INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS

SCHEDULER DE SUCESIÓN BINARIA CON CAMBIO DE CONTEXTO

Tesina para obtención de grado en:

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presentan: **Juan Carlos Campeche Valencia.**

Jesús Manuel Espinoza Norzagaray.

Sergio Solano Alonso.

Director: Ing. Francisco Martínez Chávez

San Pedro Tlaquepaque, Jalisco a 11 de julio de 2018.

Agradecimiento al Consejo Nacional de Ciencia y Tecnología (CONACyT) por su apoyo económico para la realización de este proyecto.

Agradecimiento al Instituto Tecnológico de Estudios Superiores de Occidente (ITESO), institución de gran calidad, que nos brindó todo el apoyo durante el estudio del posgrado.

Queremos agradecerle a nuestro asesor de tesis, Francisco Martínez Chávez, por sus conocimientos brindados para llevar a cabo esta investigación y por su gran paciencia para la finalización de este proyecto.

Abstract

Resource optimization in embedded systems is mandatory because resources are very limited. To solve this problem, it is important to use multitasking techniques that allow software designers to allocate resources efficiently. The scheduler is the basis to achieve a multitasking system since it manages the execution order task and distributes the processor load. In this thesis, the implementation of a binary succession scheduler with the ability to configure a variable number of tasks is presented, including their execution frequency and the offset to avoid collisions. The scheduler has a context switching mechanism that complements the automatic process of stacking and unstacking the processor registers when an interruption occurs, preventing tasks from losing information. Using the ARM programming model, the scheduler code was developed in C language with blocks assembly instructions to implement the context switch, also the shadowed stack pointer was used to insulate the scheduler memory. The system exceptions provide a lower priority in the scheduler processes than the external interruptions triggered by peripherals and the system clock define the task execution time. The context switch in a scheduler can be a critical weakness in the system by compromising the response time of a multitasking system and the selection of a bad strategy could affect up to 350µsec in a 200MHz processor [5].

Resumen

En un sistema embebido es primordial la optimización de los recursos, por lo que es importante el uso de técnicas multitareas que permitan a los diseñadores de software asignar los recursos de manera eficiente. El scheduler o planificador es la base para lograr un sistema multitareas, porque administra el orden de ejecución de las tareas y distribuye la carga del procesador. En esta tesis se presenta la implementación de un scheduler de sucesión binaria con la capacidad de configurar un número variable de tareas, su frecuencia de ejecución y el desface para evitar colisiones. El scheduler tiene un mecanismo de cambio de contexto que complementa el proceso automático de guardado y restaurado de los registros del procesador cuando una interrupción ocurre, evitando la perdida de información de las tareas. Usando el modelo de programación para ARM se desarrolló código en lenguaje c el scheduler con bloques de instrucciones en ensamblador para implementar el cambio de contexto, además se utilizaron dos punteros a la pila para separar la memoria el scheduler de las tareas, excepciones del sistema con la finalidad de dar menor prioridad en los procesos del scheduler que las interrupciones externas invocadas por periféricos y el reloj del sistema definiendo los tiempos de ejecución de las tareas. El cambio de contexto en un scheduler puede llegar a ser una debilidad importante en el sistema, comprometiendo el tiempo de respuesta de un sistema multitareas, la selección de una mala estrategia pudiera afectar hasta ≈ 350µsec en un procesador de 200MHz [5].

Lista de figuras

Lista de tablas

Abreviaturas y acrónimos

AAPCS	Procedimiento estandar de llamadas de la arquitectura ARM(del inglés, *ARM Architecture Procedure Call Standard*)
ABI	Interfaz de aplicación binaria (del inglés, Application Binary Interface)
BASEPRI	Registro de la máscara de la prioridad base(del inglés, *Base Priority Mask Register*)
CLKSOURCE	Fuente de reloj (del inglés, Clock Source)
EXC_RETURN	Valor de retorno de la excepción (del inglés, Exeption Return)
FAULTMASK	Registro de la máscara de error (del inglés, *Fault Mask Register*)
FCSE	Extensión para cambio de contexto rápido (del inglés, *Fast Context Switch Extension*)
FPU	Unidad de punto flotante (del inglés, *Float Point Unit*)
LDF	Archivo de descripción del ligador(del inglés, *Linker Description File*)
LR	Registro de enlace (del inglés, *Link Register*)
MMU	Unidad de administración de memoria (del inglés, *Memory Management Unit*)
MPU	Unidad de protección de memoria (del inglés, *Memory Protection Unit*)
NVIC	Controlador de interrupciones vectorial anidado(del inglés, Nested Vectored Interrupt Controller)
PC	Contador del programa (del inglés, *Program Counter*)
PID	Identificador del proceso (del inglés, *Process ID*)
PRIMASK	Registro máscara de prioridad (del inglés, *Priority Mask Register*)
PSR	Registro de estatus del programa(del inglés, *Program Status Register*)
SP	Apuntador a la pila(del inglés, *Stack Pointer*)
SVC	Llamada de servicio (del inglés, Service Call)
TCM	Memoria estrechamente unida (del inglés, *Tightly Coupled Memory)*
TLB	Búfer de traducción de direcciones (del inglés, *Translation Lookaside Buffer)*

Contenido

Introducción

En un sistema embebido es primordial la optimización de los recursos, por lo que es importante el uso de técnicas multitareas que permitan a los diseñadores de software asignar los recursos del sistema entre diferentes tareas. Para que varias tareas puedan ejecutarse simultáneamente es necesario un elemento que organice el tiempo de procesamiento y esto nos lleva al uso del scheduler.

El scheduler o planificador es la base para lograr un sistema multitareas, porque administra las tareas que se ejecutan en la unidad de procesamiento central y distribuye la carga del procesador [1]. Básicamente cuando una tarea de la más alta prioridad esté lista para ejecutarse (Ready) se le cede el uso del procesador y la tarea actual en ejecución es suspendida y su estado cambia de Running a Suspended.

Durante el proceso de cambio de tareas es necesario guardar en la memoria los registros fundamentales del procesador usados por la tarea en ejecución, para luego restaurar de la memoria los registros de la tarea próxima a ejecutar, a este proceso se le conoce como cambio de contexto [1]. El cambio de contexto permite a las tareas continuar su ejecución justo en el momento donde fueron interrumpidas, lo que permite que las tareas no pierdan información y se tenga un sistema capaz de ejecutar múltiples tareas.

ARM es proveedor de arquitectura de procesadores, su amplia variedad de productos lo han colocado como el principal proveedor de procesadores para sistemas embebidos en el mundo, ya que ofrece procesadores de aplicación final, para procesamiento en tiempo real y procesadores para microcontroladores.

Dentro de su portafolio se encuentran los procesadores de la familia Cortex-M, los cuales presentan un equilibrio entre tamaño, desempeño y alta eficiencia energética [2]. La escalabilidad es una característica que hace muy popular a la familia Cortex-M, debido a que todos los procesadores de esta familia usan el mismo núcleo, lo que permite que el código para el procesador más básico también sea funcional para el procesador más complejo de esta familia.

El objetivo de esta tesina es implementar el cambio de contexto en un scheduler de sucesión binaria en la arquitectura ARMv7-M, en este trabajo se hace uso del microcontrolador ATSAMV71Q21, el cual está contenido en la tarjeta de evaluación SAM V71 Xplained Ultra proporcionada por el Instituto Tecnológico y de Estudios Superiores de Occidente (ITESO). En la tesina se muestra la implementación de un scheduler que selecciona la siguiente tarea a ejecutar y hace el cambio de contexto cuando ocurre una interrupción de ciclo de reloj u ocurre una interrupción de alta prioridad que necesite ser atendida inmediatamente.

1. Antecedentes

Para reducir el tiempo de ejecución del cambio de contexto, en la literatura se mencionan principalmente dos alternativas; la primera es optimizar algoritmos de planificación o scheduling y, la segunda, es administrar de buena manera la memoria en el sistema.

En el artículo “An Improved Method of Task Context Switching in OSEK Operating System” [4], se propone modificar el algoritmo de cambio de contexto del sistema operativo OSEK, debido a que éste siempre guarda o restaura el contexto, sin importar si la tarea es ejecutada por primera o última vez. La propuesta es definir estrategias de cambio de contexto que dependan del estado de la tarea en ejecución y de la siguiente tarea a ejecutar, permitiendo guardar o restaurar el contexto de una tarea innecesariamente.

En [4] se explica cómo disminuir el tiempo de ejecución del cambio de contexto cuando se configura correctamente el MMU (del inglés, *Memory Management Unit*). Dentro de éste se encuentra el registro FCSE (del inglés, *Fast Context Switch Extension*), que permite que múltiples tareas tengan una región fija en la memoria, ligada a un identificador. El uso de este identificador reduce el tiempo de búsqueda de la memoria de cada tarea; además, presenta ventajas sobre las técnicas que usan la memoria caché y el TLB (del inglés, *Translation Lookaside Buffer*) porque no necesita paginación y vaciado. Es importante mencionar que el FCSE no está disponible en todos los procesadores de ARM.

Adam Wiggins [5] muestra las desventajas del uso de la caché sin la implementación de los registros de la MMU, debido a que se tiene que hacer un vaciado del TLB y de la caché del sistema en cada cambio de contexto, aumentando el costo del rendimiento del sistema. El artículo propone evitar el costo del rendimiento haciendo uso de las características especiales proporcionadas por los dominios o identificadores y la reubicación PID (del inglés, *Process ID*).

El artículo *Implementation and validation of dynamic scheduler based on LST on FreeRTOS* destaca un caso de estudio donde se desarrolla y prueba un scheduler en la plataforma ARM Cortex-M4. La investigación propone un scheduler donde las tareas tienen prioridades que varían de forma dinámica, para ello recurren al uso del algoritmo *Least Slack Time First* (LST) y aplican los conceptos de mutex y semáforos, sobresaliendo la comprobación del scheduler en tiempo real, por último, para validar el scheduler los autores desarrollaron las aplicaciones apropiadas que representaron la naturaleza de las tareas (tarea periódica, tarea aperiódica y tarea esporádica) [6].

A. Arya Paul, B. Anju. S. Pillai, en su investigación *Reducing the Number of Context Switches in Real Time Systems,* describen un modelo computacional para simulación de tareas periódicas donde reducen el número de preemptions. Su trabajo se enfoca en los algoritmos *Earliest Deadline First* (EDF) y *Dynamic Preemption Threshold Scheduling*; En el artículo se realiza un análisis de cambio de contexto y la sincronización de tareas con los protocolos *Basic Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP)*. El resultado de la simulación fue la documentación del reúso del 100% de CPU, usando una política de scheduling basada en la teoría del algoritmo EDF [7]

2. Marco Teórico.

2.1. Procesadores ARM Cortex M

ARM es un proveedor de arquitectura de procesadores. Ofrece procesadores para aplicaciones de alto desempeño (Cortex-A), para procesamiento en tiempo real (Cortex-R) y procesadores para microcontroladores (Cortex-M). Los procesadores de la familia Cortex-M son los más usados en el mundo porque presentan un equilibrio entre tamaño, desempeño y alta eficiencia energética [8].

Una de sus características principales es que todos los procesadores de esta familia usan el mismo núcleo, permitiendo que el código desarrollado para el procesador más básico (Cortex M0) sea funcional en los procesadores más complejos de la familia (Cortex M7).

La familia de los procesadores Cortex M comparte [9]:

- Modelo base de programación.
- NVIC (del inglés, Nested Vectored Interrupt Controller) para el manejo de interrupciones.
- Características para soporte de sistemas operativos.
- Soporte para depurar.

2.2. Modelo de programación

El modelo de programación de la familia Cortex-M está compuesto por los modos de procesador, niveles de ejecución de software y los registros que componen al núcleo del procesador. El procesador tiene un total de 32 registros, de los cuales 13 son de propósito general y el resto son de propósito específico, como se muestra en Figura 2-1 [9].

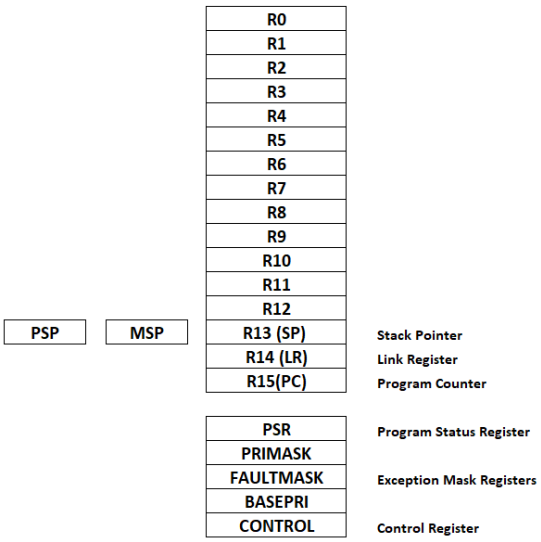

Figura 2-1 Registros del núcleo del procesador

Los registros de propósito general (R0-R12) son usados para manejar datos y direcciones. El registro R13 es conocido como SP (del inglés, *Stack Pointer*) y es un apuntador a la pila del sistema operativo (MSP) o a la pila de las tareas (PSP).

El registro R14 o LR (del inglés, *Link Register*) se encarga de guardar la dirección de retorno cuando una subrutina, una llamada a una función o una excepción es llamada. El PC (del inglés, *Program Counter*) contiene la dirección de la instrucción que se está ejecutando.

El registro PSR (del inglés, *Program Status Register*) está compuesto por tres registros. El primero nos indica el resultado de la última operación realizada, el segundo contiene el número de interrupción atendida y el último nos indica que tipo de instrucción está en ejecución.

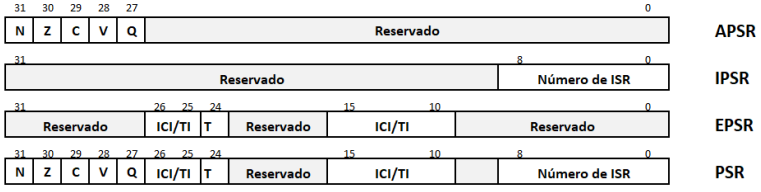

Figura 2-2 Program Status Register

Los registros para excepciones nos ayudan a deshabilitar el manejo de excepciones por el procesador. El registro PRIMASK (del inglés, *Priority Mask Register*) evita la activación de todas las excepciones que tengan prioridad configurable. El registro FAULTMASK (del inglés, *Fault Mask Register*) previene la activación de todas las excepciones, menos de las excepciones no enmascaradas. El registro BASEPRI (del inglés, *Base Priority Mask Register*) configura a partir de qué prioridad son permitidas las interrupciones como se muestra en la Figura 2-3.

