RTOS en placa de desarrollo EDU-CIAA para el control de un accionamiento electromecánico básico

Gonzalo Gabriel Fernández

***Resumen*—Proyecto final de la cátedra Microcontroladores y Electrónica de Potencia de la Facultad de Ingeniería de la Universidad Nacional de Cuyo. Consiste en el diseño e implementación de un sistema embebido para el control de un accionamiento electromecánico básico, basado en el sistema operativo de tiempo real FreeRTOS, sobre la placa de desarrollo EDU-CIAA-NXP del proyecto CIAA. El principal objetivo del proyecto es la implementación de todas las características fundamentales del RTOS.**

I. Introducción

El objetivo del trabajo es la implementación de una aplicación basada en el sistema operativo de tiempo real FreeRTOS, sobre la placa de desarrollo EDU-CIAA, para el control de un sistema electromecánico básico que tiene como fin únicamente la demostración visual de lo que sucede en la electrónica. Se busca implementar todos los conceptos fundamentales de FreeRTOS, basándose en el libro *Mastering the FreeRTOS Real Time Kernel*[1], por más que algunos de estos conceptos tengan una aplicación forzada y se alejen de lo práctico o lo más conveniente. Esto es porque la principal finalidad del proyecto es aprender a utilizar FreeRTOS por sobre obtener un controlador del sistema con buen criterio de diseño y eficiente.

Además, el proyecto también busca una introducción al flujo de trabajo del proyecto CIAA (Computadora Industrial Abierta Argentina). Salvo algunas excepciones, no se profundizó en la programación de los microcontroladores a nivel de registros, sino que se utilizó la capa de abstracción proveída por las librerías sAPI y LPCOpen, incluidas en el firmware de dicho proyecto CIAA.

II. Aplicaciones de tiempo real

Previo al planteo de la elección de un sistema operativo de tiempo real como base en el diseño de una determinada aplicación, es importante definir cuándo una aplicación es "de tiempo real". Se dice que una aplicación es de tiempo real cuando posee características tales que en su ejecución debe cumplir distintos requisitos de tiempo. Es decir, el diseñador debe garantizar que ciertos eventos se desarrollarán en un tiempo establecido. Decimos que la aplicación es **determinista**.

Un sistema perfectamente determinista no experimenta variación en la sincronización de los eventos. Sin embargo, en el mundo real, hasta el sistema más determinista presentará una variación. La variación entre el tiempo especificado en que debe ejecutarse un evento y el tiempo en que se ejecuta realmente se denomina **jitter**. El *jitter* ocurre tanto para un evento que se ejecuta después del tiempo establecido como para uno que se produce de manera temprana.

La cantidad de *jitter* que una aplicación puede tolerar sin dejar de cumplir los requisitos de tiempo define el *jitter aceptable*. Teniendo en cuenta este concepto, ahora podemos clasificar las aplicaciones en sistemas de requisitos de tiempo tipo *soft* (o su traducción al español como suave o blando) y requisitos de tiempo tipo *hard* (o su traducción al español como duro). Gran parte de la bibliografía consultada concuerda en que los requisitos de tiempo *soft* son aquellos que, de no cumplirse el tiempo especificado, tiene como consecuencia una disminución en la calidad del desempeño de la aplicación; y por otro lado, los requisitos de tiempo *hard* aquellos que, de no cumplirse el tiempo especificado, resulta en una falla del sistema. Sin embargo, teniendo claro el concepto de *jitter aceptable* podemos decir que requisitos de tiempo blandos se darán en sistemas con *jitter aceptable* alto y, por el contrario, los requisitos de tiempo duros se darán en sistemas con *jitter aceptable* bajo.

Para clarificar la diferencia entre requisitos duros y blandos, ciertas fuentes suelen colocar como ejemplos de aplicaciones de tiempo real duro la navegación autónoma, marcapasos o instrumentación médica sensible, es decir, aplicaciones donde una falla es realmente crítica; y otros ejemplos donde solo se afecta la calidad (servicios de chat y similares) como aplicaciones de tiempo real blando. Al realizar este trabajo, una de las conclusiones es que este tipo de ejemplos generan confusión, ya que quién se esta introduciendo en el concepto puede confundir la medida de qué tan duro es un requisito de tiempo con una medida de severidad de la falla de un sistema.

Lo principal a tener en cuenta es que en última instancia, ante un determinado comportamiento de un sistema, quien establece el límite entre disminución en la calidad de desempeño y una falla es el diseñador. También es importante que, en general, no es todo el sistema el que posee estas restricciones de tiempo, sino un determinado aspecto en particular. Para resaltar esto, se puede utilizar como ejemplo el proyecto llevado a cabo, donde los motores deben moverse a posiciones definidas en un determinado lapso de tiempo (esto establece la velocidad de los motores), por lo que esa es una característica del sistema que requiere tiempo real (*jitter* aceptable bajo); mientras que, por ejemplo, no tiene relevancia en qué momento en específico se expone el valor de posición del motor en un display (*jitter* aceptable muy alto) (en otra aplicación diferente esto si puede llegar a ser una característica que en el diseño se contemple como de tiempo real duro).

En definitiva, una aplicación es de tiempo real si como diseñador se debe garantizar que ciertas funcionalidades del sistema se ejecutan en determinados instantes de tiempo. Si al diseñar un sistema, se poseen diferentes procesos en los que el tiempo directamente no es una variable relevante y se

ejecutarán cuando el procesador este disponible, entonces el requisito de tiempo real es nulo y no es necesario considerar una plataforma que soporte esta característica.

III. Sistemas Operativos de Tiempo Real

Un sistema operativo con la característica de **tiempo real** es entonces aquel cuya estructura está diseñada para cumplir requisitos de tiempo. Estos sistemas brindan todas las ventajas de un sistema operativo convencional: el programa ahora se plantea en diversos hilos que se ejecutarán en pseudo paralelismo, prácticamente independientes entre sí. Esto hace que la aplicación sea más fácil de mantener y escalar, favorece la modularidad del sistema, la tarea de desarrollo es más sencilla si esta involucrada más de una persona, mayor reutilización de código, mayor eficiencia (si la aplicación es lo suficientemente compleja como para justificar un RTOS), testing más sencillo, etc. Incluso se dispone del *Idle* que básicamente es el proceso ejecutado en el tiempo muerto, que puede utilizarse para distintos fines como bajo consumo, medir tiempo libre, diferentes chequeos, etc.

Específicamente la característica de tiempo real del sistema operativo genera una capa de abstracción donde el desarrollador puede despreocuparse (hasta cierto punto) del orden de ejecución del programa, utilizando las API relacionadas con la sincronización en el tiempo de los distintos procesos. Esto marca la principal diferencia con los sistemas embebidos sin sistema operativo, donde la programación sigue un esquema de "super loop".

En una estructura super loop o super bucle el sistema se desenvolverá (usualmente) a través de una máquina de estados, se utilizan esperas pasivas (*delay*), el control de tiempo puede realizarse a través del uso de *timers* y su rutina de interrupción asociada, la comunicación entre "tareas" se implementará mediante el uso de variables globales, y las prioridades de ejecución de cada tarea dependerá del orden en el bloque principal. En sistemas simples esto no es un problema, pero en sistemas complejos, la opción más sencilla puede ser el uso de un RTOS donde se plantean diferentes procesos por separado y el sistema operativo se encarga de cuál ocupa el procesador en cada instante.

III-A. FreeRTOS

FreeRTOS es una de la varias opciones de sistemas operativos de tiempo real de código abierto. Su objetivo es ser simple y pequeño, es desarrollado y mantenido por *Real Time Engineers Ltd.*. Es distribuido bajo licencia MIT, y la amplia lista de arquitecturas que soporta es uno de los motivos por el cual es tan popular.

IV. Computadora Industrial Abierta Argentina

La **Computadora Industrial Abierta Argentina** (CIAA) es un proyecto que nace en el año 2013 como una iniciativa conjunta entre el sector académico y el industrial de Argentina.

