Universidad de Oriente

Facultad de Ingeniería Eléctrica

Departamento de Telecomunicaciones

TRABAJO DE DIPLOMA

Software soportado en FreeRTOS sobre FPGA para interfaz IEEE-1394 y una PC vía USB.

Autor: Daniel Eugenio Pérez Guzmán

Tutores: M.Sc. Ing. Berta Pallerols Mir

M.Sc. Ing. Alexander A. Suárez León

Santiago de Cuba

Junio, 2015

Universidad de Oriente

Facultad de Ingeniería Eléctrica

Departamento de Telecomunicaciones

TRABAJO DE DIPLOMA

Software soportado en FreeRTOS sobre FPGA para interfaz IEEE-1394 y una PC vía USB.

Autor: Daniel Eugenio Pérez Guzmán

daniel.perez@tle.fie.uo.edu.cu

Tutores: M.Sc. Ing. Berta Pallerols Mir[1]

M.Sc. Ing. Alexander A. Suárez León[2]

[1]Profesor Auxiliar, Departamento de Telecomunicaciones,
Facultad de Ingeniería Eléctrica, e-mail: bertapm@fie.uo.edu.cu

[2]Profesor Asistente, Departamento de Ingeniería Biomédica,
Facultad de Ingeniería Eléctrica, e-mail: aasl@fie.uo.edu.cu

Santiago de Cuba

Junio, 2015

COMPROMISO DEL AUTOR

Hago constar que el presente trabajo de diploma es de mi autoría exclusivamente, no constituyendo copia de ningún trabajo realizado anteriormente y las fuentes usadas para la realización del trabajo se encuentran referidas en la bibliografía. Doy mi consentimiento a que el mismo sea utilizado por la Institución, para los fines que estime conveniente, tanto de forma parcial como total, además no podrá ser presentado en eventos, ni publicados sin autorización del Tutor o Institución.

_____

Firma del Autor

PENSAMIENTO

"... Milito en el grupo de los impacientes, y milito en el bando de los apurados, y de los que siempre presionan para que las cosas se hagan, y de los que muchas veces tratan de hacer más de lo que se puede..."

Fidel Castro Ruz

DEDICATORIA

A mi familia, en especial a mis padres, por su cariño, apoyo y aliento en estos años de estudio.

A mi esposa Aimé por todo el amor, sacrificio y apoyo en cada momento de mi vida.

A mi hijo Liam Daniel regalo anticipado a mi defensa.

AGRADECIMIENTOS

A mi familia, mis padres Clara Guzmán Góngora y Mario Pérez Pérez por ser siempre mi ejemplo a seguir. A mis hermanos. A mi esposa, mi hijo y suegros.

A mis tutores quienes depositaron toda su confianza en mí, para la realización de este trabajo, y además aportaron sus ideas, conocimientos, tiempo y experiencias: Berta Pallerols Mir y Alexander A. Suárez León

A mis amigos y todas las personas que de una forma u otra contribuyeron a la realización de este trabajo y a mi formación profesional.

RESUMEN

En el presente trabajo se diseñó un *software* con un sistema operativo de tiempo real, gratuito de fuentes abiertas FreeRTOS, sistema para ser portado sobre Arreglos Programables de Campos de Compuertas (FPGA) definido en Lenguaje de Descripción de *Hardware* de Alta Velocidad (VHDL), usado para representar las capas más bajas del diseño del dispositivo final, empleado para interconectar cualquier dispositivo que implemente el protocolo de comunicación IEEE-1394 a una computadora vía Bus Universal en Serie (USB). En el mismo se presentan los fundamentos de la comunicación usando el protocolo IEEE-1394, el cual se estudia a profundidad. También se explican las principales razones de selección del sistema operativo FreeRTOS para crear el *software,* así como, sus ventajas frente a otros sistemas y las mejoras que tiene el uso de sistemas multitareas o multihilos. Por último, se ofrecen algunos programas de muestra para usarlos como base para futuras implementaciones.

Palabras clave: Sistema operativo, FPGA, VHDL, FreeRTOS, IEEE-1394.

ABSTRACT

In this work, a software was performed using a real-time operating system, free with open source FreeRTOS, system to be carried onField Programmable Gate Array (FPGA) described inVery High Speed Hardware Description Language (VHDL), used to describe the lowest layers in the design of the final device, used to interconnect any device that implements the communication protocol IIEEE-1394 to a computer way Universal Serial Bus (USB). In the same are presented the fundamentals of communication using the IEEE-1394, protocol which is studied in depth. The main reasons why the FreeRTOS operating system was selected to create the software and its advantages over other systems and the improvements that have the use of multitasking or multithreaded systems are explained. Finally offer some sample programs are provided to use as a base for future implementations.

Keywords: *Operating system, FPGA, VHDL, FreeRTOS, IEEE 1394.*

ÍNDICE

INTRODUCCIÓN

En 1986 *Apple™ Computer*, hoy *Apple™* [1], presentó una especificación de bus de alto rendimiento para microcomputadoras que denominó *FireWire™* [2], como un medio para simplificar la red de interconexión de cables en sistemas de computadoras personales y proporcionar un mecanismo para transferir bloques de datos en tiempo real a alta velocidad. El bus se mostraba como una interfaz de comunicación serie de alta velocidad entre periféricos y la computadora con direccionamiento de 64 bits, según el estándar IEEE Std 1212-1991 *Command and Status Register Architecture (CSR)* [3].

También ofrece un modelo punto a punto, en el que cualquier dispositivo puede comunicarse directamente con cualquier otro, siempre que utilicen el mismo protocolo, brindando una mayor flexibilidad en la conexión, ello elimina la dependencia de un control centralizado. Asimismo, estos dispositivos pueden ser conectados o desconectados del bus sin que se produzca algún fallo en el sistema, ya que los cambios de topología son reconocidos automáticamente por los diferentes mecanismos y elementos de control.

En 1994 la especificación inicial fue sometida a evaluación por el comité de estandarización del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) y posteriormente, en diciembre de 1995 es aprobada por dicha institución como el estándar IEEE Std 1394-1995, IEEE *Standard for a High Performance Serial Bus* [2].

Desde su surgimiento el estándar ha evolucionado a través de una serie de revisiones y suplementos desde 1394a [4] hasta las más avanzadas *FireWire* s1600 y s3200 (IEEE 1394-2008) anunciados en diciembre de 2007. Esto permite un incremento del ancho de banda de 1,6 y 3,2 Gbit/s, cuadruplicando la velocidad del *FireWire* 800 [5]. Resulta ideal su utilización en aplicaciones multimedia y almacenamiento, como videocámaras, discos duros y dispositivos ópticos.

Paralelamente a la ampliación y perfeccionamiento del estándar surgen nuevos campos de aplicación del protocolo, algunos de ellos, de rápido crecimiento en los últimos años como: la industria electrodoméstica (electrónica de consumo), industria de ordenadores personales, industria automovilística, industria militar y aeroespacial, medios no estándares e instrumentación y control industrial.

Algunos ejemplos de la aplicación de este protocolo en los referidos sectores son: dentro de la electrónica de consumo IEEE – 1394 ha encontrado aplicación importante en equipos electrodomésticos de AV (Audio/Vídeo) y sus tecnologías asociadas. En la industria de la computadora personal se han definido nuevas normas para la aplicación de dicho estándar, tanto en periféricos como en la propia computadora. En la industria automovilística las aplicaciones están orientadas a sistemas de entretenimiento.

La industria militar y aeroespacial utiliza principalmente el estándar IEEE 1394b [5] en el transbordador espacial de la NASA para monitorear restos de espuma o hielo [6] que pueden chocar con la nave durante el lanzamiento. IEEE – 1394 se ha expandido a nuevos medios físicos diferentes a los de la especificación inicial, conocidos como no estándares. Estas especificaciones adicionales se asocian a redes domésticas y la tecnología inalámbrica. En el campo del control y la instrumentación industrial también se ha aplicado este protocolo, destacándose, las cámaras destinadas al usuario con fines industriales y de instrumentación, que permiten el control sincronizado e intercambio de datos en tiempo real para dispositivos como sensores, actuadores, motores, entre otros.

En Cuba, diversas instituciones hospitalarias cuentan con tecnologías médicas que emplean cámaras compatibles con la norma IIDC, ejemplo, la cámara de retina TRC 50-DX [7] del fabricante de equipos médicos TOPCON. El Centro Oftalmológico Clínico Quirúrgico del Hospital Dr. Juan B. Zayas de la provincia de Santiago de Cuba, cuenta con cámaras para tomar imágenes o video del segmento posterior del ojo a través de la pupila. Estos dispositivos están equipados con dos cámaras, una de las cuales es la AVT Dolphin F-201B [8] compatible con IIDC versión 1.31.

En laboratorios de microscopía resulta de interés el procesamiento de imágenes digitales fijas y en movimiento para su estudio inmediato o a posteriori. Se emplean cámaras acopladas a microscopios ópticos basadas en IIDC, para el estudio y seguimiento de procesos biológicos con fines metodológicos e investigativos.

En el año 2010 comenzó un proyecto desarrollado por el grupo de Electrónica Programable Ni, del Departamento de Ingeniería Biomédica de la Universidad de Oriente con el objetivo general de implementar la totalidad del protocolo IEEE – 1394 para la comunicación con la cámara BST – HDCE, sobre una estructura electrónica distinta a una computadora. Esta implementación tiene como objetivo permitir la transferencia de imágenes y vídeo desde la cámara BST – HDCE hacia una computadora que no soporte

este protocolo. Este trabajo se inscribe dentro de este proyecto encargándose del desarrollo de un software que sirva de base para la comunicación de dispositivos IEEE-1394 y una PC vía USB.

Antecedentes del problema

El laboratorio de microscopía del Departamento de Ingeniería Biomédica de la Universidad de Oriente, dispone de la cámara digital BST – HDCE [9] para la adquisición de imagen y vídeo en tiempo real. Actualmente, esta institución cuenta con una única computadora que incorpora una interfaz IEEE – 1394 en la placa base ASUS P5L – VM 1394. El soporte en *hardware* lo proporciona un chip VLSI de VIA, específicamente VIA Fire II VT6308 1394a [10]. Este chip provee soporte OHCI e interfaz PCI 2.1. La computadora disponible opera con el sistema operativo (SO) Windows y el *driver* está contenido en un CD que acompaña la cámara.

En estas condiciones, la explotación de la cámara se limita a esta única computadora, pues no se dispone de otras que incorporen dicha tecnología. Otro aspecto negativo que lastra la posibilidad de hacer extensiva la explotación de este dispositivo, es que para configurar la cámara resulta necesaria la ejecución de un *software* que se comunica con el *driver* en la computadora; de manera que acciones básicas como encender, apagar y establecer el modo de operación de la cámara requieren el uso de este *software*.

Problema de investigación

La inexistencia de un *software* que permita el manejo y transmisión de información en tiempo real entre dispositivos IEEE-1394 y una PC vía USB sobre una plataforma en FPGA.

Objeto de estudio

Sistemas operativos de tiempo real sobre FPGA

Campo de acción

Software de comunicación sobre sistema operativo de tiempo real en FPGA con soporte para protocolo IEEE – 1394.

Objetivo

Diseñar un *software* que permita la transmisión de información en tiempo real entre dispositivos IEEE – 1394 y una PC vía USB en una plataforma embebida en una FPGA.

Hipótesis

Si se diseña un *software* que permita implementar el protocolo de comunicación IEEE – 1394 sobre un sistema operativo de tiempo real en una plataforma embebida sobre una FPGA, se soportará la comunicación entre cualquier dispositivo IEEE-1394 y una PC vía USB.

Tareas

- Caracterizar el protocolo de comunicación IEEE – 1394.
- Evaluar los requisitos principales que debe cumplir un sistema operativo que soporte el protocolo IEEE – 1394.
- Caracterizar el sistema operativo basado en FreeRTOS que se utiliza en el trabajo.
- Caracterizar la plataforma Wishbone-Plasma.
- Hacer portable el sistema operativo sobre la plataforma Wishbone-Plasma.
- Elaborar programas de prueba para analizar el desempeño del sistema operativo.
- Diseñar el *software* de comunicación para el dispositivo sintetizado en FPGA.

Métodos de trabajo

La investigación se desarrolla sobre la base de la concepción **dialéctico materialista** partiendo de los principios establecidos para el diseño de sistemas operativos.

En el proceso investigativo se utilizó el método de **análisis y síntesis** para el estudio del contenido referente al diseño de sistemas operativos en la bibliografía especializada y determinar los principales bloques funcionales del diseño y su interrelación, así como en la elección de las herramientas necesarias para su realización.

Otro método utilizado es el de **inducción-deducción**, ya que se partió de la visión general para determinar los aspectos esenciales del diseño a partir de modelos existentes. También se usó el método de **modelación y simulación** en el proceso creativo.

Se emplean los métodos **teóricos y prácticos** para la concepción del diseño partiendo de la fundamentación teórica que sustenta los sistemas operativos para *hardware* programable FPGA.

CAPITULO 1 . ESTÁNDAR IEEE - 1394.

El presente capítulo caracteriza el protocolo de comunicación IEEE – 1394 utilizado para la comunicación serie de alta velocidad y la interconexión de gran cantidad de dispositivos. Se especifican las velocidades de trabajo, así como los distintos modos de comunicación, los que favorecen diversas aplicaciones. Además se presentan varias configuraciones de cable en dependencia de la aplicación para la que se desee. Ello posibilita evaluar los requisitos principales que debe cumplir un sistema operativo cuyo soporte sea el protocolo objeto de estudio en el capítulo.

1.1 Características del estándar IEEE 1394

El estándar IEEE 1394-1995[2] o bus HPSB (*High Performance Serial Bus*) describe un bus serie de alta velocidad propuesto inicialmente por *Apple*™ con el nombre de *FireWire*™ y después por el Comité de Estándares de Microprocesadores y Microcomputadoras de la Sociedad de Computación del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) para su estandarización en 1995. A su vez, basado en el estándar ISO/IEC 13213:1994 (ANSI/IEEE 1212) [2], que describe una arquitectura de comunicación entre buses de sistemas en microcomputadoras, a través de Registros de Comando y Estado (CSR). IEEE-1394 soporta velocidades de transferencia de 100, 200 y 400 megabits por segundo permitiendo su conexión con cuatro tipo de conectores, de 4, 6, 9 y 12 terminales.

Tiene gran flexibilidad de conexión al permitir un máximo de 63 dispositivos con longitudes de cable de hasta 425 cm. Posee respuesta momentánea, puede garantizar una distribución de los datos en perfecta sincronía, su alimentación por el bus y puede alcanzar hasta 40 V de corriente directa (DC).Utiliza el modo conocido como *plug&play* (conectar, enchufar y listo) y además permite realizar conexiones en caliente (conectar dispositivos con la computadora encendida).

Este estándar fue sujeto a actualizaciones para mejorar su velocidad y aumentar su ancho de banda según [2] entre las que se destacan: *FireWire* 400, *FireWire* 800, *FireWire* s1600 y *FireWire* s3200. Se liberan algunos sectores a la interpretación de los

desarrolladores de las distintas aplicaciones. Aunque el estándar fue ampliamente utilizado, su escasa popularidad entre los fabricantes respecto a otras tecnologías, ha dado lugar a que los dispositivos periféricos actuales no posean esta interfaz, sino vengan provistos tan solo de puertos USB.

1.2 Topología

El protocolo IEEE-1394 presenta una red entre pares (nodos) con un ambiente de señalización punto a punto. Cada nodo dentro del bus puede tener varios puertos, donde cada uno de ellos actúa como repetidor, retransmitiendo los paquetes que se reciben de otros puertos dentro del nodo. Como es una implementación par a par no hay necesidad de seleccionar una anfitriona específica para interconectar con la computadora como se muestra en la Figura 1.1.

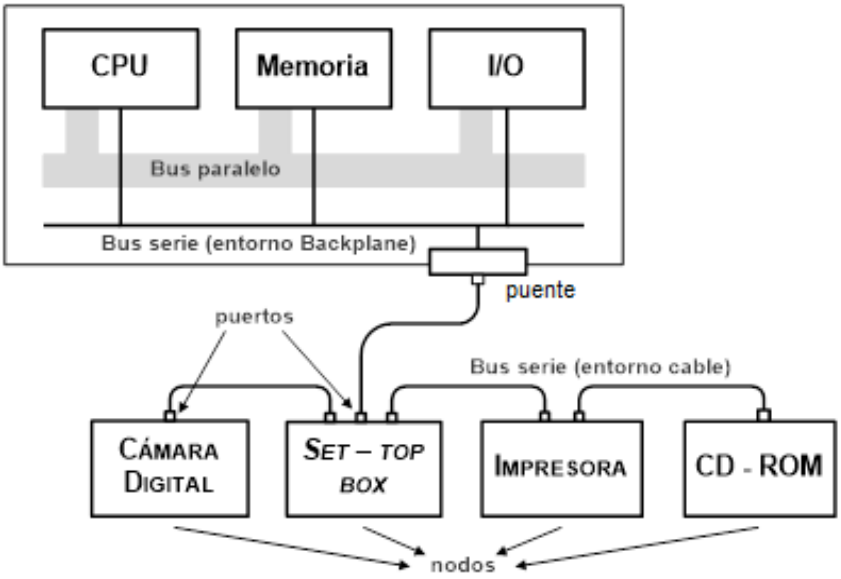

Figura 1.1.Topología física del bus serie.

En este estándar se especifican dos medios físicos distintos, cable y *backplane* o vías de circuito impreso, sobre los cuales se implementa la topología del bus. Los nodos pueden interconectarse entre sí en cualquier medio sin restricciones y para conectar buses con medios físicos distintos se debe utilizar un puente.

Al emplear cable la interconexión entre los nodos es a través de hilos conductores de cobre por lo que su topología es una red no cíclica de ramas y extensión finita, donde los lazos cerrados (bucles) no están soportados. El medio consta de dos pares de

conductores para señales de datos y un par para la alimentación. El par alimentación–tierra permite a la capa física de cada nodo continuar en funcionamiento incluso si la alimentación local del nodo desaparece.

El par puede suministrar energía a un nodo completo si sus requerimientos son moderados al proveer de 8 - 40 V de DC y hasta 1,5 A. En el caso de los pares de transmisión de señales, en sus versiones iniciales se definían tres velocidades distintas de transmisión: 100 Mbps, 200 Mbps y 400 Mbps, aunque luego fueron alcanzadas velocidades superiores. En el medio *backplane* la interconexión es a través de vías en el circuito impreso o PCB con una topología en forma de bus multipunto o tipo *daisy-chain*. El medio consta de dos conductores de terminación simple a lo largo del PCB que permiten a los nodos conectarse al bus con un OR cableado, definiéndose dos velocidades 25 y 50 Mbps [2].

1.3 Modos de transferencia asíncrono e isócrono

Este protocolo soporta dos tipos de servicios básicos de transferencia de datos: asíncronos e isócronos. En el caso de la transferencia de datos isócronos, proporciona un protocolo de entrega de paquetes de tamaño variable transferidos a intervalos regulares donde la difusión puede ser de uno a uno o de uno a muchos. Este servicio no es tolerante a errores, ni posee acuse de recibo, por lo que se emplea en aplicaciones donde es más importante la velocidad de la información y donde la transmisión de unos bits erróneos no es significativa, como en flujo de video o sonido. Mientras la transferencia de datos asíncronos proporciona un protocolo de entrega de paquete de tamaño variable a un nodo específico con una dirección explícita y retorna una confirmación de recepción conocida como acuse de recibo o *acknowledge* [11]. Esto permite la comprobación de errores y la retransmisión de datos en caso de existencia de los mismos.