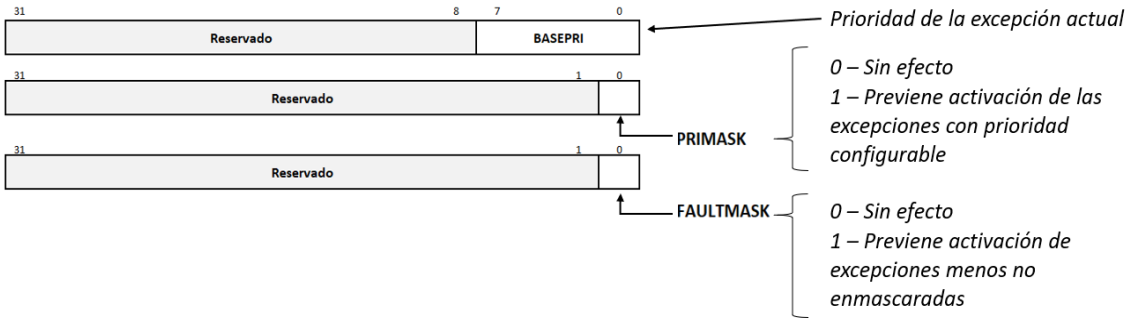

Figura 2-3 Registros PRIMASK, FAULTMASK y BASEPRIO

El registro de control nos permite activar la unidad de punto flotante, seleccionar el SP como MSP o PSP y definir el nivel de ejecución como se muestra a continuación.

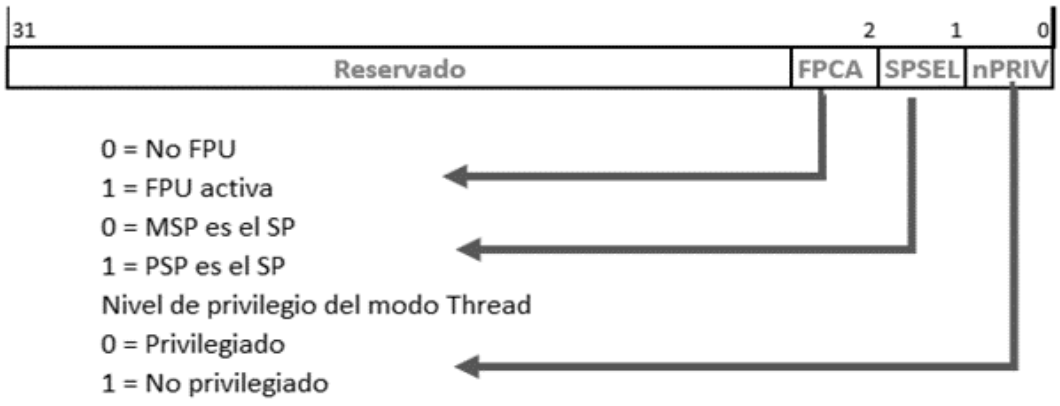

Figura 2-4 Registro CONTROL

2.3.Modelos de operación en ARM Cortex-M

Los procesadores Cortex-M soportan la ejecución de código en modo privilegiado y modo usuario (No privilegiado). El modo privilegiado asegura un acceso a todas las regiones de memoria; sin embargo, el modo usuario tiene restricciones a ciertas regiones de memoria como es el reloj de sistema (SysTick) y el manejador de interrupciones (NVIC). El usuario puede cambiar entre estos dos modos como se muestra en la Figura 2-5.

También soporta dos modos de operación, el *Thread* y *Handler***.** El modo Handler siempre es ejecutado es modo privilegiado, usa el MSP y se entra cuando ocurre una excepción en el microcontrolador. El modo *Thread* tiene la posibilidad de ser ejecutado en modo privilegiado o no, además permite decidir si se usa el MSP o PSP; es usado cuando no ocurren excepciones como se muestra en la Figura 2-5.

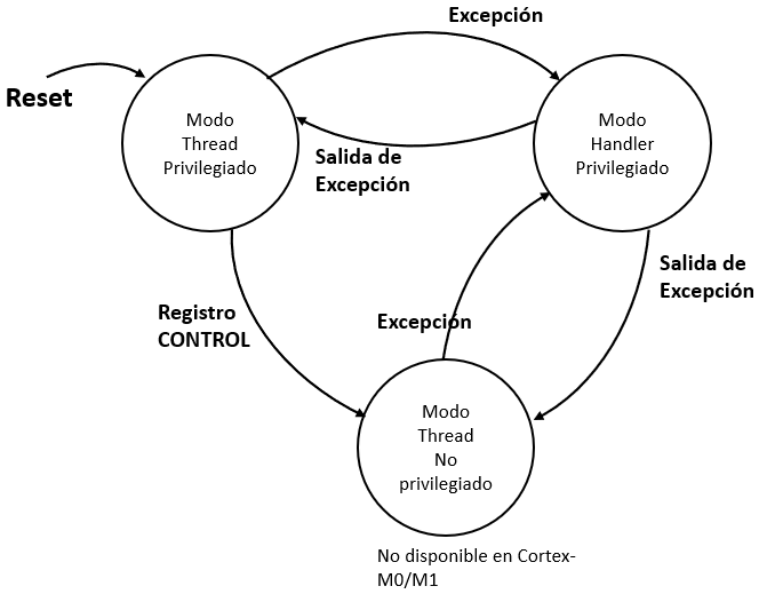

Figura 2-5 Modos y niveles de ejecución

Tener privilegios de ejecución permite que el desarrollador proteja las regiones de memoria que son críticas para el sistema. El modo de operación permite separar los recursos del sistema operativo y de las tareas, debido a que se tiene una pila de memoria exclusiva para el sistema operativo (MSP) y otra para una o varias tareas (PSP).

2.4. ARM Application Binary Interface

ARM Application Binary Interface es una colección de estándares que un archivo ejecutable debe cumplir para poder ejecutarse en un ambiente específico. Dentro de estos estándares se encuentra el AAPCS (del inglés, *ARM Architecture Procedure Call Standard*), que indica los tipos de datos y el uso del stack durante el tiempo de ejecución, permitiéndonos hacer compatibles programas escritos en lenguaje ensamblador y C.

A continuación, se presentan los procedimientos que son más relevantes para este trabajo:

- Los registros R0-R3 son usados para pasar parámetros de entrada a una función.
- En el registro R0 se encuentra el parámetro retornado por la función
- Los registros R0-R3 y R12 pueden ser modificados libremente
- Si se necesita usar los registros R4-R11, estos deben ser guardados
- El stack debe esta siempre alineado a una palabra doble para un uso público.

2.5. SAMV71 Xplained Ultra

SAMV71 Xplained Ultra es un kit de desarrollo para evaluar el microcontrolador ATSAMV71Q21 y otros procesadores de la familia Cortex-M.
Sus principales características son:

- Microcontrolador ATSAMV71Q21
- Botón de reset
- 12.0 MHz cristal y 32.768 kHz cristal
- 2 MB SDRAM

- 2 MB QSPI Flash
- ATA6561 CAN Transceiver
- Arduino compatible
- Conector externo para el debugger
- USB interface
- Debugger embebido

El ATSAMV71Q21 es un microcontrolador basado en la familia Cortex-M7 RISC, tiene una arquitectura de 32 bits y cuenta con FPU (del inglés, *Float Point Unit*) y sus principales características se muestran a continuación:

- ARM Cortex-M7 funcionando hasta 300 MHz
- 6 Kbytes de ICache y 16 Kbytes de DCache con Error Code Correction (ECC)
- Simple y doble precisión FPU
- MPU (del inglés, *Memory Protection Unit*) con 16 zonas
- Set de instrucciones de DSP
- ETM (del inglés, *Embedded Trace Module*
- Hasta 2048 Kbytes de Flash
- Hasta 384 Kbytes de Multi-port SRAM
- TCM (del inglés, *Tightly Coupled Memory)* con cuatro configuraciones (disabled, 2 x 32 Kbytes, 2 x 64 Kbytes, 2 x 128 Kbytes)
- 16 Kbytes ROM con rutinas embebidas (UART0, USB) and IAP routines

2.6.Real-Time Scheduling

Como en la preparación de una agenda de horario en la vida diaria, agendar un conjunto de tareas computacionales (también conocidas como procesos), significa determinar cuándo se ejecutará cada tarea y cuál será su orden de ejecución. Esta planificación de tareas por personas específicas y las tareas computacionales son una analogía. Scheduling es una actividad central de un sistema computacional usualmente llevada a cabo por el sistema operativo.

En sistemas que no son en tiempo real, el objetivo típico de la programación de tarea es maximizar el rendimiento promedio (cantidad de tareas completadas por unidad de tiempo) y / o minimizar el tiempo de espera promedio de las tareas. En el caso de la programación en tiempo real, el objetivo es cumplir con tiempo límite de cada tarea al garantizar que cada una puede completar la ejecución dentro del límite especificado. Este plazo se deriva de las limitaciones de ambiente impuestas por la aplicación.

El análisis schedulability nos ayuda para determinar si un conjunto específico de tareas o un conjunto de tareas que satisfacen ciertas restricciones se pueden programar con éxito (completando la ejecución de cada tarea en su fecha límite especificada), usando un programador específico.

El scheduler es la base para lograr la multiprogramación. Un sistema multiprogramado tendrá varios procesos que requerirán recursos del procesador a la vez. Esto sucede cuando los procesos están en estado Ready. Si existe un procesador disponible y existen procesos en estado Ready se debe elegir el que será asignado al recurso para ejecutar.

2.7. Administración de memoria

La asignación de memoria dinámica es importante tanto en términos de demanda de memoria para las tareas de las aplicaciones, como para los requerimientos del sistema operativo.

Las tareas de aplicaciones usan memoria explícitamente a través de solicitudes de memoria de pila e, implícitamente, mediante la acción de reservar memoria necesaria para código en tiempo de ejecución requerido por un lenguaje de alto nivel, como por ejemplo C. Un sistema operativo necesita realizar una administración de memoria para mantener las tareas aisladas. La asignación inadecuada de memoria puede destruir el determinismo de los eventos; por ejemplo, un desbordamiento de pila.

2.7.1 Controlador del proceso de Stack

En un sistema multitarea, los datos de cada tarea necesitan ser guardados y restaurados en un orden de cambio de procesos. Esto se puede hacer usando una o más pilas en tiempo de ejecución, o con un modelo de bloque de control de tareas (task-control block). Las pilas funcionan mejor cuando usamos interrupciones, y el modelo de task-control block trabaja mejor con sistemas operativos en tiempo real.

Si se va a utilizar una pila para el contexto de las tareas será necesario dos simples rutinas, guardar y restaurar.

La rutina de guardar será llamada por un manejador de interrupciones para guardar en cierta área de la pila el contexto actual de la tarea. Para prevenir desastres, se recomienda hacer esta *callback* inmediatamente después de que las interrupciones hayan sido deshabilitadas

La rutina de restauración debe invocarse justo antes de que se activan las interrupciones y antes de que regrese desde el manejador de interrupciones.

2.7.2 Modelo de bloque Task-Control

Si se usa el modelo *task-control block*, entonces será necesaria una lista, esta lista puede ser fija o dinámica.

En el caso de una lista fija, un número “n” de task-control blocks se asignan en el momento de su generación en el sistema, todos en estado suspendido y su teoría de comportamiento es el siguiente:

- Cuando una tarea está en estado ready, el task-control block ingresa al estado listo.
- La tarea con prioridad más alta moverá la tarea al estado de ejecución.
- No es necesario administrar la memoria en tiempo real.

En el caso dinámico, los task-control blocks se agregan a una lista enlazada o a otro tipo de estructura de datos dinámica, a medida que se crean las tareas.

- Las tareas están en estado suspendido desde su creación y cambian a estado ready por medio de una llamada del sistema operativo o disparada por un evento.
- Las tareas cambian a estado de ejecución por su prioridad o por división de tiempo.
- Cuando una tarea es eliminada, su task-control block se remueve de la lista enlazada.
- Su espacio de memoria en el *heap* se libera y puede ser usado para una nueva tarea.

2.8. Cambio de Contexto

El cambio de contexto es el proceso de guardar el estado de una tarea, para luego ser restaurada y ejecutada en el mismo punto en donde se había dejado antes de iniciar el proceso. El proceso de cambio de contexto comprende el guardar el estado de una tarea y la restauración de la siguiente tarea en ejecutarse (ver la Figura 2-6 para más claridad en el de cambio de contexto). Esto permite que múltiples tareas compartan un solo procesador, y es una característica esencial en los sistemas multitareas.

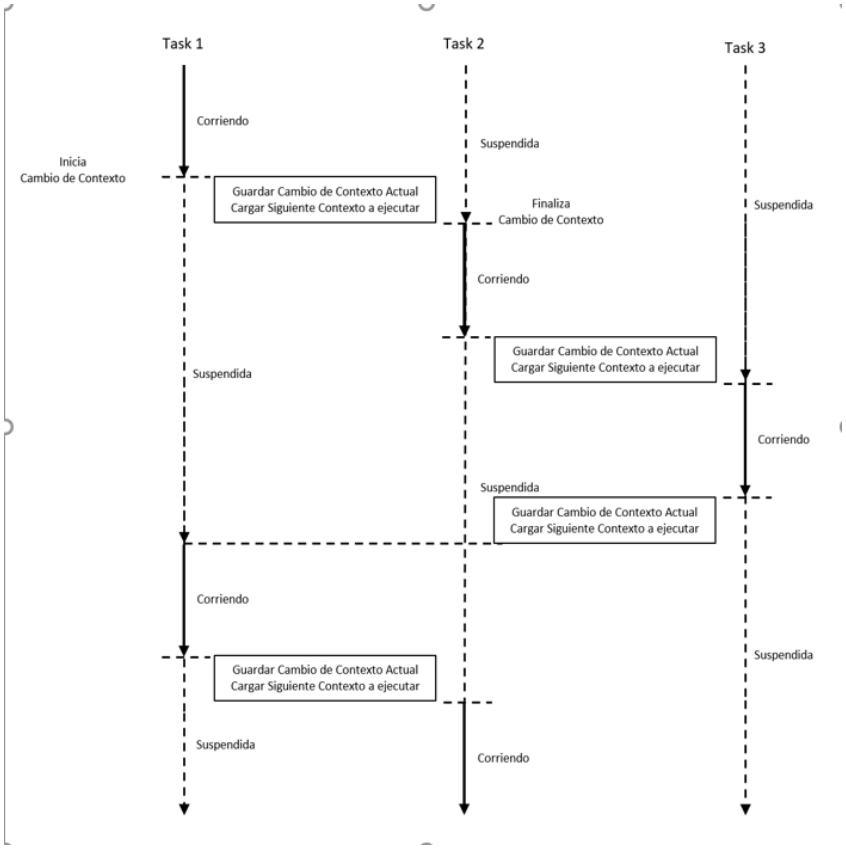

Figura 2-6 Cambio de contexto

El rendimiento de un sistema multitareas puede verse impactado por un cambio de contexto. Depende directamente de la estrategia que se esté ejecutando en el proceso de cambio de contexto, pudiendo afectar hasta ≈ 350µsec en un procesador de 200MHz [5].