La primera versión de la CIAA es denominada CIAA-NXP, por estar basada en un procesador de la empresa *NXP Semiconductors*. Luego se desarrollaron otras versiones basadas en procesadores de otras marcas, como la CIAA-FSL, la CIAA-INTEL, la CIAA-PIC, etc. Esto convierte a la CIAA no solo en la primer y única computadora **industrial** y **abierta**, sino también en la primera realmente **libre**, ya que su diseño no está atado a los procesadores de una determinada compañía.

Además, se diseñó una versión educativa de la plataforma, denominada **EDU-CIAA**, más simple y de menor costo, para lograr un impacto en la enseñanza primaria, secundaria y universitaria.

IV-A. Placa de desarrollo EDU-CIAA-NXP

La EDU-CIAA-NXP (figura 1) es entonces, una versión de bajo costo de la CIAA-NXP y es la utilizada en este proyecto.

Figura 1. Placa de desarrollo EDU-CIAA-NXP.

Su microcontrolador es el LPC4337 (dual core ARM Cortex-M4F y Cortex-M0). La placa también cuenta con los módulos que pueden observarse en la figura 2.

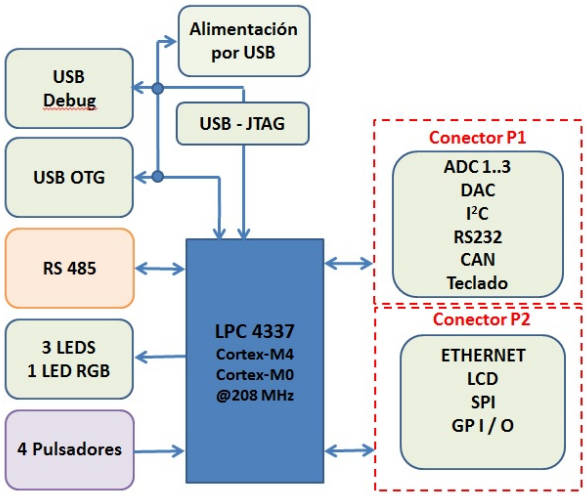

Figura 2. Diagrama de bloques con módulos de placa EDU-CIAA-NXP

V. Accionamiento electromecánico: brazo robótico paralelo básico

Para reflejar de una forma concreta las acciones ejecutadas en la electrónica del sistema (y también para tener objetivos definidos en su diseño), se optó por implementar el control de un brazo robótico paralelo básico. Su tamaño es reducido, ya

que solo tiene un fin educacional, y por lo tanto los actuadores son sencillos y de poca potencia. El robot construído es denominado EEZYBOTARM MK3, publicado por EEZYrobots y diseñado por Carlo Franciscone.

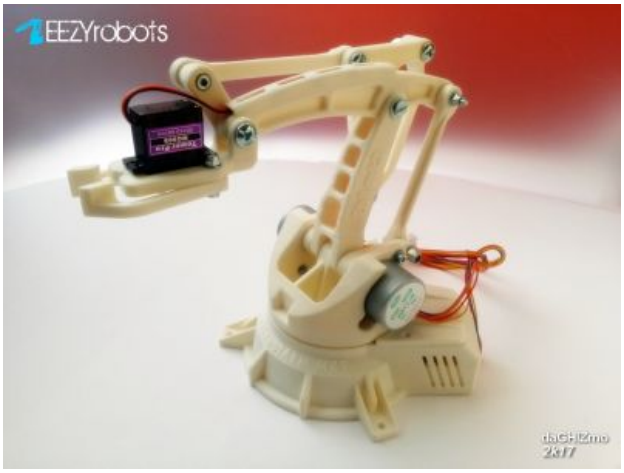

Figura 3. Brazo robótico paralelo EEZYBOTARM MK3 de EEZYrobots, diseñado por Carlo Franciscone.

Los actuadores del robot consisten en tres motores paso a paso monopolares 28BYJ-48 con drivers ULN2003, y su extremo operativo es accionado por un servo motor Tower Pro SG90.

Además del brazo robótico con sus tres motores paso a paso y servo motor, la aplicación también tiene un display donde se observará la información del estado en que se encuentra el brazo. Las consignas que recibirán los motores será a través de protocolo UART desde PC o mediante un encoder rotativo. Como accesorio secundario en la aplicación se encuentran los LEDs de la placa EDU-CIAA como indicadores visuales.

VI. Flujo de trabajo y herramientas utilizadas

Todo el software se desarrolló en una PC con sistema operativo Linux. Para la programación en lenguaje C se utilizó el programa Eclipse IDE, configurado para compilar y cargar el programa a la placa desde el mismo entorno de desarrollo (ver documentación del firmware en la wiki del proyecto[2] en GitHub). Para completar el conjunto de herramientas se puede trabajar con OpenOCD como debugger, sin embargo en este proyecto no fue necesario su uso. Para la comunicación serial con la placa (UART) se utilizó el programa picocom, una herramienta sencilla por comando de línea. Para el control de versiones del proyecto, además del uso del firmware del proyecto CIAA, se utilizó git (con repositorio remoto en GitHub).

En cuánto a FreeRTOS, el port para la placa de desarrollo EDU-CIAA se encuentra dentro del firmware, por lo tanto no es necesario descargarlo desde ningun sitio externo al proyecto CIAA. Su configuración está dada por el archivo *config.mk* (definiciones de la compilación propias de la aplicación desarrollada) y principalmente por el archivo *FreeRTOSConfig.h* dentro de la carpeta de *headers* de la aplicación.

VII. Componentes fundamentales de FreeRTOS

A continuación se describe brevemente los diferentes componentes escenciales de FreeRTOS según el libro *Mastering the FreeRTOS Real Time Kernel*, y su uso dentro de la aplicación.

VII-A. Asignación de memoria dinámica

El proceso de asignación de memoria dinámica (asignación de memoria en tiempo de ejecución) es un aspecto muy importante en FreeRTOS, ya que es el modo en que el sistema asigna RAM cada vez que un objeto del kernel es creado.

El lenguaje C tiene dos funciones en la librería estándar para este tipo de asignación de memoria: *malloc()* y *free()*, para asignar y liberar memoria respectivamente. Algunos inconvenientes al usar este tipo de funciones son:

- Su implementación no suele ser pequeña para un sistema embebido.
- No suelen ser *thread-safe*, es decir que no se asegura que al llamarlas no se esta modificando o leyendo recursos compartidos.
- No son determinísticas. Dos llamados diferentes tomarán tiempos de ejecución diferentes.
- Pueden provocar fragmentación.

Es por esto que FreeRTOS brinda (y utiliza en su código fuente) como APIs públicas *pvPortMalloc()* y *vPortFree()*, que tienen el mismo prototipo que sus equivalentes de la librería estándar de C. Es importante estudiar los 5 tipos de pilas de memoria que brinda FreeRTOS como esquemas de asignación:

- *Heap 1*: Implementación básica de *pvPortMalloc()* y sin implementación de *vPortFree()*. Se utiliza en aplicaciones que nunca eliminan tareas ni otro tipos de objetos de kernel.
- *Heap 2*: Esquema que solo se mantiene por compatibilidad, pero no se recomienda utilizar.
- *Heap 3*: Utiliza las funciones de la librería estándar de C *malloc()* y *free()*.
- *Heap 4*: A diferencia del 3, combina bloques de memoria libre en un bloque de mayor tamaño, y utiliza un algoritmo de *first fit* (se asigna el primer bloque libre con espacio suficiente), lo que previene la fragmentación.
- *Heap 5*: La única diferencia con el 4, es que no esta limitado a la asignación de memoria partiendo de un único array asignado estáticamente.

En el proyecto no se utiliza asignación de memoria dinámica, ya que es desaconsejada en microcontroladores y sistemas críticos (regla MISRA 20.4, ver MISRA [3]) se optó por utilizar un esquema *heap 1*.

