Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería Mecánica Eléctrica

ELABORACIÓN DE MATERIAL DIDÁCTICO, PRÁCTICO Y TEÓRICO, PARA LA FAMILIARIZACIÓN CON LOS SISTEMAS OPERATIVOS EN TIEMPO REAL (RTOS)

Werner Oswaldo Florián Samayoa
Asesorado por el Ing. Iván René Morales Argueta

Guatemala, mayo de 2018

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ELABORACIÓN DE MATERIAL DIDÁCTICO, PRÁCTICO Y TEÓRICO, PARA LA FAMILIARIZACIÓN CON LOS SISTEMAS OPERATIVOS EN TIEMPO REAL (RTOS)

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR:

WERNER OSWALDO FLORIÁN SAMAYOA
ASESORADO POR EL ING. IVÁN RENÉ MORALES ARGUETA

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN ELECTRÓNICA

GUATEMALA, MAYO DE 2018

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Pedro Antonio Aguilar Polanco
VOCAL I	Ing. Angel Roberto Sic García
VOCAL II	Ing. Pablo Christian de León Rodríguez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Oscar Humberto Galicia Nuñez
VOCAL V	Br. Carlos Enrique Gómez Donis
SECRETARIA	Inga. Lesbia Magalí Herrera López

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Pedro Antonio Aguilar Polanco
EXAMINADOR	Ing. Byron Odilio Arrivillaga Méndez
EXAMINADOR	Ing. Guillermo Antonio Puente Romero
EXAMINADOR	Ing. Carlos Alberto Navarro Fuentes
SECRETARIA	Inga. Lesbia Magalí Herrera López

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

ELABORACIÓN DE MATERIAL DIDÁCTICO, PRÁCTICO Y TEÓRICO, PARA LA FAMILIARIZACIÓN CON LOS SISTEMAS OPERATIVOS EN TIEMPO REAL (RTOS)

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería Mecánica Eléctrica, con fecha 6 de junio 2017.

Werner Oswaldo Florián Samayoa

Guatemala 10 de enero de 2,018

Ingeniero
Julio Cesar Solares Peñate
Coordinador del Área de Electrónica
Escuela de Mecánica Eléctrica
Facultad de Ingeniería, USAC

Estimado Ingeniero Solares.

Me permito dar aprobación al trabajo de graduación titular. **"Elaboración de material didáctico, práctico y teórico, para la familiarización con los sistemas operativos en tiempo real (rtos)"**, del señor Werner Oswaldo Florián Samayoa, por considerar que cumple con los requisitos establecidos.

Por tanto, el autor de este trabajo de graduación y, yo, como su asesor, nos hacemos responsables por el contenido y conclusiones del mismo.

Sin otro particular, me es grato saludarle.

Atentamente.

Iván René Morales Argueta
Ingeniero Electrónico
Colegiado 12489

F._____
Ing. Iván Rene Morales Argueta
Colegiado 12,489
Asesor

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Guatemala, 24 de enero de 2018

Señor Director
Ing. Otto Fernando Andrino González
Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingenieria, USAC.

Señor Director:

Por este medio me permito dar aprobación al Trabajo de Graduación titulado: **ELABORACIÓN DE MATERIAL DIDÁCTICO, PRÁCTICO Y TEÓRICO, PARA LA FAMILIARIZACIÓN CON LOS SISTEMAS OPERATIVOS EN TIEMPO REAL (RTOS)**, desarrollado por el estudiante **Werner Oswaldo Florián Samayoa**, ya que considero que cumple con los requisitos establecidos.

Sin otro particular, aprovecho la oportunidad para saludarlo.

Atentamente,

ID Y ENSEÑAD A TODOS

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION ESCUELA DE INGENIERIA MECANICA ELECTRICA
FACULTAD DE INGENIERIA

Ing. Julio Cesar Solares Peñate
Coordinador de Electrónica

ERSIDAD DE SAN CARLOS
DE GUATEMALA

JLTAD DE INGENIERIA

REF. EIME 07. 2018.

El Director de la Escuela de Ingeniería Mecánica Eléctrica, después de conocer el dictamen del Asesor, con el Visto bueno del Coordinador de Área, al trabajo de Graduación del estudiante: WERNER OSWALDO FLORIÁN SAMAYOA Titulado: ELABORACIÓN DE MATERIAL DIDÁCTICO, PRÁCTICO Y TEÓRICO, PARA LA FAMILIARIZACIÓN CON LOS SISTEMAS OPERATIVOS EN TIEMPO REAL (RTOS), procede a la autorización del mismo.

Ing. Otto Fernando Andrino González

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION ESCUELA DE INGENIERIA MECANICA ELECTRICA
FACULTAD DE INGENIERIA

GUATEMALA, 1 DE MARZO 2018.

Universidad de San Carlos
De Guatemala

Facultad de Ingeniería
Decanato

Ref. DTG.D.195.2018

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería Mecánica Eléctrica al trabajo de graduación titulado: **ELABORACIÓN DE MATERIAL DIDÁCTICO, PRÁCTICO Y TEÓRICO, PARA LA FAMILIARIZACIÓN CON LOS SISTEMAS OPERATIVOS EN TIEMPO REAL (RTOS)**, presentado por el estudiante universitario: **Werner Oswaldo Florián Samayoa,** y después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, se autoriza la impresión del mismo.

IMPRÍMASE.

Ing. Pedro Antonio Aguilar Polanco
Decano

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANO
FACULTAD DE INGENIERIA

Guatemala, mayo de 2018

ACTO QUE DEDICO A:

Dios	Por guiarme y darme la fortaleza para superar las dificultades.
Mi familia	Por darme las herramientas para superarme.
Mis amigos y compañeros	Por darme su amistad.

AGRADECIMIENTOS A:

Dios	Por haber provisto a mis padres y a mí con la sabiduría para alcanzar esta meta.
Mi familia	Por siempre haberme motivado a esforzarme y dar lo mejor de mí.
Profesores y tutores	Por enseñarme con paciencia y dedicación.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES V
LISTA DE SÍMBOLOS IX
GLOSARIO XI
RESUMEN XV
OBJETIVOS XVII
INTRODUCCIÓN XIX

1. INTRODUCCIÓN A FREERTOS 1
1.1. ¿Qué es un sistema operativo en tiempo real? 1
1.1.1. BSP (*board support package*) 2
1.1.2. HAL (*hardware abstraction layer*) 3
1.1.3. Pilas de protocolos o *middleware* 3
1.1.4. OSAL (*operating system abstraction layer*) 3
1.2. ¿Qué es una tarea? 4
1.3. Gestor de tareas 6
1.4. Mecanismos de comunicación y sincronización de tareas 9
1.4.1. Semáforos 9
1.4.2. Mutex 12
1.4.3. Banderas de eventos 15
1.4.4. Temporizadores y retardos 16
1.4.5. Buzones de mensajes 16
1.4.6. Colas de mensajes 16
1.5. Práctica 1: introducción a RTOS. 17

2. GESTIÓN DE TAREAS EN RTOS Y SINCRONIZACIÓN 19
2.1. Introducción a FreeRTOS.....19
2.1.1. Estructura del código de FreeRTOS.....19
2.1.2. Configuración de FreeRTOS 22
2.1.3. Anatomía de un proyecto 23
2.2. Definición de tareas y corutinas 29
2.2.1. Creación de tareas 29
2.2.2. Estados de las tareas FreeRTOS.....34
2.2.3. Funciones específicas de FreeRTOS para tareas...35
2.3. Gestores de tarea estáticos.....38
2.3.1. Planificador RM (*rate monotonic*) 38
2.3.2. Planificación *round-robin* 39
2.3.3. Gestor de tareas de FreeRTOS.....41
2.4. Introducción a Simso.....43
2.5. Práctica 2: gestores de tareas estáticos y medición de tiempos en FreeRTOS 48
2.5.1. Simulación de gestores de tareas 48
2.5.2. Medición de periodos y tiempos de ejecución 49

3. TEMPORIZACIÓN DE TAREAS, INTERRUPCIONES Y SEMÁFOROS 53
3.1. Semáforos.....53
3.1.1. Semáforos binarios.....54
3.1.2. Semáforos con contador 56
3.2. Mutex 59
3.3. Temporización de tareas.....60
3.3.1. Configuración de temporizadores.....61
3.3.2. Creación de temporizadores 62
3.3.3. Interacción con temporizadores.....64

3.4. Planificadores dinámicos 66
3.4.1. LST *(least slack time*) menor tiempo de holgura..... 67
3.4.2. EDF *(earliest deadline first*) plazo más próximo primero 68
3.4.3. Comprobación de planificadores cíclicos..... 68
3.5. Práctica 3: marcos de planificación, temporizadores y mutex..... 70
3.5.1. Cálculo de marcos de planificación 70
3.5.2. Manejo de temporizadores y mutex en FreeRTOS 72

4. FREERTOS EN ZYNQ-7000, COLAS DE MENSAJES, MEDIOS DE COMUNICACIÓN ENTRE TAREAS 73
4.1. FreeRTOS en zynq 7000..... 73
4.1.1. Entorno de desarrollo 75
4.1.2. Crear un proyecto en vivado..... 77
4.1.3. Definir hardware en vivado 80
4.1.4. FreeRTOS en vivado 85
4.1.5. Configurar FreeRTOS en vivado 93
4.2. Hola mundo en vivado..... 95
4.3. Interactuando con GPIO’s 98
4.4. Tareas aperiódicas..... 101
4.4.1. Planificador *latest release time* (LTR)..... 102
4.4.2. Servidor retardador (*deferrable server*) 104
4.5. Práctica 4: servidor retardador y *blinking led* en FreeRTOS.. 105
4.5.1. Servidor retardador..... 105
4.5.2. *Blinking led* en FreeRTOS 106

5. BANDERAS DE EVENTOS Y SISTEMA DE COLAS DE TAREAS 109
5.1. Colas 109
5.1.1. Escribir en una cola 111
5.1.2. Leer de una cola 113
5.2. Grupos de eventos 115
5.2.1. Interactuando a través de grupos de eventos 116
5.2.2. Sincronización de tareas con grupos de eventos 118
5.3. Notificaciones 122
5.3.1. Interactuar a través de notificaciones 123
5.4. Interrupciones 126
5.4.1. Prioridades en FreeRTOS 127
5.4.2. Información específica del hardware 129
5.4.3. Interrupciones y tareas 133
5.4.4. Interrupciones en vivado 133
5.4.5. Ejemplo de interrupción en vivado 136
5.5. Práctica 5: IoT 140

CONCLUSIONES 141
RECOMENDACIONES 143
BIBLIOGRAFÍA 145

ÍNDICE DE ILUSTRACIONES

FIGURAS

1. Diagrama interno de un sistema embebido 2
2. Diagrama de estados de una tarea en un RTOS 5
3. Línea de tiempo de sistema no expropiativo 7
4. Línea de tiempo de sistema expropiativo 8
5. Toma de semáforo de una tarea 10
6. Utilización de un semáforo 10
7. Bloqueo de tarea por indisponibilidad del semáforo 11
8. Bloqueo mortal entre dos tareas 12
9. La tarea 1 toma el mutex para acceder a los recursos 13
10. La tarea 2 solicita el mutex que la tarea uno está utilizando 14
11. La tarea 1 retorna el mutex 14
12. La tarea 2 recibe el mutex y puede continuar su ejecución 14
13. Colas…….… 17
14. Estructura del código de FreeRTOS, carpeta principal 20
15. Estructura de FreeRTOS, archivos específicos de la arquitectura 21
16. Estructura del código de ejemplo de FreeRTOS en TivaWare 24
17. Función principal del archivo freertos_demo.c 25
18. Encabezados…… 26
19. Función para iniciar el gestor de tareas 27
20. Ejemplo de una tarea definida por usuario 29
21. Prototipo de la función para crear tareas 30
22. Creación de una tarea 33

23. Diagrama de estados de una tarea 34
24. Gestor de tareas tipo round-robin 40
25. Gestor de tareas de FreeRTOS 41
26. Simso, pantalla de inicio 44
27. Simso, configuración del gestor de tareas 45
28. Simso, configuración del procesador 46
29. Simso, configuración de las tareas a simular 46
30. Simso, diagrama de Gantt 47
31. Ejemplo de la creación de un semáforo 55
32. Ejemplo de la creación de un semáforo estático 56
33. Ejemplo de la creación de un semáforo con contador 58
34. Ejemplo de la creación de un mutex 60
35. Prototipo de la función para crear temporizadores 62
36. Ejemplo de la implementación de un temporizador 64
37. Diagrama Gantt de un sistema cíclico 69
38. Diagrama interno del zynq 7000 74
39. Diagrama de flujo de la creación de hardware en vivado 76
40. Ventana de bienvenida de vivado 77
41. Selección de tipo de proyecto 78
42. Selección de lenguaje para el proyecto 78
43. Selección de plataforma de desarrollo 79
44. Proyecto abierto para edición en vivado 80
45. Menú de herramientas y selección de la opción para creación de IP ... 81
46. Menú para la creación de IP 82
47. Ventana para creación de interfaces 83
48. Ventana para edición de IP 85
49. IP del módulo de procesamiento para zybo 86
50. Ventana para modificación del módulo de procesamiento 87
51. Opción de asistencia de diseño 87

52. Asistencias de diseño 88
53. Diagrama de diseño 88
54. Creación de HDL *wrapper* 89
55. Exportar el hardware 89
56. Ventana de inicio del kit de desarrollo de software de vivado 90
57. Creación de un proyecto 91
58. Configuración de un nuevo proyecto 92
59. Jerarquía del proyecto creado 93
60. Archivo system.mss 94
61. Parámetros de configuración de FreeRTOS y de los controladores 95
62. Selección de tipo de proyecto para un nuevo proyecto de aplicación 96
63. Jerarquía de archivos en un proyecto 97
64. La carpeta *include* del BSP contiene todas las librerías de controladores 98
65. Creación de variables tipo XGpio 99
66. Identificadores de dispositivos en *xparameters.h* 100
67. Ejemplo de planificador LTR 103
68. Ejemplo de creación de una cola 110
69. Ejemplo de escritura en una cola 112
70. Ejemplo del xQueueRecieve 114
71. Ejemplo de xQueuePeek 115
72. Creación de un grupo de banderas 116
73. Ejemplo de uso de grupos de eventos 117
74. Bloqueo de función utilizando xEventGroupWaitBits 119
75. Sincronización de tres tareas con xEventGroupSync 121
76. Diagrama interno del ARM Cortex A9 con el GIC señalado 131
77. Configuración del sistema de procesamiento de zynq7000 134
78. Ejemplo de conexión de una interrupción PL-PS 134

79. *System.mss* con los periféricos para interrupciones 135
80. Librerías con los controladores para los periféricos GIC y SUC 136
81. Implementación del GIC.....136
82. Redefinición de registros para mayor legibilidad.....137
83. Configuración del GIC y el GPIO 138
84. Rutina de interrupción genérica 139

TABLAS

I. Tareas para simulación de planificador RM 49
II. Planificador cíclico 1 70
III. Planificador cíclico 2 71
IV. Planificador cíclico 3 71
V. Planificador LTR 103
VI. Sondeo contra interrupción 126
VII. Niveles de prioridades de FreeRTOS 128
VIII. Interrupciones PPI de cada núcleo del ARM Cortex A9.....132

LISTA DE SÍMBOLOS

Símbolo	Significado
!	Operación lógica NOT
%	Porcentaje
*****	Puntero
\n	Salto de línea
=	Comparación
CPU	Unidad central de procesamiento
D	Plazo de vencimiento
E	Tiempo de ejecución
HZ	Hertz
NULL	Nada
P	Periodo
ram	Memoria de acceso aleatorio
RTOS	Sistema operativo en tiempo real
U	Utilización del sistema

WCET	Peor caso de tiempo de ejecución

GLOSARIO

Bandera	Bit que se utiliza para determinar un cambio de estado. Se relaciona con un evento y se escribe 1 o 0 dependiendo de la lógica que el programador decida utilizar.
Bare metal	Programar directamente en la lógica del dispositivo sin utilizar un sistema operativo.
Code composer studio	IDE gratuito provisto por Texas Instruments para programar sus microcontroladores.
FPGA	Arreglo de compuertas programables, un microchip capaz de modificar su arquitectura interna de acuerdo a un código definido por el usuario.
Heap	Estructura de datos en forma de árbol.
Idle	Estado de una tarea en el cual no realiza trabajo de computación o escritura, ocio.
Interprocess communication	Proceso de comunicación entre las tareas que permite el intercambio seguro de información.
ISR	Rutina de interrupción del sistema, la porción de

	código que el sistema ejecuta en caso de una interrupción.
Kernel	Núcleo, la parte fundamental de un sistema operativo es la que se encarga de los accesos a hardware.
Librería	Archivo que contiene funciones, objetos y definiciones de uso frecuente. Permiten reducir el tamaño del código principal distribuyéndolo en varios archivos.
Microcontrolador	Circuito integrado programable capaz de ejecutar tareas. Contiene una unidad de procesamiento central, memoria y periféricos de entrada y salida.
Sistema embebido	Sistema de computación diseñado para una tarea específica. Los sistemas embebidos usualmente tienen todos sus componentes en una sola placa, de allí su nombre. Los programas de sistemas embebidos se enfrentan normalmente a tareas de procesamiento en tiempo real.
***Visual studio* 2017**	IDE de Microsoft para proyectos en C, C++, .NET, entre otros.

WCET	Peor caso de tiempo de ejecución.
WIN32-MSVC	Archivos para Microsoft Visual en Windows de 32 bits, escrito en C.

RESUMEN

En el siguiente trabajo de graduación se desglosan las funciones, herramientas y utilidades que un sistema en tiempo real ofrece. Específicamente se estudia FreeRTOS y su desarrollo para aplicaciones en plataformas de desarrollo. La primera mitad del manual está diseñada para implementarse en la plataforma de desarrollo de Texas Instruments, tiva C. La segunda parte se enfoca en los beneficios que un sistema operativo en tiempo real aporta a aplicaciones desarrolladas en sistemas en chip. Se describe el proceso completo de desarrollo de una aplicación para la plataforma de desarrollo zybo, iniciando con la descripción del hardware y finalizando con la implementación de una aplicación en FreeRTOS.

En el primer capítulo se detallan las ventajas de un sistema operativo en tiempo real. Se describen las partes esenciales para su funcionamiento. Se introducen los conceptos necesarios para trabajar con gestores de tareas y se describen las estructuras que brinda FreeRTOS al programador.

En el segundo capítulo se estudia más a fondo el sistema operativo. Se presentan las herramientas para definir trabajos y se detalla la organización del código en FreeRTOS. Se presentan y explican las opciones de configuración de FreeRTOS. Las primera nociones de gestores de tareas se presentan en este capítulo, así como el primer ejercicio práctico.

En el tercer capítulo se presentan las estructuras de control de FreeRTOS, temporizadores y semáforos. Se indaga más profundamente en los gestores de tareas para poder validarlos matemáticamente.

En el cuarto capítulo se introduce al entorno de desarrollo de vivado para su uso con la plataforma zybo. Se describe el proceso de creación de hardware y la implementación de FreeRTOS para el desarrollo de software. Se presentan los servidores retardadores como herramientas para resolver problemas de tiempo ante tareas aperiódicas y esporádicas.

En el quinto capítulo se desarrolla el tema de las interrupciones para ambas plataformas. Para la tiva C solo se menciona la forma en la que se deben definir las prioridades dentro del ARM Cortex 4F. En el caso de la zybo se describe el proceso de habilitación de interrupciones desde hardware definido. Al ser externo al procesador el proceso es distinto y requiere mayor detalle que en el caso de la tiva C.

OBJETIVOS

General

Elaborar un manual que desarrolle el contenido necesario para la familiarización con los sistemas operativos en tiempo real y facilite su utilización para solucionar problemas reales.

Específicos

1. Diseñar ejercicios prácticos que exijan la aplicación de los conocimientos adquiridos en las plataformas de desarrollo tiva C y zybo.

2. Diseñar un proyecto que englobe todos los temas a tratar para demostrar una aplicación real.

3. Facilitar la adquisición de conocimiento sobre el desarrollo de sistemas en tiempo real para computación física.

INTRODUCCIÓN

Los sistemas operativos en tiempo real son tecnología muy importante. Estos presentan una solución más realista, escalable y de más fácil despliegue para un sistema complejo. Esto se debe que a diferencia de los sistemas operativos de uso específicos, como los vistos en la carrera de ingeniería electrónica, permiten un nivel de abstracción ligeramente más elevado y un uso de periféricos más sencillo, pero sin llegar a un sistema operativo de uso general.

Dada la importancia que tienen en la industria e investigación los sistemas operativos de tiempo real, es necesario contar con material de apoyo que permite una introducción sencilla y eficaz. Lo más importante de una tecnología no es solo su alcance, sino la facilidad para aplicarla, comprenderla, actualizarla a los requerimientos que surgen y su portabilidad, ya que son sistemas de bajo consumo.

La facilidad para adoptar una nueva tecnología es solo posible a través de los manuales creados por los fabricantes. La utilidad de un material didáctico para nivel estudiantil permite evaluar los temas con mayor detalle, pero evitando la complejidad de un texto completamente técnico. Esto se logra utilizando ejercicios prácticos para reforzar los conocimientos y de esta forma llevar un manejo de la complejidad ordenado evitando agobiar al lector con situaciones innecesarias en el momento.

1. INTRODUCCIÓN A FREERTOS

1.1. ¿Qué es un sistema operativo en tiempo real?

Los sistemas embebidos usualmente están fuera de la vista de usuario. Lo más probable es que solamente sean notorios cuando no funcionan correctamente. Actualmente, los sistemas embebidos forman parte del 98 % de los sistemas de computación en el mundo, y muchos utilizan sistemas operativos en tiempo real.

Un sistema operativo en tiempo real es aquel que su funcionamiento correcto no solo depende de los resultados de las computaciones, pero también depende del momento en el que esos resultados son generados.

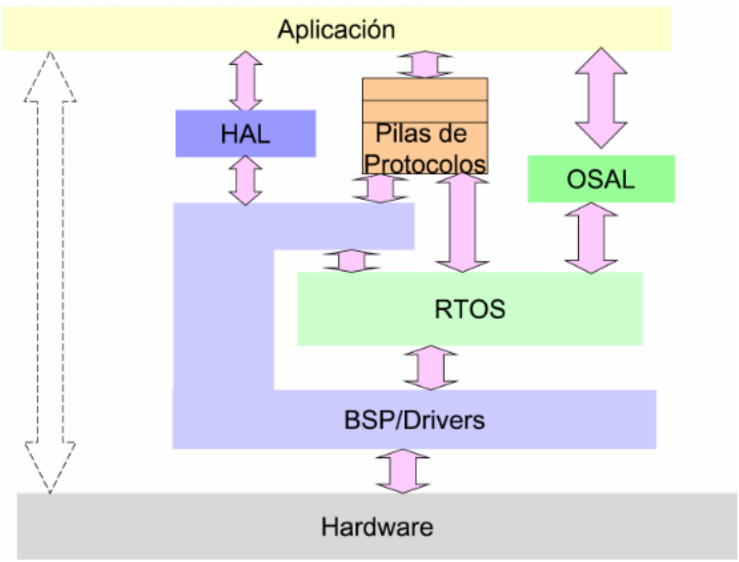
Antes de iniciar debemos tomar en cuenta las varias formas de programación que se pueden emplear en sistemas embebidos y reconocer que una de las soluciones es un sistema operativo en tiempo real (RTOS de este punto en adelante). Es necesario entender que el RTOS por sí solo no describe una funcionalidad completa en un sistema embebido, más bien es parte del *firmware*.

Uno de los principales objetivos de un RTOS es abstraer la funcionalidad de cada tarea para que se comporten como entidades independientes. Las tareas, aunque aisladas, son capaces de comunicarse entre ellas por varios mecanismos que provee el RTOS. Estos mecanismos se les llama comunicación entre procesos, o IPC (*inter process communication*).

Las ventajas de un RTOS son apreciadas en sistemas complejos que requieren de muchas tareas concurrentes, las cuales tiene que realizar el microcontrolador.

Los principales componentes en el software de un sistema embebido son los siguientes:

Figura 1. **Diagrama interno de un sistema embebido**



Fuente: Universidad de Málaga, curso en línea de la maestría de sistemas empotrados. (2016). Micro *kernels*. Acceso restringido a estudiantes del curso.

1.1.1. BSP (board support package)

Es un conjunto de módulos que proveen las herramientas para la configuración y el uso de los periféricos del microcontrolador. Provee la posibilidad de hacer uso de los periféricos a través de una interfaz de alto nivel

de abstracción. Debido a que estas librerías son creadas por el fabricante puede complicarse su uso en *hardware* distintos.

1.1.2. HAL (hardware abstraction layer)

Esta capa de abstracción de hardware provee herramientas a través de librerías que permiten tratar el hardware de forma genérica. Resuelve el problema de las particularidades de la BSP ocultándose detrás de una interfaz de programación genérica. Esto permite independizar el software del hardware y brindar mayor portabilidad.

1.1.3. Pilas de protocolos o *middleware*

Son librerías que contienen todos los protocolos de comunicación necesarios para la interacción con otros sistemas. El *middleware* no solo comprende el código, sino también la implementación, debido a que debe de comunicarse con varios sistemas requiere un proceso independiente.

1.1.4. OSAL (operating system abstraction layer)

Esta capa de abstracción permite a la aplicación correr sobre cualquier RTOS. El OSAL se encarga de proveer un entorno de programación genérico para que la aplicación pueda ser utilizado en varios RTOS sin modificaciones.

Un RTOS no solo brinda la ventaja de tiempos de ejecución estrictos para múltiples tareas, sino que también permite simplificar el desarrollo de software. Otras ventajas son:

- Aplicaciones simplificadas a conjuntos de tareas independientes capaces de interactuar entre sí.
- Permite modularidad, lo cual facilita la introducción de nuevas funcionalidades.
- Ejecución en hilos permite aislar fallas para evitar que el sistema completo se detenga ante un error.

Existen dos tipos de sistemas operativos en tiempo real crítico y sistemas en tiempo real suave. La diferencia es que en los sistemas operativos en tiempo real crítico las tareas deben ejecutarse en su plazo todas las veces para que el resultado de las mismas sea útil. En los sistemas de tiempo real suaves en ocasiones las tareas pueden no ser terminadas en el plazo e inclusive el resultado puede ser útil.

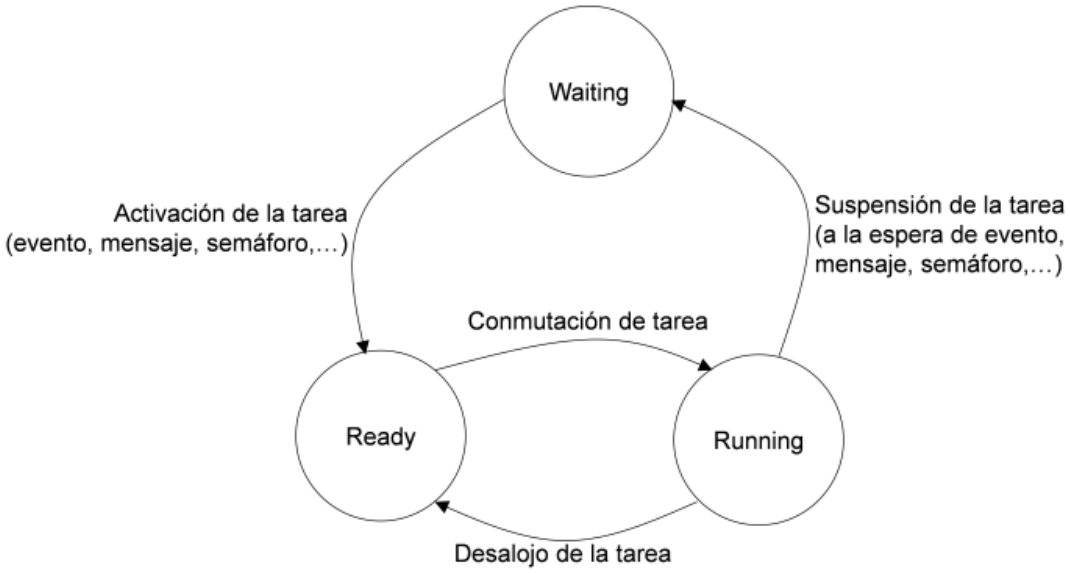
1.2. ¿Qué es una tarea?

Consiste en un programa que percibe el procesador como exclusivo. Esto se debe a que el RTOS se encarga de crear el contexto exclusivo para la tarea. Por contexto entiéndase los registros de CPU y la pila. Así que en caso de que haya varias tareas, el RTOS almacena el contexto de la tarea antes de cambiar a otra. De esta forma todas las tareas al ejecutarse tienen su propio contexto. Siempre es necesario que las tareas puedan interactuar entre ellas, de eso se encarga el *kernel*.

Las tareas en RTOS son procesos con bucles infinitos que permiten que la tarea espere indeterminadamente en el sistema hasta que se cumplan las condiciones necesarias para continuar con su ejecución. El RTOS permite que se ejecute otra tarea mientras las otras esperan.

Las tareas pueden tomar diversos estados ejecutando (*running*), bloqueada (*waiting*) o disponible (*ready*). Algunas tareas pueden estar durmientes (*dormant*), esto significa que la tarea está definida en memoria, pero aún no está activada para ser tomada en cuenta por el *kernel*.