1.4 Arquitectura del protocolo

El protocolo IEEE - 1394 describe cuatro capas fundamentales: la capa física encargada de transmitir y recibir las señales codificadas en dígitos binarios que recibe a través de los medios físicos (alambres de cobre, fibra óptica o medio inalámbrico); la capa de enlace se encarga de preparar los paquetes [12] para su transmisión y control del acceso a los medios físicos; la capa de transacciones emplea el modo de transferencia asincrónica ya

que posee un mecanismo de solicitud y respuesta, por último, el nivel de control del bus se encarga del manejo de este y conecta entre sí todas las capas.

En este protocolo el servicio de transferencia de datos asíncronos recibe y entrega datos desde y hacia la capa de transacciones, mientras que el servicio de transferencia de datos isócronos recibe y entrega directamente desde y hacia la capa de aplicaciones. Las capas y su interrelación se muestran en la Figura 1.2.

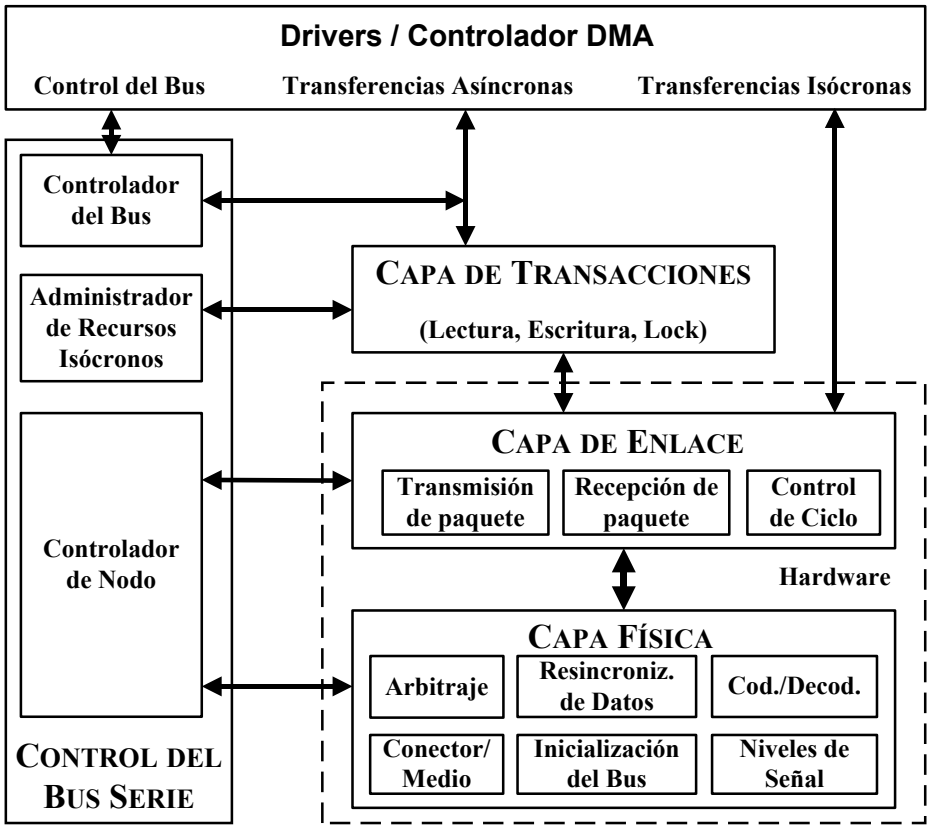

Figura 1.2. Estructura en capas del protocolo IEEE – 1394

1.4.1 Capa física

La capa física [13] del protocolo en cuestión incluye las características eléctricas, los conectores mecánicos y el cableado. Cumple tres funciones principales: transmisión y recepción de bits, arbitraje y suministro de la interfaz electro - mecánica del bus. Los entornos cable y *backplane* tienen capas físicas diferentes, no obstante, las capas físicas de ambos comparten dos conceptos fundamentales: codificación *data - strobe* no retorno

a cero (NRZ) para los bits de datos y un método simple para asegurar el acceso igualitario al bus.

La capa física, en el entorno cable: es una red de nodos conectados por enlaces punto a punto llamados conexiones físicas. La conexión física está constituida por un puerto de cada nodo y el cable entre ellos, puede tener múltiples puertos dado que permite una interconexión ramificada. Su restricción principal es que los nodos no se pueden conectar a través de un lazo cíclico al no permitir bucles. Ella traduce esta topología física punto a punto en un bus de multidifusión virtual, toma los datos que entran por el puerto de recepción, los resincroniza con respecto a un reloj local, y transmite los datos resincronizados al resto de sus puertos.

1.4.2 Capa de enlace

La capa de enlace es la interfaz entre la capa de transacciones y la capa física, responsable de calcular los códigos de redundancia cíclica de los paquetes transmitidos para anexarle esta prueba a cada uno de ellos. También provee los servicios de transferencia de datos semidúplex entre un nodo fuente y un nodo destino, así como la confirmación de recepción de datagrama a la capa de transacciones. Proporciona direccionamiento, comprobación y estructura de datos para la transmisión y recepción de paquetes. También brinda servicio de transferencia de datos isócronos directamente a la capa de aplicación, incluyendo la generación del paquete de inicio de ciclo empleado en la sincronización de los nodos.

1.4.3 Direccionamiento en el bus

El direccionamiento en el bus sigue la arquitectura CSR de 64 bits, donde los 16 bits más significativos de cada dirección conforman el campo identificador de nodo, ello aporta espacio de direccionamiento de hasta 64 k nodos. Internamente está dividido en dos campos: los 10 bits más significativos especifican el identificador de bus y los 6 bits menos significativos especifican el identificador físico. Cada uno de los campos reserva el valor de todos en uno (1) para propósitos especiales, por lo que este esquema de direccionamiento proporciona 1023 buses, cada uno con 63 nodos direccionables independientemente [14].

Los 48 bits menos significativos en esta arquitectura constituyen el espacio de nodo (256 TB). Este espacio se divide en: espacio inicial de memoria, espacio privado, espacio inicial de registro y espacio inicial de unidad. El espacio inicial de registro es de 2048 bytes (2 kB) reservados para recursos de la arquitectura CSR, registros específicos del bus y los primeros 1024 bytes para la memoria de solo lectura (ROM) de identificación. El espacio inicial de unidad es el área reservada para los recursos específicos de un nodo. El espacio privado está reservado para el uso local de los nodos. En la Figura 1.3 se muestra esta distribución.

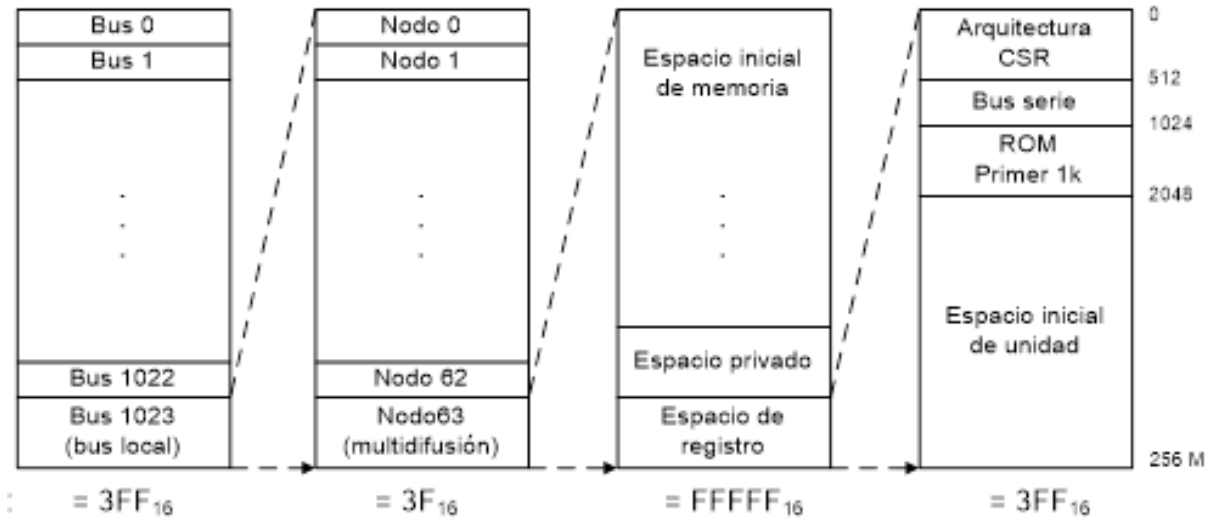

Figura 1.3. Direccionamiento del Bus Serie.

1.4.4 Estructura de los paquetes de la capa de enlace

La unidad básica de datos empleada en el protocolo en la capa de enlace es el paquete. El protocolo IEEE-1394 ha definido tres tipos fundamentales de paquetes:

- Paquetes de capa física.
- Paquetes de confirmación de recepción o acuse de recibo.
- Paquetes primarios.

De estos paquetes la capa de enlace debe ser capaz de enviar los de tipo 2 ó 3 y de recibir cualquiera de los tres.

Los paquetes primarios son una secuencia de dos o más *quadlets* (32 bits) que contienen al menos un encabezado y podrían contener (no es obligatorio) un bloque de datos. Los paquetes primarios comparten un formato común, mostrado en la Figura 1.4.

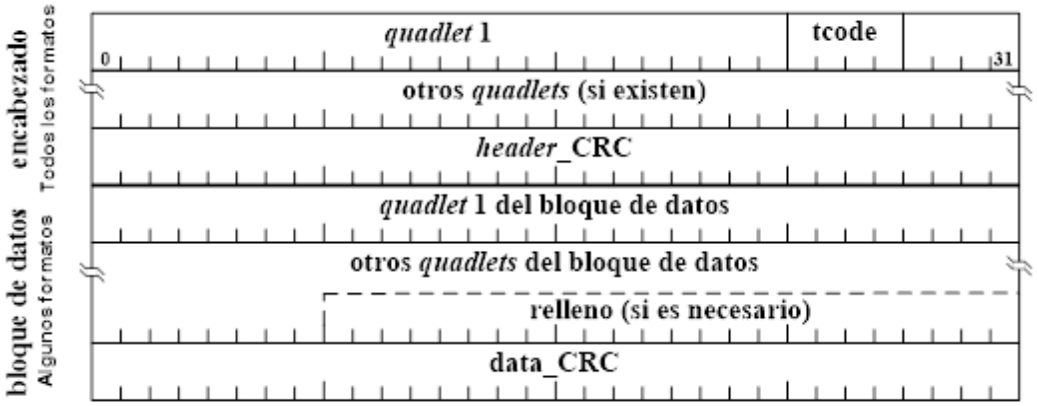

Figura 1.4. Formato de los paquetes primarios.

Todos los encabezados de los paquetes primarios contienen uno o más *quadlets* seguidos del código de redundancia cíclica (CRC) del encabezado [15]. Un nodo no debe generar un paquete de confirmación si el CRC del encabezado del paquete recibido no es correcto. El primer *quadlet* del encabezado de todos los paquetes primarios contiene el código de transacción, en el que se define el tipo de paquete primario: asíncrono o isócrono.

1.4.5 Capa de transacciones

La capa de transacciones se emplea para transiciones asincrónicas, con un protocolo completo de solicitud – respuesta con confirmación para efectuar las transacciones de bus requeridas como soporte de la arquitectura CSR. La transferencia de datos entre nodos en el bus se realiza a través de tres tipos de transacciones, en la que los datos pueden tener longitud variable, que son [16]:

- **Lectura**: los datos en una dirección en particular dentro de un nodo que responde son trasferidos hacia el nodo solicitante.
- **Escritura**: los datos son transferidos desde un nodo solicitante a una dirección dentro de uno o más nodos.

- **Bloqueo:** los datos son transferidos de un nodo solicitante a un nodo que responde, son procesados en una dirección en particular dentro del nodo que responde y luego transferidos de retorno al nodo solicitante.

La capa de transacciones no provee ningún servicio para transacciones isócronas, no obstante, proporciona una vía para el control de las operaciones con datos isócronos a través de acciones de lectura y de bloqueo sobre el registro de control isócrono. Más de una transacción puede ser iniciada por un nodo solicitante antes de que la respuesta correspondiente sea retornada.

1.4.6 Servicios de la capa de transacciones

El estándar define para la capa de transacciones cuatro servicios abstractos fundamentales denominados servicios primitivos. Estos servicios son: solicitud, indicación, respuesta y confirmación. En la Figura 1.5 se muestran estos servicios.

- Solicitud: empleado por un nodo solicitante para iniciar la transacción.
- Indicación: dedicado a la acción de notificar al nodo que responde de una solicitud entrante.
- Respuesta: del nodo que responde para retornar estado y posiblemente datos al nodo solicitante.
- Confirmación: para notificar al nodo solicitante la llegada de la respuesta correspondiente desde el nodo que responde.

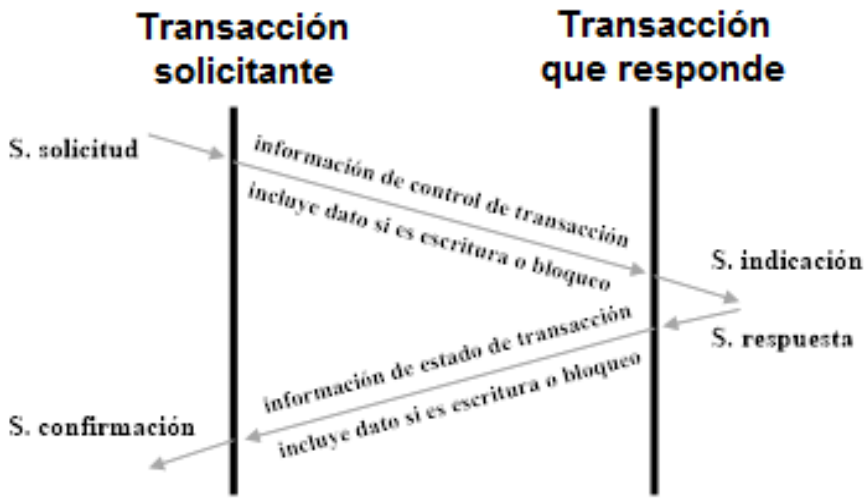

Figura 1.5. Relación entre los servicios primitivos de la capa transacciones.

1.4.7 Formato de los paquetes de la capa de transacción

La cabecera de los paquetes asincrónicos tiene un formato estándar junto con un bloque opcional de datos. Los paquetes son ensamblados y desensamblados por el controlador de capa de enlace. Estructura mostrada en la Figura 1.6.

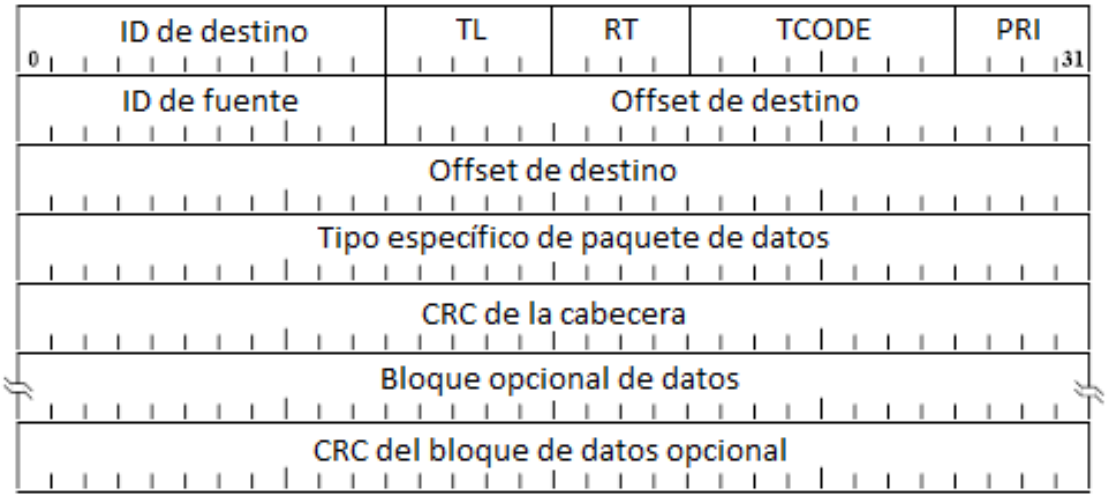

Figura1.6. Formato de los paquetes asincrónicos

ID de destino – Representa la concatenación de la dirección del bus y del nodo.

TL – Celda de transición que especifica el nodo solicitante.

RT – Código que define el mecanismo de retorno empleado.

TCODE – Código de transición que precisa el tipo de transición (solicitud de lectura, respuesta de lectura, acuse de recibo, entre otros).

PRI – Prioridad, solo se emplea en el entorno *backplane*.

ID de fuente – Identifica el bus y el nodo donde se generó el paquete.

Offset de destino – Expresa la dirección local de destino dentro del nodo.

Tipo específico de paquete de datos –Indica el tipo de paquete a ser leído o escrito.

CRC de la cabecera – Código de redundancia cíclica de la cabecera.

Bloque opcional de datos – Datos especificados en el campo: tipo de datos.

CRC del bloque de datos opcional - CRC del bloque de datos opcional.

1.4.8 Nivel de control del bus

El nivel de control de bus está también incluido en la especificación del protocolo, tal nivel proporciona las funciones básicas de control y los registros del estándar CSR [3] necesarios para controlar los nodos y/o manejar los recursos del bus. La entidad gestora del bus debe estar activa en un único nodo que ejerce la responsabilidad de control de todo el bus. El controlador del bus proporciona algunos servicios: el manejo de la alimentación eléctrica avanzada del bus [17], el mantenimiento del mapa de velocidad y del mapa topológico así como, la optimización del bus sobre la base de la información obtenida del mapa topológico.

El controlador del bus posee diferentes responsabilidades que pueden ser distribuidas en más de un nodo. Los nodos en el bus pueden asumir varios papeles: el de amo del ciclo, el de administrador de recursos isócronos y el de administrador del bus.

El amo del ciclo debe ser el nodo raíz, este inicia ciclos de 125us para la sincronización de los nodos. Si un nodo que no es amo del ciclo se convierte en nodo raíz, el bus es reseteado y el nodo que sea amo del ciclo se ve forzado a convertirse en nodo raíz.

El administrador de recursos isócronos es una entidad adicional que centraliza los servicios necesarios para gestionar y monitorear las transacciones isócronas en el bus. Este contiene los registros CSR necesarios para la asignación de ancho de banda a canales o recursos isócronos en el bus y la selección del controlador de ciclo.

El administrador del bus posee varias funciones entre las que se encuentran: la presentación de los mapas de topología y los mapas de velocidad, además, del manejo de la energía y la optimización del tráfico en el bus. El administrador del bus también se encarga de determinar si el nodo que se convierte en nodo raíz es capaz de ser amo del ciclo.

Definidas las característica principales del protocolo de comunicación IIEE-1394 se impone profundizar en los requerimientos del sistema operativo basado en FreeRTOS que será objeto de indagación en el siguiente capítulo.

CAPITULO 2 . FreeRTOS.

Desde su creación, los dispositivos digitales han utilizado un sistema de codificación de instrucciones por medio de la numeración binaria, esto se debe a que los circuitos integrados funcionan con este principio, es decir, hay corriente o no hay corriente. En sus orígenes la introducción de un programa para ser ejecutado se convertía en un increíble esfuerzo que solo podía ser llevado a cabo por muy pocos expertos. Para simplificar la labor del operador o el usuario surge la idea de crear un medio para que este último pueda operar el dispositivo con un entorno más asequible, entonces surgen los sistemas operativos. Este capítulo estudiará los aspectos fundamentales de los sistemas operativos, específicamente el sistema operativo de tiempo real FreeRTOS [18].

2.1 Sistema Operativo (SO)

El sistema operativo es el encargado de brindar al usuario una forma amigable y sencilla de operar, interpretar, codificar y emitir las ordenes al procesador central para que realice las tareas necesarias y específicas que cumplimenten una orden. Un sistema operativo se define como un conjunto de procedimientos manuales y automáticos, que permiten a un grupo de usuarios compartir un dispositivo digital complejo eficientemente.