VII-B. Tareas

Las tareas son básicamente funciones implementadas en C y son programas en sí mismas. Su estructura es similar a la de una función *main()* típica en sistemas embebidos, con un bucle que debe ejecutarse de forma infinita. Cada tarea se ejecuta de forma independiente en *pseudoparalelismo*, y al crearse se les asigna automáticamente su propia pila de memoria.

Sus estados principales son *running* y *not running*, cuando el procesador esta ejecutando el código de la tarea y cuando no, respectivamente. El estado *not running* puede expandirse, y podemos decir que una tarea se puede encontrar en una de las siguientes situaciones:

- Bloqueada: La tarea se encuentra esperando un determinado evento.
- Suspendida: No están disponibles para el *scheduler*.
- Lista: Ya esta disponible para pasar a estado *running*, pero todavía no es seleccionada por el *scheduler*.

El cambio de estado en una tarea es únicamente dirigido por el *scheduler* de FreeRTOS. Para definir cuál tarea tiene más importancia de ejecución sobre otra, existe el concepcto de prioridad: cuando el *scheduler* deba decidir cuál es la próxima tarea a colocar en estado *running* deberá observar cuál es la de mayor prioridad.

Para seleccionar esa próxima tarea a ejecutar, el *scheduler* se ejecuta mediante una interrupción periódica denominada *tick interrupt*. En el proyecto la frecuencia de dicha interrupción es de 1000 Hz.

En el proyecto se implementaron las siguientes tareas:

- Para la comunicación UART se utilizan dos tareas, una para la recepción de datos (prioridad 5) y otra para transmisión (prioridad 4).
- Para el control del servomotor y otra para el control de los motores paso a paso, ambas con prioridad 3.
- Para el procesamiento de la información recibida a través del encoder incremental (prioridad 2).
- Para la manipulación y clasificación de mensajes y sus posibles consecuentes errores (prioridad 2).
- Una tarea que simplemente realiza un *toggle* se un LED para verificar visualmente que la aplicación sigue en funcionamiento (prioridad 1).
- Para el control del display LCD (prioridad 1).

La mayor prioridad permitida en la aplicación es 7 (mayor valor, mayor prioridad), al RTOS daemon se le asignó prioridad 6.

VII-C. Algoritmos de scheduler

El algoritmo del *scheduler* es el que decidirá qué tarea ejecutará de entre la lista de tareas en estado listas o *ready* y cuándo cambiar de una a otra.

El tipo de algoritmo a utilizar se configura con dos variables: uso de **preemption** y uso de **time slicing**:

- *Scheduling de prioridad fija pre-emptive con time slicing*: Ambas variables en 1. No cambia la prioridad de las tareas (sí pueden cambiarse su prioridad las tareas mismas), inmediatamente se hace "pre-empt" de la tarea en estado *running* si otra de mayor prioridad entra en estado *ready* (esto quiere decir que el scheduler automáticamente en el *tick* le dará el procesador a la tarea de mayor prioridad en estado *ready*), y el *time slicing* indica que a tareas de igual prioridad en estado *ready* se les brindará el procesador alternadamente en intervalos de tiempo iguales (esos intervalos de tiempo son iguales a dos interrupciones por tick).
- *Scheduling de prioridad fija pre-emptive sin time slicing*: *time slicing* en 0. En este caso al no haber *time slicing*, cuando tareas de igual prioridad se encuentren en estado *ready*, solo habrá cambio de contexto (el *scheduler* asigna el procesador a una tarea diferente) cuando la tarea en estado *running* se bloquee o se suspenda o haya *pre-empt* de una tarea de prioridad mayor.
- *Scheduling cooperativo*: *preemption* en 0 y *time slicing* en cualquier valor. Solo hay cambio de contexto cuando la tarea en estado *running* se bloquee o suspenda, o cuando explícitamente hace un *yield* (solicitar explícitamente la entrada del *scheduler*).

En este trabajo se optó por utilizar un algoritmo de *scheduling* de prioridad fija pre-emptive con time slicing.

VII-D. Queues o colas

Las colas brindan un mecanismo de comunicación tarea a tarea, tarea a interrupción e interrupción a tarea. Normalmente se utilizan como buffers FIFO (*First Input First Output*). FreeRTOS implementa colas por copia y no por referencia, ya que estas presentan ventajas y en última instancia pueden utilizarse como colas por referencia guardando punteros.

Las características principales de esta estructura son las siguientes:

- Pueden ser accedidas por cualquier número de tareas o rutinas de interrupción.
- Se puede especificar un tiempo de bloqueo de una tarea si intentó leer una cola y ésta se encontraba vacía.
- También puede especificarse un tiempo de bloqueo de una tarea si intentó escribir en una cola y ésta se encontraba completa.
- Se pueden agrupar de forma que una tarea quede bloqueada hasta que alguna del grupo se encuentre disponible para su lectura o escritura.

En el proyecto se utilizaron colas de recepción y transmisión que contienen los caracteres recibidos o a ser enviados por UART, y también colas con elementos de tamaño variable como lo son la cola de mensajes recibidos y la cola de consignas de los motores (ver figuras 4 y 5).

Si una tarea recibe información desde múltiples fuentes (esto aplica tanto a colas como a semáforos), FreeRTOS nos permite agrupar esas fuentes en *sets*. Estos *sets* evitan que la tarea deba hacer un polling en turnos de las fuentes.

Un ejemplo de la aplicación de *sets* se implementó en la comunicación de pulsos del encoder a la tarea de control de dicho componente. Allí se implementaron dos semáforos contadores, ambos con pulsos, que se diferencian en la dirección enviada, es decir, el sentido de giro (ver figura 6).

VII-E. Mailbox

El mailbox dentro de los RTOS es una estructura de datos que permite guardar información a ser leída por múltiples tareas o interrupciones. FreeRTOS no hace diferencia entre un mailbox y una cola de longitud unitaria. En lugar de implementar una nueva estructura con diferente funcionamiento a una cola, lo que brinda son dos APIs que permiten adaptar las colas al funcionamiento de un mailbox.

Estas dos APIs son *xQueueOverwrite()* que permite sobrescribir el elemento de información en la cola (de hecho solo debe utilizarse en colas con longitud uno), y *xQueuePeek()* que permite realizar la lectura sin eliminar el dato de la cola (que es lo que sucedería ante una acción de *Receive*).

Un ejemplo de mailbox se encuentra en la selección del motor a modificar con el pulsador del encoder rotativo (ver figura 6).

VII-F. Timers por software

Los timers por software se utilizan para planificar la ejecución de una función en un determinado tiempo en el futuro, o a intervalos regulares a una frecuencia fija. Al ser por software no requieren de hardware, y son implementados completamente por el RTOS.

Existen dos tipos:

- De un disparo: Donde una vez ejecutada la función de *callback* deben resetearse manualmente.
- De recarga automática: Se resetean automáticamente por lo que la función de *callback* se ejecutará periódicamente.

Se dice que los timers tienen dos estados: *running* cuando expiran y ejecutan una función de *callback* y dormidos el resto del tiempo.

Todas las funciones de *callback* de los timers por software se ejecutan en el contexto de la misma tarea *daemon* del RTOS. Esta tarea se crea automáticamente al lanzar el *scheduler*, por eso se le denomina *daemon*, haciendo referencia a un proceso que permanece de forma continua a lo largo de toda la ejecución del programa.

Esto es importante, ya que entonces no se debe utilizar APIs dentro de las funciones de *callback* que puedan generar un cambio de contexto, es decir, que bloqueen el RTOS *daemon*. Por ejemplo, se puede llamar *xQueueReceive()* pero solo si el parámetro *xTicksToWait* esta seteado en 0.

En el proyecto hay tres instancias de timers por software, y son los encargados del movimiento de los motores paso a paso (ver figura 5). Es importante aclarar que el uso de un *timer* por software para el control de un motor paso a paso dista mucho de ser la solución más óptima. Se los utiliza en este trabajo solo como ejercicio para poner en práctica su uso.

