Programación

PID_00247324

Màrius Montón Macián

Tiempo mínimo de dedicación recomendado: 3 horas

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Introducción 5

1. Particularidades de la programación para sistemas empotrados 6

2. C y C++ 8

3. Compiladores cruzados y flujo de trabajo 9

4. Protección de memoria 12

5. Organización de código 13

6. Normas de codificación 15
- 6.1. Normativa MISRA 15
- 6.2. Otras normativas 16

7. Multitarea 18
- 7.1. Tareas 19
- 7.2. Comunicación entre tareas 20

8. *Drivers* para periféricos 21
- 8.1. *Drivers* de bajo nivel 21
- 8.2. Librerías de alto nivel 22

9. Programación para bajo consumo 24

10. Librerías habituales 27
- 10.1. Lightweight IP 27
- 10.2. FatFS 27
- 10.3. Librerías gráficas 28
- 10.4. CMSIS 30
  - 10.4.1. CMSIS-CORE 30
  - 10.4.2. CMSIS-Driver 30
  - 10.4.3. CMSIS-DSP 30
  - 10.4.4. CMSIS-RTOS 31
  - 10.4.5. CMSIS-Pack 31
  - 10.4.6. CMSIS-SVD 31
  - 10.4.7. CMSIS-DAP 31

Bibliografía 33

Introducción

En este contenido se introducirán las particularidades y buenas prácticas de la programación para sistemas empotrados. Aquí se incluye una descripción de las particularidades de este tipo de sistemas, y cómo esto afecta al estilo de programación.

Luego se comentarán las herramientas de compilación necesaria y cómo se manejan para generar ejecutables para el o los microcontroladores del sistema.

A continuación, se discutirán qué particularidades especiales existen en el ámbito de la gestión de la memoria en los microcontroladores y cómo debe programarse a tal efecto.

En el siguiente apartado se introducen aspectos relacionados con los SO, como son la multitarea y el uso de *drivers*. Aunque ya se han introducido estos aspectos en apartados anteriores, aquí se hará un análisis práctico con ejemplos sencillos.

A continuación se presentarán los aspectos que cabe tener en cuenta a la hora de programar para obtener un bajo consumo del sistema, tanto aspectos de SO como de programación.

Por último, se presentan las librerías más comunes para tareas complejas, como son la implementación de un *stack* TCP/IP o USB.

1. Particularidades de la programación para sistemas empotrados

A diferencia de lo que ocurre en sistemas más potentes, en un sistema empotrado los recursos son escasos o muy limitados, como ya se ha visto. Hay que recordar que la cantidad de memoria RAM disponible será del orden de decenas de kilobytes y la cantidad de memoria ROM (o FLASH) para almacenar código ejecutable será del orden de centenares de kilobytes (actualmente un tamaño máximo típico es 1024 kilobytes).

También habrá que tener en consideración que la capacidad de cómputo de un microcontrolador no es equiparable a un microprocesador moderno, tanto por la velocidad de su reloj (pocos MHz frente a pocos GHz), como por la arquitectura y los recursos de cómputo disponibles.

También hay que tener en cuenta que o bien no se usa sistema operativo, o bien el que se usa no tiene las características habituales de un sistema operativo completo de ordenador personal.

La primera particularidad y más obvia es que en el caso del desarrollo para sistemas empotrados se diseña y programa el código teniendo muy en cuenta el HW sobre el que se estará ejecutando. Esto hace que, por ejemplo, no sea extraño trabajar a nivel de bit para escribir un *flag* concreto en un registro de un periférico.

Otra particularidad de trabajar con sistemas empotrados es la gestión de la memoria. En términos generales, no se recomienda el uso de memoria dinámica en entornos empotrados por varias razones:

- Fragmentación: pedir memoria y liberarla provocará problemas de fragmentación de esta, haciendo que en algún caso, habiendo suficiente memoria libre disponible, esta no sea adyacente y no se pueda reutilizar de forma óptima.

- Velocidad: la gestión de la memoria dinámica puede ralentizar un sistema debido a la complejidad de las funciones de pedir y liberar memoria.

- No requerida: usar memoria dinámica permite reutilizar la misma memoria para distintas tareas en momentos distintos. Normalmente este no es el caso en sistemas empotrados, ya que en estos sistemas se tiende a realizar siempre las mismas tareas todo el tiempo.

- Tiempo de respuesta: si el sistema es multitarea y, por tanto, las funciones de gestión de la memoria dinámica usan primitivas de exclusión mutua, esto puede suponer que el tiempo de respuesta a interrupciones se vea aumentado.