Una de las características de los sistemas operativos es la interfaz de línea de comandos en la que se establece la interrelación entre el sistema operativo y el usuario sobre la base de la escritura de comandos, utilizando un lenguaje especial. Los sistemas con interfaz de línea de comandos se consideran más difíciles de aprender y utilizar que los de las interfaces gráficas. Sin embargo, los primeros son por lo general programables, lo que les otorga una flexibilidad que no tienen los gráficos carentes de una interfaz de programación.

2.1.1 Surgimiento de los sistemas operativos actuales

Los sistemas operativos, al igual que el *hardware* de las computadoras, han sufrido una serie de cambios revolucionarios llamados generaciones [19]. En el caso del *hardware*, las generaciones han sido enmarcadas por avances en los componentes utilizados,

pasando de válvulas a transistores, a circuitos integrados y a circuitos integrados de gran y muy alta escala de integración. Cada generación sucesiva de *hardware* ha sido acompañada de reducciones substanciales en los costos, tamaño, emisión de calor y consumo de energía, y por incrementos notables en velocidad y capacidad.

Generación cero: las máquinas de ese tiempo eran tan primitivas que los programas por lo regular manejaban un bit a la vez en columnas de interruptores mecánicos.

Primera generación: los laboratorios de investigación de la *General Motors* implementaron el primer sistema operativo en los 50's generalmente corría una tarea a la vez y suavizó la transición entre tareas.

Segunda generación: desarrolló los sistemas compartidos con multiprogramación y los principios del multiprocesamiento.

Tercera generación: introduce la familia de computadoras Sistemas/360, las que fueron diseñadas para ser compatibles con el *hardware*, para usar el SO/360, y para ofrecer mayor poder computacional a medida que iba avanzando el usuario en las series.

Cuarta generación: estado actual de la tecnología, que garantiza la ampliación del uso de redes de computadoras y del procesamiento en línea. Los usuarios obtienen acceso a computadoras alejadas geográficamente a través de varios tipos de terminales.

2.1.2 Llamadas al sistema

La interfaz entre el sistema operativo y los programas del usuario se define por medio del conjunto de "instrucciones extendidas" que el sistema operativo proporciona. Estas instrucciones extendidas se conocen como llamadas al sistema. Las llamadas al sistema se clasifican normalmente en dos categorías generales: aquellas que se relacionan con procesos y las que lo hacen con el sistema de archivo [19].

- Por procesos: un proceso es básicamente un programa en ejecución. Consta de programa ejecutable, pila del programa, contador de programa, puntero de pila y otros registros, así como información que se necesite para ejecutar el programa. El concepto de proceso en los sistemas operativos está muy relacionado al concepto de sistema de tiempo compartido. Es decir, en forma periódica, el sistema operativo decide suspender la ejecución de un proceso y dar inicio a la ejecución de otro. Cuando un proceso se suspende temporalmente, debe reiniciarse después exactamente en el mismo estado

en que se encontraba cuando se detuvo. Esto significa que toda la información relativa al proceso debe guardarse en forma explícita en algún lugar durante la suspensión. Por lo tanto, un proceso suspendido consta de su espacio de direcciones.

- Por sistema de archivo: una función importante del SO consiste en ocultar las peculiaridades de los discos y otros dispositivos de entrada/salida y presentar al programador un modelo abstracto, limpio y agradable de archivos, independientes del dispositivo. Las llamadas al sistema se necesitan con claridad para crear archivos, eliminarlos, leerlos y escribirlos. Antes de que un archivo pueda leerse o escribirse, éste debe abrirse, en cuyo instante se verifican los permisos.

2.1.3 Funciones de los Sistemas Operativos

Las funciones de los sistemas operativos son múltiples de acuerdo a [17], [18] y [20], por lo que se realiza una selección de las más importantes a consideración del autor:

- Interpreta los comandos que permiten al usuario comunicarse con la computadora.
- Coordina y manipula el hardware de la computadora, como la memoria, las impresoras, las unidades de disco, el teclado o el mouse.
- Organiza los archivos en diversos dispositivos de almacenamiento, como discos flexibles, discos duros, discos compactos o cintas magnéticas.
- Gestiona los errores de hardware y la pérdida de datos.
- Sirve de base para la creación del *software*, lo que permite que equipos de marcas distintas funcionen de manera análoga, salvando las diferencias existentes entre ambos.
- Configura el entorno para el uso del *software* y los periféricos, al establecer de forma lógica la disposición y características del equipo.

2.1.4 Categoría de los Sistemas Operativos

Sistema Operativo Monousuario.

Los sistemas monousuarios son aquellos que solo pueden atender a un usuario, gracias a las limitaciones creadas por el hardware, los programas o el tipo de aplicación que se esté ejecutando. Estos tipos de sistemas son muy simples, porque todos los dispositivos de entrada, salida y control dependen de la tarea que se está realizando, sus instrucciones son procesadas de inmediato al existir un solo usuario.

Sistema Operativo Multiusuario.

Es todo lo contrario a monousuario y en esta categoría se encuentran todos los sistemas que cumplen simultáneamente las necesidades de dos o más usuarios, que comparten mismos recursos.

Sistema Operativo Monotarea.

Los sistemas operativos monotarea son más primitivos, es decir, solo pueden manejar un proceso en cada momento y ejecutar las tareas de una en una. Estos SO tienen una única tarea en ejecución, excepto por interrupciones. Posen una pila única, son sistemas complejos, difíciles de mantener y/o extender, de controlar su precisión en las temporizaciones, además de que la programación se vuelve compleja para sistemas grandes como se muestra en la Figura 2.1.

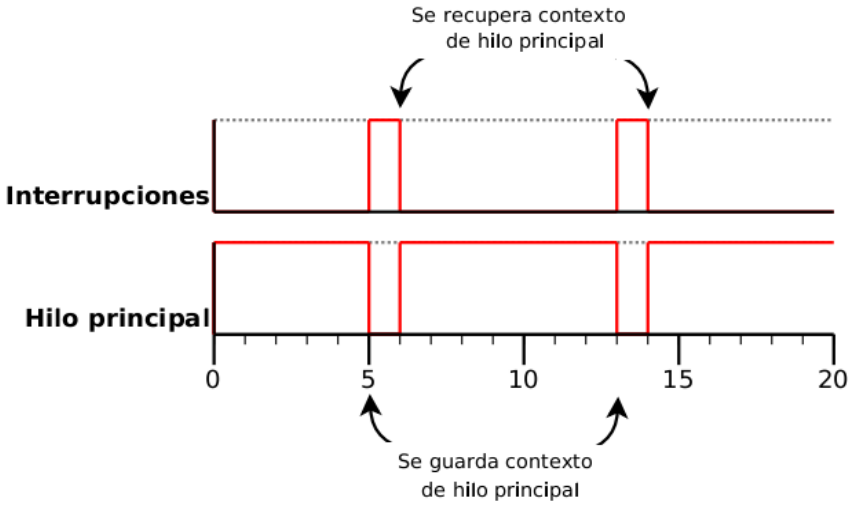

Figura 2.1. Sistema de hilo único

Sistema Operativo Multitareas

Es el modo de funcionamiento disponible en algunos sistemas operativos, mediante el cual una computadora procesa varias tareas al mismo tiempo. Existen varios tipos de multitareas. Un tipo simple de multitarea es la conmutación de contextos en el que dos o más aplicaciones se cargan al mismo tiempo, pero solo se procesa la aplicación que se encuentra en primer plano (la que ve el usuario). Otro tipo de sistema es el de multitarea de tiempo compartido, en el cual cada tarea recibe la atención del microprocesador durante una fracción de segundo. Para mantener el sistema en orden, cada tarea recibe

un nivel de prioridad y se procesa en orden secuencial. Desde esta aplicación el sentido temporal del usuario es mucho más lento que la velocidad de procesamiento de la computadora, las operaciones de multitarea en tiempo compartido parecen ser simultáneas. Las Figura 2.2 y Figura 2.3 lo ejemplifican.

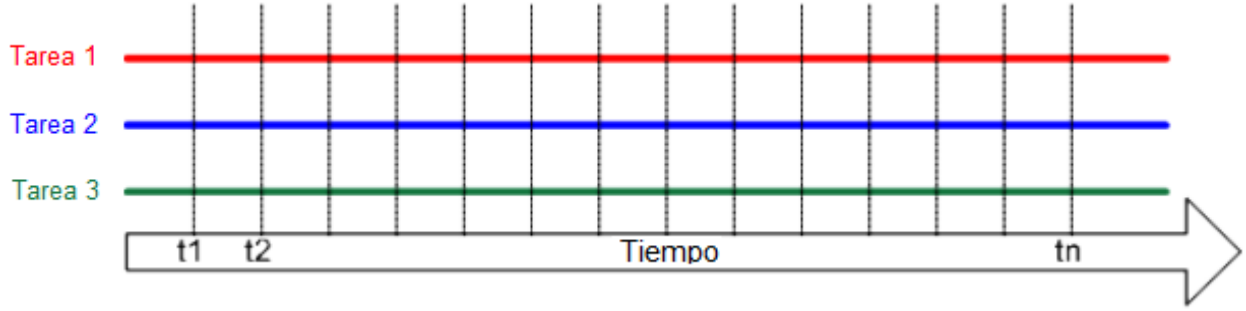

Figura 2.2 Simulación de ejecución de las tareas simultáneamente

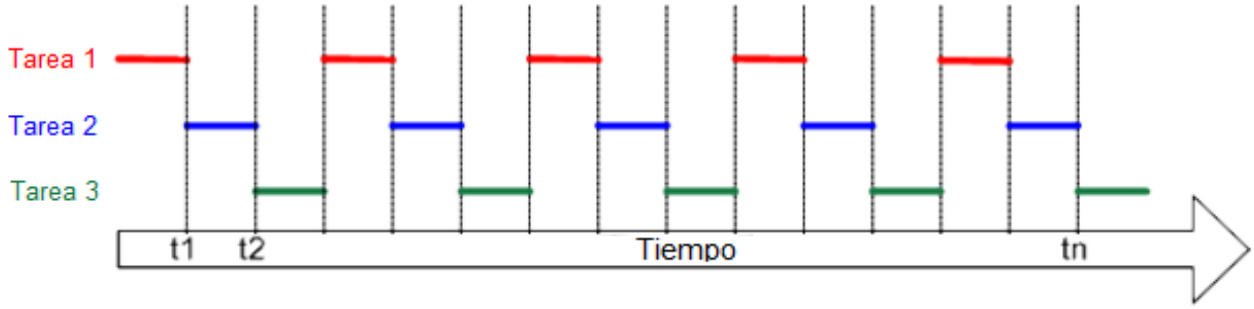

Figura 2.3. Distribución real del tiempo de ejecución entre tareas

En los sistemas multihilado se simula una CPU propia para cada tarea o hilo, cada hilo tiene una prioridad asignada, ello define el tiempo y su frecuencia de ejecución. El mantenimiento y la extensibilidad se hacen más sencillos en sistemas grandes, se pueden temporizar eventos de manera más precisa, aunque en general requieren más memoria de acceso aleatorio (RAM) que un sistema de hilo único. Los cambios de contexto no son instantáneos.

Tiempo Real.

Un sistema operativo de tiempo real procesa las instrucciones recibidas al instante, una vez procesadas muestra el resultado. Este tipo tiene relación con los sistemas operativos

monousuarios, ya que existe un solo operador y no necesita compartir el procesador entre varias solicitudes. Su característica principal es dar respuestas rápidas [20].

2.1.5 Sistema operativo en tiempo real (RTOS).

Existen muchas técnicas establecidas al elaborar un *software* para sistemas embebidos sin el empleo de un sistema operativo en tiempo real. Estas técnicas suelen ser la opción adecuada en sistemas sencillos, pero en los casos más complejos esta es la opción acertada, aunque la complejidad de los problemas es subjetiva [21]. Una de sus características fundamentales es el uso del núcleo o *scheduler* a través del cual se realiza la priorización de tareas, y su vez ofrece otros beneficios menos obvios tales como:

- Provee resúmenes de la información en el tiempo: es responsable de ejecuciones cronometradas y provee información temporal a la interfaz de programación de aplicaciones (API). Esto permite que la estructura de la aplicación sea relativamente simple y que el código en sea pequeño.
- Soporte y extensión: resume los detalles de tiempo del que resultan menos dependencias entre los módulos y deja el *software* evolucionar en forma controlada y previsible. También, el núcleo es responsable del tiempo, por lo que la función aplicativa es menos susceptible a los cambios en el *hardware*.
- Módulos: las tareas son módulos independientes donde cada uno tiene un propósito bien definido.
- Fácil experimentación: al ser las tareas módulos independientes bien definidos y con interfaces limpias, pueden ser puestos a prueba de forma independiente.
- Reutilización de código: la existencia de módulos resulta en código que puede ser reutilizado con menor esfuerzo.
- Mejorada eficiencia: usar un núcleo permite al *software* controlar los eventos, no se desperdicia tiempo de procesamiento revisando acontecimientos que no han ocurrido y el código se ejecuta sólo cuando hay alguna acción a realizarse.
- Tiempo desocupado: la tarea inactiva es creada automáticamente cuando el núcleo se inicia. Se ejecuta cada vez que no hay tareas aplicativas en espera. La tarea

inactiva puede usarse para medir capacidad, realizar comprobaciones de fondo, o simplemente para colocar el procesador en modo de bajo consumo.

- Manejo flexible de las interrupciones: la gestión de interrupciones permite retrasar el procesamiento de una terea determinada.
- Mezcla de los procedimientos: los patrones simples del diseño pueden lograr una mezcla del periodo del procesamiento de una aplicación, ellos pueden ser continuos o conducidos por acontecimientos. Además, los requisitos fuertes/suaves y de tiempo real son usados para seleccionar la tarea adecuada y la prioridad en las interrupciones.
- Fácil control sobre los periféricos: el núcleo puede usarse para controlar el acceso a los periféricos.

¿Cuándo usar un RTOS?

Está pregunta está directamente relacionada con el tipo de aplicación que se desee realizar y la capacidad que tenga el microprocesador [22]. El primero, depende de lo que se desee crear y los distintos periféricos. Su capacidad de trabajo a velocidades distintas implica un mayor control en los tiempos de ejecución de cada periférico, lo que le confiere complejidad al código. Por otro lado al aumentar la capacidad del procesador y crecer el poder de procesamiento y los recursos que este posea, se llega un punto en el que conviene hacer una solución del tipo RTOS.

2.2 FreeRTOS

FreeRTOS es un sistema operativo de tiempo real desarrollado por Richard Barry de *Real Time Engineers Ltd* en el año 2002. Fue diseñado para portarse en sistemas embebidos e implementarse con un pequeño *set* de funciones: manejo básico de las tareas, así como, el control de memoria. Posee buena sincronización de la interfaz de programación de aplicaciones y no se especifica absolutamente a nada para la comunicación de red, dispositivos de *hardware* externo, o el acceso al sistema de archivos [23].

Contiene tareas preventivas, soporta 32 tipos de arquitecturas de micro controladores, es un sistema escrito en C por sus desarrolladores y puedes ser compilado con diversos lenguajes de programación. También permite que un número ilimitado de tareas se

ejecuten al mismo tiempo sin ninguna limitación en cuanto a sus prioridades, estas pueden ser tan largas como el *hardware* utilizado permita. Implementa colas, semáforos de conteo y *mutex*. (Las funciones que se explicarán a continuación se encuentran ejemplificadas en los anexos).

2.2.1 Tareas en FreeRTOS

Una tarea es un segmento de código independiente, que tendrá su propia pila, banderas de estados y registros de datos. FreeRTOS posee un sinnúmero de tareas para ejecutar, siempre y cuando el hardware lo permita; además estas pueden ser cíclicas o no. Una tarea se define a partir de una función simple que no devuelve nada **(void)**, a la que se le pasan los parámetros que va a utilizar **(pvParameters).**

Ejemplo: **void tarea_1 (void *pvParameters).**

Ciclo de vida de las tareas

Las tareas poseen un ciclo de vida desde el momento que son creadas hasta su destrucción. En este intervalo de tiempo pueden tener dos estados posibles: ejecutadas (*running*) o no (*not running*), donde en cada momento sólo una tarea se puede ejecutar. Revelado en la Figura 2.4.

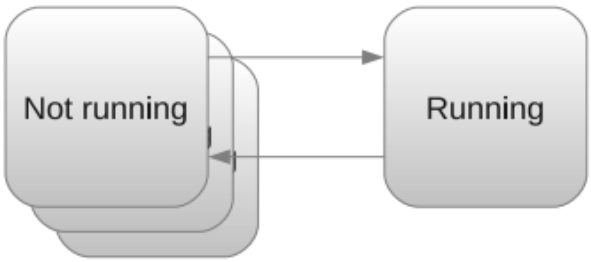

Figura 2.4. Distribución entre una terea que ejecutándose y otras que no

Sin embargo, el estado "*not running*" presenta formas distintas para las tareas por lo que puede sub-dividirse en varios estados según [20] y [21]:

Tarea bloqueada (*Blocked*)

Estado en el cual la tarea está esperando que suceda un "evento" para que la misma pase a ser ejecutada. Los tipos de eventos pueden ser:

- Temporal: demora típica, donde la tarea no tiene nada que hacer hasta que transcurra cierto intervalo de tiempo.
- Sincronización: existen varias formas de sincronizar las tareas mediante el uso de interrupciones, semáforos y colas.

Tarea suspendida (*Suspended*)

Es un estado donde se extrae la tarea de la cola de ejecución directamente por el programador, para lo cual se utilizan las funciones **vTaskSuspend ()** para entrar a ese modo o **vTaskResume ()** para salir del mismo.

Tarea preparada (*Ready*)

Se llega a este estado cuando la tarea está lista para ser ejecutada. Previamente el programador coloca mayor prioridad a una tarea sobre otra. Cuando se sale de un estado bloqueado/suspendido se pasa por este estado antes de ser ejecutada la tarea como se muestra en la Figura 2.5.