VII-G. Uso de FreeRTOS desde una rutina de interrupción

Dado que muchas veces es necesario el uso de funcionalidades del RTOS dentro de una rutina de interrupción, y teniendo en cuenta que al ejecutarse una rutina de interrupción estamos saliendo del control del sistema operativo en tiempo real, FreeRTOS provee una versión de las APIs convencionales pero para ser ejecutadas en estas circunstancias. Todas aquellas APIs terminadas en "FromISR" son de este tipo.

Por este mismo motivo es muy aconsejable que las rutinas de interrupción sean cortas, y que si es necesario realizar un procesamiento considerable, dirigirlo a una tarea para que lo realice y finalizar la rutina de interrupción. Una rutina de interrupción larga **agregará jitter a la aplicación**.

Para sincronizar las interrupciones con las tareas que ejecutarán el procesamiento vinculado a ellas, se utilizan objetos como semáforos y colas.

En el proyecto se utilizaron APIs de FreeRTOS dentro de rutinas de interrupción como las de UART o las de entradas utilizadas por el encoder (ver figuras 4 y 6).

Otra forma de mantener las rutinas de interrupción cortas es difererir el procesamiento a la tarea *daemon* del RTOS. Esto evita la necesidad de crear una tarea solo con este prósito. Este método consume menos recursos, es más simple, pero hay que tener en cuenta que es menos flexible. La tarea *daemon* ya tiene una prioridad definida.

En este proyecto también se utilizó éste método, al diferir el procesamiento relacionado al pulsador del encoder (lectura de mailbox, escritura de mailbox y actualización de display), ver figura 6. Es importante el hecho de que se ejecuta en el mismo contexto que las funciones de *callback* de los timers y tienen la misma prioridad. Por lo tanto, se le está asignando igual prioridad a los motores paso a paso que a la selección de motor en la interfaz.

VII-H. Semáforos

Los semáforos son objetos de comunicación que permiten la sincronización entre interrupciones y tareas, aunque su uso no se restringe solamente a éste. Su funcionamiento consiste en bloquear a la tarea hasta que desde la interrupción se realice la acción de "dar" el semáforo y la tarea pueda "tomarlo".

La descripción anterior es la de un semáforo binario, pero también existen los semáforos contadores que son equivalentes a colas con una longitud mayor a uno. La tarea que lee dicho semáforo no esta interesada en la información que contiene, sino en la cantidad de elementos. Cada vez que se "da" al semáforo se agrega un elemento, y cada vez que se "toma" del semáforo se elimina un elemento.

En el proyecto se utilizó dos semáforos contadores en la implementación del encoder, donde ambos acumulan los pulsos que se reciben (ver figura 6).

VII-I. Gestión de recursos y mutex

Un problema que hay que tener en cuenta en sistemas multitarea como el de este proyecto, es la posibilidad de que una tarea acceda a un recurso compartido y sea expulsada del estado *Running* antes de completar ese acceso. Si esto ocurre, mientras la tarea se encuentra en este estado, otra tarea que comparta el mismo recurso puede acceder y modificarlo, resultando en una corrupción de datos.

Una forma de mantener y asegurar la consistencia de datos es a través de la técnica de **exclusión mutua**. El objetivo es que una vez que una tarea accede a un recurso compartido, ninguna otra puede acceder hasta que ese recurso haya vuelto a un estado de consistencia, o lo que es lo mismo, esté disponible para volver a ser accedido. De esta forma, las tareas tienen acceso exclusivo soble dicho recurso.

El objeto que permite poner en práctica la técnica de exclusión mutua es el *mutex*. Básicamente consiste en un tipo de semáforo binario para controlar el acceso a un recurso. Al acceder a un recurso compartido la tarea tomará el *mutex*, y ningun otro proceso podra acceder hasta que ese *mutex* no sea "devuelto".

Algo a tener en cuenta al utilizar este recurso, es el fenómeno de **inversión de prioridad**. La inversión de prioridad sucede cuando una tarea de baja prioridad toma el *mutex* y luego otra de alta prioridad intenta acceder al mismo recurso y, por lo tanto, se bloquea esperando su liberación. Si antes de que la tarea de baja prioridad libere el *mutex*, una tercera tarea de prioridad media (prioridad mayor que la de baja prioridad pero menor que la de alta prioridad) hace un *pre-empt*, el resultado es que la tarea de prioridad alta no solo tiene que esperar a la tarea de prioridad baja, sino también a la de prioridad media. Para disminuir el impacto de este fenómeno, los *mutex* se diferencian de los semáforos convencionales en su característica de **prioridad heredada**. Básicamente se aumenta temporalmente la prioridad de la tarea que toma el *mutex*, y de esta forma las tareas como la de prioridad media en el ejemplo anterior no realizan el *pre-empt* sobre la tarea que tomó dicho *mutex*.

Otra forma muy efectiva de garantizar la exclusión mutua de un recurso, son las tareas *gatekeeper*. Basicamente consiste en una tarea convencional que tiene el control total sobre el recurso a proteger, cualquier otro proceso que necesite el acceso lo deberá hacer indirectamente a través de los servicios que brinde la tarea *gatekeeper*.

En el proyecto se utilizaron dos de este tipo de tareas a la hora de implementar la comunicación UART: una encargada de la transmisión de datos y otra del procesamiento de la información recibida.

VII-J. Grupos de eventos

A diferencia de otros objetos como las colas y semáforos, los grupos de eventos permiten a una tarea esperar bloqueada por una combinación de uno o más eventos y también desbloquean todas las tareas que esten esperando una misma o diferentes combinaciones.

Los grupos de eventos consisten basicamente en banderas que se setean en 1 o 0 según si un evento ocurrió o no. Dichas banderas se agrupan en una variable de 24 bits.

En la aplicación se utilizaron en el módulo de control de motores paso a paso. Los *timers* por software asociados a cado motor al finalizar su funcionamiento (es decir al llegar a la consigna el motor) setean un determinado bit en el grupo de eventos. Al finalizar todos los motores, se produce la combinación que desbloquea la tarea encargada del control de dichos timers permitiendo que se ejecute la siguiente consigna en la cola.

VII-K. Notificación de tareas

Las notificaciones de tarea proporcionan una forma de comunicación entre tareas (o tareas y rutinas de interrupción) sin la necesidad de un objeto de comunicación, como lo son las colas, semáforos y grupos de eventos. Esto permite que la comunicación sea directa.

Algunas limitaciones de las notificaciones de tareas son:

- Solo se puede notificar a una única tarea.
- No se puede acumular múltiples elementos con información.
- Bloquear un proceso hasta que el envío sea válido. Si se envía una notificación a una tarea que ya tenía una notificación pendiente, no se puede esperar en estado bloqueado a que la tarea resetee su estado de notificación.

Un ejemplo de notificaciones implementadas en el proyecto, es la gestión de errores por notificación en la tarea de sincronización de mensajes. Allí cada bit del valor de notificación da la información de cuál fue el error en el mensaje recibido (ver figura 10).

VIII. Implementación de los diferentes módulos del proyecto

A continuación se describe, mediante diagramas de flujo, el diseño e implementación de los diferentes módulos que conforman la aplicación.

VIII-A. Comunicación UART

Las colas permiten una forma sencilla de comunicación de interrupción a tarea, sin embargo hay que tener en cuenta que no es eficiente si la entrada de datos es a una frecuencia muy alta. En este proyecto se aplicó solo con fines de demostración, opciones más eficientes pueden ser el uso de DMA (*Direct Memory Access*) o procesar todo dentro de la rutina de interrupción.

En la figura 4 puede observarse tanto el diagrama de transmisión como de recepción de mensajes por UART.

La transmisión es muy sencilla: consiste en una tarea que realiza la lectura de una cola de mensajes pendientes a enviar, para luego efectivamente enviarlos por UART.

Para la recepción de información se utilizan dos colas. La primera es escrita por la rutina de interrupción, y consiste en los caracteres recibidos. Dichos caracteres son leídos por la tarea encargada de ese procesamiento, y se construye como único string que finaliza cuando se lee un final de línea para ser enviado a una segunda cola de mensajes recibidos.

