posee plataformas de desarrollo gratuitas que no requieren licencias para su uso.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

Como resultado de la investigación se puede concluir que:

1. Se puede obtener un RTOS basado en software libre compilado para plataformas ARM como es el caso de Xenomai.
2. Xenomai constituye una solución genérica para lograr capacidades de tiempo real duro en Linux.
3. Con la selección de la plataforma de hardware libre Raspberry Pi basada en arquitectura ARM se pueden desarrollar aplicaciones de tiempo real.
4. Con los análisis comparativos reportados por la bibliografía como resultado del estudio de la comunidad científica se evidenció que Xenomai cumple con las latencias requeridas para desarrollar una aplicación de tiempo real duro y además con otra serie de parámetros que influyen en la selección de un RTOS como son la extensibilidad, la portabilidad y el mantenimiento.
5. Con la validación del algoritmo de control con exigencias de tiempo se pudo corroborar el funcionamiento del RTOS y del microcontrolador Raspberry Pi.

Recomendaciones

1. Extender la investigación sobre el tema de manera que se puedan desarrollar más aplicaciones en el GARP que requieran de exigencias de tiempo.
2. Explotar las demás funcionalidades y servicios de Xenomai.

REFERENCIAS BIBLIOGRÁFICAS

BAKER, T. P. & SHAW, A. C. 1989. The cyclic executive model and Ada. Real-Time Systems.

BARABANOV, M. 1997. A Linux based Real Time Operating System.

BEAL, D. 2000. RTAI (Real Time Application Interface).

BRITO, A. C. D. & PERDOMO, E. F. 2011. Sistemas Empotrados y de Tiempo Real. Universidad de Las Palmas de Gran Canaria.

BROWN, J. H. & MARTIN, B. 2010. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. *proc. of the 12th Real-Time Linux Workshop (RTLWS'12).*

BURNS, A. & WELLINGS, A. 1996. Real-Time Systems and Programming Languages. *In:* -WESLEY, A. (ed.).

BUTTAZZO, G., LIPARI, G., L., M. A. & CACCAMO, L. 2005. *Soft Real-Time Systems*, Springer.

CARABALLO, M. 2011. Tipos de licencias de software: software libre, propietario y demás.

CORBET, J., RUBINI, A. & KROAH-HARTMAN, G. 2005. Linux Device Drivers. 3th edition ed.: O'Reilly.

CRESPO, A. & ALONSO, A. 2006. UNA PANORAMICA DE LOS SISTEMAS DE TIEMPO REAL. vol. 3.

DOUGLASS, B. P. 2002. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* Addison Wesley.

DOZIO, L. & MANTEGAZZA, P. 2003. Linux Real Time Application Interface (RTAI) in Low Cost High Performance Motion Control.

EBEN, UPTON, HALFACREE & GARETH 2014. *Raspberry Pi user guide*, John Wiley & Sons.

FARAGON. 2005. Available from: http://www.voluntariado.net/

FLYNN & M., I. 2010. *Sistemas operativos*, Cengage Learning Editores.

GERUM, P. 2010. The XENOMAI Project Implementing a RTOS emulation framework on GNU/Linux.

GONZÁLEZ, R., ESCALONA, N. & HERRERA, M. ¿Tiempo real sobre Windows? Empresa de Servicios Informáticos Villa Clara, Cuba.

GUIRE, N. M. 2007. RTLinux/GPL Kickstart.

IGLESIAS, J. J. A. 1998. Sistemas de Tiempo Real: Realización en Linux.

INSAURRALDE, R. 2014. Sistemas Operativos de Tiempo Real. Universidad Católica "Nuestra Señora de la Asunción".

KOPETZ, H. 1997. *REAL-TIME SYSTEMS Design Principles for Distributed Embedded Applications*, New York / Boston / Dordrecht / London / Moscow.

KRTEN, R. 2010. System Architecture. *In:* KG., Q. S. S. G. C. (ed.). 175 Terence Matthews Crescent Kanata, Ontario, Canadá.

LUCHETTA, A., MANDUCHI, G., MORO, M., SOPPELSA, A. & TALIERCIO, C. 2007. Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real Time Application.