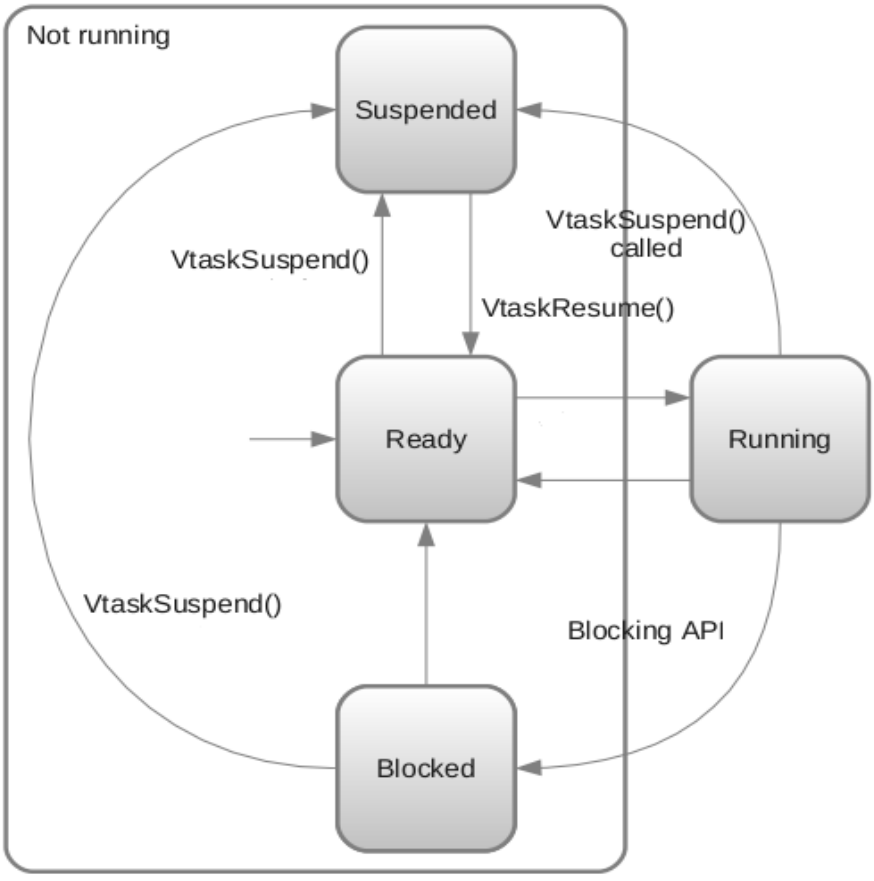

Figura 2.5. Estados de las tareas

2.2.2 Creación y eliminación de una tarea

Las tareas se definen empleando una función sencilla de **C** con la siguiente forma **xTaskCreate ().** Cualquier tarea creada nunca debe terminar antes de que se destruya. Es común que su código sea envuelto en un lazo infinito, o invocar **vTaskDestroy (NULO)** antes que se termine de ejecutar la misma. El argumento de esta función se explica a continuación:

pvTaskCode: puntero que señala la dirección donde se encuentra la implementación dela tarea.

pcName: será un *string* (cadena) que se utiliza para asignar el nombre a la tarea, es útil cuando queremos saber que tareas están detenidas.

usStackDepth: define que tan grande será la pila dedicada a la tarea. Se debe prestar atención al valor asignado, que de ser muy pequeño ocasionará problemas al ejecutar la tarea y de ser muy grande, impide la creación de la tarea por falta de memoria.

pvParameters: puntero al argumento para pasarle a la tarea.

uxPriority: define el nivel de prioridad que tendrá la tarea durante su ejecución. La mínima prioridad será “0” y la máxima quedará definida por el programador en el archivo Config.h.

pxCreatedTask: sirve para pasarle la dirección de un controlador en el manejo de la tarea.

La función devuelve **pdTrue** si la tarea se creó correctamente o **errCOULD_NOT_ALLOCATE_REQUIERED_MEMORY** si hubo un error debido a la falta de memoria RAM.

Para destruir una terea simplemente se usa la función **xTaskDestroy ()** y se toma como argumento el valor **pxCreatedTask** controlador de la tarea.

2.2.3 El núcleo y la prioridad de las tareas

La política empleada por el núcleo es un algoritmo usado para decidir que tarea ejecutar en cada momento. En los sistemas que no son en tiempo real la política es asignar el mismo tiempo de ejecución a cada tarea, mientras, la utilizada en tiempo real consiste en determinar la prioridad de las tareas por el programador cuando las crea. Este es el

parámetro más importante que tiene en cuenta el núcleo para decidir (en cada pulso de reloj (*tick*)) cual es la tarea que se va a ejecutar [24]. La Figura 2.6 muestra el papel del núcleo en el control y ejecución de las tareas.

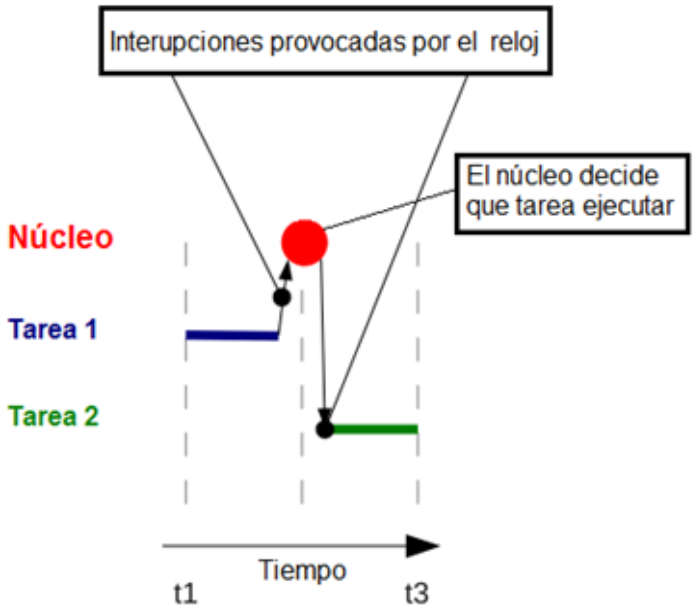

Figura 2.6. El núcleo controla la ejecución de las tareas

Cuando se crean tareas con la misma prioridad el núcleo las trata como iguales, asignándole el mismo tiempo de ejecución a cada una de ellas, conmutando ambas tareas con cada ciclo de reloj. Además siempre debe haber una tarea en ejecución para lo que se emplea la tarea holgazana o inactiva. Es importante asegurar que la tarea de mayor prioridad no se apropie de todo el tiempo de ejecución, por lo que debe otorgarle algún tiempo a la tarea holgazana, puesto que esta realiza importantes funciones con el acceso a memoria, la eliminación de tareas y la conmutación de dispositivos dentro del modo de reposo.

2.2.4 Funciones disponibles para el trabajo con las tareas

Para controlar las tareas existen una serie de funciones que permiten hacer más fácil la programación de las mismas:

vTaskDelay- Ofrece demorar las tareas.

vTaskDelayUntil- Ofrece la opción de demorar las tareas un tiempo específico.

uxTaskPriorityGet- Obtiene la prioridad de las tareas.

vTaskPrioritySet- Asigna una nueva prioridad a las tareas.

vTaskSuspend- Suspende la ejecución de una tarea.

vTaskResume- Cambia una tarea suspendida a lista para su ejecución.

vTaskStartScheduler- Función que enciende el núcleo.

vTaskEndScheduler- Función que apaga el núcleo.

xTaskResumeAll- Función que cambia todas las tareas a listas para su ejecución.

vTaskSuspendAll- Función que suspende todas las tareas que se encontraban listas para su ejecución.

xTaskGetTickCount- Función que obtiene el valor del contador de *ticks.*

uxTaskGetNumberOfTasks- Función que obtiene el número de tareas que está manejando el núcleo.

vTaskList- Muestra en una lista el estado actual de las tareas representándolas como bloqueada *blocked* ('B'), lista *ready* ('R'), eliminada *deleted* ('D') o suspendida *suspended* ('S').

vTaskStartTrace- Inicia un mapeo de la actividad del núcleo y que tareas son ejecutadas y cuando.

ulTaskEndTrace- Detiene el mapeo del núcleo.

2.3 Colas

Las colas son una forma de comunicar o sincronizar dos o más tareas en las cuales es necesaria cierta transferencia de datos, son objetos que no pertenecen a ninguna tarea. Esto posibilita que cualquier tarea pueda escribir o leer en una cola sin importar su prioridad. Una cola puede almacenar un conjunto de datos (bytes, words, entre otros) según el tamaño que se le asigne durante su creación. Se las suele usar como *buffers* FIFO y al escribir un dato en la cola se almacena en la misma, mientras que al leerlo se obtiene una copia del dato y se remueve de la cola.

Creación de una cola

Para crear una cola se usa la función **xQueueCreate** la cual nos devolverá el controlador de la cola. Los parámetros de la cola son:

uxQueueLength: que indica el tamaño de la cola, es decir, cuantos elementos contiene.

uxDatosize: indica el tamaño de cada elemento de la cola **(**int, double, constant, entre otros).

Si se pudo crear la cola, la función devolverá un puntero al controlador de la cola distinto de "*null*", de lo contrario hubo algún inconveniente a la hora de crear la cola. Esta función debe ser asignada a una variable de tipo **xQueueHandle**. Otra función a tener en cuenta es **vQueueDelete** la cual elimina una cola liberando el espacio de memoria que esta ocupaba y tiene como parámetro **xQueue** que es el controlador de la cola.

2.3.1 Lectura de una cola.

Bloqueo por lectura:

Cuando una tarea se dispone a leer una cola, como opción puede bloquearse durante un determinado tiempo hasta que haya un dato a leer. Entonces, mientras la cola esté vacía, la tarea que se dispone a leer se encuentra bloqueada hasta que otra tarea (o incluso una interrupción) escriba un dato en la cola; a partir de ese momento la tarea de lectura pasa al estado preparado.

En caso de sobrepasar el tiempo de espera debido a la ausencia de datos en la cola, la tarea pasa al estado preparado. Si hubiera dos tareas de lectura, siempre la de mayor prioridad pasará a este estado, para obtener el dato almacenado en la cola. En cambio, si ambas tareas tienen la misma prioridad, la tarea que mayor tiempo se encuentre en estado bloqueado será la que tenga mayor preferencia. La Figura 2.7muestra los procesos de lectura y escritura en una cola.

La lectura en una cola se realiza a través de la función **xQueueReceive ()** que sirve para leer y eliminar el dato de la lista, la cual tiene el siguiente argumento:

xQueue: el controlador de la cola que será leída.

pvBuffer: es un puntero a donde se almacenará el dato leído.

xTicksToWait: almacena la máxima cantidad de pulsos de reloj, en los cuales la tarea que desea leer la cola permanecerá en estado bloqueado, hasta que haya un dato disponible. En caso de valer “0”; si la cola está vacía, automáticamente la tarea entrará en el estado preparado (*ready*).

Esta función **(xQueueReceive ())** devolverá **pdPASS** cuando se lea un dato válido de la cola o **errQUEUE_EMPTY** que significa que la cola estaba vacía.

Otro aspecto interesante es que para obtener la cantidad de datos disponibles en la cola se emplea la función **uxQueueMessagesWaiting (),** la que devolverá la cantidad de datos disponibles para leer que hay en la cola, siendo su argumento **xQueue** como controlador de la cola. Nunca se deben emplear estas funciones en una rutina de atención a interrupción, debido a que en estas no se permite cambiar el contexto de las tareas, por lo que para esos casos se deberá utiliza r**xQueueReceiveFromISR ()** y **uxQueueMessagesWaitingFromISR ()**, como funciones optimizadas que permiten modificar contextos.

2.3.2 Escritura en una cola

Bloqueo por escritura:

Al igual que el bloqueo por lectura una tarea, opcionalmente, puede bloquearse durante un determinado tiempo a la espera de que en la cola, exista lugar para un nuevo dato. Si hubiera dos o más tareas que quisieran escribir en la cola, el funcionamiento de las prioridades es similar al explicado en el bloqueo por lectura. La Figura 2.7muestra los procesos de lectura y escritura en una cola.

La escritura en una cola se realiza a través de las funciones **xQueueSend ()** y **xQueueSendToBack ()** que sirven para escribir al final de la cola y sus parámetros son:

xQueue: controlador de la cola en la que se escribirá.

pvItemToQueue: puntero del dato original desde el cual se copiara a la cola.

xTicksToWait: cantidad máxima de pulsos de reloj, en los cuales la tarea que desea escribir en la cola permanecerá en estado bloqueado, hasta que haya lugar disponible. En caso de valer “0”; si la cola está llena, automáticamente la tarea entrará en estado preparado (ready).

Esta función devolverá **pdPASS** si se escribió un dato válido de la cola o **errQUEUE_FULL** si estaba llena.

Nunca se deben emplear estas funciones en una rutina de atención a interrupción, debido a que en estas no se permite cambiar el contexto de las tareas, por lo que para esos casos se deberá utilizar **xQueueSendToBackFromISR (),** como función optimizada que permite modificar contextos.

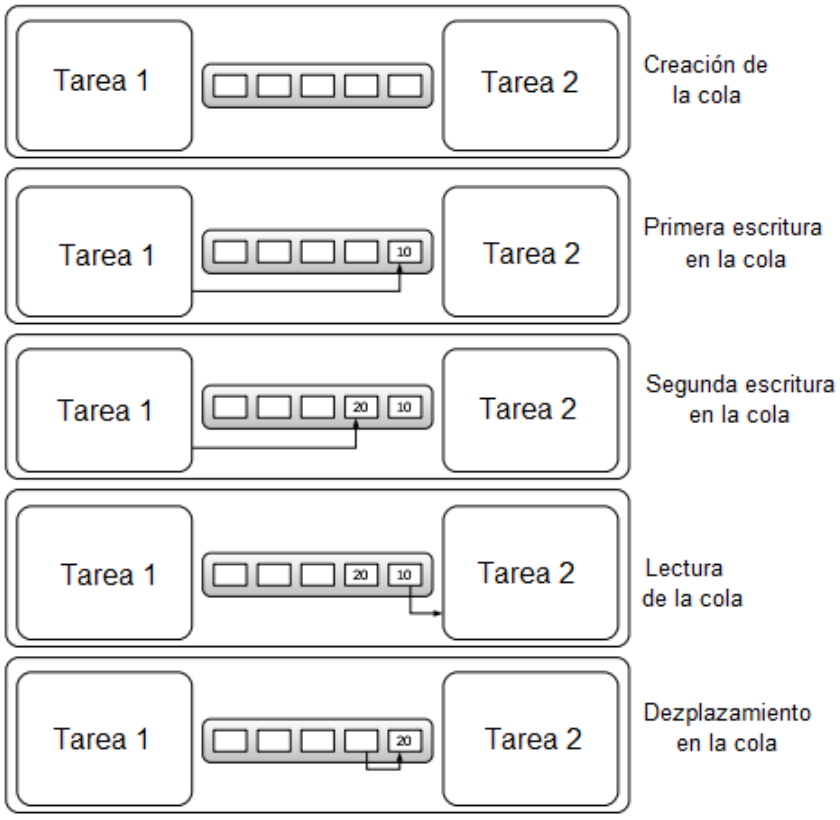

Figura 2.7. Operaciones de una cola entre dos tareas

2.4 Semáforos binarios

Un semáforo binario es la forma más sencilla de sincronizar las tareas ya que consiste en revisar los recursos entrantes. Este puede ser visto como una cola que posee un solo elemento y se puede utilizar para desbloquear una tarea cuando ocurre una interrupción.

Para crear un semáforo binario se usa la función que **vSemaphoreCreateBinary** no devuelve ningún valor, donde **xSemaphore** es el controlador del semáforo creado y debe

ser del tipo **xSemaphoreHandle**. Las operaciones con semáforos son mostradas en la Figura 2.8.

2.4.1 Tomar y entregar un semáforo

La operación de tomar un semáforo se puede comparar con la operación de recepción en una cola ya que la tarea que toma el semáforo debe esperar su habilitación. La función empleada para tomar un semáforo es **xSemaphoreTake**, donde su argumento será:

xSemaphore: controlador del semáforo a tomar.

xTicksToWait: almacena la máxima cantidad de pulsos de reloj en la cual la tarea permanecerá bloqueada. Si este valor es "0", desbloqueará de inmediato la tarea. Si es**portMAX_DELAY**, permanecerá bloqueada indefinidamente hasta que se produzca un evento y le permita tomar el semáforo. Esta función devolverá **pdPASS** si la tarea pudo tomar el semáforo o **pdFALSE** si no lo logra.

La operación de entregar un semáforo es similar a la operación de escritura en una cola, donde se usa la función **xSemaphoreGive** para tomarlo. Su argumento es:

xSemaphore: controlador del semáforo a entregar.

Devolverá **pdPASS** si se pudo entregar el semáforo o **pdFAIL** si el semáforo ya estaba disponible y no pudo ser entregado nuevamente.

Esta función no debe ser empleada en una rutina de atención a interrupción, debido a que no puede cambiar el contexto de las tareas, por lo que para esos casos se deberá utilizar **xSemaphoreGiveFromISR** y se le pasa el controlador del semáforo. En caso de haber dos o más tareas bloqueadas por el mismo semáforo se utiliza un segundo parámetro que permite entregar este a la tarea de mayor prioridad cuyo nombre es **pxHigherPriorityTaskWoken**. Los semáforos se crean y se toman con las mismas funciones.

2.4.2 Semáforos con contadores

El gran problema que tienen los semáforos binarios, es que solo permiten manejar un evento a la vez, por lo que un semáforo binario solo sirve cuando la frecuencia de los eventos es baja [25]. Sin embargo, si durante la ejecución de la tarea sucede más de un evento, inevitablemente con un semáforo binario se estarían perdiendo uno o más

eventos. Para remediar ese inconveniente, se utiliza un semáforo con contador, que permite contar la cantidad de eventos que se produjeron durante la ejecución. Estos semáforos se entregan y se toman usando las mismas funciones que los semáforos binarios, solo cambia la creación del mismo. Para crear un semáforo con contador se usa la función **xSemaphoreCreateCounting** donde su argumento es:

uxMaxCount: representa la máxima cantidad de eventos que contará el semáforo.

uxInitialCount: representa la cuenta de eventos inicial que tendrá el semáforo.

Esta función que devolverá **xSemaphore** que es el controlador del semáforo creado.

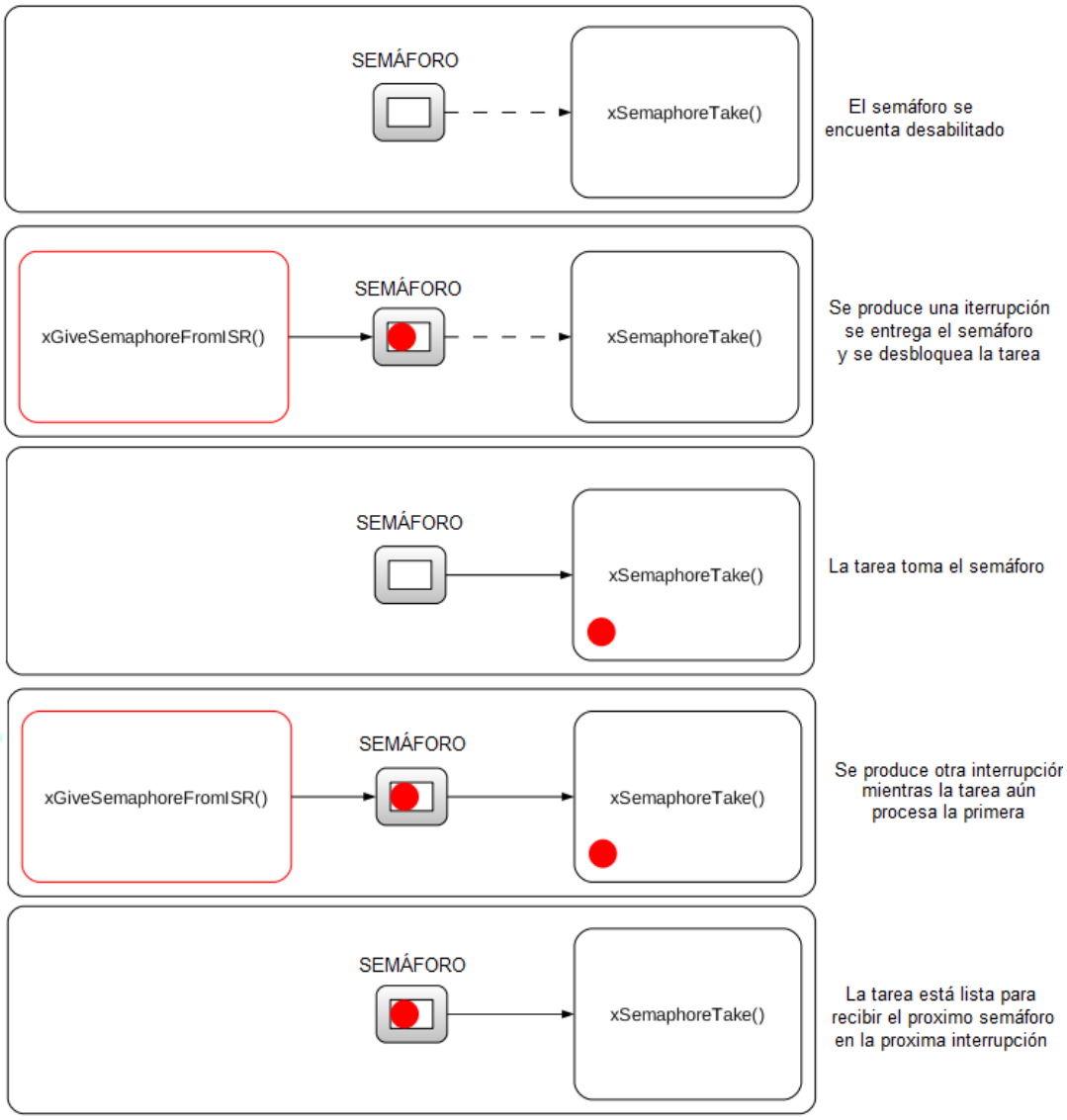

Figura 2.8. Operaciones con semáforos

2.5 Algoritmos de exclusión mutua (Mutex)

Los mutex se emplean de manera similar a los semáforos, para gestionar los recursos. Fundamentalmente evitan la exclusión mutua, con la diferencia de que una cuando la tarea tome el mutex debe devolverlo al finalizar la operación. En la figura 2.9 se explica este modo de trabajo.