Figura 2. **Diagrama de estados de una tarea en un RTOS**



Fuente: Universidad de Málaga, curso en línea de la maestría de sistemas empotrados. (2016). Micro *kernels*. Acceso restringido a estudiantes del curso.

El estado de cada una de las tareas es controlado en el *kernel* mediante el bloque de control de tareas (TCB, *task control block*). Toda la carga de trabajo en un procesador es cuantificada en porciones más pequeños y se llaman tareas. Existen varios tipos de tareas por sus propiedades temporales. Las tareas periódicas se repiten en espacios de tiempo uniformes. Las tareas esporádicas resultan de condiciones externas y no obedecen un periodo.

Las características generales de una tarea son periodo (P), tiempo de ejecución (e), *deadline* (D, tiempo límite). El tiempo de ejecución se toma a partir del peor caso posible de ejecución (WCET). El tiempo límite o *deadline,* es tiempo en el cual la tarea debe terminarse, este puede ser absoluto o relativo.

1.3. Gestor de tareas

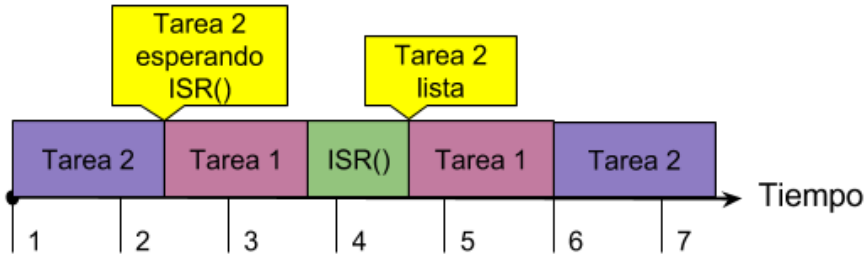
La necesidad de un sistema operativo que gestione las tareas implica que se utilizarán más de una tarea. En un sistema con un solo CPU y varias tareas, solo una tarea puede acceder al procesador una a la vez. El gestor de tareas o planificador es el encargado de determinar qué tarea se ejecuta en qué instante del tiempo. Cuando el gestor de tareas decide cambiar la tarea en ejecución, el *kernel* se encarga de recuperar el contexto de la tarea actual y guardarlo. Para luego cargar el contexto de la siguiente tarea.

El gestor de tareas al solicitar los cambios de contexto introduce carga sobre el sistema (*overhead*). Esta carga no produce resultados computacionales útiles, por lo tanto reduce la utilización real del procesador. De acuerdo a la cantidad de veces que es necesario cambiar contexto se incrementa la carga al sistema. Un gestor de tareas ideal es aquel que es capaz de organizar los tiempos de las tareas de tal forma que todas se ejecuten antes de su tiempo límite, pero también es capaz de mantener la utilización del procesador abajo del 100 %.

En un sistema correctamente diseñado, el *kernel* consume entre 2 % y 5 %. Para esto se diseñan distintos tipos de *kernels.* Los cuales se diferencian en la forma en la que seleccionan las tareas para su ejecución.

Existen *kernels* expropiativos (*preemptive*) y no expropiativos (*non-preemtive*). Los *kernels* expropiativos son aquellos que al recibir una tarea de mayor prioridad empujan la tarea en ejecución en favor de la nueva tarea.

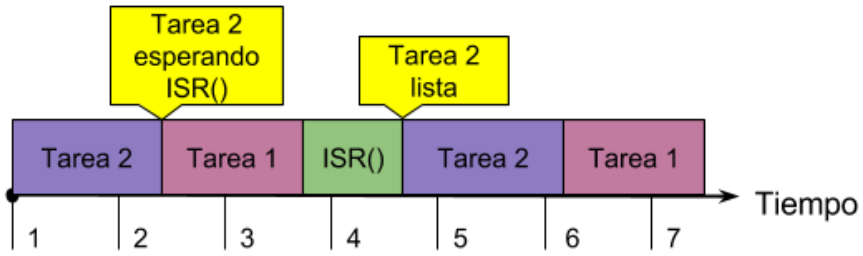
Figura 3. **Línea de tiempo de sistema no expropiativo**



Fuente: elaboración propia, empleando Visio 2015.

Los *kernels* no expropiativos permiten a las tareas completar su ejecución antes de cambiar a otra, a menos que la tarea en el procesador pase al estado de espera (*waiting*) en este caso se hará un cambio de contexto a una tarea que esté lista (*ready*). Este tipo de funcionamiento se le llama multitarea cooperativa, deben estar dispuestas a ser desalojadas del procesador. En este caso las tareas son las que determinan cuando se ejecuta otra, ya que el procesador espera el cambio de estado para realizar el cambio de contexto.

Figura 4. **Línea de tiempo de sistema expropiativo**



Fuente: elaboración propia, empleando Visio 2015.

En un sistema expropiativo el procesador siempre cambia a la tarea de mayor prioridad sin importar el estado de la tarea actual. La tarea de mayor prioridad pasa de un estado de espera (*waiting*) a uno de lista (*ready*) cuando ocurre el evento que la tarea esperaba, ya sea por un ISR o por la tarea en ejecución. Esto fuerza a tareas en ejecución a pasar a espera. Este tipo de *kernel* asegura un tiempo respuesta bajo para las tareas de mayor prioridad. Es importante tomar en cuenta que no todas las tareas pueden retirarse del procesador en todo momento, ya que al utilizar este tipo de *kernel* hay que tomar en cuenta que la expropiación de recursos puede ocurrir en cualquier momento y lugar del programa.

Los problemas pueden darse cuando la tarea accede a una función externa a la cual otras tareas también acceden. Si la función utiliza variables globales, al cambiar de contexto habrá problemas, pero si la función solo utiliza variables locales entonces puede utilizarse entre distintas tareas sin conflictos. Este tipo de funciones se les llaman reentrantes. Las variables locales se almacenan en una pila y no en un espacio de memoria fija, lo cual las hace parte del contexto de la tarea.

Debido a que el *kernel* responde de acuerdo a la prioridad de las tareas pueden tener prioridades fijas o dinámicas. Debido a esto existen varios gestores de tareas los cuales determinan la prioridad de la tarea de acuerdo a parámetros distintos. También es importante mencionar que debido a que lo sistemas embebidos han tomado la dirección hacia sistemas multiprocesador, los gestores de tareas también han evolucionado para permitir formas de garantizar la ejecución y asignación correcta de tareas en estos sistemas más complejos.

1.4. Mecanismos de comunicación y sincronización de tareas

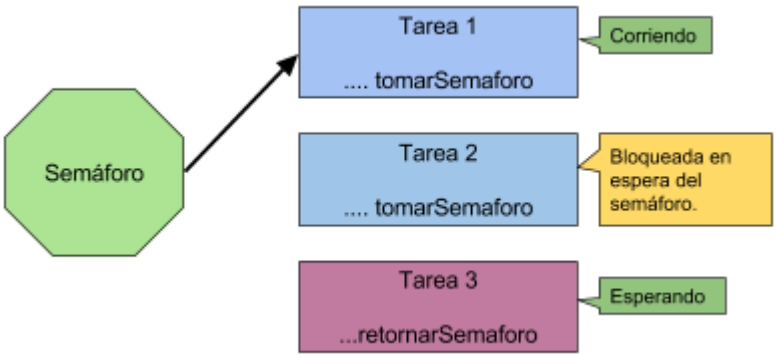
Las tareas se ejecutan en forma independiente, pero siempre es necesario que puedan comunicarse entre ellas. Los IPC permiten que las tareas compartan información entre ellas. El kernel es el encargado de administrar estos recursos entre tareas que permiten su comunicación y sincronización. A continuación, se presentan algunos de los mecanismos en forma superficial y breve.

1.4.1. Semáforos

Son un método de sincronización de tareas que permite controlar el acceso a recursos compartidos y señalización de eventos. Son llamados también semáforos binarios, esto debido a que solo tienen dos estados. Sus estados son modificados por código en las tareas, esto permite establecer un orden estricto al acceso de recursos y a la ejecución de tareas.

El funcionamiento de los semáforos es sencillo, una tarea puede adquirir el semáforo siempre que este esté libre. Una vez adquirido el semáforo, cualquier tarea puede devolverlo para permitir el acceso nuevamente.

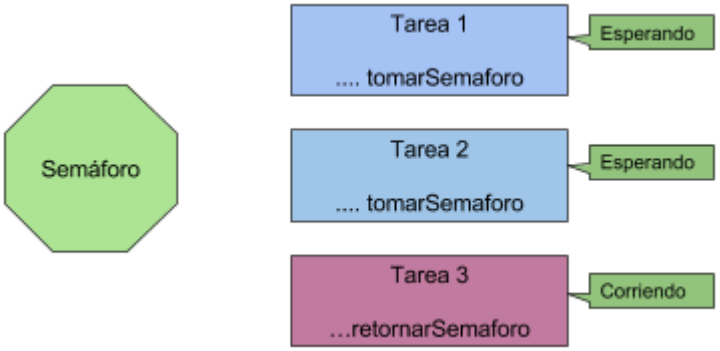
Figura 5. **Toma de semáforo de una tarea**



Fuente: elaboración propia, empleando Visio 2015.

La tarea 1 solicita el semáforo al estar este libre se le concede. Todas las demás tareas que soliciten el semáforo serán bloqueadas hasta que el semáforo sea entregado. La tarea 2 continúa bloqueada hasta que se ejecute la tarea 3 la cual es la encargada de retornar el semáforo.

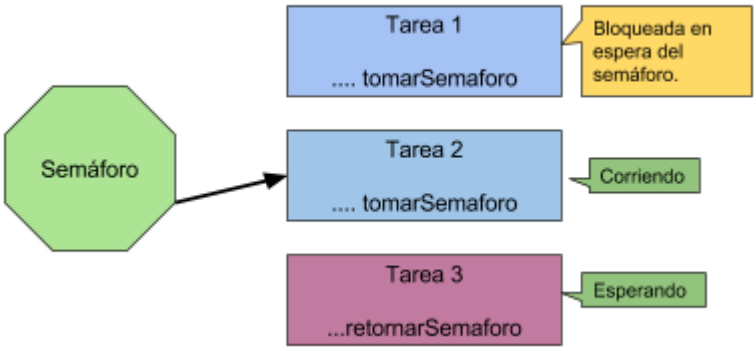
Figura 6. **Utilización de un semáforo**



Fuente: elaboración propia, empleando Visio 2015.

La tarea 3 retorna el semáforo y permite que el semáforo sea apropiado por otra tarea. Esto desbloquea la tarea 2.

Figura 7. **Bloqueo de tarea por indisponibilidad del semáforo**



Fuente: elaboración propia, empleando Visio 2015.

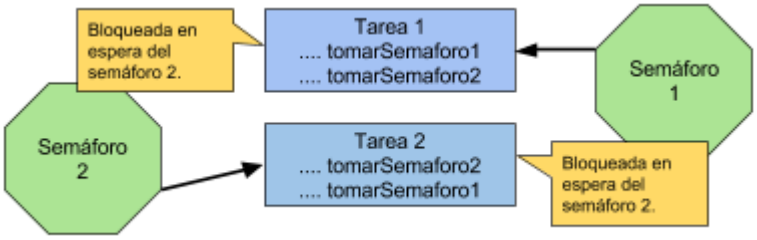
La tarea 2 obtiene el semáforo y la tarea 1 se bloquea en espera del semáforo.

Los semáforos pueden ser entregados por tareas o por rutinas de interrupción del sistema (ISR). El semáforo actúa como sincronizador de tareas cuando se quiere se ejecuten de forma consecutiva dos tareas específicas. Pero, también actúa como control de recursos.

Como una analogía, el semáforo se comporta como una llave. Ya que cuando una tarea desea acceder a un recurso toma el semáforo, la llave. Cualquier otra tarea que quiera acceder no podrá debido a que no posee la llave. Cuando la llave es retornada a su lugar, solo entonces otra tarea puede tomarla para acceder al recurso.

Por la forma en la que operan los semáforos es posible llegar a un bloqueo mortal (*deadlock*). Esto puede suceder cuando dos tareas quedan bloqueadas esperando recursos, que por el mismo hecho de estar bloqueadas no pueden entregar.

Figura 8. **Bloqueo mortal entre dos tareas**



Fuente: elaboración propia, empleando Visio 2015.

Es muy importante que el programador analice la lógica del programa para evitar este tipo de situaciones, ya que bloquearon las tareas por tiempo indefinido.

1.4.2. Mutex

Los mutex son también conocidos como semáforos binarios mutuamente excluyentes. Estos semáforos incluyen la mecánica de herencia de prioridad. Los semáforos binarios son mejores para sincronización de tareas. Los mutex son mejores para tareas de mutua exclusión (de allí su nombre *mutual exclusion*).

El mutex actúa como una llave de acceso para recursos. Cuando una tarea solicita los recursos, esta toma la llave. Cuando termina de utilizar los recursos la devuelve para que otra tarea pueda acceder a los recursos.

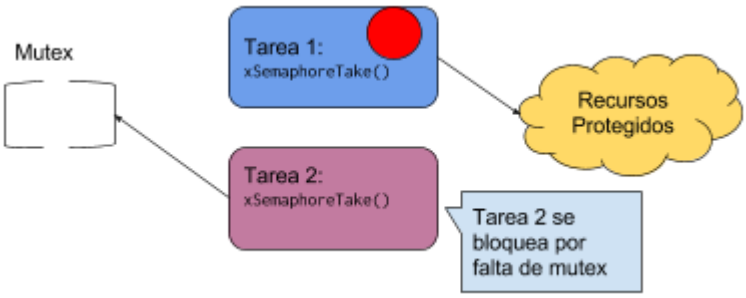
El mecanismo de herencia de prioridad significa que cuando una tarea de prioridad baja toma el mutex y hay una tarea de mayor prioridad esperando, la prioridad de la primera tarea se aumenta a la de la tarea bloqueada. Este mecanismo se asegura de mantener la tarea de mayor prioridad bloqueada por la menor cantidad de tiempo posible. Este cambio de prioridades se le llama inversión de prioridad y debe ser evitado al máximo.

Los mutex no deben ser utilizados con interrupciones, debido a que la herencia de prioridad sólo tiene sentido entre tareas. También porque no puede bloquearse una interrupción.

Figura 9. **La tarea 1 toma el mutex para acceder a los recursos**

Fuente: elaboración propia, empleando Visio 2015.

Figura 10. **La tarea 2 solicita el mutex que la tarea uno está utilizando**

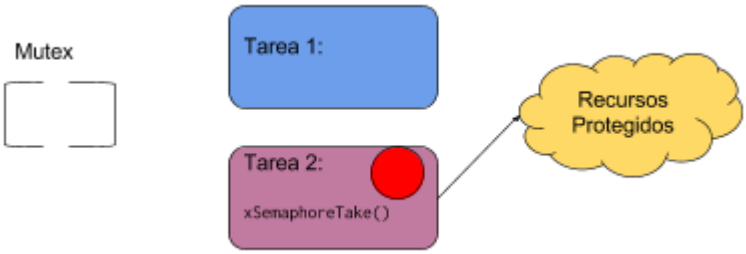


Fuente: elaboración propia, empleando Visio 2015.

Figura 11. **La tarea 1 retorna el mutex**

Fuente: elaboración propia, empleando Visio 2015.

Figura 12. **La tarea 2 recibe el mutex y puede continuar su ejecución**



Fuente: elaboración propia, empleando Visio 2015.

1.4.3. Banderas de eventos

Se utilizan para indicar si un evento ocurrió. Son útiles cuando una tarea requiere de varios eventos para su ejecución, por ejemplo:

- Se define una bandera para determinar si hay un mensaje para procesar.
- Una bandera que determine si la aplicación tiene mensajes en cola para procesar.
- Una bandera que indique que es momento de enviar una señal de vida en un en enlace de red.

Cuando son varias banderas de eventos estas pueden agruparse en grupos de eventos. Estos grupos de eventos consisten de bloques de 8, 16 o 32 bits. Cada una de las banderas en el grupo de eventos puede ser activado o borrado individualmente por tareas o por interrupciones. Las banderas pueden utilizarse junto con lógica combinacional para crear condiciones entre tareas para activación conjuntiva (AND) o disyuntiva (OR).

Es necesario estar conscientes de las complicaciones que pueden surgir debidas a una mala implementación. Principalmente pueden darse dos casos:

Evitar las condiciones de carrera en la aplicación. La condición de carrera se crea cuando en la aplicación cuando:

- No está claro quién está a cargo de borrar la bandera
- No está claro cuando se borra una bandera.
- No está claro quien modificó la bandera.
- Evitar los indeterminismos:

La conducta indeterminista surge cuando en un grupo de eventos no se conoce la cantidad de tareas que serán bloqueadas, por lo tanto, se desconoce cuántas condiciones deben cumplirse o cuantas tareas tienen que desbloquearse cuando una bandera cambia.

1.4.4. Temporizadores y retardos

Los RTOS proveen herramientas para temporizar tareas, esto ayuda la sincronización cuando no es necesario que se ejecuten inmediatamente sino luego de un periodo de tiempo determinado. Los temporizadores activan interrupciones en el sistema, luego de contar una cantidad determinada de *ticks* del sistema. Estas interrupciones permiten activar otras tareas con mayor prioridad y modificar el orden de ejecución.

Estos temporizadores están implementados en software, lo que permite mayor flexibilidad. Los temporizadores están expuesto a pequeñas variaciones temporales llamadas *jitter*.

1.4.5. Buzones de mensajes

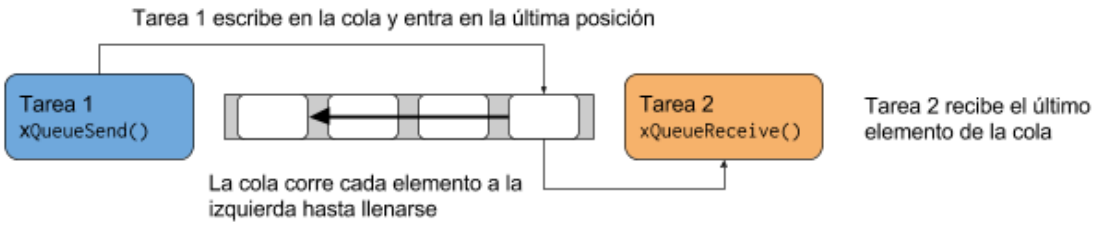
Los buzones son un mecanismo sencillo de comunicación entre tareas. Permiten enviar un mensaje, que usualmente es un puntero que apunta a la zona de memoria donde están los datos de interés. Una tarea que espera un mensaje para a estado de espera hasta que el mensaje llega.

1.4.6. Colas de mensajes

Una cola de mensajes extiende el concepto del buzón. Esta estructura permite almacenar una cantidad predeterminada de mensajes de tamaño

específico. La cola crea su propio espacio en memoria y a menos de que un mensaje sea recuperado de la cola este permanece inmutable, ya que es una copia del mensaje original.

Figura 13. **Colas**



Fuente: elaboración propia, empleando Visio 2015.

1.5. Práctica 1: introducción a RTOS.

Antes de iniciar con el contenido práctico es necesario crear nuestro laboratorio. Para esto se utiliza la plataforma de desarrollo tiva C y algunos simuladores.

- Descargar e instalar code composer studio: http://www.ti.com/tool/CCSTUDIO
- Descargar FreeRTOS para tiva C: http://www.ti.com/tool/SW-TM4C
- Descargar e instalar el simulador de gestores de tareas: http://projects.laas.fr/Simso/

Opcional, en caso de no poseer el hardware se puede implementar el código en un simulador de Freertos:

- Descargar e instalar Visual studio 2017: https://www.visualstudio.com/es/downloads/
- Descargar el simulador de FreeRTOS: https://sourceforge.net/projects/freertos/files/latest/

Para utilizar el simulador con visual studio, es necesario utilizar el proyecto de la carpeta WIN32-MSVC.

2. GESTIÓN DE TAREAS EN RTOS Y SINCRONIZACIÓN

2.1. Introducción a FreeRTOS

El sistema operativo en tiempo real a utilizar es FreeRTOS. Este presenta varias ventajas, entre ellas es software libre. Lo cual ha permitido que varios fabricantes hayan hecho las modificaciones necesarias para que sea compatible con su sistema. Actualmente soporta numerosos controladores. La documentación del mismo se encuentra en la página web.

Algunas otras ventajas son:

- FreeRTOS no realiza operaciones determinísticas. Como por ejemplo recorrer una lista enlazada dentro de una interrupción o una tarea crítica.
- Los *timers* no consumen tiempo en el CPU a menos que la tarea de servicio sea activada.
- Las notificaciones entre tareas se hacen de forma directa y su consumo de ram es prácticamente nulo.

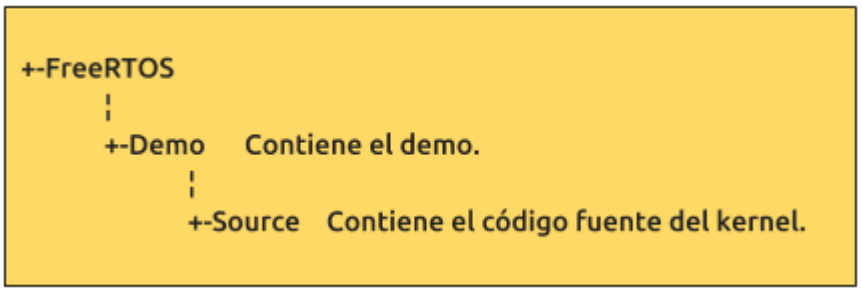
Como introducción se verán los aspectos básicos para construir un proyecto y se familiarizarán con la estructura de los mismos.

2.1.1. Estructura del código de FreeRTOS

Cada implementación de FreeRTOS viene con una aplicación demos preconfigurada. Esta contiene todos los archivos necesarios para construir el

sistema operativo con las tareas de usuario. La estructura de FreeRTOS es la siguiente:

Figura 14. **Estructura del código de FreeRTOS, carpeta principal**

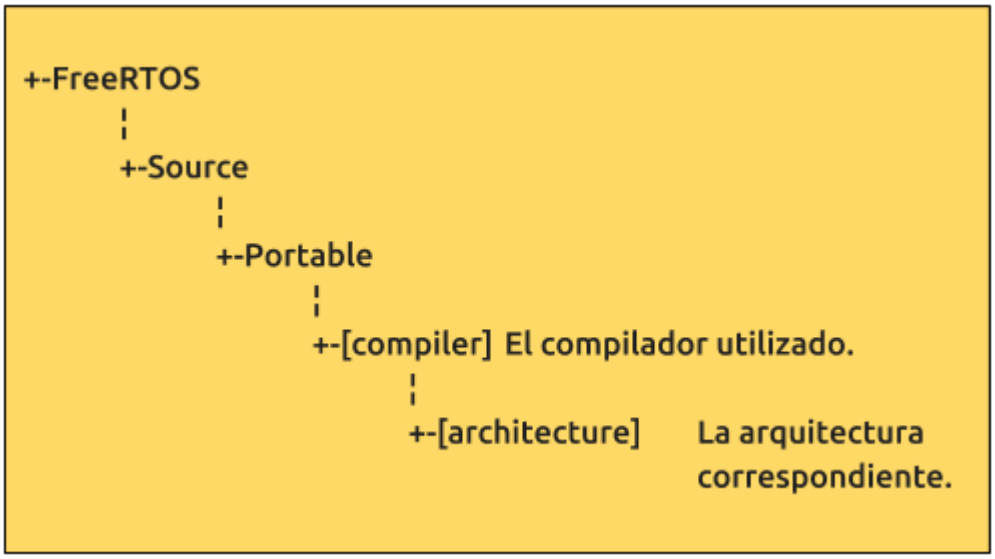


Fuente: *FreeRTOS - Free RTOS Source Code.* http://www.freertos.org/a00017.html. Consulta: 14 de noviembre de 2017.

El núcleo de FreeRTOS está distribuido en tres archivos, *task.c*, *queue.c*, y *list.c.* En el mismo directorio hay dos archivos más, *timers.c* y *croutine.c*. Estos dos implementan los *timers* y las corutinas.

Cada adaptación de FreeRTOS requiere un código específico, que está dentro de la capa de portabilidad de RTOS y está localizado en el siguiente directorio.

Figura 15. **Estructura de FreeRTOS, archivos específicos de la arquitectura**



Fuente: *FreeRTOS - Free RTOS Source Code.* http://www.freertos.org/a00017.html. Consulta: 14 de noviembre de 2017.

Debido a que FreeRTOS necesita ram y ya que esta depende de la arquitectura, está localizada en el directorio *portable*. La memoria está definida en los archivos *heap_x.c* donde *x* determina el nivel de la memoria. Siendo estos:

- heap_1: la implementación más sencilla no permite que la memoria se libere.
- heap_2 : permite que la memoria se libere pero no es coalescente con bloques adyacentes.
- heap_3: envuelve el malloc() y free() para seguridad de los hilos.
- heap_4: es coalescente con bloques adyacentes para evitar fragmentación. Permite el uso de direcciones absolutas.

- heap_5: escala heap_4 para que sea extensible en varios bloques no adyacentes de memoria.

2.1.2. Configuración de FreeRTOS

La configuración es gobernada por el archivo *FreeRTOSConfig.h.* Vale la pena mencionar solo algunas de las entradas ya que varias son bastante intuitivas. Todas las configuraciones se activan con 1 y se desactivan colocándolas en 0.

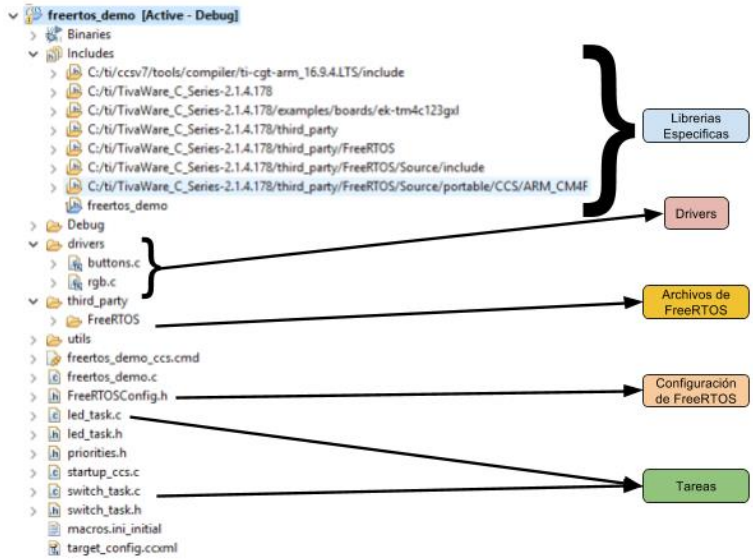
- configUSE_PREEMPTION: esta entrada es activada cuando se desea que el *kernel* sea expropiativo.
- configUSE_IDLE_HOOK: permite que una función se ejecute cada ciclo que una tarea esté en espera. Es ideal para pruebas, pero no es recomendable mantenerlo activado para implementación.
- configUSE_TICK_HOOK: si se configura en 1, una función definida por el usuario se ejecutará cada vez que en el *kernel* ocurra una interrupción por *ticks* o pulsos*.*
- configCPU_CLOCK_HZ: corresponde al reloj interno del microcontrolador.
- configTICK_RATE_HZ: determina la frecuencia de los conteos para interrupciones. Al ser mayor permite mayor resolución temporal para las interrupciones, pero fuerza al procesador a procesar más frecuentemente las solicitudes de conteo de interrupción. Es necesario analizar la aplicación para determinar la frecuencia óptima.
- configMAX_PRIORITIES: permite determinar los niveles de prioridades para las tareas. Cada nivel crea una nueva lista en memoria, por lo tanto, en plataformas con memoria reducida es recomendable mantenerlo al mínimo necesario.

- configIDLE_SHOULD_YIELD: controla la forma en la que las tareas en espera son expropiadas por tareas de mayor prioridad. Si está activado la tarea en espera es expropiada inmediatamente. Pero, la tarea de mayor prioridad solo se le asignará el periodo de tiempo que le resta a la de menor prioridad.
- configMINIMAL_STACK_SIZE: determina el tamaño de la pila utilizada por tareas en espera.
- configTOTAL_HEAP_SIZE: determina el tamaño disponible en la ram para las memorias *heap* de FreeRTOS. Estas serán utilizadas almacenar todas las estructuras y funciones del sistema operativo.

2.1.3. Anatomía de un proyecto

Al instalar TivaWare se tiene la opción de importar varios ejemplos. Para esta sección en específico se utilizará el ejemplo *freertos_demo.*

Figura 16. **Estructura del código de ejemplo de FreeRTOS en TivaWare**



Fuente: elaboración propia.

En este proyecto no se sigue la convención, el archivo principal en este caso no se llama *main.c* esto se debe a que forma parte de un directorio con varios ejemplos y así es más fácil diferenciarlo. En este ejemplo el programa principal es *freertos_demo.c*.

La estructura del programa principal sigue el funcionamiento previamente explicado. El encargado de decidir qué tarea se ejecuta es el gestor de tareas. Por lo tanto, en el programa se encontrará que en la definición de la función principal no hay código específico de la aplicación. El código específico lo se encuentra en cada una de las tareas. En el caso de FreeRTOS en la función principal se inicializa el sistema, los periféricos a utilizar y las memorias.

Figura 17. **Función principal del archivo freertos_demo.c**

```
Int main(void)
{
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
                       SYSCTL_OSC_MAIN);
    ConfigureUART();
    UARTprintf("\n\nWelcome to the EK-TM4C123GXL FreeRTOS Demo!\n");
    g_pUARTSemaphore = xSemaphoreCreateMutex();
    if(LEDTaskInit() != 0)
    {
        while(1)
        {
        }
    }
    if(SwitchTaskInit() != 0)
    {
        while(1)
        {
        }
    }
    vTaskStartScheduler();
    while(1)
    {
    }
}
```

Fuente: elaboración propia.