LUIS, M. & TOVAR, S. 2015. Minicomputador educacional de bajo costo Raspberry Pi: Primera parte. *REVISTA ETHOS VENEZOLANA,* Vol. 7**,** 31-38.

MELANSON & TAFAZOLI 2003. A selection methodology for the rtos market.

O'BOYLE, M. 2014. Embedded Software.

PERLES, Á. 2016. Empezar con la Raspberry Pi (RPi).

PROCTOR, F. 2002. Introduction to Linux for Real-Time Control. National Institute of

Standards and Technology.

PUENTE, D. L. & ANTONIO, J. 2000. *Introducción a los Sistemas en Tiempo Real*, Universidad Politécnica de Madrid.

RACCIU, G. & MANTEGAZZA, P. 2006. RTAI 3.4 User Manual.

RASPBERRY 2013. Raspberry pi.

RASPBERRY 2015. Raspberry pi 2 model b.

RIPOLL, PISA, P., ABENI, L., GAI, P., LANUSSE, A. & SAEZ, S. 2002. RTOS state of the art analysis.

RIPOLL, I., CRESPO, A. & GARCÍA-FORNES, A. 1997. An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems. 23.

ROMERO, F., IGLESIAS, L., GUISÁNDEZ, R., GIUSTI, A. E. D. & TINETTI, F. G. 2011. Experimentación y evaluación de sistemas operativos de tiempo real.

SAAVEDRA, Y. 2015. LynxOS. Available from: https://soperativosblog.wordpress.com/ [Accessed enero 10, 2017].

SPRUNT, BRINKLEY, A.C., S., MARCO, L., LEHOCZKY & JOHN 1989. Aperiodic task scheduling for hard-real-time systems.

TANENBAUM & ANDREW 1992. Modern Operating Systems Prentice Hall.

TIMMERMAN 2000. RTOS market survey preliminary results.

TUÑÓN, M. I. C. & TERRIZA, J. A. H. 2010. Actas de las XIII Jornadas de Tiempo Real.

VIVANCOS, E., HERNÁNDEZ, L. & BOTTI, V. 1997. Construcción y análisis temporal de sistemas basados en reglas para entornos de tiempo real.

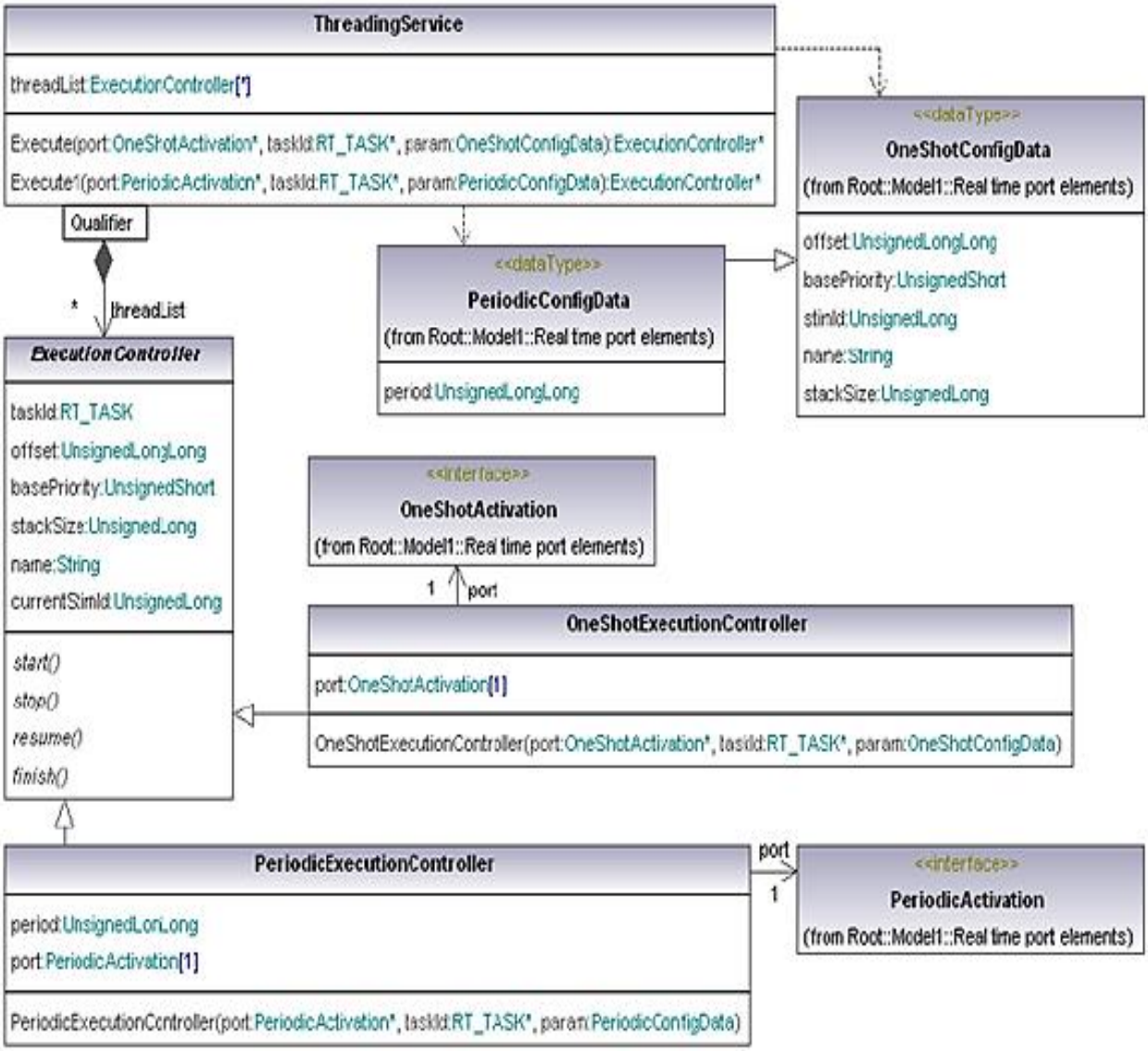
WHILSHIRE, P., ALUMINUM, K., YODAIKEN, V., NILSSON, J. & RYTTERLUND, D. 2000. RTLinux Installation Manual. Real Time Linux Documentation Project. Vol. 1.