2.8.1 Escenarios del cambio de contexto

Principalmente, el proceso de cambio de contexto ocurre cuando un Scheduler decide alternar una tarea por otra, por lo que se ejecuta el proceso del cambio de contexto como se muestra en la Figura 2-6. En este escenario se realiza por lo general en una interrupción, permitiendo el cambio entre el modo de ejecución de tareas y el contexto del Scheduler.

Otro escenario donde se realiza un cambio de contexto es al entrar a una interrupción estando en el contexto de una tarea que actualmente está siendo ejecutada. La diferencia de este escenario y el escenario anterior es el auto guardado que realiza el microcontrolador sólo de los registros principales para el sistema y fue realizado justo antes de que el manejador de la interrupción inicie con la ejecución del código.

En el escenario de interrupción los registros que son guardados automáticamente por el microprocesador al dispararse la interrupción son los mostrados en la Figura 2-7. En el caso de un cambio de tareas si es necesario realizar el cambio de contexto de todos los registros del procesador.

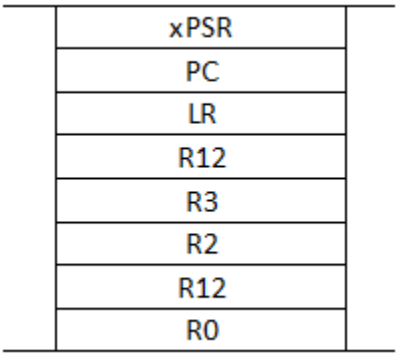

Figura 2-7 Registros de autoguardado en interrupciones

2.8.2 Estrategia en el Cambio de Contexto

La estrategia más utilizada en el proceso de cambio de contexto es utilizar el modelo de interrupciones o de excepciones, el cual consiste en la utilización de interrupciones de servicio del sistema con prioridades diferentes que provee el microcontrolador ARM. Esto permite distribuir el código núcleo de un scheduler y los procesos que conlleva los sistemas multitareas definiéndose diferentes prioridades para cada uno. Los procesos pueden ser como el cambio de contexto y la definición de la siguiente tarea a ejecutarse.

2.9.Rutinas de Servicio de Interrupción (ISR)

Las ISR son respuestas automáticas a una interrupción invocada por el hardware y son subrutinas especiales que tiene implementación de código según sea el tipo de ISR. La implementación predeterminada de una ISR es un simple bucle infinito sin ninguna funcionalidad en particular, la implementación puede ser sobrescrita por otra definición de esa misma ISR con código funcional según la conveniencia del programador.

Los procesadores ARM Cortex-M para el manejo de interrupciones utilizan un controlador de interrupciones vectorial anidado (del inglés, NVIC), esto permite al programador habilitar interrupciones específicas y configurar los diferentes niveles de prioridad para cada interrupción habilitada, debido a que el procesador se basa en los niveles de prioridad de las interrupciones para darles servicio. Significa que, si ocurre una interrupción con una prioridad mayor que la interrupción actual ejecutándose, la interrupción actual es detenida y es ejecutada la de mayor prioridad, luego de finalizar la interrupción de mayor prioridad, la de menor prioridad es ejecutada nuevamente en donde se detuvo anteriormente.

En el vector de interrupciones se encuentran definidas todas las interrupciones y se encuentran divididas según el tipo. Las excepciones del sistema son las primeras 15 interrupciones (incluidas las reservadas) y manejan las anormalidades del sistema. Las interrupciones llamadas interrupciones externas lidian con las actividades de los periféricos del microcontrolador y son de menor prioridad que las excepciones del sistema. En la Figura 2-8 presenta la tabla de interrupciones descrita.

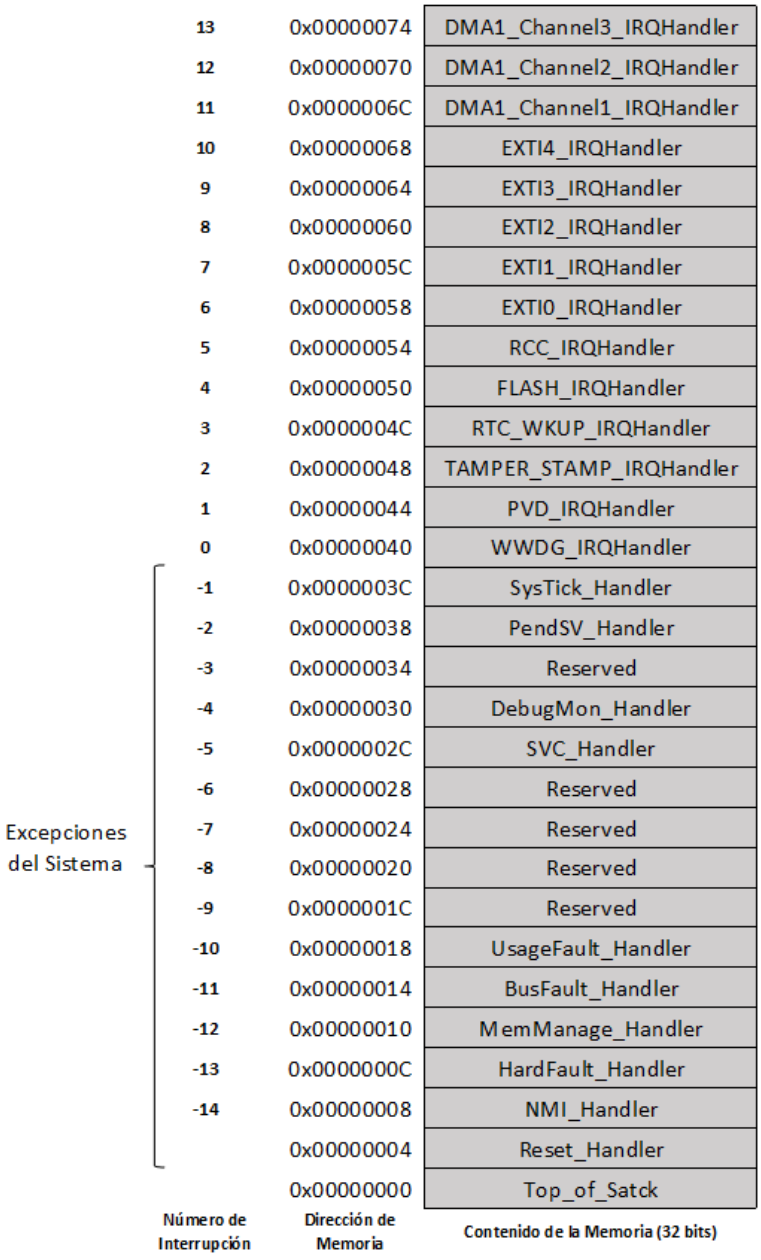

Figura 2-8 Vector de interrupciones

La tabla del vector de interrupciones puede ser reubicado en memoria, mientras la tabla del vector de interrupciones es alojada en la dirección de memoria 0x00000004. Esta dirección de memoria baja puede ser re-mapeada físicamente a diferentes regiones de memoria, como memoria flash embebida en el chip, memoria RAM embebida en el chip o memoria ROM. Esto le permite al procesador arrancar desde varias regiones de memoria [10].

Los procesadores ARM Cortex-M tienen la funcionalidad de habilitar o deshabilitar las interrupciones globalmente, esto a pesar de que el vector de interrupción o NVIC se encuentre configurado con interrupciones habilitadas. Las ISRs son habilitadas y deshabilitadas en grupo, utilizando la instrucción cambiar el estado del procesador (del inglés, *CPS*). La instrucción *CPS* hace uso del registro máscara de prioridad (del inglés, *PRIMASK*) para habilitar o deshabilitar interrupciones globalmente, excepto las interrupciones HardFault y las interrupciones no enmascarables (del inglés, *NMI*); también hace uso del registro de la máscara de error (del inglés, *FUALTMASK*) para habilitar y deshabilitar las interrupciones globalmente excepto las *NMI*. En la Tabla 2-1 Tabla para habilitar instrucciones se detalla las instrucciones para habilitar o deshabilitar las interrupciones globalmente.

Tabla 2-1 Tabla para habilitar instrucciones

Instrucción	Acción	Equivalencia
CPSID i	Deshabilita interrupciones y sólo los manejadores de errores cofigurables	MOVS r0, #0 MSR PRISMASK, r0
CPSID F	Deshabilita interrupciones y todos los manejadores de errores	MOVS r0, #1 MSR FAULTMASK, r0
CPSIE i	Habilita interrupciones y sólo los manejadores de errores cofigurables	MOVS r0, #1 MSR PRISMASK, r0
CPSIE F	Habilita interrupciones y todos los manejadores de errores	MOVS r0, #0 MSR FAULTMASK, r0
N/A	Deshabilita interrupciones con la prioridad 0x05 - 0xFF	MOVS r0, #5 MSR BASEPRI, r0

2.9.1 SysTick

El microcontrolador utiliza el SysTick para generar interrupciones periódicas y ejecutar una tarea repetidamente. La interrupción SysTick es un contador regresivo de 24-bit que produce un pequeño intervalo de tiempo fijo y es configurable por software. Se dice que es un contador regresivo debido a que inicia el conteo desde N-1 hasta 0. El procesador genera una interrupción al llegar el contador a cero y luego se reinicia al valor inicial del contador, que se almacena en un registro especial llamado valor de recarga del SysTick (del inglés, SysTick_LOAD).

Si el procesador del sistema se encuentra detenido, el contador del SysTick no para el conteo regresivo; y si se encuentra el sistema en proceso de depuración (del inglés, *debugging*), el procesador no deja de generar las interrupciones del SysTick cada vez que el contador llega a cero. El principal uso de la interrupción del SysTick, en un sistema de tiempo real con característica multitareas, es ofrecer un patrón de tiempo por hardware; Para que un Scheduler con múltiples tareas concurrentes se base en este patrón para darle un intervalo de tiempo a cada una de las tareas para su ejecución que dependiendo de las políticas del scheduler, puede ser igual para todas las tareas. La interrupción generada periódicamente por el SysTick informa al procesador que la tarea en curso debe ser detenida y el algoritmo del scheduler define la tarea próxima a ser ejecutada.

En los procesadores ARMv7 existen 4 registros de 32-bit para la configuración del SysTick o temporizador del sistema, sus direcciones de memoria se encuentran enlistadas en la Tabla 2-2 Tabla de los registros del Systick.

Tabla 2-2 Tabla de los registros del Systick

SysTick_CTRL	EQU	0xE000E010	Registro de Control y Estado del SysTick
SysTick_LOAD	EQU	0xE000E014	Registro de Valor de Recarga del Systick
SysTick_VAL	EQU	0xE000E018	Registro de Valor Actual del SysTick

El registro de control y estado del SysTick (del inglés, SysTick_CTRL) se encuentra conformado por 32-bit, el cual el bit 0 es el bit que habilita el contador, el bit 1 habilita o deshabilita la interrupción del SysTick sin deshabilitar el contador, el bit 2 indica la fuente de reloj, el bit 16 indica cuando un evento especial ha ocurrido y los bits restantes se encuentran reservados, tal y como se muestra en la Figura 2-9 Registro SysTick_CTRL.

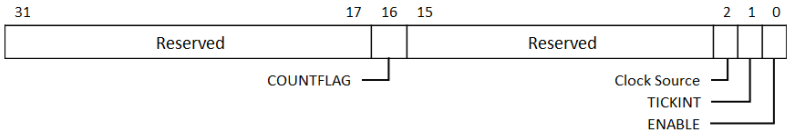

Figura 2-9 Registro SysTick_CTRL

La fuente de reloj (del inglés, CLKSOURCE) del registro SysTick_CTRL, puede ser configurado de la siguiente manera:

0 = Reloj Externo, esto significa que la frecuencia de reloj del SysTick es la frecuencia de un reloj tipo AHB divido por 8.
1 = Reloj del Procesador, el reloj que le da base de tiempo al SysTick es el mismo que alimenta al procesador del sistema.

El bit TICKINT, que habilita la interrupción del SysTick puede tomar los siguientes valores:

0 = El conteo regresivo a cero no dispara la interrupción del SysTick.
1 = El conteo regresivo a cero dispara la interrupción del SysTick.

El bit ENABLE, si su valor es 0 el contador regresivo del SysTick se encuentra deshabilitado y en caso contrario el valor de 1 habilita el contador regresivo del SysTick.

COUNTFLAG que es el bit de eventos especiales ocurridos, puede tener los siguientes dos valores:

1 = El contador ha cambiado su valor de 1 a 0 desde la última lectura del registro SysTick_CTRL.
0 = El bit COUNTFLAG ha sido limpiado por la lectura del registro SysTick_CTRL o por la escritura del registro SysTick_VAL.

El registro de valor de recarga del SysTick o SysTick_LOAD proporciona el valor inicial del conteo regresivo, al llegar el contador a cero este se reinicia con el valor que se encuentra en el registro SysTick_LOAD.

El valor actual del SysTick (del inglés, SysTick_VAL), es el registro cuyo valor es decrementado automáticamente por el procesador por cada pulso de reloj que es recibido por el temporizador (SysTick). En caso de una escritura directa al registro SysTick_VAL, ignora cualquier valor escrito y el valor es reemplazado por el valor del registro SysTick_LOAD, es decir el contador es reiniciado en el siguiente pulso de reloj. La lectura de este registro proporciona su valor actual.
Para habilitar el contador del SysTick y la interrupción, es necesario que se configure el bit TICKINT en el registro SysTick_CTRL, habilitar la interrupción en el vector NVIC el cual por default se encuentra habilitado y habilitar el bit ENABLE del registro SysTick_CTRL para habilitar el temporizador SysTick. En la Figura 2-10 Ejemplo del funcionamiento de SysTick se puede apreciar el funcionamiento con mayor detalle del contador regresivo del sistema o SysTick.