Como se observa en la estructura de la función principal, primero se configura el reloj del microcontrolador. El código para configurar el sistema es TivaWare, en un apartado se hará referencia a este para refrescar la memoria. Luego de inicializar el reloj se configura el UART, se crea un mutex para el UART. Las tareas a ejecutar son ledTask y SwitchTask, las cuales retornan un código de error mayor a 0 en caso de que la creación de la tarea haya fallado.

- Encabezado:

En esta sección se contienen todas las llamadas a librerías específicas de la arquitectura, estas son las seguidas del *inc/* y *driverlib/*. En este ejemplo particularmente las tareas son creadas externamente.

En el encabezado también se puede observar que se carga el sistema operativo.

Figura 18. **Encabezados**

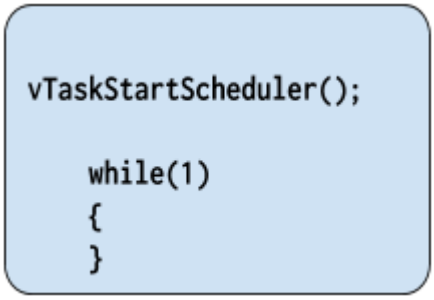
```
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include
"driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "led_task.h"
#include "switch_task.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
```

Fuente: elaboración propia.

- Función *main*:

Las líneas de la figura son las últimas de esta función. Estas dos se encargaron de iniciar el gestor de tareas luego de que las tareas ya fueron creadas. El ciclo infinito se utiliza para que en caso de un error y el gestor de tareas retorne, la aplicación quede suspendida.

Figura 19. **Función para iniciar el gestor de tareas**



Fuente: elaboración propia.

La estructura genérica de una aplicación de FreeRTOS. Como es de esperar primero que nada sigue todas las directivas de C. Primero las librerías a utilizar, luego las definiciones de macros. Luego la declaración de variables globales, funciones y por último el main. En FreeRTOS se cuenta con varias funciones útiles, estas son predefinidas por el sistema operativo y permiten en varios casos interactuar con el *kernel.* Estas funciones son:

vApplicationMallocFailedHook (): esta función es una función gancho que es llamada cuando el pvPortMalloc() falla. Esta función se habilita en la configuración de FreeRTOS colocando configUSE_MALLOC_FAILED_HOOK en 1. pvPortMalloc() es llamada por el kernel cada vez que se crea una tarea, una cola, un semáforo o un timer. También es activada cuando pvPortMalloc() falla en implementaciones del usuario.

vApplicationIdleHook (): consiste en una función que se ejecutará con la menor prioridad posible. Esto significa que siempre que no haya una tarea de mayor prioridad esta continuará ejecutándose. Esto permite que sea una función ideal para condiciones de bajo consumo. Se habilita colocando en 1 la entrada configUSE_IDLE_HOOK en el archivo de configuración de FreeRTOS.

vApplicationstackOverflowHook(): al momento de crear una tarea, esta se le crea su propia pila. Esto hace que el desbordamiento de pila sea una causa común de inestabilidad en la aplicación. FreeRTOS provee dos mecanismos de detección, dependiendo de la constante utilizada para configurar en el archivo *FreeRTOSConfig.h* configCHECK_FOR_STACK_OVERFLOW. Existen dos tipos de detección de desborde, los cuales se seleccionan con 1 o 2 al modificar la entrada correspondiente en la configuración:

- Método 1: el RTOS comprueba que el puntero de la pila esté en el rango válido al momento de un cambio de contexto. Esto se debe a que durante el cambio de contexto toda la información de la tarea es empujada a la pila. Este método es veloz pero no garantiza atrapar todos los desbordes. Se activa colocando en 1 la entrada.

- Método 2: al crear una tarea la pila se llena de valores predeterminados, en el momento de un cambio de contexto el *kernel* comprueba que los últimos 16 bytes no cambiaron. Es menos eficiente que el primer método y tampoco garantiza atrapar todos los desbordes. Al colocar la entrada en 2 se utiliza este método en conjunto con el primero.

vApplicationTickHook()**:** la interrupción del *timer* puede opcionalmente llamar una función. Esto permite temporizar eventos de forma sencilla. La función será llamada solamente si configUSE_TICK_HOOK es colocado en 1 dentro de *FreeRTOSConfig.h*. vApplicationTickHook() se ejecuta dentro de una rutina de interrupción, por lo tanto tiene que ser corto, sin pila, y no puede llamar funciones del API que no sean FROM_ISR.

2.2. Definición de tareas y corutinas

Continuando con el ejemplo anterior, ahora se analizará el código en cada una de las definiciones de nuestras tareas. Primero, veremos led_*task.h*.

Figura 20. **Ejemplo de una tarea definida por usuario**

```
Uint32_t LEDTaskInit(void)
{
    RGBInit(1);
    RGBIntensitySet(0.3f);
    g_ui8ColorsIndx = 0;
    g_pui32Colors[g_ui8ColorsIndx] = 0x8000;
    RGBColorSet(g_pui32Colors);
    UARTprintf("\nLed %d is blinking. [R, G, B]\n", g_ui8ColorsIndx);
    UARTprintf("Led blinking frequency is %d ms.\n", (LED_TOGGLE_DELAY *
2));
    g_pLEDQueue = xQueueCreate(LED_QUEUE_SIZE, LED_ITEM_SIZE);
    if(xTaskCreate(LEDTask, (const portCHAR *)"LED", LEDTASKSTACKSIZE,
NULL, tskIDLE_PRIORITY + PRIORITY_LED_TASK, NULL) != pdTRUE)
    {
        return(1);
    }
    return(0);
}
```

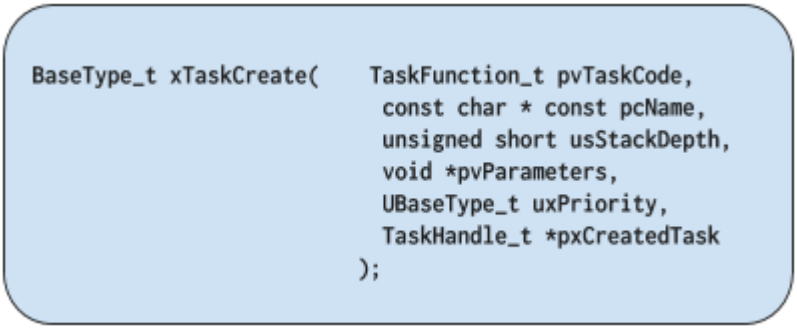
Fuente: elaboración propia.

El código corresponde a la función que define la tarea led y la crea. Primero se inicializan los ledes con la función RGBInit(1) y se procede a configurar la intensidad entre otras funciones.

2.2.1. Creación de tareas

Esta corresponde a la creación de la tarea. La función que se utiliza para crear una tarea es xTaskCreate(). Esta tarea tiene el siguiente prototipo:

Figura 21. **Prototipo de la función para crear tareas**



```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           unsigned short usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                          );
```

Fuente: (n.d.). *RTOS xTaskCreate() FreeRTOS API function*. http://www.freertos.org/a00125.html. Consulta: 14 de noviembre de 2017.

pvTaskCode — Puntero a la función que implementa la tarea. Las tareas se implementan en ciclos infinitos. Una tarea jamás debe de retornar o salir de la función, pero si pueden eliminarse a sí mismas.

pcName — Un puntero tipo *char* que sirve para darle un nombre descriptivo a la tarea. Facilita el *debug*, pero también es útil para obtener el *handler* de la tarea. El tamaño máximo se configura con el parámetro configMAX_TASK_NAME_LEN.

usstackDepth — El número de palabras para crear la pila de la tarea. El tamaño de la pila multiplicado por el ancho de la pila no tiene que exceder el valor máximo que puede ser contenido en una variable de tipo size_t.

pvParameters	Un puntero de tipo *void* que contiene los parámetros que serán pasados en la tarea recién creada. Si entre los parámetros se envía la dirección de un variable, entonces esta variable debe existir al momento de crear la tarea y también cuando la tarea se ejecuta.
uxPriority	La prioridad con la cual la tarea se ejecutará. Los valores que se le puede asignar a cada tarea van desde 0 a configMAX_PRIORITIES-, el cual se define en el *FreeRTOSConfig.h*.
pxCreatedTask	Es el manipulador que se utilizará para crear la tarea. Es opcional y puede dejarse como *NULL*.

En sistemas multitarea es necesario almacenar el contexto de la tarea en la pila, y restaurar el contexto de la tarea que se va a ejecutar. Para esto se utilizan bloques de control de tareas (TCB, *task control block* por sus siglas en inglés). Existe un TCB por cada tarea y cada TCB almacena el puntero de pila (*stack pointer*) de la tarea. En cada cambio de contexto, se almacenan el estado de la tarea junto con sus variables en la pila correspondiente. Es importante que la pida no se desborde, porque de suceder esto sería imposible restaurar la tarea debido a datos insuficientes. Considerando que al trabajar con microcontroladores la memoria es limitada tarea el tamaño del puntero en la ram tiene que considerar otras tareas también.

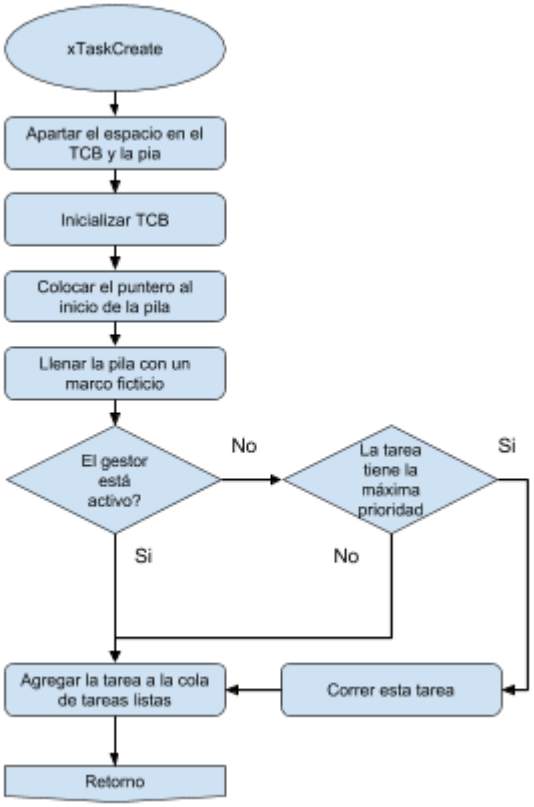
Debido a que no es sencillo calcular el tamaño de la pida requerido por las tareas de antemano, FreeRTOS provee una función para esto.

- uxTaskGetstackHighWaterMark(TaskHandle_t xTask): la pila utilizada por una tarea varía al ser ejecutada. Esta función retorna la cantidad mínima de espacio libre en la pila desde que la tarea comenzó a ejecutarse. Retorna el espacio de la pila que no se utiliza cuando la tarea llegó a su valor máximo de espacio ocupado en la pila. Toma como parámetro el manipulador de la tarea a medir. Retorna el valor en palabras del punto máximo. Si el valor es cero, la tarea se desbordó. Al enviar el manipulador como NULL le indicamos que mida la tarea actual.

Al definir la pila de una tarea se tiene que evitar siempre una pila menor que la definida en configMINIMAL_STACK_SIZE.

El proceso de creación de una tarea se muestra a continuación:

Figura 22. **Creación de una tarea**



Fuente: GOYETTE, Richard. *An Analysis and Description of the Inner Workings of FreeRTOS.* http://richardgoyette.com/Research/Papers/FreeRTOSPaper.pdf. Consulta: 14 de noviembre de 2017.

xTaskCreate primero debe apartar el espacio para la tarea en el TCB y en la pila. Esto lo hace utilizando la función portMalloc para obtener el bloque de memoria el TCB y de la pila. portMalloc está implementado en el HAL.

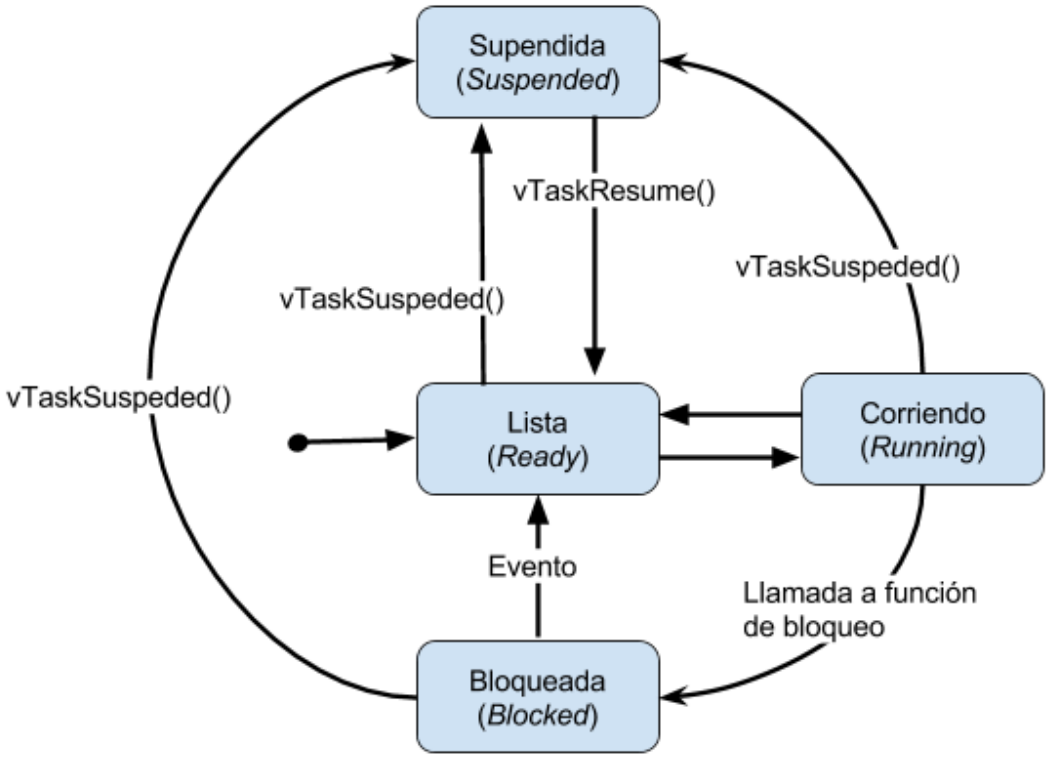
Ya que se tiene el espacio en el TCB este se llena con la información de la tarea.

El marco ficticio contiene todo lo requerido para un cambio de contexto. Lo más importante del marco ficticio es que retorna la dirección que indicará la dirección donde inicia el código de la tarea.

2.2.2. Estados de las tareas FreeRTOS

FreeRTOS provee varias funciones para interactuar con las tareas. El *kernel* interactúa con las tareas para modificar su estado. En FreeRTOS las tareas se comportan como se describe en la figura 23.

Figura 23. **Diagrama de estados de una tarea**



Fuente: GOYETTE, Richard. *An Analysis and Description of the Inner Workings of FreeRTOS.* http://richardgoyette.com/Research/Papers/FreeRTOSPaper.pdf. Consulta: 14 de noviembre de 2017.

Las tareas pueden estar en uno de estos cuatro estados:

- Corriendo (*running*): cuando una tarea está en ejecución. Si el sistema solo tiene un núcleo sólo una tarea puede estar en este estado a la vez.
- Lista (*ready*): estas tareas están listas para ejecución, pero estar esperando que se libere el procesador para ser ejecutadas
- Bloqueada (*blocked*): una tarea está bloqueada cuando espera un evento temporal o externo. Por ejemplo vTaskDelay() bloqueará la tarea hasta que se cumpla el tiempo. Las tareas también pueden ser bloqueadas cuando esperan una cola, un semáforo, banderas de eventos o notificaciones. Las tareas bloqueadas tienen un tiempo de espera luego del cual son desbloqueadas sin importar si ocurrió el evento que estaban esperando. En este estado no ocupan CPU ni pueden pasar al estado corriendo.
- Suspendida (*suspended*): al igual que las tareas bloqueadas estas no pueden pasar a ejecución, pero no tienen un tiempo de espera. En cambio las tareas solo pueden entrar a estado suspendido de forma explícita con las funciones vTaskSusped() y vTaskResume().

2.2.3. Funciones específicas de FreeRTOS para tareas

Como ya se describió en la sección anterior, el *kernel* tiene funciones específicas para interactuar con tareas. Es importante resaltar algunas que permiten no solo la manipulación de parámetros de tareas luego de su creación, sino que también permiten recuperar información de las mismas. Entre estas funciones existen las funciones de control y las utilitarias. No es necesario conocer todos. Pero es importante conocer los recursos que se pueden utilizar sin necesidad de implementar más código.

- Funciones de control

vTaskDelay()	Bloquea una tarea por una cantidad de *ticks* determinada. El tiempo es relativo al instante en el que se llama a la función.
vTaskDelayUntil()	Bloquea la tarea hasta que se cumple un plazo. Especifica un periodo absoluto. Lleva un control de última vez que la tarea se bloqueó para crear periodos de ejecución más estrictos.
vTaskPrioirtyGet()	Devuelve la prioridad de una tarea en un momento específico.
vTaskPrioiritySet()	Permite modificar la prioridad de una tarea ya creada.
vTaskSuspend()	Suspende una tarea. La tarea suspendida nunca se ejecutará hasta que explícitamente se indique.
vTaskResume()	Cuando una tarea fue suspendida, esta función es la única que permite que se vuelva a ejecutar.
vTaskResumeFromISR()	Una función que permite resumir una tarea desde una interrupción. No es recomendable ya que si la tarea se suspende luego de la

	interrupción esta pierde.
vTaskAbortDelay()	Fuerza una tarea a salir del estado bloqueada sin importar si ocurrió el evento por el cual estaba bloqueada.

- Funciones utilitarias

uxTaskGetSystemState()	Devuelve el manipulador, el nombre de la tarea, el número de la tarea, el estado actual, la prioridad, prioridad base en caso de que haya sido heredada, el tiempo de ejecución asignado, la dirección de la pila y la marca de desborde. Llena una estructura tipo Task_Status_t para cada tarea del sistema.
vTaskGetInfo()	Igual a uxTaskGetSystemState() pero solo lo hace para una tarea. Al usarse suspende el gestor de tareas así que no debe usarse para otra cosa que no sea *debug.*
eTaskGetState()	Devuelve el estado de la tarea.
vTaskGetTickCount()	Devuelve la cantidad de *ticks* desde que se inició el gestor de tareas.
xTaskGetSchedulerSate()	Devuelve el estado actual del gestor de tareas, el cual puede ser:

	taskSCHEDULER_NOT_STARTED taskSCHEDULER_RUNNING taskSCHEDULER_SUSPENDED
vTaskGetRunTimeStats()	Llama la función uxTaskGetSystemState() y le da formato a los datos para crear una tabla con los tiempos de ejecución de cada tarea.

Todas estas funciones permiten interactuar con las tareas para determinar si las tareas se cumplen en el tiempo requerido. Es importante entender que la mayoría de estas solo están para ser utilizadas durante pruebas y no para operación normal.

2.3. Gestores de tarea estáticos

Existen varios tipos de gestores de tareas. Los estáticos son los que no cambian la prioridad de las tareas. Estos son muy comunes en sistemas operativos de tiempo real. Son simples y predecibles, ambas propiedades son deseables.

El gestor de tareas es el encargado de colocar las tareas en la línea de tiempo, por lo tanto, tiene que realizarlo de tal forma que se cumplan los plazos de ejecución.

2.3.1. Planificador RM (*rate monotonic*)

Este es un planificador cíclico que asigna prioridad a las tareas de acuerdo a la duración del trabajo. Por lo tanto, las tareas con menos tiempo de ejecución

reciben mayor prioridad. Estos suelen ser expriopiativos. Para utilizar un planificador RM las tareas deben cumplir con las siguientes características:

- Los procesos no comparten recursos de hardware. No existe dependencia entre procesos.
- Prioridades estáticas y expropiación.
- Prioridades asignadas de acuerdo a los periodos de las tareas.

El sistema es realizable solo si se cumple la siguiente relación.

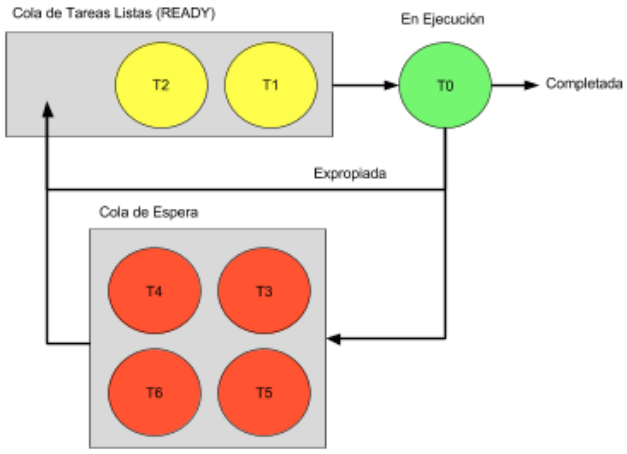
$$U = \sum_{i=1}^{n} \frac{e_i}{P_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Ecuación 1: validación de un sistema RM

2.3.2. Planificación *round-robin*

Este algoritmo utiliza prioridades estáticas. Es un algoritmo expropiativo que concede a cada tarea un tiempo en el procesador determinado, llamado cuanto o porción de tiempo. Todos los procesos tienen la misma porción de tiempo en el procesador. Si la tarea se ejecuta en menos tiempo que el que tiene asignado, el gestor pasa a la siguiente tarea.

Figura 24. **Gestor de tareas tipo round-robin**



Fuente: elaboración propia.

En esta política no existe concepto de prioridad, ya que todas las tareas reciben el mismo tiempo y ninguna es expropiada en favor de otra.

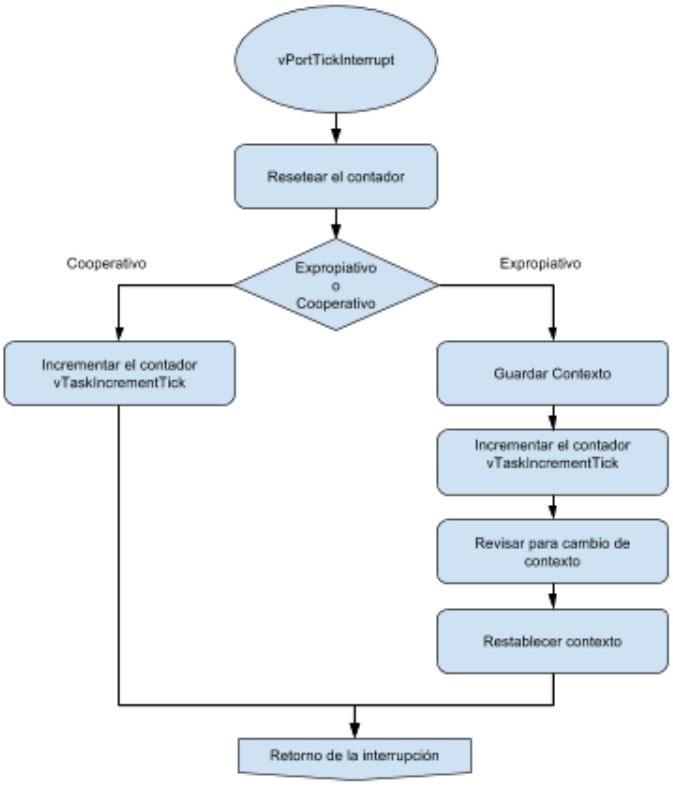
Cuando la tarea requiere más tiempo es expropiada al terminarse su porción de tiempo y retorna a la cola de tareas listas para esperar su turno. Algunas características de este algoritmo son:

- Si la porción de tiempo es pequeña, se pierde mucho tiempo en cambios de contexto.
- Si la porción de tiempo es suficientemente grande es una FIFO que completa los trabajos de acuerdo al orden de llegada.
- Es muy difícil que todas las tareas cumplan con los plazos debido a que la espera depende de las tareas en el sistema.

2.3.3. Gestor de tareas de FreeRTOS

El mecanismo utilizado por el gestor de tareas de FreeRTOS se describe más adelante. Debido a que en el gestor la cooperación y la suspensión son configurables, el mecanismo tiene complejidad considerable. El gestor opera como una rutina de interrupción del *timer* vPortTickInterrupt() que se activa cada periodo. El periodo se configura en el archivo *FreeRTOSConfig.h* con el parámetro configTICK_RATE_HZ.

Figura 25. **Gestor de tareas de FreeRTOS**



Fuente: GOYETTE, Richard. *An Analysis and Description of the Inner Workings of FreeRTOS.* http://richardgoyette.com/Research/Papers/FreeRTOSPaper.pdf. Consulta: 14 de noviembre de 2017.

Como el gestor de tareas actúa como una interrupción, forma parte del HAL y requiere de código específico para la plataforma. El compilador es el encargado de interpretar el código específico. La primera operación realizada por el gestor es resetear el contador para iniciar el periodo de conteo. Luego, dependiendo de la configuración de FreeRTOS, expropiativo y cooperativo, el gestor cambiara su comportamiento.

En el caso cooperativo la única operación que realiza antes de retornar es aumentar el conteo.

En el caso expropiativo primero se coloca en la pila el contexto de la tarea actual, en caso de que sea necesario. Se aumenta el contador, y se verifica el estado de las tareas. Si una tarea de mayor prioridad se desbloqueó, entonces se realiza un cambio de contexto. Finalmente, el contexto es restaurado, y el gestor retorna de la interrupción.

Los sistemas estáticos permiten analizar la utilización del procesador con una simple fórmula.

$$U = \sum_{i=1}^{n} \frac{e_i}{P_i}$$

Ecuación 2: utilización de un CPU

Donde e_i representa el peor caso de tiempo de ejecución (*WCET*) y P_i es el periodo de la tarea. Para que un sistema sea realizable tiene que cumplirse:

$$U < 1$$

Ecuación 3: restricción de utilización

Esto significa que nuestras tareas se ejecutan dentro de su periodo y consumen menos del 100 % de nuestro CPU. En caso de que el gestor sea expropiativo se debe tomar en cuenta que los cambios de contexto producen un gasto de tiempo de procesamiento, por lo tanto, las tareas reciben menos tiempo en el procesador.

La política del gestor de tareas de FreeRTOS permite cierta flexibilidad para asegurar que las tareas se completen sin necesidad de modificar el código del gestor de tareas.

2.4. Introducción a Simso

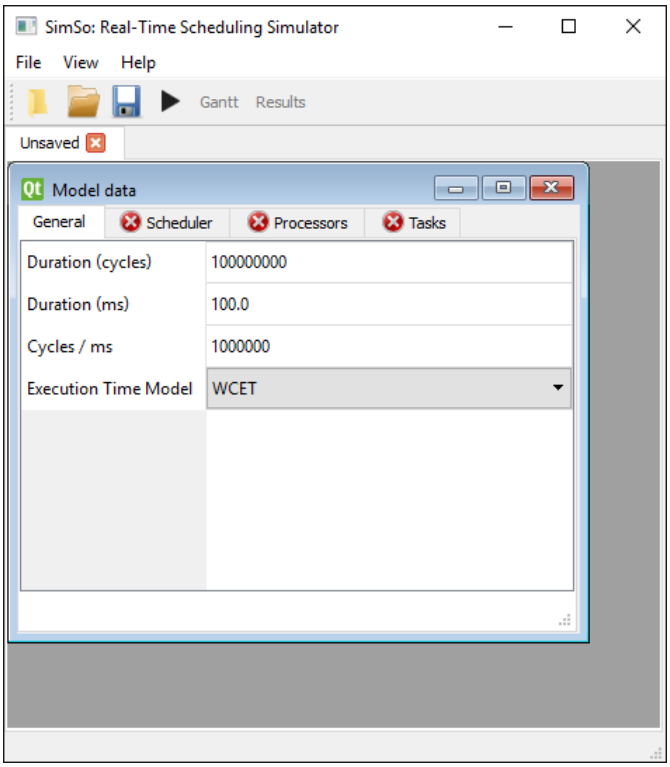
Simso es una herramienta basada en python para la simulación de gestores de tareas. Permite visualizar con facilidad si el sistema será capaz de cumplir con las tareas en el tiempo necesario. Simso tiene una interfaz sencilla y es fácil de familiarizarse con las opciones. Al abrirlo tenemos la siguiente pantalla.

La ventana de *model data* es la que permite modificar los parámetros para las simulaciones. Cada una de las pestañas presenta opciones individuales para ese elemento en específico.

En el panel *general* se tiene la cantidad de ciclos que se simularán. La duración de cada ciclo en milisegundos y la relación de ciclos por milisegundo.

Al final se puede seleccionar el modelo de tiempo de ejecución. Modelo de cache, para sistemas con ram, penalización fija, peor caso de tiempo de ejecución (WCET) y caso promedio de tiempo de ejecución, para las simulaciones sólo se utilizará WCET.

