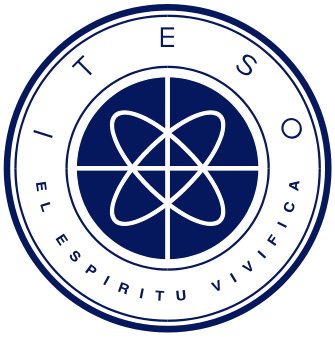
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

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 PARA SISTEMAS OPERATIVOS EN TIEMPO REAL

Tesina para obtener el grado de:

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presentan: Mitsuki Hino Aguilar
Luis Enrique Preciado Bautista

Tutor: M.C. Héctor Antonio Rivas Silva

San Pedro Tlaquepaque, Jalisco. Diciembre 2017.

Agradecimientos

Primeramente, queremos dar gracias a Dios, que nos dio la salud y los medios para poder estar en esta excelente universidad como lo es el ITESO. También queremos dar gracias por el apoyo incondicional de nuestras familias ya que estuvieron en las buenas y malas para poder lograr este objetivo tan importante en nuestras vidas.

Agradecemos al ITESO por este programa de especialización que nos ha permitido desarrollar habilidades que podemos poner en práctica en nuestros trabajos ya que es realmente algo que nos gusta hacer día con día. Al Maestro Abraham Tezmol Otero, quien brindó su experiencia y soporte en el presente trabajo y al Maestro Héctor Rivas Silva quien nos asesoró en todo momento brindando su conocimiento y apoyo incondicional.

Finalmente, agradecemos al CONACYT por el apoyo que nos brindó económicamente y por haber confiado en nosotros para realizar nuestros estudios con los siguientes números de registro:

Mitsuki Hino Aguilar CVU: 723549.
Luis Enrique Preciado Bautista CVU: 829884.

Abstract

In an operating system, the most important component in charge of deciding which task runs next, is the scheduler. The scheduler must be precise at the time of granting the execution time for the tasks in order to not to affect other tasks in their execution time, and thus, have a stable scheduler and avoid latency in the operating system. The present work explains the implementation of a scheduler in an ATSAMV71Q21 microcontroller and a SAMV71 development card, both from ATMEL, by means of the property of the binary numbers, since the sequence of binary numbers is repetitive, the objective is to perform 6 tasks, 3 independent and 3 dependent. Each of these tasks is executed in due time by a counter that increases every 500 microseconds. This counter is controlled by a system timer, configured to generate interruptions and each interruption can increase the counter, which will tell us which task is still running. Since the timer of the system is independent of the other peripherals of the microcontroller, there is an exact interruption and thus, an activation of the task in the calculated time is generated, obtaining a stable and reliable scheduler.

Resumen

En un sistema operativo, la parte más importante que realiza la toma de decisiones de que tarea es la siguiente en ejecutarse, es el scheduler. El scheduler debe ser exacto a la hora de otorgar tiempo de ejecución a las tareas para no afectar a las demás tareas en su tiempo de ejecución y así poder tener un scheduler estable y evitar la latencia en el sistema operativo. El presente trabajo propone la implementación de un scheduler en un microcontrolador ATSAMV71Q21 y una tarjeta de desarrollo SAM V71, ambos de ATMEL, mediante la propiedad de los números binarios, ya que la secuencia de los números binarios es repetitiva, se toma ventaja de ello para poder realizar 6 tareas, 3 independientes y 3 dependientes. Cada una de estas tareas se ejecuta en su debido tiempo mediante un contador que se incrementa cada 500 microsegundos. Este contador es controlado por un temporizador del sistema, configurado para que genere interrupciones y cada interrupción pueda incrementar el contador, el cual nos dirá que tarea es la que sigue en ejecutarse. Al ser el temporizador del sistema independiente de los demás periféricos del microcontrolador, se tiene una interrupción exacta, generando así una activación de la tarea en el tiempo calculado, obteniendo un scheduler bastante estable y confiable.

Lista de figuras

Lista de tablas

Abreviaturas y acrónimos

API	Interfaz de Programa de Aplicación (del inglés, *Application Program Interface*)
AUTOSAR	Arquitectura de Sistemas Abierta para el sector del Automóvil (del inglés, *AUTomotive Open System ARchitecture*)
CAN	Controlador de Área de Red(del inglés, Contoller Area Network)
CPU	Unidad de Procesamiento Central(del inglés, *Cental Process Unit)*
DMA	Acceso Directo a Memoria (del inglés, *Direct Memory Access*)
GPU	Unidad de Procesamiento Gráfico (del inglés, *Graphic Processing Unit*)
GUI	Interfaz Gráfica de Usuario (del inglés, *Graphic User Interface*)
LED	Diodo Emisor de Luz (Light-emitting diode)
LIN	Red Interconectada Local (del inglés, Local Interconnect Network)
MCU	Microcontrolador (del inglés,Microcontroller)
OS	Sistema Operativo (del inglés, *Operative System*)
RAM	Memoria de Acceso Aleatorio (del inglés, *Random Access Memory*)
ROM	Memoria de Solo Lectura (del inglés, *Read Only Memory*)
RTE	Entorno de Ejecución (del inglés, *Runtime Environment*)
RTOS	Sistema Operativo en Tiempo Real (del inglés, *Real Time Operating System*)

Contenido

Introducción

Un sistema operativo es un conjunto de rutinas y archivos de programa que controla los recursos de una computadora y proporciona acceso a los servicios de una computadora. Específicamente, un sistema operativo permite que los componentes de hardware de una computadora, incluidos los procesadores y las unidades, se comuniquen con sus componentes de software, como son las aplicaciones y los conjuntos de instrucciones de datos. En las computadoras personales, estaciones de trabajo y otros dispositivos informáticos modernos, los sistemas operativos son componentes esenciales sin los cuales las computadoras no pueden funcionar [1].

Los ingenieros originalmente desarrollaron sistemas operativos como un medio para superar el tiempo de inactividad de las unidades centrales de procesamiento. Estas unidades centrales de procesamiento son responsables de ejecutar comandos de computadora, e incluso las unidades más lentas pueden procesar datos en microsegundos o millonésimas de segundo. En comparación, otros componentes de la computadora, como los discos duros, son considerablemente más lentos. Para evitar que las unidades centrales de procesamiento dejen de funcionar o permanezcan inactivas hasta que otros componentes tengan la oportunidad de responder, los ingenieros crearon los sistemas operativos. Estos sistemas superan el tiempo de inactividad ordenando o secuenciando automáticamente las unidades centrales de procesamiento para completar tareas sucesivas mientras esperan la finalización de las tareas anteriores. El proceso que utilizan los sistemas operativos para las tareas a ejecutar se conoce como multiprogramación [1].

La multiprogramación se logra mediante la parte más importante de un sistema operativo, conocido como el scheduler o planificador, el cual administra los procesos que se ejecutan en la unidad de procesamiento central y elimina las tareas ya ejecutadas, liberando el espacio de la memoria y la carga al procesador. Dichos sistemas operativos permiten que se cargue más de un proceso en la memoria ejecutable a la vez y el proceso cargado comparte el procesador mediante el multiplexado de tiempo [2].

Un scheduler en un sistema operativo en tiempo real y está diseñado para proporcionar un patrón de ejecución predecible (normalmente descrito como determinista). Esto es particularmente de interés para los sistemas embebidos ya que a menudo tienen requisitos en tiempo real. Un requisito de tiempo real es aquel que especifica que el sistema integrado debe responder a un determinado evento dentro de un tiempo estrictamente definido (la fecha límite). Solo se puede garantizar que se cumplan los requisitos de tiempo real si se puede predecir el comportamiento del programador del sistema operativo, y por lo tanto, es determinista [3].

Para lograr un scheduler determinista y en tiempo real se realizará un intercambio entre la reutilización de los núcleos embebidos, el paralelismo en el hardware y la complejidad de tiempo del algoritmo del scheduler. Se propone utilizar un contador de 16 bits, dada la arquitectura reconfigurable, donde el objetivo es implementar un scheduler en la tarjeta de evaluación ATMEL SAMV71 Xplained Ultra que brinda la academia a lo largo de la Especialidad en Sistemas Embebidos en el Instituto Tecnológico y de Estudios Superiores de Occidente (ITESO) [4].

El scheduler se encargará de manejar y repartir los tiempos de ejecución instantáneos de manera que tenga la capacidad de configurar y activar tareas en tiempo de 1 ms, 2A ms, 2B ms, 10 ms, 20 ms, 50 ms, 100 ms, y podrá atender tareas de alta y baja prioridad dependiendo de la configuración y la prioridad dada a cada tarea. Finalmente, el scheduler tendrá un reloj de 500us que marcará la pauta y la activación de las tareas de los tiempos descritos anteriormente.

1. Antecedentes

Un primer trabajo relacionado a los sistemas operativos es de Mario Aldea Rivas (2002), quien propone un scheduler para tareas en sistemas operativos de tiempo real, el cual se centra en los mecanismos proporcionados por los sistemas operativos y utilizando algoritmos, logra una mejora de la prioridad de las tareas a ejecutar. Rivas utiliza además mutexs para controlar los recursos del procesador, y este es implementado por una tarea de planificación que lo hace más eficiente y confiable a la hora de liberar recursos y restringir su uso.

El proyecto es desarrollado en el lenguaje embebido C, ya que ayuda a optimizar el espacio de memoria del microcontrolador. Rivas propone un sistema operativo denominado Marte OS para realizar las pruebas al scheduler propuesto ya que es posible especificar el máximo número de procesos que pueden existir al mismo tiempo, el número de prioridades, la longitud máxima de colas de tareas en espera, el máximo de temporizadores, el número máximo de señales pendientes, el stack de cada tarea y la memoria dinámica del sistema [5].

Un segundo trabajo de Luis Eduardo Leyva del Foyo (2008), propone un sistema de control de administración de interrupciones para sistemas operativos en tiempo real bajo una planificación y análisis de factibilidad. Este sistema está basado en la utilización total del procesador al aplicar una variedad de conjuntos de tareas periódicas que son planificadas usando un algoritmo. El algoritmo asigna una proporción a los ciclos de trabajo del procesador y ayuda a definir que entre menor el periodo, mayor la prioridad de la tarea. Además, utiliza puntos de planificación y análisis de tiempos de respuesta las cuales se utilizan para establecer que los desfases cumplan con los tiempos de cada tarea y que no sobrepasen su tiempo de trabajo. Mediante las interrupciones del sistema, se controla el tiempo de cada tarea y esa petición de interrupción es generada mediante el hardware [6].

Una última referencia de Glenn A. Elliott (2015), habla sobre el scheduler en tiempo real para aplicaciones avanzadas en la industria automotriz para el GPU - Unidad de Procesamiento Gráfico (del inglés, *Graphic Processing Unit*), en donde argumenta que las unidades de GPU

representan una alternativa factible, pero se deben desarrollar nuevos algoritmos y técnicas analíticas para integrar estos procesadores únicos limitados en un sistema en tiempo real [7].

En este trabajo realizado por Elliot, se desempeñó una investigación a partir los problemas que impiden el uso de las GPU en sistemas en tiempo real. El autor diseña e implementa un programador de múltiples GPU en tiempo real, denominado GPUSync. Este programador controla de manera estricta el acceso a los procesadores computacionales y DMA – Acceso Directo a Memoria (del inglés, *Direct Memory Access*) de una GPU, lo que permite el uso simultáneo a pesar de las posibles limitaciones en el hardware de la GPU. Así mismo, el programador permite que las tareas migren entre las GPU, autorizando nuevas clases de plataformas de computación multi-GPU en tiempo real. GPUSync emplea un scheduler para guiar las decisiones de programación y mejorar la eficiencia del sistema sin correr el riesgo de incumplir las restricciones en tiempo real. También puede combinarse con una amplia variedad de programadores de CPU – Unidad de Procesamiento Central (del inglés, Central Processing Unit) comunes en tiempo real. Finalmente, GPUSync admite tiempos de ejecución y controladores de GPU de código cerrado sin pérdida de funcionalidad [7].

Derivado de lo anterior, existen muchos trabajos de investigación donde se explica el cómo mejorar un sistema operativo en tiempo real mediante la optimización del scheduler para una mejor toma de decisiones en las tareas y no hacer el sistema lento por pérdidas de tiempo en la lógica programada. Es por ello, que se pretende desarrollar un scheduler capaz de otorgar de manera precisa la toma de decisiones sin pérdida de tiempo en la ejecución de las tareas, permitiendo que los tiempos de frecuencia del core se desempeñen con una eficiencia óptima.