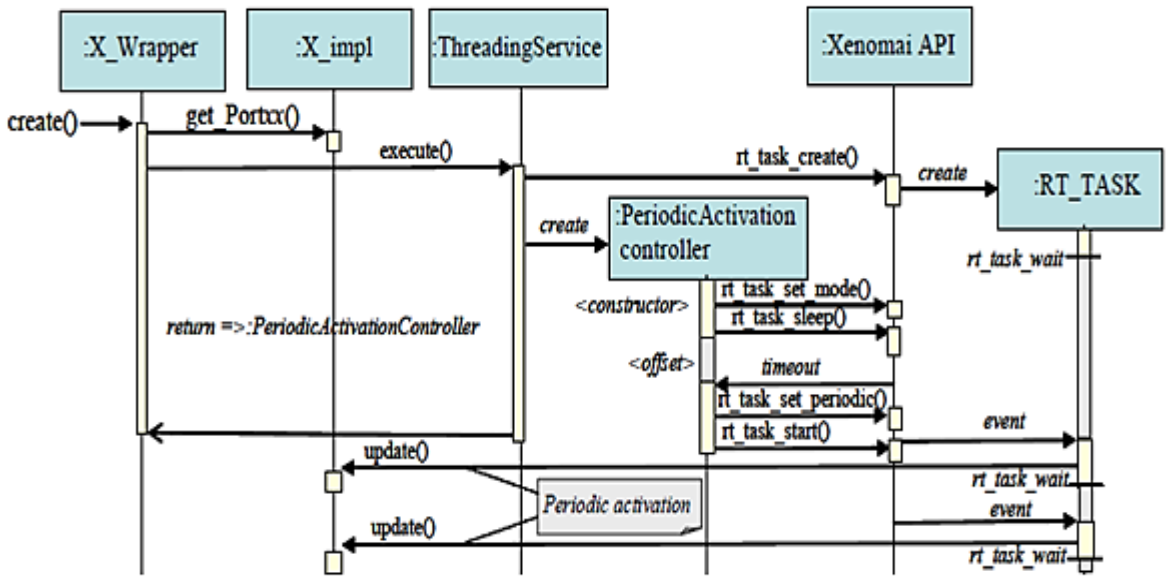
YODAIKEN, V. 1999. An Introduction to RTLinux.