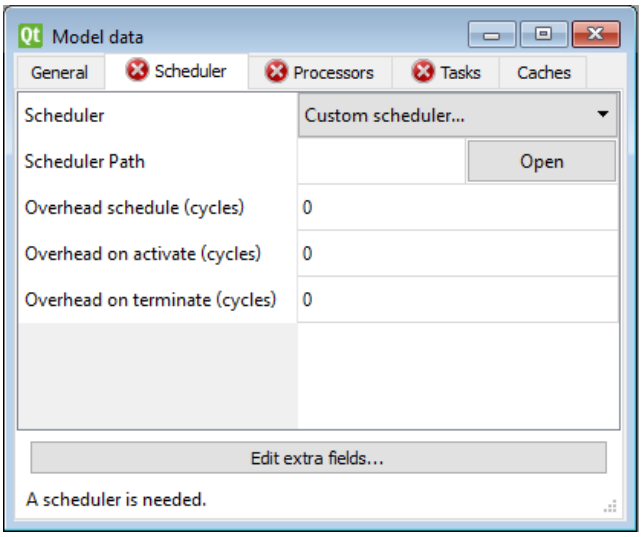
Figura 26. **Simso, pantalla de inicio**



Fuente: elaboración propia, empleando Simso.

La figura 27 muestra el panel que permite seleccionar el gestor de tareas. Los gestores cambian dependiendo de las reglas que utilizan para priorizar las tareas. Simso permite utilizar gestores dinámicos y estáticos. Asimismo, permite utilizar gestores definidos por el usuario. Permite especificar la sobrecarga por el gestor para tener una simulación más cercana a la realidad.

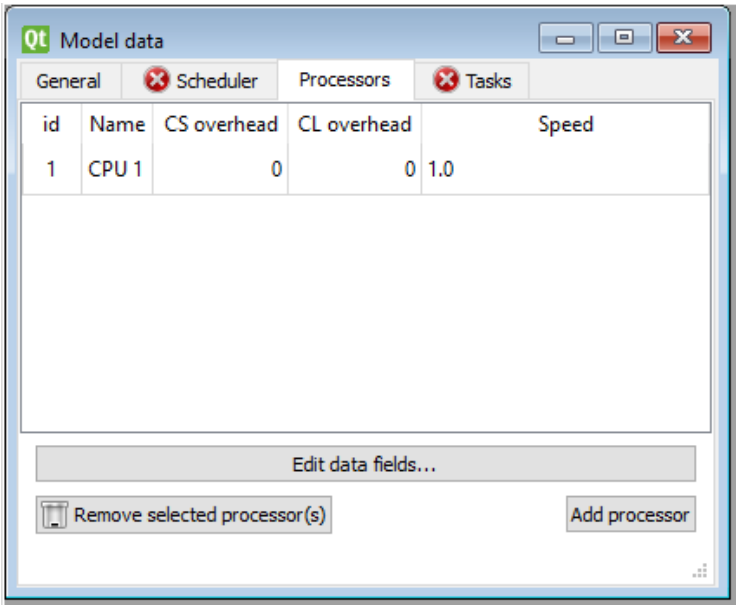
Figura 27. **Simso, configuración del gestor de tareas**



Fuente: elaboración propia, empleando Simso.

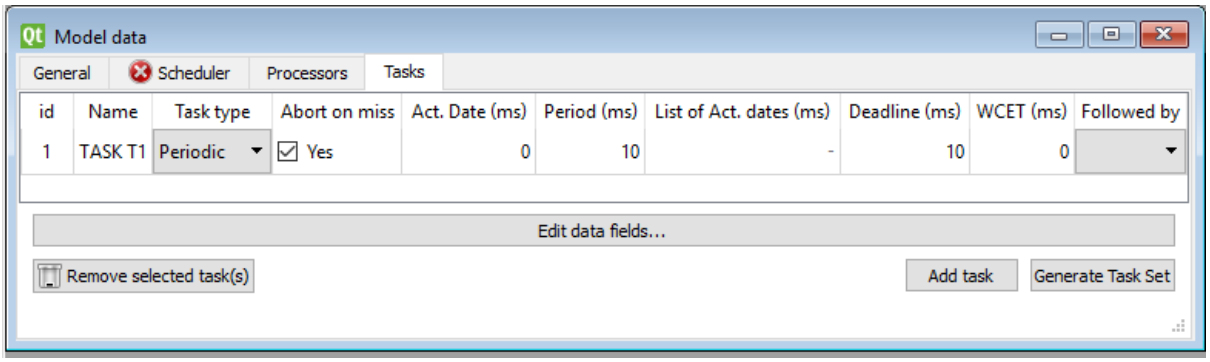
El panel de procesador permite agregar procesadores y modificar sus parámetros. Como la sobrecarga por cambio de contexto y la velocidad del procesador. Simso permite realizar simulaciones para sistemas de múltiples núcleos.

Figura 28. **Simso, configuración del procesador**



Fuente: elaboración propia, empleando Simso.

Figura 29. **Simso, configuración de las tareas a simular**



Fuente: elaboración propia, empleando Simso.

El panel de tareas permite crear tareas con varios parámetros importantes. Como ya se mencionó previamente, el periodo de la tarea, el plazo (*deadline*),

el peor caso de tiempo de ejecución (WCET) y otras opciones. Permite determinar el tipo de tarea, periódica, esporádica o irregular (*aperiodic*).

Como otras opciones de sincronización permite determinar el tiempo de activación de la tarea. Determinar si hay tareas que le sigue, es decir, que dependen de la ejecución exitosa de esta.

Luego de crear el procesador con las características deseadas, haber elegido el gestor de tareas y haber llenado la lista de tareas, se prosigue a realizar la simulación presionando el botón con la flecha negra. Luego de realizada la simulación se puede observar la línea de tiempo presionando el botón de *Gantt.*

Figura 30. **Simso, diagrama de Gantt**

Fuente: elaboración propia, empleando Simso.

En esta simulación se puede observar que son tres tareas corriendo en un solo procesador. El gestor de tareas es estático, por lo tanto, toma la tarea de mayor prioridad que esté disponible en ese instante. Una propiedad de los sistemas periódicos es el hiperperiódo.

El hiperperiódo es el espacio de tiempo en el que el sistema completo comienza a repetir un patrón previo. Se puede encontrar con facilidad ya que el hiperperiódo es el mínimo común múltiplo de todos los periodos de las tareas. Esto solo aplica cuando no hay tareas esporádicas o irregulares en el sistema.

2.5. Práctica 2: gestores de tareas estáticos y medición de tiempos en FreeRTOS

En esta sección se muestran ejercicios prácticos para la implementación del conocimiento adquirido sobre planificadores estáticos y el uso de herramientas de programación para medición de tiempos.

2.5.1. Simulación de gestores de tareas

Para la simulación se utilizará Simso. La práctica consiste en:

- Determinar el sí es posible planificar las siguientes tareas de acuerdo a la utilización total.
- Luego, realizar la simulación con un gestor de tareas tipo RM

Tabla I. **Tareas para simulación de planificador RM**

	Periodo (P)	Ejecución (e)	Plazo (D)
Tarea 1	3	0.5	
Tarea 2	4	1.5	3
Tarea 3	7	1	5

Fuente: elaboración propia.

Contestar las siguientes preguntas:

- ¿Alguna tarea no cumple con su plazo? ¿Qué tarea? ¿Dónde?

- ¿Si alguna tarea no cumple con su plazo, podría ser evitado cambiando el algoritmo, por qué?

Nota: Las tareas tienen que cumplirse antes de que su periodo.

2.5.2. Medición de periodos y tiempos de ejecución

El objetivo de esta práctica es familiarizarse con las herramientas de *debug* que provee FreeRTOS.

Como ya se mencionó previamente es necesario conocer el tiempo de ejecución de las tareas y el periodo al cual se realizan, de lo contrario la sincronización con aplicaciones reales se vuelve una tarea imposible.

Esta práctica puede realizarse ya se en el emulador, utilizando Visual studio, o también utilizando el microcontrolador tiva C.

- Proveer el código fuente para su revisión.
- Un reporte con una captura de pantalla y detalles sobre la solución implementada.

En el caso que se utilice el microcontrolador proveer una captura de pantalla de la consola e imprimir por UART la información requerida.

En caso que se utilice el emulador proveer la captura de pantalla de la consola y los datos del procesador.

Abra un proyecto de FreeRTOS y copie la siguiente tarea.

```
#define SIZE 10
#define ROW SIZE
#define COL SIZE
xTaskHandle matrix_handle;

static void matrix_task()
{
	int i;
	double **a = (double **)pvPortMalloc(ROW * sizeof(double*));
	for (i = 0; i < ROW; i++) a[i] = (double *)pvPortMalloc(COL * sizeof(double));
	double **b = (double **)pvPortMalloc(ROW * sizeof(double*));
	for (i = 0; i < ROW; i++) b[i] = (double *)pvPortMalloc(COL * sizeof(double));
	double **c = (double **)pvPortMalloc(ROW * sizeof(double*));
	for (i = 0; i < ROW; i++) c[i] = (double *)pvPortMalloc(COL * sizeof(double));
	double sum = 0.0;
	int j, k, l;
	for (i = 0; i < SIZE; i++) {
		for (j = 0; j < SIZE; j++) {
```

```
                a[i][j] = 1.5;
                b[i][j] = 2.6;
            }
        }

        while (1) {
            long simulationdelay;
            for (simulationdelay = 0; simulationdelay<1000000000; simulationdelay++);
            for (i = 0; i < SIZE; i++) {
                for (j = 0; j < SIZE; j++) {
                    c[i][j] = 0.0;
                }
            }
            for (i = 0; i < SIZE; i++) {
                for (j = 0; j < SIZE; j++) {
                    sum = 0.0;
                    for (k = 0; k < SIZE; k++) {
                        for (l = 0; l<10; l++) {
                            sum = sum + a[i][k] * b[k][j];
                        }
                    }
                    c[i][j] = sum;
                }
            }
            vTaskDelay(100);
        }
}
```

Implemente la tarea dentro de la función main con la siguiente línea para TivaWare:

```
xTaskCreate((pdTASK_CODE)matrix_task, (const char *)"Matrix", 1000, NULL, 3, &matrix_handle);
```

Implemente la tarea dentro de la función main con la siguiente línea para el emulador de Visual studio:

xTaskCreate((pdTASK_CODE)matrix_task, (static char *)"Matrix", 1000, NULL, 3, &matrix_handle);

3. TEMPORIZACIÓN DE TAREAS, INTERRUPCIONES Y SEMÁFOROS

3.1. Semáforos

Los semáforos permiten controlar el orden en el que las tareas se ejecutan. Primero se tiene la consideración de las prioridades en el *kernel* para determinar qué tareas se ejecutan primero y a cuál se le da más tiempo de ejecución. Pero también es necesario una forma de organizar tareas de acuerdo a orden de ejecución.

Los semáforos permiten bloquear tareas y desbloquearlas cuando alguna condición necesaria sea haya cumplido, ya sea por otra tarea o por una interrupción.

Para utilizar los semáforos, es necesario incluir el archivo de definición respectivo de FreeRTOS. Este archivo es *semphr.h.* En este archivo se especifican los semáforos binarios, los semáforos contables y los mutex.

Las tareas interactúan con todos los semáforos de la misma forma, pero depende del tipo de semáforo cómo reacciona ante las peticiones de las tareas.

Las tareas toman semáforos utilizando la función de la API xSemaphoreTake(). La función solicita dos parámetros:

```
xSemaphoreTake(SemaphoreHandle_t xSemaphore,
               TickType_t xTicksToWait);
```

Primero el manipulador del semáforo y opcionalmente un tiempo de espera. Si el tiempo de espera se agota antes de recibir el semáforo entonces retorna pdFALSE. Lo cual indica que no se pudo obtener el semáforo.

Para entregar un semáforo, una tarea llama la función:

xSemaphoreGive(SemaphoreHandle_t xSemaphore)

La tarea sólo debe proveer el manipulador del semáforo a retornar. La función puede retornar pdTRUE si se liberó el semáforo, o pdFALSE si hubo un error. Lo semáforos funcionan como colas, así que un error podría significar que no hay espacio en la cola, lo cual indica que no se tomó el semáforo correctamente o que no hay semáforo para retornar.

3.1.1. Semáforos binarios

Los semáforos binarios como se mencionó anteriormente permiten controlar acceso a recursos o restringir el orden de ejecución de dos tareas. Estos semáforos solo pueden estar en dos estados, ocupado o disponible.

Para crear un semáforo binario y para interactuar con este, se utilizan las funciones definidas dentro del API de FreeRTOS. Para habilitar la creación de semáforos es necesario que configSUPPORT_DYNAMIC_ALLOCATION este configurado con un valor de 1 o sin definir (el estado por defecto es habilitado).

La API de FreeRTOS provee dos formas de crear un semáforo dependiendo de cómo se quiera manejar la memoria ram. Esto se debe a que los semáforos requieren espacio en la ram para almacenar su estado.

FreeRTOS permite crear los semáforos con memoria dinámico y con memoria estática, el usuario define la memoria.

Primero es necesario definir un manipulador de tipo xSemaphore. Luego de creado el manipulador se puede crear el objeto tipo xSemaphoreBinary() o xSemaphoreBinaryStatic().

Al crear un semáforo con memoria estática es necesario pasar como argumento de la función un puntero tipo StaticSemaphore_t, esta variable será la encargada de almacenar el estado del semáforo.

Figura 31. **Ejemplo de la creación de un semáforo**

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL )
    {
       // El semáforo no se pudo crear
    }
}
```

Fuente: elaboración propia.

La figura 31 muestra un ejemplo del uso, la plantilla es genérica y utiliza la comparación para determinar si el semáforo fue creado con éxito. Si el semáforo no se crea exitosamente es porque no hay suficiente espacio en la memoria de FreeRTOS. Los semáforos se crean vacíos, por lo tanto antes de

poder utilizarlos (tomarlos), es necesario habilitarlo con la función xSemaphoreGive() y luego ya puede utilizarse con normalidad.

Es importante recordar que un semáforo permite ser retornado por otra tarea o interrupción.

En el caso de un semáforo estático no cambia demasiado solo es necesario crear la variable estática para el almacenamiento del estado del semáforo.

Figura 32. **Ejemplo de la creación de un semáforo estático**

```
SemaphoreHandle_t xSemaphore = NULL;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    xSemaphore = xSemaphoreCreateBinaryStatic(&xSemaphoreBuffer);
    if( xSemaphore == NULL )
    {
       // El semáforo no se pudo crear
    }
}
```

Fuente: elaboración propia.

3.1.2. Semáforos con contador

Los semáforos con contador permiten múltiples accesos. Estos llevan un control de la cantidad de usos en un momento determinado, estos semáforos son ideales para conteos de eventos y manejo de recursos. Es recomendable utilizarlos cuando no es importante la información de la cola sino solo si está disponible o no.

En el caso en el que un semáforo se utilice para contabilizar eventos, cada vez que ocurra el evento, se concederá un semáforo, esto aumentará el valor del contador. Cada vez que uno de estos eventos sea procesado se tomará el semáforo al manipulador, lo cual disminuye la cuenta. En este caso el valor de interés será la diferencia entre los eventos ocurridos y los eventos procesados. Idealmente se busca que este valor se mantenga en cero, lo cual significa que todos los eventos ocurridos han sido procesados. En este caso el semáforo se inicializa con un valor de cero.

En el caso que se utilice el semáforo para manejo de recursos, en el momento que una tarea necesite un recurso tomará un semáforo, esto disminuye el conteo. En el momento que la tarea termine de utilizar el recurso lo entregará junto con el semáforo, aumentando la cuenta. Al alcanzar cero no se concederán más recursos, ya que se habrán agotado. Para este caso el semáforo se inicializa en su valor máximo.

Los semáforos con contador también tienen su versión con memoria estática xSemaphoreCreateCountingStatic(), siendo la versión con memoria dinámica xSemaphoreCreateCounting().

La diferencia entre ambas es únicamente que uno utiliza una variable predefinida por el usuario para almacenar el estado del semáforo.

Figura 33. **Ejemplo de la creación de un semáforo con contador**

```
//SemaphoreHandle_t xSemaphoreCreateCounting(
                            UBaseType_t uxMaxCount,
                            UBaseType_t
                       uxInitialCount);
SemaphoreHandle_t xSemaphore;

void vATask(void * pvParameters )
{
     xSemaphore = xSemaphoreCreateCounting( 10, 0
);
     if( xSemaphore == NULL )
     {
        /*Se creó el semáforo con contador */
     }
}
```

Fuente: elaboración propia.

En este ejemplo la línea en rojo no es parte del código a ejecutar. Esta línea muestra los parámetros que recibe la función y que tipo de variable debe recibir el resultado. En este caso como en todos los semáforos, el resultado lo recibe el manipulador de tipo SemaphoreHandle_t. Los parámetros que recibe la función son:

- uxMaxCount: esta es una variable de tipo UBaseType_t, lo cual significa que es un número sin signo. Este corresponde al valor máximo del semáforo.
- uxInitialCount: este parámetro permite crear un semáforo con un valor inicial.

Para obtener el valor de conteo del semáforo basta con llamar la función uxSemaphoreGetCount(xSemaphore), con el manipulador del semáforo.

3.2. Mutex

Los mutex son un subtipo de los semáforos que incluyen un mecanismo de herencia. Esto significa que el mutex reconoce la tarea que lo tomo y solo puede ser retornado por esa tarea. Por lo tanto, los mutex son mutuamente excluyentes.

FreeRTOS ofrece cuatro tipos de mutex. Mutex con memoria dinámica y con memoria estática, y mutex recursivo con ambos modelos de memoria.

El mutex recursivo permite que su semáforo sea tomado varias veces por la misma tarea y se retorna hasta que la tarea lo devuelva la cantidad de veces que lo tomo. En contraste el mutex no recursivo solo es tomado una vez y luego retornado.

Este es un ejemplo de la creación de un semáforo tipo mutex. Para las variantes de memoria estática es igual que en el caso del semáforo binario, se agrega static al final de la función de creación y se pasa la variable que almacenará el estado del semáforo. En el caso del mutex recursivo, se utiliza la función xSemaphoreCreateRecursiveMutex().

Figura 34. **Ejemplo de la creación de un mutex**

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    xSemaphore = xSemaphoreCreateMutex();
    if( xSemaphore == NULL )
    {
       // El semáforo no se pudo crear
    }
}
```

Fuente: elaboración propia.

Este es un ejemplo de una creación de un mutex. Luego de creado el mutex una tarea puede necesitar conocer qué tarea bloqueo el mutex. Esto se logra utilizando la función:

xSemaphoreGetMutexHolder(SemaphoreHandle_t xMutex)

Esta función retorna el manipulador de la tarea que tomó el mutex.

3.3. Temporización de tareas

Los microcontroladores siempre proveen temporizadores de hardware. Los cuales, suelen ser más precisos al contar directamente del reloj y no depender del CPU. Al estar implementados en hardware se tiene un número limitado por el hardware. FreeRTOS provee temporizadores por software, estos son sencillos de implementar, pero su correcta implementación requiere de cierto cuidado.

FreeRTOS no ejecuta ninguna función de temporizadores desde el contexto de una interrupción. Los temporizadores de FreeRTOS no consumen tiempo de procesamiento extra, solo cuando expiró el temporizador y ejecuta alguna función. El temporizador utiliza recursos ya existentes en FreeRTOS para que su implementación tenga un impacto mínimo en el tamaño del código de la aplicación.

Las funciones llamadas por los temporizadores se ejecutan en el contexto del temporizador, por lo tanto, no pueden ser bloqueadas.

3.3.1. Configuración de temporizadores

Para utilizar los *timers* es necesario incluir el código en el archivo *timers.c.* También es necesario modificar algunas constantes en el archivo de configuración de FreeRTOS. Estas son:

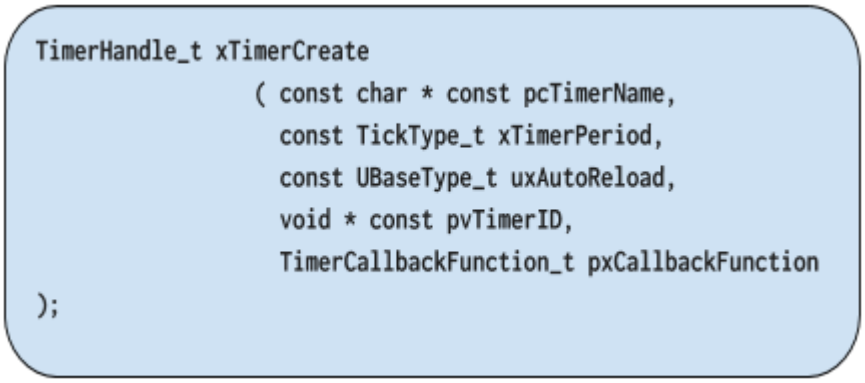
configUSE_TIMERS	Se configura en 1 para incluir la funcionalidad del temporizador. La tarea del *timer* se crea automáticamente cuando inicia el gestor de tareas.
configTIMER_TASK_PRIORITY	Configura la prioridad de la tarea del servicio del *timer*. Al configurarlo es necesario estar consciente de que si tiene la máxima prioridad, las tareas del *timer* fuerzan un cambio de contexto cada vez que estén listas. Si tienen una prioridad baja, deberán esperar para ser ejecutadas.

configTIMER_QUEUE_LENGTH	Cantidad de comandos sin procesar que puede almacenar la cola del *timer*. Esto puede suceder si se hacen múltiples llamadas al *timer* antes de iniciado, múltiples llamadas a la función del *timer* desde una interrupción o una tarea con mayor prioridad.
configTIMER_TASK_STACK_DEPTH	Determina el tamaño de la pila para el contexto de ejecución de las funciones del *timer*.

3.3.2. Creación de temporizadores

Los temporizadores son estructura definidas dentro de la API de FreeRTOS.

Figura 35. **Prototipo de la función para crear temporizadores**



```
TimerHandle_t xTimerCreate
                ( const char * const pcTimerName,
                  const TickType_t xTimerPeriod,
                  const UBaseType_t uxAutoReload,
                  void * const pvTimerID,
                  TimerCallbackFunction_t pxCallbackFunction
);
```

Fuente: xTimerCreate() – FreeRTOS. http://www.freertos.org/FreeRTOS-timers-xTimerCreate.html. Consulta: octubre 9, 2017.

pcTimerName	Un puntero tipo *char* que sirve para darle un nombre

	descriptivo al temporizador. Facilita el debug.
xTimerPeriod	El periodo del temporizador tiene que ir especificado en *ticks* del reloj del CPU. Se puede utilizar la función pdMS_TO_TICKS() para convertir de milisegundos en *ticks*. La función pdMS_TO_TICKS() solo puede ser utilizada si configTICK_RATE_HZ es igual o menor a 1 000.
uxAutoReload	Determina el comportamiento del temporizador al expirar. Si se configura com pdTRUE, el temporizador se reiniciará automáticamente. Si se configura com pdFALSE, el temporizador sólo se disparará una vez y luego de expirará quedará inactivo.
pvTimerID	Un identificador que le permite a las funciones determinar qué temporizador ha expirado en caso de que sean activadas por varios.
pxCallbackFunction	La función a llamar cuando el temporizador expire. Las funciones de retorno tienen que estar definidas como TimerCallbackFunction_t, y siguen el siguiente prototipo: void vCallbackFunction (TimerHandle_t xTimer);

Esta función retorna un manipulador si es ejecutada exitosamente. En caso de que falle, la función retorna NULL, y sea porque no hay espacio en la memoria de FreeRTOS para colocar la nueva estructura o porque el periodo se configuró como 0.

Figura 36. **Ejemplo de la implementación de un temporizador**

```
/* Manipulador del temporizador */
 TimerHandle_t xTimer;

 void vTimerCallback( TimerHandle_t xTimer )
 {
      /* Función del temporizador */
 }

 void main( void )
 {
      xTimer = xTimerCreate( "Timer", 100, pdTRUE,( void * ) 0,vTimerCallback);
      if( xTimers== NULL )
      {
            /* No pudo crearse el temporizador */
      }
     vTaskStartScheduler();

     for( ;; );
 }
```

Fuente: elaboración propia.

Un ejemplo de la creación de un temporizador llamado *timer,* con un periodo de 100 *ticks,* con reinicio automático, identificador cero y su respectiva función.

3.3.3. Interacción con temporizadores

Ya creado un temporizador se puede interactuar con el mismo a través del manipulador. FreeRTOS provee funciones específicas para facilitar la obtención de información de un temporizador.

Al crear un *timer* es necesario iniciarlo, esto se hace con la función:

xTimerStart(TimerHandle_t xTimer, TickType_t xBlockTime);

- TimerHandle_t xTimer: el manipulador del temporizador a iniciar.
- TickType_t xBlockTime: el tiempo que se bloquea la tarea que inició el temporizador para esperar que llegue el mensaje al temporizador.

La función retorna pdPASS si el temporizador recibe el comando. En caso contrario, si el periodo de bloqueo expira, la función retorna pdFAIL.

Ya que inició el *timer* se puede interactuar con sus valores con varias funciones. Por ejemplo, modificar su periodo, con la siguiente función:

xTimerChangePeriod(TimerHandle_t xTimer, TickType_t xNewPeriod,
TickType_t xBlockTime);

- TimerHandle_t xTimer: el manipulador del temporizador a modificar.
- TickType_t xNewPeriod: el nuevo periodo en *ticks* para el temporizador.
- TickType_t xBlockTime: el tiempo que se bloquea la tarea para esperar que llegue el mensaje al temporizador.

Aparte de cambiar el periodo, se puede recuperar el periodo utilizando el manipulador del *timer* con la función xTimerGetPeriod (). También es posible obtener el tiempo en el que expira un temporizador utilizando la función xTimerGetExpiryTime (). Aparte de modificar el periodo,es posible recuperar el nombre del *timer* con pcTimerGetName(). Es posible modificar el identificador del temporizador con vTimerSetTimerID() y también recuperarlo con pvTimerGetTimerID().

xTimerIsTimerActive () verifica si el temporizador asociado al manipulador está activo, retorna pdFALSE si el temporizador esta inactivo. Para reiniciar el *timer* basta con llamar la función xTimerReset() con el respectivo manipulador. Para detener un temporizador se utiliza la función xTimerStop() y el manipulador respectivo.

Se puede eliminar un temporizador con la función xTimerDelete(), que recibe como parámetros el manipulador y un periodo de bloqueo para esperar que se ejecute el comando.

Es posible reiniciar un temporizador cuando ya inicio su ejecución. El tiempo de expiración será recalculado y será relativo al momento en el que se reinicie el temporizador.

3.4. Planificadores dinámicos

Un planificador dinámico puede modificar la prioridad de una tarea en tiempo de ejecución. Esto permite que mientras más cercano esté el tiempo de expiración la tarea reciba más tiempo en el CPU para asegurar que se ejecute antes de que su resultado sea inútil.

Los planificadores analizarán las tareas en cada punto de planificación para determinar si es necesario modificar las prioridades de las tareas. El punto de planificación puede ser una interrupción de un *timer* o cuando una tarea terminó su ejecución. Los algoritmos de planificaciones pueden tomar en cuenta varios aspectos:

- Tiempo previo a la expiración
- El periodo

- El tiempo de holgura
- El tiempo de liberación de la tarea

La flexibilidad los hace una solución atractiva. Pero también es importante considerar que debido a que evalúan en cada cierto tiempo, introducen una carga extra al procesador. Es importante notar que estos planificadores solo aplican para sistemas con expropiación. Modificar prioridades sólo tiene sentido si puede cambiarse el contexto para favorecer otra tarea en cualquier momento.

3.4.1. LST *(least slack time*) menor tiempo de holgura

Este algoritmo evalúa cuánto tiempo tiene la tarea para completar su ejecución antes de que tenga que descartarse el resultado. El tiempo de holgura se define como:

$$s_i = d_i - t - e_i'$$

Ecuación 4: ecuación para el cálculo del tiempo de holgura

- d_i: es el tiempo de expiración de la tarea
- t: es el momento actual
- e_i': es el tiempo de ejecución restante

El algoritmo determinará el tiempo de holgura de cada en tarea. La tarea con mayor prioridad será la tarea con menor tiempo de holgura. En caso de que dos tareas tengan el mismo tiempo de holgura se seleccionara la que fue liberada primero. Esto evitará cambios de contexto innecesarios y reducirá la sobrecarga al procesador.

Existen dos variantes dependiendo del momento en el que se realiza la reevaluación.

- LST estricto: realiza la evaluación cada vez que el tiempo de holgura de la tarea actual es mayor que el de otra tarea en la pila provoca mayor sobrecargar.
- LST no estricto: realiza la evaluación cuando termina una tarea.

Debido a que requiere bastante información, se opta por utilizar otro algoritmo más sencillo, plazo más próximo primero (EDF, *earliest deadline first*).

3.4.2. EDF *(earliest deadline first*) plazo más próximo primero

Es mucho más sencillo que LST ya que no considera el tiempo de holgura sino solamente el plazo o tiempo de expiración. Esto permite que las evaluaciones sean más rápidas ya que solicita menos información a las tareas. La tarea con el plazo más cercano recibe la máxima prioridad.

Este algoritmo es óptimo sólo si:

- El sistema tiene solo un procesador
- El sistema tiene expropiación
- Las tareas no comparten recursos

3.4.3. Comprobación de planificadores cíclicos

Se les llaman planificadores cíclicos a todos aquellos que son periódicos. Es decir, sus tareas se ejecutan continuamente por un tiempo indefinido. Como se vio en el capítulo 2, es necesario conocer las propiedades de los planificadores y poder determinar un criterio de evaluación teórico cuando la simulación no es una opción confiable.

Las ecuaciones del capítulo 2 describen las condiciones básicas para un sistema.