2. Marco Teórico

2.1 Sistema Operativo

Un OS - Sistema Operativo (del inglés, *Operative System*) es el programa que después de haber sido cargado inicialmente en la computadora por un gestor de arranque, administra todos los demás programas. Los otros programas se denominan aplicaciones o programas de aplicación. Los programas de aplicación hacen uso del sistema operativo realizando solicitudes de servicios a través de una API - Interfaz de Programa de Aplicación (del inglés, *Application Program Interface*) definida. Además, los usuarios pueden interactuar directamente con el sistema operativo a través de una interfaz de usuario, como una línea de comandos o una GUI - interfaz gráfica de usuario (del inglés, *Graphic User Interface*) [8]. Un sistema operativo realiza los siguientes servicios para las aplicaciones:

- En un sistema operativo multitarea donde se pueden ejecutar varios programas al mismo tiempo, el sistema operativo determina qué aplicaciones deben ejecutarse, en qué orden y cuánto tiempo debe permitirse para cada aplicación antes de darle otro turno a otra aplicación.
- Administra el intercambio de memoria interna entre múltiples aplicaciones.
- Maneja la entrada y salida desde y hacia dispositivos de hardware conectados, como discos duros, impresoras y puertos de acceso.
- Envía mensajes a cada aplicación o usuario interactivo (o a un operador del sistema) sobre el estado de la operación y cualquier error que pueda haber ocurrido.
- Puede liberar la gestión de los llamados trabajos por lotes (por ejemplo, imprimir) para que la aplicación inicial se libere de este trabajo.
- En las computadoras que pueden proporcionar procesamiento en paralelo, un sistema operativo puede administrar cómo dividir el programa para que se ejecute en más de un procesador a la vez [8].

2.1.1 Sistemas operativos más comunes

Windows es el sistema operativo insignia de Microsoft, el estándar para computadoras domésticas y comerciales. Introducido en 1985, el sistema operativo basado en GUI, se ha lanzado en muchas versiones desde entonces. Windows 95 fue en gran parte responsable del rápido desarrollo de la informática personal [8].

Mac OS es el sistema operativo para la línea de ordenadores personales y estaciones de trabajo Macintosh de Apple. Linux es un sistema operativo similar a Unix que fue diseñado para proporcionar a los usuarios de computadoras personales una alternativa gratuita o de muy bajo costo. Linux tiene una reputación como un sistema muy eficiente y de rápido rendimiento [8].

Los sistemas operativos de Windows han dominado durante mucho tiempo el mercado y continúan haciéndolo. A partir de agosto de 2016, los sistemas de Windows tenían una cuota de mercado de más del 85%. En contraste, Mac OS estaba en poco más del 6% y Linux era un poco más del 2% [8].

Un sistema operativo móvil permite que teléfonos inteligentes, tabletas y otros dispositivos móviles ejecuten aplicaciones y programas. Los sistemas operativos móviles incluyen Apple iOS, Google Android, BlackBerry OS y Windows 10 Mobile [8].

Un sistema operativo embebido está especializado para su uso en sistemas como automóviles, semáforos, televisores digitales, cajeros automáticos, controles de avión, terminales de punto de venta, cámaras digitales, sistemas de navegación, ascensores, medios digitales receptores y medidores inteligentes [8].

2.2 Sistemas Operativos Embebidos

Un sistema operativo embebido es específico para sistemas informáticos embebidos. Este tipo de sistema operativo generalmente está diseñado para ser eficiente en cuanto a recursos y es muy confiable. La eficiencia de los recursos tiene un precio alto, como el de perder funcionalidades que proporcionan los sistemas operativos de computadora más grandes, incluidas las funciones que no pueden ser utilizadas por las aplicaciones embebidas. Según el método utilizado para la multitarea, este tipo de sistema operativo se considera a menudo en tiempo real [9].

El hardware que ejecuta un sistema operativo embebido puede ser muy limitado en recursos como RAM – Memoria de Acceso Aleatorio (del inglés, *Random Access Memory*) y ROM – Memoria de Solo Lectura (del inglés, *Read Only Memory*). Por lo tanto, el diseño integrado de estos sistemas operativos puede tener un alcance limitado adaptado a una aplicación específica para lograr la operación deseada bajo estas restricciones. Con el fin de aprovechar mejor la capacidad de procesamiento del procesador, los desarrolladores de software pueden escribir código crítico directamente en el ensamblaje. Este lenguaje de máquina eficiente puede generar ganancias de velocidad y determinismo. De esta manera, en su mayoría los sistemas integrados están escritos enteramente en más idiomas portátiles, como el lenguaje C [9].

Una diferencia importante entre la mayoría de los sistemas operativos embebidos y los sistemas operativos de escritorio es que la aplicación, incluido el sistema operativo, generalmente está unida estáticamente en una sola imagen ejecutable. A diferencia de un sistema operativo de escritorio, el sistema operativo integrado no carga y ejecuta aplicaciones. Esto significa que el sistema puede ejecutar una sola aplicación [9].

2.2.1 Propósito de los sistemas operativos embebidos

- Proceso rápido y ligero.
- La política de programación es en tiempo real y el módulo despachador es parte del programador.
- Tamaño pequeño.
- Responde a interrupciones del exterior con una alta velocidad.
- Minimiza los intervalos durante los cuales las interrupciones están deshabilitadas.
- Proporciona segmentos de tamaño fijo o variable para la gestión de la memoria.

El sistema operativo implementado en este trabajo es un sistema operativo de tiempo real [10].

2.3 Sistemas operativos en tiempo real

La computación en tiempo real se refiere a que la corrección del sistema no solo depende de la corrección del resultado lógico, sino también del tiempo de entrega del resultado. Por lo tanto, el sistema operativo debe tener características que respalden este requisito crítico para convertirlo en un RTOS - Sistema Operativo en Tiempo Real (del inglés, *Real Time Operating System*). El RTOS debe tener un comportamiento predecible ante eventos externos impredecibles. P. M. Sagar argumenta que "un buen RTOS es aquel que tiene un comportamiento limitado (predecible) en todo escenario de carga del sistema, es decir, incluso en interrupciones simultáneas y ejecución de subprocesos" [4]. Un verdadero RTOS será determinista en todas las condiciones. Estos sistemas operativos ocupan poco espacio de 10 KB a 100 KB, en comparación con los sistemas operativos generales que requieren cientos de megabytes [10].

2.3.1 RTOS críticos

El sistema en tiempo real crítico es puramente determinista y el tiempo de respuesta debe ser exacto al calculado por el usuario. Por ejemplo, los usuarios esperan la salida para la entrada dada en 10 segundos, entonces el sistema debe procesar los datos de entrada y dar la salida exactamente en 10 segundos para completar el proceso de los datos entregados. No debe dar la salida en 11 segundos o en 9 segundos, sino exactamente en el décimo segundo. Es por ello que el sistema de tiempo real critico debe cumplir la fecha límite, ya que de no ser así, ocurrirían fallas en el sistema, ocasionando posibles daños a los datos o perdidas completas de los mismos [11].

2.3.2 RTOS no críticos

En el sistema no critico en tiempo real, el cumplimiento del plazo no es obligatorio para las tareas en todo momento, pero el proceso debe procesarse y dar el resultado. Incluso los sistemas no críticos en tiempo real no pueden perder la fecha límite para cada tarea o proceso de acuerdo con la prioridad que debe cumplir el plazo o de lo contrario, pueden perder la fecha límite. Si el sistema falla un determinado número de veces, el rendimiento del sistema empeorará y los usuarios no podrán utilizarlo. El mejor ejemplo para un sistema en tiempo real no critico es una computadora personal, así como sistemas de audio y video [11]. El elemento más importante de cualquier RTOS es el scheduler, el cual determina que tarea es la siguiente a ejecutarse mediante las prioridades del sistema.

2.4 Scheduler

La función del scheduler es la actividad del administrador de procesos que maneja la eliminación del proceso en ejecución del CPU y la selección de otro proceso sobre la base de una estrategia en particular. El scheduler es una parte esencial de los sistemas operativos multiprogramación. Dichos sistemas operativos permiten que se cargue más de un proceso en la memoria ejecutable a la vez y el proceso cargado comparte el CPU usando el multiplexado de tiempo [12].

El sistema operativo mantiene una cola separada para cada uno de los estados del proceso y las tareas de todos los procesos en el mismo estado de ejecución se colocan en la misma cola. Cuando se cambia el estado de un proceso, su tarea se desvincula de su cola actual y se mueve a su nueva cola de estado. El sistema operativo mantiene la siguiente cola actualizando su estado del proceso en el scheduler:

- Cola de trabajos: mantiene todos los procesos en el sistema.
- Cola lista: mantiene un conjunto de todos los procesos que residen en la memoria principal y que se encuentran listos y esperando a ser ejecutados. Un nuevo proceso siempre se pone en esta fila.
- Colas de dispositivo: los procesos que están bloqueados debido a la falta de disponibilidad de un dispositivo de entradas y salidas constituyen esta cola.

El sistema operativo puede usar diferentes políticas para administrar cada cola (FIFO, Prioridad, etc.). El scheduler del sistema operativo determina cómo mover los procesos entre las colas de espera y ejecución, ya que solo pueden tener una entrada por núcleo del procesador en el sistema. El scheduler puede tener varios estados durante un ciclo de vida de la tarea, mismos que se clasifican como ejecutándose, no ejecutándose y bloqueado. Los schedulers son un software de sistema especial que maneja la programación de procesos de varias maneras. Su tarea principal es seleccionar los trabajos que se enviarán al sistema y decidir qué proceso ejecutar. Por otro lado, los programadores son de tres tipos denominados scheduler a largo plazo, corto plazo y mediano plazo [12].

2.4.1 Scheduler a largo plazo

Un planificador a largo plazo determina qué programas son admitidos en el sistema para su procesamiento. Selecciona los procesos de la cola y los carga en la memoria para su ejecución y procesa cargas en la memoria para la programación del CPU.

El objetivo principal del scheduler a largo plazo es proporcionar una combinación equilibrada de trabajos, como el límite de entradas y salidas y el límite del procesador así como también controla el grado de multiprogramación. Si el grado de multiprogramación es estable como se ilustra en la Figura 2-1, donde se muestran los diferentes estados actuales que puede manejar un scheduler Ready (tarea lista para ejecución), running (tarea corriendo), terminated (tarea terminada), new (tarea nueva), interrupt (tarea interrumpida) y waiting (tarea en espera). La tasa promedio de creación del proceso debe ser igual a la tasa promedio de salida de los procesos que abandonan el sistema.

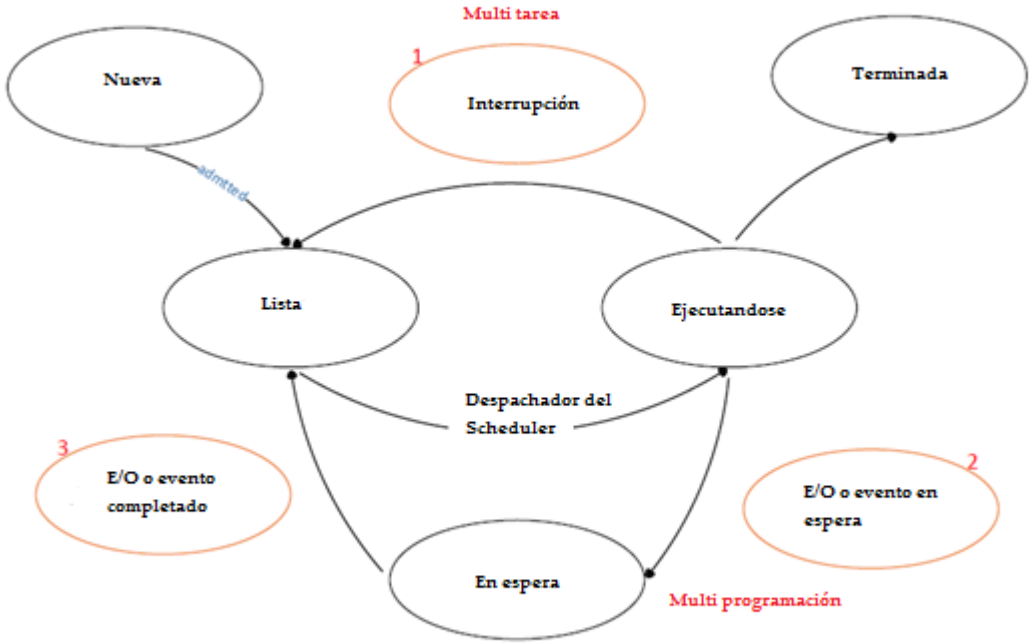


Figura 1-1 Diagrama de estados de un scheduler multitarea.