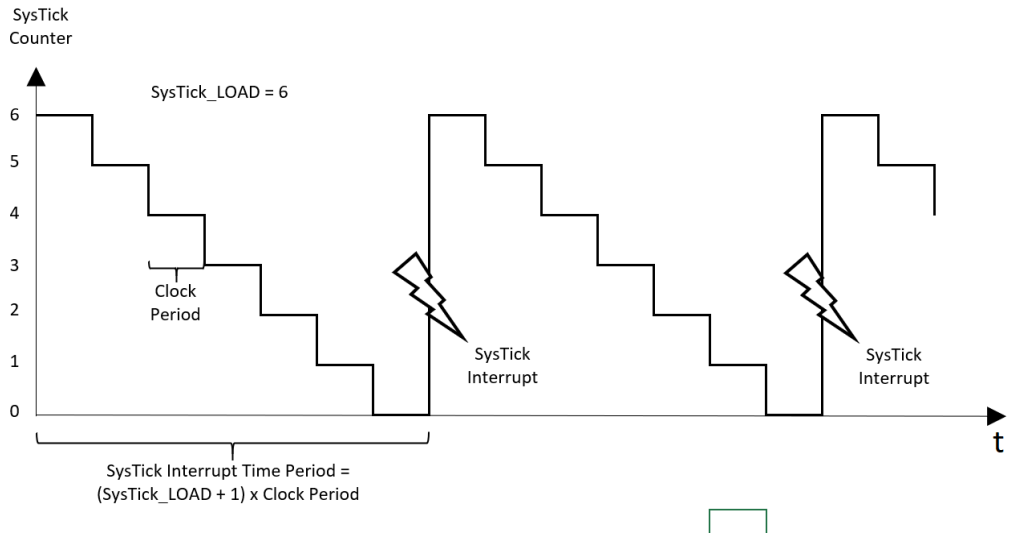

Figura 2-10 Ejemplo del funcionamiento de SysTick

2.9.2 Llamada de Servicio (SvcCall)

Debido a que el software de un microcontrolador ARM puede correr en dos modos diferentes, en modo privilegiado o en modo des privilegiado, el procesador tiene acceso limitado a los recursos y no puede ejecutar instrucciones privilegiadas cuando este se encuentra en modo des privilegiado. Para hacer uso de estos recursos sin violar las restricciones del sistema, se hace una llamada a un servicio privilegiado habilitando al procesador la utilización de los recursos, este servicio se le conoce como llamada de servicio (del inglés, SVC o SvcCall).

La llamada de servicio o SVC genera una excepción en el sistema que pone al procesador inmediatamente cuando es invocada en modo privilegiado. La SVC puede recibir parámetros que pueden ser utilizados en la subrutina del controlador SVC. Un parámetro importante puede ser el número de SVC, esto permite de manera conveniente controlar diferentes servicios por medio del único controlador SVC que existe en el sistema.

SVC puede realizar diferentes operaciones críticas como el acceso a algunos recursos compartidos o datos y es recomendable en el uso del controlador de SVC primeramente deshabilitar todas las interrupciones. Esto se logra con la instrucción “CPSID I”. SVC solo puede ser llamada por medio de una instrucción en ensamblador incrustada en el código en C, debido a que no hay una llamada correspondiente en el estándar del lenguaje C. La instrucción para hacer una llamada de servicio o SVC en ensamblador es “SVC parm1”, seguido de los parámetros (parm1) que se pasarán al controlador del SVC.

2.9.3 PendSV

El PendSV, al igual que el SVC, es una excepción del sistema utilizada para realizar servicios a nivel sistema, la cual puede ser activada configurando el registro correspondiente en el vector NVIC [11]. La diferencia principal entre la SVC y el PendSV es que la activación del PendSV puede ser postergada por el procesador debido a la prioridad que tiene en el vector NVIC; sin embargo, la interrupción es puesta en estado pendiente cuando una interrupción de mayor prioridad es ejecutada primero.

La excepción PendSV es útil en el proceso de cambio de contexto en sistemas multitareas y en la separación de una interrupción dividida en dos tareas distintas. Por ejemplo, la primera mitad puede ejecutar código de manera rápida por la ISR de mayor prioridad y la otra mitad puede ejecutar código menos crítico en la excepción PendSV que puede ser pospuesta por otras ISRs de mayor prioridad.

2.9.4 Valor de Retorno de la Excepción (ExecReturn)

El procesador ARM, al momento de entrar a una subrutina automáticamente, guarda en el registro LINK (LR) la dirección de la cual el procesador tiene que regresar luego de finalizar la subrutina en la que se encuentra. En el caso de las interrupciones o ISR definidas como subrutinas también el proceso es muy similar, pero en el registro LR se almacena el valor de retorno de la excepción (del inglés, *EXC_RETURN*).

El EXC_RETURN es un valor especial de 32-bit generado por el procesador al entrar al controlador de la interrupción y automáticamente es guardado en el registro LR. Luego de que la subrutina de la interrupción es finalizada es necesario ejecutar la instrucción en ensamblador "BX LR", para regresar al programa interrumpido por la ISR. La instrucción hace una copia del EXC_RETURN al registro PC, esto origina que el procesador regrese los valores de los registros desde la pila de interrupciones, a los registros del procesador justo en el estado donde se encontraba antes de la interrupción.

3. Metodología

3.1. Diseño General del Proyecto

El objetivo de este proyecto es la implementación del cambio del contexto en un scheduler de sucesión binaria en la tarjeta SAMV71 de Atmel, con procesador ARMv7 de la familia de los procesadores M.

De manera general, el proyecto se puede dividir en dos componentes principales para fines prácticos de planteamiento del proyecto: el Scheduler y el cambio de contexto. De estos elementos, el scheduler y el cambio de contexto se ha mejorado el código de prácticas realizadas en la especialidad.

La relación de los elementos scheduler y cambio de contexto es muy estrecha. El cambio de contexto es parte del scheduler que le permite hacer el cambio de tareas. En términos generales, el proyecto utiliza un scheduler con un cambio de contexto; esta relación puede apreciarse en la Figura 3-1 Funcionamiento Gral. del sistemaFigura 3-1.

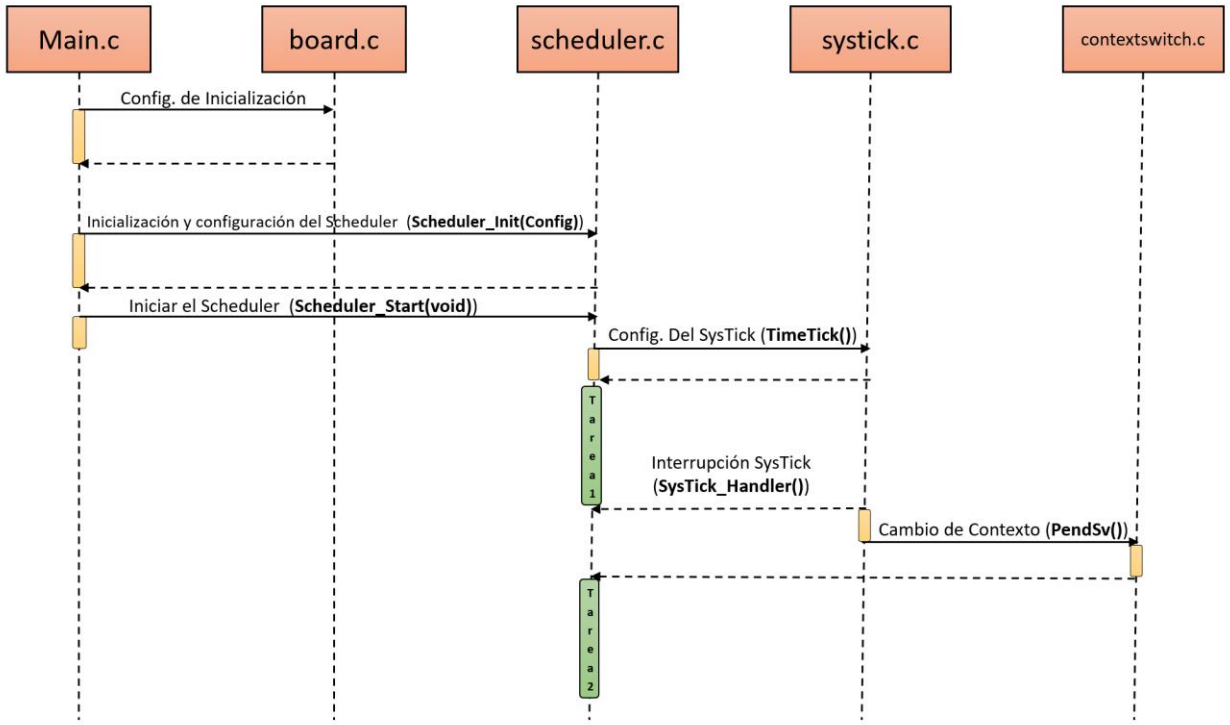

Figura 3-1 Funcionamiento Gral. del sistema

La Figura 3-2 es la base del proyecto. Es el primer diagrama que se realizó después de la investigación inicial del proyecto; fue el punto de partida para el desarrollo de todo el proyecto. Luego del desarrollo del concepto de todo el sistema que conforma este proyecto, se obtuvo el código de ejemplo de la tarjeta SAMV71 del sitio oficial del compilador [12], compilador utilizado es el "*winIDEA*" versión "*Open*" [13]. En el Código de ejemplo de la tarjeta SAMV71 se extrajo el código definido en la aplicación de winIDEA con el nombre de "*getting_started*".

La estructura de archivos de la aplicación "*getting_started*" y del código ejemplo de la SAMV71, se reestructuraron siguiendo la estructura base del estándar AUTOSAR. La estructura de archivos exactamente utilizada en el proyecto es detallada en la Figura 3-2.

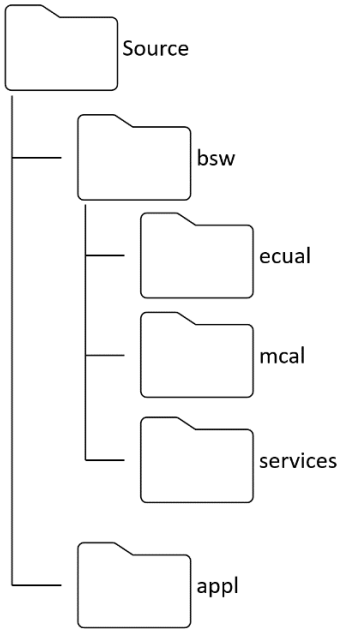

Figura 3-2 Estructura de archivos del proyecto

Bajo la estructura descrita en la Figura 3-2, se desarrolló la implementación completa del proyecto. Al definir la estructura de archivos, se realizó la integración del scheduler que se seleccionó como base al código de la aplicación "*getting_started*" del ejemplo de la tarjeta SAMV71.

3.2. Cambio de Contexto

Para poder realizar el algoritmo de cambio de contexto es necesario definir las restricciones del sistema. La primera condición es que el procesador siempre debe funcionar en modo privilegiado, es decir siempre se tiene acceso a todos los recursos del sistema, por lo que el bit 2 del registro CONTROL se mantendrá en 0 durante la ejecución del programa.

La segunda condición es que la pila de las tareas y del sistema sean totalmente independientes. Para lograr esto es necesario usar la característica de los procesadores Cortex-M que nos permite tener un SP dinámico, y así asignar el MSP exclusivamente al sistema y el PSP a las tareas como se muestra en la Figura 3-3**.**

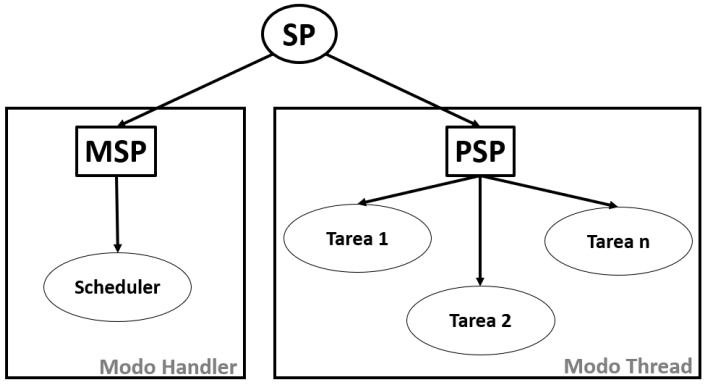

Figura 3-3 Funcionamiento del SP

La independencia de pilas evita que las tareas tengan acceso a las regiones de memoria que le corresponden únicamente al sistema; con esto se evita que las tareas puedan corromper estas regiones de memoria o que si una tarea dejara de funcionar el sistema sea capaz de seguir su operación normal.

3.3.Proceso de mantenimiento de la Pila

Para este proyecto se implementó el modelo TCB, donde cada tarea tiene un elemento que contiene información relacionada con la tarea. Se implementó un TCB que contara con el puntero a su pila, la prioridad de la tarea, un identificador único de cada tarea, un puntero a la función que va a ejecutar la tarea y el estado actual de la tarea, como se muestra en la Figura 3-4.