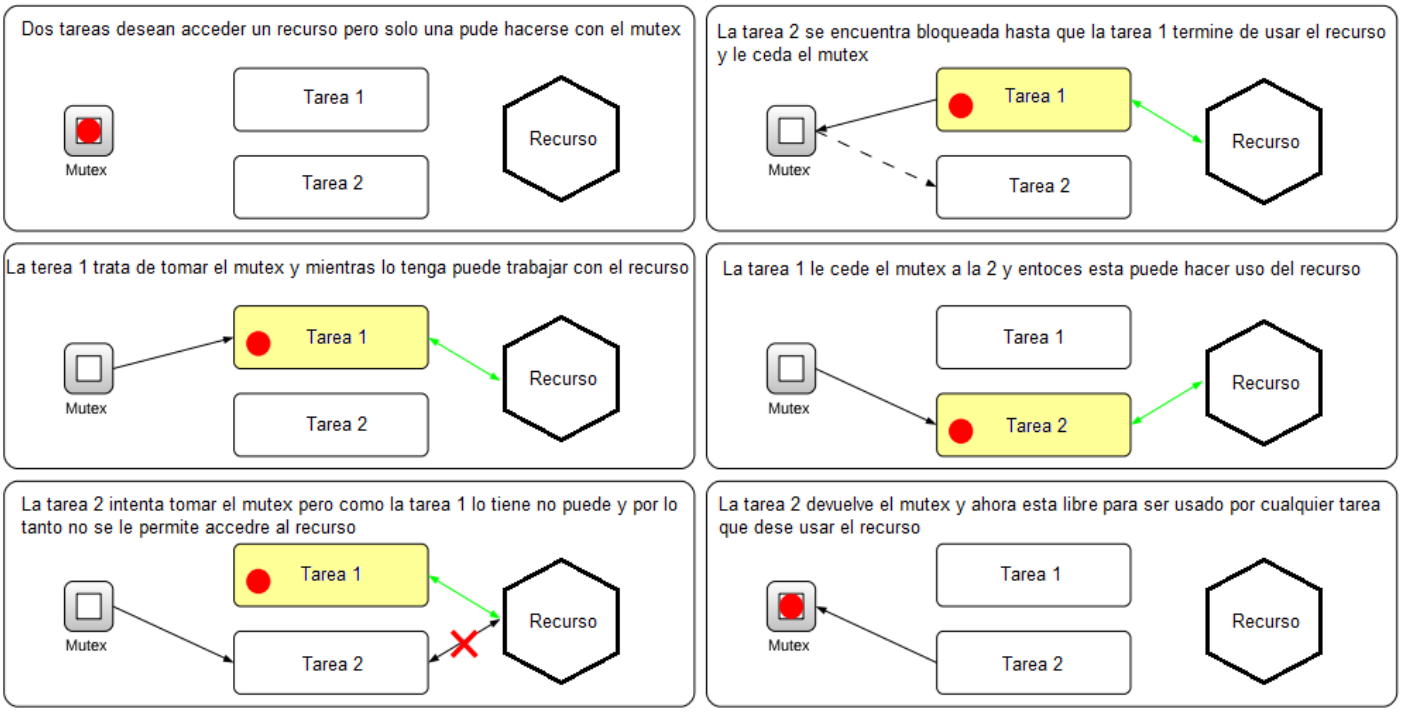

Figura 2.9. Modo de empleo de los mutex

2.6 Interrupciones

Las interrupciones son mecanismos implementados y controlados por hardware, donde el *software* solo aporta métodos para manejarlas o para llamar instrucciones de hardware. La cantidad de interrupciones que puede manejar un programa depende de la capacidad que tenga el procesador y con ello la prioridad que tendrán las mismas, siendo las más prioritarias las mayores. Las funciones presentadas anteriormente no pueden ser usadas para dar atención a una interrupción pero FreeRTOS aporta funciones espaciales como **xSemaphoreGiveFromISR ()** en vez de **xSemaphoreGive ().**

El control de las interrupciones se puede configurar en el archivo Config.h modificando los siguientes campos:

configKERNEL_INTERRUPT_PRIORITY que coloca el nivel de interrupción prioritario para las interrupciones.

configMAX_SYSCALL_INTERRUPT_PRIORITY que define el máximo nivel habilitado para una interfaz en FreeRTOS.

Como la gestión de interrupciones son porciones de códigos que ejecuta el controlador, FreeRTOS no las puede gestionar directamente y esto pude traer demoras. Por este motivo, para aumentar la velocidad de ejecución se configura dentro de la rutina de gestión un vínculo a una tarea para darle tratamiento, reduciendo el tiempo de ejecución de la misma, tratándola como si fuese una tarea más, como se muestra en la Figura 2.10.

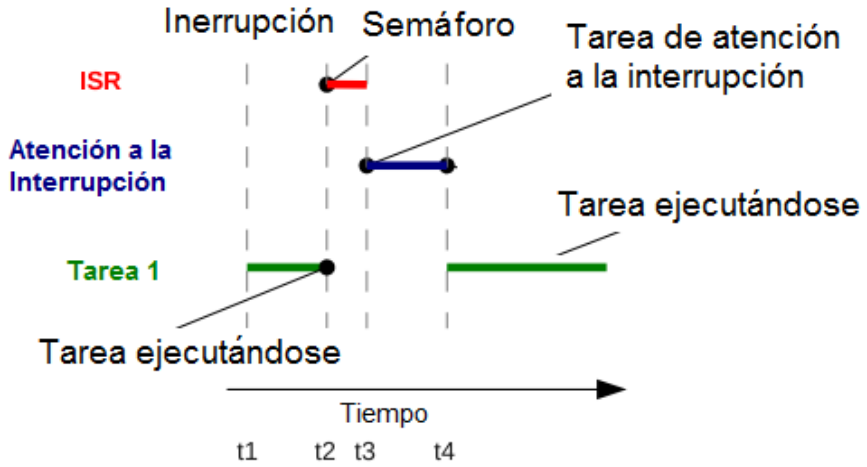

Figura 2.10. Atención a la interrupción mediante una tarea con semáforo binario.

Cuando se están tratando las interrupciones es importante que algunos parámetros dentro de las tareas no sean modificados, por lo que el sistema operativo debe detener las tareas a partir de dos métodos: el primero es entrar a la sección crítica mediante las funciones **ENTER_CRITICAL ()** para iniciarla y **EXIT_CRITICAL ()** para finalizarla. El segundo es detener el núcleo mediante la función **vTaskSuspendAll ()**. Estos dos métodos aseguran que el contexto de las tareas no cambie y que las mismas no modifiquen su estado (“*Running*” o “*No Running*”).

2.7 Gestión de la memoria

Uno de los principales problemas que presentan los sistemas embebidos es la asignación de memoria para las tareas, colas y semáforos; donde la asignación de memoria para los eventos prioritarios y la memoria libre trae como resultado la fragmentación de la misma

[26]. Todas las asignaciones y los retiros de memoria se hacen a través de dos funciones: **void *pvPortMalloc (size_t xWantedSize)** para asignar espacios de memoria en la que se utiliza **xWantedSize** como tamaño de memoria a asignar y en la que retorna un puntero a la dirección de la memoria asignada. Mientras que la función **void pvPortFree (void *pv)** se utiliza para retirar memoria asignada donde **pv** es un puntero a la memoria que se va a retirar.

Estas dos funciones se pueden usar en cualquier momento, pero en algunos casos, pueden presentar problemas como: en ocasiones no se encuentran disponibles en sistemas embebidos o no son funciones determinísticas (el tiempo dedicado a la ejecución de cada función varia de llamada a llamada). FreeRTOS brinda tres métodos para resolver el problema de asignación de la memoria, cada uno adaptado para diferentes situaciones

Memoria única para todos: caso más simple en el cual el programador asigna espacios de memoria fijos para cada evento al iniciar el programa y no debe hacer más cambios en la memoria, ya que los eventos no modifican su tamaño, ni hay que agregar nuevos eventos. Esta configuración simple de la memoria se puede hacer modificando en el archivo Config.h el parámetro **configTOTAL_HEAP_SIZE** y dividirlo en secciones según requiera cada tarea.

Este método es recomendable si la aplicación no elimina tareas o colas, (no se realizan llamadas a **vTaskDelete ()** o a **vQueueDelete ()**) además, es determinista pues siempre tarda el mismo tiempo en devolver un bloque, razón por la cual, este método es adecuado para una gran cantidad de RTOS que cumplan con la única condición de que todas las tareas y las colas se creen antes de que el núcleo sea iniciado. Esta variante se muestra en la Figura 2.11.

Memoria constante y numerada: caso utilizado para asignar y retirar la memoria de forma dinámica. Cuando el ciclo de vida de una tarea no es constante esta sencillamente termina. La memoria que ocupaba queda libre y puede ser asignada a otra tarea. Este método es más efectivo que el anterior en tanto consume mucho menos RAM.

Es recomendable para aplicaciones que reserven bloques de memoria RAM del mismo tamaño, es decir, tareas, pila y colas de igual tamaño. No es determinista por lo que resulta adecuado para RTOS pequeños que requieran crear dinámicamente las tareas. Se referencia en la Figura 2.12.

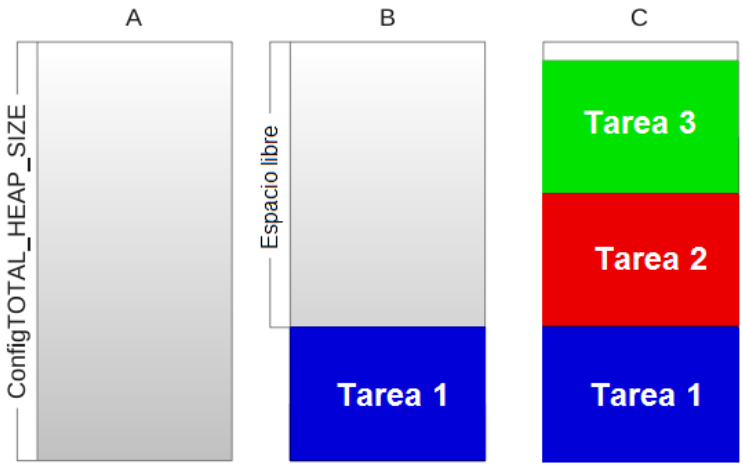

Figura 2.11. Asignación de memoria por el método de memoria única para todos

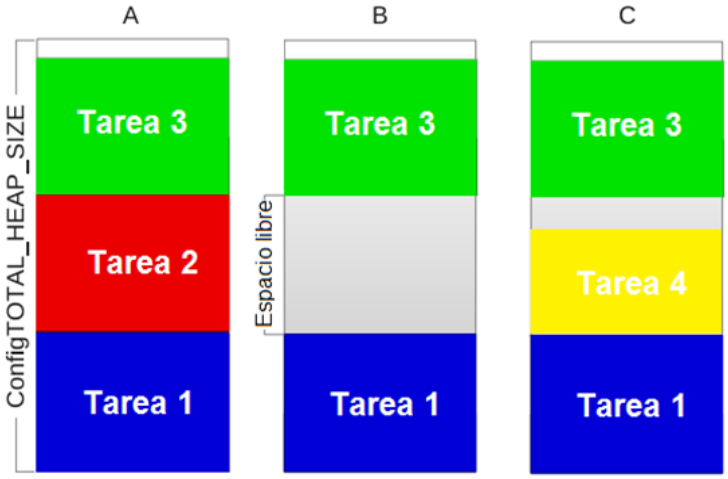

Figura 2.12. Asignación de memoria de forma dinámica

Asignación y retiro de la memoria libre: se caracteriza por el uso de forma segura de las funciones **malloc ()** y **free ()** explicadas anteriormente, pero tiene como inconvenientes que el código a compilar es amplio y la ejecución suele ser poco determinista, además, para asignar o retirar la memoria de forma segura es necesario detener el núcleo. Este método aumentará considerablemente el tamaño del código del núcleo, aunque es menos restrictivo.

CAPITULO 3 . FREERTOS EN NUEVA PLATAFORMA. *SOFTWARE* DE COMUNICACIÓN.

Simular el *software* requiere de una plataforma para probarlo, ello precisa de la selección de una tarjeta de desarrollo Digilent® Nexys2 [27], que incorpora una FPGA Spartan-3E de Xilinx, lo que permite evaluar el desempeño del programa. Realizar la simulación implica estar seguro de que el compilador reconoce el lenguaje de programación empleado en FreeRTOS y que la plataforma soporta el sistema operativo en tiempo real, como base a la elaboración de la propuesta de *software* que se realiza. Razones que se abordan en el capítulo.

3.1 *System-on-chip* (SoC)

System-on-a-Chip o SoC (*system-on-chip*), describe el proceso de usar tecnologías de fabricación que integran todos, o gran parte de los módulos componentes de una computadora o cualquier otro sistema informático-electrónico en un único circuito integrado o chip. El diseño de estos sistemas puede estar basado en circuitos de señal digital, analógica, o incluso mixta. Un ámbito común de aplicación de la tecnología SoC son los sistemas embebidos.

Además de los SoC existen otras variantes para la creación de sistemas complejos en un solo circuito integrado: primero está *System-in-Package* (SiP), el cual comprende un número determinado de chips ensamblados en uno solo y el segundo corresponde a *Package-on-Package* (PoP), que es el apilado, de diferentes placas de circuitos al ensamblarse el sistema. Se estima que la fabricación en gran volumen de SoC será más rentable que la de SiP o PoP, debido a su mayor rendimiento de fabricación unitario para un SoC y su montaje y empaquetado más sencillos.

3.2 Tarjeta de desarrollo para la evaluación

El sistema a diseñar se evaluará sobre la tarjeta de desarrollo Digilent® Nexys2 [27], considerada como un SoC que incorpora una FPGA Spartan-3E de Xilinx, memorias *Flash* y PSRAM de 16 MB cada una, periféricos PS/2, VGA, botones, LED y lámparas 7

segmentos. Incorpora dos memorias SDRAM y FLASH, una interfaz USB2 de alta velocidad, un puerto VGA, y otro RS232 con dispositivos de entrada/salida. Esta plataforma es ideal para sistemas digitales de todo tipo, entre ellos, los sistemas embebidos basados en Xilinx's MicroBlaze [28]. Como se observa en la Figura 3.1.

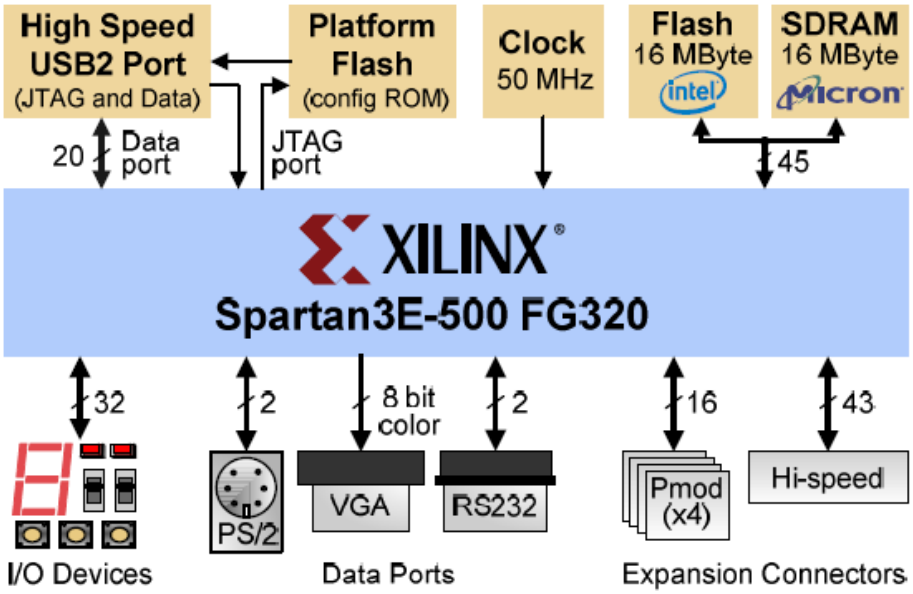

Figura 3.1. Diagrama en bloque de la tarjeta de desarrollo Nexys2

Componentes de la tarjeta de desarrollo Nexys2

- Núcleo Xilinx Spartan 3E FPGA500K-gate.
- Puerto USB2 y puerto serie para la transferencia de información.
- Varias fuentes de alimentación: por USB, por baterías o por transformador externo.
- 16MB de PSDRAM Micron y16MB de ROM Intel StrataFlash.
- Plataforma Flash Xilinx para configuraciones FPGA no volátiles.
- Oscilador de 50MHz y conector para colocar un segundo oscilador.
- 8 LEDs, 4 displays 7 segmentos, 4 botones de presión y 8 interruptores.
- Puerto PS/2 para la conexión de mouse o teclado.
- Puerto VGA para la conexión de un monitor.
- Puertos de expansión (un puerto alta velocidad de 50 terminales Hirose FX2 y cuatro puertos de6 terminales)
- Eficiencia en la conmutación de los modos de alimentación, resulta práctico para aplicaciones portátiles.

3.3 Módulos de propiedad intelectual

Los módulos de propiedad intelectual o núcleos IP (*IP Cores*) son componentes desarrollados en lenguaje de descripción de hardware (HDL), para portarse generalmente sobre hardware libre, permite a los diseñadores reducir los tiempos de desarrollo de las distintas aplicaciones. Se basan en la generación de componentes reutilizables que puedan inter-operar bajo ciertas circunstancias, y unirse para formar un sistema digital complejo, donde pueden ser considerados bloques básicos de construcción, para su posterior implementación del sistema sobre un ASIC (Circuito Integrado para Aplicaciones Específicas) o FPGA.

Esta metodología fomenta principalmente el abaratamiento de costes y la reutilización. Dentro de su desarrollo se potencia la interoperabilidad entre módulos IP para hacerlos compatibles entre sí. Estos núcleos van desde funciones simples como contadores, coprocesadores, controladores de comunicación y memoria, hasta sistemas complejos como microcontroladores, procesadores (CPU), módulos, entre otros. Algunas de las plataformas libres que poseen módulos IP son OpenRISC [29] y MIPS Lite [30] que pertenecen a OpenCores [31] y la plataforma LEON [32] mantenida por la corporación GaislerResearch. En el trabajo se centra la atención en un componente de la plataforma Plasma-Wishbone el procesador MIPS Lite [25] base sobre la cual se simulará el sistema operativo.

3.4 Plasma-Wishbone

Plasma-Wishbone es un SoC presentado por Ryder [33] resulta de la unión del SoC libre Plasma [34], desarrollado por Steve Rhoads y la arquitectura de interconexión para módulos IP Wishbone [35] propuesta por Wade Peterson, ambos disponibles en el sitio web de OpenCores.