En algunos sistemas, el scheduler a largo plazo puede no estar disponible en comparación con los sistemas operativos de tiempo compartido, ya que en ellos no se tiene un programador a

largo plazo. No obstante, cuando un proceso cambia el estado de *new* a *ready,* se utiliza el planificador a largo plazo [12].

2.4.2 Scheduler de corto plazo

Su objetivo principal es aumentar el rendimiento del sistema de acuerdo con el conjunto de criterios elegido. El scheduler de corto plazo se refiere al cambio del estado listo para usarse al estado de ejecución en proceso. El programador del CPU selecciona un proceso entre los procesos que están listos para ejecutarse y asigna el CPU a uno de ellos.

Los scheduler a corto plazo, también conocidos como despachadores, toman la decisión de qué proceso ejecutar a continuación. Cabe señalar que los programadores a corto plazo son más rápidos que los programadores a largo plazo [12].

2.4.3 Scheduler de mediano plazo

El scheduler a mediano plazo es una parte del intercambio, elimina los procesos de la memoria y reduce el grado de multiprogramación. El scheduler de mediano plazo es el encargado de manejar los procesos externos intercambiados.

Un proceso en ejecución puede suspenderse si realiza una solicitud de entrada o salida. A los procesos suspendidos no se le puede dar un avance para su finalización. En esta condición, para eliminar el proceso de la memoria y dejar espacio para otros procesos, el proceso suspendido se traslada al almacenamiento secundario. A este proceso se llama *wapping* y se dice que el proceso se intercambia o se implementa. El intercambio puede ser necesario para mejorar la mezcla del proceso, para ello, los scheduler se dividen en *pre-em*pt, *no pre-empt* y *cooperative* [12].

2.4.4 No Pre-empt

Un no pre-empt es una disposición de elegir todos los elementos en un grupo en un orden racional, generalmente desde la parte superior a la parte inferior de una lista y luego comenzar de nuevo en la parte superior de la lista, y así sucesivamente. Una forma simple de describir un no pre-empt es que se trata de tomar turnos.

En el funcionamiento de la computadora, un método para tener diferentes procesos de programa por turnos usando los recursos de la computadora, es limitar cada proceso a un cierto período de tiempo corto, luego suspender ese proceso para dar otro giro a un proceso (o "intervalo de tiempo"). Esto a menudo se describe como la programación de procesos no pre-empt. Por ejemplo, en los torneos deportivos y en otros juegos, la programación por turnos permite que todos los equipos o jugadores se turnen para jugarse entre sí, y el ganador sale de la sucesión de eventos [13].

2.4.5 Pre-empt

En términos simples, pre-empt implica el uso de un mecanismo de interrupción que suspende el proceso que se está ejecutando actualmente e invoca un programador para determinar qué proceso debe ejecutarse a continuación. Por lo tanto, todos los procesos obtendrán cierta cantidad de tiempo del CPU en un momento dado.

En el pre-empt, el kernel del sistema operativo también puede iniciar un cambio de contexto para satisfacer la restricción de prioridad de la política de programación. Cuando la tarea de alta prioridad en esa instancia se apodera de la tarea que se está ejecutando, actualmente se conoce como programación pre-empt. El término pre-empt a veces se utiliza de manera errónea cuando el significado previsto es más específico, refiriéndose en cambio a la clase de políticas de programación conocida como programación de tiempo compartido o tiempo compartido.

El pre-empt permite que el sistema garantice de manera confiable cada proceso como un segmento regular de tiempo de operación. También le permite al sistema lidiar rápidamente con eventos externos importantes, como los datos entrantes, que pueden requerir la atención inmediata de uno u otro proceso.

En cualquier momento específico, los procesos se pueden agrupar en dos categorías: los que están esperando entrada o salida, y los que están utilizando completamente la CPU. En los primeros sistemas, los procesos solían esperar un tiempo considerable mientras esperaban la entrada solicitada. Durante este tiempo, el proceso no estaba realizando un trabajo útil, pero aún mantenía un control completo del CPU. Con el uso de las interrupciones y el pre-empt, estos procesos de entrada y salida pueden ser bloqueados, o puestos en espera a la llegada de los datos necesarios, permitiendo que otros procesos utilicen el CPU. Como la llegada de los datos solicitados generaría una interrupción, los procesos bloqueados podrían garantizar un retorno oportuno a la ejecución.

Aunque las técnicas del pre-empt se desarrollaron originalmente para permitir que varios usuarios compartan una sola máquina, pronto se hizo evidente que la multitarea era útil independientemente de la cantidad de usuarios. Muchos sistemas operativos han reconocido la utilidad del soporte multitarea por una variedad de razones. El pre-empt hace posible que un solo usuario ejecute varias aplicaciones al mismo tiempo o que ejecute procesos de "fondo" sin perder el control de la computadora.

2.4.6 Cooperative

Cooperative es un estilo de multitarea en la que el sistema operativo nunca inicia un cambio de contexto de un proceso en ejecución a otro. En cambio, los procesos rinden control de forma voluntaria periódicamente o cuando están inactivos para permitir que se ejecuten varias aplicaciones al mismo tiempo. Este tipo de multitarea se llama cooperative porque todos los programas deben cooperar para que funcione todo el esquema de programación. En este esquema,

el scheduler de un sistema operativo se conoce como scheduler cooperative, teniendo su rol reducido en comenzar los procesos y permitiéndoles regresar el control de forma voluntaria.

Un sistema cooperative depende de que cada proceso ceda regularmente su tiempo a otros procesos en el sistema. Por otro lado, un programa mal diseñado puede consumir todo el tiempo del CPU por sí mismo, ya sea al realizar cálculos extensos o por la espera ocupada, ya que ambos causarían que todo el sistema se cuelgue. En un entorno de servidor, este es un peligro que hace que todo el entorno sea inaceptablemente frágil. Sin embargo, la multitarea cooperative permite una implementación mucho más simple de las aplicaciones porque el planificador del proceso nunca interrumpe inesperadamente su ejecución.

A continuación se muestra en la Figura 2-2, la comparativa entre ambos mecanismos que permite un mejor análisis para determinar el sistema más adecuado para la aplicación a desarrollar; por ejemplo, varias funciones dentro de la aplicación no necesitan ser reentrantes. El scheduler a implementar se realizará en el hardware de una tarjeta de desarrollo de Atmel.

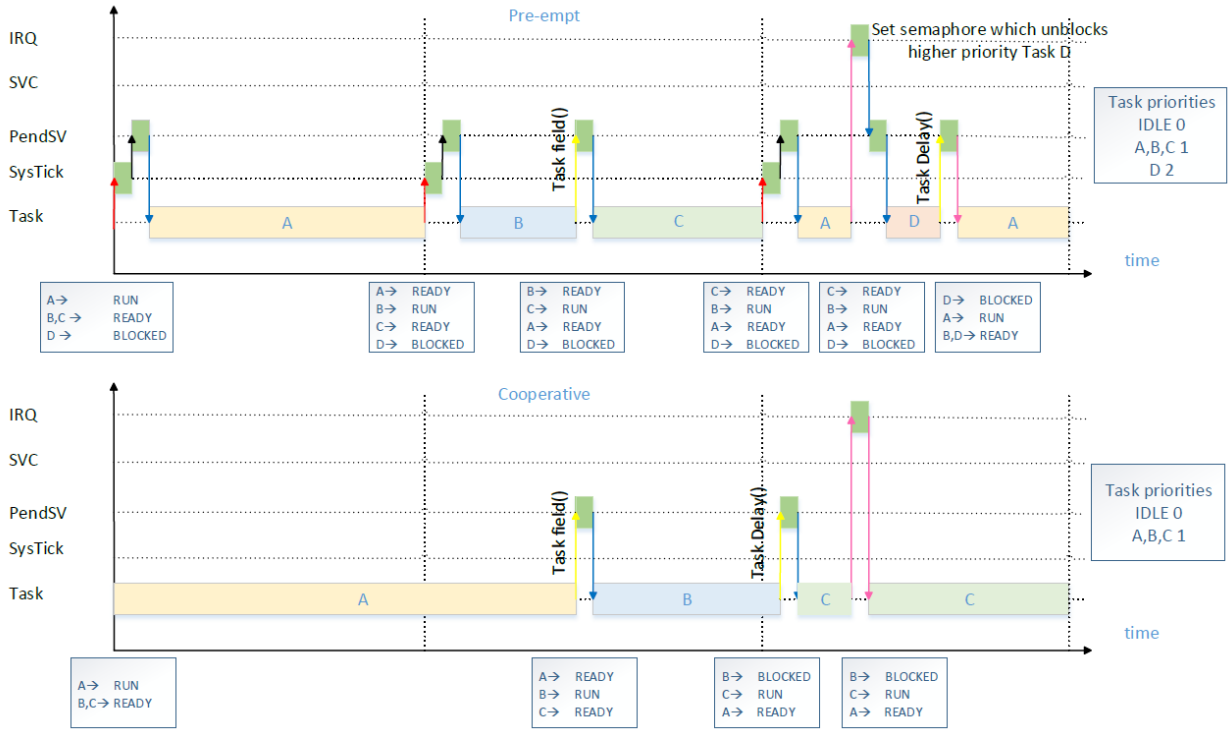


Figura 2-3 Programación Pre-empt y Cooperative.

2.5 Atmel

Atmel Corporation es un diseñador y fabricante estadounidense de semiconductores, fundado en 1984. La compañía se enfoca en sistemas embebidos construidos alrededor de microcontroladores. Sus productos incluyen microcontroladores (AVR de 8 bits, AVR de 32 bits, ARM de 32 bits, grado automotriz y derivados Intel 8051 de 8 bits), así como dispositivos de radiofrecuencia que incluyen Wi-Fi y dispositivos de memoria flash, chips de seguridad simétricos y asimétricos, sensores táctiles y controladores, y productos específicos de la aplicación. Atmel suministra sus dispositivos como productos estándar, circuitos integrados específicos de aplicaciones o productos específicos de aplicaciones según los requisitos de sus clientes.

Atmel presta servicios en aplicaciones de consumo, comunicaciones, redes informáticas, industrial, médica, automotriz, aeroespacial y militar. Se especializa en microcontroladores y sistemas táctiles, especialmente para sistemas integrados. La sede corporativa de Atmel se

encuentra en San José, California. Otros lugares incluyen Trondheim, Noruega; Colorado Springs, Colorado; Chennai, India; Shanghai, China; Taipei, Taiwán; Rousset, Francia; Nantes, Francia; Patras, Grecia; Heilbronn, Alemania; Munich, Alemania; Whiteley, Reino Unido y El Cairo, Egipto. Atmel hace gran parte de su línea de productos en las instalaciones de fabricación de proveedores [13].

2.5.1 V71 SAM kit de desarrollo

El kit de evaluación Xplained Ultra SMART SAM V71 es ideal para evaluar y crear prototipos con los microcontroladores Atmel SAM V71, SAM V70, SAM S70 y SAM E70 ARM® Cortex®-M7 [13]. Sus características principales son las siguientes:

- Microcontrolador ATSAMV71Q21
- Un botón de reset
- Un botón de encendido
- Dos botones mecánicos para uso general
- Dos LEDS amarillos para uso en general
- Supercap de respaldo
- 12.0 MHz cristal
- 32.768 kHz cristal
- 2 MB SDRAM
- 2 MB QSPI Flash
- IEEE 802.3az 10Base-T/100Base-TX Ethernet RMII PHY
- AT24MAC402 256KByte EEPROM con EUI-48
- Stereo audio codec
- ATA6561 CAN Transceiver
- Conector SD
- Interface para camara
- Conector MediaLB
- Two Xplained Pro extension headers

- Coresight 20
- Arduino compatible
- Conector externo para el debugger
- USB interface
- Debugger embebido
- Fuente de poder externa (5-14V)
- USB de encendido

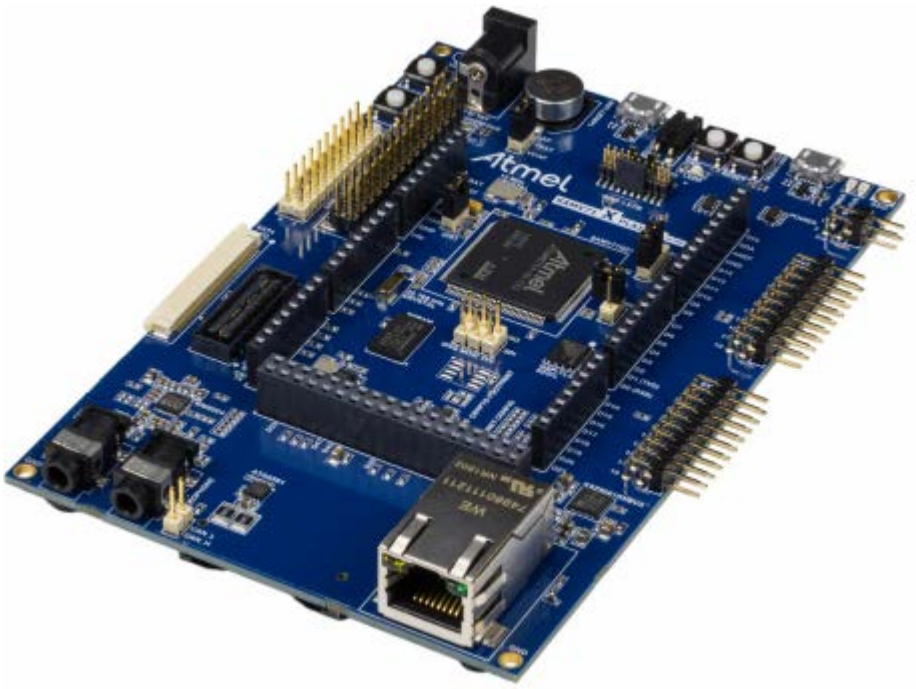


Figura 4-3. SAM V71 Xplained Ultra Evaluation Kit.