ZAMARRÓN, D. L. 2004. Análisis de Sistemas Operativos de Tiempo Real.

ANEXOS

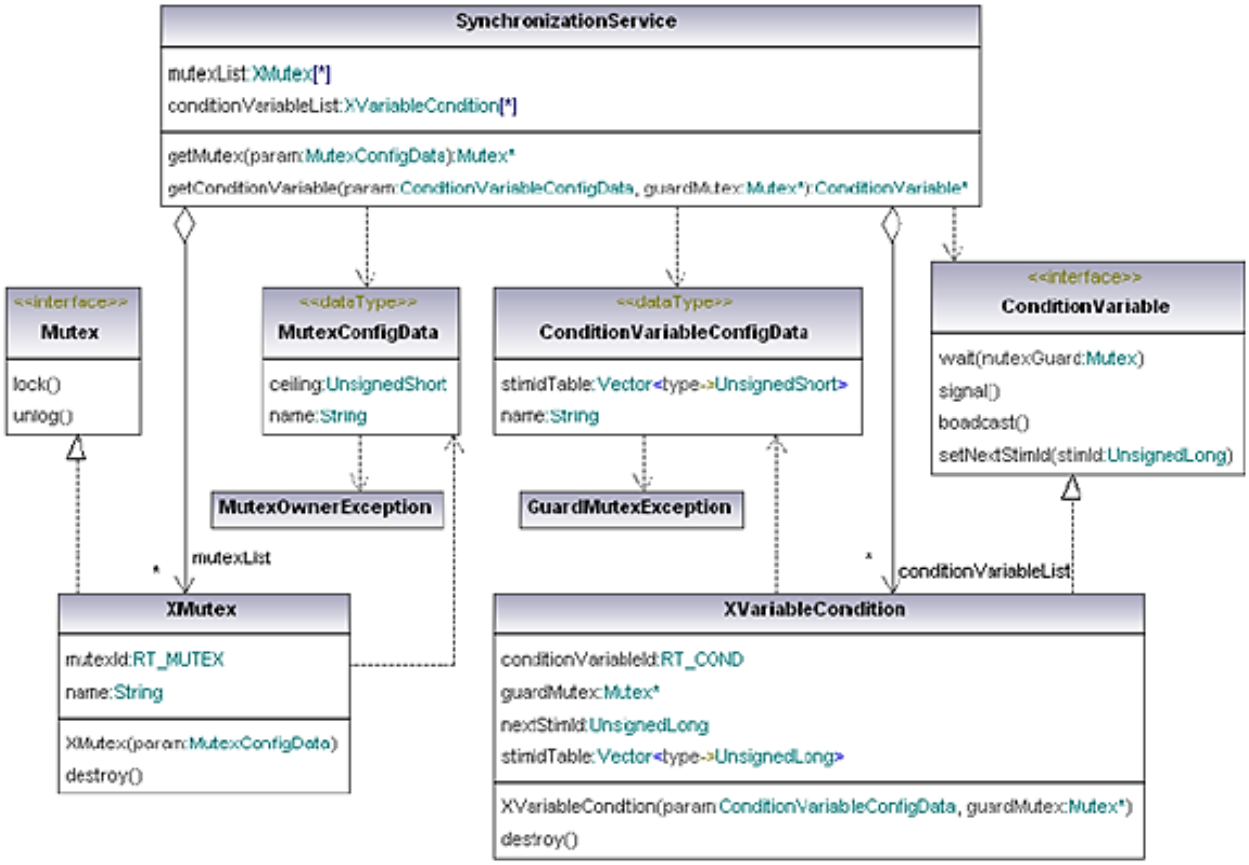
Anexo I Estructura y funcionabilidad del ThreadingService





Interacciones con el API de Xenomai para la gestión de *threads*.