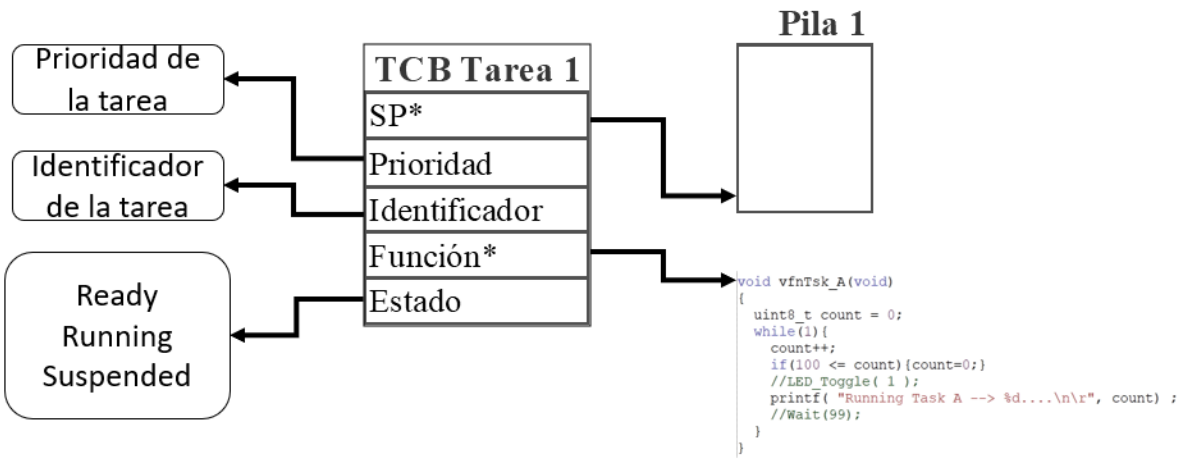

Figura 3-4 Modelo TCB

Con el modelo TCB cada tarea tiene una región exclusiva de memoria para guardar su contexto. Para implementar el modelo TCB se define una estructura del tipo TaskType, que está formada por un puntero a un entero de 32 bits, tres enteros de 8 bits y un puntero a una función como se muestra en la Figura 3-5.

```
typedef struct
{
  uint32_t      *stack;
  uint8_t       taskPriority;
  TaskIdType    taskId;
  FuncPtr       tskFcnPtr;
  TaskStateType taskState;
}TaskType;
```

Figura 3-5 Definición del TCB

El elemento declarado como stack contiene la dirección de memoria de la pila de cada tarea. El elemento taskPriority guarda la prioridad de la tarea, la variable task contiene el identificador único de la tarea, el elemento tskFncPtr es un puntero a la sección de código que ejecutará la tarea cuando sea su turno en el procesador y el elemento taskState será el que indique si la tarea está suspendida, en espera o ejecutada.

3.4. Organización e inicialización de la pila

Como se describió en el marco teórico el proceso de usar una pila de almacenamiento exige básicamente dos condiciones. La primera condición es que los elementos en la pila se guarden de una manera ordenada porque su funcionamiento es LIFO, es decir el último en ser guardado es el primero en salir. La segunda condición indica que el modo de guardar los elementos es descendente o que la memoria se use de una dirección mayor a una menor. Para cumplir estas condiciones es necesario inicializar la pila como se muestra en la Figura 3-6.

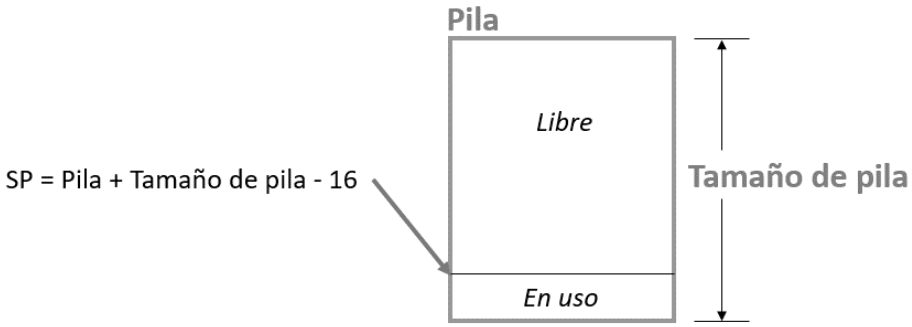

Figura 3-6 Definición de SP de cada tarea

Como podemos observar, el SP ya está considerando el uso de 16 direcciones de memoria. Esto se debe a que se están reservando posiciones de memoria exclusivas para el cambio de contexto, además el guardar el contexto de una tarea exige que cada elemento guardado tenga una posición especifica como se muestra en la Figura 3-7.

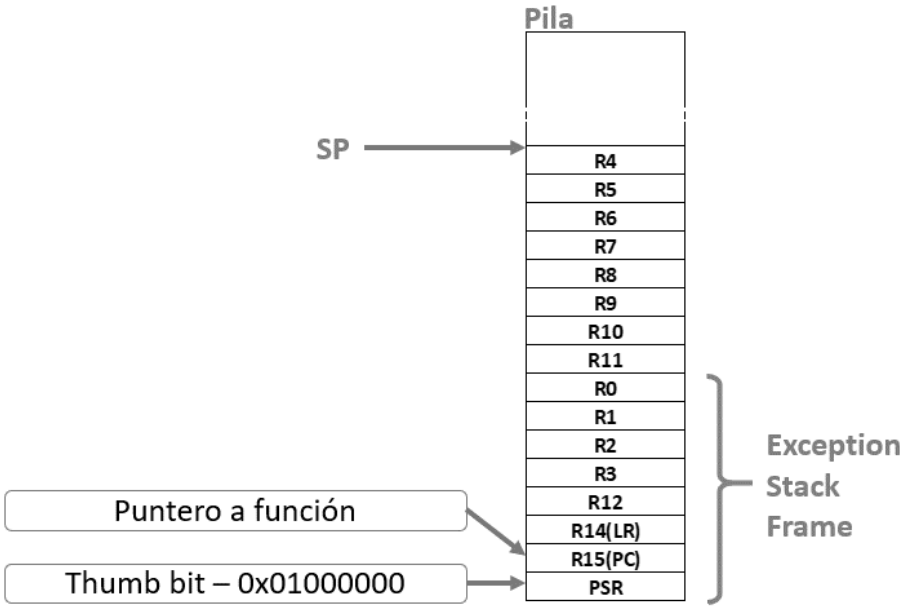

Figura 3-7 Inicialización de la pila de almacenamiento

Las posiciones reservadas corresponden a los registros del procesador y, además, tienen un orden muy específico. Primero se guarda el contexto de manera manual y, después, el *exception frame* es guardado cuando se entra en una interrupción.

En el proceso de inicialización en todas las tareas se deben guardar dos valores clave para el correcto funcionamiento. El primer valor corresponde al puntero de la función de cada tarea y está guardado en la posición del PC, que al guardarlo en esta posición se garantiza que cuando se restaure el contexto de la tarea por primera vez, la tarea se empiece a ejecutar. El segundo valor le indica al procesador que las instrucciones que se van a ejecutar son del tipo Thumb.

Para inicializar la pila de cada tarea se implementó el código que se presenta en la Figura 3-8.

```
void SchM_Init(const SchM_TaskConfigType *SchM_Config)
{
  SchMPtrTaskConfig = SchM_Config;
  sch_current_count = 0;
  uint8_t task_idx;
  uint32_t *Stack = NULL;

  for (task_idx = 0; task_idx < SchMPtrTaskConfig->sch_num_of_task; task_idx++)
  {
    Stack = (uint32_t*) Mem_Alloc(StackSize*sizeof(uint32_t));
    SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sp = Stack + StackSize - 16;
    if(task_idx == 0 )
    {
      SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sp[8] =  SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sch_ptr_to_fcn;
    }
    else
    {
      SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sp[8] = 0xfffffffd;
      SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sp[15] = SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sch_ptr_to_fcn;
      SchMPtrTaskConfig->sch_ptr_task_list[task_idx].sp[16] = 0x01000000;
    }
  }
}
```

Figura 3-8 Código de inicialización de la pila

3.5. Iniciación del sistema

Para iniciar el sistema se necesita implementar una secuencia de configuración del microcontrolador, después se debe configurar el scheduler y, por último, el scheduler inicia la primera tarea como muestra la Figura 3-9.

Figura 3-9 Secuencia de inicialización del sistema

El inicio de la primera tarea es fundamental para implementar el modelo TCB, porque en esta etapa se configura el cambio de MSP a PSP para uso exclusivo de las tareas; también se restaura el contexto de la tarea inicial y, por último, se ejecuta la función de la tarea inicial como se muestra en la Figura 3-10.

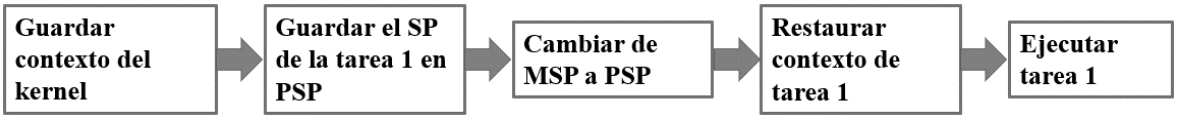

Figura 3-10 Arranque de la primera tarea

Como se mencionó en el marco teórico, el procesador por defecto usa el MSP. Por esta razón, antes de cambiar al nuevo SP es necesario guardar el contexto del núcleo. Esto se hace con la instrucción MRS, que mueve el contenido de un registro de propósito especial a uno de propósito general. Esta primera instrucción guarda el registro de estatus PSR en el registro R12 o IP, después se guarda el contexto de todo el núcleo con la instrucción push como se muestra en la Figura 3-11.

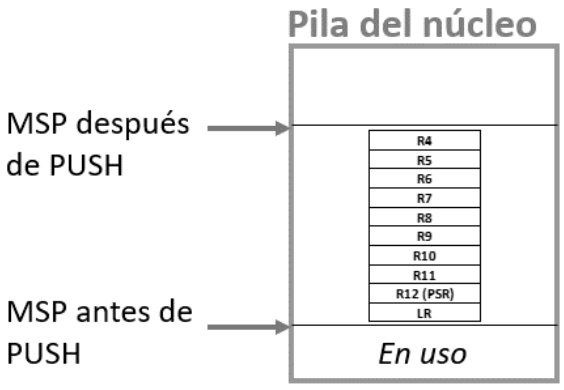

Figura 3-11 Guardar contexto del núcleo

Una vez que el contexto del núcleo está seguro ya se puede hacer el cambio al nuevo SP. Primero se guarda en R0 la dirección del SP de la tarea inicial; después, la instrucción MSR mueve el contenido del registro R0 al registro correspondiente al PSP como se muestra en la Figura 3-12.

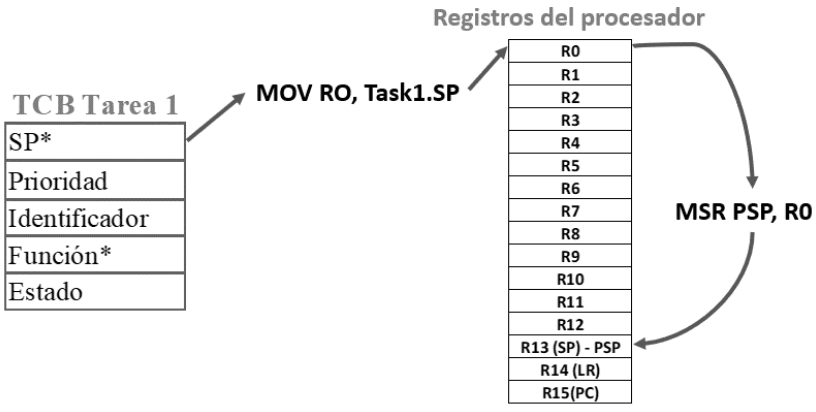

Figura 3-12 Cambio de MSP a PSP

Una vez que el valor del PSP se actualiza es necesario indicarle al procesador que el SP ahora es el PSP. Esto se hace modificando el bit 1 del registro CONTROL, nuevamente la instrucción MSR para cargar el valor 2 en el registro especial CONTROL. Por último, se restaura el contexto de la tarea inicial en los registros de procesador y se ejecuta la función de la tarea inicial como se muestra en la Figura 3-13.

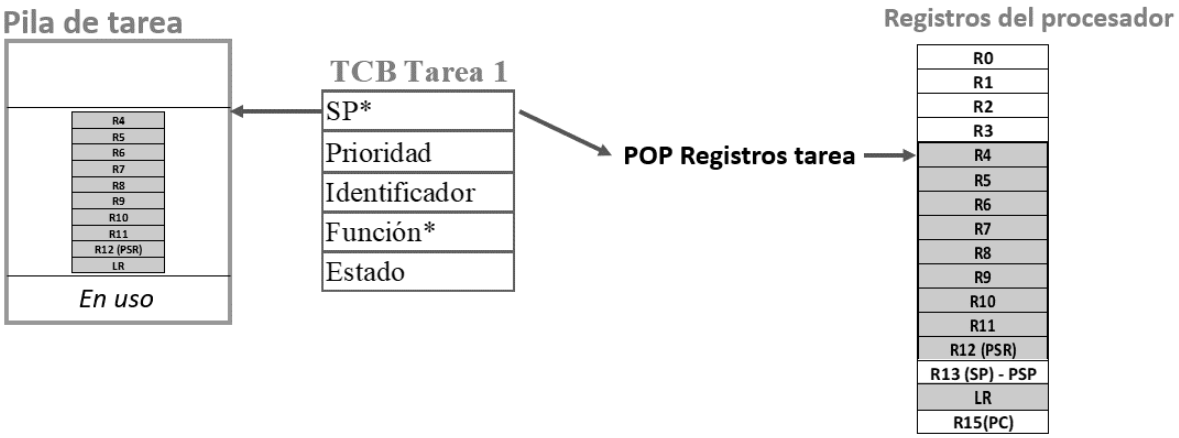

Figura 3-13 Restauración del contexto de la tarea inicial

De esta manera se inicia la primera tarea y continuará su ejecución hasta que el SysTick se desborde, active su interrupción y el scheduler decida la siguiente tarea en ser ejecutada. La inicialización escrita en lenguaje ensamblador se muestra en la Figura 3-14.

```
void thread_start(TaskCtrlType *task_ctrl_array)
{
  my_task_ctrl_array = task_ctrl_array;
  lastTask = 0;

  /* Save kernel context */
  __asm ("mrs ip, psr \n");
  __asm ("push {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr} \n");

  /* switch to process stack */
  asm volatile("MOV     R0, %0\n" : : "r" (my_task_ctrl_array[lastTask].sp));
  __asm ("msr psp, r0 \n");
  __asm ("mov r0, #2 \n");
  __asm ("msr control, r0 \n");
  __asm ("pop {r4, r5, r6, r7, r8, r9, r10, r11, lr} \n");
  my_task_ctrl_array[lastTask].tskFcnPtr();
}
```

Figura 3-14 Código para iniciar el sistema

La primera tarea continuará su ejecución hasta que el reloj del sistema indique que su tiempo de ejecución ha terminado. La primera parte del cambio de contexto inicia cuando la interrupción del Systick guarda de manera automática el Exception Frame; después de ello, el procesador se cambia a modo handler y de manera automática usa el MSP como SP. El proceso de guardado se muestra en la Figura 3-15.