Al ser cíclico el planificador es de interés determinar el periodo del mismo. A este periodo se le llama hiperperiódo. Para aumentar la eficiencia los planificadores es necesario modificar el tamaño de los marcos de tiempo en las cuales permiten que las tareas se ejecuten. Esta debe tener un tamaño óptimo, no muy pequeño para provocar demasiada sobrecarga en el procesador o muy grande como para retrasar los cambios de contextos.

Figura 37. **Diagrama Gantt de un sistema cíclico**

Fuente: elaboración propia, empleando Visio 2015.

El hiperperiódo es determinado por el mínimo común múltiplo del periodo de todas las áreas.

$$H = LCM\{P_1, P_2, P_3, \dots, P_n\}$$

Ecuación 4: definición del hiperperiódo

Luego de determinar el hiperperiódo es importante encontrar el tamaño óptimo para los marcos de planificación. Para sistemas periódicos estructurados no expropiativos, el método para encontrar el marco de planificación más grande consiste en una serie de tres evaluaciones sencillas:

1. $f > LCM\{P_1, P_2, P_3, \dots, P_n\}$

Ecuación 5: definición de marco de planificación

2. f es divisible en H
3. $2f - GCD(P_i, f) < D_i$

Ecuación 6: condición de validez para el marco de planificación óptimo

En caso de que alguna de las condiciones no se logre satisfacer, el planificador debe ser expropiativo.

3.5. Práctica 3: marcos de planificación, temporizadores y mutex

Esta sección contiene ejercicios prácticos para aplicación de planificadores dinámicos y su validación. También contiene ejercicios prácticos para la implementación de herramientas estudiadas como los temporizadores y los mutex.

3.5.1. Cálculo de marcos de planificación

Determine si existe un tamaño óptimo de marco de planificación para los siguientes planificadores.

Tabla II. **Planificador cíclico 1**

	Periodo (*P*)	**Ejecución (*e*)**	**Plazo (*D*)**
Tarea 1	15	1	14
Tarea 2	20	2	26
Tarea 3	22	3	

Fuente: elaboración propia.

Tabla III. **Planificador cíclico 2**

	Periodo (*P*)	**Ejecución (*e*)**	**Plazo (*D*)**
Tarea 1	4	1	
Tarea 2	5	2	7
Tarea 3	20	5	

Fuente: elaboración propia.

Tabla IV. **Planificador cíclico 3**

	Periodo (*P*)	**Ejecución (*e*)**
Tarea 1	5	0.1
Tarea 2	7	1
Tarea 3	12	6
Tarea 4	45	9

Fuente: elaboración propia.

Incluir detalladamente el procedimiento y los resultados de cada una de las pruebas. Luego, incluir una simulación de cada uno utilizando un planificador monotónico (RM) no expropiativo.

3.5.2. Manejo de temporizadores y mutex en FreeRTOS

La práctica consiste en un variador de frecuencia. Los requerimientos son los siguientes:

- 3 temporizadores cada uno controlará la frecuencia de cada uno de los ledes en la tiva.
- Utilizar los pulsadores uno para seleccionar led y el otro para variar el periodo del led seleccionado.
- Configurar como mínimo 4 periodos distintos.
- Cada cambio de periodo y de led debe ser desplegado en la consola serial de la computadora, mostrando el led seleccionado y el periodo seleccionado.

Nota: utilizar el ejemplo provisto en TivaWare como guía para la interacción con los ledes y el hardware.

4. FREERTOS EN ZYNQ-7000, COLAS DE MENSAJES, MEDIOS DE COMUNICACIÓN ENTRE TAREAS

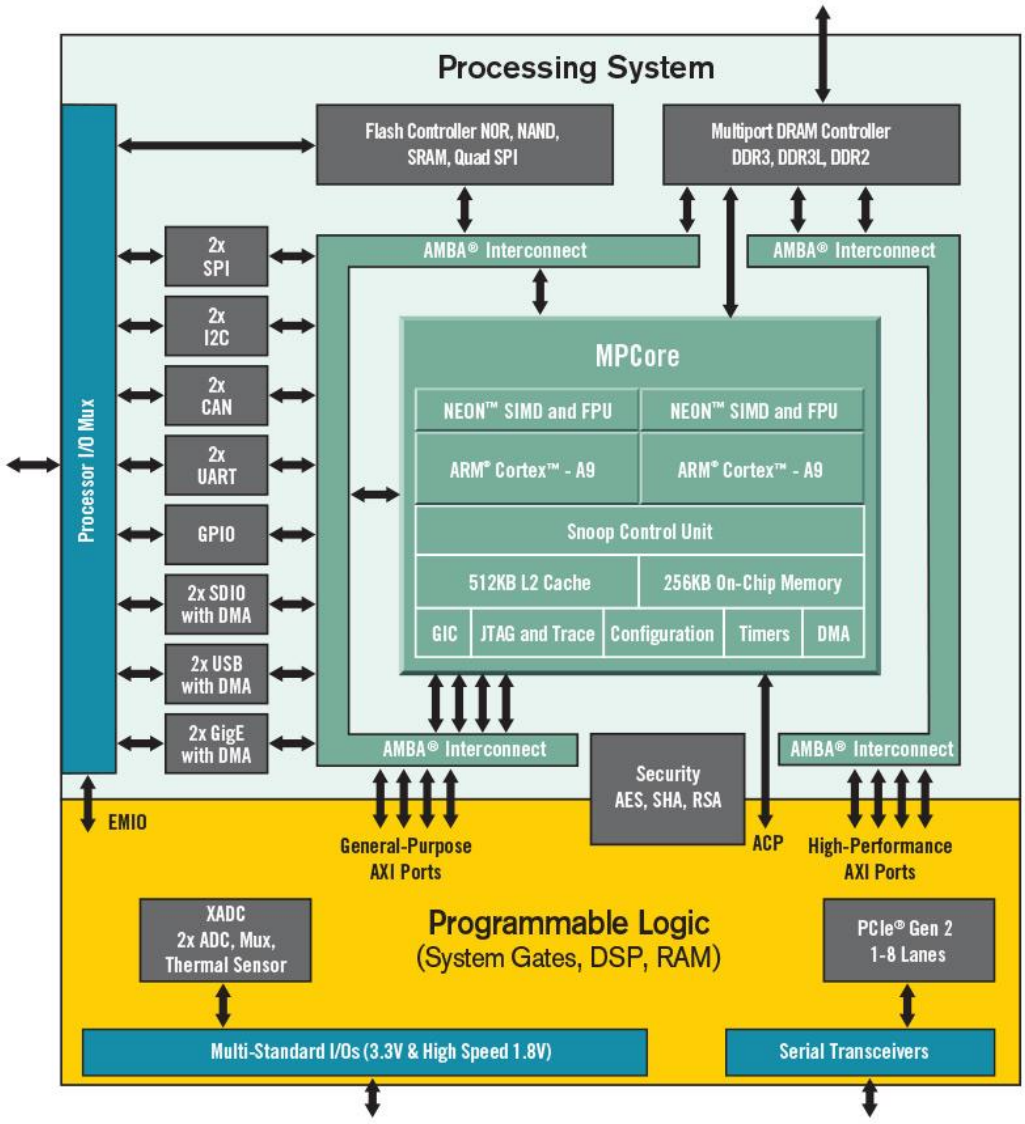
4.1. FreeRTOS en zynq 7000

La placa de desarrollo a utilizar es la zybo. Esta contiene un Soc (*system on chip,* sistema en chip) que incluye un procesador ARM A9 de dos núcleos y un FPGA (*field programmable gate array,* arreglo de compuertas programables) de la empresa Xilinx. La interacción entre ambos se realiza a través de buses *AXI* (*advanced extensible interface*, interfaz extensible avanzada) de alta velocidad, estos buses forman parte de la interconexión AMBA (*advanced microcontroller bus architecture,* arquitectura de bus avanzada para microcontroladores).

La AMBA es un grupo de especificaciones que forman el estándar para las comunicaciones dentro de sistemas en chip (SoC). AMBA es un estándar abierto para la conexión y la administración de bloques funcionales en sistemas en chip.

Las ventajas de los sistemas en chip es su alta flexibilidad. Al contener un FPGA, se les puede modificar la arquitectura con facilidad. Lo cual, hace que se puedan reducir los desperdicios al tener sistemas reutilizables y reprogramables.

Figura 38. **Diagrama interno del zynq 7000**



Fuente: *zynq-7000 A Generation Ahead Backgrounder Xilinx*. https://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf. Consulta: 14 de noviembre de 2017.

Es necesario especificar la arquitectura del microcontrolador antes de programarlo, por lo tanto, el flujo de trabajo es distinto. Cuando se trabaja en un

sistema embebido como la tiva, la arquitectura viene predefinida, pero en este caso, la arquitectura puede cambiar y es definida por el usuario.

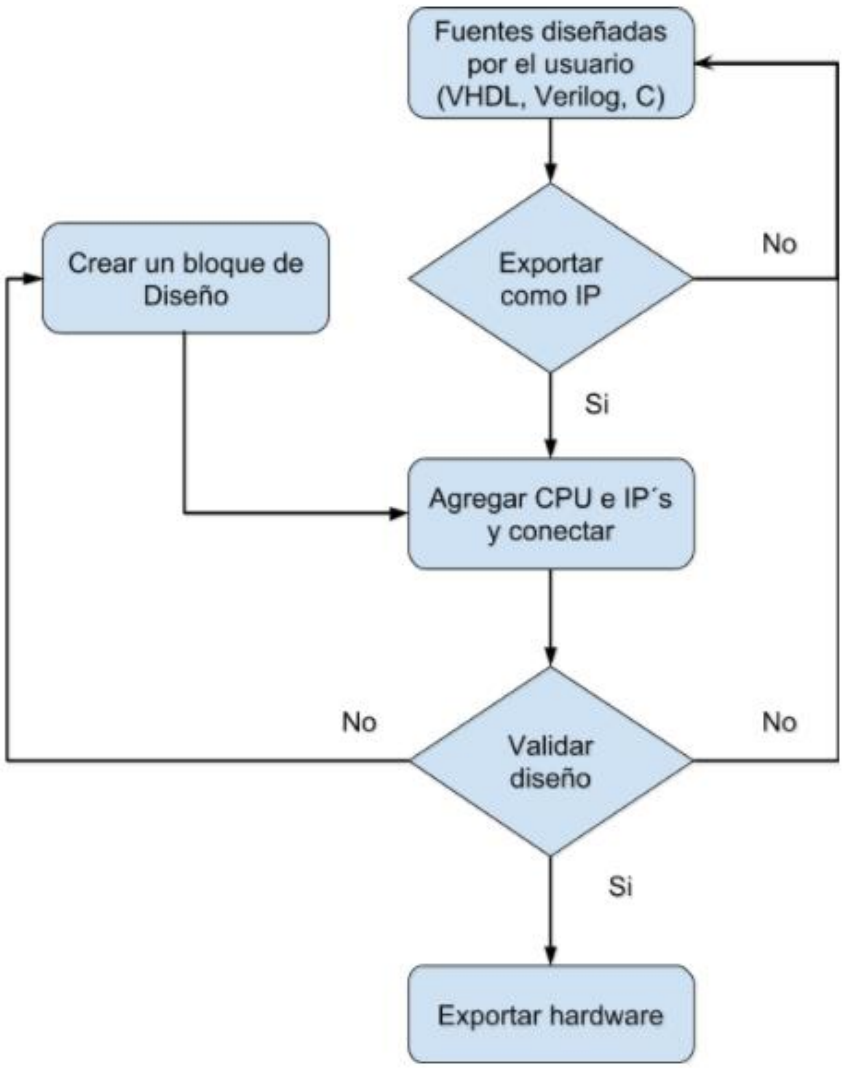
Esto permite más flexibilidad, pero también un grado mayor de complejidad. Por lo tanto, es necesario un entorno de desarrollo distinto.

4.1.1. Entorno de desarrollo

El entorno de desarrollo a utilizar es vivado. Este es provisto por el fabricante en su sitio web.

Vivado permite diseñar e implementar una arquitectura específica en el FPGA utilizando VHDL, Verilog y C (a través de HLS). Debido a esto el flujo de trabajo cambia y es necesario definir primero el hardware de nuestro microcontrolador.

Figura 39. **Diagrama de flujo de la creación de hardware en vivado**



Fuente: elaboración propia, empleando Visio 2015.

Una vez definido el hardware con el que se trabajara, vivado automáticamente importa el hardware para el entorno de programación del microcontrolador.

4.1.2. Crear un proyecto en vivado

Es muy importante considerar el orden del proceso de creación de un proyecto. Ya que vivado realiza revisiones de compatibilidad y validaciones de las opciones elegidas por el usuario.

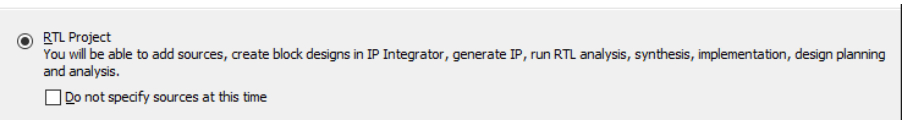
Primer paso: crear el proyecto

Figura 40. **Ventana de bienvenida de vivado**

Fuente: elaboración propia, empleando vivado.

Segundo paso: seleccionar un proyecto RTL

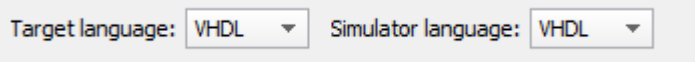
Figura 41. **Selección de tipo de proyecto**



Fuente: elaboración propia, empleando vivado.

Tercer paso: seleccionar el lenguaje a utilizar

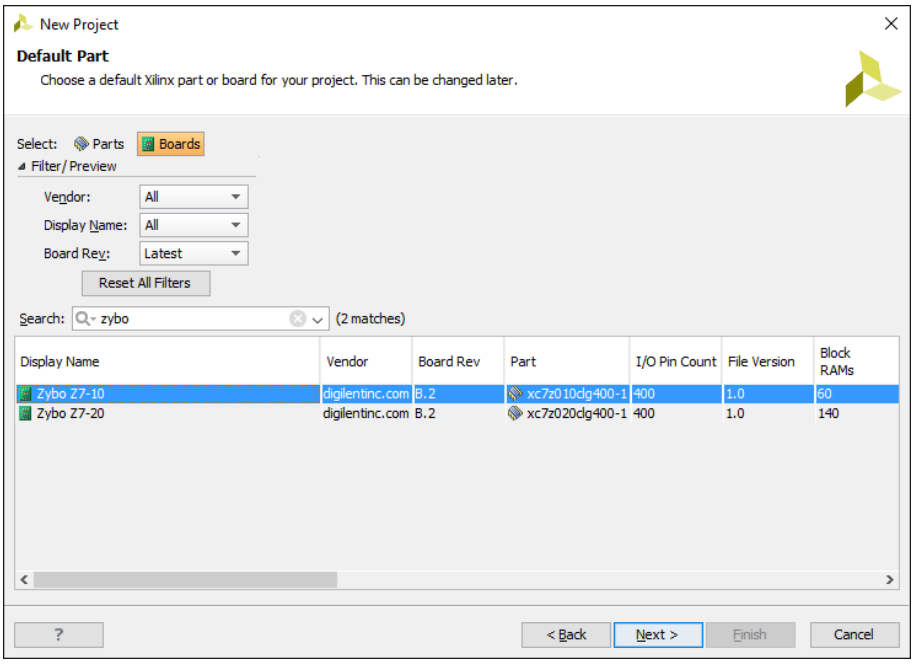
Figura 42. **Selección de lenguaje para el proyecto**



Fuente: elaboración propia, empleando vivado.

Cuarto paso: importar archivos previamente creados, agregar archivos de restricciones, por último, es necesario seleccionar la plataforma de desarrollo.

Figura 43. **Selección de plataforma de desarrollo**

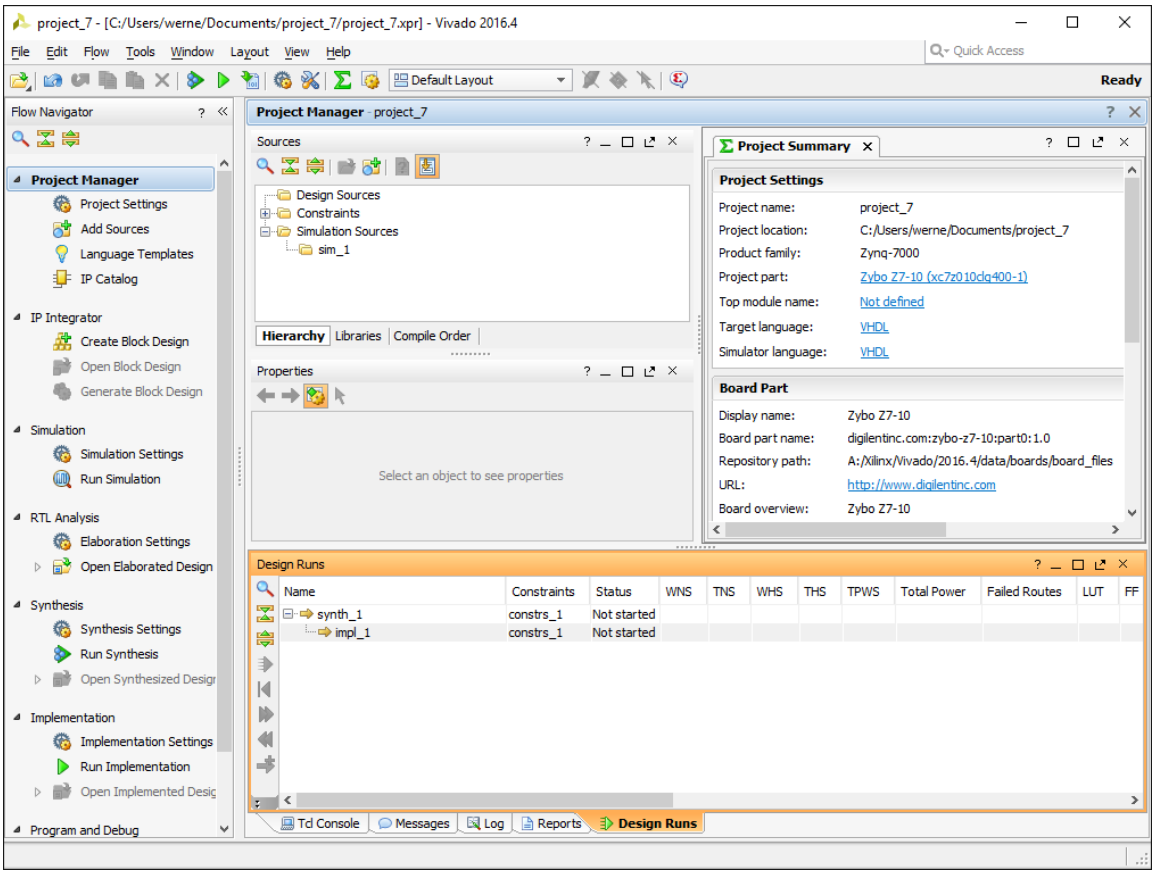


Fuente: elaboración propia, empleando vivado.

Al finalizar se crea una carpeta con el nombre dado del proyecto. Ahora es necesario ver el proceso para la realización de un programa.

Ya creado el proyecto este puede ser abierto en vivado para editar el hardware y el software. Al abrir el proyecto en vivado, se despliega la siguiente ventana.

Figura 44. **Proyecto abierto para edición en vivado**



Fuente: elaboración propia, empleando vivado.

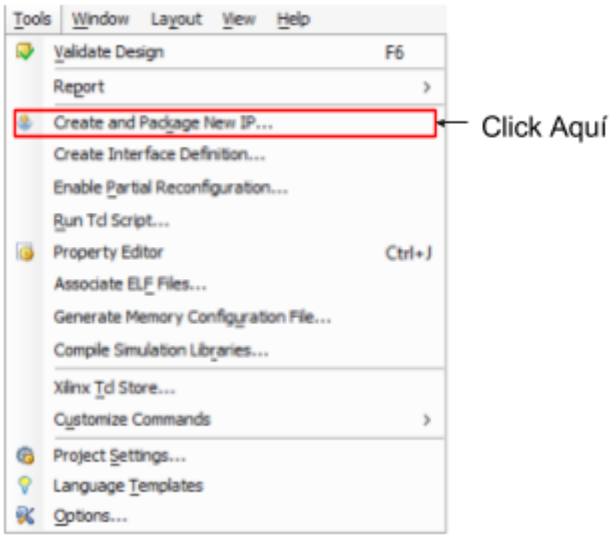
Vivado provee un grupo de herramientas para la edición del proyecto, dependen de lo que se esté editando.

4.1.3. Definir hardware en vivado

Es importante considerar que cada bloque IP se crea externamente y luego se importan como entidades al proyecto en el cual se implementaran. Estás IP pueden describirse externamente.

Vivado provee una herramienta para diseñar las IP's sin abandonar el proyecto en el cual son necesarias. En el caso en el que se desee implementar una IP ya creada, esta puede importarse al crear el proyecto.

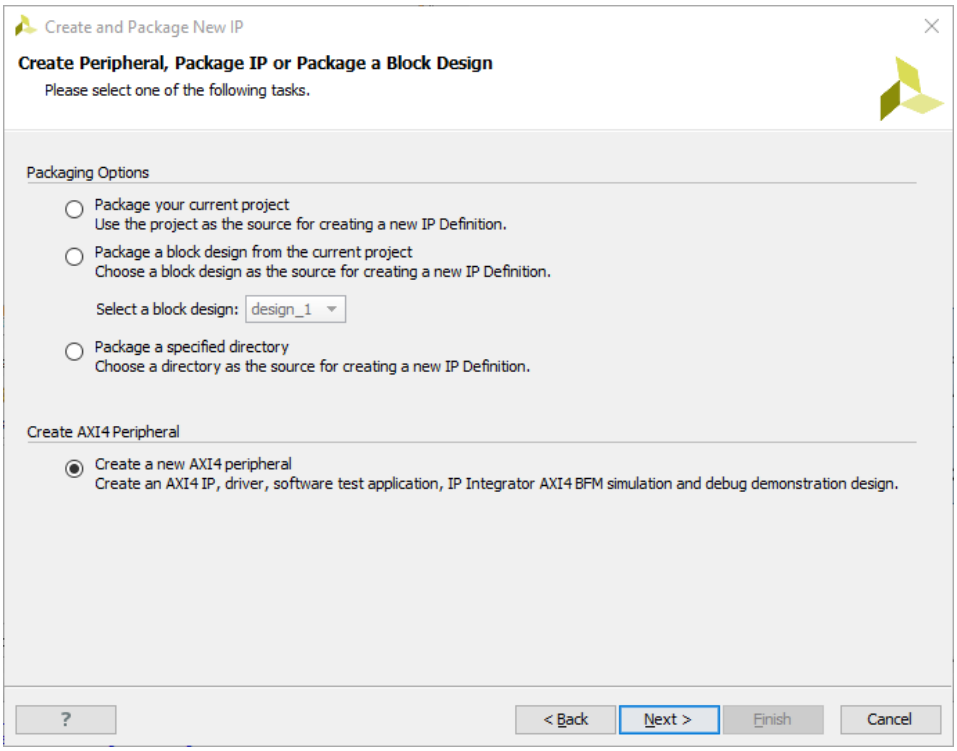
Figura 45. **Menú de herramientas y selección de la opción para creación de IP**



Fuente: elaboración propia, empleando vivado.

Luego de haber seleccionado la opción anterior se despliega la ventana de la figura 46:

Figura 46. **Menú para la creación de IP**



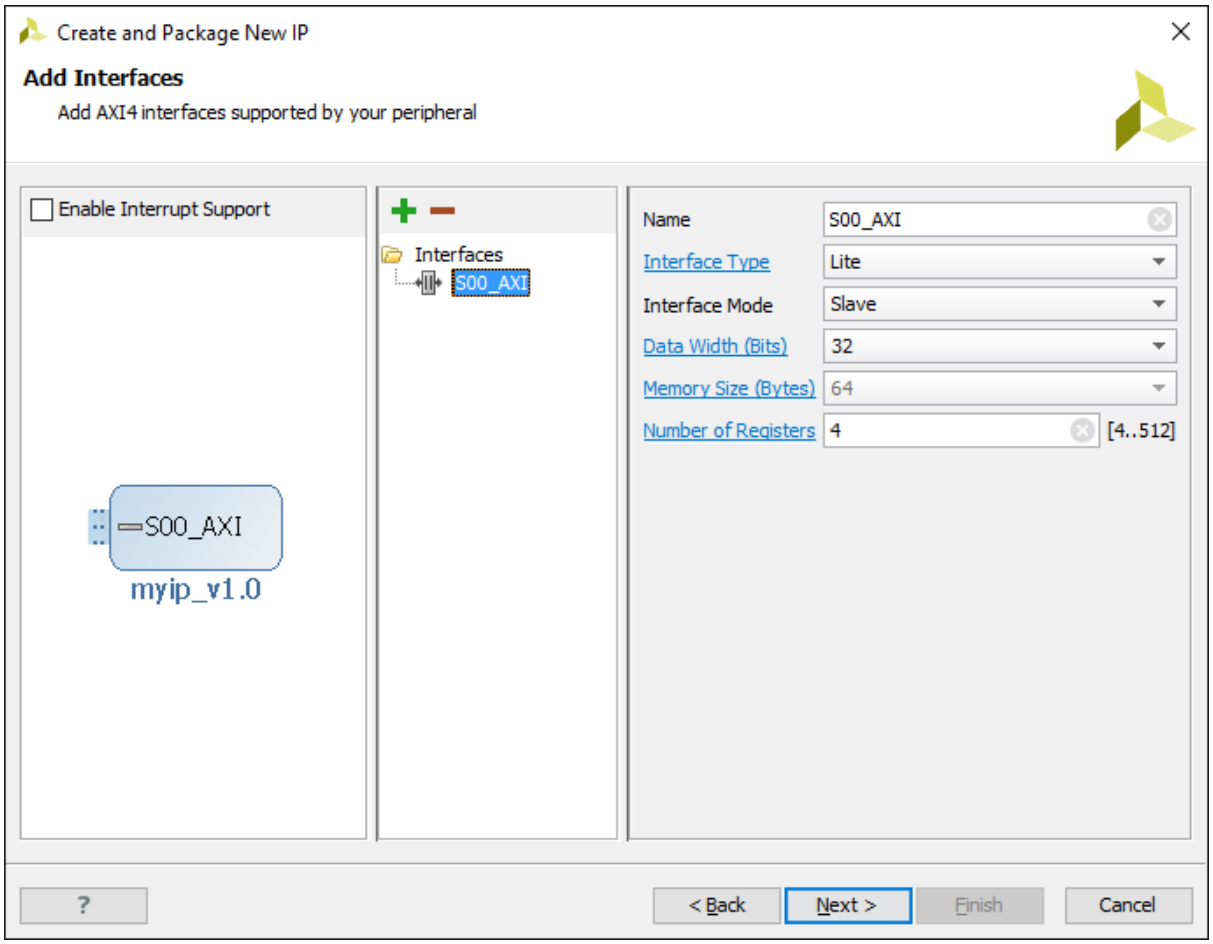
Fuente: elaboración propia, empleando vivado.

Esta ventana permite exportar el proyecto actual como un IP, así como seleccionar un bloque de diseño para exportar, permite exportar un directorio y también permite crear un nuevo periférico.

Para la creación de un nuevo IP es necesario definir un nuevo periférico, por lo tanto, se selecciona la última opción en este caso.

Al seleccionar la opción para creación de nueva IP, vivado despliega la ventana para selección de interfaces. Estas interfaces serán las que utilizará el microcontrolador para comunicarse con el módulo o para comunicación entre módulos.

Figura 47. **Ventana para creación de interfaces**



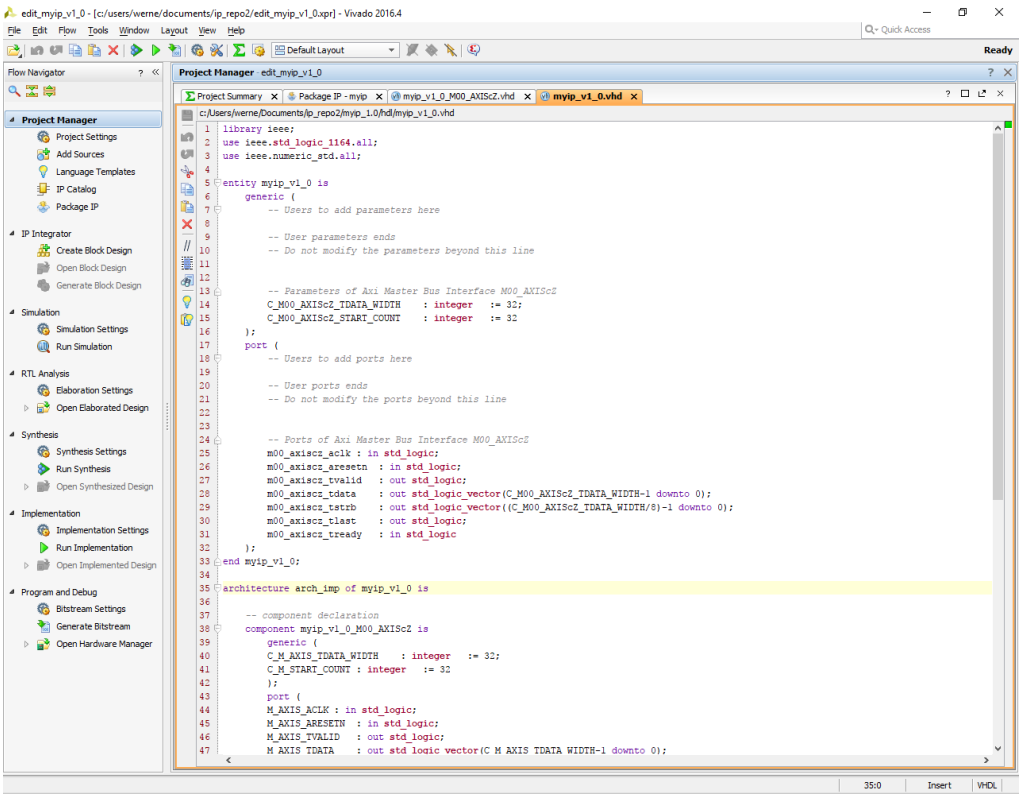
Fuente: elaboración propia, empleando vivado.