Anexo II Estructura y funcionalidad del SyncronizationService



Anexo III Prueba de latencia de una tarea periódica en Xenomai

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>

RT_TASK test_task;
long PERIOD = 1000000000;

int  MAX_MUESTRAS = 10000;
long muestra_entero[MAX_MUESTRAS];
long muestra_flotante[MAX_MUESTRAS];
int  index = 0;

void test_periodicidad(void *arg)
{
        RTIME now, previous;

        rt_task_set_periodic(NULL, TM_NOW, PERIOD);
        previous = rt_timer_read();

        while (index < MAX_MUESTRAS) {
                rt_task_wait_period(NULL);
                now = rt_timer_read();
                //Tiempo desde la última ejecución,
                muestra_entero[index] = (long)(now - previous) / PERIOD;
                muestra_flotante[index] = (long)(now - previous) %
PERIOD;
                previous = now;
                   index ++;
        }
}

void catch_signal(int sig)
{
}

int main(int argc, char* argv[])
{
        signal(SIGTERM, catch_signal);
        signal(SIGINT, catch_signal);

        /* Avoids memory swapping for this program */
        mlockall(MCL_CURRENT|MCL_FUTURE);

        rt_task_create(&test_task, "test_periodicidad", 0, 99, 0);

        index = 0;

        rt_task_start(&test_task, &periodicidad, NULL);

        for (int i = 1; i < MAX_MUESTRAS; i++)
            // Este tiempo transcurrido-periodo es un test de latencia
```

```
            printf("Tiempo transcurrido entre ejecución del período:
%ld.%06ld ms\n",
                   (long)(muestra_entero[i] - muestra_entero[i-1]) /
PERIOD,
                   (long)(muestra_flotante[i] - muestra_flotante[i-1]) %
PERIOD);


        pause();