VIII-B. Control de motores paso a paso

En la figura 5 puede observarse el diagrama de la implementación del control de motores paso a paso mediante el RTOS.

El primer componente del módulo es una cola de los mensajes recibidos vinculados a la actividad de los motores. El tipo de elementos que guarda la cola son punteros a los mensajes (es un ejemplo de la utilización de colas para la comunicación de información de distintas dimensiones).

Una tarea es la encargada de leer la información en la cola de consignas, que es escrita mediante acciones externas al módulo, y procesar el mensaje identificando los parámetros a setear: ID del motor, velocidad, posición y/o dirección. Una vez realizado ese procesamiento del mensaje, la tarea ejecuta una de las siguientes acciones:

- Se devolverá un error si el mensaje es inválido, mediante el uso de notificación de tareas, seteando los bits correspondientes al error que se encontró.
- Se cambiará el período del timer indicado, para afectar la velocidad del motor.

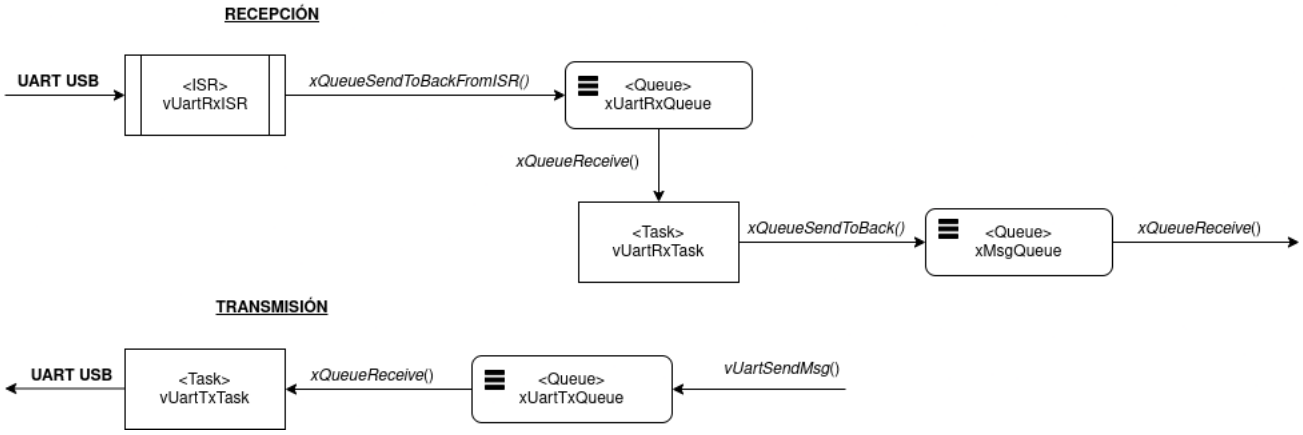

Figura 4. Implementación de comunicación UART con colas.

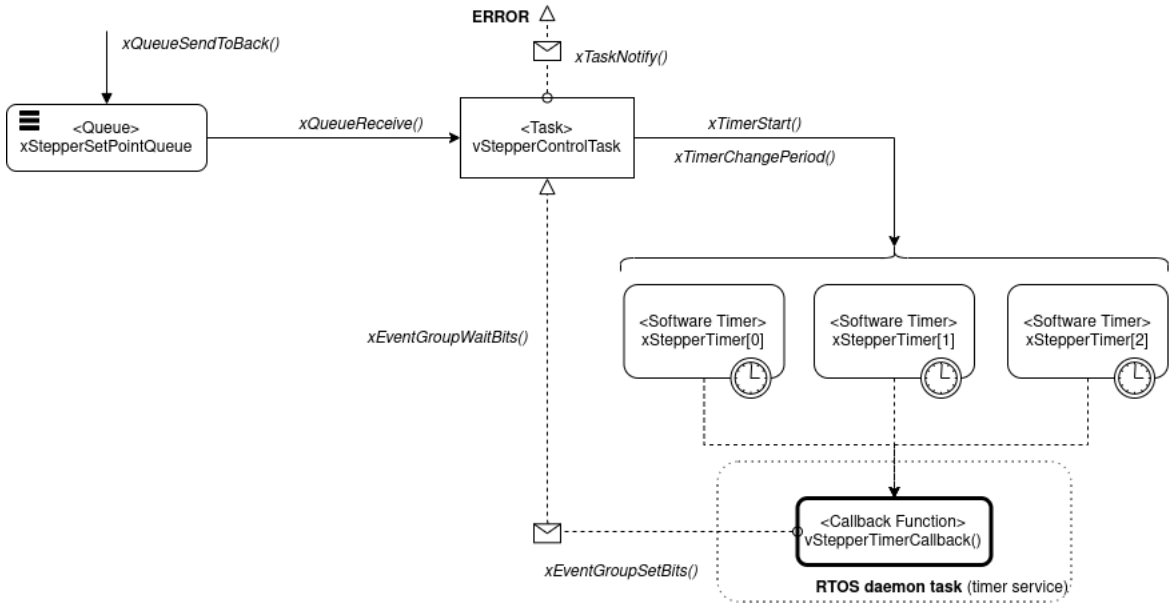

Figura 5. Implementación de control de motores paso a paso.

- Se inicializa el timer indicado, con la información obtenida actualizada en su ID.

Probablemente los componentes más importantes del módulo son los timers por software. Uno por cada motor (por lo tanto, tres) y en su ID se encuentra un puntero a la estructura que contiene la información del motor: pasos pendientes, dirección, identificación y estado del driver. Cada vez que los timers pasen a estado *Running*, se ejecuta la función de *callback* correspondiente. que decrementa la cantidad de pasos pendientes y actualiza el driver para afectar la posición del motor.

Si la cantidad de pasos pendientes llega a cero, el timer se detiene a sí mismo y luego setea su bit correspondiente en el grupo de eventos. Una vez que finalizan todos los timers, se produce el evento que permite a la tarea de control de los motores que pueda procesar la siguiente consigna en la cola y ejecutarla si es correcta.

Es importante aclarar que en el diseño planteado en las consignas puede setearse los tres motores paso a paso a la vez o motores individuales.

Como se mencionó en la sección VII-F, la función de *callback* de los timers se ejecuta en el contexto de la tarea *daemon* del RTOS. Entonces, al *daemon* se le asignó la prioridad más alta de la aplicación, ya que es imprescindible que los tiempos del timer se cumplan para un correcto funcionamiento de los motores.

VIII-C. Encoder rotativo incremental

En la figura 6 se puede observar el diseño del módulo encargado de procesar la información recibida del encoder rotativo.

El rol del encoder en la aplicación es proveer una interfaz HMI muy simple que le permita al usuario seleccionar el motor a mover y moverlo. Para su lectura se deben manejar 3 entradas:

- Una entrada de pulsador, que le permitirá al usuario cambiar de motor a mover.