En esta sección se pueden configurar una interfaz de interrupción, se conecta automáticamente a la interfaz de interrupciones del microcontrolador. Los distintos parámetros son descritos en detalle a continuación:

Nombre	El nombre dado al módulo que se creara. Con este nombre aparecerá en el catálogo de IP's luego de ser importado.
Tipo de interfaz	*Lite*: para una tasa de transferencia baja de datos en memoria, como registros. *Full*: para transmisiones de alta velocidad de datos en memoria. *Stream*: para transmisiones continuas de datos donde no es necesario una confirmación de recibido.
Modo de la interfaz	Si la interfaz es esclavo o maestro. Esto indica el comportamiento de la interfaz. En esclavo, esta espera que el maestro le solicite los datos antes de transmitirlos. En maestro, transmite datos todo el tiempo.
Ancho de datos	Está determinado por la arquitectura del microprocesador. En este caso el ARM A9 tiene registros de 32 bits, por lo tanto, el ancho de los datos está fijo en este valor.
Tamaño de memoria	Solo disponible para las interfaces de tipo Full. Determina el tamaño de la memoria disponible en forma de arreglo para el módulo.
Número de registros	Solo disponible para las interfaces de tipo Lite. Determina la cantidad de registros del módulo

Al terminar de configurar las interfaces basta con dar clic en siguiente y seleccionar editar IP.

Figura 48. **Ventana para edición de IP**



Fuente: elaboración propia, empleando vivado.

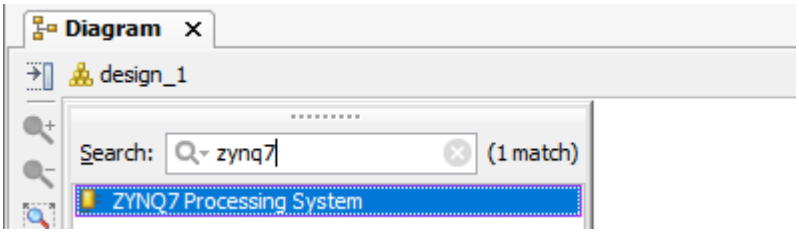
Se puede proceder a modificar el comportamiento del módulo. Como en cualquier módulo, es necesario sintetizarlo antes de poder utilizarlo.

4.1.4. FreeRTOS en vivado

En esta sección se implementará un proyecto de FreeRTOS en la zybo utilizando vivado.

- Crear un proyecto siguiendo las restricciones del ejemplo anterior.
- Crear un bloque de diseño.
- Agregar el módulo de procesamiento (CPU) al bloque de diseño.

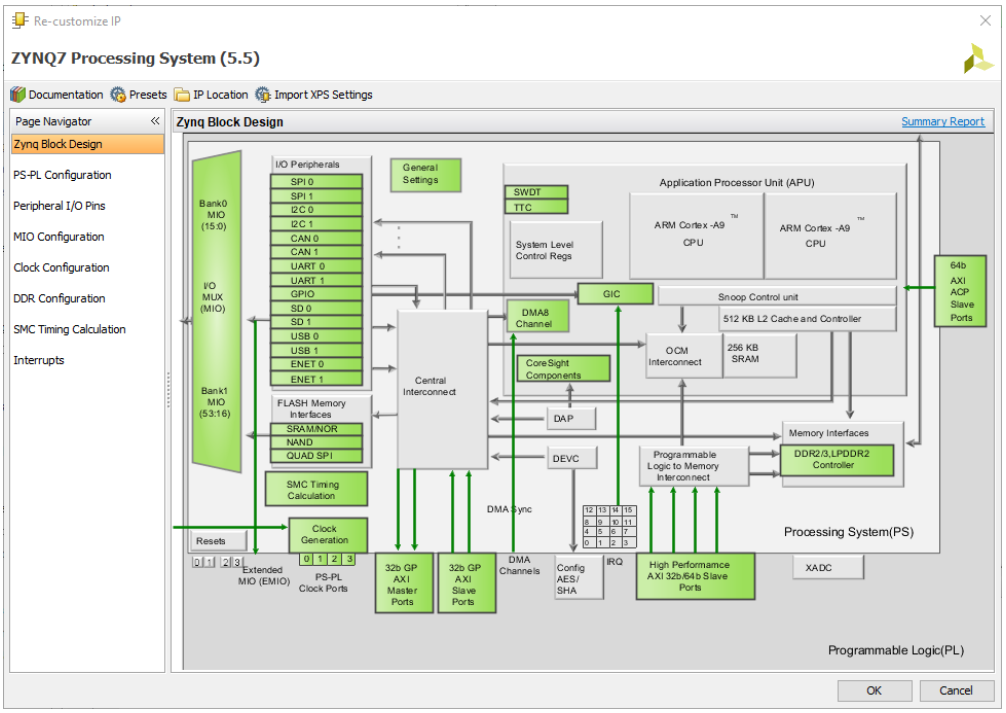
Figura 49. **IP del módulo de procesamiento para zybo**



Fuente: elaboración propia, empleando vivado.

Para personalizar el bloque de procesamiento solo dar doble clic sobre el bloque. Permite modificar los buses de entrada y salida. Permite configurar los periféricos del procesador, interrupciones, relojes, entre otros. Agregar las IP's deseadas.

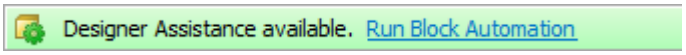
Figura 50. **Ventana para modificación del módulo de procesamiento**



Fuente: elaboración propia, empleando vivado.

Correr automatización de bloque (*run block automation*). Esta opción permite conectar de forma automática las interfaces relacionadas en la configuración de cada módulo.

Figura 51. **Opción de asistencia de diseño**



Fuente: elaboración propia, empleando vivado.

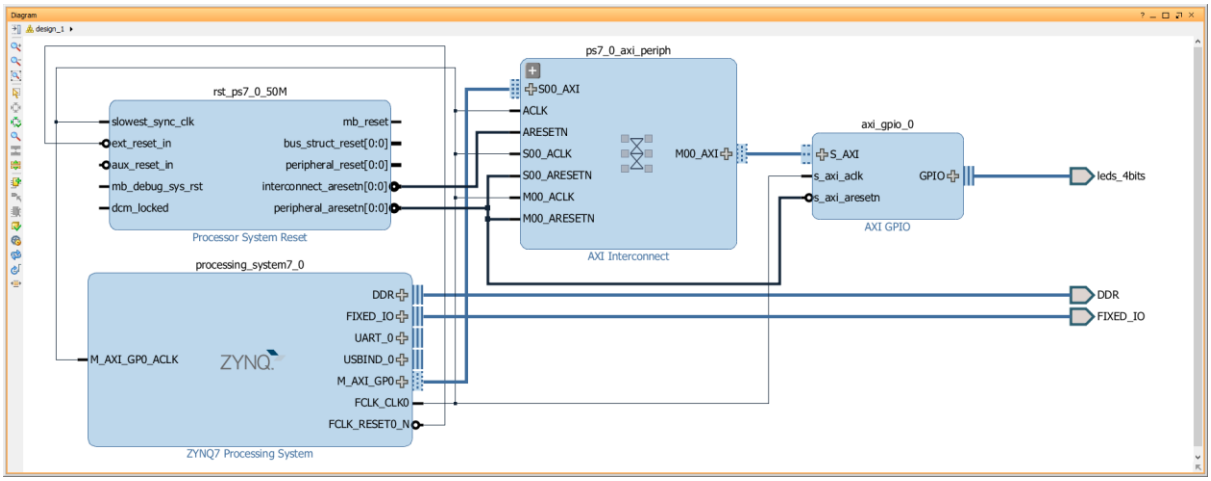
Agregar las IP's deseadas. En este ejemplo se agregó un módulo de propósito general (AXI GPIO) el cual se configuró para conectarse a los ledes en la placa de desarrollo. Aparecen dos asistencias de diseño.

Figura 52. **Asistencias de diseño**

Fuente: elaboración propia, empleando vivado.

La primera automatiza los bloques, la segunda crea automáticamente las rutas entre cada bloque luego de que sus interfaces fueron configuradas para su interconexión. Luego de correr ambas opciones el diseño se actualiza. Vivado agrega automáticamente dos bloques, uno de multiplexores para las señales AXI de los módulos y otra con multiplexores para las señales de *reset*.

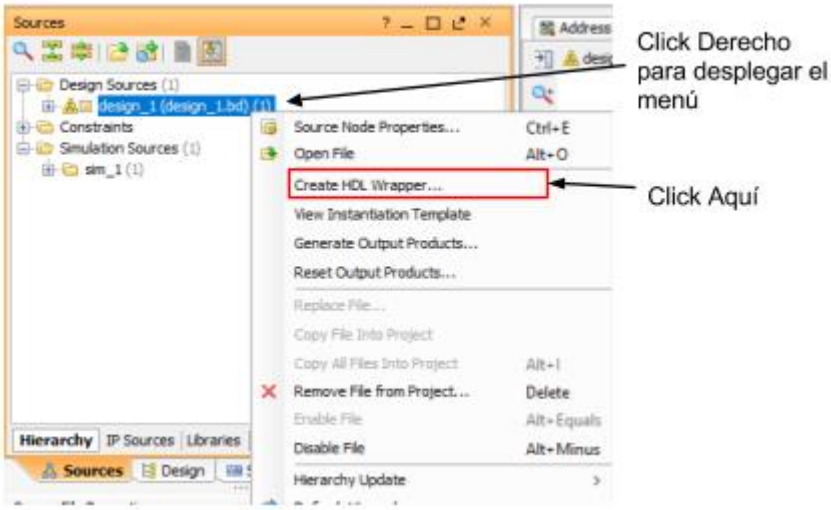
Figura 53. **Diagrama de diseño**



Fuente: elaboración propia, empleando vivado.

Validar el diseño (con la tecla F6), en caso de errores regresar a la configuración. Ya validado el diseño es necesario definir la implementación del módulo. Esto se hace creando una envoltura HDL.

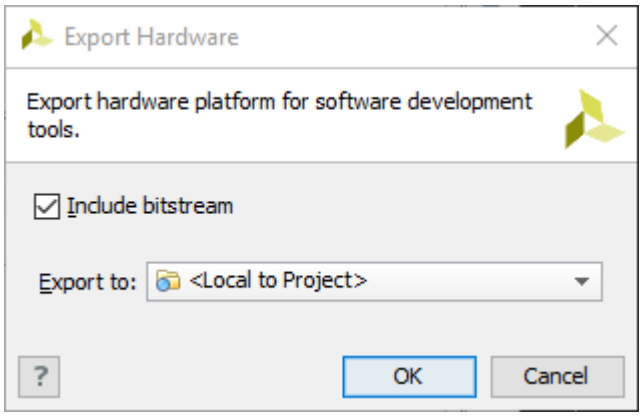
Figura 54. **Creación de HDL *wrapper***



Fuente: elaboración propia, empleando vivado.

Sintetizar y generar el archivo para configuración de la FPGA. Luego se exporta el hardware incluyendo el *bitstream* dando clic en archivo, exportar, exportar hardware.

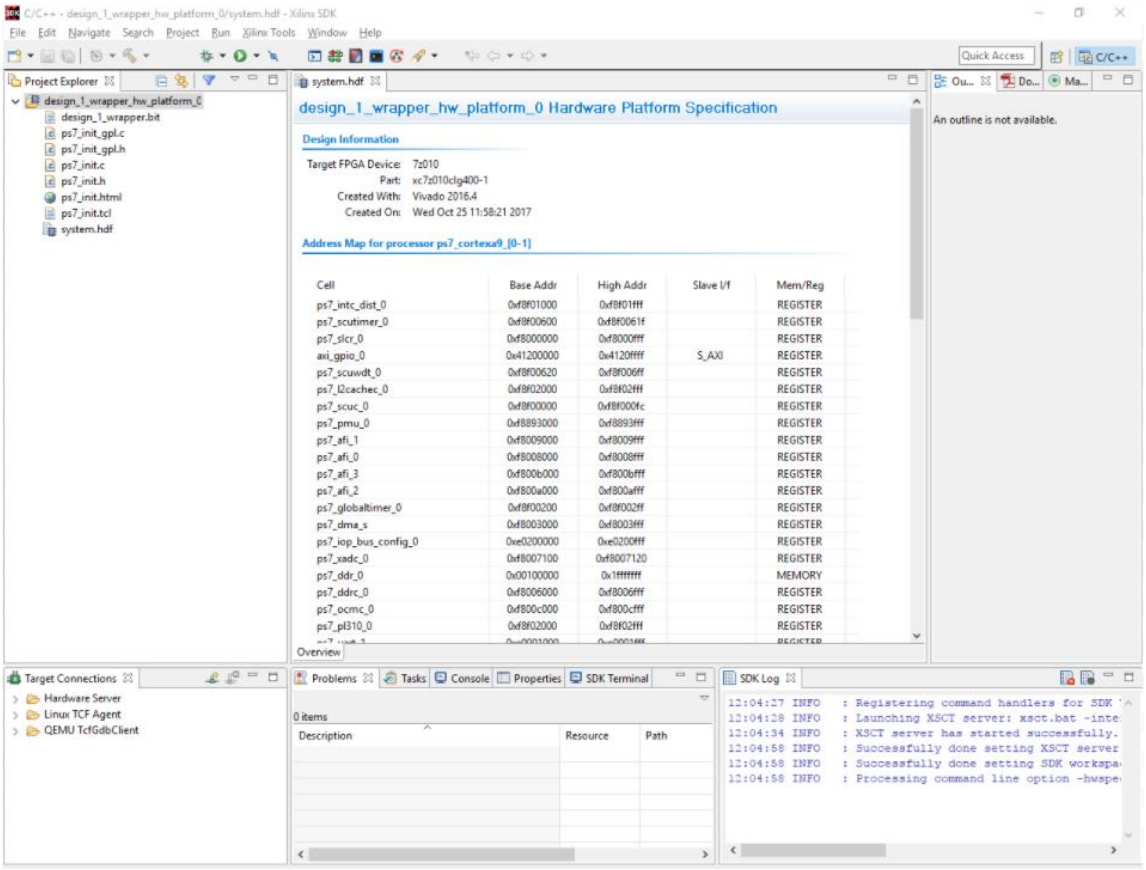
Figura 55. **Exportar el hardware**



Fuente: elaboración propia, empleando vivado.

Al finalizar los pasos anteriores ya se tiene un *hardware* definido lo cual permite al usuario programar el hardware. Para esto es necesario ejecutar el SDK de vivado. Este se encargará de cargar los archivos generados previamente. Vivado crea de forma automática los controladores para el hardware creado de manera que el procesador solo los ve como periféricos. Para comunicarse con los módulos el procesador utiliza los registros definidos al crear las interfaces de los módulos.

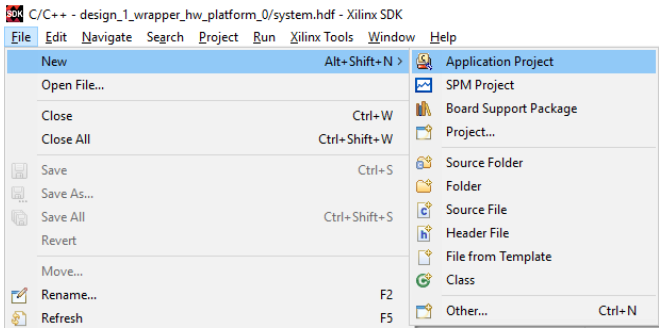
Figura 56. **Ventana de inicio del kit de desarrollo de software de vivado**



Fuente: elaboración propia, empleando vivado.

Al abrirse el SDK nuestra arquitectura es importada. El SDK permite crear aplicaciones en *bare-metal* lo cual significa que se programa directamente en el microcontrolador o también permite que se utilicen sistemas operativos. En este caso se utilizará FreeRTOS. Para crear un proyecto con FreeRTOS es necesario crear un *application project.*

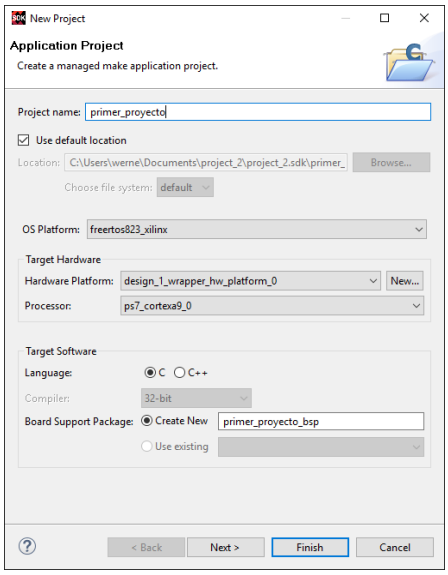
Figura 57. **Creación de un proyecto**



Fuente: elaboración propia, empleando vivado.

Al seleccionar el *application project* se muestra una ventana con las especificaciones para el proyecto.

Figura 58. **Configuración de un nuevo proyecto**

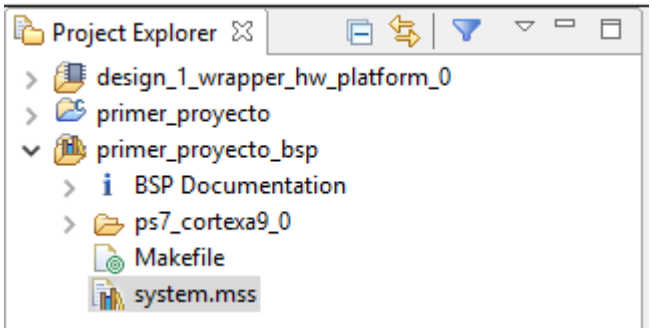


Fuente: elaboración propia, empleando vivado.

Esta ventana permite seleccionar el lugar donde se guardarán los archivos del proyecto. Permite seleccionar un sistema operativo para la aplicación. La plataforma de hardware para la cual se desarrolla el hardware puede ser una predefinida por Xilinx, o en este caso la definida en los pasos anteriores. Debido a que la placa de desarrollo tiene dos procesadores, el SDK permite seleccionar el que se utilizará para la aplicación, así como el lenguaje de programación. El *BSP* lo crea el SDK a partir del diseño HDL.

Dando clic en finalizar el SDK carga los archivos necesarios. En este punto ya se tiene FreeRTOS en el hardware definido por el usuario.

Figura 59. **Jerarquía del proyecto creado**



Fuente: elaboración propia, empleando vivado.

Una donde están todas las especificaciones del hardware. Al trabajar con un FPGA se puede modificar el hardware y se actualizará todo el flujo de trabajo luego de validar los cambios y aplicarlos. Es importante indicar que una vez utilizados los registros en el archivo de programación habrá conflictos si en al modificar el hardware los registros desaparecen. Por lo tanto, al modificar hardware lo mejor es validar cada paso del flujo de trabajo nuevamente.

Otra carpeta donde estarán los archivos de la aplicación como tal. Estos incluyen las librerías, controladores y todos los archivos necesarios para correr la aplicación en FreeRTOS sobre la plataforma diseñada.

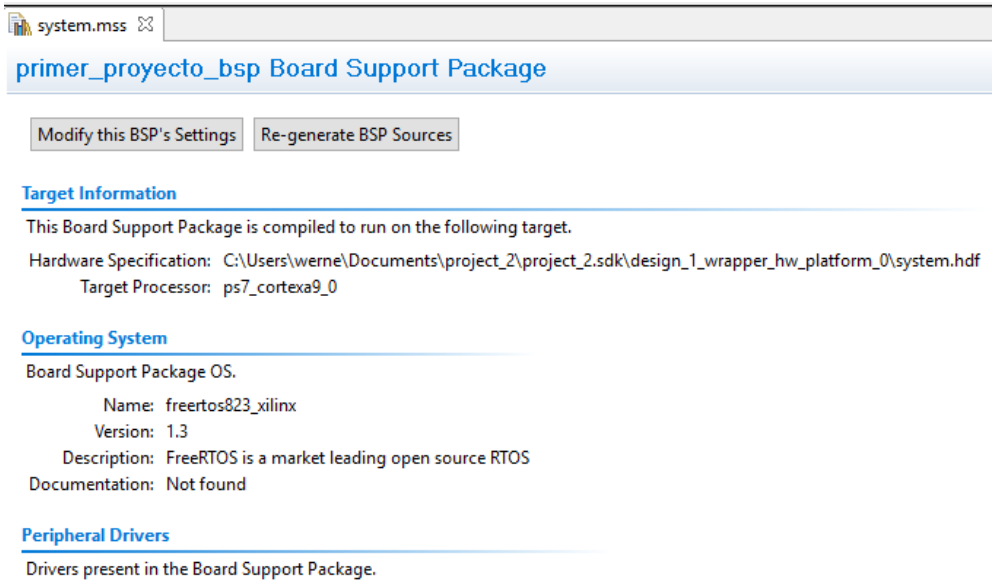
La tercera carpeta contiene el paquete de soporte para la placa. En esta están incluidos todos los drivers generados automáticamente por vivado, así como las configuraciones para los dispositivos y para el sistema.

4.1.5. Configurar FreeRTOS en vivado

Para modificar los parámetros del archivo *FreeRTOSConfig.h* es necesario modificar los parámetros en el archivo *system.mss*. Este contiene todos los

parámetros de configuración de la placa. Al seleccionarlo se despliega toda su información una ventana.

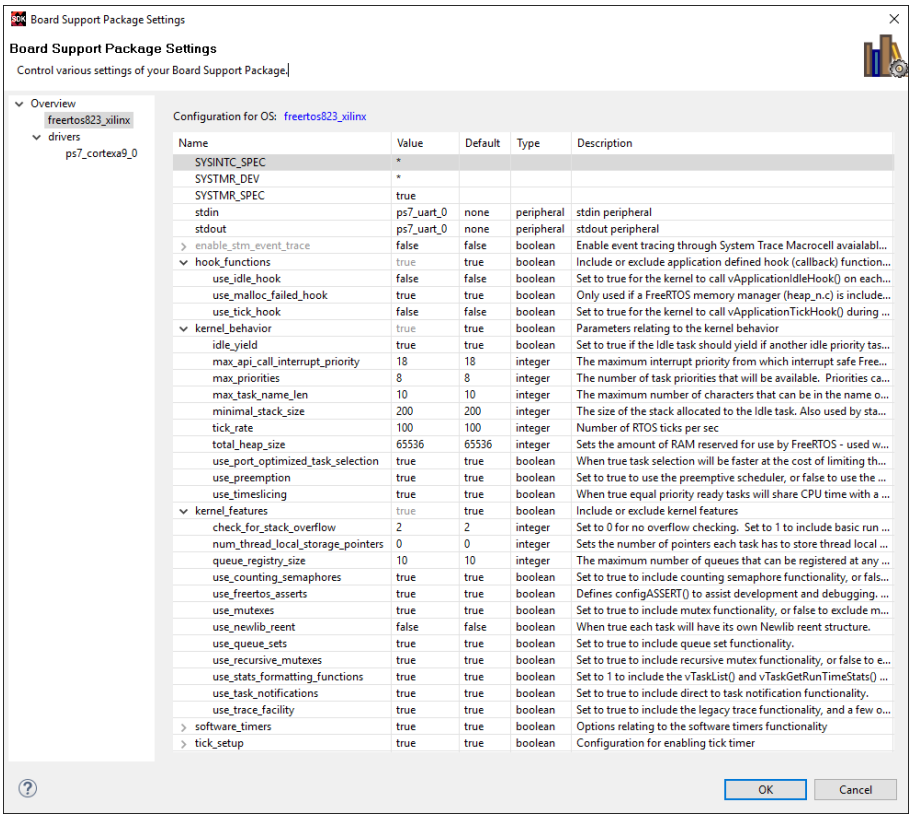
Figura 60. **Archivo system.mss**



Fuente: elaboración propia, empleando vivado.

Este archivo contiene toda la información referente al sistema. La información de la placa de desarrollo, la información del sistema operativo, y la información de todos los periféricos disponibles está descrita en este archivo. Lo más importante de este archivo son los dos botones al tope. En el botón de modificación se accede a la ventana con los parámetros.

Figura 61. **Parámetros de configuración de FreeRTOS y de los controladores**



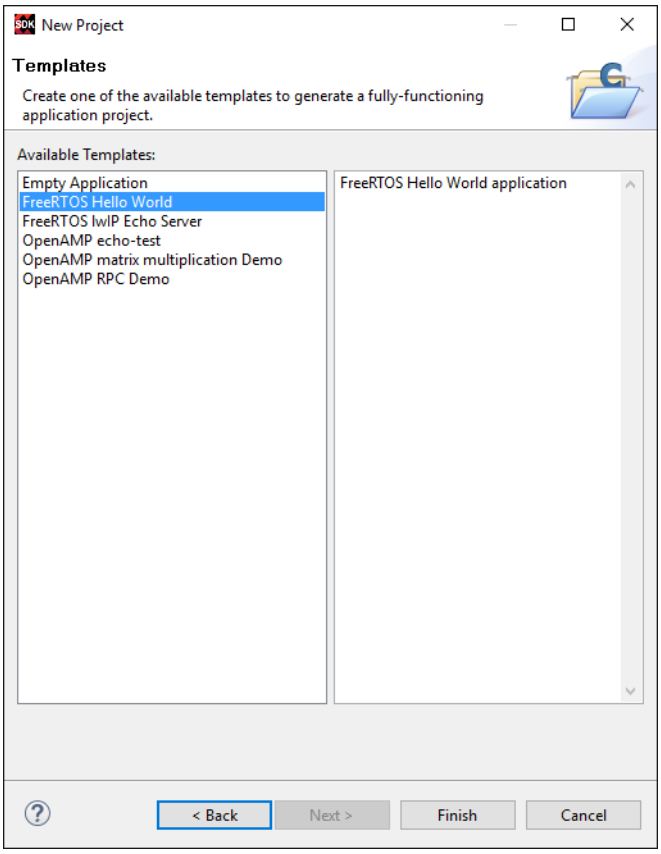
Fuente: elaboración propia, empleando vivado.

Al modificar un parámetro en esta ventana es necesario regenerar el BSP para aplicar los cambios.

4.2. Hola mundo en vivado

Al crear un nuevo proyecto de aplicación vivado permite seleccionar entre proyecto vacío y algunos ejemplos disponibles para la plataforma seleccionada.

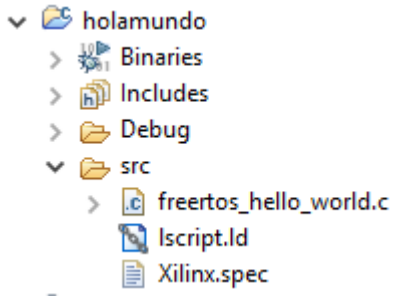
Figura 62. **Selección de tipo de proyecto para un nuevo proyecto de aplicación**



Fuente: elaboración propia, empleando vivado.

Seleccionando la opción de *FreeRTOS hello world* vivado extrae los archivos y los copia en el proyecto creado. Debido a que el hardware seleccionado es el mismo que para el proyecto de ejemplo el BSP no cambia nada. Los archivos específicos de *FreeRTOS hello world* se encuentran en la carpeta de *src* del mismo proyecto.

Figura 63. **Jerarquía de archivos en un proyecto**



Fuente: elaboración propia, empleando vivado.

Al abrir el archivo *freertos_hello_world.c* lo primero que se puede notar que difiere de los proyectos hasta ahora vistos en FreeRTOS son las librerías extras. *xil_printf.h* contiene la definición de las funciones para interactuar con el UART. *xparameters.h* contiene los parámetros del hardware, estos están descritos en el *system.mss.*

El archivo *freertos_hello_world.c* contiene la siguiente línea:

xil_printf (“Hello from Freertos example main\r\n");

La cual llama a la función que vivado utiliza como abstracción para el UART, la función que se encarga de escribir en el UART es:

XUartPs_SendByte ()

Para leer del UART vivado provee la siguiente función:

XUartPs_RecvByte ()

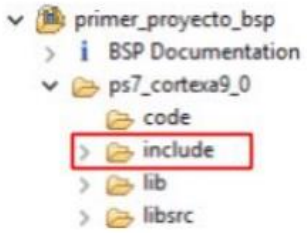
Estas están definidas dentro del archivo *xil_printf.h* donde se definen a su vez dos funciones externas inbyte y outbyte. La primera para leer y la segunda para escribir. La tasa de transmisión por defecto es 11520.

4.3. Interactuando con GPIO's

Para interactuar con pines de uso general en el procesador es necesario definirlos en la arquitectura del mismo. Vivado crea un archivo de controladores para manejar los registros que el microprocesador utilizara para comunicarse con el módulo GPIO.

El archivo que contiene las funciones es *xgpio.h* este archivo se encuentra en la carpeta de *include* del BSP.