        rt_task_delete(&demo_task);
}
```

Anexo IV Control de una estructura cinemática dos articulaciones controladas por motores por paso usando Xenomai

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>
#include <math.h>
#include <wiringPi.h>

#include <native/task.h>
#include <native/timer.h>

RT_TASK cinematic_task;

long PERIOD = 10000000
int INDEX = 0;

//+++++++++++++++++++++++++++++ MakeStepLeft
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+
void MakeStepLeft(int direction){
  StepX += direction;

  if(StepX > 3){
    StepX = 0;
  }
  if(StepX < 0){
    StepX = 3;
  }

  if(StepX == 0){
    digitalWrite(LEFT_STEPPER01, 1);
    usleep(STEP_PAUSE);
    digitalWrite(LEFT_STEPPER02, 0);
    digitalWrite(LEFT_STEPPER03, 0);
    digitalWrite(LEFT_STEPPER04, 0);
  }
  if(StepX == 1){
    digitalWrite(LEFT_STEPPER03, 1);
    usleep(STEP_PAUSE);
```

```
    digitalWrite(LEFT_STEPPER01, 0);
    digitalWrite(LEFT_STEPPER02, 0);
    digitalWrite(LEFT_STEPPER04, 0);
  }
  if(StepX == 2){
    digitalWrite(LEFT_STEPPER02, 1);
    usleep(STEP_PAUSE);
    digitalWrite(LEFT_STEPPER01, 0);
    digitalWrite(LEFT_STEPPER03, 0);
    digitalWrite(LEFT_STEPPER04, 0);
  }
  if(StepX == 3){
    digitalWrite(LEFT_STEPPER04, 1);
    usleep(STEP_PAUSE);
    digitalWrite(LEFT_STEPPER01, 0);
    digitalWrite(LEFT_STEPPER02, 0);
    digitalWrite(LEFT_STEPPER03, 0);
  }

  usleep(STEP_PAUSE);
}

//+++++++++++++++++++++++++++++ MakeStepRight
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+
void MakeStepRight(int direction){
  StepY += direction;

  if(StepY > 3){
    StepY = 0;
  }
  if(StepY < 0){
    StepY = 3;
  }


  if(StepY == 0){
    digitalWrite(RIGHT_STEPPER01, 1);
    usleep(STEP_PAUSE);
    digitalWrite(RIGHT_STEPPER02, 0);
    digitalWrite(RIGHT_STEPPER03, 0);
    digitalWrite(RIGHT_STEPPER04, 0);
  }
  if(StepY == 1){
    digitalWrite(RIGHT_STEPPER03, 1);
    usleep(STEP_PAUSE);
    digitalWrite(RIGHT_STEPPER01, 0);
    digitalWrite(RIGHT_STEPPER02, 0);
    digitalWrite(RIGHT_STEPPER04, 0);
  }
  if(StepY == 2){
    digitalWrite(RIGHT_STEPPER02, 1);
    usleep(STEP_PAUSE);
    digitalWrite(RIGHT_STEPPER01, 0);
    digitalWrite(RIGHT_STEPPER03, 0);
    digitalWrite(RIGHT_STEPPER04, 0);
  }
```

```
  if(StepY == 3){
    digitalWrite(RIGHT_STEPPER04, 1);
    usleep(STEP_PAUSE);
    digitalWrite(RIGHT_STEPPER01, 0);
    digitalWrite(RIGHT_STEPPER02, 0);
    digitalWrite(RIGHT_STEPPER03, 0);
  }

//  printf("StepY\n");
  usleep(STEP_PAUSE);
}


//+++++++++++++++++++++++++++++++++++ moveXY
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void moveXY(long X, long Y){
  long newCordLengthLeft;
  long newCordLengthRight;
  double forceLeft, forceRight;
  double deltaCordLeft, deltaCordRight;
  double deltaCordLeft0, deltaCordRight0;
  double alpha;
  char TextLine[1000];

  //calculate initial stretch of cords (experimental)
  alpha = atan((double)(X0) / (double)(Y0));
  forceLeft = 1.0 * cos(alpha);
  forceRight = 1.0 * sin(alpha);

  deltaCordLeft0 = (double)(CORDLENGTH_LEFT * StepsPermm) * forceLeft *
CORDFLEXFACTOR;
  deltaCordRight0 = (double)(CORDLENGTH_RIGHT * StepsPermm) * forceRight
* CORDFLEXFACTOR;


  //forces at current coordinates (experimental)
  alpha = atan((double)(X + X0) / (double)(Y + Y0));
  forceLeft = 1.0 * cos(alpha);
  forceRight = 1.0 * sin(alpha);

  X += X0;
  Y += Y0;

  newCordLengthLeft = sqrt((double)(X * X) + (double)(Y * Y));
  newCordLengthRight = sqrt((double)((BaseLength-X) * (BaseLength-X)) +
(double)(Y * Y));

  deltaCordLeft = (double)(newCordLengthLeft) * forceLeft *
CORDFLEXFACTOR - deltaCordLeft0;
  deltaCordRight = (double)(newCordLengthRight) * forceRight *
CORDFLEXFACTOR - deltaCordRight0;


  newCordLengthLeft -= deltaCordLeft;
  newCordLengthRight -= deltaCordRight;

  while(newCordLengthLeft > CordLengthLeft){
```

```
    MakeStepLeft(1);
    CordLengthLeft++;
  }
  while(newCordLengthLeft < CordLengthLeft){
    MakeStepLeft(-1);
    CordLengthLeft--;
  }

  while(newCordLengthRight > CordLengthRight){
    MakeStepRight(1);
    CordLengthRight++;
  }
  while(newCordLengthRight < CordLengthRight){
    MakeStepRight(-1);
    CordLengthRight--;
  }


}

//+++++++++++++++++++++++++++++++++++ CalculateLine
++++++++++++++++++++++++++++++++++++++++++++++++++++++
int CalculateLine(long moveToX, long moveToY){
  char TextLine[1000] = "";
  long  tempX = 0, tempY = 0;
  int i = 0;

 // sprintf(TextLine, "Moving X: %ld, Moving Y: %ld", moveToX, moveToY);
  MessageText(TextLine, MessageX, MessageY, 0);
//  getch();


  if(moveToX - currentX != 0 && moveToY - currentY != 0){
    tempX = currentX;
    tempY = currentY;
    if(abs(moveToX - currentX) > abs(moveToY - currentY)){
      while(currentX < moveToX){
        currentX++;
        moveXY(currentX, currentY);
        currentY = tempY + (currentX - tempX) * (moveToY - tempY) /
(moveToX - tempX);
        moveXY(currentX, currentY);
        BoldLinePattern(currentX, currentY);
      }
      while(currentX > moveToX){
        currentX--;
        moveXY(currentX, currentY);
        currentY = tempY + (currentX - tempX) * (moveToY - tempY) /
(moveToX - tempX);
        moveXY(currentX, currentY);
        BoldLinePattern(currentX, currentY);
      }
    }
    else{
      while(currentY < moveToY){
        currentY++;
        moveXY(currentX, currentY);
```