3. Metodología

El diseño del scheduler propuesto utiliza un temporizador del sistema siendo el que lleva los tiempos de cada tarea. Es muy importante configurarlo correctamente ya que es el corazón del scheduler. A continuación, se detalla su configuración:

El propósito del temporizador del sistema, SysTick, proporciona un contador simple de 24 bits con escritura libre, decremento y un mecanismo de control flexible. El scheduler para nuestro proyecto usa el SysTick para proporcionar una base de tiempo de 500us. Cada vez que se activa la interrupción, se agrega un contador de 8 bits. Existe función de configuración que utiliza este contador para proporcionar una forma precisa para que la aplicación se active o se suspenda por un período de tiempo específico.

La inicialización se realiza mediante el siguiente código y fórmula como se muestra en la Figura 3-1:

SysTick_Config(SystemCoreClock/TASK_SCHEDULER_BASE_FREQ);

$$SycTick = \frac{SystemcoreClock}{TASK\_SCHEDULER\_BASE\_FREQ}$$

Figura 3-1. Fórmula para el cálculo del tiempo base.

Donde el SystemCoreClock es igual a 4MHz y el TASK_SCHEDULER_BASE_FREQ se definió como 2000, haciendo la división, nos quedan 2MHz y al realizar el inverso nos da el resultado de 500us, lo cual significa que cada 500us estaremos generando una interrupción para incrementar nuestro contador y poder revisar que tarea es la siguiente en ejecutarse.

3.1 Diseño del contador para el scheduler

Para el diseño del contador se realizó el siguiente esquema que se muestra en la Figura 3-2:

Pauta del tiempo	gu8Scheduler_Thread_ID	gu8Scheduler_Counter (binary)	vfnScheduler_Callback	1ms Periodic Tasks	2ms A Periodic Tasks	2ms B Periodic Tasks	10ms Periodic Tasks	50ms Periodic Tasks	100ms Periodic Tasks
0	0	00000000							
0.5	1	00000001	x	x					
1	2	00000010	x		x				
1.5	3	00000011	x	x					
2	4	00000100	x			x			
2.5	5	00000101	x	x					
3	6	00000110	x		x				
3.5	7	00000111	x	x					
4	8	00001000	x			x			
4.5	9	00001001	x	x					
5	10	00001010	x		x				
5.5	11	00001011	x	x					
6	12	00001100	x			x			
6.5	13	00001101	x	x					
7	14	00001110	x		x				
7.5	15	00001111	x	x					
8	16	00010000	x			x			
8.5	17	00010001	x	x					
9	18	00010010	x		x				
9.5	19	00010011	x	x					
10	20	00010100	x			x	x		
10.5	21	00010101	x	x					
11	22	00010110	x		x				
11.5	23	00010111	x	x					
12	24	00011000	x			x			
12.5	25	00011001	x	x					
13	26	00011010	x		x				

13.5	27	00011011	x	x					
14	28	00011100	x			x			
14.5	29	00011101	x	x					
15	30	00011110	x		x				
15.5	31	00011111	x	x					
16	32	00100000	x			x			
16.5	33	00100001	x	x					
17	34	00100010	x		x				
17.5	35	00100011	x	x					
18	36	00100100	x			x			
18.5	37	00100101	x	x					
19	38	00100110	x		x				
19.5	39	00100111	x	x					
20	40	00101000	x			x	x		
20.5	41	00101001	x	x					
21	42	00101010	x		x				
21.5	43	00101011	x	x					
22	44	00101100	x			x			
22.5	45	00101101	x	x					
23	46	00101110	x		x				
23.5	47	00101111	x	x					
24	48	00110000	x			x			
24.5	49	00110001	x	x					
25	50	00110010	x		x				
25.5	51	00110011	x	x					
26	52	00110100	x			x			
26.5	53	00110101	x	x					
27	54	00110110	x		x				
27.5	55	00110111	x	x					
28	56	00111000	x			x			
28.5	57	00111001	x	x					
29	58	00111010	x		x				
29.5	59	00111011	x	x					
30	60	00111100	x			x	x		
30.5	61	00111101	x	x					
31	62	00111110	x		x				
31.5	63	00111111	x	x					
32	64	01000000	x			x			
32.5	65	01000001	x	x					
33	66	01000010	x		x				
33.5	67	01000011	x	x					
34	68	01000100	x			x			
34.5	69	01000101	x	x					
35	70	01000110	x		x				
35.5	71	01000111	x	x					
36	72	01001000	x			x			
36.5	73	01001001	x	x					

37	74	01001010	x		x				
37.5	75	01001011	x	x					
38	76	01001100	x			x			
38.5	77	01001101	x	x					
39	78	01001110	x		x				
39.5	79	01001111	x	x					
40	80	01010000	x			x	x		
40.5	81	01010001	x	x					
41	82	01010010	x		x				
41.5	83	01010011	x	x					
42	84	01010100	x			x			
42.5	85	01010101	x	x					
43	86	01010110	x		x				
43.5	87	01010111	x	x					
44	88	01011000	x			x			
44.5	89	01011001	x	x					
45	90	01011010	x		x				
45.5	91	01011011	x	x					
46	92	01011100	x			x			
46.5	93	01011101	x	x					
47	94	01011110	x		x				
47.5	95	01011111	x	x					
48	96	01100000	x			x			
48.5	97	01100001	x	x					
49	98	01100010	x		x			x	
49.5	99	01100011	x	x					
50	100	01100100	x			x	x		
50.5	101	01100101	x	x					
51	102	01100110	x		x				
51.5	103	01100111	x	x					
52	104	01101000	x			x			
52.5	105	01101001	x	x					
53	106	01101010	x		x				
53.5	107	01101011	x	x					
54	108	01101100	x			x			
54.5	109	01101101	x	x					
55	110	01101110	x		x				
55.5	111	01101111	x	x					
56	112	01110000	x			x			
56.5	113	01110001	x	x					
57	114	01110010	x		x				
57.5	115	01110011	x	x					
58	116	01110100	x			x			
58.5	117	01110101	x	x					
59	118	01110110	x		x				
59.5	119	01110111	x	x					
60	120	01111000	x			x	x		

60.5	121	01111001	x	x					
61	122	01111010	x		x				
61.5	123	01111011	x	x					
62	124	01111100	x			x			
62.5	125	01111101	x	x					
63	126	01111110	x		x				
63.5	127	01111111	x	x					
64	128	10000000	x			x			
64.5	129	10000001	x	x					
65	130	10000010	x		x				
65.5	131	10000011	x	x					
66	132	10000100	x			x			
66.5	133	10000101	x	x					
67	134	10000110	x		x				
67.5	135	10000111	x	x					
68	136	10001000	x			x			
68.5	137	10001001	x	x					
69	138	10001010	x		x				
69.5	139	10001011	x	x					
70	140	10001100	x			x	x		
70.5	141	10001101	x	x					
71	142	10001110	x		x				
71.5	143	10001111	x	x					
72	144	10010000	x			x			
72.5	145	10010001	x	x					
73	146	10010010	x		x				
73.5	147	10010011	x	x					
74	148	10010100	x			x			
74.5	149	10010101	x	x					
75	150	10010110	x		x				
75.5	151	10010111	x	x					
76	152	10011000	x			x			
76.5	153	10011001	x	x					
77	154	10011010	x		x				
77.5	155	10011011	x	x					
78	156	10011100	x			x			
78.5	157	10011101	x	x					
79	158	10011110	x		x				
79.5	159	10011111	x	x					
80	160	10100000	x			x	x		
80.5	161	10100001	x	x					
81	162	10100010	x		x				
81.5	163	10100011	x	x					
82	164	10100100	x			x			
82.5	165	10100101	x	x					
83	166	10100110	x		x				
83.5	167	10100111	x	x					

84	168	10101000	x			x			
84.5	169	10101001	x	x					
85	170	10101010	x		x				
85.5	171	10101011	x	x					
86	172	10101100	x			x			
86.5	173	10101101	x	x					
87	174	10101110	x		x				
87.5	175	10101111	x	x					
88	176	10110000	x			x			
88.5	177	10110001	x	x					
89	178	10110010	x		x				
89.5	179	10110011	x	x					
90	180	10110100	x			x	x		
90.5	181	10110101	x	x					
91	182	10110110	x		x				
91.5	183	10110111	x	x					
92	184	10111000	x			x			
92.5	185	10111001	x	x					
93	186	10111010	x		x				
93.5	187	10111011	x	x					
94	188	10111100	x			x			
94.5	189	10111101	x	x					
95	190	10111110	x		x				
95.5	191	10111111	x	x					
96	192	11000000	x			x			
96.5	193	11000001	x	x					
97	194	11000010	x		x				
97.5	195	11000011	x	x					
98	196	11000100	x			x			
98.5	197	11000101	x	x					
99	198	11000110	x		x			x	
99.5	199	11000111	x	x					x
100	200	11001000	x			x	x		

Figura 3-2. Lógica del scheduler.

El máximo conteo que alcanzará el contador será de 100ms siendo la tarea más lenta que se tiene en el scheduler. Una propiedad interesante de los números binarios, es que cada vez que se van incrementando podemos observar como ciertas secuencias son repetidas. Por ejemplo, el bit menos significativo siempre es 0 o 1 y se repite igual hasta los 100 ms, el segundo bit menos significativo siempre es 0, 0, 1 y 1 y sucesivamente en los demás bits podemos encontrar un patrón

de repetición. Este patrón de repetición es que nos permite el poder agendar las tareas y no colisionar en su tiempo de ejecución en cada una de ellas.

3.2 Especificación funcional

El scheduler proporciona 4 subprocesos de ejecución de la siguiente manera:

- Tarea de 1ms
- Tarea de 10ms
- Tarea de 50ms
- Tarea de 100ms

Para equilibrar la carga de trabajo del CPU, el scheduler implementa dos subprocesos de ejecución independientes de la siguiente manera:

- Tarea de 2ms A
- Tarea de 2ms B

Tales subprocesos comparten el tiempo de ejecución con los subprocesos de ejecución definidos previamente en la lógica del scherduler como se muestra en la Tabla 3-1:

Tabla 3-1.Tareas Primarias y Dependientes.

Tarea Primaria	Tarea Dependiente
Tarea de 1ms	Tarea de 100ms
Tarea de 2ms A	Tarea de 50ms
Tarea de 2ms B	Tarea de 10ms

3.3 Diseño de las funciones

El diseño de las funciones a implementar está basado en los requerimientos como se describirán a continuación en cada una de las tablas. En primera instancia, se deben de implementar las funciones que permiten la inicialización y configuración del scheduler como se menciona enseguida:

3.3.1 vfnScheduler_Init

La función de inicializar se encarga de poner en un estado inicial a todas las variables que se estarán comparando para saber que tarea es la siguiente en ejecutarse, los parámetros que se necesitan se muestran en la Tabla 3-2.

Tabla 3-2. Función de inicialización del scheduler.