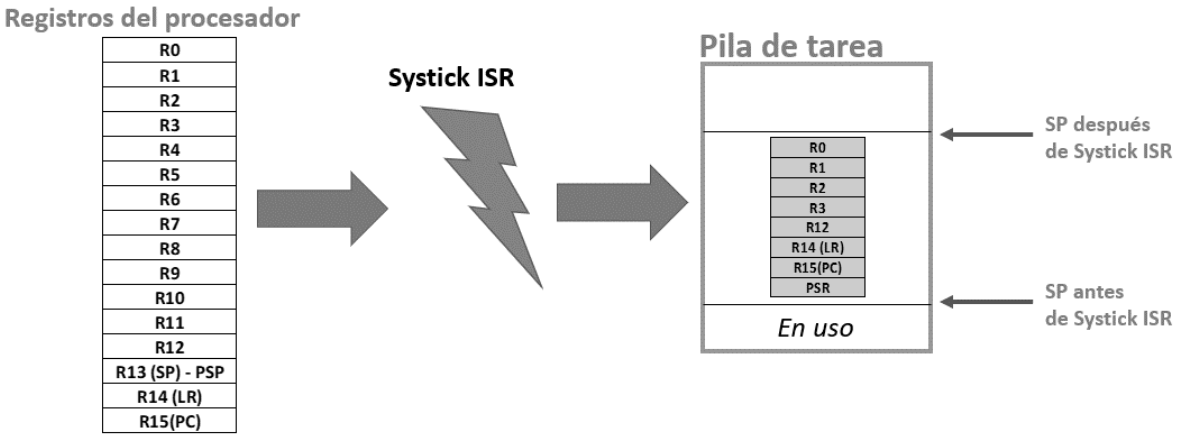

Figura 3-15 Interrupción del Systick

3.6. Implementación del cambio de contexto

Para implementar el cambio de contexto se hace uso de la excepción PendSV_Handler. Es llamada dentro de la rutina de interrupción del Systick con la instrucción SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk.

La segunda parte del cambio de contexto implica guardar manualmente los registros que no lo hicieron de manera automática. El proceso se muestra en la Figura 3-16.

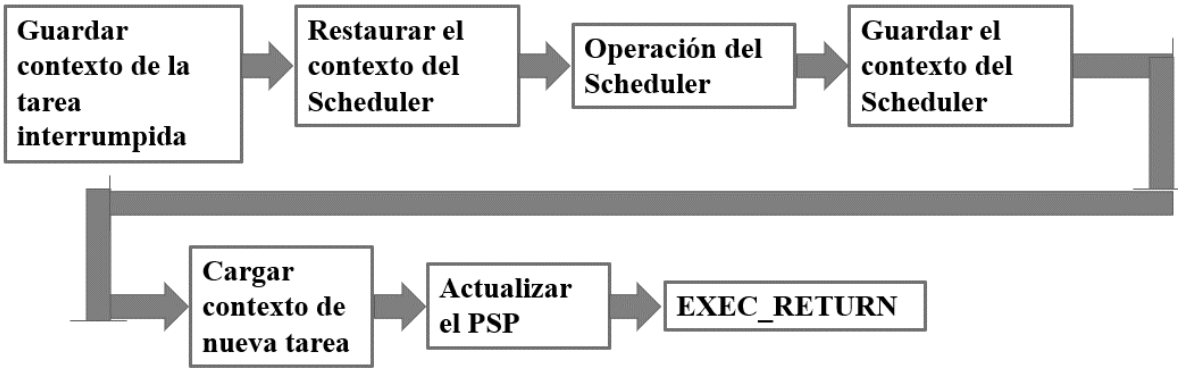

Figura 3-16 Proceso de cambio de contexto

Primero se deben guardar manualmente los registros de la tarea interrumpida, pero el guardado no se puede hacer con la instrucción push debido a que el procesador está en modo handler y, en este modo, solo se puede usar el MSP. Por esta razón es necesario recuperar el valor del PSP con la instrucción MRS; posteriormente, con la instrucción STMDB, se guardan los registros faltantes; y, por último, se actualiza en el TCB de la tarea interrumpida con el nuevo valor del PSP como se muestra en la Figura 3-17.

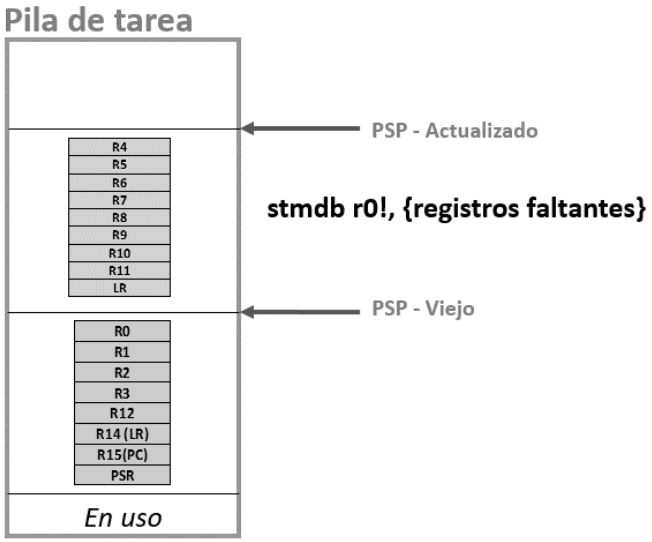

Figura 3-17 Guardado de contexto de la tarea interrumpida

El código en lenguaje ensamblador que realiza el proceso anteriormente descrito se muestra en la Figura 3-18.

```
/* Save the old task's context */
__asm("mrs r0, psp \n");
__asm("stmdb r0!, {r4, r5, r6, r7, r8, r9, r10, r11, lr} \n"); /*  stmdb sav
asm volatile("MOV      %0, R0\n" : "=r" (my_task_ctrl_array[lastTask].sp));
```

Figura 3-18 Código para guardar contexto

La siguiente etapa corresponde a operaciones del scheduler. Primero se restaura el contexto del scheduler usando la operación POP y después se restaura el PSR. Una vez restaurado el contexto del scheduler, este selecciona el TCB de la siguiente tarea a ejecutar y por último debe guardar su contexto porque el trabajo del scheduler ha terminado y es momento de cederle el procesador a la nueva tarea. Las líneas de código que realizan este proceso se presentan en la Figura 3-19.

```
/* load kernel state */
__asm("pop {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr}  \n");
__asm("msr psr, ip \n");

/* Find a new task to run */
while (1) {
  lastTask++;
  if (lastTask >= OS_TASK_MAXNUM){lastTask = 0;}

/* save kernel state */
__asm ("mrs ip, psr \n");
__asm ("push {r4, r5, r6, r7, r8, r9, r10, r11, ip, lr} \n");
```

Figura 3-19 Código para restaurar y guardar contexto del scheduler

El siguiente paso es obtener el SP de la nueva tarea para poder restaurar su contexto. Esta información se encuentra dentro del primer elemento del TCB y la instrucción MOV nos ayuda a obtener la dirección del SP de esta tarea; después, la instrucción LDMIA es usada para restaurar el contexto de la tarea. La instrucción MSR nos ayuda a cargar en el procesador el nuevo valor del PSP e indicarle que ahora se está usando una nueva pila de almacenamiento. Este proceso se muestra en la Figura 3-20.

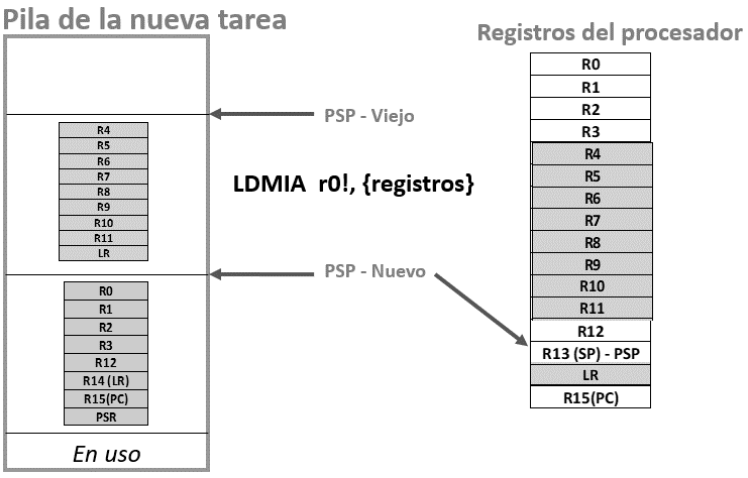

Figura 3-20 Restauración manual del contexto de la nueva tarea

En la última etapa se le indica al procesador cómo va a ser su funcionamiento saliendo del PendSV, que consiste en cargar un valor especial llamado EXEC_RETURN en el LR y hacer la salida de la excepción con la instrucción BX. El valor del EXEC_RETURN depende de tres valores: modo de operación, unidad de punto flotante y SP. En este proyecto no se está usando la unidad de punto flotante, las tareas deben usar el PSP y tienen que funcionar en modo thread; por esta razón el valor del EXEC_RETURN debe ser 0xfffffffd y se muestra en la Figura 3-21.

```
/* load user state */
asm volatile("MOV   | R0, %0\n" : : "r" (my_task_ctrl_array[lastTask].sp));/
__asm ("ldmia r0!, {r4, r5, r6, r7, r8, r9, r10, r11, lr} \n"); //Cargamos d

__asm ("msr psp, r0 \n");//Cambiamos al nuevo PSP

__asm ("mov r0, #0xfffffffd \n");
__asm ("mov lr, r0 \n");

/* Restore the new task's context and jump to the task */

asm volatile("CPSIE   I\n");
asm volatile("BX      LR\n");
```

Figura 3-21 Código para restaurar contexto y ejecutar nueva tarea

Al realizar la instrucción BX con el EXEC_RETURN el procesador toma el PSP como SP, regresa a modo thread y debe restaurar el exception frame. Con esto se puede observar que la tarea comienza su ejecución con todos sus registros restaurados justo en el momento en el cual fue interrumpida, como lo muestra en la Figura 3-22.

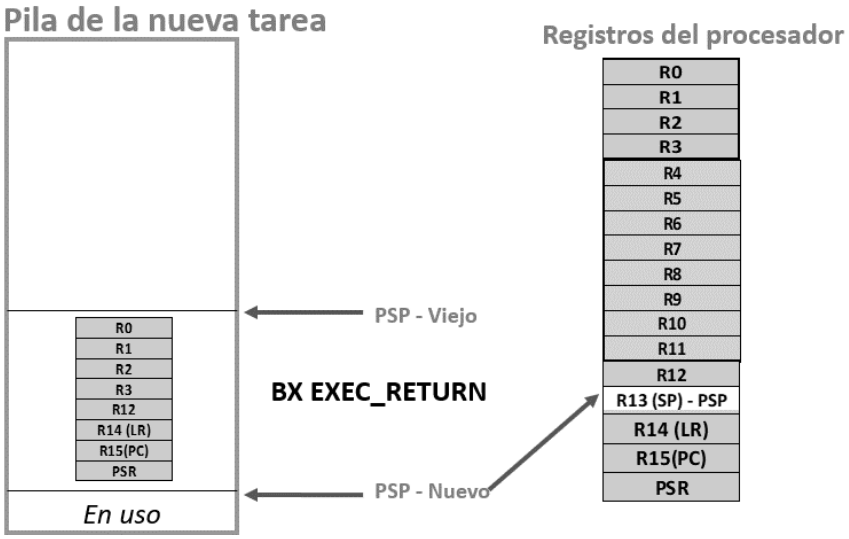

Figura 3-22 Resultado del cambio de contexto

3.7.Modificación del archivo descriptor del ligador del compilador

Para tener un mayor control del manejo de memoria del sistema se propone asignar una región exclusiva para guardar las pilas de memoria de las tareas, esto se logra modificando los archivos LDF (del inglés, *Linker Description File*), que son un conjunto de archivos que le indican al ligador en que regiones de memoria colocar el código objeto del proyecto.

El primer paso es definir una región de memoria en el archivo principal del ligador, en este proyecto la región es llamada HEAP inicia en la posición de memoria 0x20450000 y tiene un tamaño de 65536 bytes como se muestra en la Figura 3-23.

```
/* Memory Spaces Definitions */
MEMORY
{
  rom (rx)   :       ORIGIN = 0x00400000, LENGTH = 0x00200000
  ram (rwx) :       ORIGIN = 0x20400000, LENGTH = 0x00050000
  HEAP (rwx) : ORIGIN = 0x20450000, LENGTH = 0x0010000
  sdram(rwx): ORIGIN = 0x70000000, LENGTH = 0x00200000
}

/* The stack size used by the application. NOTE: you need to ad
STACK_SIZE = DEFINED(STACK_SIZE) ? STACK_SIZE : 0x2000;

/* The heapsize used by the application. NOTE: you need to adju
HEAP_SIZE = DEFINED(HEAP_SIZE) ? HEAP_SIZE : 0x1000;

MY_HEAP_SIZE = DEFINED(MY_HEAP_SIZE) ? MY_HEAP_SIZE : 0x10000;
```

Figura 3-23 Creación de regiones de memoria

También es necesario declarar una variable llamada MY_HEAP_SIZE que corresponda al tamaño de la región creada (0x10000 bytes). El siguiente paso es definir la sección en la que el ligador guardará los archivos que se le indiquen, como se puede observar en la Figura 3-24 en este proyecto no se cargan archivos durante la etapa de compilación y únicamente se usan variables que nos ayudan a inicializar esta sección de memoria en el inicio del programa.

```
. = ALIGN(4);
/* my heap section */
.HEAP (NOLOAD):
{
    . = ALIGN(4);
    MEM_HEAP_START = .;
    . = . + MY_HEAP_SIZE;
    . = ALIGN(4);
    MEM_HEAP_END = .;
} > HEAP
```

Figura 3-24 Asignación de secciones de memoria

La creación de esta región de memoria nos permite usarla para el alojamiento dinámico de memoria.

4. Resultados

4.1. Verificación de inicialización y arranque del scheduler

El correcto funcionamiento del scheduler depende de que las estructuras de configuración, el alojamiento dinámico de la memoria y la inicialización de la pila de memoria de cada tarea se haga correctamente. Para comprobar que el código desarrollado realiza la configuración de manera correcta se hace uso de los archivos generados por el compilador y del depurador contenido en el ambiente de desarrollo WinIDEA, que muestra los registros del procesador y direcciones de memoria en tiempo de ejecución.