- Dificultad al *debug*: los errores provocados por una mala codificación de la gestión de memoria son muy difíciles de detectar y de *debuggar*.

- Gestión de errores: ¿cómo se gestiona que una petición de memoria dinámica falle porque no hay más memoria?

2. C y C++

Aunque hay otras alternativas, el lenguaje mayormente utilizado para el desarrollo de código para sistemas empotrados es el lenguaje C. A pesar de que históricamente se ha usado ampliamente código ensamblador, la buena evolución de los compiladores de C para sistemas simples y la progresiva complejidad de los microprocesadores ha provocado que actualmente casi todo el código se escriba en C.

Así pues, en sistemas empotrados y usando lenguaje C, no es nada recomendable ni habitual usar los comandos *malloc* y *free* para la gestión de la memoria de forma dinámica. Por tanto, se usan siempre definiciones estáticas para variables, *arrays*, *buffers*, etc.

Malloc* y *free

Estas dos funciones de C reservan memoria de forma dinámica (*malloc*) o la liberan (*free*). Se usan cuando, *a priori*, no se sabe cuánto espacio hace falta y debe hacerse de manera dinámica.

Una discusión que sigue abierta en este tema es el uso o no de C++ para sistemas empotrados. C++ es el lenguaje derivado de C con orientación a objetos (OO).

Debido al uso intensivo de memoria dinámica en programación OO, la controversia de usar o no este tipo de programación sigue abierta. Para sus defensores, la OO permite una mejor codificación, código más ordenado y mantenible, mejor estructurado y con más facilidades para el desarrollador.

Por contra, los detractores opinan que evitar el uso de memoria dinámica en C++ provoca que muchas de las ventajas que tiene la programación orientada a objetos no puedan aprovecharse. También existen cuestiones de cómo de optimizados son los compiladores para C++ respecto a los compiladores de C y de la cantidad de memoria RAM que necesita un programa en C respecto a la misma funcionalidad en C++.

Con todo esto, hay desarrolladores de software para sistemas empotrados que están usando C++ en lugar de C, mientras que hay otros que siguen aferrados al C sin contemplar, por ahora, el salto a C++.

3. Compiladores cruzados y flujo de trabajo

Como todo proceso de desarrollo de software, se usan compiladores para trasladar (compilar) el texto escrito por el desarrollador a lenguaje máquina ejecutable por el (micro)procesador.

Hay compiladores de distintos fabricantes o proyectos para distintos procesadores o arquitecturas. Así, se pueden encontrar compiladores para la familia AVR de Atmel de 8 bits, otros para la misma familia de 32 bits, otro compilador para la familia Cortex-M3 de ARM y compatible con todos (o la mayoría de) los microcontroladores de todos los fabricantes, etc.

Estos compiladores reciben el código fuente del desarrollador, las librerías que se estén usando y la información dada por los fabricantes para generar un fichero binario ejecutable.

Este compilador se puede ejecutar en el ordenador y el fichero binario debe trasladarse a la memoria de programa del microcontrolador para su ejecución. Esta tarea se realiza por unos dispositivos especiales, llamados programadores o *debuggers*, que, mediante un puerto especial del microcontrolador, son capaces de, entre otras cosas, escribir directamente en la memoria de programa de este.

Una vez finalizada esta tarea, se reinicia el microcontrolador para que empiece la ejecución del nuevo programa.

Entonces, el flujo de trabajo habitual para la programación de sistemas empotrados es el siguiente:

1) Editor de texto para la edición del código fuente del desarrollador.