Service Name	vfnScheduler_Init
Syntax	void vfnScheduler_Init(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Inicializa el módulo del scheduler

3.3.2 vfnScheduler_Start

La función de inicio es la encargada de configurar el SysTick a una frecuencia de 2 MHz para que cada 500 us ocurra la interrupción y se esté incrementando los contadores de las tareas para así saber cuál es la siguiente en ejecutar, los parámetros necesarios se muestran en la Tabla 3-3.

Tabla 3-3. Función de arranque del scheduler.

Service Name	vfnScheduler_Start

Syntax	void vfnScheduler_Start(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Inicializa el scheduler

3.3.3 vfnScheduler_Stop

La función de parada es la encargada de detener el shceduler, dejando su estado en inhabilitado. En la Tabla 3-4 se muestran los parámetros requeridos para su llamada.

Tabla 3-4. Función de paro del scheduler.

Service Name	vfnScheduler_Stop
Syntax	void vfnScheduler_Stop(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Actualiza el estatus del scheduler

3.3.4 vfnScheduler_TaskStart

La función de inicio de tarea se encarga de poner la tarea que le llega como parámetro en un estado de correr, y a su vez ejecuta la tarea una vez finalizada se pone el estatus de la tarea en suspendida, los parámetros se muestran en la Tabla 3-5.

Tabla 3-5. Función de inicio de la tarea.

Service Name	vfnScheduler_TaskStart
Syntax	void vfnScheduler_TaskStart(tSchedulingTask * Task)
Param(in)	Tarea – puntero a tarea
Param(out)	None
Return value	None
Description	Actualiza el estatus del scheduler a Start

3.3.5 vfnScheduler_TaskActivate

La siguiente función se encarga de poner a la tarea en un estado de listo, esto se realiza para saber que la tarea esta lista para ser ejecutada y poder pasar al estado de ejecución. Los parámetros necesarios para la función se muestran en la Tabla 3-6.

Tabla 3-6. Función del estatus de la tarea.

Service Name	Sch vfnScheduler_TaskActivate eduler Task Activate
Syntax	Void vfnScheduler_TaskActivate(tSchedulingTask * Task)
Param(in)	Tarea – puntero a tarea
Param(out)	None
Return value	None
Description	Actualiza el estatus del scheduler a Activete

3.3.6 vfnTask_Scheduler

La función de scheduler de la tarea se encarga de revisar si el estado del contador es igual a las diferentes tareas que tenemos, si es igual llama a la función de activar tarea. Los parámetros requeridos para la función se muestran en la Tabla 3-7.

Tabla 3-7. Función de programación de tareas.

Service Name	vfnTask_Scheduler
Syntax	void vfnTask_Scheduler(void)
Param(in)	None
Param(out)	None
Return value	None
Description	La tarea del Scheduler invoca las diferentes tareas de tiempo base

3.3.7 vfnScheduler_Callback

La función de retro llamada es la encargada de incrementar y de reiniciar los contadores para todas las tareas, cada 500 us es llamada esta función mediante la interrupción del SysTick. Los parámetros de la función de muestran en la Tabla 3-8.

Tabla 3-8. Función de la retro llamada del SysTick.

Service Name	vfnScheduler_Callback
Syntax	void vfnScheduler_Callback(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Interrupción periódica que será interrumpida por alguna tarea base, la tarea del scheduler se ejecutará cada 500uS

Las tareas de tiempo permiten tener el control de la ejecución de eventos internos o externos del microcontrolador, configurando banderas de interrupción en los periféricos de salida o entrada. El tiempo de ejecución de las tareas depende de la administración del scheduler y del desarrollador, al decidir la prioridad de cada tarea para el cumplimiento de los eventos. A continuación, se describe el diseño de los requerimientos de cada una de las tareas de tiempo.

3.3.8 TASKS_LIST_1MS

La tarea de 1 ms es llamada cada 2 SysTick del sistema, la Tabla 3-9 muestra el prototipo de la función.

Tabla 3-9. Función de la tarea de 1 ms.

Service Name	TASKS_LIST_1MS
Syntax	void TASKS_LIST_1MS(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Tarea de 1ms

3.3.9 TASKS_LIST_2MS_A

La tarea de 2 ms A es llamada cada 4 SysTick del sistema (Tabla 3-10).

Tabla 3-10. Función de la tarea de 2A ms.

Service Name	TASKS_LIST_2MS_A
Syntax	void TASKS_LIST_2MS_A(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Tarea de 2ms_B

3.3.10 TASKS_LIST_2MS_B

La tarea de 2 ms B es llamada cada 4 SysTick del sistema, la Tabla 3-11 indica el prototipo de la función.

Tabla 3-11. Función de la tarea de 2B ms.

Service Name	TASKS_LIST_2MS_B
Syntax	void TASKS_LIST_2MS_B(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Tarea de 2ms_B

3.3.11 TASKS_LIST_10MS

La tarea de 10 ms es llamada cada 20 SysTick del sistema (Tabla 3-12).

Tabla 3-12. Función de la tarea de 10 ms.

Service Name	TASKS_LIST_10MS
Syntax	void TASKS_LIST_10MS(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Tarea de 10ms

3.3.12 TASKS_LIST_50MS

La tarea de 50 ms es llamada cada 100 SysTick del sistema (Tabla 3-13).

Tabla 3-13. Función de la tarea de 50 ms.

Service Name	TASKS_LIST_50MS
Syntax	void TASKS_LIST_50MS(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Tarea de 50ms

3.3.13 TASKS_LIST_100MS

La tarea de 100 ms es llamada cada 200 SysTick del sistema (Tabla 3-14).

Tabla 3-14.Función de la tarea de 100 ms.

Service Name	TASKS_LIST_100MS
Syntax	void TASKS_LIST_100MS(void)
Param(in)	None
Param(out)	None
Return value	None
Description	Tarea de 100ms

3.4 Implementación de las funciones

La declaración de las funciones se realizó en el App_scheduler.h y App_tasks.h, mientras que la implementación se llevó a cabo en App_sheduler.c y en App_tasks.c como se muestran a continuación.

3.4.1 Declaración e implementación del scheduler

La declaración de los módulos y cada una de las funciones del scheduler se muestran en la Figura 3-3.

```
/*****************************************************************************************************
* Declaration of module wide FUNCTIONS
*****************************************************************************************************/

/** Scheduler Initalization (arming) */
void vfnScheduler_Init(void);

/** Scheduler kick-off function */
void vfnScheduler_Start(void);

/** Scheduler stop function */
void vfnScheduler_Stop(void);

/** Multi-thread round robin task scheduler */
void vfnTask_Scheduler(void);

/*****************************************************************************/
```

Figura 3-3. Prototipos de las funciones del scheduler.

La declaración de la función de retro para el scheduler se muestra en la Figura 3-4.

```
/*****************************************************************************************************
* Code of module wide private FUNCTIONS
*****************************************************************************************************/
void vfnScheduler_Callback(void);
```

Figura 3-4. Prototipo de la retro llamada del SysTick.

La implementación de la función de inicialización del scheduler se muestra en la Figura 3-5.

```
void vfnScheduler_Init(void)
{
    /* Init Global and local Task Scheduler variables */
    gu8Scheduler_Counter    = 0u;
    TaskScheduler_Task_ID_Activated = TASK_NULL;
    TaskScheduler_Task_ID_Running = TASK_NULL;
    TasksScheduler_Task_ID_Backup = TASK_NULL;
    u8_10ms_Counter         = 0u;
    u8_50ms_Counter         = 0u;
    u8_100ms_Counter        = 0u;
    gu8Scheduler_Status     = TASK_SCHEDULER_INIT;
}
```

Figura 3-5 Implementación de la inicialización del scheduler.

La implementación de la función que comienza la ejecución del scheduler a la frecuencia de 2 MHz, se muestra en la Figura 3-6.

```
void vfnScheduler_Start(void)
{
  if (sysTick_init(TASK_SCHEDULER_BASE_FREQ, vfnScheduler_Callback))
  {
    while (1);
  }
    gu8Scheduler_Status = TASK_SCHEDULER_RUNNING;
}
```

Figura 3-6 Implementación del arranque del scheduler.

La implementación de la función detendrá la tarea que se encuentra en ejecución (Figura 3-7).

```
void vfnScheduler_Stop(void)
{
    /* Update scheduler status accordingly */
    gu8Scheduler_Status = TASK_SCHEDULER_HALTED;
}
```

Figura 3-7 Implementación de la parada del scheduler.

La implementación de la función que proporciona la ejecución de las tareas del scheduler que se configuraron cambiando el estatus en correr, se muestra en la Figura 3-8:

```
void vfnScheduler_TaskStart( tSchedulingTask * Task )
{
    /* Indicate that this Task has gained CPU allocation */
    Task->enTaskState = RUNNING;
    TaskScheduler_Task_ID_Running =  Task->TaskId;
    /* Perform actual execution of task */
    Task->ptrTask();
    /* Indicate that Task execution has completed */
    Task->enTaskState = SUSPENDED;
}
```

Figura 3-8 Implementación del inicio de la tarea.

La implementación de la función que activa a las tareas del scheduler actualizando el estado en listo, se muestra en la Figura 3-9.

```
void vfnScheduler_TaskActivate( tSchedulingTask * Task )
{
    TaskScheduler_Task_ID_Activated = Task->TaskId;
    Task->enTaskState = READY;
}
```

Figura 3-9. Implementación de la activación de la tarea.

La implementación de la función dará comienzo a las tareas del scheduler que se configuraron (Figura 3-10).

```c
void vfnTask_Scheduler(void)
{
    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    /*  1ms execution thread - used to derive two execution threads:                  */
    /*  a) 1ms thread (high priority tasks)                                           */
    /*  b) 100ms thread (lowest priority tasks)                                       */
    /*  As any other thread on this scheduler, all tasks must be executed in <=500us*/
    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    if( ( TaskScheduler_Task_ID_Activated == TASKS_1_MS )
         || ( TaskScheduler_Task_ID_Activated == TASKS_100_MS ) )
    {
        /* Make a copy of scheduled task ID */
        TasksScheduler_Task_ID_Backup = TaskScheduler_Task_ID_Activated;

        vfnScheduler_TaskStart (&TimeTriggeredTasks[TASKS_1_MS]);
        if( TaskScheduler_Task_ID_Activated == TASKS_100_MS )
        {
            vfnScheduler_TaskStart (&TimeTriggeredTasks[TASKS_100_MS]);
        }
        /* Verify that thread execution took less than 500 us */
        if( TasksScheduler_Task_ID_Backup == TaskScheduler_Task_ID_Activated )
        {
             /* In case execution of all thread took less than 500us */
            TaskScheduler_Task_ID_Activated = TASK_NULL;
        }
        else
        {
            gu8Scheduler_Status = TASK_SCHEDULER_OVERLOAD_1MS;
        }
    }
    else
    {
```

```c
        /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        /*  2ms execution thread - used to derive two execution threads:                  */
        /*  a) 2ms group A thread (high priority tasks)                                   */
        /*  b) 50ms thread (second lowest priority tasks)                                 */
        /*  As any other thread on this scheduler, all tasks must be executed in <=500us*/
        /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        if( ( TaskScheduler_Task_ID_Activated == TASKS_2_MS_A )
             || ( TaskScheduler_Task_ID_Activated == TASKS_50_MS ) )
        {
            /* Make a copy of scheduled task ID */
            TasksScheduler_Task_ID_Backup = TaskScheduler_Task_ID_Activated;

            vfnScheduler_TaskStart (&TimeTriggeredTasks[TASKS_2_MS_A]);
            if( TaskScheduler_Task_ID_Activated == TASKS_50_MS )
            {
                vfnScheduler_TaskStart (&TimeTriggeredTasks[TASKS_50_MS]);
            }
            /* Verify that thread execution took less than 500 us */
            if( TasksScheduler_Task_ID_Backup == TaskScheduler_Task_ID_Activated )
            {
                 /* In case execution of all thread took less than 500us */
                TaskScheduler_Task_ID_Activated = TASK_NULL;
            }
            else
            {
                gu8Scheduler_Status = TASK_SCHEDULER_OVERLOAD_2MS_A;
            }
        }
        else
        {
```

```
            /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
            /*  2ms execution thread - used to derive two execution threads:              */
            /*  a) 2ms group B thread (high priority tasks)                                */
            /*  b) 10ms thread (medium priority tasks)                                     */
            /*  As any other thread on this scheduler, all tasks must be executed in <=500us*/
            /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
            if( ( TaskScheduler_Task_ID_Activated == TASKS_2_MS_B )
                 || ( TaskScheduler_Task_ID_Activated == TASKS_10_MS ) )
            {
                /* Make a copy of scheduled task ID */
                TasksScheduler_Task_ID_Backup = TaskScheduler_Task_ID_Activated;

                vfnScheduler_TaskStart (&TimeTriggeredTasks[TASKS_2_MS_B]);
                if( TaskScheduler_Task_ID_Activated == TASKS_10_MS )
                {
                    vfnScheduler_TaskStart (&TimeTriggeredTasks[TASKS_10_MS]);
                }
                 /* Verify that thread execution took less than 500 us */
                if( TasksScheduler_Task_ID_Backup == TaskScheduler_Task_ID_Activated )
                {
                    /* In case execution of all thread took less than 500us */
                    TaskScheduler_Task_ID_Activated = TASK_NULL;
                }
                else
                {
                    gu8Scheduler_Status = TASK_SCHEDULER_OVERLOAD_2MS_B;
                }
            }
        }
    }
}
```

Figura 3-9. Implementación del scheduler.