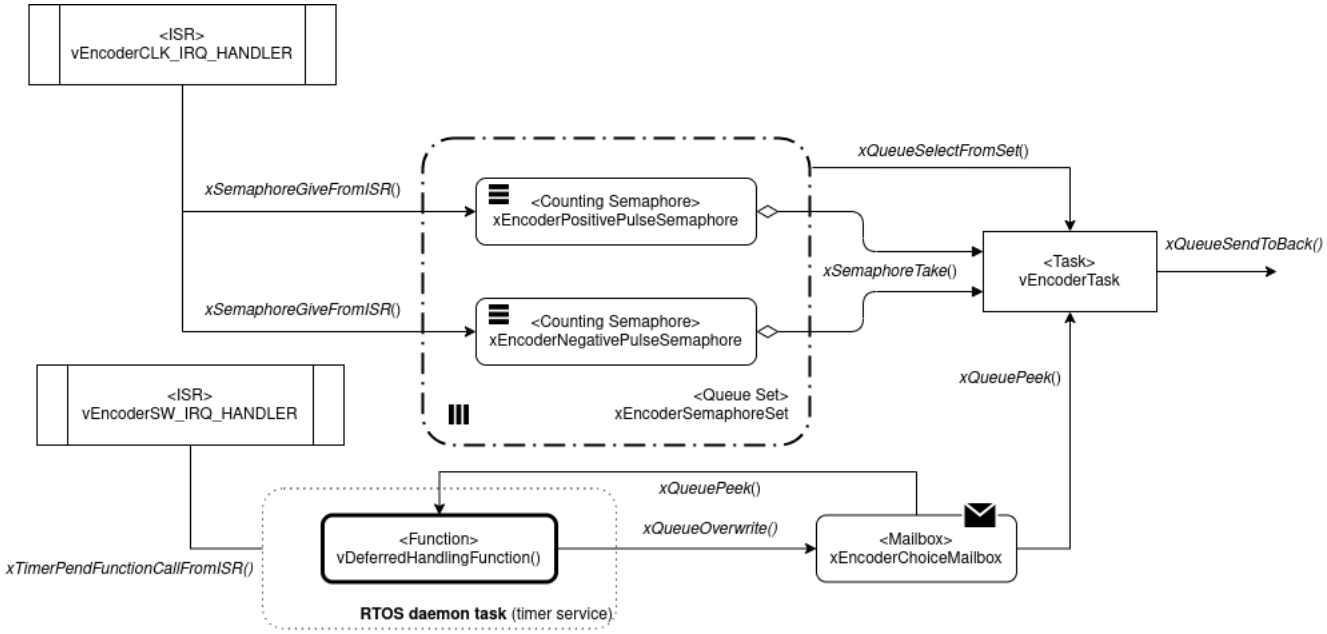

Figura 6. Implementación de encoder rotativo como interfaz de usuario.

- Una entrada de clock, que da los pulsos del movimiento de rotación del encoder. Cada pulso es equivalente a una consigna de un determinado ángulo.
- Una entrada de dirección, que al leerse junto con el pulso de clock recibido nos permite saber la dirección de rotación del encoder.

Entonces, el sistema recibe la interrupción tanto del pulsador como la señal de clock.

Ante el evento del pulsador, la rutina de interrupción difiere el procesamiento a la tarea *daemon* del RTOS. La función ejecutada en el *daemon* lee e incrementa el valor de un *mailbox* (sumado a una actualización del display). Este *mailbox* guarda el motor elegido por el usuario y es a la vez el que visualiza en el display.

Ante el evento del pulso de clock, se lee la entrada de dirección del encoder y según sea su valor 1 o 0 se agregara un ítem al semáforo contador de pulsos positivos o al semáforo contador de pulsos negativos respectivamente.

Una tarea es la encargada de procesar la información recibida. Dado que ambos semáforos contadores pertenecen a un mismo *set*, la tarea se bloquea esperando información en cualquiera de los dos semáforos. Al recibir un elemento en alguno de ellos, se construye el mensaje de consigna leyendo el motor seleccionado desde el mailbos, y se envía como un arreglo de caracteres a la cola de consignas pendientes.

Si el motor seleccionado es el servomotor del extremo operativo del robot, antes de enviar la consigna se lee el mailbox donde se encuentra la posición actual del motor, para poder incrementar o decrementar dicho valor (esto porque la consigna al servomotor es de posición absoluta).

VIII-D. Servo motor en extremo operativo

Para el control del servomotor es necesario una señal PWM con un período de 20 ms (señal de 50 Hz) que tenga un *duty-cycle* variable entre 1 y 2 ms, siendo 1 ms para un ángulo en el motor de 0°, y 2 ms para un ángulo de 180°. Dicha señal PWM fue implementada con un periférico del microcontrolador y no con APIs de FreeRTOS. El periférico utilizado es el *SCT Timer* o *State Configurable Timer* que fue configurado hanciendo uso de la librería LPCOpen.

La resolución es similar a la de los motores paso a paso, sin el uso de timers por software: Se posee una cola de mensajes con las consignas al motor, que es leída por una tarea. La tarea se encuentrada bloqueada hasta recibir una consigna, y el entrar en estado *Running* ejecuta una función que varía el *duty-cycle* de la señal PWM.

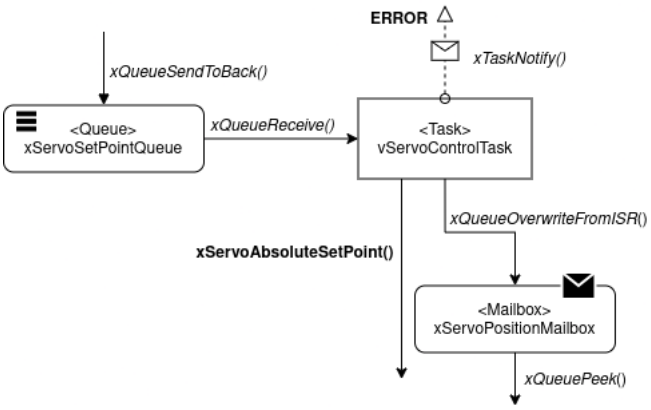

Figura 7. Implementación de control de servomotor.

En la figura 7 se puede observar el diseño del módulo encargado del control del servomotor.

Para verificar que la señal PWM enviada al motor es correcta, se utilizó un a analizador lógico. Las lecturas realizadas se pueden observar en las figuras 8 y 9.

Por último, para poder llevar un seguimiento de la posición del motor, se implementó un *mailbox* que contiene el valor del

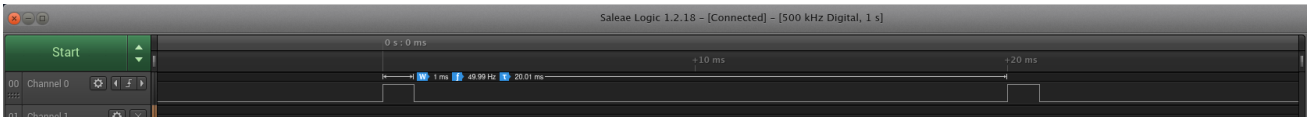

Figura 8. Lectura con analizador lógico de señal PWM correspondiente a 0°del servomotor.

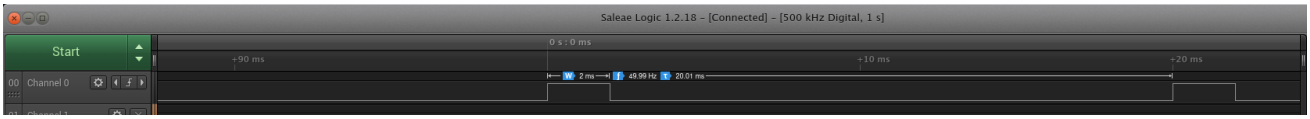

Figura 9. Lectura con analizador lógico de señal PWM correspondiente a 180°del servomotor.

ángulo, y puede ser leído, por ejemplo, por el encoder para realizar incrementos o decrementos.

VIII-E. Display LCD de dos líneas

En el proyecto se utilizó un display LCD como salida HMI. Para la comunicación entre el display y el microcontrolador se utilizaron las funciones proveídas por la sAPI del firmware del proyecto CIAA.

El display muestra durante toda la ejecución en su primer línea el título del proyecto y la cátedra. En su segunda línea, una etiqueta correspondiente al motor seleccionado seguido de los pasos pendientes si la selección es de un motor paso a paso, o la posición actual del servomotor.

Ante el evento del pulsador del encoder, se difiere el procesamiento al *daemon* (ver sección VIII-C), y desde allí se actualiza la selección en el display.

Para el caso de los motores paso a paso, se implementó una tarea (de muy baja prioridad) que realiza un polling cada 500 ms del ID del timer correspondiente al motor seleccionado. De allí se obtiene la cantidad de pasos pendientes del motor.

Por último, si lo seleccionado es el servomotor, el polling cada 500 ms es al *mailbox* que contiene su posición.