Figura 64. **La carpeta *include* del BSP contiene todas las librerías de controladores**



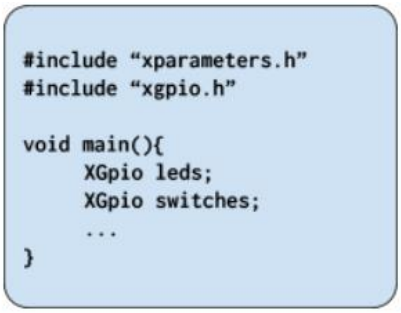
Fuente: elaboración propia, empleando vivado.

Pasos para leer o escribir un GPIO

- Crear una instancia: de acuerdo a la información provista dentro del archivo *xgpio.h* el usuario debe crear variables de este tipo para cada dispositivo GPIO en el sistema. Luego, un puntero es entregado a las

funciones del controlador. Para definir una variable de tipo xgpio es necesario incluir la librería.

Figura 65. **Creación de variables tipo XGpio**



```
#include “xparameters.h”
#include “xgpio.h”

void main(){
     XGpio leds;
     XGpio switches;
     ...
}
```

Fuente: elaboración propia.

- Inicializar el GPIO: la función que provee xgpio.h para inicializar un dispositivo es:

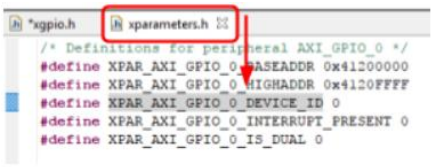
(int) XGpio_Initialize(XGpio *InstancePtr, u16 DeviceID);

InstancePtr: es el puntero que se utilizará como manipulador del componente. Corresponde a las variables definidas previamente por el usuario.

DeviceID: es un identificador único para el dispositivo. Este identificador asocia la instancia *XGPIO* con un dispositivo específico elegido por el programador. El identificador de cada dispositivo se puede encontrar en el archivo *xparameters.h* en la misma carpeta que *xgpio.h.*

@return: la función retorna un XST_SUCCESS si la inicialización fue exitosa y un XST_DEVICE_NOT_FOUND si no se encontró el dispositivo, ambos valores están definidos dentro de *xstatus.h* .

Figura 66. **Identificadores de dispositivos en *xparameters.h***



Fuente: elaboración propia, empleando vivado.

- Configurar la dirección del GPIO: ya inicializados los puertos GPIO a utilizar es necesario configurar como entrada o salida, esto se hace con la función:

```
void XGpio_SetDataDirection(XGpio *IntancePtr,
                            unsigned Channel,
                            u32 DirectionMask);
```

IntancePtr: corresponde al puntero que actúa como manipulador de la instancia de XGpio en la cual opera la función.

Channel: indica el canal del GPIO en el que opera (1 o 2).

DirectionMask: una máscara del tamaño de un registro, especifica que bits son entradas y cuales son salidas. Se especifican las salidas con 0 y las entradas con un 1.

- Leer datos: *xgpio.h* provee la función:

```
U32 XGpio_DiscreteRead(XGpio *InstancePtr, unsigned Channel);
```

InstancePtr: corresponde al puntero que actúa como manipulador de la instancia de XGpio en la cual opera la función.

Channel: el canal del dispositivo a leer.

@return: retorna el valor de la lectura en un registro de 32 *bits* sin signo. En el caso en el que se retorna más de un dato en el registro se pueden utilizar corrimientos y máscaras para seleccionar sólo los *bits* de interés.

- Escribir en un GPio: análoga a la función de lectura, se tiene:

U32 XGpio_DiscreteWrite(XGpio *InstancePtr, unsigned Channel,
u32 Data);

InstancePtr: corresponde al puntero que actúa como manipulador de la instancia de XGpio en la cual opera la función.

Channel: el canal del dispositivo en que se escribe.

Data: los datos a escribir en un formato de 32 *bits.*

4.4. Tareas aperiódicas

Particularmente en los sistemas operativos en tiempo real no todas las tareas son periódicas. Esto se debe a que usualmente existen fuentes externas que interactúan de forma impredecible con el procesador.

Usualmente es mejor completar las tareas no periódicas lo más pronto posible para tener una mejor interacción con el exterior. Existen métodos que

permiten reducir el tiempo de ejecución de las tareas no periódicas sin afectar las tareas periódicas.

El algoritmo más sencillo es el que espera a que el procesador esté libre para ejecutar las tareas no periódicas, donde las tareas no periódicas se ejecutan conforme llegan.

Ya que en los sistemas en tiempo real el tiempo de ejecución no es importante sino solamente que la tarea se complete antes del plazo de vencimiento, entonces se puede modificar el planificador para que ejecute las tareas periódicas lo más tarde posible. Esto permite espacio para tareas no periódicas antes. Este algoritmo de reorganización de tareas es llamado *slack stealing*.

4.4.1. Planificador latest release time (LTR)

Este planificador es completamente dinámico y utiliza *slack stealing.* Ya que el rendimiento de un RTOS no depende del momento en el que se completen las tareas, es posible esperar hasta el último momento para completarlas, siempre y cuando se ejecuten antes del plazo de vencimiento. Este planificador es el utilizado por defecto en estudiantes, se busca maximizar el tiempo de ocio empujando las tareas al último instante posible.

Para este algoritmo de planificación los tiempos de liberación de las tareas se toman como los plazos de vencimiento, y los plazos de vencimiento como los instantes de liberación de las tareas. Las tareas se calendarizan al revés.

Por ejemplo: considerando un planificador LTR en un sistema expropiativo las siguientes tareas.

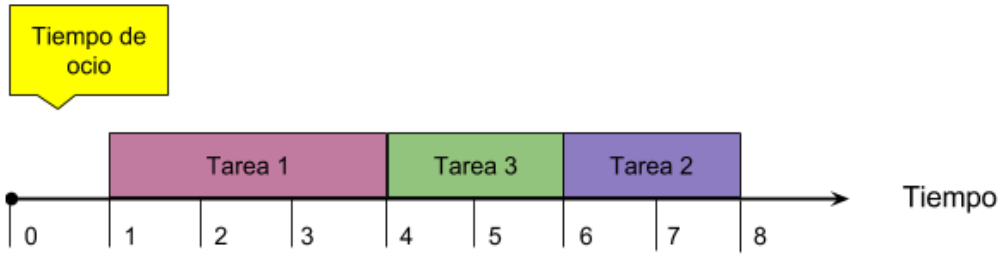
Tabla V. **Planificador LTR**

	Liberación (r)	**Ejecución (e)**	**Plazo (D)**
Tarea 1	0	3	6
Tarea 2	5	2	8
Tarea 3	2	2	7

Fuente: elaboración propia.

En un espacio de 8 el planificador tomará como prioridad la tarea con el tiempo de liberación más lejano, en este caso la tarea 2. Esta será calendarizada en 6 para completarse justo en el plazo de vencimiento. Luego, se tiene la tarea 3 la cual se libera en 2 y su plazo se cumple en 7. Ya que 6 está ocupado por la tarea 2, la tarea 3 se planifica de 4 a 6. Por último, la tarea 1 se libera en 0 y se calendariza lo más tarde posible, de 1 a 3. De esta forma el tiempo de ocio queda al inicio y una tarea no periódica se ejecutaría desde 0 para reducir su tiempo de respuesta.

Figura 67. **Ejemplo de planificador LTR**



Fuente: elaboración propia, empleando Visio 2015.

4.4.2. Servidor retardador (*deferrable server*)

Los servidores retardadores proveen una herramienta para poder disminuir el tiempo de respuesta del sistema ante tareas no periódicas. Usualmente las tareas no periódicas se les da la prioridad más baja para evitar que interrumpan las tareas periódicas. Pero, este acercamiento provoca retrasos innecesarios y grande a las tareas no periódicas.

El tiempo de respuesta se puede mejorar utilizando una tarea de sondeo. Esta tarea tiene un periodo fijo y la máxima prioridad. La tarea también tiene un presupuesto de CPU, esto le permitirá ejecutar tareas no periódicas. Esta tarea es el llamado servidor retardador.

Su única función es determinar si hay tareas no periódicas y si tiene suficiente espacio en el CPU como para ejecutarlas. El presupuesto de un servidor retardador se terminar de acuerdo a dos reglas.

- Regla de consumo: el presupuesto del servidor es consumido una unidad por unidad de tiempo cuando una tarea no periódica se está ejecutando.

- Regla de Provisión: el presupuesto el igual al tiempo de ejecución dado al servidor al inicio de cada periodo. El presupuesto se restablece en cada periodo.

Para diseñar el servidor es necesario determinar el periodo y el tiempo de ejecución, que corresponde al presupuesto.

- Es necesario determinar la utilización total del sistema con la ecuación 2.
- Seleccionar el periodo (P_s) y el tiempo de ejecución (e_s).

- Determinar la utilización del servidor:

$$U_s = \frac{e_s}{P_s}$$

Ecuación 6: utilización del servidor retardador

- Comprobar que la utilización total de sistema más la utilización del servidor sea menor que el *Urm*.
- En caso de que no se satisfaga la condición regresar al paso 2.

4.5. Práctica 4: servidor retardador y *blinking led* en FreeRTOS

En esta sección se muestran ejercicios prácticos, se incluyen dos proyectos. En los cuales es imperativo comprender los temas anteriores, manejo de gpio´s, temporizadores, configuración de FreeRTOS, proyectos en vivado, entre otros.

4.5.1. Servidor retardador

La práctica consiste en:

- Implementar la funcion matrix_task.
- Crear un temporizador con un periodo de 5 segundos y reinicio automático.

```
void vTimerCallback(TimerHandle_t pxTimer)
{
        xTaskCreate((pdTASK_CODE)aperiodic_task,
              (signed char *)"Aperiodic",
              configMINIMAL_STACK_SIZE,
                 NULL, 2,
```

```
                &aperiodic_handle);
        const long xMaxExpiryCountBeforeStopping = 10;
        configASSERT(pxTimer);
        lExpireCounters += 1;
/* El temporizador correrá 10 veces y luego se detendrá */
        if (lExpireCounters == xMaxExpiryCountBeforeStopping) {
                xTimerStop(pxTimer, 0);
        }
}
```

- El temporizador debe ejecutar la siguiente tarea cada 5 segundos:

```
static void aperiodic_task()
{
        long i;
        for (i = 0; i<1000000000; i++); // ciclo para simular trabajo
        vTaskDelete(aperiodic_handle);}
```

Contestar las siguientes preguntas:

- ¿Es el sistema suficientemente rápido para ejecutar todas las tareas aperiódicas?
- Si no lo es solucione el problema utilizando un servidor retardador.
- ¿Cuál es el tiempo de respuesta de la tarea aperiódica?

4.5.2. *Blinking led* en FreeRTOS

La práctica consiste en:

- Implementar un módulo GPIO para leer los botones y los interruptores en la placa.
- Implementar 4 temporizadores con 4 periodos distintos para cada uno.
- Los interruptores deben indicar qué ledes se iluminan.
- Los botones cambiarán entre los periodos de parpadeo para cada led.
- Cada led debe tener un interruptor de encendido, un botón para cambiar su periodo y un temporizador.

5. BANDERAS DE EVENTOS Y SISTEMA DE COLAS DE TAREAS

5.1. Colas

Las colas permiten enviar información entre tarea de forma asíncrona. Los datos son copiados a la cola para prevenir que cambien.

Las colas utilizan manipuladores del tipo xQueue. Una vez creada una cola y asociada a un manipulador se pueden escribir y leer de la cola, así como solicitar algunas de sus características, como espacios libres.

Para crear una cola FreeRTOS provee dos funciones:

```
QueueHandle_t xQueueCreate (UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize);

QueueHandle_t xQueueCreateStatic (
                            UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize,
                            uint8_t *pucQueueStorageBuffer,
                            StaticQueue_t *pxQueueBuffer);
```

La diferencia esencial es que xQueueCreateStatic permite al usuario especificar el espacio en memoria para la cola y necesita un puntero tipo StaticQueue_t para almacenar la estructura.

uxQueueLength	Indica el número máximo de elementos que la cola puede contener.
uxItemSize	Corresponde al tamaño en *bytes* que ocupa cada elemento dentro de la cola. Cada elemento en la cola tiene el mismo tamaño.

Si la cola es creada exitosamente se retorna un manipulador para la cola, en caso contrario se retorna un NULL.

Figura 68. **Ejemplo de creación de una cola**

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
};

void CreateQueue( void *pvParameters )
{
QueueHandle_t xQueue;
    /* Una cola capaz de almacenar 10 estructuras tipo AMessage */
    xQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    if( xQueue  == NULL )
    {
        /* No se creó la cola */
    }
    ...
 }
```

Fuente: elaboración propia.

Ya creada una cola puede eliminarse con la siguiente función.

```
void vQueueDelete (QueueHandle_t xQueue );
```

Esta función libera la memoria apartada para la cola en su creación. La función toma como parámetro el puntero que indica la cola a eliminar. En caso de que solo se desee restablecer la cola a su estado original. FreeRTOS provee la siguiente función:

```
BaseType_t xQueueReset (QueueHandle_t xQueue );
```

Limpia la cola xQueue devolviéndola a su estado inicial.

5.1.1. Escribir en una cola

Al tener una cola creada esta puede llenarse enviando datos a la cola con la función:

```
BaseType_t xQueueSend(
        QueueHandle_t xQueue,
        const void * pvItemToQueue,
        TickType_t xTicksToWait);
```

Esta función envía el dato pvItemToQueue a la cola especificada por el manipulador. El dato enviado no tiene que exceder el tamaño de la cola. El parámetro xTicksToWait es opcional y determina el tiempo máximo que la función esperará por un espacio en la cola en caso de que esté llena. La función retorna pdTRUE en caso de que se haya escrito exitosamente en la cola, en caso contrario retorna errQUEUE_FULL.

Figura 69. **Ejemplo de escritura en una cola**

```
struct AMessage
 {
    char ucMessageID;
    char ucData[ 20 ];
 } xMessage;

 void WriteQueue( void *pvParameters )
 {
 QueueHandle_t xQueue;
 struct AMessage *pxMessage;
    /* Una cola para datos AMessage. */
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );

    if( xQueue != 0 )
    {
        pxMessage = & xMessage;
        /* Enviar a la cola xQueue el pxMessage y no esperar */
        xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );
    }
 }
```

Fuente: elaboración propia.

Otras funciones similares son xQueueSendToFront y xQueueSendToBack. xQueueSendToBack es equivalente a xQueueSend, pero xQueueSendToFront difiere en que inserta el dato al principio de la cola.

Al escribir en una cola es útil conocer previamente si hay espacios disponibles.

UBaseType_t uxQueueSpacesAvailable (QueueHandle_t xQueue);

Esta función devuelve la cantidad de espacios libres en la cola.

FreeRTOS también provee una función en caso de que la cola ya esté llena, pero sea necesario escribir en ella de igual forma. Esto será útil en sistemas tipo *stream* donde lo importante es que los datos sean lo más recientes posibles y no hay problemas en perder algún dato ya que la información no depende solo de un dato.

```
BaseType_t xQueueOverwrite(QueueHandle_t xQueue,
                           const void * pvItemToQueue);
```

Esta función permite reescribir una cola. Copia en la cola xQueue sustituyendo el último valor por el indicado por el puntero pvItemToQueue. Según la documentación de FreeRTOS la función fue diseñada con el propósito de que se use con colas de un elemento. Pero, al ser otra versión de xQueueSendToBack(), se puede utilizar para sobre escribir el último dato cargado a la cola.

5.1.2. Leer de una cola

Ya que se puede escribir en una cola lo importante es recuperar la información. Para esto existen dos funciones xQueueReceive y xQueuePeek.

```
BaseType_t xQueuePeek( QueueHandle_t xQueue,
                       void *pvBuffer,
                       TickType_t xTicksToWait);

BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void *pvBuffer,
                          TickType_t xTicksToWait);
```

Ambas funciones permiten recuperar el último dato agregado a la cola. De igual forma que al escribir se especifica la cola a leer con el puntero xQueue. Debido a que las colas son genéricas y C es un lenguaje con tipos estrictos es imposible retornar un valor. Por lo tanto, al recuperar un valor de la cola es necesario indicar con un puntero la dirección de memoria donde se copiará la información. Esta es la labor del puntero pvBuffer, este punto tiene que ser del mismo tipo que el dato que se está leyendo para que haya suficiente espacio en memoria reservado. De igual forma que al escribir, la función provee un parámetro para esperar un elemento en el caso que la cola esté vacía, xTicksToWait.

La diferencia esencial entre ambas funciones es, xQueueReceive retira el elemento de la cola y xQueuePeek solo copia el elemento sin modificar la cola.

Continuado con el ejemplo de la figura 69:

Figura 70. **Ejemplo del xQueueRecieve**

```
 void ReadQueue(void *pvParameters )
 {
 struct AMessage *pxRxedMessage;
         if(xQueue != 0 ) // Comprobar que la cola fue creada
    {
        if( xQueueReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
        {
            // pxRxedMessage apunta a una copia del último dato ingresado a la
cola
                 // por ser Receive, la cola tiene un elemento menos ahora
              }
    }
 }
```

Fuente: elaboración propia.

Al leer una cola es útil conocer la cantidad de mensajes en espera.

UBaseType_t uxQueueMessagesWaiting (QueueHandle_t xQueue);

Retorna la cantidad de mensajes almacenados en la cola asociada al puntero xQueue.

Figura 71. **Ejemplo de xQueuePeek**

```
void PeekQueue( void *pvParameters )
 {
 struct AMessage *pxRxedMessage;
         if( xQueue != 0 ) // Comprobar que la cola fue creada
    {
        if( xQueuePeek( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
        {
            // pxRxedMessage apunta a una copia del último dato ingresado a la
cola
            // el dato permanece en la cola.
                  ...
        }
    }
```

Fuente: elaboración propia.

5.2. Grupos de eventos

Las banderas de eventos son otra estructura de control que provee FreeRTOS, viene definida en el archivo *event_group.h.*

EventGroupHandle_t xEventGroupCreate (void);

Esta función crea un grupo de banderas de eventos y retorna un manipulador con el cual se puede referenciar el objeto Las banderas se almacenan en variables tipo EventBits_t, las cuales son de 8 bits si

configUSE_16_BIT_TICKS está en 1, o de 24 bits si configUSE_16_BIT_TICKS está en 0.

Figura 72. **Creación de un grupo de banderas**

```
      /* Declarar una variable para almacenar el
grupo de banderas creado.  */
     EventGroupHandle_t xCreatedEventGroup;

    /* Crear el grupo de banderas */
    xCreatedEventGroup = xEventGroupCreate();

    /* Comprobar que se logró crear */
    if( xCreatedEventGroup == NULL )
    {
        /* El grupo de banderas no se pudo crear */
    }
    else
    {
        /* Se pudo crear con exito */
    }
```

Fuente: elaboración propia.

Se puede eliminar un grupo de banderas utilizando la siguiente función:

void vEventGroupDelete (EventGroupHandle_t xEventGroup);

Todas las tareas bloqueadas por el grupo de eventos se desbloquearán y recibirán un valor de 0 en las banderas de eventos.

5.2.1. Interactuando a través de grupos de eventos

Para que sean útiles es necesario poder modificar las banderas de acuerdo a condiciones o eventos provocados por el programa. Los grupos de

banderas no son más que bits que actúan como indicadores binarios de que algo es cierto o no. Para colocar los bits en 1 se utiliza:

EventBits_t xEventGroupSetBits (EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet);

La función requiere el manipulador asociado a ese grupo de banderas y las banderas a configurar como 1. Por ejemplo:

Figura 73. **Ejemplo de uso de grupos de eventos**

```
#define BIT_0     ( 1 << 0 )
#define BIT_4     ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

  /* Fija el bit 1 y el bit 4 como 1 en el grupo de eventos */
  uxBits = xEventGroupSetBits(
                              xEventGroup,    /* El grupo de eventos a actualizar */
                              BIT_0 | BIT_4 );/* Los bits a fijar como 1 */

}
```

Fuente: elaboración propia.

La función retorna los bits luego de actualizar el grupo de eventos. En caso de que solo se quiera leer el estado del grupo de evento basta con la función:

uxBits = xEventGroupGetBits(EventGroupHandle_t xEventGroup);

Esta función recupera la trama de bits del grupo de eventos.

Cuando es necesario fijar bits específicos en 0 se utiliza la función:

```
EventBits_t xEventGroupClearBits (EventGroupHandle_t xEventGroup,
                                  constEventBits_t uxBitsToClear
                                  );
```

Al igual que xEventGroupSetBits está función requiere el grupo de eventos a modificar y los bits a fijar en 0. La funcionalidad es la misma, pero esta función retorna la trama antes de haber sido modificada.

5.2.2. Sincronización de tareas con grupos de eventos

Para bloquear una tarea en espera de ciertas condiciones FreeRTO ofrece dos funciones xEventGroupWaitBits y xEventGroupSync. Ambas funciones evalúan un grupo de eventos esperando cumplir con ciertos bits por cierto tiempo, pero cada una sirve propósitos distintos.

```
EventBits_t xEventGroupWaitBits(
        const EventGroupHandle_t xEventGroup,
        const EventBits_t uxBitsToWaitFor,
        const BaseType_t xClearOnExit,
        const BaseType_t xWaitForAllBits,
        TickType_t xTicksToWait);
```

Bloqueará la tarea por el tiempo xTicksToWait esperando que se cumpla la condición descrita por uxBitsToWaitFor. xClearOnExit permite que la función al comprobar que los bits se activaron los limpie al salir. La función también permite que se evalúe de forma disyuntiva al pasar el parámetro xWaitForAllBits como pdFALSE en caso contrario (pdTRUE) la función evalúa los bits en forma

conjuntiva. En otras palabras, al estar en pdTRUE se requiere que se cumplan todas las condiciones y al estar en pdFALSE basta con que se cumpla una.

Figura 74. **Bloqueo de función utilizando xEventGroupWaitBits**

```
void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;
  uxBits = xEventGroupWaitBits(
            xEventGroup,   /* El grupos de eventos a probar */
            BIT_0 | BIT_4, /* Los bits  */
                  pdTRUE,        /* Los bits se colocan en 0 al
        salir */
                pdFALSE,       /* No espera a todas las condiciones
        */
                   xTicksToWait );/* Tiempo de espera de 100ms*/
  // Siempre es necesario evaluar el retorno de la función
  if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
  {
      /* Ambos bits están en 1 */
  }
  else if( ( uxBits & BIT_0 ) != 0 )
  {
      /* Solo el bit 0 está en 1 */
  }
  else if( ( uxBits & BIT_4 ) != 0 )
  {
      /* Solo el bit 4 está en 1 */
  }
  else
  {
      /* La función retornó porque se acabó el tiempo de espera
*/
  }
}
```

Fuente: elaboración propia.

En cambio xEventGroupSync, está diseñada pensando en *rendezvous* de tareas (encuentro de tareas). Ya que típicamente se utiliza para sincronizar varias tareas. Donde una tarea tiene que esperar a que otras alcancen el punto de sincronización para continuar con la ejecución.

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,
                             const EventBits_t uxBitsToSet,
                             const EventBits_t uxBitsToWaitFor,
                             TickType_t xTicksToWait );
```

La función opera sobre el grupo de eventos especificado por xEventGroup. Primero coloca algunos bits especificados por el programador (uxBitsToSet) para indicar que la tarea ya alcanzó en el punto de sincronización. Luego, la función bloquea la tarea en espera de que los bits especificados en uxBitsToWaitFor se fijen en 1.

Al igual que en xEventGroupWaitBits, la función permite que se configure un tiempo de espera a través de xTicksToWait. Y al igual que en la función anterior, es necesario probar el valor retornado por la función para corroborar que los eventos.

Figura 75. **Sincronización de tres tareas con xEventGroupSync**

```
#define TASK_0_BIT        ( 1 << 0 )  // Los bits usados por cada tarea
#define TASK_1_BIT        ( 1 << 1 )
#define TASK_2_BIT        ( 1 << 2 )
#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

EventGroupHandle_t xEventBits; // El grupo de eventos para sincronizar las tareas
void vTask0( void *pvParameters )
{
EventBits_t uxReturn;
TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;
    for( ;; )
    {
        // Funcionalidad antes de la sincronización
        uxReturn = xEventGroupSync( xEventBits,    // El grupo de eventos
                                    TASK_0_BIT,    // La tarea 0 ya llegó
                                    ALL_SYNC_BITS, // Espera las demás tareas
                                    xTicksToWait );// Espera 100 ms
        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
        {
        /* Todas las tareas se sincronizan antes de que el tiempo se acabara */
        }
    }
}

void vTask1( void *pvParameters )
{
    for( ;; )
         {
        // Funcionalidad antes de la sincronización
        xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS, portMAX_DELAY );
       // Funcionalidad luego de la sincronización por el portMAX_DELAY
          }
}

void vTask2( void *pvParameters )
{
    for( ;; )
    {
         // Funcionalidad antes de la sincronización
         xEventGroupSync( xEventBits, TASK_2_BIT, ALL_SYNC_BITS, portMAX_DELAY
);
         // Funcionalidad luego de la sincronización por el portMAX_DELAY
          }
}
```

Fuente: elaboración propia.

5.3. Notificaciones

Cada tarea en FreeRTOS tiene un espacio de 32 bits dedicado a notificaciones. Una notificación de tarea en FreeRTOS es un evento enviado directamente a la tarea y puede desbloquear la tarea receptora, y opcionalmente puede actualizar el valor de la notificación de la tarea.

Las notificaciones requieren menos memoria que las otras estructuras de FreeRTOS por lo tanto ofrecen soluciones más rápidas y más eficientes respecto a recursos. Las notificaciones pueden interactuar de varias formas con el valor de notificación de la tarea que recibe la notificación. Estas son:

- Puede configurar el valor de la notificación sin reescribir el anterior
- Puede sobrescribir el valor de notificación de la tarea.
- Puede fijar en 1 solo algunos bits del valor de notificación de la tarea.
- Incrementar el valor de notificación de la tarea.

Las notificaciones pueden enviarse usando xTaskNotify, xTaskNotifyAndQuery o xTaskNotifyGive. Para esperar notificaciones se utilizan xTaskNotifyWait o ulTaskNotifyTake. Las notificaciones están habilitadas por defecto en *FreeRTOSConfig.h* pero se pueden deshabilitar fijando configUSE_TASK_NOTIFICATIONS en 0. Cuando se deshabilitan las notificaciones las tareas ahorran 8 bytes de memoria.

5.3.1. Interactuar a través de notificaciones

Entre las funciones para enviar una notificación existe:

```
BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify,
                       uint32_t ulValue,
                       eNotifyAction eAction );
```

Esta función toma como parámetros la tarea a notificar xTaskToNotify, el valor a enviar ulValue, y por último un especificador (eAction) que le indica a la tarea que recibe la notificación que hacer con el valor.

El último parámetro puede provocar varias conductas en la tarea receptora dependiendo del parámetro:

eNoAction	La tarea recibe la notificación, pero no hace nada con el valor recibido.
eSetBits	Al recibir la notificación la tarea actualizará su valor de notificación realizando una operación *or* entre el valor actual y el recibido en la notificación.
eIncrement	La tarea aumenta el valor de su notificación en uno.
eSetValueWithOverwrite	El valor de la notificación de la tarea se sobrescribe con el valor recibido.
eSetValueWithoutOverwrite	Si la tarea que recibe la notificación no tiene un valor de notificación previo, toma el ulValue de la notificación recibida. En caso contrario la función

xTaskNotify retorna un pdFALSE ya que no se puede escribir en el valor de notificación de la tarea.

FreeRTOS ofrece una versión alternativa a esta función:

```
BaseType_t xTaskNotifyAndQuery (TaskHandle_t xTaskToNotify,
        uint32_t ulValue,
            eNotifyAction eAction,
            uint32_t *pulPreviousNotifyValue );
```

La diferencia es que xTaskNotifyAndQuery permite recuperar el valor de notificación de la tarea antes de que sea modificado. Esto se logra pasando un puntero pulPreviousNotifyValue, en el cual se copia el valor.

Y por último se tiene la forma más sencilla de enviar una notificación, xTaskNotifyGive. Esta solo permite aumentar el valor de la notificación de una tarea. Por lo tanto, es ideal para un semáforo con conteo. La función es un macro de la función xTaskNotify con el parámetro de eAction fijado en eIncrement.

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

Naturalmente FreeRTOS también provee funciones para bloquear tareas en espera de una notificación. Estas funciones son xTaskNotifyWait y xTaskNotifyTake. Ambas permiten bloquear la tarea por cierto tiempo (xTicksToWait) mientras espera recibir una notificación.

xTaskNotifyWait brinda mayor funcionalidad:

```
BaseType_t xTaskNotifyWait(uint32_t ulBitsToClearOnEntry,
                           uint32_t ulBitsToClearOnExit,
                           uint32_t *pulNotificationValue,
                           TickType_t xTicksToWait );
```

Esta función permite modificar el valor de notificacion propio al llegar a este punto con ulBitsToClearOnEntry. La función colocara en 0 los bits especificados antes de esperar otra notificación. De forma similar, ulBitsToClearOnExit permite fijar bits específicos en 0 luego de haber recibido la notificación. El puntero pulNotificationValue permite capturar el valor que viene con la notificación en caso de que no sea necesario puede dejarse en NULL.