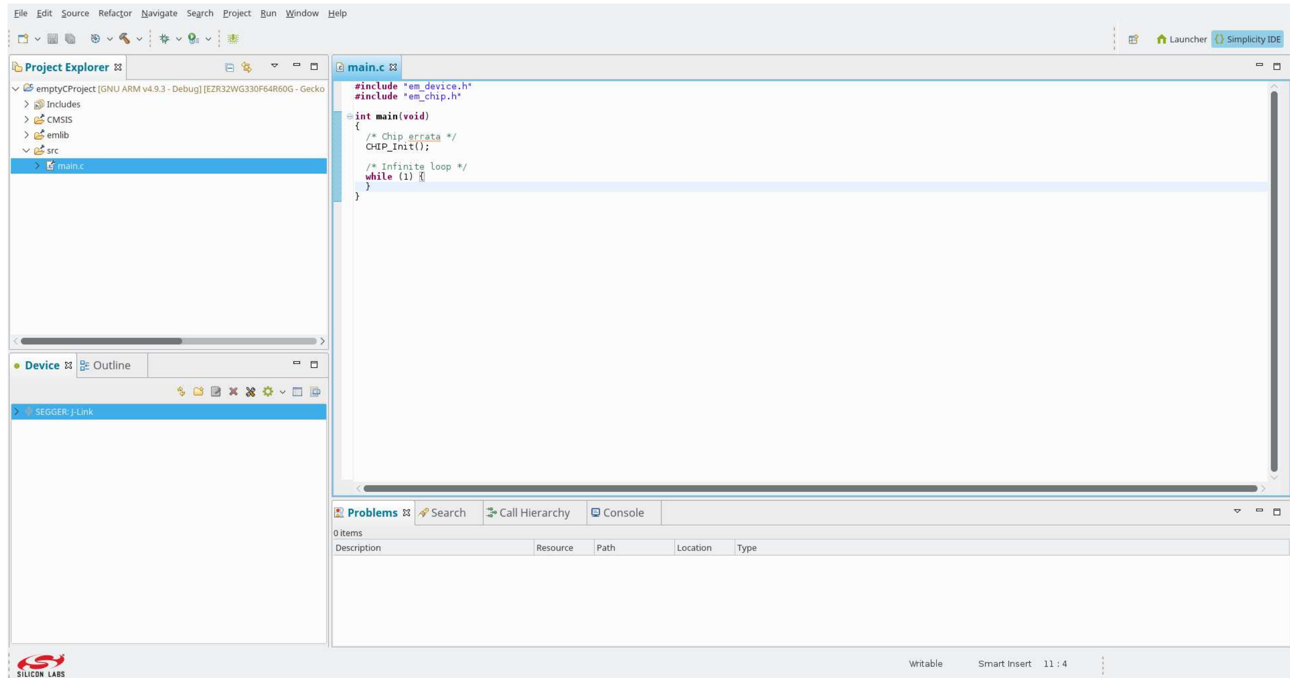
2) El compilador cruzado compila todo el código fuente y las librerías necesarias y genera un solo fichero binario compatible con el microcontrolador de destino.

3) El dispositivo programador junto con su software de programación graban la memoria de programa del microcontrolador con el contenido del fichero binario previamente generado.

4) El dispositivo programador reinicia el microcontrolador para que este empiece la ejecución del nuevo código.

Este proceso puede verse escondido por ciertas herramientas integradas, que incorporan todos estos pasos de forma transparente o sencilla para el desarrollador, de manera que con una sola orden se ejecuten, por ejemplo, los pasos 2, 3 y 4 (véase figura 1).

Figura 1. Captura de pantalla de Simplicity IDE de Silicon Labs



Elaboración propia.

Después de haber programado el microcontrolador, es habitual poder empezar también una sesión de *debug* en lugar de una sesión de ejecución libre. En este caso, una vez programado el microcontrolador, este se reinicia y se procede a controlar la ejecución, de manera que el desarrollador puede controlar a qué ritmo o qué partes del programa se van ejecutando. De esta forma, la siempre ardua tarea de la detección de *bugs* y su arreglo se ve simplificada por las herramientas.

Estas herramientas de *debug* permiten, al menos, controlar la ejecución de código por parte del desarrollador. Esto es, una vez programado el microcontrolador, se puede ir ejecutando línea a línea el código.

Además, la mayoría de las opciones de *debug* permiten también la inserción de *breakpoints*, esto es, puntos de parada. Así, puede ejecutarse el código normalmente y parar el microcontrolador cuando la ejecución llegue a un *breakpoint*. De esta manera, se puede ejecutar la aplicación a su velocidad habitual y pararse solo cuando haga falta por la razón que sea.

En *debuggers* más avanzados se permite incluso los *breakpoints* complejos, donde es posible parar en un punto la ejecución del programa si se cumplen o no ciertas condiciones.

Una vez que se ha detenido la ejecución por parte de un *breakpoint*, es posible tanto analizar el estado del microcontrolador y sus periféricos, como ver los valores de las variables de nuestro código. En algunas ocasiones, incluso es posible modificar valores de las variables mientras se tiene al microcontrolador detenido para poder hacer pruebas.

Todo esto permite encontrar y arreglar los *bugs* que van surgiendo conforme se va desarrollando la aplicación.

4. Protección de memoria

Como se ha comentado, los microcontroladores no disponen de mecanismos de protección de memoria como sí tienen los procesadores de propósito general.

Estas unidades ayudan al sistema operativo (SO) a proteger ciertas zonas de memoria para que solo partes privilegiadas de este puedan usarlas y permiten también proteger las zonas de memorias de distintas tareas para que no puedan accederse por otras tareas ajenas.