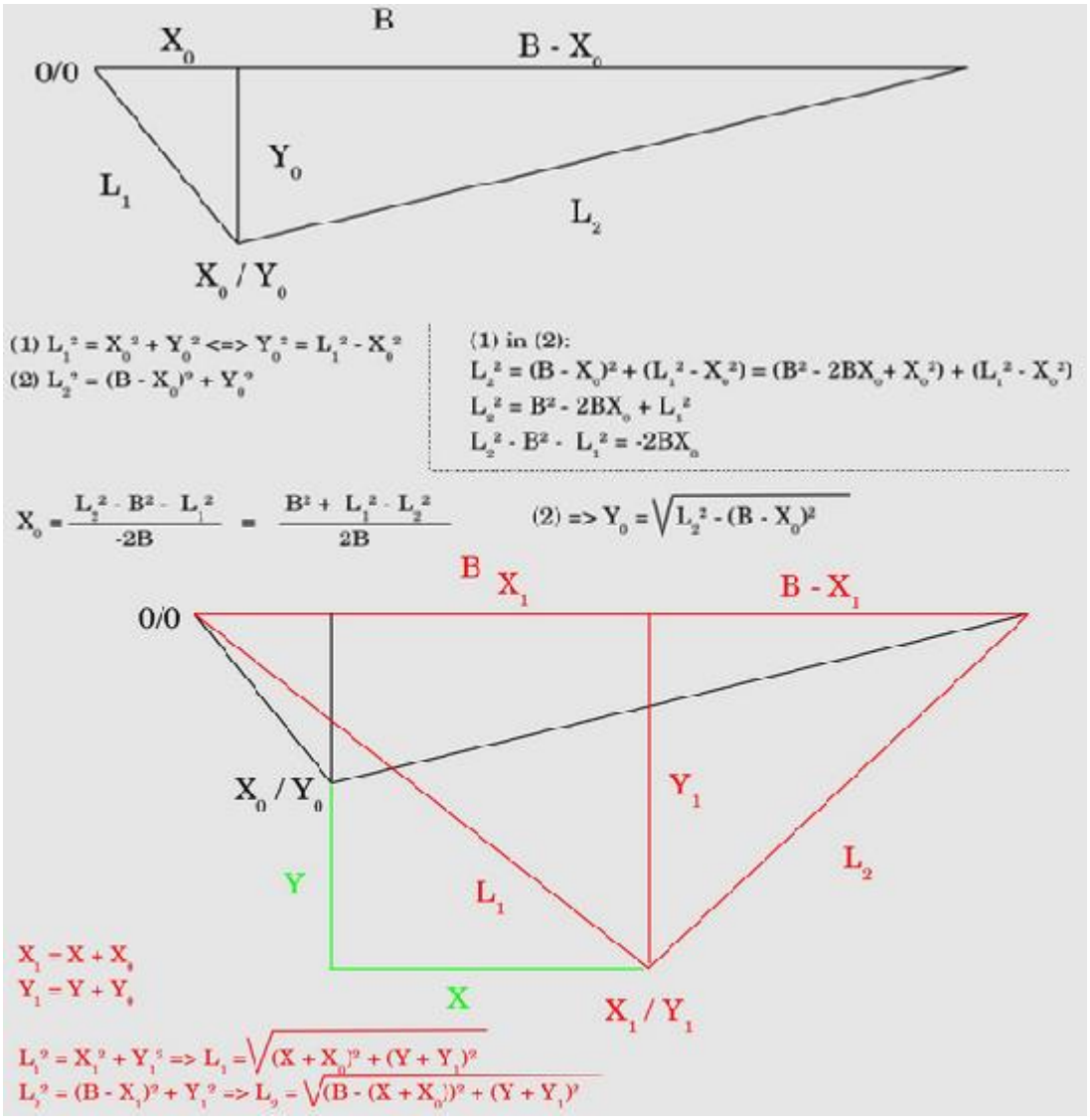
```
        currentX = tempX + (currentY - tempY) * (moveToX - tempX) /
(moveToY - tempY);
        moveXY(currentX, currentY);
        BoldLinePattern(currentX, currentY);
      }
      while(currentY > moveToY){
        currentY--;
        moveXY(currentX, currentY);
        currentX = tempX + (currentY - tempY) * (moveToX - tempX) /
(moveToY - tempY);
        moveXY(currentX, currentY);
        BoldLinePattern(currentX, currentY);
      }
    }
  }


  while(moveToY > currentY){
    currentY++;
    moveXY(currentX, currentY);
    BoldLinePattern(currentX, currentY);
  }
  while(moveToY < currentY){
    currentY--;
    moveXY(currentX, currentY);
    BoldLinePattern(currentX, currentY);
  }

  while(moveToX > currentX){
    currentX++;
    moveXY(currentX, currentY);
    BoldLinePattern(currentX, currentY);
  }
  while(moveToX < currentX){
    currentX--;
    moveXY(currentX, currentY);
    BoldLinePattern(currentX, currentY);
  }

  return 0;
}
//------------------------------------ CalculateLine -----------------
-------------------------------------

void cinematica(void *arg)
{
        RTIME now, previous;

        /*
         * Arguments: &task (NULL=self),
         *            start time,
         *            period (here: 10 ms)
         */
        rt_task_set_periodic(NULL, TM_NOW, PERIOD);

        while (trayetoria_terminada == FALSE) {
                rt_task_wait_period(NULL);
```

```
                if (INDEX < MaxPoint){
                   CalculateLine (trayectoriaX[INDEX],
trayectoriaY[INDEX]);
                   INDEX ++;
                }
                else
                   trayectoria_terminada = TRUE;


        }
}

void catch_signal(int sig)
{
}

int main(int argc, char* argv[])
{
        signal(SIGTERM, catch_signal);
        signal(SIGINT, catch_signal);

        /* Avoids memory swapping for this program */
        mlockall(MCL_CURRENT|MCL_FUTURE);

        /*
         * Arguments: &task,
         *            name,
         *            stack size (0=default),
         *            priority,
         *            mode (FPU, start suspended, ...)
         */
        rt_task_create(&cinematic_task, "cinematica", 0, 99, 0);

        //.................
        // Algoritmo de leer la trayectoria.
        // Son funciones que parsean el archivo y leen las coordenadas
cartsianas.
        // No se ponen en este anexo para simplificar el programa
        //-------------------

        INDEX = 0;
        /*
         * Arguments: &task,
         *            task function,
         *            function argument
         */
        rt_task_start(&cinematic_task, &cinematica, NULL);

        pause();

        rt_task_delete(&cinematic_task);
}
```