La implementación de la función de retro llamada que ejecutará el systick cada 500us, se muestra en la Figura 3-11.

```
void vfnScheduler_Callback(void)
{
    /*-- Update scheduler control variable --*/
    gu8Scheduler_Counter++;

    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    /*  1ms execution thread - used to derive two execution threads:               */
    /*  a) 1ms thread (highest priority tasks)                                      */
    /*  b) 100ms thread (lowest priority tasks)                                     */
    /*  As any other thread on this scheduling scheme,                              */
    /*  all tasks must be executed in <= 500us                                      */
    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    if( ( gu8Scheduler_Counter & 0x01u ) == 0x01u )
    {
        u8_100ms_Counter++;
        /*-- Allow 100 ms periodic tasks to be executed --*/
        if( u8_100ms_Counter >= 100u )
        {
            /* Indicate that Task is Ready to be executed */
            vfnScheduler_TaskActivate(&TimeTriggeredTasks[TASKS_100_MS]);
            u8_100ms_Counter       = 0u;
        }
        /*-- Allow 1 ms periodic tasks to be executed --*/
        else
        {
            vfnScheduler_TaskActivate(&TimeTriggeredTasks[TASKS_1_MS]);
        }
    }
    else
    {
```

```
        /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        /*  2ms execution thread - used to derive two execution threads:               */
        /*  a) 2ms group A thread (high priority tasks)                                 */
        /*  b) 50ms thread (second lowest priority tasks)                               */
        /*  As any other thread on this scheduling scheme,                              */
        /*  all tasks must be executed in <= 500us                                      */
        /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        if( ( gu8Scheduler_Counter & 0x02u ) == 0x02u )
        {
            u8_50ms_Counter++;
            /*-- Allow 50 ms periodic tasks to be executed --*/
            if( u8_50ms_Counter >= 25u )
            {
                vfnScheduler_TaskActivate(&TimeTriggeredTasks[TASKS_50_MS]);
                u8_50ms_Counter        = 0u;
            }
            /*-- Allow 2 ms group A periodic tasks to be executed --*/
            else
            {
                vfnScheduler_TaskActivate(&TimeTriggeredTasks[TASKS_2_MS_A]);
            }
        }
        else
        {
```

```
            /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
            /*  2ms execution thread - used to derive two execution threads:                  */
            /*  a) 2ms group B thread (high priority tasks)                                   */
            /*  b) 10ms thread (medium priority tasks)                                        */
            /*  As any other thread on this scheduling scheme,                                */
            /*  all tasks must be executed in <= 500us                                        */
            /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
            if( ( gu8Scheduler_Counter & 0x03u ) == 0x00u )
            {
                u8_10ms_Counter++;
                /*-- Allow 10 ms periodic tasks to be executed --*/
                if( u8_10ms_Counter >= 5u )
                {
                    vfnScheduler_TaskActivate(&TimeTriggeredTasks[TASKS_10_MS]);
                    u8_10ms_Counter        = 0u;
                }
                /*-- Allow 2 ms group B periodic tasks to be executed --*/
                else
                {
                    vfnScheduler_TaskActivate(&TimeTriggeredTasks[TASKS_2_MS_B]);
                }
            }
        }
    }
}
/*****************************************************************************************************/
```

Figura 3-10. Implementación de la retro llamada del SysTick.

3.4.2 Declaración e implementación de las tareas

4. La declaración de la lista de tareas que procesará el scheduler durante su ejecución se muestra en la Figura 3-12.

```
/*****************************************************************************************************
* Declaration of module wide FUNCTIONS
*****************************************************************************************************/

/* List of tasks to be executed @ 1ms */
void TASKS_LIST_1MS( void );

/* List of tasks to be executed @ 2ms, first group */
void TASKS_LIST_2MS_A( void );

/* List of tasks to be executed @ 2ms, second group */
void TASKS_LIST_2MS_B( void );

/* List of tasks to be executed @ 10ms */
void TASKS_LIST_10MS(  void );

/* List of tasks to be executed @ 50ms */
void TASKS_LIST_50MS( void );

/* List of tasks to be executed @ 100ms */
void TASKS_LIST_100MS( void );
```

Figura 3-11. Prototipos de las tareas.

La implementación de las tareas que se ejecutarán cada instante de tiempo, dependiendo de su configuración, se muestra a continuación en la Figura 3-13.

```
/* List of tasks to be executed @ 1ms */
void TASKS_LIST_1MS( void )
{;}
/* List of tasks to be executed @ 2ms, first group */
void TASKS_LIST_2MS_A(void)
{;}
/* List of tasks to be executed @ 2ms, second group */
void TASKS_LIST_2MS_B( void )
{;}
/* List of tasks to be executed @ 10ms */
void TASKS_LIST_10MS( void )
{;}
/* List of tasks to be executed @ 50ms */
void TASKS_LIST_50MS( void )
{;}
/* List of tasks to be executed @ 100ms */
void TASKS_LIST_100MS( void )
{;}
```

Figura 3-12. Implementación de las tareas.

4.1 Resultados finales

Se realizaron varias pruebas para poder saber que tan preciso es nuestro sistema al momento de ejecutar las tareas definidas al ser programado. Se utilizó un LED de la misma tarjeta de desarrollo para medir el tiempo de activación y ejecución, además se agregaron tareas al scheduler, tales como transmisión de tramas de CAN, transmisión de tramas de LIN, conversiones del DAC y encender y apagar el LED. Todo esto se hizo con la finalidad de poder comprobar que las tareas se ejecuten de acuerdo a lo esperado y se minimicen los errores al momento de la ejecución.

4.1.1 Tarea de 1 ms

La medición de la tarea de 1 ms se muestra en la Imagen 4-1, donde podemos observar que cada 1 ms la señal cambia de 0 a 5 volts. Esto quiere decir que el led enciende y apaga cada 1 ms, obteniendo una medición correcta del scheduler.

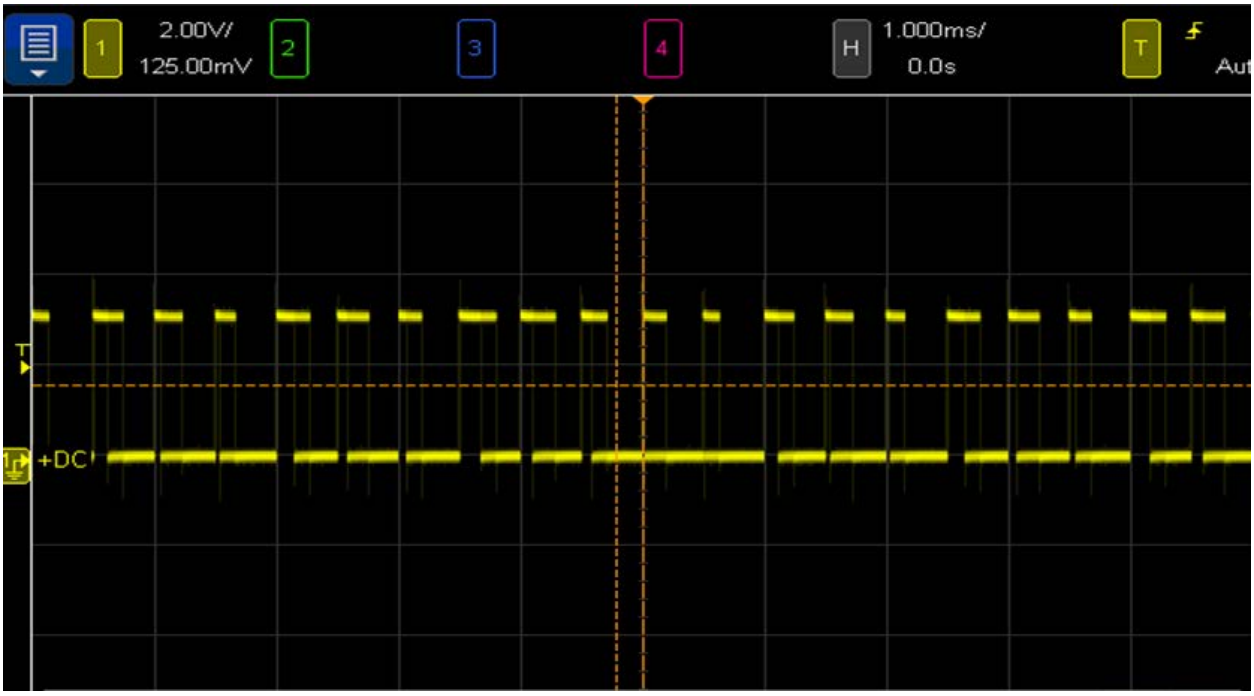


Figura 13-14. Tarea de 1 ms.

4.1.2 Tarea de 2 ms

La medición de la tarea de 2 ms A y B tienen el mismo comportamiento y la ejecución de ambas controlan el mismo estándar de tiempo. En la Figura 4-2 se muestra que cada 2 ms la señal cambia de 0 a 5 volts, esto indica que el LED enciende y apaga cada 2 ms, obteniendo una medición correcta de nuestro scheduler.

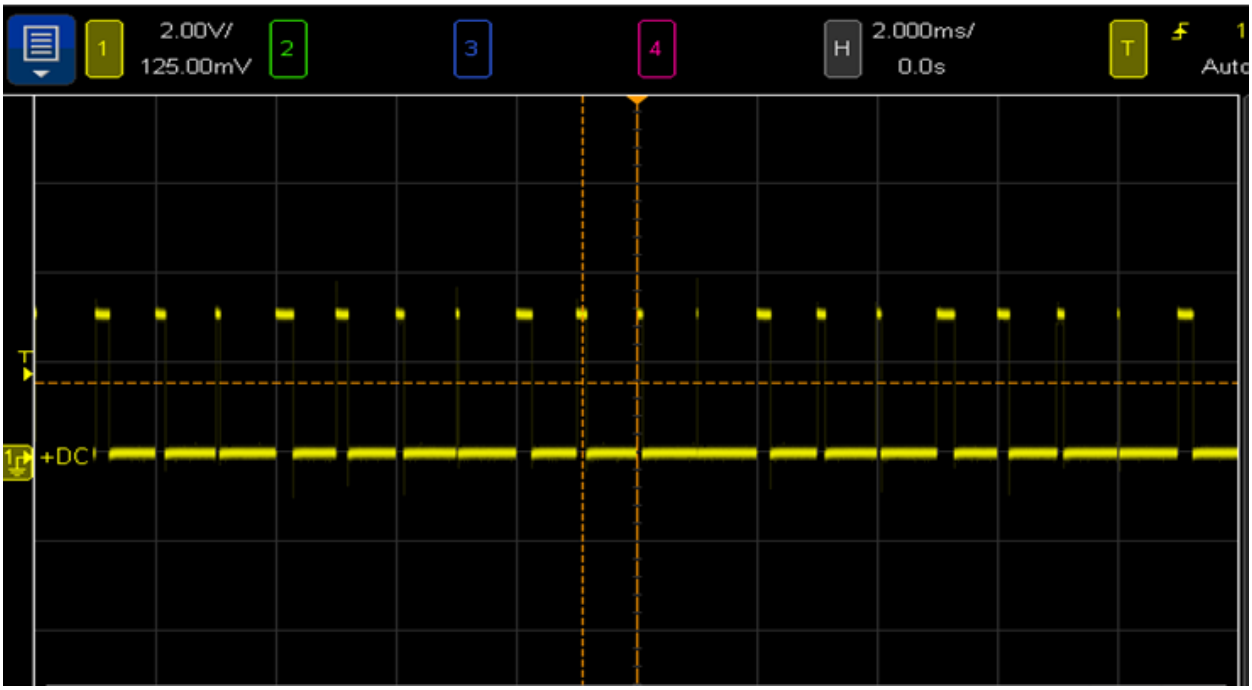


Figura 15-1. Tarea de 2 ms A y B.