Con este tipo de periféricos, el sistema entero está protegido contra código erróneo o malicioso, ya que no hay forma de que se pueda corromper el sistema operativo o una tarea pueda afectar a otra.

Sin embargo, no es muy común encontrar este tipo de unidades de protección de memoria en los microcontroladores actuales. Así pues, el SO no dispone de ningún mecanismo por parte del hardware que lo ayude a controlar quién accede a las distintas partes de la memoria.

Esto provoca que el SO esté desprotegido ante código erróneo o malicioso que pueda provocar el mal funcionamiento del sistema entero. Por esta razón se debe tener precaución extrema con la gestión de la memoria a la hora de programar para sistemas empotrados.

5. Organización de código

A la hora de organizar un proyecto de desarrollo de software empotrado, habitualmente se sigue una estructura común. Esta estructura se presenta a continuación y puede tener diferencias y variaciones según el entorno de trabajo, el fabricante o la costumbre del equipo de desarrolladores.

Lo más habitual es tener uno o varios ficheros que conforman el BSP (*board support package*). Este *package* es el que maneja el HW a más bajo nivel e inicializa todo el sistema. Así, en el BSP encontraremos funciones para configurar los periféricos del microcontrolador, configurar la entrada y salida de este, funciones para manejar partes específicas del sistema (ejemplos habituales son LED, o entradas o salidas simples), etc.

Dentro del BSP se pueden incluir las librerías de bajo nivel del fabricante (véase el subapartado 8.1 más adelante), aunque debido a la complejidad de estas, pueden mantenerse aparte del BSP.

Las librerías de más alto nivel (véase el subapartado 8.2) se suelen organizar en un nivel propio, ya que es código que raramente se va a editar o consultar. Estas librerías, normalmente de terceros, suelen estar ya probadas y con toda la funcionalidad, por lo que se usan como están sin necesidad de intervenir en su código.

En este apartado también se incluye el código o librería del SO en caso de que se use uno (véase el material titulado «Sistemas operativos» para más detalles). Normalmente, para el caso de sistemas empotrados, estos tienen un número limitado de líneas de código, y una vez configurado y parametrizado para nuestra aplicación no se suele editar ese código.

Por último, se suelen agrupar todos los ficheros de aplicación, con un fichero para cada módulo, dispositivo o función. Así, en el caso de usar un sistema operativo, cada tarea se escribe en un fichero separado. En el caso de usar varios dispositivos externos al microcontrolador, el controlador de cada uno se especifica en un fichero propio.

Hay que remarcar que esta organización no es fija, ni que todos los equipos de desarrolladores usan la misma. A pesar de ello, es muy habitual y recomendable organizar el código en diversos ficheros, y estos en directorios para una visión rápida, sencilla y global de todo el proyecto.

Lo que hay que evitar en todo caso es el llamado *spaghetti code*, que es código no estructurado, donde el flujo de ejecución no está claro y no hay ningún tipo de estructura de código.

6. Normas de codificación

Con el objetivo de normalizar las reglas a la hora de programar sistemas empotrados, han surgido diversas propuestas de conjuntos de reglas o buenas prácticas. Estas directrices deben ayudar a los desarrolladores a no cometer errores o introducirlos cuando escriben código para sistemas críticos.

Estas normas permiten mejorar la calidad del código, mejorando la seguridad, eliminando errores y asegurando su portabilidad a otros sistemas.

El uso de estas normas se está popularizando y es cada vez más común encontrar clientes o sectores que requieren certificaciones en esta materia para poder trabajar con ellos.

Normalmente estas normas son una lista de recomendaciones con diversos grados de importancia que todo el código de un sistema empotrado debe cumplir. Así, normalmente estas recomendaciones se dividen según su importancia en:

- Obligatorias: todas las reglas de este tipo deben cumplirse obligatoriamente en todo el código.

- Opcionales: todas las reglas de este tipo deben cumplirse o, si no se cumplen, se debe documentar el motivo y dar una muestra de que ese incumplimiento no introduce errores en ese proyecto.

- Aconsejables: se aconseja usar las reglas de este tipo, pero no son obligatorias.

6.1. Normativa MISRA

Esta normativa, conocida por MISRA por las siglas de The Motor Industry Software Reliability Association, fue promovida por los fabricantes de automoción (originalmente de Reino Unido) para el desarrollo de los sistemas electrónicos empotrados en sus vehículos [MISRA (b)].

La primera edición de estas guías fue en 1998 y se han ido ampliando y mejorando con el tiempo. La versión actual es la MISRA-C:2012, versión que incluye cerca de 160 reglas divididas en las tres categorías mencionadas anteriormente.