El primer archivo para verificar es del tipo ".map" y hace referencia al mapa de memoria que generó el ligador al crear el proyecto. En la Figura 4-1 podemos observar que variables para el reloj del sistema, el cambio de contexto y el alojamiento dinámico de memoria son guardadas consecutivamente en la región de la RAM, también se verifica que la asignación de memoria definida en la metodología fue exitosa porque inicia en 0x20450000 y termina en la dirección 0x20460000.

```
COMMON          0x204009d8        0x4 C:\Users\hao_2\Desktop\Especialidad_SE\Tesis\Tesis_Manuel\BaseProject\workspace\Debug\Context_Switch.
                0x204009d8              my_task_ctrl_array
COMMON          0x204009dc       0x10 C:\Users\hao_2\Desktop\Especialidad_SE\Tesis\Tesis_Manuel\BaseProject\workspace\Debug\mem_alloc.o
                0x204009dc              New_Heap
COMMON          0x204009ec        0x4 c:/isystem/winideaopen9/gcc/arm/bin/../lib/gcc/arm-none-eabi/4.9.3/../../../../arm-none-eabi/lib/cort
                0x204009ec              errno
                0x204009f0              . = ALIGN (0x4)
                0x204009f0              _ebss = .
                0x204009f0              _ezero = .

.stack          0x204009f0     0x2000 load address 0x0040a2c8
                0x204009f0              . = ALIGN (0x8)
                0x204009f0              _sstack = .
                0x204029f0              . = (. + STACK_SIZE)
*fill*          0x204009f0     0x2000
                0x204029f0              . = ALIGN (0x8)
                0x204029f0              _estack = .

.heap           0x204029f0     0x1000 load address 0x0040c2c8
                0x204029f0              . = ALIGN (0x8)
                0x204029f0              _sheap = .
                0x204039f0              . = (. + HEAP_SIZE)
*fill*          0x204029f0     0x1000
                0x204039f0              . = ALIGN (0x8)
                0x204039f0              _eheap = .
                0x204039f0              . = ALIGN (0x4)

.HEAP           0x20450000    0x10000
                0x20450000              . = ALIGN (0x4)
                0x20450000              MEM_HEAP_START = .
                0x20460000              . = (. + MY_HEAP_SIZE)
*fill*          0x20450000    0x10000
                0x20460000              . = ALIGN (0x4)
                0x20460000              MEM_HEAP_END = .
                0x20460000              . = ALIGN (0x4)
                0x20460000              _end = .
                0x2044ffff              _ram_end_ = ((ORIGIN (ram) + 0x50000) - 0x1)
                0x20460000              _sdram_lma = .
```

Figura 4-1 Creación de región exclusiva de memoria

La funcionalidad de la interfaz de inicialización del scheduler y de los TCBs puede ser comprobada al comparar la información que se presenta en la Tabla 4-1 con la información que el depurador muestra en tiempo de ejecución

Tabla 4-1 Tabla de relación tarea y pila de memoria

Tarea	Inicio de pila	Tamaño de pila	Fin de pila	SP inicial
task_Task_3p125ms	0x20450000	0x7D0	0x204507D0	0x20450790
task_Task_6p25ms	0x204507D0	0x7D0	0x20450FA0	0x20450FA0
task_Task_12p5ms	0x20450FA0	0x7D0	0x20451770	0x20451770
task_Task_25ms	0x20451770	0x7D0	0x20451F40	0x20451F40
task_Task_50ms	0x20451F40	0x7D0	0x20452710	0x20452710
task_Task_100ms	0x20452710	0x7D0	0x20452EE0	0x20452EE0

En el código de la inicialización de la pila de la primer tarea únicamente nos indica que se tiene que copiar en el "SP[8]" la dirección de la función de la primer tarea. En la Figura 4-2 se verifica que el "SP[8]" apunta a la dirección *0x00402641*, que corresponde a la sección de código de la primera tarea.

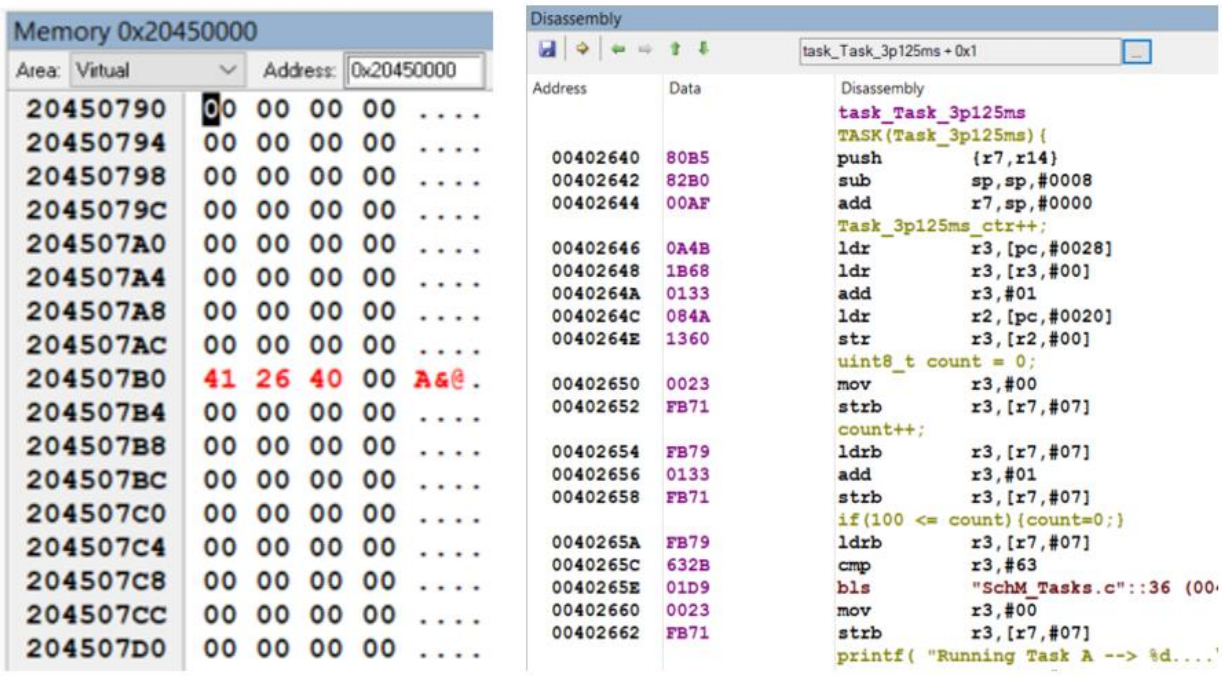

Figura 4-2 Inicialización de pila de memoria de la primera tarea

A partir de la segunda tarea la inicialización es completamente diferente porque no solo se carga el valor de la función en la posición correspondiente al PC, también se carga el valor del retorno de la excepción en la posición ocho del SP y el valor que indica el tipo de instrucciones a ejecutar. En la Figura 4-3 se comprueba que la inicialización de las tareas y su pila de memoria se inicializan correctamente.

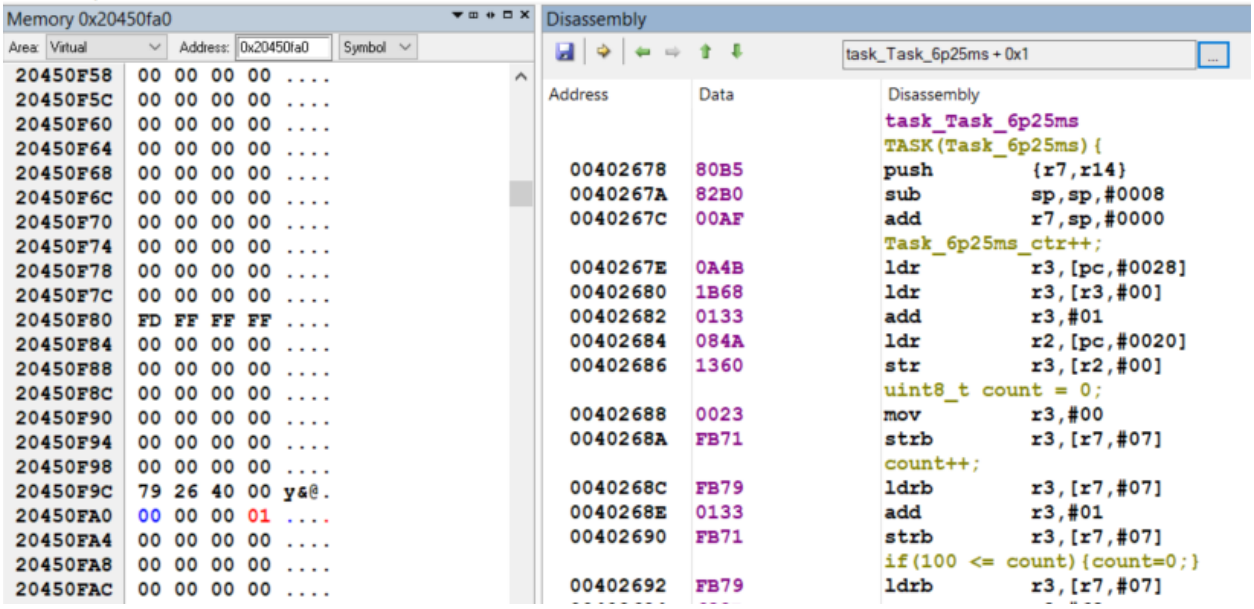

Figura 4-3 Inicialización de pila de memoria del resto de las tareas

4.2. Verificación de arranque del sistema

El arranque del sistema inicia con la ejecución de la primera tarea, es en este punto donde se cambia el SP de MSP a PSP y el procesador se configura para que las tareas funcionen en modo thread y el scheduler en modo handler. La Figura 4-4 muestra que el registro CONTROL está configurando el procesador para usar el PSP.

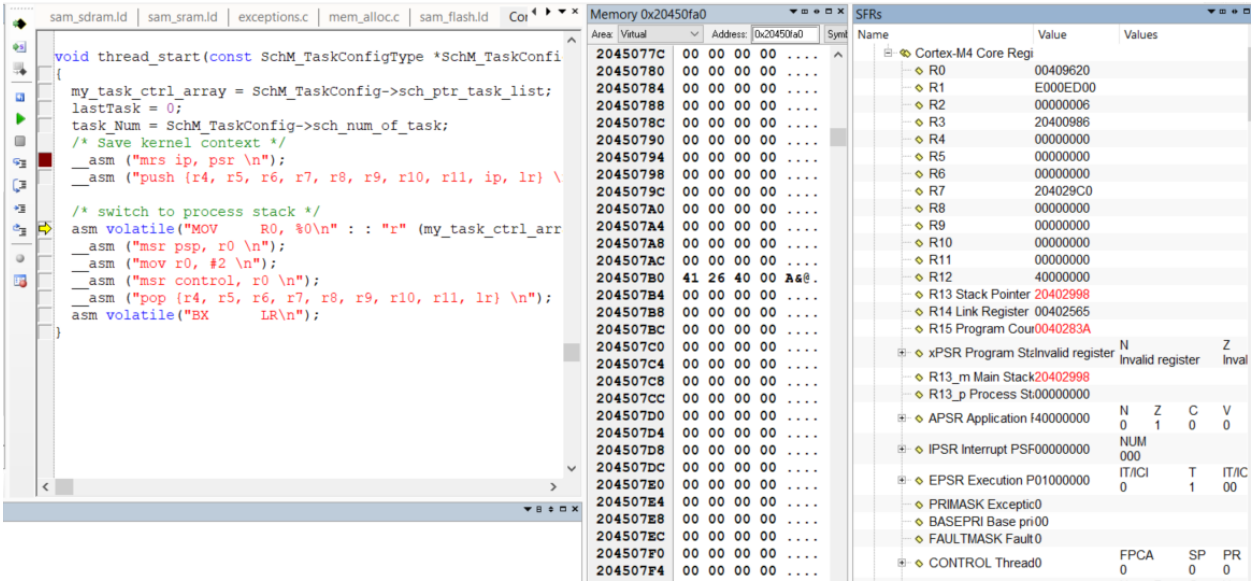

Figura 4-4 SP en modo MSP

Una vez que la instrucción de cambio de SP es ejecutada se comprueba que el registro CONTROL cambia y ahora R13(SP) está apuntando a la pila de memoria de la tarea inicial como se muestra a continuación.

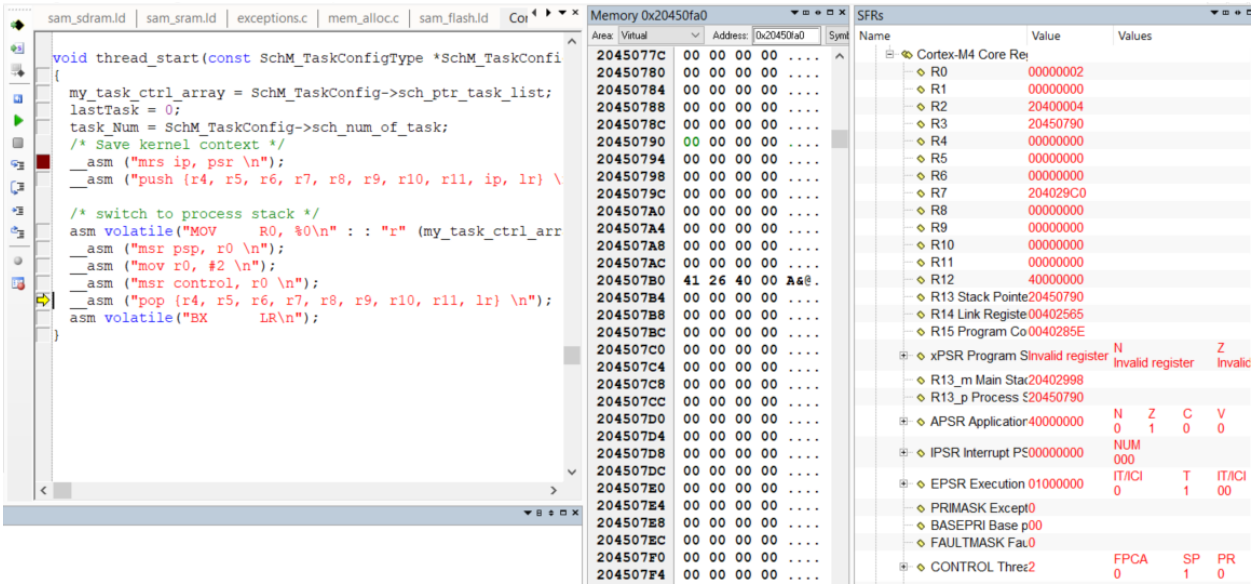

Figura 4-5 SP en modo PSP

La tarea continua su ejecución hasta que la interrupción del SysTick indique el cambio de contexto y le ceda el procesador a la siguiente tarea.