Plasma como sistema demanda menos recursos del área de la tecnología de implementación que otras plataformas factor que abarata los costos. En su núcleo se encuentra el procesador MIPS Lite de 32 bits que implementa la mayoría de las instrucciones del *set* de MIPS I [36].

Wishbone es una propuesta de interfaz de uso general que define un estándar de intercambio de datos entre módulos IP, no especifica las características internas y funciones del módulo que la implementa. Es similar a un bus de microcomputadora,

proporciona una solución de integración flexible y fácilmente diseñada a la medida para una aplicación específica. Ofrece variedad de ciclos de bus y tamaños variables de palabra de datos y permite el empleo de módulos IP de diferentes fuentes.

Plasma y Wishbone no son compatibles directamente en tanto, de Plasma se tomaron algunos elementos, siendo el más importante el procesador, al cual se le realizan adecuaciones para hacerlos compatibles con Wishbone y colocarlos en una sola arquitectura como se muestra en la Figura 3.2.

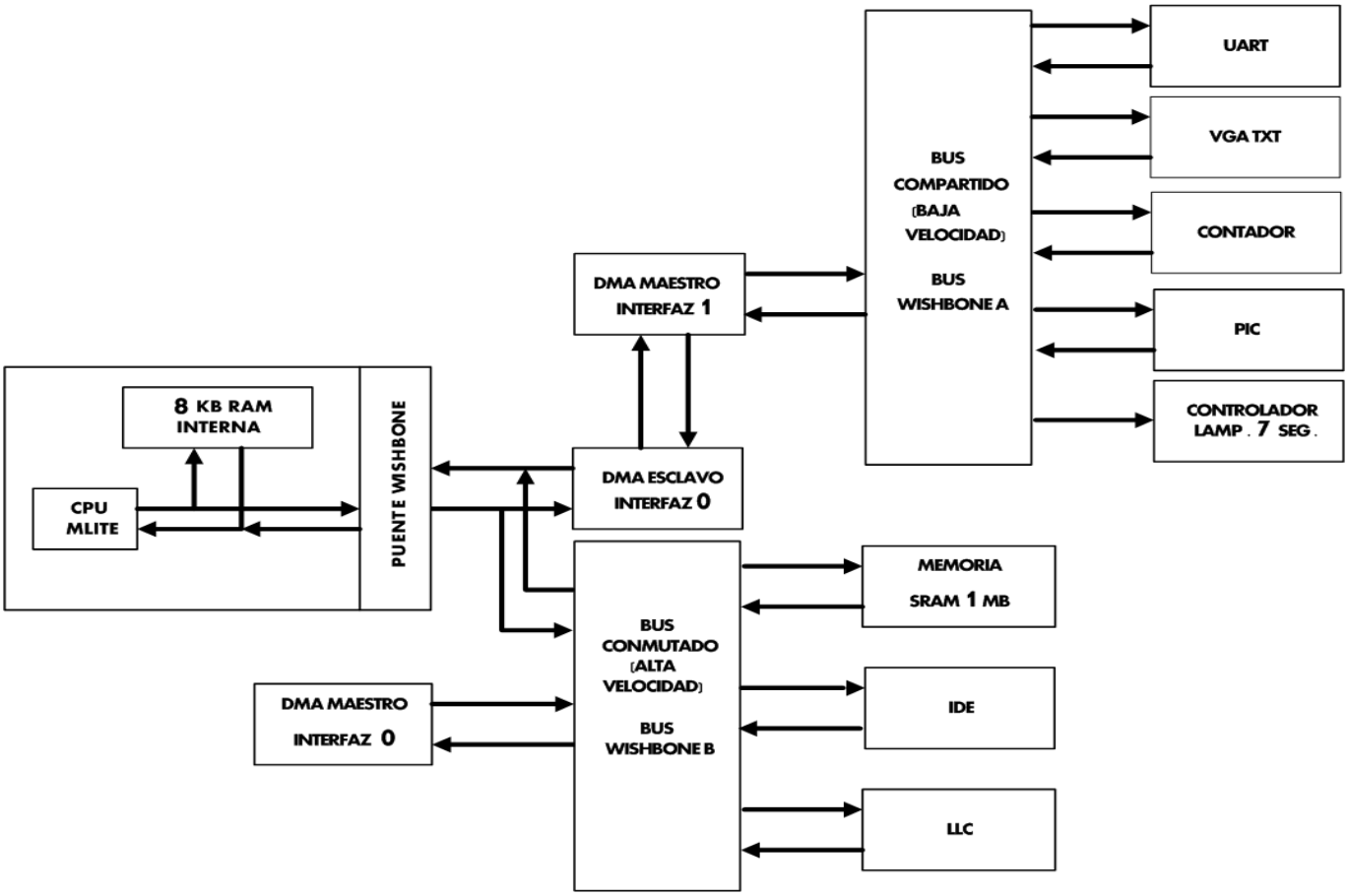

Figura 3.2. Arquitectura del SoC Plasma-Wishbone.

- De esta arquitectura nos interesan particularmente tres componentes: el módulo contador de pulsos de reloj, el módulo controlador programable de interrupciones (PIC) y el procesador MIPS Lite.

3.4.1 Módulo Contador de pulsos de reloj

Este dispositivo es de propósito general y de solo lectura. Es un contador de pulsos de reloj de 32 bits (se desborda pasados 4294967296ciclos de reloj).Incluye una interfaz Wishbone que permite obtener el valor del contador y dos salidas conectadas al PIC. Estas últimas para dar soporte al sistema operativo de tiempo real. De tal forma, las interrupciones generadas por las dos líneas del contador informan al sistema que es momento de cambiar de tareas, según el nivel de prioridad que estas ocupen en el sistema operativo.

3.4.2 Módulo Controlador Programable de Interrupciones (PIC)

- El (PIC) resulta un dispositivo esencial para todo el sistema, dado que las tareas como: la recepción de datos vía IEEE-1394, la visualización, la ocurrencia de eventos en la capa de enlace y otras, requieren de un tiempo crítico fijo y no pueden ser tratadas por encuesta. Es así que el controlador permite la gestión de numerosas fuentes de interrupción (hasta 8). Determinado por las características del tipo de dispositivo y el nivel de prioridad que tengan estas para el sistema. En este sistema deben ser atendidas 6 fuentes de interrupción [33]:
- IRQ0 -> VGA
- IRQ1 -> DMA0
- IRQ2 -> DMA1
- IRQ4 -> UART
- IRQ5 -> Contador_salida_1
- IRQ6 -> Contador_salida_2

Resulta de vital importancia IRQ5 e IRQ6 al tomarse para activar la conmutación de tareas en el programa.

3.4.3 MIPS

Con el nombre de MIPS (*Microprocessor without Interlocked Pipeline Stages*) se conoce a toda una familia de microprocesadores de arquitectura RISC o Computador con Conjunto de Instrucciones Reducidas (*Reduced Instruction Set Computer*) [37] desarrollados por *MIPS Technologies*. Los diseños del MIPS son utilizados en líneas de productos

informáticos, en muchos sistemas embebidos, en dispositivos para WindowsCE, *routers* Cisco, y videoconsolas. Las primeras arquitecturas MIPS fueron implementadas en 32 bits (generalmente rutas de datos y registros de 32 bits de ancho), y luego versiones posteriores fueron desarrolladas para 64 bits. Existen cinco revisiones compatibles hacia atrás del conjunto de instrucciones de MIPS, cada una mejorada o con más instrucciones que la anterior.

3.4.4 MIPS I

Dentro de la familia de los microprocesadores MIPS se encuentra uno de los más sencillos pero versátiles: el MIPS I, microprocesador base para sintetizar Plasma y centro del sistema operativo [38]. Dentro de las características de este microprocesador destacan:

- Soporta los tipos de datos enteros: byte, wordy doubleword.
- Antes de ser utilizado en una instrucción aritmética, todo dato debe ser cargado previamente en un registro de propósito general.
- Presenta instrucciones aritméticas con 3 operandos de 32 bits en registros.
- Tiene varios modos de funcionamiento: usuario, núcleo (*kernel*), supervisor y depuración.
- Posee 32 registros de propósito general pero es importante destacar que cuando se producen los cambios de contexto solo salvan algunos registros, perdiéndose der8-r15, r24 y r25 ($t0-$t7 y $t8 y $t9) ya que son registros temporales.
- Tiene varios registros de propósito específico como son **HI** y **LO** empleados en operaciones. **PC** que es el contador de programa y **EPC** que es el contador de excepciones de programa.
- Puede funcionar con ordenamiento de datos *little-endian* o *big-endian*.
- Un procesador de la familia de MIPS puede tener hasta 4 coprocesadores de hasta 32 registros cada uno, accesibles mediante instrucciones especiales y se pueden reservar para implementaciones específicas.

En el caso de MIPS I solo posee 2 coprocesadores, el 0 y el 1. El Coprocesador 1 está reservado para la unidad de coma flotante, o sea para trabajar con datos no enteros,

mientras, el Coprocesador 0 es el más importante y de obligatoria incorporación en el chip de la CPU ya que se encarga del control del procesador [36].

Funciones del Coprocesador 0:

- Coprocesador de control de sistema (*System Control Coprocessor*).
- Controla el subsistema de memoria caché.
- Soporta el sistema de memoria virtual y traduce direcciones virtuales en físicas.
- Soporta el manejo de excepciones.
- Maneja los cambios en el modo de ejecución (usuario, núcleo, supervisor).
- Proporciona control de diagnósticos y recuperación ante errores.

El Coprocesador 0 puede contener hasta 32 registros y los más importantes son:

BadVAddr (registro 8): cuando un acceso a memoria produce una excepción, este registro contiene la dirección de memoria que la provocó.

Registro de estado (registro 12): habilita o deshabilita las interrupciones.

Causa (registro 13): contiene información acerca de la causa que ha producido una excepción.

EPC (registro 14): si una instrucción produce una excepción, contiene la dirección en memoria de dicha instrucción (contador de programa de excepción).

Además hay otros registros que se emplean para gestionar el sistema de memoria, los errores de paridad en accesos a memoria y las informaciones de traza entre otras. La estructura de MIPS I se muestra en la Figura 3.3.

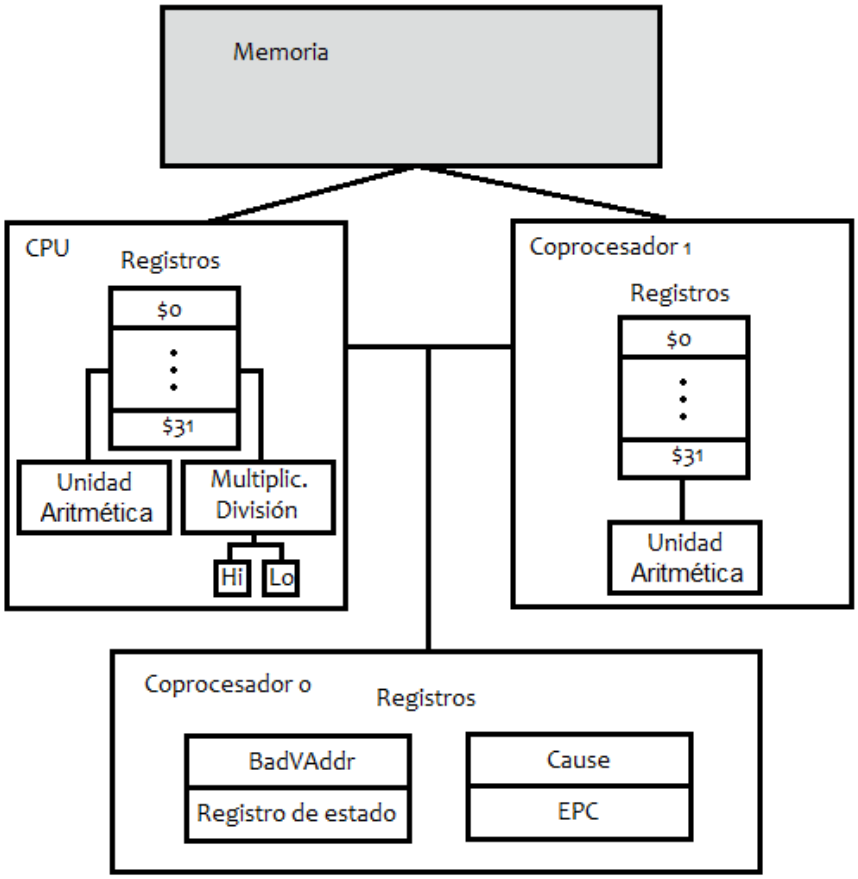

Figura 3.3. Estructura de MIPS I.

3.4.5 MIPS Lite

El CPU MIPS Lite es un procesador de 32-bit sintetizable, que es capaz de ejecutar todas las instrucciones de MIPS I, excepto las operaciones de carga/almacenamiento no alineadas. El procesador tiene una arquitectura Von Neumann y es capaz de direccionar hasta 4 GB de memoria RAM distribuidos en 4 bancos de 1 GB cada uno, además es configurable en 2 y 3 etapas de pipeline y soporta los tipos de datos enteros: byte (8 bits), word (16 bits) y doubleword (32 bits). Este procesador está implementado en lenguaje VHDL.

Arquitectura externa del procesador MIPS Lite

El bus del sistema está compuesto por las siguientes señales; dos de datos **data_r[31:0]** como entrada y **data_w[31:0]** como salida, una señal de direcciones, **address[31:2]**y otra de control**, byte_we[3:0]**. La señal de control **byte_we** es de 4 bits, activos en nivel alto, donde cada bit representa la habilitación en escritura de un banco de memoria. Las

señales de dirección y control se encuentran en dos versiones: valor actual y valor después del próximo flanco del reloj, a las que se le añaden el sufijo *next* para identificarlas. Se incluyen además la señal de reloj **clk**, de reinicio **reset_in**, una señal de entrada de interrupción **intr._in** y una señal para introducir estados de espera en accesos a memoria o periféricos **mem_pause** como se muestra en la Figura 3.4.

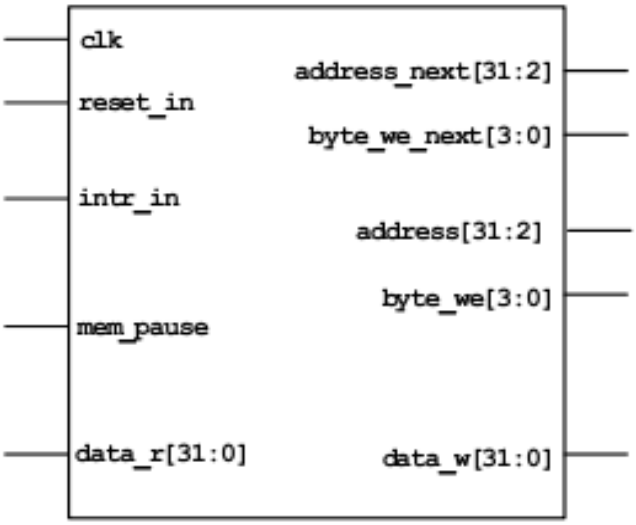

Figura 3.4. Arquitectura externa del procesador MIPS Lite

Arquitectura interna del procesador MIPS Lite

La arquitectura interna del procesador está compuesta por 8 bloques funcionales que se describen a continuación y se muestran en la Figura 3.5:

- Puntero de instrucción **PC_next**: determina las funciones lógicas del puntero de instrucción (IP) o contador de programa (PC) que apunta a la próxima instrucción a leer de la memoria.
- Controlador de memoria **Mem_ctrl**: determina el comportamiento del bus del sistema para las diferentes transacciones que soporta el procesador.
- Banco de registros **Reg_bank**: posee 32 registros de propósito general similar al disponible en la arquitectura MIPS convencional. Incorpora dos canales de lectura y un canal de escritura.
- Unidad de Multiplexación de Buses **Bus_mux**: es el principal bloque de interconexión interna de los módulos que componen el procesador, en este sentido, se comporta como un bus interno. En dependencia de las señales de la

unidad de control dirige los operandos provenientes de múltiples fuentes a la unidad del camino de datos que corresponda.

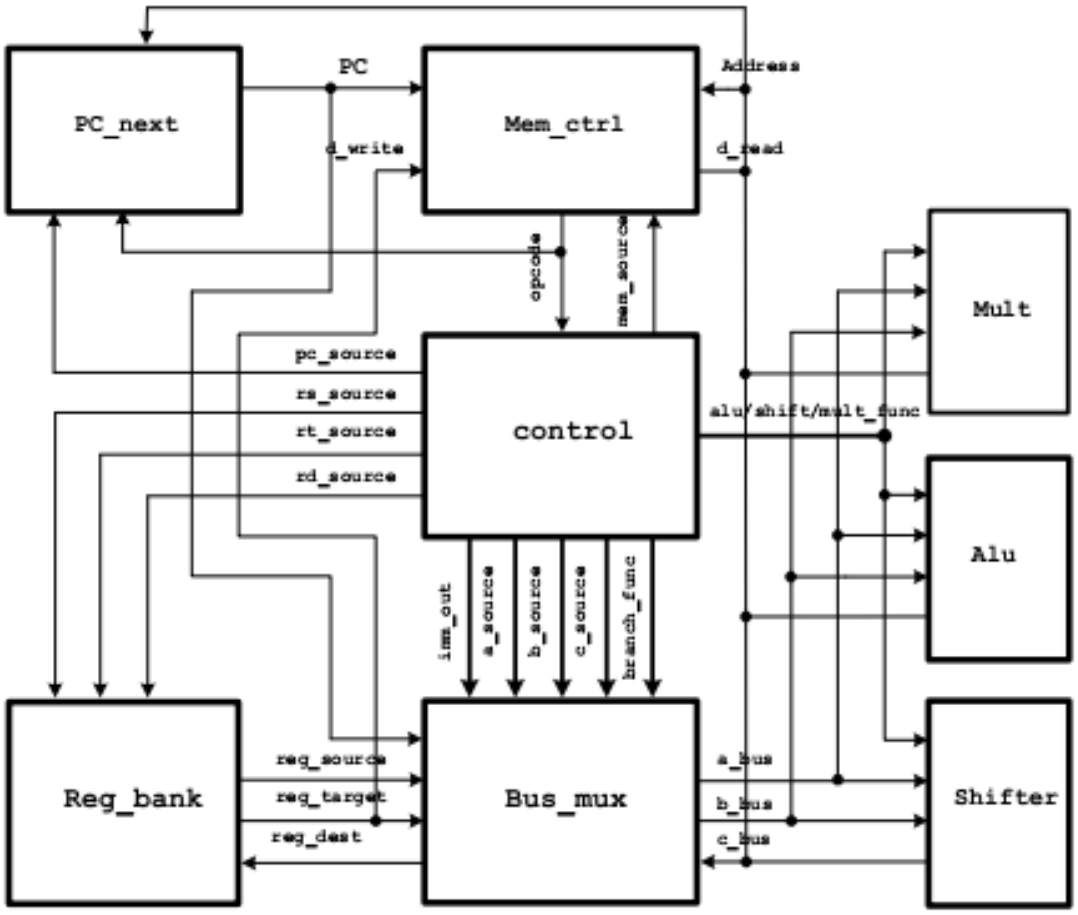

Figura 3.5. Arquitectura interna del procesador MIPS Lite

- Multiplicador **Mult**: implementa las operaciones aritméticas de multiplicación y división. Esta unidad es opcional y para cada una de las operaciones toma 32 ciclos de reloj.
- Unidad Aritmética–Lógica **ALU**: realiza las operaciones aritméticas suma, resta, comparaciones y las operaciones lógicas habituales: AND, OR, XOR y NOR.
- Desplazador ***Shifter***: es un desplazador combinacional de 32 bits. Permite las instrucciones de desplazamiento lógico.
- Unidad de Control: es al unísono unidad de control y decodificador de instrucción. El bloque decodifica el *set* de instrucciones MIPS (32 bits) y lo convierte al formato

Verilog Instruction Word (VLIW) (60 bits).Estas señales decodificadas son enviadas como señales de control hacia el resto de las unidades del procesador.