La interacción del módulo con el resto de la aplicación puede observarse en la figura 10.

IX. Interrupciones anidadas y particularidades de procesadores ARM Cortex-M

El desafío más complejo en el desarrollo de la aplicación fue la implementación para que funcionen correctamente las rutinas de interrupción ejecutadas por el encoder rotativo (rutinas de interrupción ante flancos en pines GPIO).

Como ya se mencionó en la sección VIII-C, para la implementación del encoder además de las APIs de FreeRTOS se utilizó la librería LPCOpen, que brinda una capa de abstracción de más bajo nivel que la de la librería sAPI. Que sea de más bajo nivel da espacio a una mayor libertad en la configuración de las interrupciones, algo necesario para hacer la interrupción compatible con el RTOS.

Al plantear una configuración inicial de estas interrupciones con la librería LPCOpen, las rutinas de interrupción se ejecutaban correctamente a la hora de hacer parpadear un LED o imprimir un mensaje por UART, sin embargo a la hora de llamar una API de FreeRTOS cualquiera (terminada en “FromISR”) el sistema fallaba dejando de funcionar.

En su libro, Barry [1] menciona que es común confundir la prioridad de tareas en el FreeRTOS con la prioridad de interrupciones en el microcontrolador. También se aclara que es importante considerar que la relación entre el valor numérico de la prioridad y su valor lógico depende de la arquitectura del procesador.

Es importante, entonces, configurar la variable *configMAX_SYSCALL_INTERRUPT_PRIORITY* en aquellas plataformas que admitan el anidado de interrupciones, que es el caso del microcontrolador NXP LPC4337 en la placa EDU-CIAA. Dicha variable da la máxima prioridad **de interrupción** desde la cuál se puede llamar APIs de FreeRTOS.

Un modelo completo de anidado de interrupciones (donde todas las interrupciones pueden llamar APIs de FreeRTOS) se da cuando el valor de *configMAX_SYSCALL_INTERRUPT_PRIORITY* es seteada a un **valor lógico** mayor que *configKERNEL_INTERRUPT_PRIORITY*. Esta última variable, da el valor de prioridad de la interrupción por tick y, por lo tanto, debe setearse al mínimo posible.

En los procesadores ARM Cortex-M, valores numéricos bajos de prioridad equivalen a valores lógicos altos de prioridad, es decir, una interrupción con valor 1 tiene mayor prioridad que otra con valor 5. Esto se presta especialmente a confusión, ya que en FreeRTOS la relación es la opuesta.

El controlador de interrupción de Cortex-M permite especificar la prioridad con un máximo de 8 bits, es decir, la prioridad de interrupción más baja posible será con valor numérico 255. Generalmente solo se implementa un subgrupo de esos 8 bits. De la librería CMSIS del microcontrolador LPC4337 se obtuvo que ese subgrupo es de 2 bits.

Con esta información se solucionó el problema seteando, mediante un llamado de LPCOpen, la prioridad de NVIC de las interrupciones al mínimo (en el proyecto se las configuró en 255).

X. Vinculación de los diferentes módulos de la aplicación

Una vez desarrollados los distintos módulos explicados en detalle en la sección VIII, se procedió a implementar su comunicación para poder cumplir el objetivo de mover el brazo robótico mediante consignas (ya sea por UART o a través del encoder rotativo).

Para esto, se implemento una última tarea que es la encargada de procesar los mensajes recibidos, enviarlos a los módulos

que correspondan, y a su vez recibir los errores que estos mensajes puedan tener para poder comunicarlos por UART.

En la figura 10 se puede observar la interconexión de todas las partes que conforman el proyecto.

En la figura 11 se puede observar la conexión física de los distintos componentes utilizados a la placa de desarrollo EDU-CIAA-NXP.

XI. Descripción de formato de consigna a motores

En la tabla puede observarse el formato de las consignas a enviar al microcontrolador a través de UART.

Motor PaP	
Posición	:S{Id1}D{Dirección1}A{Ángulo 1 con tres dígitos} S{Id2}D{Dirección2}A{Ángulo 2 con tres dígitos} S{Id3}D{Dirección3}A{Ángulo 3 con tres dígitos}
Velocidad	:S{Id}V{Período en ms}
Servomotor	
Posición	:X{Ángulo}

El módulo del servomotor recibe consignas para ángulos entre 0°y 180°. Valores fuera de este rango devolverán un error.

XII. Debug para control de errores con assert y funciones de memoria

A la hora de implementar en conjunto el encoder rotativo, el control de los motores paso a paso y el control del motor servo, se presentaron problemas de memoria.

Para poder realizar una especie de *debugging* sencillo, se utiliza la macro de C *assert*, que permite verificar si una expresión es válida (si un puntero es nulo por ejemplo). FreeRTOS no llama la función estándar de C *assert* por no estar disponible en todos los compiladores, en su lugar llama a *configASSERT* que se puede definir en el archivo de configuración de FreeRTOS.

Una forma de obtener información de dónde se producen errores es, por ejemplo, imprimir por salida estándar el archivo y número de línea donde se produjo el error. Esto se logra definiendo *configASSERT* como una función que realice esto. No fue necesario la implementación de dicha función ya que se encontraba en el código fuente de FreeRTOS (específicamente en el archivo *hooks.c*).

Además, también se utilizo la API de FreeRTOS *xPortGetFreeHeapSize*, que permite obtener el espacio libre disponible en pila de memoria.

Se pudo encontrar que al juntar todos lo módulos, se sobrepasaba el tamaño de pila dado en la configuración (de 8 kilobytes). Se duplicó a **16 kilobytes** y así contar con espacio suficiente para toda la aplicación.

Analizando este problema también se obtuvo el tamaño de cada módulo, se pueden observar en el cuadro I.

Módulo	**Tamaño [Bytes]**
UART	2646
Motor PaP	1200
Servomotor	1112
Display	896
Encoder	1160
Disponible	*7864*

Cuadro I
Tamaño de los diferentes módulos en su inicialización.

XIII. Conclusiones

Todo el proyecto se desarrolló siguiendo el estilo de código propuesto por FreeRTOS, esto permitió programar de forma mucho más eficiente, estructurada y limpia. Además, para la documentación del código se siguieron los lineamientos del proyecto Doxygen, esto también hizo que el desarrollo del proyecto a medida que se iba complejizando fuera más sencillo de seguir.

Durante el desarrollo se cometió el error de proteger una cola con un mutex, cuando en realidad todas las APIs de FreeRTOS son *thread-safe* por lo que no es necesario.

Uno de los puntos a resaltar al finalizar el trabajo, es que aunque los diagramas parezcan complejos, el desarrollo de la aplicación no presenta dificultades si el diseño de la comunicación entre tareas esta bien planteada. Es muy sencillo seguir un diagrama e implementar todas los módulos con las APIs que provee el RTOS, trabajando sobre los procesos de forma independiente y llevando a cabo su intercomunicación hacia el final. La misma aplicación siguiendo un esquema de *superloop* sería mucho más compleja de diseñar e implementar.

XIV. Trabajo a futuro

Analizar distintas herramientas de debugging para RTOS, como también estudiar en profundidad el desempeño que tiene el diseño implementado (tiempo en IDLE, cantidad de stack asignada a las diferentes tareas, etc.).

Fabricación de PCB donde poder montar todas las conexiones del hardware de forma más prolija (actualmente se encuentra en placa de prototipado).

Mejorar el tipo de mensajes que se reciben por UART al igual que agregar más para una mayor interacción con el proyecto.

Referencias

[1] Richard Barry. *Mastering the FreeRTOS Real Timer Kernel: A Hands-On Tutorial Guide*. 2016.

[2] Proyecto CIAA. *Framework para desarrollo de Firmware de Sistemas Embebidos en C/C++*. URL: https://github.com/epernia/firmware_v3/blob/master/documentation/firmware/readme/readme-es.md.