4.3. Verificación de cambio de contexto

La tarea que tenga posesión del procesador se ejecutará hasta que la interrupción del Systick se desborde, en este momento el procesador cambia a modo handler y comienza a usar el MSP como se puede apreciar a continuación.

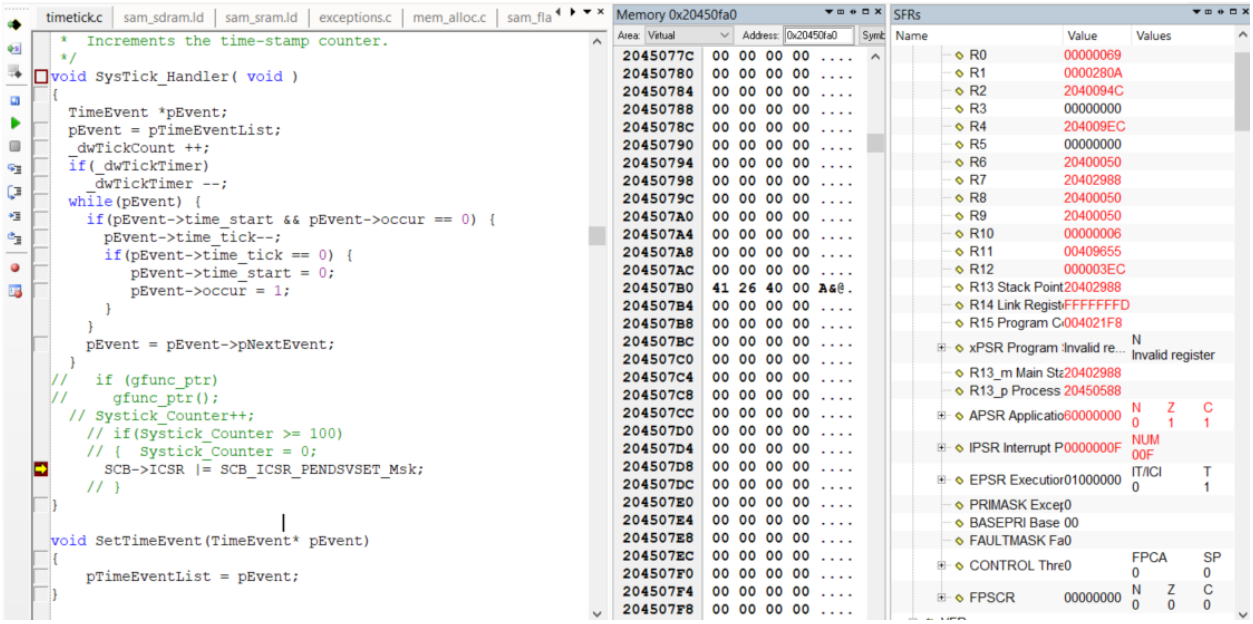

Figura 4-6 SP al momento de Systick ISR

En la ISR del Systick se activa la bandera que activa la excepción PendSV y realiza el cambio de contexto entre la tarea interrumpida y la nueva tarea. En la Figura 4-7 se muestra como el cambio de contexto se ejecutó correctamente porque los valores de los registros del procesador corresponden a los que fueron cargados en la inicialización y el valor del PSP se actualizó al de la nueva tarea. Por último, el valor de EXEC_RETURN nos indica que se regresa al modo thread y se usa el PSP.

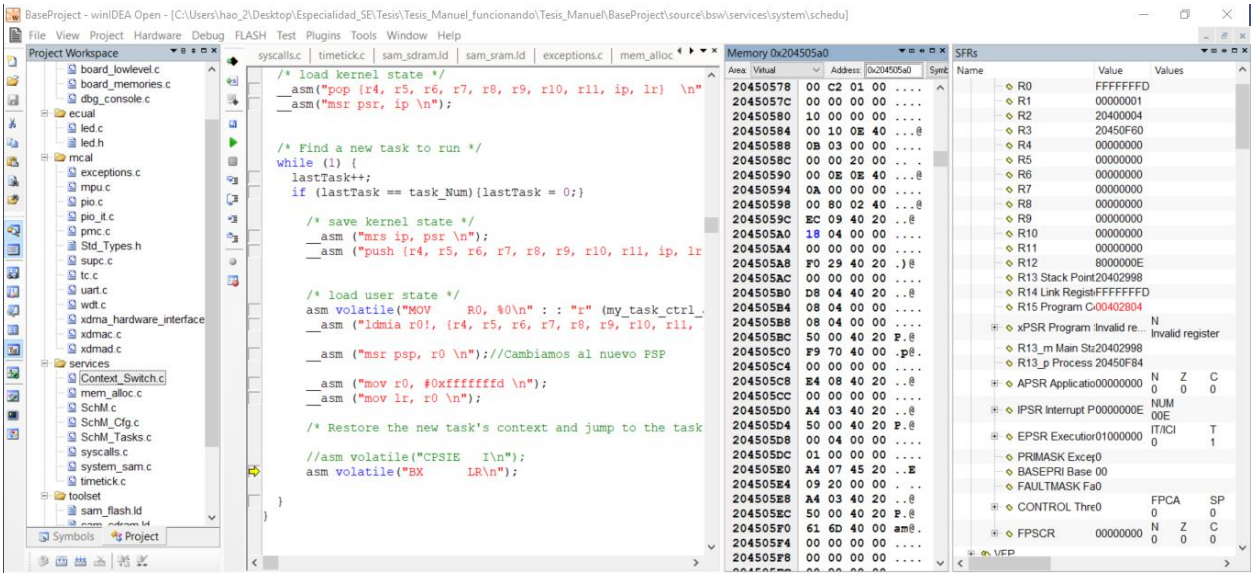

Figura 4-7 SP Después de cambio de contexto

Para comprobar el funcionamiento del cambio de contexto entre tareas de una manera más clara, se implementaros tres tareas que tuvieran una variable local e imprimieran esta variable por la terminal serial como se indica en la Figura 4-8.

```
void vfnTsk_A(void)
{
  uint8_t count = 0;
  while(1){
    count++;
    if(100 <= count){count=0;}
    printf( "Running Task A --> %d....\n\r", count) ;
  }
}

void vfnTsk_B(void)
{
  uint8_t count = 0;
  while(1){
    count++;
    if(100 <= count){count=0;}
    printf( "Running Task B --> %d....\n\r", count) ;
  }
}
void vfnTsk_C(void)
{
  uint8_t count = 0;
  while(1){
    count++;
    if(100 <= count){count=0;}
    printf( "Running Task C --> %d....\n\r", count) ;
  }
}
```

Figura 4-8 Tareas de prueba

En la terminal serial (Figura 4-9) podemos observar que cada tarea puede enviar su mensaje al menos 4 veces y en ese momento ocurre la interrupción del Systick hace el cambio de contexto e inicia la siguiente tarea.

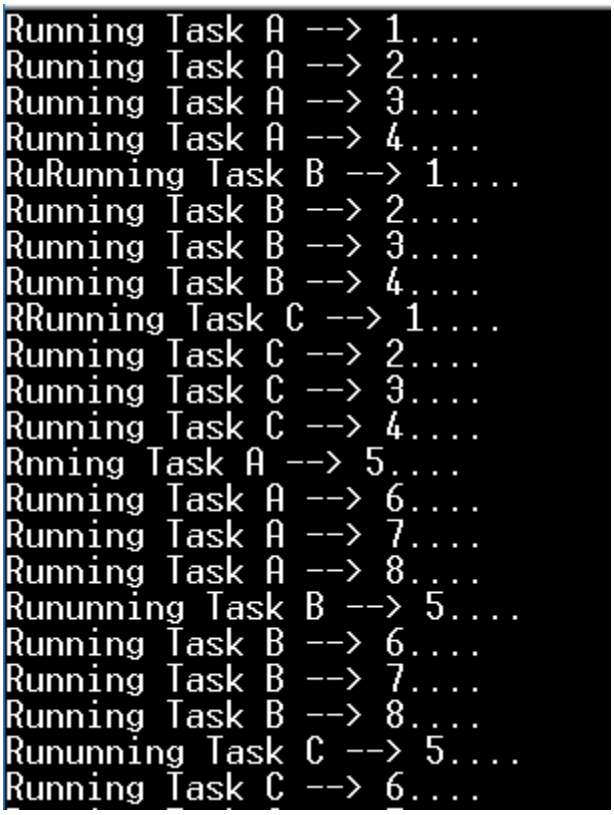

Figura 4-9 Resultados de cambio de contexto

En la siguiente imagen (Figura 4-10 Resultados de cambio de contexto) se aprecia como el cambio de contexto se sigue ejecutando exitosamente a pesar de que la tarea se interrumpa en cualquier

momento, ya que si observa detenidamente la tarea continua su ejecución justo en el momento donde fue interrumpida.

Figura 4-10 Resultados de cambio de contexto

5. Discusión

Como se describió en la sección 4.1 la inicialización de la pila de memoria de cada tarea no solo implica en reservar una variable para almacenar datos, es muy importante conocer el modelo de programación del procesador para saber exactamente en qué posición ubicar cada elemento de la tarea para que cuando la restauración de contexto ocurra el procesador tenga la información correcta para funcionar por primera vez o continuar su ejecución exactamente donde fue interrumpido.

El propósito de la comprobación a detalle del arranque del sistema en la sección 4.2 se debe a que en esta parte se configuran todos los registros necesarios para asignar los modos de procesamiento *handler* y *thread*, además se hace la activación del PSP para generar una separación entre la pila de memoria del scheduler y de las tareas, esta separación nos permite que las tareas no puedan corromper toda la memoria.

En la parte de comprobación del cambio de contexto se observa que a pesar de que varias tareas comparten el procesador, las tareas siempre continúan su ejecución como si nunca fueran interrumpidas y generan resultados correctos, ya que todas sus variables internas tienen persistencia en memoria. La ventaja que presenta un sistema que cuente con cambio de contexto sobre uno sin cambio de contexto, es que el usuario puede implementar tareas con tiempo de ejecución mayor al de su tiempo procesamiento, ya que en sistemas sin cambio de contexto las tareas pierden toda su información local al momento de ser interrumpidas.

6. Conclusión

Como resultado de este trabajo, se obtuvo un scheduler que puede ser usado en aplicaciones donde se necesite ejecutar múltiples tareas con diferentes tiempos de ejecución, ya que el cambio de contexto permite que una tarea pueda ser completada a pesar de ser interrumpida en múltiples ocasiones. Las etapas que se concluyeron exitosamente son las siguientes:

- Arquitectura de software dividida en capas funcionales.
- Implementación de un scheduler escalable en el número de tareas.
- Implementación del modelo TCB para las tareas.
- Creación y asignación de regiones de memoria para uso exclusivo.
- Algoritmo de cambio de contexto.

El scheduler utiliza apuntadores y estructuras que permiten que el número de tareas sea configurable y se puedan agregar o eliminar tareas únicamente agregando elementos a una estructura de configuración; Además, esta configuración permite que se agreguen elementos a las tareas con modificaciones menores al código.

Una característica importante del scheduler que se propone, es el uso de la excepción PendSV porque permite que el cambio de contexto pueda ser interrumpido por interrupciones que tengan una mayor prioridad y la asignación de una región exclusiva para la pila de memoria desde los archivos LDF permite un mayor control de la memoria de las tareas.

El entender y tener un código que implemente el cambio de contexto es un referente a un punto de partida para trabajos posteriores de investigación sobre la optimización del cambio de contexto, agregándole valor a la propuesta de scheduler presentada en este trabajo.

Finalmente, el objetivo de hacer uso de todos los conocimientos aprendidos en el posgrado se cumplió, además este trabajo puede servir a futuros estudiantes que necesiten entender el cambio de contexto porque puede llegar a ser un proceso confuso.

7. Referencias

[1] P. A. Laplante, Real-time systems design and analysis, New York: Wiley, 2004.

[2] J. Yiu, The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors, Newnes, 2013.

[3] Z. L. H. G. Z. S. J. &. L. J. Wu, «An improved method of task context switching in OSEK operating system,» *In Advanced Information Networking and Applications, 2006,* vol. 1, pp. 6-pp, 2006.

[4] G. &. C. R. Chanteperdrix, "The ARM fast context switch extension for Linux," *In Real Time Linux Workshop,* 2009.

[5] A. T. H. U. V. &. H. G. Wiggins, «Implementation of fast address-space switching and TLB sharing on the StrongARM processor.,» *Asia-Pacific Conference on Advances in Computer Systems Architecture* , pp. 352-364, 2003.

[6] R. K. S. H. V. &. M. G. Belagali, «Implementation and validation of dynamic scheduler based on LST on FreeRTOS,» *Electrical, Electronics, Communication, Computer and Optimization Techniques (ICEECCOT),* pp. 325-330, 2016.

[7] A. A. &. P. B. A. S. Paul, "Reducing the number of context switches in real time systems," *In Process Automation, Control and Computing (PACC),* pp. 1-6, 2011.

[8] Arm: Architecting a Smarter World, «Arm: Catalog,» arm.com, S.F.. [En línea]. Available: https://www.arm.com/. [Último acceso: 2018 Julio 02].

[9] Arm: Architecting a Smarter World, «ARM Cortex-M7 Devices Generic User Guide,» Arm, S.F.. [En línea]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/CHDDIGAC.html. [Último acceso: 2018 Junio 02].

[10] Y. Zhu, Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C (Third Edition), E-Man Press LLC, 2017.

[11] A. G. Dean, Embedded Systems Fundamentals with ARM Cortex-M Based Microcontrollers: A Practical Approach., UK: ARM Education Media, 2017.

[12] iSYSTEM, «winIDEA/testIDEA - iSYSTEM - Enabling Safer Embedded Systems,» iSYSTEM, S.F,. [En línea]. Available: https://www.isystem.com/downloads/winidea.html. [Último acceso: 10 Junio 2018].

[13] iSYSTEM, «winIDEA Open Free IDE,» iSYSTEM, S.F.. [En línea]. Available: https://www.isystem.com/products/software/winidea-open.html. [Último acceso: 3 MAYO 2018].