En contraste, la función xTaskNotifyTake funciona como un semáforo de conteo ligero. Esta espera un xTaskNotifyGive. Al recibir la notificación solo decrece su valor de notificación (al igual que un semáforo con conteo) o lo reinicia a 0, funcionando como un semáforo binario.

```
uint32_t ulTaskNotifyTake(BaseType_t xClearCountOnExit,
                          TickType_t xTicksToWait );
```

xClearCountOnExit determina el comportamiento de la tarea ante la notificación. Si está en pdTRUE la función regresará el valor de la notificación de la tarea a 0 al salir. Si se pasa como un pdFALSE el valor de notificación de la tarea decrece en 1. La función retorna el valor de notificación de la tarea luego de ser modificado.

Por último para limpiar el valor de notificación de una tarea basta con pasar el manipulador de la tarea a la función:

```
BaseType_t xTaskNotifyStateClear(TaskHandle_t xTask );
```

En caso de que se quiera borrar el valor en la tarea que llama la función basta con pasar el argumento como NULL, de esta forma la tarea se autorreferencia para limpiar su valor de notificación.

5.4. Interrupciones

Para el diseño de sistemas embebidos es preciso determinar si los componentes aportan funcionalidad extra que sea útil. Nadie aprecia trabajar horas en algo que no brinde ningún beneficio.

En este caso se tienen dos formas para determinar si un evento externo ocurrió o no. La primera es llamada *polling* o sondeo. Consiste en evaluar cada cierto tiempo el periférico para averiguar si hubo algún cambio. La segunda son las famosas interrupciones permiten que el periférico indique cuando sucede un cambio.

Tabla VI. **Sondeo contra interrupción**

Interrupción	**Sondeo**
Asíncrono: la interrupción puede suceder en cualquier instante y no hay forma de predecirla.	Síncrono: Se sabe que el evento sucederá en un espacio de tiempo corto.

Continuación tabla VI.

Urgente: es crítico que el sistema reaccione lo más pronto posible ante el evento.	No urgente: no es tan importante y otra tarea puede efectuarse entre sondeos.
Infrecuente: son eventos esporádicos.	Frecuente: la mayoría de sondeos resultan en un evento.

Fuente: SISTERNA, Cristian.
Interrupts in zynq Systems. ICTP-IAEA.

Una analogía es útil es cocinar una pizza. El sondeo es abrir la puerta del horno cada 2 minutos para averiguar si la pizza está lista y la interrupción es esperar la alarma del horno para abrir la puerta.

Las interrupciones dependen de la plataforma a utilizar. Ya que estas, a su vez, dependen de que la plataforma las soporte y al ser interacciones directas con el hardware, estas están configuradas dentro de la HAL. Por lo tanto, no hay forma de modificarlas a través de FreeRTOS, sino que es necesario configurarlas con los registros de los microcontroladores o microprocesadores en los que se haga el despliegue de la aplicación.

5.4.1. Prioridades en FreeRTOS

Es importante considerar que al utilizar FreeRTOS, las prioridades son administradas por este. FreeRTOS define dos grupos de interrupciones, las que son enmascarables por FreeRTOS, y las que no son enmascarables, estas siempre están habilitadas. La división entre estos dos grupos se especifica en el archivo de configuración de FreeRTOS con la entrada:

configMAX_SYSCALL_INTERRUPT_PRIORITY

Por lo tanto, es importante definir este número de acuerdo a la cantidad de bits que el microcontrolador tiene implementados. Al configurar los registros de prioridad en FreeRTOS se definen los niveles utilizables por las tareas del sistema y los utilizables por las interrupciones de hardware.

Tabla VII. **Niveles de prioridades de FreeRTOS**

<table>
<tr><td rowspan="8">Las interrupciones que no llaman funciones del API de FreeRTOS pueden utilizar todos los niveles, y se anidaran.</td><td>Prioridad 7</td><td rowspan="3">Interrupciones que usen estas prioridades serán retrasadas si el kernel está ocupado.</td></tr>
<tr><td>Prioridad 6</td></tr>
<tr><td>Prioridad 5</td></tr>
<tr><td>Prioridad 4</td><td rowspan="5">Todas las interrupciones que utilicen funciones terminadas en “FromISR” del API utilizan estas prioridades, y se anidaran.</td></tr>
<tr><td>Prioridad 3</td></tr>
<tr><td>Prioridad 2</td></tr>
<tr><td>Prioridad 1</td></tr>
<tr><td>Prioridad 0</td></tr>
</table>

Fuente: GOYETTE, Richard. *An Analysis and Description of the Inner Workings of FreeRTOS.* http://richardgoyette.com/Research/Papers/FreeRTOSPaper.pdf. Consulta: 14 de noviembre de 2017.

5.4.2. Información específica del hardware

La configuración del *kernel* de FreeRTOS provee una entrada que permite modificar el comportamiento del *kernel* ante este evento. Dependiendo del microcontrolador:

- Para las plataformas ARM Cortex-M3, PIC24, dsPIC, PIC32, SuperH y RX600:
  El nombre de la entrada es:
  configKERNEL_INTERRUPT_PRIORITY

- Para las plataformas PIC32, RX600, ARM Cortex-A y ARM Cortex-M:
  El nombre de la entrada es:
  configMAX_SYSCALL_INTERRUPT_PRIORITY

Existen innumerables aplicaciones en procesadores de la familia Cortex-M3, Cortex-M4, Cortex-M4F y Cortex-M7. Estos procesadores incluyen en su lista de registros uno llamado BASEPRI.

El Registro BASEPRI es el encargado de definir la prioridad mínima para excepciones. Es importante notar que dependiendo del procesador la cantidad de niveles únicos de prioridades cambia. Para la arquitectura ARM Cortex-M el número máximo de prioridades es 256, pero los microcontroladores solo tienen acceso a algunos de estos. Por ejemplo, el TI Stellaris Cortex-M3 tienen 3 bits de prioridades lo cual permite 8 valores de prioridad únicos.

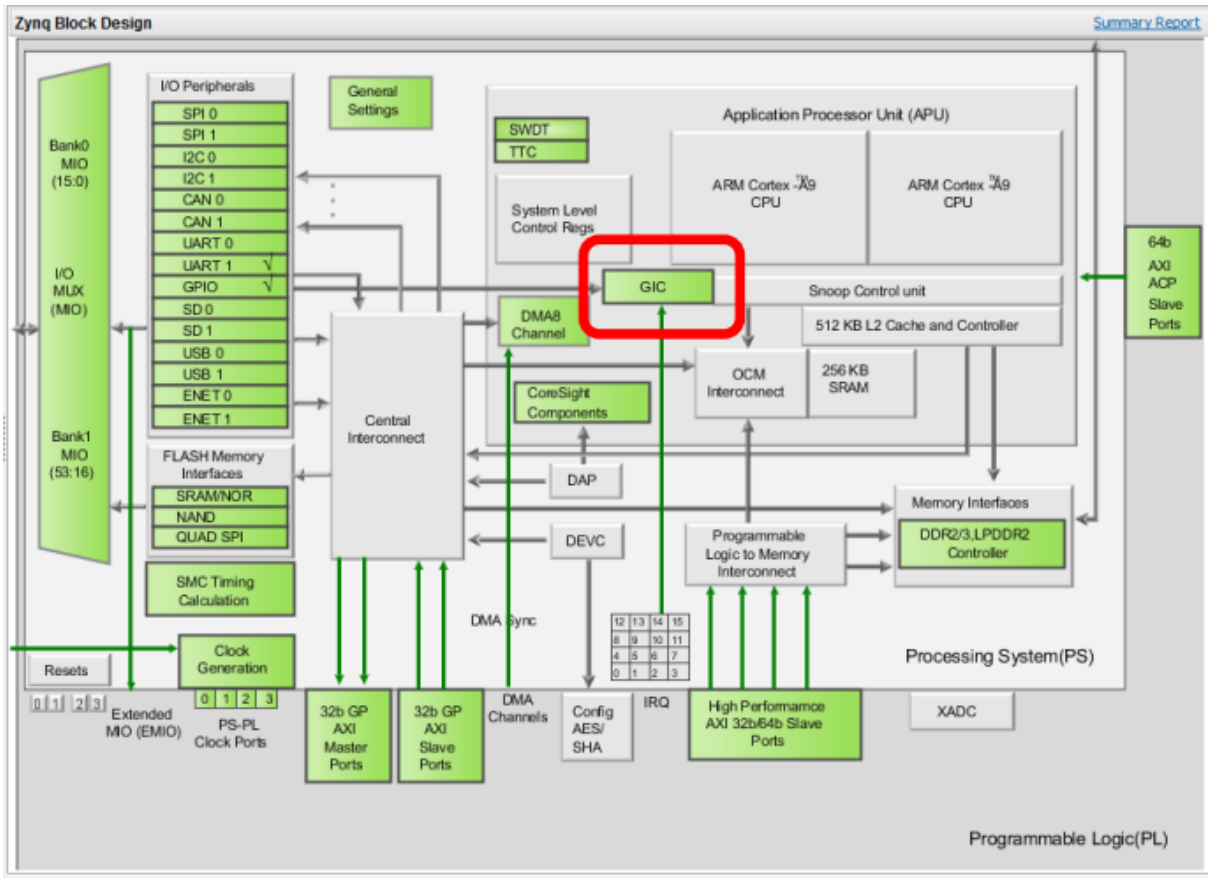
En caso de que el proyecto incluya la librería CMSIS (*Cortex Microcontroller* Software *Interface Standard*, por sus siglas en inglés), se puede

inspeccionar el archivo en busca de la definición __NVIC_PRIO_BITS que define el número de bits disponibles para prioridades.

En el caso del ARM Cortex A9 las interrupciones son manejadas por un Controlador de Interrupciones interno. Es una unidad funcional que cumple solamente con esta tarea.

Este controlador de interrupciones es el módulo de hardware que permite programar interrupciones por software. Permite enmascarar las interrupciones para especificar qué procesador se encargará de darle servicio. Soporta 16 interrupciones generadas por software (SGI) y 64 interrupciones de periféricos compartidas (SPI). Aparte, cada procesador tiene cinco interrupciones privada de periféricos dedicadas (PPI), estas no son seleccionables por el usuario. Este módulo se le conoce GIC controlador genérico de interrupciones.

Figura 76. **Diagrama interno del ARM Cortex A9 con el GIC señalado**



Fuente: elaboración propia.

El GIC permite dos tipos de interrupciones, por cambio de flanco y por cambio de nivel. Una evalúa si es flanco de subida o flanco de bajada, y la otra evalúa si se pasó a un estado bajo o a un estado alto.

Las SGI solo pueden ser de cambio de flanco. Para generarlas es necesario escribir el número de la interrupción en el registro ICDSGIR. Cada procesador tiene acceso a 5 PPI's.

Tabla VIII. **Interrupciones PPI de cada núcleo del ARM Cortex A9**

ID#	Nombre	Tipo	Descripción
27	Temporizador Global	Flanco de subida	
28	nFIQ	Bajo nivel	Interrupciones rápidas desde la lógica programable (PL, en otras palabras, la FPGA)
29	Temporizador Privado del CPU	Flanco de subida	
30	AWDT	Flanco de subida	Perro guardián o *watch dog,* de cada CPU
31	nIRQ	Bajo nivel	Interrupciones desde la lógica programable (PL, en otras palabras, la FPGA

Fuente: ARM. *Cortex-A9 MPCore Technical Reference Manual: 3.3.7. PPI Status*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407e/Bhacbfdb.html. Consulta: noviembre 8, 2017.

Los PPI's comprenden 60 interrupciones de distintas fuentes que pueden dirigirse a cualquier procesador a través de la lógica programable.

5.4.3. Interrupciones y tareas

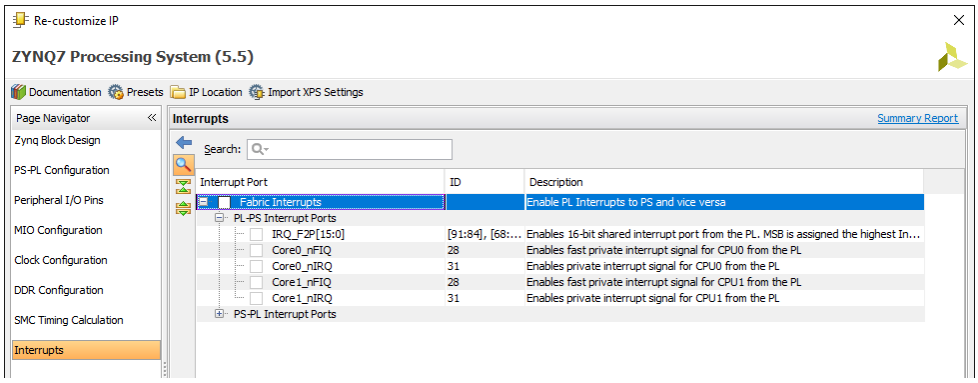
En muchos casos una tarea depende de que una ISR determine que los recursos necesarios o el momento apropiado para la ejecución de la tarea. Una rutina de ISR debe cumplir con ciertas directivas para el correcto funcionamiento del sistema:

- La rutina debe ser corta.
- FreeRTOS requiere que sea de tipo void.
- No permitir anidar interrupciones dentro de interrupciones.
- El tiempo es esencial, mientras más tiempo se pase en una interrupción más probable es que se pierda otra interrupción (la interrupción bloquea el CPU por lo que todas las demás tareas o interrupciones con menor prioridad tendrán que esperar).
- Utilizar el orden recomendado dentro de la función:
    - Borrar el bit de interrupción
    - Realizar la tarea
    - Rehabilitar la interrupción al salir *(no todas se deshabilitan manualmente)*

5.4.4. Interrupciones en vivado

Para habilitar interrupciones en vivado es necesario activarlas primero en el procesador.

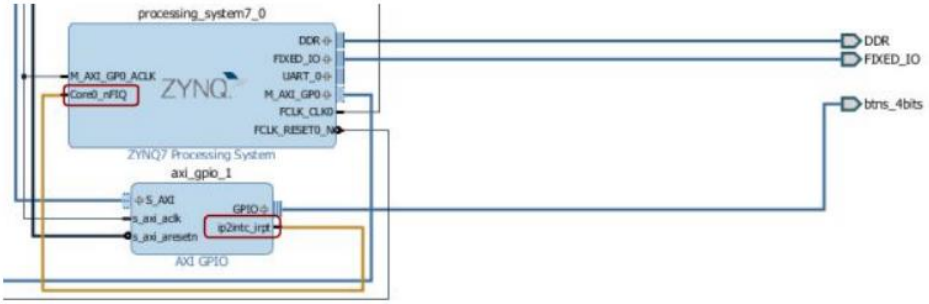
Figura 77. **Configuración del sistema de procesamiento de zynq7000**



Fuente: elaboración propia, empleando vivado.

Luego, se deben conectar los pines para asociar los bloques en la forma apropiada. Si la interrupción la genera el procesador entonces es de tipo PS-PL y si la interrupción la genera un bloque en la FPGA es PL-PS.

Figura 78. **Ejemplo de conexión de una interrupción PL-PS**

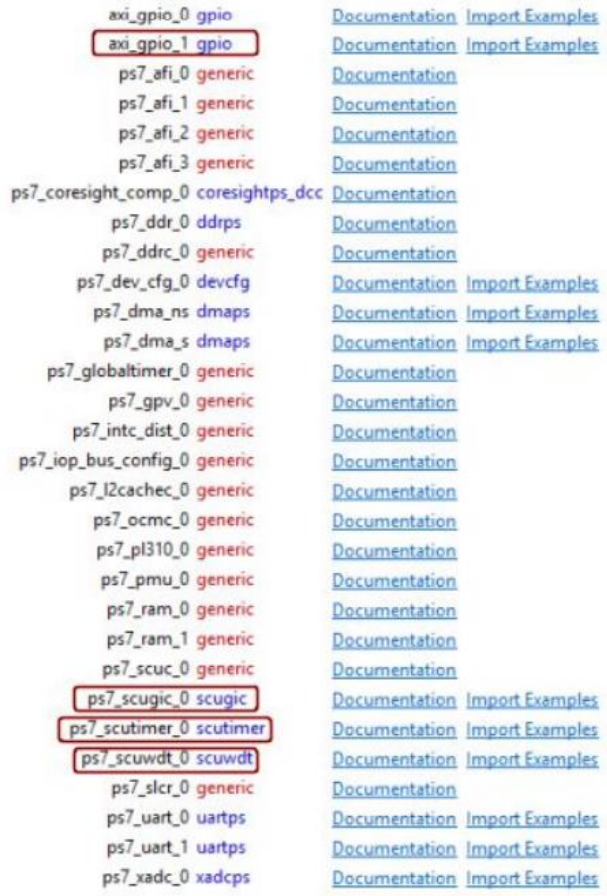


Fuente: elaboración propia, empleando vivado.

En el ejemplo se conecta un bloque genérico de entrada y salida con el procesador 0. El bloque enviará una interrupción desde los botones al CPU0 a través del registro nFIQ (28). Ya que las interrupciones están habilitadas y conectadas basta con definir la programación para utilizarlas.

Al abrir el *system.mss* en el SDK de vivado se puede observar que se agregaron cuatro periféricos al procesador. El nuevo módulo GPIO, un temporizador del SCU, el GIC y el perro guardián de SCU (scuwdt).

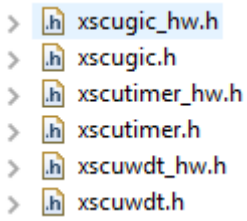
Figura 79. ***System.mss* con los periféricos para interrupciones**



Fuente: elaboración propia, empleando vivado.

A su vez aparecen las librerías nuevas para interactuar con los periféricos en la carpeta *include*.

Figura 80. **Librerías con los controladores para los periféricos GIC y SUC**



Fuente: elaboración propia, empleando vivado.

Entre todas las librerías nuevas en la carpeta *include* se encuentra *xil_exception.h.*

La librería *xscugic.h* contiene los controladores para la configuración del controlador genérico de interrupciones, y la librería *xil_expcetion.h* contiene las funciones de excepciones para los procesadores Cortex-A9.

5.4.5. Ejemplo de interrupción en vivado

Ya que se trabaja con un módulo de interrupciones es necesario configurar el módulo. Para esto se crea una instancia del módulo GIC y una instancia del objeto que contiene la información para su configuración:

Figura 81. **Implementación del GIC**

```
// Declarar las dos estructuras para interactuar con el
GIC
    XScuGic mi_Gic;            // Instancia del GIC
    XScuGic_Config *mi_Gic_Config; // Configuración
```

Fuente: elaboración propia.

Ya implementado es necesario configurarlo. Para esto es necesario el registro de identificación del periférico a utilizar como interrupción, en este caso el GPIO 1. El nombre del registro se encuentra definido en el archivo *xparameters.h.* Para este periférico el registro es XPAR_AXI_GPIO_1_DEVICE_ID. Otros parámetros necesarios son:

- El identificador de las interrupciones en el procesador XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
- El identificador de los dispositivos
- El identificador de las interrupciones de GPIO, este se encuentra en el archivo *xgpio_l.h*

Figura 82. **Redefinición de registros para mayor legibilidad**

```
#define INTC_DEVICE_ID              XPAR_PS7_SCUGIC_0_DEVICE_ID
#define BTNS_DEVICE_ID              XPAR_AXI_GPIO_1_DEVICE_ID
#define ledes_DEVICE_ID                   XPAR_AXI_GPIO_0_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID      XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define BTN_INT                     XGPIO_IR_CH1_MASK
```

Fuente: elaboración propia.

Luego ya se puede comenzar el proceso de la configuración de la interrupción. Primero se crea la instancia del GIC y su configuración, también es necesario configurar los XGpio como interrupciones.

Figura 83. **Configuración del GIC y el GPIO**

```
XGpio ledInst, BTNInst;       // Declara los Gpios a utilizar
XScuGic INTCInst;       // Crea el objeto GIC
XScuGic_Config *IntcConfig;   // Crea la configuración

IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);    // Busca información del
dispositivo
XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);

Xil_ExceptionInit();

XGpio_InterruptEnable(&BTNInst, BTN_INT);       // Habilita la interrupción en
ese canal
XGpio_InterruptGlobalEnable(&BTNInst);          // Habilita interrupciones en el
puerto

// Conecta la fuente de la interrupción con su manipulador
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, (Xil_ExceptionHandler)
XScuGic_InterruptHandler, &INTCInst);

// Conecta la interrupción con la función que le da servicio
XScuGic_Connect(&INTCInst, INTC_GPIO_INTERRUPT_ID, (Xil_ExceptionHandler)
BTN_Intr_Handler, (void *)&BTNInst);

// Habilita interrupciones de GPIO en el GIC
XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);

Xil_ExceptionEnable(); // Habilita las interrupciones
```

Fuente: elaboración propia.

Es necesario definir la función BTN_Intr_Handler de acuerdo a las necesidades del programa. Se sugiere seguir el siguiente esquema:

Figura 84. **Rutina de interrupción genérica**

```
void Generic_Intr_Handler(void *InstancePtr)
{
// Deshabilitar la interrupción para evitar que entre de nuevo
XGpio_InterruptDisable(&BTNInst, BTN_INT);
// Comprobar que la interrupción la proco el dispositivo
if ((XGpio_InterruptGetStatus(&BTNInst) & BTN_INT) != BTN_INT)
     {
      return;
     }
// Código específico a la aplicación
// Limpiar el bit de interrupción
(void) XGpio_InterruptClear(&BTNInst, BTN_INT);
// Habilitar la interrupción
XGpio_InterruptEnable(&BTNInst, BTN_INT);
```

Fuente: elaboración propia.

Basta con sustituir los registros por los necesarios en caso de un tipo distinto de interrupción o un origen distinto.

Al modificar las conexiones en vivado siempre se actualizan las librerías en el SDK, por lo tanto, los registros necesarios siempre se encontrarán en los archivos indicados a continuación:

- *xgpio.h* Contiene las definiciones de funciones para los GPIO.
- *xgpio_l.h* Contiene las definiciones de los registros
- *xparameter.h* Contiene todos los registros de identificación
- *xil_exception.h* Contiene las funciones para manejo de interrupciones
- *Xscugic.h* Contiene los controladores para el GIC

5.5. Práctica 5: IoT

La práctica consiste en elaborar un dispositivo capaz de leer interacciones del ambiente y comunicarlas a un cliente por protocolo TCP/IP. Para esto se recomienda estudiar el ejemplo *Hello World Ethernet* de vivado. El objetivo de la práctica es conocer los distintos periféricos y cómo implementarlos en un proyecto.

Los datos que debe comunicar el dispositivo son:

- Uso de ram
- Uso de CPU
- Temperatura del CPU
- 5 segundos de audio

Nota: utilizar un temporizador con interrupción para medir el tiempo del conversor analógico digital. En caso de que se utilice un módulo externo buscar una alternativa que permita garantizar el tiempo sin bloquear el procesador por 5 segundos. El audio tiene que reproducirse en tiempo real del lado del cliente.

CONCLUSIONES

1. Se desarrollan los temas necesarios para que el lector pueda adquirir un conocimiento profundo sobre la utilidad de los sistemas operativos en tiempo real, especialmente FreeRTOS.

2. Se desarrollan los temas necesarios para familiarizarse con diversos entornos de programación para ampliar la disponibilidad de herramientas del lector al momento de desarrollar una aplicación.

3. Se desarrollan ejemplos y se presentan las herramientas para que luego el lector pueda demostrar sus conocimientos con ejercicios prácticos sencillos.

4. Se introdujo FreeRTOS para su uso en la plataforma de desarrollo tiva C, en la plataforma de desarrollo zybo y un simulador para computadora.

5. El flujo de trabajo, la definición de hardware en la FPGA y la programación de software en el procesador se demuestran utilizando el IDE vivado.

6. Se brindan herramientas para la validación de planificadores de tareas sencillos y un emulador para su comprobación.

RECOMENDACIONES

1. Adquirir los conocimientos básicos de programación en C previo a la lectura de este manual, ya que esos temas se obvian asumiendo que el lector ya tiene cierta práctica en el lenguaje.

2. Al importar placas de desarrollo a vivado, asegurarse que los archivos contengan los nombres de los periféricos que vivado solicita. Esto se puede corroborar en la ventana de diseño del periférico.

3. Siempre que se desea implementar una aplicación en un sistema embebido, es recomendable crear un esquema de los módulos a utilizar. De esta forma el proceso de creación de módulos específicos es más claro y no interfiere directamente con la programación de la aplicación.

4. Un manual presenta información técnica detallada, pero es imposible prescindir de la hoja técnica del microcontrolador a programar. Todos los dispositivos electrónicos tienen una hoja de datos donde se indica toda la información técnica. A diferencia del manual la hoja técnica solo describe las características del dispositivo y muy raras veces su uso.

5. Vivado genera el código necesario para implementar un módulo definido en VHDL.

6. En este documento solo se mencionan los algoritmos más básicos para gestores de tareas. Asimismo, los periféricos mencionados en el

manual y en las prácticas son lo más comunes. Para profundizar en los temas se puede continuar el estudio con las bibliografías utilizadas.

BIBLIOGRAFÍA

1. AMBA Specifications – Arm. [en línea]. <https://www.arm.com/products/system-ip/amba-specifications>. [Consulta: 24 de octubre de 2017].

2. ARM Generic Interrupt Controller Architecture ... - ETH Systems Group. [enlínea]. <https://www.systems.ethz.ch/sites/default/files/file/aos2012/ReferenceMaterial/InterruptHandling/GIC_architecture_spec_v1_0.pdf>. [Consulta: 8 de noviembre de 2017].

3. AXI Reference Guide - Xilinx. [en línea]. <https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf>. [Consulta: 25 de octubre de 2017].

4. Chapter-2 Real-Time System Concepts [en línea]. <https://www.csie.ntu.edu.tw/~d6526009/ucOS2/Chapter-2.pdf.> [Consulta: 16 de octubre de 2017].

5. Cortex-A9 MPCore Technical Reference Manual: 3.1.2. Interrupt types [en línea]. <http://infocenter.arm.com/help/topic/com.arm.doc.di0407e/CCHDBEBE.html>. [Consulta: 7 de noviembre de 2017].

6. Cortex-M3 Devices Generic User Guide: 2.1.3. Core ... - ARM Infocenter. [en línea]. <http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/CHDBIBGJ.html>. [Consulta: 16 de octubre de 2017].

7. Development of Real-Time Systems | Coursera [en línea]. <https://www.coursera.org/learn/real-time-systems>. [Consulta: 12 de octubre de 2017].

8. Embedded Basics – API's vs HAL's – Beningo Embedded Group [en línea]. <https://www.beningo.com/embedded-basics-apis-vs-hals/>. [Consulta: 12 de octubre de 2017].

9. FAQ: What is a Board Support Package? - Wind River Systems [en línea]. <https://www.windriver.com/products/bsp_web/what_is_a_bsp.pdf>. [Consulta: 12 de octubre de 2017].

10. FreeRTOS task states and state transitions described. [en línea]. <http://www.freertos.org/RTOS-task-states.html>. [Consulta: 16 de octubre de 2017].

11. Goyette, Richard. An Analysis and Description of the Inner Workings of FreeRTOS [en línea]. <http://richardgoyette.com/Research/Papers/FreeRTOSPaper.pdf>. [Consulta: 16 de octubre de 2017].

12. Interprocess communication [en línea]. <http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/slides/Chapter6%20RTOS%20Communication.pdf>. [Consulta: 12 de octubre de 2017].

13. Interrupts - byu.net. [en línea]. <http://ecen330wiki.groups.et.byu.net/wiki/lib/exe/fetch.php?medi a=ch7_ug585-zynq-7000-trm.pdf>. [Consulta: 8 de noviembre de 2017].

14. Interrupts AXI GPIO and AXI Timer ECE 699: Lecture 4 - the GMU ECE [en línea] <http://ece.gmu.edu/coursewebpages/ECE/ECE699_SW_HWS15/viewgraphs/ECE699_lecture_4.pdf>. [Consulta: 7 de noviembre de 2017].

15. Introduction — Simso documentation. [en línea]. <http://projects.laas.fr/Simso/doc/introduction.html>. [Consulta: 16 de octubre de 2017].

16. Object-oriented C: HAL meets RTOS | Embedded [en línea]. <https://www.embedded.com/electronics-blogs/in-the-trenches/4410305/HAL-meets-RTOS>. [Consulta: 12 de octubre de 2017].

17. OSAL - NASA - GSFC Open Source Software [en línea]. <https://opensource.gsfc.nasa.gov/projects/osal/>. [Consulta: 12 de octubre de 2017].

18. Presentación de PowerPoint - Indico - ICTP. [en línea]. <http://indico.ictp.it/event/7987/session/34/contribution/128/material/slides/0.pdf>. [Consulta: 7 de noviembre de 2017].

19. Process Scheduling - CS Rutgers. [en línea]. <https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html>. [Consulta: 16 de octubre de 2017].

20. Real-time Scheduling of Periodic Tasks (2). [en línea]. <https://csperkins.org/teaching/2013-2014/adv-os/lecture03.pdf>. [Consulta: 16 de octubre de 2017].

21. URRIZA, José; OROZCO, Javier. *Métodos rápidos para el cálculo del Slack Stealing* [en línea]. <http://www.ingelec.uns.edu.ar/rts/papers/Urriza_Orozco_Metodos_rapidos_de_Slack_Stealing.pdf>. [Consulta: 31 de octubre de 2017].

22. Vivado Design Suite User Guide: Creating and Packaging ... - Xilinx. [en línea]. <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug1118-vivado-creating-packaging-custom-ip.pdf>. [Consulta: 24 de octubre de 2017].

23. Zynq-7000 A Generation Ahead Backgrounder - Xilinx. [en línea]. <https://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf>. [Consulta: 24 de octubre de 2017].