[3] MISRA. *MISRA-C:2004 Guidelines for the use of the C language in critical systems*. MIRA Limited, 2004. ISBN: 9780952415640.

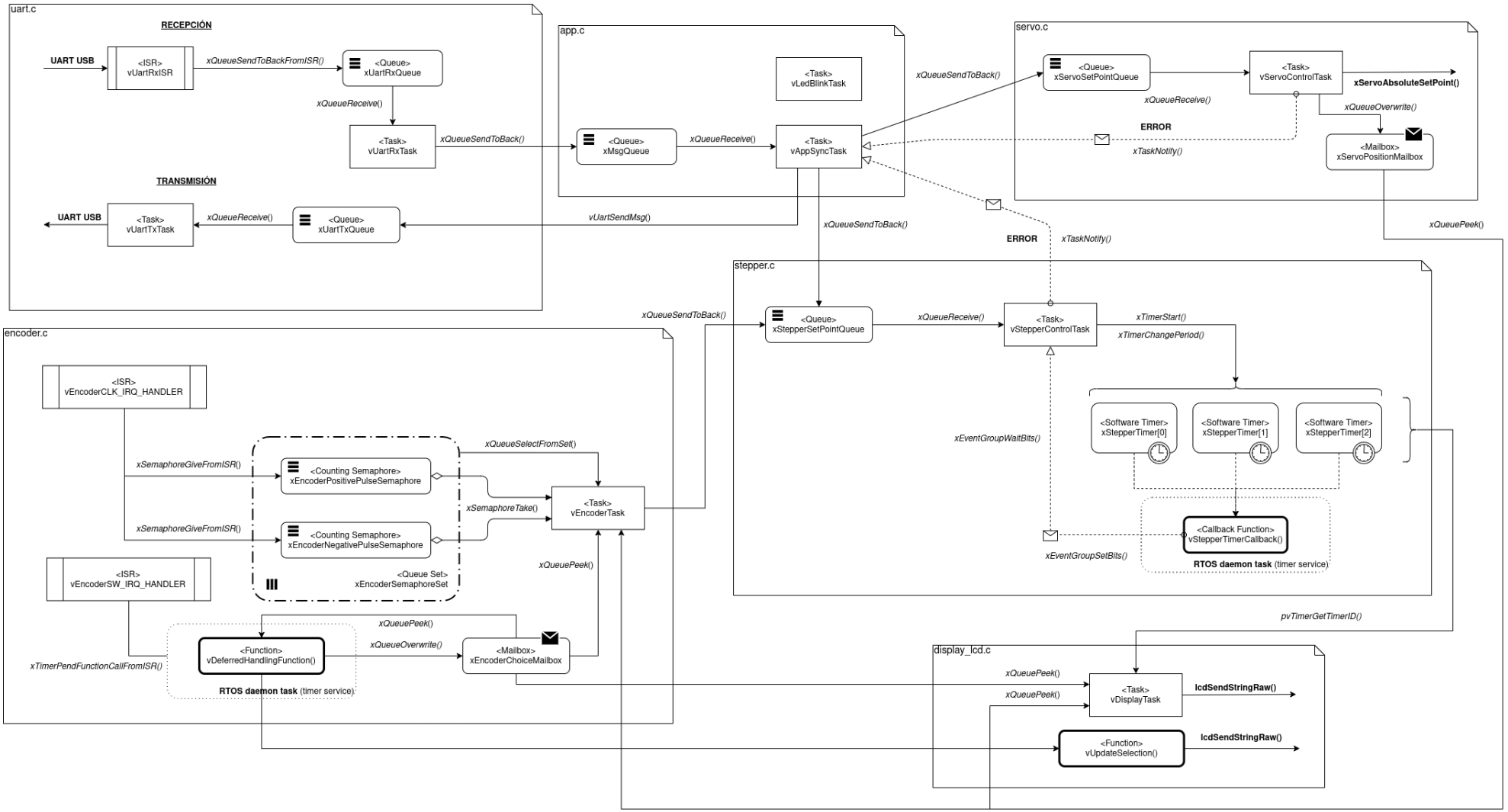

Figura 10. Diagrama de interconexión de los diferentes módulos de la aplicación.

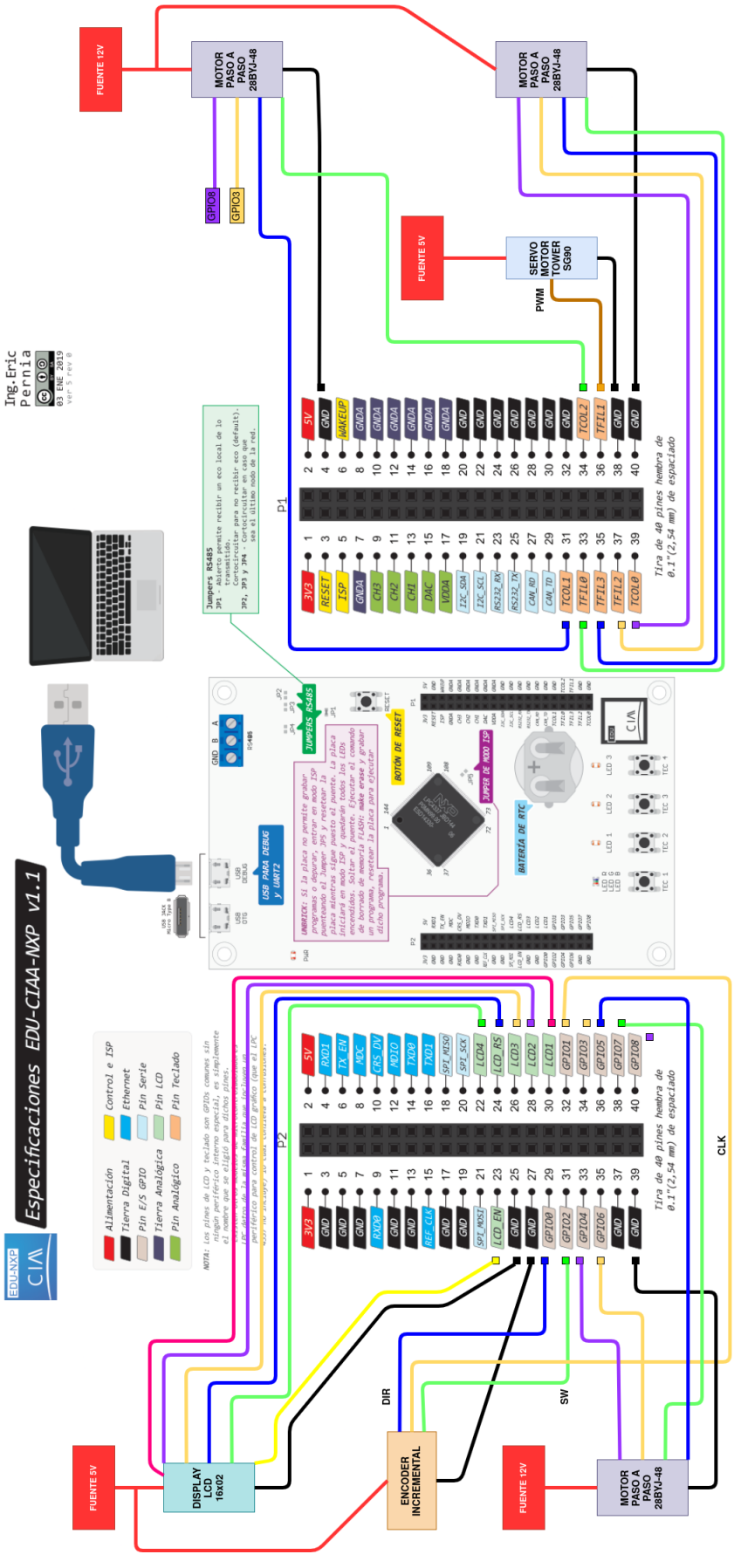

Figura 11. Diagrama de conexión física de los diferentes componentes a la placa de desarrollo EDU-CIAA.