4.1.3 Tarea de 10 ms

La medición de la tarea de 10 ms se muestra en la Figura 4-3, donde podemos observar que cada 10 ms la señal cambia de 0 a 5 volts. Esto indica que el LED enciende y apaga cada 10 ms, obteniendo una medición correcta de nuestro scheduler.

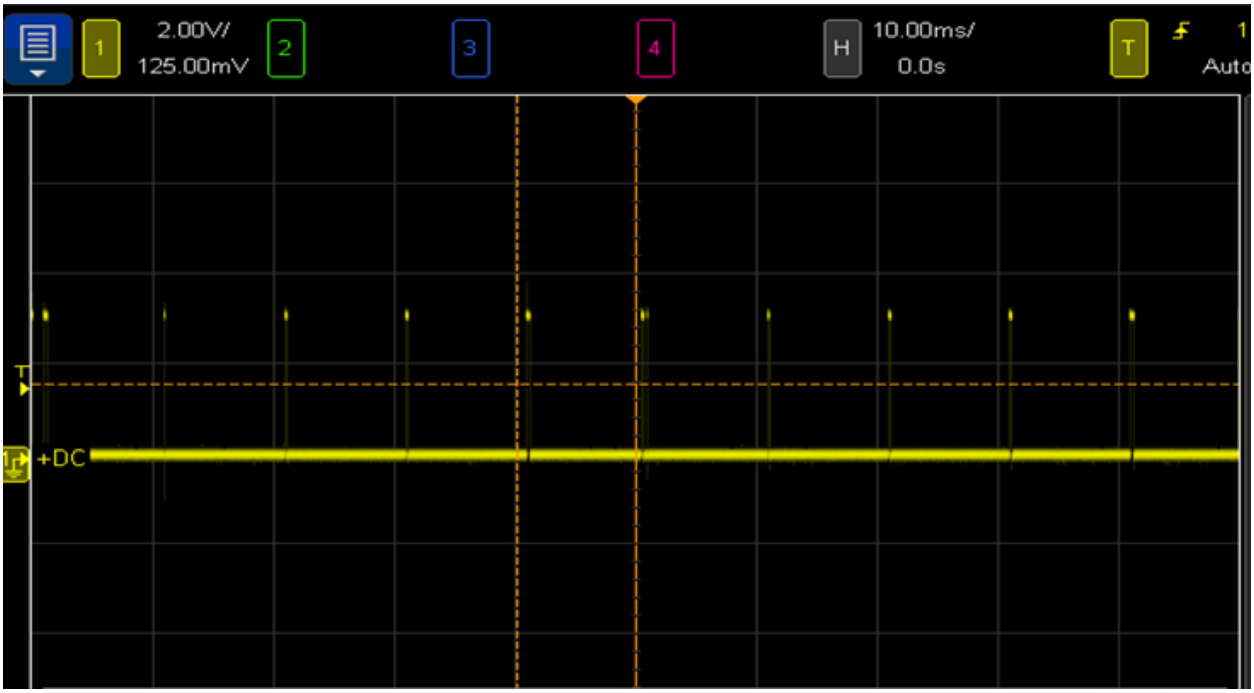


Figura 16-2. Tarea de 10 ms.

4.1.4 Tarea de 50 ms

La medición de la tarea de 50 ms se muestra en la Figura 4-4, donde podemos observar que cada 10 ms la señal cambia de estado lógico bajo a alto (0V a 5V). Esto indica que el LED enciende y apaga cada 10 ms, obteniendo una medición correcta de nuestro scheduler.

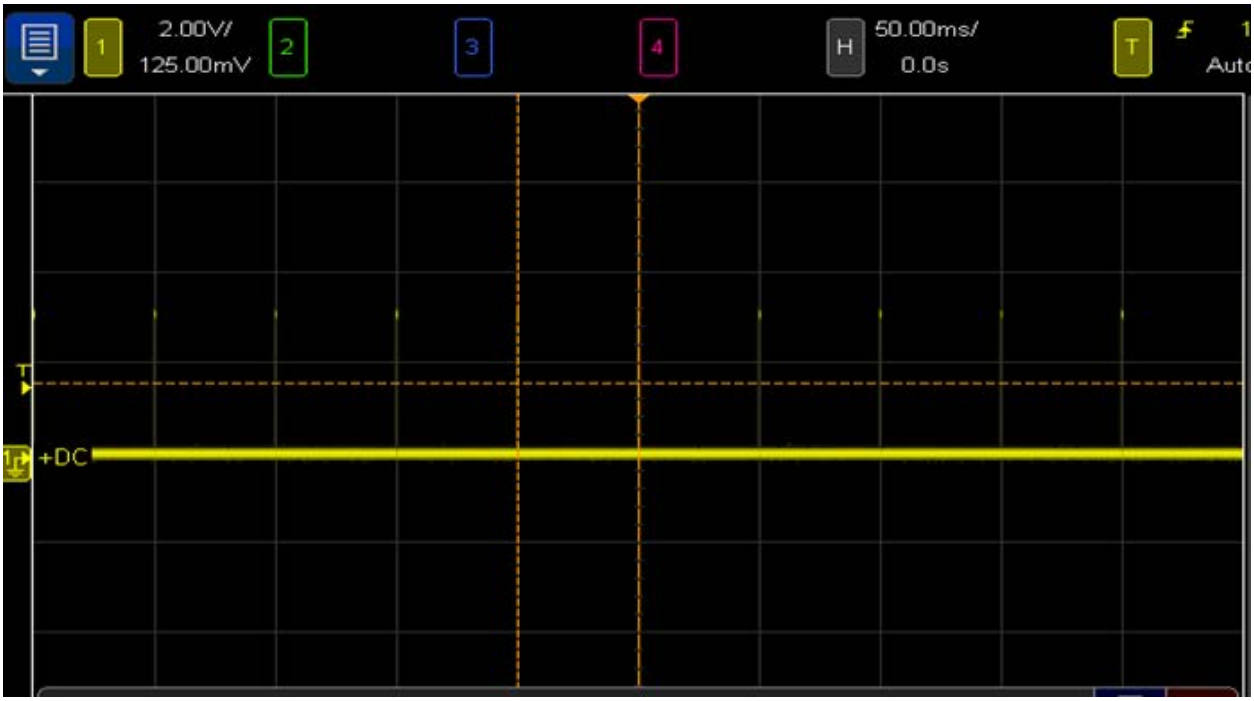


Figura 17-3. Tarea de 50 ms.

4.1.5 Tarea de 100 ms

La medición de la tarea de 100 ms se muestra en la Figura 4-5, brindándonos una perspectiva periódica de 100 ms donde la señal cambia de 0 a 5 volts. Lo anterior, indica que el LED enciende y apaga cada 100 ms, obteniendo una medición correcta de nuestro scheduler.

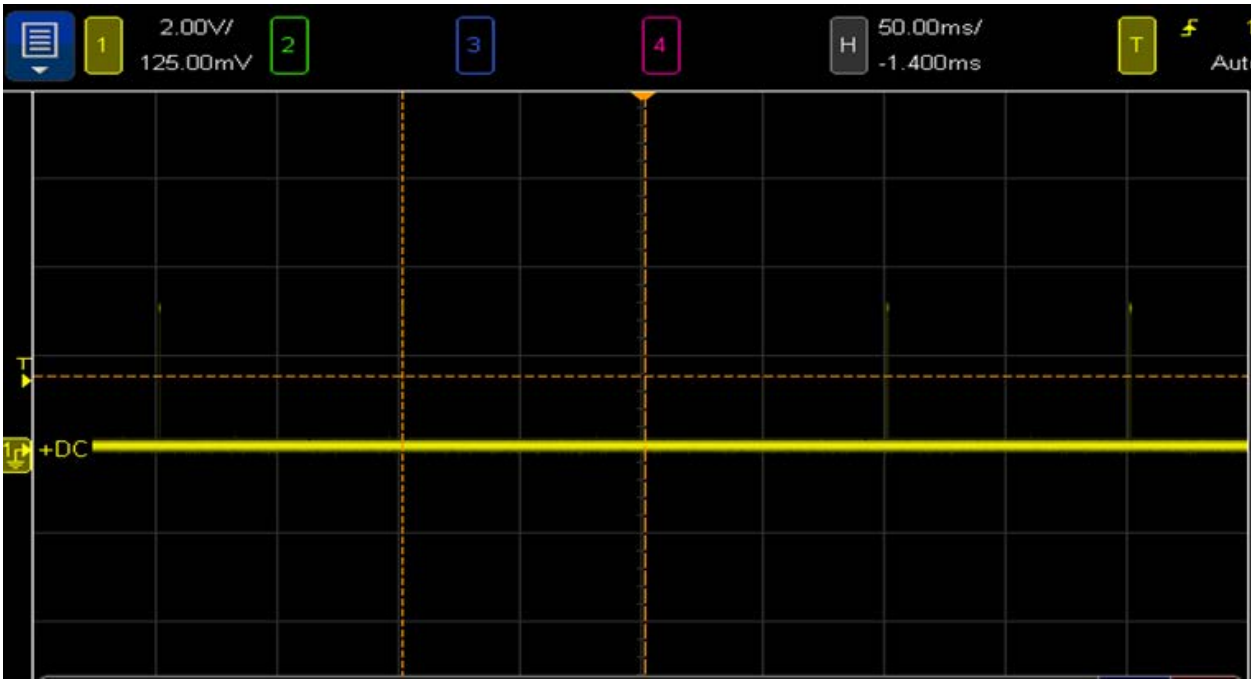


Figura 18-4. Tarea de 100 ms.

4.1.6 Ejecución de DAC y LIN

Al ejecutar las tareas de LIN y DAC como se muestra en la Figura 4-6, ambas tareas se ejecutan en el tiempo establecido por el scheduler. La tarea de LIN está en la tarea de 2 ms A y la tarea de DAC está en la de 2ms B.

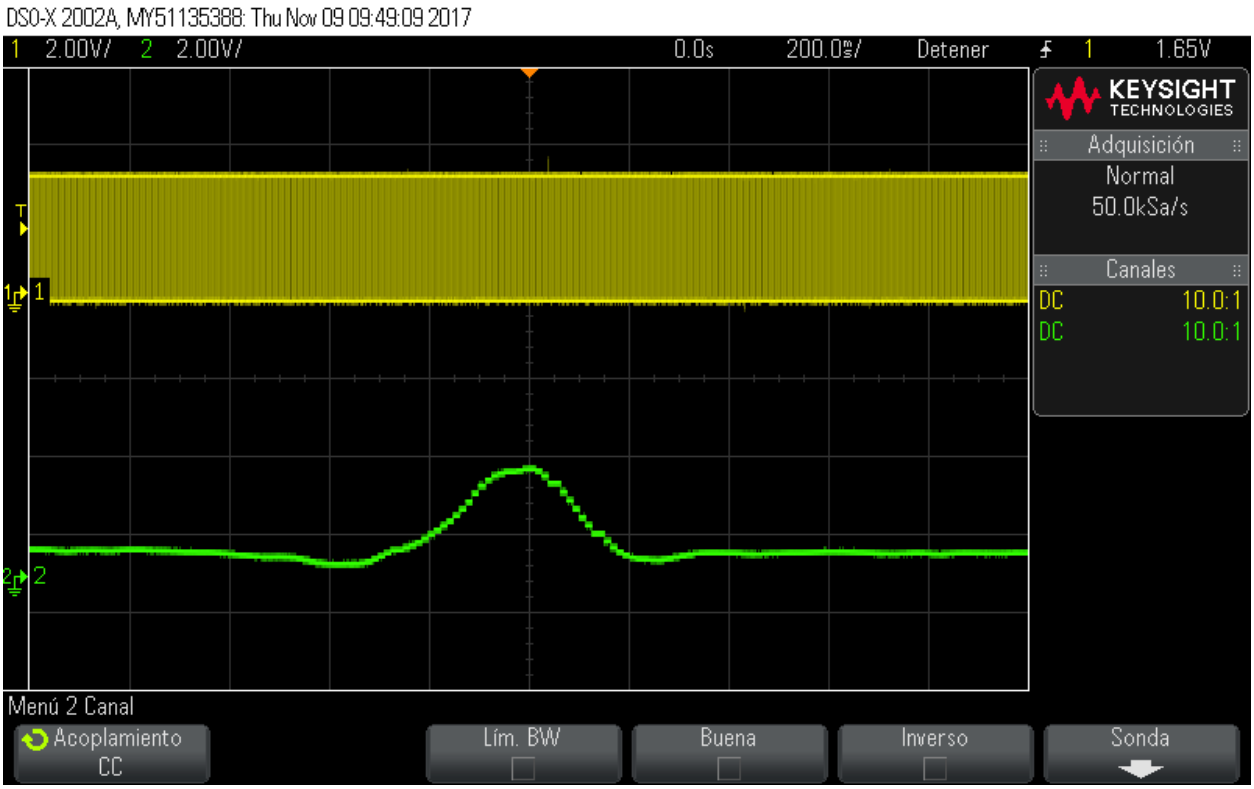


Figura 19-5. Tarea de LIN y DAC.