Los registros que pueden ser usados en este procesador son mostrados en la Tabla 3.1 a partir de los criterios de [30]:

Tabla. 3.1. Registros disponibles en MIPS Lite

Nombre del registro	Número	Uso del registro
zero	0	Constante “0”
at	1	Reservada para ensamblador
v0	2	Evaluación de expresiones y resultado de funciones
v1	3	Evaluación de expresiones y resultado de funciones
a0	4	Argumento 1
a1	5	Argumento 2
a2	6	Argumento 3
a3	7	Argumento 4
t0..t7	8..15	Temporal (no se guarda valor entre llamadas)
s0..s7	16..23	Temporal (el valor se guarda entre llamadas)
t8, t9	24, 25	Temporal (no se guarda valor entre llamadas)
k0, k1	26, 27	Reservado para el *kernel* del sistema operativo
gp	28	Puntero al área global
sp	29	Puntero de pila
fp	30	Puntero de marco de pila
ra	31	Dirección de retorno, usada por llamadas a función
HI-LO	lo-hi	Resultados de multiplicación y división
PC	PC	Contador de programa
EPC	epc	Contador de excepción de programa

Las principales limitaciones del procesador MIPS Lite se encuentran en las estrechas posibilidades del bus del sistema, ya que es un bus sencillo, no estándar e incompatible con los buses más extendidos de la actualidad. Además es un bus compartido de maestro único por lo que es difícil hacerlo compatible con otros sistemas que incorporen uno o más maestros. Este problema fue resuelto en Ryder [33] empleando el bus Wishbone.

3.5 Archivo Config.h

Para crear un sistema operativo empleando FreeRTOS hay que configurar un gran número parámetros, los que se establecen de manera distinta para cada aplicación. Estos parámetros se encuentran en el archivo Config.h [20] y van desde la frecuencia de reloj que se empleará, hasta los niveles de prioridad que utilizan las tareas.

Programador o Núcleo (*Scheduler* o *Kernel*)

El núcleo es una de las partes más importantes del sistema operativo FreeRTOS ya que es el encargado de conmutar las tareas en dependencia de su prioridad y posee dos posibles métodos de configuración. Estos métodos son:

- Modo preventivo, en el cual siempre se ejecuta la tarea disponible de más alta prioridad y donde tareas con igual prioridad, comparten el tiempo de procesamiento con el mismo intervalo para cada tarea.
- Modo cooperativo donde los cambios de contexto sólo ocurren si una tarea se bloquea, o explícitamente llama a la función **taskYIELD.**

Esta configuración se efectúa con **configUSE_PREEMPTION.** Definiendo en “1” este parámetro se configura el núcleo para trabajar en modo preventivo o “0” para colocarlo en modo cooperativo. En la aplicación se coloca en modo preventivo.

Uso del “gancho holgazán”

La terea holgazana es una tarea que crea el propio sistema operativo la primera vez que se llama la función **xTaskCreate ()** y su función es liberar la memoria ocupada por las tareas un vez que se hayan eliminado. Esta tarea es la de más baja prioridad con valor de “0” aunque este parámetro se puede modificar. Un gancho holgazán es una función que se llama cada vez que se ejecuta la tarea holgazana y el código de esta función debe ser definido por el usuario. Su uso se especifica colocando en “1”el parámetro **configUSE_IDLE_HOOK** o colocándolo en “0” para ignorarla.

Frecuencia del reloj

Este parámetro establece la frecuencia de reloj a la cual se va a trabajar y tiene que estar relacionado con la frecuencia disponible en el *hardware.* Se modifica con **configCPU_CLOCK_HZ.**

Frecuencia de la interrupción *tick*.

Las interrupciones *tick* son una forma de medir el tiempo usado para que el núcleo pueda determinar el periodo de ejecución a cada tarea. El núcleo emplea el *tick* y la prioridad de las tareas para determinar cuántos *ticks* corresponden a cada tarea. Esta frecuencia se define con **configTICK_RATE_HZ.**

Número de prioridades

Se definen el número de prioridades disponibles para la aplicación. Muchas tareas pueden compartir la misma prioridad pero cada prioridad disponible consume memoria RAM dentro del núcleo, por lo que este valor no debería ser más alto que lo requerido por la aplicación. Se configura con **configMAX_PRIORITIES.**

Tamaño de la pila para la tarea holgazana

Define el tamaño de la pila utilizada por la tarea desocupada. Normalmente este tamaño no se modifica ya que es suministrado por el fabricante y se configura con **configMINIMAL_STACK_SIZE.**

RAM ocupada por el Núcleo

Define la porción de la memoria RAM ocupada por el núcleo y tiene estrecha relación con el modelo de gestión de memoria empleado. Se configura con **configTOTAL_HEAP_SIZE.**

Longitud de los nombres de las tareas

Define la máxima longitud permisible para los nombres de las tareas incluyendo los detalles nulos. Se define con **configMAX_TASK_NAME_LEN.**

Uso de la visualización de huellas

La visualización de huellas es una herramienta que brinda el sistema operativo para visualizar el tiempo de ejecución de cada tarea y permite guardar la información para ser visualizada en plataformas de texto (.doc., .txt.). Esta función se habilita con **configUSE_TRACE_FACILITY** colocado en “1” y se deshabilita con “0”.

Configuración del *tick*

Como el tiempo en los RTOS se mide en *tick* es necesario la existencia de un contador para los *tick* y este contador puede ser definido de dos formas: como un entero sin signo de 16bits, o como entero sin signo de 32bits. El empleo de un contador de 16 bits tiene un gran desempeño para las arquitecturas de procesadores de 8 y 16 bits pero posee un periodo mucho menor que los de 32 bits. Asumiendo una frecuencia de 250Hz, el tiempo máximo que una tarea puede demorarse o bloquearse con 16bits son 262 segundos, mucho menor comparado con los 17179869 segundos cuándo se emplean 32 bits. Estos modos se definen al colocar**configUSE_16_BIT_TICKS** en “1” para 16 bits y “0” para 32 bits.

Prioridad de tareas igual a la tarea holgazana

Este parámetro controla el comportamiento de las tareas cuando tienen la misma prioridad (A, B, C) que la tarea holgazana (I) y solo se puede usar cuando el núcleo está configurado en modo preventivo. Esta función permite que, cuando se tienen varias tareas con prioridad “0” ejecutándose y una de ellas no consume completamente su ranura de tiempo, la tarea holgazana permite que se ejecute esta en su mismo espacio de tiempo, lo que provoca una reducción del tiempo de ejecución de la tarea holgazana (I) y también se reduce el tiempo de la tarea (A), lo que mejora el rendimiento del sistema operativo. Se configura colocando **configIDLE_SHOULD_YIELD**a “1” como se muestra en la Figura 3.6.

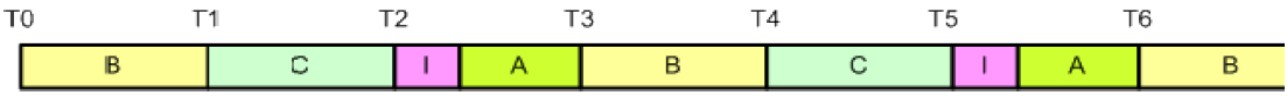

Figura 3.6. Ejecución de varias tareas con prioridad “0”

Parámetros INCLUDE

Las macroinstrucciones comenzadas por 'INCLUDE' [18] permiten que los componentes que no son utilizados por el núcleo se eliminen de la ejecución, lo que permite ahorrar memoria RAM. Cada macro instrucción se define por **INCLUDE_FunctionName.** Para incluir instrucciones del API solamente debe colocarse en “1”la función que se desea incluir y para eliminarla se coloca en “0”. Por ejemplo:

#define INCLUDE_vTaskDelete 1

#define INCLUDE_vTaskDelete0

3.6 FreeRTOS soportado sobre MIPS

Una vez se tienen definidos los parámetros imprescindibles del sistema operativo se hace necesario portarlo a la plataforma Plasma-Wishbone y dentro de ella al procesador MIPS Lite. Los fabricantes de FreeRTOS, suministran los archivos necesarios para que el sistema operativo pueda ejecutarse sobre diversas arquitecturas de microprocesadores, entre las que podemos encontrar: ARM, ATMega, Nios, Tricore entre otras; pero no los suministra para el procesador MIPS Lite ni para MIPS. Por esta razón se toma como base el procesador MicroBlaze, para el cual si está definido el portable, el procesador tiene puntos en común con MIPS y se modifican sus archivos para que el sistema operativo funcione en MIPS Lite.

3.6.1 MIPS Lite vs MicroBlaze

Para modificar los archivos de MicroBlaze se hizo necesario un estudio profundo de las dos arquitecturas de microprocesadores MIPS Lite y MicroBlaze, ello permitió conocer sus semejanzas y diferencias, razones que a continuación son resumidas:

- Ambas arquitecturas poseen 32 registros de propósito general y los dos registros especiales **HI** y **LO**, aunque ambos no realizan las mismas funciones. En MIPS existen varios registros que pierden su valor en los cambios de contexto, no sucediendo así en MicroBlaze.
- Soportan el ordenamiento de datos *little-endian* o *big-endian*, y permiten los tipos de datos enteros: byte, word y doubleword.
- Los dos poseen una arquitectura Von Neumann y tienen *sets* de instrucciones RISC de 32 bits. Aunque algunas instrucciones son iguales otras son exclusivas para una arquitectura y no presentan equivalencia en la otra.
- Ostentan una sola entrada de interrupción pero el proceso de tratamiento a las mismas es diferente.
- Para el control del procesador, excepciones, interrupciones y otros, MicroBlaze emplea registros de propósito especial **SPR** (*Special Purpose Register*), donde se destacan: **MSR** *(Machine Status Register)* = SPR (1), **ESR** (*Exception Status Register)* = SPR (5), **EAR** (*Exception Address Register)* = SPR (3) y **FSR (***Floating Point Unit Status Register)* = SPR (7). Por su parte en MIPS se emplea el Coprocesador 0con sus

registros: **BadVAddr** (registro 8), **Registro de estado** (registro 12), **Causa** (registro 13) y **EPC** (registro 14).

Con esta información se modifican exitosamente los archivos del portable de MicroBlaze para que el sistema operativo funcione sobre la arquitectura MIPS Lite. Algunos ejemplos del trabajo realizado se muestran a continuación:

Para almacenar el valor de un registro en MicroBlaze una instrucción tendría el siguiente formato: **swi r30, r1, 12** [28]. En este caso el valor del registro **r30** se guarda en la dirección de memoria resultado de la suma del valor del registro r1 y el valor inmediato **12**. Se emplea **r1** porque en MicroBlaze **r1** es el puntero de pila. Mientras que en MIPS se utiliza la función **sw $30, 12 ($29)** [39] que almacena el valor del registro **r30** en la posición de memoria de **r29** (puntero de pila en MIPS) con el desplazamiento u *offset* del valor inmediato **12**.

Para habilitar o deshabilitar las interrupciones en MicroBlaze se emplea **mts rmsr, r3** que almacena en el registro de estado de máquina **msr** el valor del registro **r3,** mientras que en MIPS se emplea **mtc0 $3, $12** que mueve el valor del registro **r3** al registro de estado del Coprocesador “0”. Para consultar el resto de los cambios realizados, debe remitirse a los sets de instrucciones de ambos procesadores antes referenciados.

3.7 Demos

Una vez portado el sistema operativo sobre el microprocesador se crean dos *demos* que permiten demostrar el correcto funcionamiento del sistema operativo sobre la plataforma Plasma-Wishbone y que además sirven de base para realizar futuras operaciones con la plataforma. El primer ejemplo es un programa sencillo donde se crean dos tareas genéricas, cuyo objetivo es imprimir el nombre de la tarea cada vez que el núcleo ejecute una de ellas y un programa principal que se encarga crearlas, definirle su prioridad e iniciar el núcleo para que comience a trabajar.

El segundo ejemplo es la modificación de uno de los programas de prueba que ofrecen los fabricantes de FreeRTOS donde se crean varias tareas para chequear los recursos del sistema operativo. Una tarea se encarga de verificar el almacenamiento en los registros sin perder la información en los cambios de contexto. Las otras comprenden el chequeo del correcto funcionamiento de los servicios de atención a interrupciones, examina los semáforos, colas y mutex, verifica el cumplimiento de la prioridad de las tareas y la

información de tiempo y la última comprueba que todas corran correctamente y lo informa con el parpadeo de un LED. En caso de un error en alguna de las tareas se puede saber en cual se produjo porque la frecuencia de parpadeo del LED es distinta para el fallo de cada tarea.

Se emplea la opción que ofrece FreeRTOS de visualización de huellas habilitando **configUSE_TRACE_FACILITY** en el archivo Config.h y empleando la función **vTaskStartTrace** que muestra gráficamente el tiempo de ejecución de cada tarea. En las Figuras 3.7 y 3.8 se muestra el primer *demo* donde las tareas 1 y 2 comparten la misma prioridad (1). Por este motivo presentan un tiempo de ejecución igual, mientras que la tarea holgazana, creada de manera automática por el sistema operativo, tiene menor prioridad (0), por lo que posee menor tiempo de ejecución.

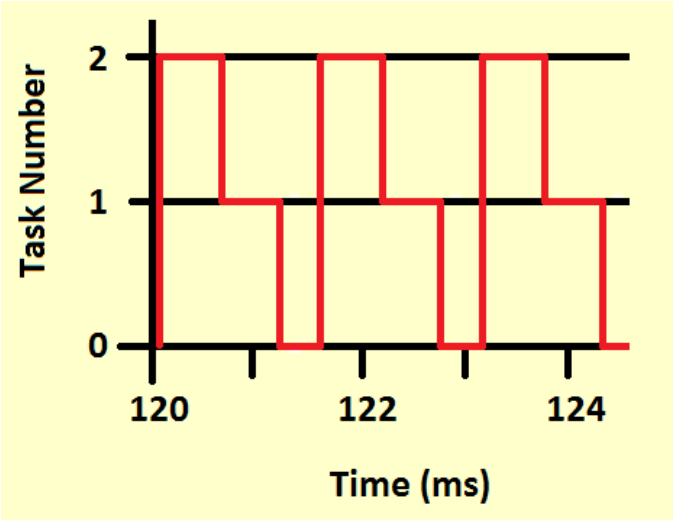

Figura 3.7. Tiempo de ejecución da cada tarea en el primer demo

Name	State	Priority	Num
Task1	R	1	2
Task2	R	1	1
IDLE	R	0	0

Figura 3.8. Estados de las tareas en el primer demo

3.8 Software

Con el sistema operativo en funcionamiento se procede a conformar el *software* que soporte la comunicación entre el protocolo IEEE-1394 y una computadora vía USB. Para diseñar el mismo se realizan dos tareas, la primera implementa IEEE-1394 y la segunda USB. Las tareas se crean para que trabajen en el nivel de transacción, por lo que para el dispositivo final se necesita una capa física que entregue y reciba información en los dos formatos de datos.

En el caso de la recepción, la primera tarea se encarga de recibir los datos con el formato IEEE-1394 y eliminar toda la información que no sea carga útil, y luego pasa a colocar los datos en una posición en la cual el resto de las tareas puedan accederlos. Por otro lado, la trasmisión toma los datos en brutos y le agrega toda la información de direccionamiento y de FireWire a los *quadlets* para enviarlos a las capas más bajas. La segunda tarea funciona de manera similar a la primera, sólo que implementa el protocolo de comunicación USB.

De esta forma cuando se inicia el núcleo si se reciben datos por una de las dos interfaces de comunicación la tarea correspondiente los leerá, modificará los datos y los colocará en la cola para que la otra tarea los lea, modifique, cree las cabeceras y envíe a través de la otra interfaz. Estas dos tareas funcionaran al unísono ya que tienen igual prioridad, lo que permite que no sea necesario esperar los envíos de todos los datos por una interfaz para empezar a transmitir por la otra.

3.9 Análisis económico

En la actualidad existen muchos dispositivos, sobre todo en electrónica de consumo que se comunican mediante el protocolo de comunicación IEEE-1394, y para lograrlo requieren de la existencia de una placa base que incorpore este tipo de conectores y el protocolo. En el mercado internacional se pueden encontrar diversos módulos de expansión que incorporan conectores y protocolos necesarios para la comunicación con IEEE-1394. En este caso los precios varían en dependencia del fabricante, las prestaciones ofrecidas y el número de puertos. A continuación se relacionan los precios de algunos de estos módulos en la Tabla 3.2 obtenidos en el sitio oficial de *StarTech* [40].

Tabla. 3.2. Características de algunos módulos de expansión IEEE-1394

Tipo de tarjeta	**Características**	**Precio (dólares)**
Tarjeta Adaptadora PCI FireWire 1394a de 4 Puertos con Kit de Edición de Video Digital StarTech.com. Modelo: PCI1394_4	Posee 4 puertos FireWire-400 externos y 1 interno compartido, con soporte para velocidades de transferencia de hasta 400Mbps. Compatible con los estándares 1394a - 2000 y 1394-1995. Incluye kit de Edición de Video: Ulead Video Studio SE y cable FireWire. Soporta *Plug and Play y Hot-Swap.*	$ 300.00
Tarjeta PCI Express con 2 Puertos FireWire 1394a - StarTech.com. Modelo: PEX1394A2	Contiene 2 puertos nativos FireWire 400 empleando una ranura de expansión PCI Express. Incluye un soporte de perfil.	$ 47.00
Tarjeta FireWire PCI A IEEE – 1394 de 3+1.	PCI VIA VT6306 chipset de la interfaz 3 +1 (3 externos y 1 puerto interno).*PHY* Soporta el dominio de alto rendimiento de bus. Soporta hasta 400Mbite/sec.	$ 40.00
Tarjeta PCI 4 Puertos FireWire 1394 + Cable.	Fácil conectividad *Plug and Play*. Transferencia de datos USB 2.0 hasta 480 Mbps, IEEE – 1394 hasta 400 Mbps. Compatible con dispositivos USB 1.1. Puertos externos USB2.0 e IEEE – 1394 FireWire perfectamente adaptables a la PC. Soporta *devices hot swap*.	$ 30.00

Adquirir algunos de estos módulos de expansión e incorporarlos a varias computadoras hubiese sido la solución más sencilla para lograr la comunicación de dispositivos IEEE-1394. Sin embargo en el país todavía no es posible realizar compras a través de internet, la mayoría de los sitios de venta se encuentran en los Estados Unidos y producto al bloqueo económico no se permite el envío de este tipo de mercancías, siendo la única vía la compra a través de terceros países y personas externas lo que encarecería el costo de las mismas y también se ve limitado.

En estas condiciones resulta necesario diseñar la interfaz de comunicación con tarjetas de desarrollo que incorporan FPGA y que se encuentran en el departamento de Ingeniería Biomédica, como una forma práctica de resolver el problema sin requerir una nueva inversión, ya que se trabaja con los recursos existentes y se ahorra un presupuesto que puede ser empleado en otras necesidades.

Con el aporte del *software* para la interfaz IEEE- 1394 con una PC vía USB se contribuye a la disminución de los costos, en tanto hoy las aplicaciones informáticas también constituyen mercancías. De esta forma se ofrece una solución práctica y sencilla a ciertos problemas que de otra forma tendrían soluciones más costosas.