Anexo V Otras funciones de las tareas de la API nativa de Xenomai

- int rt_task_delete (RT_TASK *task)
  Borra una tarea de tiempo real
- int rt_task_set_affinity (RT_TASK *task, const cpu_set_t *cpus)
  Establece la afinidad de la CPU con la tarea de tiempo real
- int rt_task_start (RT_TASK *task, void(*entry)(void *arg), void *arg)
  Inicia una tarea de tiempo real
- int rt_task_shadow (RT_TASK *task, const char *name, int prio, int mode)
  Retorna la llamada dentro de la tarea de tiempo real
- int rt_task_join (RT_TASK *task)
  Espera por la terminación de una tarea de tiempo real
- int rt_task_wait_period (unsigned long *overruns_r)
  Espera el siguiente punto periódico de liberación
- int rt_task_sleep (RTIME delay)
  Atrasa la tarea de tiempo real actual (con atraso relativo)
- int rt_task_sleep_until (RTIME date)
  Atrasa la tarea de tiempo real actual(con fecha absoluta de despertar)
- int rt_task_same (RT_TASK *task1, RT_TASK *task2)
  Compara los descriptores de las tarea de tiempo real
- int rt_task_suspend (RT_TASK *task)
  Suspende una tarea de tiempo real
- int rt_task_resume (RT_TASK *task)
  Resume una tarea de tiempo real
- RT_TASK * rt_task_self (void)
  Recupera el descriptor actual de la tarea
- int rt_task_set_priority (RT_TASK *task, int prio)
  Cambia la prioridad base de una trea de tiempo real

Anexo VI Representación geométrica y ecuaciones que describen el movimiento del plotter



Anexo VII Esquema de conexión de la Raspberry Pi con los motores de paso