4.1.7 Ejecución de DAC y LED

La ejecución con el DAC y LED se observa en la Figura 4-7. La tarea del LED se encuentra en la tarea de 10 ms, donde podemos observar cómo se ejecutan ambas tareas sin ningún error y respetando los tiempos establecidos por el scheduler.

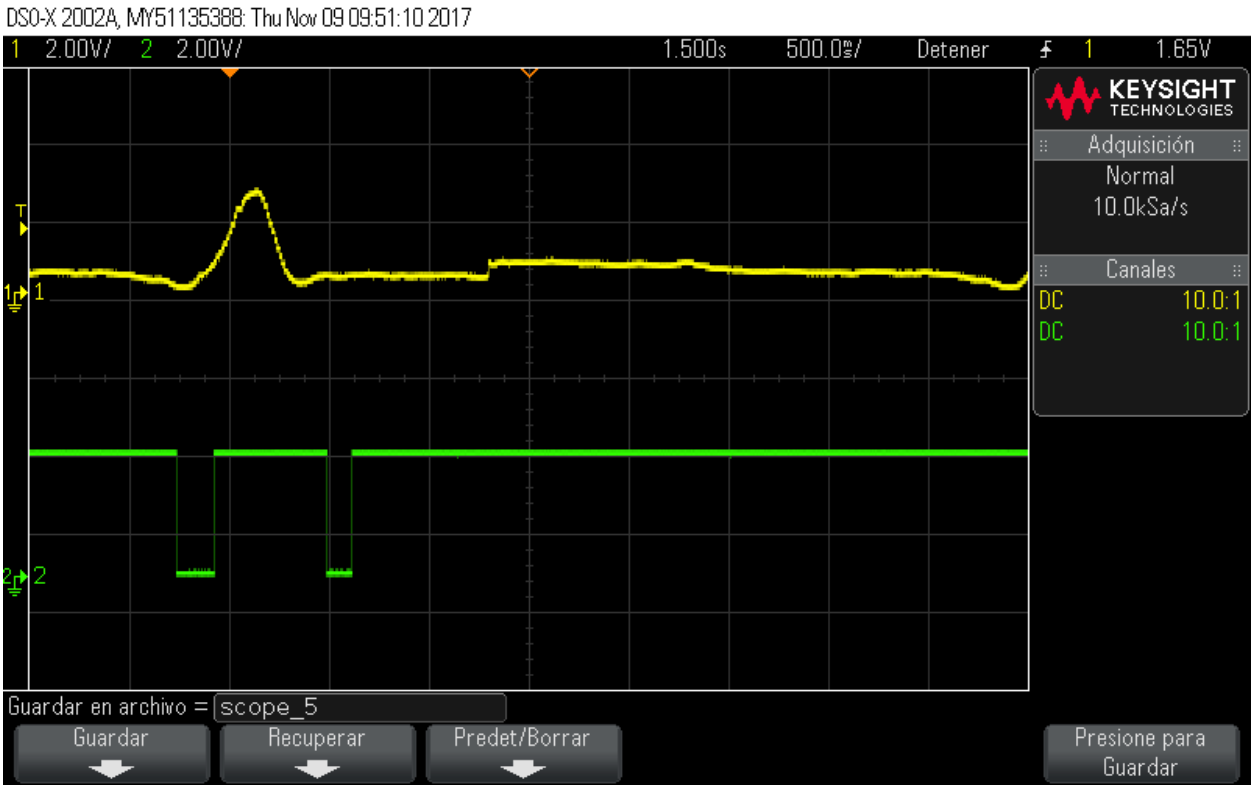


Figura 20-6. Tarea de DAC y LED.

4.1.8 Ejecución de CAN

La tarea de CAN, en la Figura 4-8 se observa la trama de información de CAN enviada desde la tarjeta de evaluación al mundo exterior. El evento es atendido por la tarea de 1 ms que ejecuta el sistema operativo. Es importante mencionar como el tiempo de la tarea es periódico y respeta la ejecución de las otras tareas sin que se traslapen unas con otras.

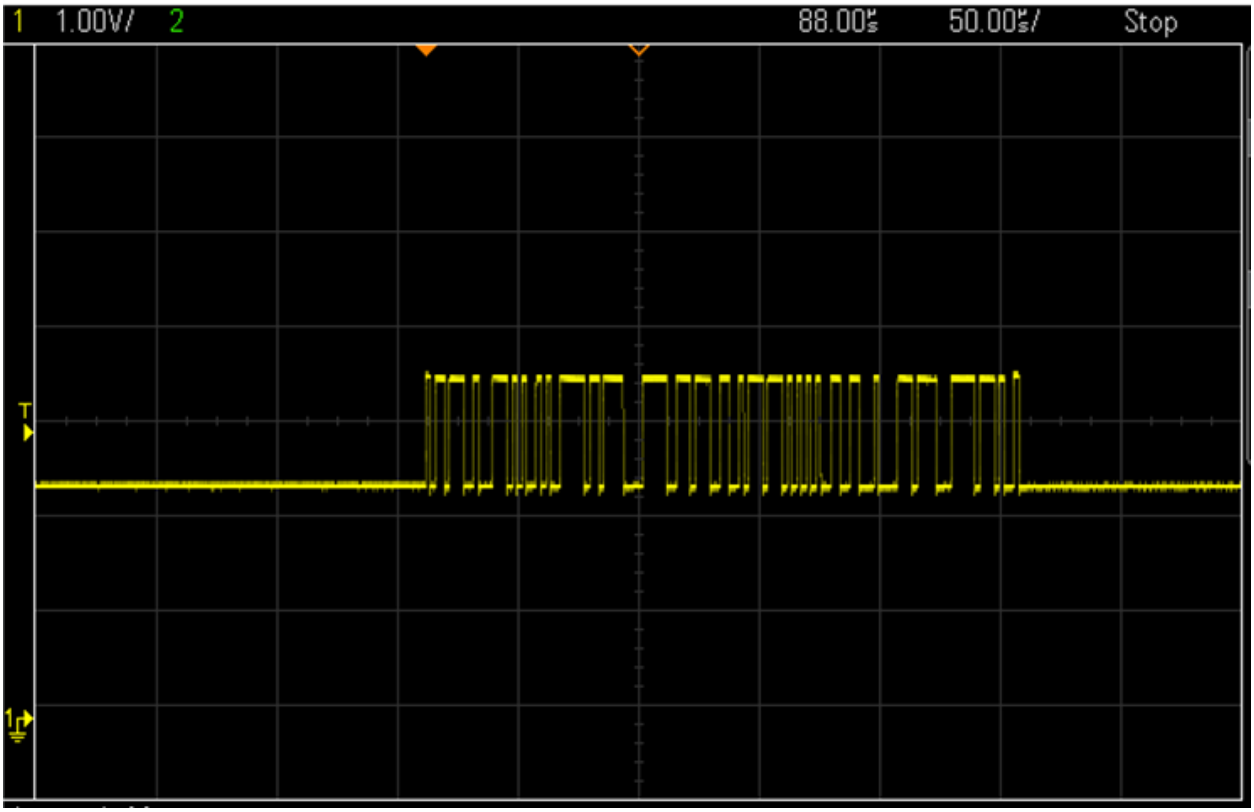


Figura 21-7. Tarea de CAN

Nuestro scheduler en comparación de otros que se han desarrollado mediante investigaciones para la disminución de latencia, se puede observar que es muy preciso en sus interrupciones, por lo cual disminuye considerablemente la latencia. Como una mejora para trabajos futuros, el Scheduler puede operar y ser configurado como un sistema pre-empt o cooperative. En ambos casos, el comportamiento del scheduler es idéntico por lo que no se ve afectado en su tiempo de ejecución de las tareas programadas, además se pueden agregar más tareas incrementando la cuenta del contador y así lograr secuencias repetitivas de los números binarios. Además, el scheduler que se desarrolló, puede ser utilizado en diferentes sectores industriales como lo es la aeroespacial, automotriz, aplicaciones en consumo de energía, robótica, automatización e instrumentación de pruebas, etc., esta es una de las ventajas de nuestro sistema aplicado en la tarjeta de desarrollo SAM V71 Xplained Ultra.

Conclusiones

El sistema desarrollado a lo largo de este trabajo comprende un scheduler estilo cooperativo capaz de operar 6 tareas en tiempos determinados por la configuración del core. El scheduler puesto en práctica comprende 4 tareas las cuales son ejecutadas en tiempo y forma como fueron programadas. Las 4 tareas son periódicas y ninguna de ellas se traslapa entre sí, permitiendo un sistema cooperativo independiente que puede ser capaz de usarse en un sistema embebido tomando como base este scheduler.

El scheduler puede ser aplicarlo en el sector automotriz, dónde los tiempos de ejecución de los sistemas operativos en tiempo real usados en divisiones de diseño como lo es powertrain. En esta división, existen diferentes aplicaciones en los sistemas de control como lo son la inyección de frenos, cambio de trasmisiones, sensores de aceite, nivel y presión en bomba de gasolina etc. Se hace énfasis en el sector automotriz, dado se usan los protocolos de comunicación automotrices estándares, como lo son Local Interconnect Network y Controller Area Network. Estos protocolos de comunicación permiten la comunicación en la red del vehículo obteniendo información de sensores en tiempo real.

El software embebido automotriz sigue el estándar de AUTOSAR, en él, la capa de servicios es quien provee la ejecución de los drivers de comunicación entre las capas RTE y MCU permitiendo comunicación y envió de información. Derivado de lo anterior las tareas de comunicación son de mayor prioridad, de esta manera, las capas la programación tienen la estandarización para el software embebido en un microcontrolador siendo así, las adecuadas para la migración del software a un microcontrolador de distintas familias sin crear conflictos al ser programado y por lo tanto, el schaduler es funcional en otro chip.

Bibliografía

[1] K. Chakrabarty. Design of system-on-a-chip test access architectures under place-and-route and power constraints. *Proc. Design Automation Conf.*

[2] K. Chakrabarty. Optimal test access architectures for system-on-achip. *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, January 2001,

[3] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM20*, 1 (Jan 1973),

[4] LEHOCZKY, J., AND THUEL, S. Algorithms for schedulinghard aperiodic tasks in fixed-priority systems using slackstealing. In *Proceedings of the IEEE Real-Time SystemsSymposium* (1994).

[5] M. A. Rivas, "Planificación de tareas en sistemas operativos de tiempo real estricto para aplicaciones empotradas", *Dr. Fac. Cienc. Dep. Electrónica Comput. Univ. Cantab. Santander Esp.*, 2002.

[6] L. E. L. del Foyo, "Administración de Interrupciones en Sistemas Operativos de Tiempo Real".

[7] G. A. Elliott, "Real-time scheduling for GPUS with applications in advanced automotive systems", The University of North Carolina at Chapel Hill, 2015.

[8] "What is operating system (OS)? - Definition from WhatIs.com", *WhatIs.com*. [En línea]. Disponible en: http://whatis.techtarget.com/definition/operating-system-OS. [Consultado: 15-oct-2017].

[9] "Embedded operating system", *Wikipedia*. 31-ago-2017.

[10] P. M. Sagar, "Embedded operating systems for real-time applications", en *M. Tech. credit seminar report, Electronic Systems Group, EE Dept, IIT Bombay, Submitted in November*, 2002.

[11] "Real Time Operating System - Hard RTOS and Soft RTOS", *Electronics Hub*, 20-ago-2015. .

[12] tutorialspoint.com, “Operating System - Process Scheduling”, *www.tutorialspoint.com*. [En línea]. Disponible en: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm. [Consultado: 22-oct-2017].

[13] “SAM V71 Xplained Ultra Evaluation Kit”. [En línea]. Disponible en: http://www.atmel.com/tools/atsamv71-xult.aspx. [Consultado: 22-oct-2017].