3.10 Conclusiones del capítulo

Una vez concluido el *software* y probado con los distintos *demos* se demuestra que:

- El sistema operativo de tiempo real ha sido portado exitosamente sobre el procesador MIPS Lite en la plataforma Plasma-Wishbone.
- Funciona de forma correcta y el núcleo dentro de éste está realizando los cambios de contexto entre las tareas adecuadamente.
- La prioridad de las tareas está bien implementada ya que el núcleo asigna distintas ranuras de tiempo a las tareas con deferente prioridad.
- La tarea holgazana libera espacio y realiza funciones de mantenimiento.

Se aporta una poderosa herramienta que se puede generalizar a muchos sistemas embebidos y dar soluciones prácticas, económicas y sencillas a disímiles problemas de la vida cotidiana. El *software* fue pensado para comunicar un dispositivo IEEE-1394 a una PC vía USB, pero si el problema fuese otro, solo se necesitaría modificar las tareas en función de las nuevas necesidades, siempre y cuando se trabaje con el mismo SoC.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

- A partir del estudio del estándar de comunicación IEEE-1394, el sistema operativo de tiempo real FreeRTOS y la plataforma Plasma-Wishbone se diseñó un *software* que permite la transmisión de información en tiempo real entre dispositivos IEEE-1394 y una PC vía USB en una plataforma embebida en una FPGA.
- Se consiguió portar el sistema operativo de tiempo real FreeRTOS a la plataforma basada en MIPS Plasma-Wishbone. Se comprobó a través de los demos el correcto funcionamiento y desempeño del sistema operativo propuesto, como base al desarrollo del software de comunicación para el dispositivo sintetizado sobre una FPGA.
- Se elaboró un software para la transmisión de información en tiempo real entre dispositivos IEEE-1394 y una PC vía USB que ofrece una interfaz genérica y simple la cual puede modificarse y adaptarse a nuevas necesidades y cambios de hardware.

Recomendaciones

- Usar el modelo de asignación y retiro de memoria “memoria constante y numerada” para lograr más eficiencia en la gestión de los recursos.

- Generalizar el uso de este *software* a nuevas plataformas y a otras aplicaciones.

REFERENCIAS BIBLIOGRÁFICAS

[1] Sitio oficial de Apple™. Disponible en: http://www.apple.com. Consultado: 15/5/14

[2] *IEEE Std 1394-1995, Standard for a High Performance Serial Bus*. 1996. ISBN 1-55937-583-3. Disponible en: http://shop.IEEE.org/IEEEstore. Consultado:15/5/14

[3] *IEEE 1212-1991: Control and Status Register (CSR) Standard*. Disponible en: http://standards.IEEE.org/findstds/standard/1212-1991.html. Consultado: 15/5/14

[4] *IEEE Std 1394a-2000 (Amendment to IEEE Std 1394-1995)*. Disponible en: http://shop.IEEE.org/IEEEstore. Consultado: 15/5/14

[5] *IEEE Std 1394b-2002 (High Speed Supplement)*. Disponible en: http://shop.IEEE.org/IEEEstore. Consultado: 15/5/14

[6] *1394b for Military Applications*. SAE-AS5643. Disponible en: http://www.sae.org/tecnical/standars/AS5643/1. Consultado: 15/5/14

[7] *1394 Trade Association*. *IEEE 488 over 1394*. Disponible en: http://www.1394ta.org/developers/specifications/1999017.html. Consultado: 20/5/14

[8] *Allied Vision Technologies*. Disponible en: http://www.alliedvisiontec.com. Consultado: 20/5/14

[9] TOPCON *Corporation*. *Retinal Camera Instruction Manual*. TRC-50DX.

[10] *1394 Automation Protocol.* Disponible en: http://www.1394ta.org/developers/specifications/2005099.html. Consultado: 20/5/14

[11] Johansson, P. *IPv4 over IEEE 1394. RFC 2734.* 1999. Disponible en: http://www.ietf.org/rfc.html. Consultado: 20/5/14

[12] Fujisawa K. *DHCP for IEEE 1394*. RFC2855. 2000. Disponible en: http://www.rfc-editor.org/rfc/rfc2855.txt. Consultado: 20/5/14

[13] VIA Fire VT6308 1394a (VIA Fire IEEE – 1394 *Link-layer* / PHY *chipset solution*)

[14] Tanembaum, Andrew S., Redes de computadoras, 3ra ed., Ed. Pearson, 2001.

[15] *Texas Instruments. TSB12LV31 Data Manual. IEEE 1394-1995 General-Purpose Link-Layer Controller.* 1998.

[16] *1394 Trade Association. FireWire Design Guide 1.0*. Disponible en:http://www.1394ta.org/developers/DesignGuide/TAFWDesignGuide.pdf. Consultado 29/5/2014

[17] Stallings, William, Comunicaciones y redes de computadoras, 6ta ed., Ed. Prentice Hall, 2000. Cap. 7.

[18] Barry, Richard. *A FREE RTOS for small real time embedded systems*. 2005. Disponible en: http://127.0.0.1:800/Default/www.freertos.org/modules.html. Consultado 20/5/2014

[19] Monografías S.A. Historia de los sistemas operativos. Disponible en: http://Monografias.com S.A. Consultado: 20/5/14

[20] Barry, Richard. *Using the FreeRTOS real time kernel. A Practical Guide*. Version 1.0.5. 2009 Richard Barry

[21] Vignoni, Roberto. Sistemas operativos de tiempo real, Universidad Nacional de La Plata, Argentina.

[22] Becker, Carlos. Sistemas Operativos en Tiempo Real (*Real Time Operative Systems*) en sistemas embebidos. Universidad nacional de Rosario, Argentina.

[23] Barry, Richard. *Using The FreeRTOS Real Time Kernel. NXP LPC17xxEdition*. 2010.

[24] Melot, Nicolas. *Study of an operating system: FreeRTOS.*

[25] Orozco, Javier. Sistemas de Tiempo Real con Requerimientos Heterogéneos. Integración *Hardware-Software*. Universidad Nacional del Sur, Bahía Blanca, Argentina.2013.

[26] Atmel Corporation. *ASF Specific FreeRTOS Functionality for Peripheral Control.* 2325 Orchard Parkway San Jose, CA 95131 USA. 2012

[27] Digilent. Digilent Nexys2 *Board Reference Manual.* Revision: 2011.Copyright Digilent, Inc.

[28] Xilinx. MicroBlaze *Processor Reference Guide. Embedded Development Kit.* EDK 10.1i UG081 (v9.0). Xilinx, Inc.2008.

[29] Lampret, D.OpenRisc1200 *IP Core Specification*, Opencores, 2001.

[30] Rhoads, Steve. *Plasma - most MIPS I(TM) opcodes*. Disponible en: http://opencores.org/project,Plasma. Consultado: 20/5/14

[31] OpenCores *organization*, OpenCores *project*, Disponible en: http://www.opencores.org.Consultado: 20/5/14

[32] Gaisler, J. *Leon 2 Processor User's Manual*", Gaisler. 2004.

[33] Melian Ávila, Rider. Propuesta de arquitectura de nodo IEEE-1394 para procesamiento y visualización de imágenes y vídeo. Universidad de Oriente. Facultad de Ingeniería Eléctrica. Departamento de Ingeniería Eléctrica.2012.

[34] Rhoads S. "*Plasma Web Server*".2005, Disponible en: http://Plasmacpu.no-ip.org:8080/.Consultado: 20/5/14

[35] Silicore Corp. *Specifications for*: WISHBONE *System-on-Chip Interconnection Architecture for Portable IP Cores*. Revisión: B.3, 2002.

[36] Rincón Córcoles, Luis y Sánchez de León, Ángel Serrano. Conferencias de Estructura y Tecnología de Computadores (ITIG), 10 Introducción a los microprocesadores MIPS. Universidad Rey Juan Carlos. Comunidad de Madrid. España.

[37] Price, Charles. *MIPS IV Instruction Set.* Revision 3.2.MIPS Technologies, Inc.1995.

[38] Rincón Córcoles, Luis y Sánchez de León, Ángel Serrano. Conferencias de Estructura y Tecnología de Computadores (ITIG), 12 Programación en ensamblador MIPS. Universidad Rey Juan Carlos. Comunidad de Madrid. España.

[39] MIPS Technologies. *MIPS 32 Architecture For Programmers* – Volumen I, Volumen II, Volumen III: The MIPS32 *Instruction Set*. MIPS Technologies Inc. 2003.

[40] Sitio oficial de StarTech Corporation. Disponible en: http://www.StartTecch.com.Consultado: 20/5/14

GLOSARIO DE TÉRMINOS

AND	Operador lógico que resulta en verdadero si los dos operadores son verdaderos.
ANSI	Instituto Nacional Estadounidense de Estándares.
API	Interfaz de programación de aplicaciones.
ARM	Arquitectura RISC de 32 bits desarrollada por ARM Holdings.
ASIC	Circuito Integrado para Aplicaciones Específicas.
ATMega	Microcontroladores AVR grandes con conjunto de instrucciones extendido y amplio conjunto de periféricos.
BadVAddr	Cuando un acceso a memoria produce una excepción, este registro contiene la dirección de memoria que la causó.
C	C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie.
Causa	Contiene información acerca de la causa que ha producido una excepción.
CPU	Unidad Central de Procesamiento.
CSR	Registros de Comando y Estado.
DC	Corriente Directa.
DMA	Acceso directo a memoria
EAR	Registro de direcciones de excepciones.
EPC	Contador de excepciones de programa.
ESR	Registro de estado de excepciones.
FIFO	Concepto utilizado en estructuras de datos y teoría de colas para referirse a algoritmos donde el primero en llegar es el primero en ser atendido.
FLASH	Memoria derivada de la memoria EEPROM que permite la lectura y escritura de múltiples posiciones de memoria en la misma operación.
FPGA	Arreglos Programables de Campos de Compuertas.
FreeRTOS	Sistema operativo de tiempo real desarrollado por Richard Barry de Real Time Engineers Ltd.

FSR	Registro de estado de la unidad de coma flotante.
GB	Unidad de medida de información que equivale a 10^9 bits.
HDL	Lenguaje de descripción de hardware.
HI-LO	Registros especiales empleados para la multiplicación y división.
HPSB	Bus serie de alta velocidad.
IEC	Comisión Electrotécnica Internacional.
IEEE	Comité de estandarización del Instituto de Ingenieros Eléctricos y Electrónicos.
IEEE-1394 o FireWire	Conexión para diversas plataformas, destinado a la entrada y salida de datos en serie a gran velocidad.
IP	Puntero de instrucción.
IP Cores	Componentes desarrollados en lenguajes descripción de hardware, para permitir a los diseñadores reducir los tiempos de desarrollo de las distintas aplicaciones.
ISO	Organización Internacional de Normalización.
Lámparas 7 segmentos	Dispositivo de ciertos aparatos electrónicos que permite mostrar Dispositivo que permite mostrar información al usuario de manera visual.
LED	Diodo emisor de luz.
MB	Unidad de medida de información muy utilizada en las transmisiones de datos que equivale a10^6 bits.
MicroBlaze	Procesador de 32-bit de propósito general desarrollado por Xilinx.
MIPS	Familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.
MIPS Lite	Procesador de 32-bit sintetizable, que es capaz de ejecutar todas las instrucciones de MIPS I, excepto las operaciones de carga/almacenamiento no alineadas.
MSR	Registro de estado de máquina
NASA	Administración Nacional Norteamericana de Aeronáutica y del Espacio.
Nios	Primer procesador embebido configurable de 16-bit de Altera para su línea de productos FPGA.

NOR	Operador lógico que implementa la disyunción lógica negada cuando todas sus entradas están en “0” su salida está en 1, mientras que cuando o en BAJA.
Norma IIDC	También conocido como DCAM es un estándar de interfaz de cámaras digitales que define el protocolo para el intercambio de datos de vídeo con el protocolo IEEE 1394.
NRZ	Codificación digital en el cual no se vuelve a cero entre bits consecutivos de valor uno
OHCI	Interfaz controladora de host abiertos.
OR	Operador lógico que resulta verdadero si cualquiera de los operadores es también verídico
PC	Contador de programa.
PC	Computadora personal u ordenador personal.
PCB	Vías de circuito impreso.
PCI	Interconexión de Componentes Periféricos.
PHY	Término empleado para referirse al nivel físico.
Plasma	SoC Libre disponible en OpenCores.
Plasma-Wishbone	Unión del SoC libre Plasma y la arquitectura de interconexión para módulos IP Wishbone.
PoP	Es el apilado de diferentes placas de circuitos al ensamblarse el sistema.
PS/2	Tipo de conector mini-DIN empleado para la conexión de teclado y mouse.
PSRAM	Memoria Estática de Acceso Aleatorio.
RAM	Memoria de acceso aleatorio.
Registro de estado	Registro que se encarga de la habilitación o deshabilitación las interrupciones.
RISC	Arquitectura computacional conocida como Computador con Conjunto de Instrucciones Reducidas.
ROM	Memoria de solo lectura.
RS232	Bus universal simple de transferencia de datos.

RTOS	Sistema operativo de tiempo real.
SDRAM	Familia de memorias dinámicas de acceso aleatorio que tienen una interfaz síncrona.
SiP	Alternativa más rentable de fabricación de SoC que comprende un número determinado de chips ensamblados en uno solo.
SO	Sistema operativo.
SoC	Tecnologías de fabricación que integran todos o gran parte de los módulos componentes de una computadora o cualquier otro sistema informático o electrónico en un único circuito integrado o chip.
Spartan	Familia de dispositivos de bajo costo desarrollada por Xilinx.
SPR	Registros de propósito especial.
TOPCON	Fabricante español de equipamiento de instrumentación.
Tricore	Familia de procesadores de arquitectura RISC de la compañía Infineon.
UART	Transmisor/Receptor asincrónico universal
USB	Bus Universal en Serie.
Verilog	Lenguaje de descripción de hardware usado para modelar sistemas electrónicos.
VGA	Adaptador Gráfico de Video.
VHDL	Lenguaje de Descripción de Hardware de Alta Velocidad.
VIA-Technologies	Desarrollador taiwanés de circuitos integrados, chipsets de placas base, GPU, CPU x86 y memorias.
VLIW	Arquitectura de CPU que implementa una forma de paralelismo a nivel de instrucción.
VLSI	Integración a escala muy grande.
WindowsCE	Es un sistema operativo diseñado para sistemas embebidos.
WISHBONE	Arquitectura de interconexión para módulos IP.
Xilinx	Es la mayor empresa en investigación y desarrollo de FPGA.
XOR	Operador lógico en la cual, cuando todas sus entradas son distintas su salida está en 1.

ANEXOS

Anexo I Características de la Cámara Digital BST – HDCE.

Tabla 1. Parámetros de la cámara digital BST – HDCE

Parámetros	**HDCE-10**	**HDCE-20**
Sensor de imagen	1/3″CCD	1/2″CCD
Resolución YUV 4:2:2	1280×1024(1.3M Pixel)	1600×1200(2M Pixel)
Tamaño de pixel	4.65μm×4.65μm	4.65μm×4.65μm
Salida digital	24–bit (color)	24–bit (color)
Formato de la imagen YUV 4:2:2	1280×1024	1600×1200
	800×600	1024×768
	640×480	800×600
		640×480
Iluminación más baja	1.5 lux	2.2 lux
Exposición	Proceso de exposición Manual/Auto, Tiempo de exposición ajustable (1~500ms).	Proceso de exposición Manual/Auto, Tiempo de exposición ajustable (1~500ms).
SNR	> 45dB	> 45dB
Rango dinámico	62 dB	62 dB
Modos de Conexión	Insertar directamente en el tubo del ocular del microscopio o usar el montaje C estándar.	Insertar directamente en el tubo del ocular del microscopio o usar el montaje C estándar.
Salida de imagen	Conectar y desconectar por IEEE – 1394 para la alimentación y el control automático de energía.	Conectar y desconectar por IEEE – 1394 para la alimentación y el control automático de energía.
Dimensión	80 mm×73 mm×45mm	80 mm×73 mm×45mm
Peso	400g	400g

Anexo II Ejemplo de archivo Config.h

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
/* Aquí se pueden agregar todas las bibliotecas que se deseen. */
#include "something.h"

#define configUSE_PREEMPTION1
#define configUSE_IDLE_HOOK “0”
#define configCPU_CLOCK_HZ58982400
#define configTICK_RATE_HZ250
#define configMAX_PRIORITIES5
#define configMINIMAL_STACK_SIZE 128
#define configTOTAL_HEAP_SIZE10240
#define configMAX_TASK_NAME_LEN16
#define configUSE_TRACE_FACILITY “0”
#define configUSE_16_BIT_TICKS “0”
#define configIDLE_SHOULD_YIELD1
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet1
#define INCLUDE_vTaskDelete1
#define INCLUDE_vTaskCleanUpResources “0”
#define INCLUDE_vTaskSuspend1
#define INCLUDE_vTaskDelayUntil1
#define INCLUDE_vTaskDelay1
#endif /* FREERTOS_CONFIG_H */
```

Anexo III Ejemplo de configuración de una tarea

```
voidvTaskCode( void * pvParameters )
{
for( ;; )
{
// El código de la tarea se coloca aquí
}
}
```

Anexo IV Ejemplo de creación de una tarea

```
voidvOtherFunction( void )
{
unsignedcharucParameterToPass;
xTaskHandlexHandle;
// Creación de la tarea
xTaskCreate(vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass,
tskIDLE_PRIORITY, &xHandle );
// Eliminación de la tarea.
vTaskDelete( xHandle );
}
```

Anexo V Ejemplo empleando colas

```
//Creación de un mensaje
structAMessage
{
portCHARucMessageID;
portCHARucData[ 20 ];
} xMessage;
xQueueHandlexQueue;
// Tarea para crear una cola
voidvATask( void *pvParameters )
{
structAMessage *pxMessage;
//Creación de una cola capaz de almacenar 10 punteros a mensajes
xQueue = xQueueCreate( 10, sizeof( structAMessage * ) );
if(xQueue == “0”)
{
// En caso de ser cero es que falló la creación
}
// Envío del puntero
pxMessage = &xMessage;

xQueueSend(xQueue, ( void * ) &pxMessage, ( portTickType ) “0”);
// ... Se resetea la tarea.
}
// Tarea para recibir de la cola
voidvADifferentTask( void *pvParameters)
{
structAMessage *pxRxedMessage;
if(xQueue != “0”)
{
// Recepción del mensaje
if( xQueueReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) )
{
// pcRxedMessage es un puntero al mensaje recibido
}
}
```

Anexo VI Ejemplo usando semáforos

```
xSemaphoreHandlexSemaphore = NULL;
voidvATask( void * pvParameters)
{
// Creación del semáforo
vSemaphoreCreateBinary(xSemaphore );
if(xSemaphore != NULL )
{
if(xSemaphoreGive( xSemaphore ) != pdTRUE )
{
// Chequea el estado del semaforo
}
```

```
// Obtención dl semáforo en caso de que no esté ocupado.
if( xSemaphoreTake( xSemaphore, ( portTickType ) “0” ) )
{
//Si tenemos el semáforo podemos acceder a los recursos que este controla
// ...
// Una vez terminado el acceso cedemos el semáforo
if( xSemaphoreGive( xSemaphore ) != pdTRUE )
{
//….
}
} } }
```

Anexo VII Ejemplo de manejo de memoria

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;
vTaskSuspendAll();
{
pvReturn = malloc( xWantedSize );
}
xTaskResumeAll();
return pvReturn;
}

void vPortFree( void *pv )
{
if( pv != NULL )
{
vTaskSuspendAll();
{
free( pv );
}
xTaskResumeAll();
}
}
```

Anexo VIII Ejemplo de un programa principal (main)

```
int main( void )
{
xTaskCreate(vTask1,"Task 1", 1000, NULL, 1, NULL );

xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

//Creación de dos tareas
vTaskStartScheduler();
//Encendido del programador
for( ;; );
  }
```