Cada una de las reglas está formulada como un enunciado formal seguido por una explicación detallada de por qué debe seguirse esa regla y qué tipo de problemas

puede dar no seguirla. En algunos casos se dan ejemplos de cómo debe ser un código que siga esa regla.

Estas directrices son de acceso público pero no gratuito, sino que tienen un pequeño coste para el usuario (actualmente 15 libras inglesas por documento) [MISRA (a)].

Las reglas dan normas de tipo muy variado, veamos algún ejemplo:

Regla MISRA 2.2: Todos los comentarios deben ser del tipo /* ... */. Esto se justifica por que algunos compiladores de C no soportan otros tipos de comentarios. También se justifica por que la versión de C que cubre esta normativa C90 solo soporta este tipo de comentarios.

Regla MISRA 13.2: Toda comparación con 0 debe hacerse explícita, a menos que el operando sea de tipo booleano. Esta regla se justifica para dejar explícito a todo programador que lea el código con qué valor se está comparando una variable.

Para conocer el grado de cumplimiento del código fuente de un proyecto para un sistema empotrado (o de cualquier otro tipo) existen herramientas capaces de analizar el código y realizar un informe sobre la mayoría de las directrices. Este tipo de herramientas se denomina *herramientas de análisis estático de código*. Algunas de estas herramientas o compiladores que integran la comprobación de las reglas son: PC-Lint, IAR Embedded Workbench, Compiladores de Texas Instruments, compiladores de Green Hills Software, etc.

6.2. Otras normativas

Entre las distintas normativas de codificación que introducimos aquí, MISRA es la más extendida actualmente, pero existen otros conjuntos de recomendaciones, como JSF AV++ [Lockheed Martin Corporation (2005)]. Esta recopilación agrupa 220 directrices.

Para aviación se usa la norma conocida por DO-178C [Wikipedia ()]. En esta norma se incluyen múltiples normas y reglas para el desarrollo de sistemas para aeronaves, no solo reglas de codificación de código. En esta norma, se clasifican los sistemas por su criticidad y se marcan parámetros de rendimiento y calidad para cada uno de los niveles de criticidad. Así, por ejemplo, un sistema marcado como crítico (cuyo fallo implica la pérdida de la aeronave) debe tener una tasa de error menor a $10^{-9}/h$. Será el fabricante del sistema el que deba probar cómo puede certificar que esa tasa de error se cumple.

Para el caso de dispositivos médicos o de salud, la situación es parecida. Existen normativas que listan los requisitos que se deben cumplir por parte de cada sistema o subsistema. La norma IEC 62304:2006 es un ejemplo de ello [ISO (2006)].

7. Multitarea

Cuando se dispone de un SO para sistemas empotrados, lo más habitual es que este disponga de tareas como unidad básica de organización de código.

Entones, una de las primeras acciones que debe realizar el desarrollador o equipo de desarrolladores será definir qué tareas existen, qué actividades realiza cada una de ellas y cómo se comunican entre sí. Esto no es algo fácil y es una parte crítica del diseño del sistema empotrado, ya que un diseño erróneo o que no cubra todos los aspectos que debe cumplir por parte del sistema provocará que al finalizar la codificación del código el sistema no cumpla con las especificaciones.

Gracias a las características de la mayoría de los SO actuales, la multitarea vendrá dada por el SO mismo, descargando de esta responsabilidad al programador.

A la hora de escribir código para sistemas multitarea, hay que tener en cuenta, entre otros, los siguientes aspectos:

- Definición de las prioridades: definir las prioridades de cada tarea es importante en aspectos de rendimiento, de cumplimiento de las especificaciones y para no caer en problemas de autobloqueo o de inversión de prioridades.

- Controlar el tamaño del *stack*: cada tarea se crea reservando para ella un cierto espacio de memoria para su *stack*. En esa zona de memoria la tarea la usará para crear variables, paso de parámetros a funciones, etc. Si el espacio de *stack* resulta insuficiente, habrá problemas de sobreescritura entre tareas y, por tanto, un comportamiento del sistema errático.

- Protección de variables: si dos tareas acceden a una misma variable compartida, se deberá proteger convenientemente el acceso a ella, de manera que en todo momento solo una de las tareas tenga acceso exclusivo a la variable. Si no se realiza esta protección, pueden encontrarse problemas de consistencia de los datos almacenados en dicha variable.

- Protección de funciones compartidas: si algún recurso del sistema manejado por sus propias funciones SW (*drivers* o librería propia) va a ser usado por varias tareas, hay que tener en cuenta que dichas funciones o librería debe estar protegida para ello.

Se denomina **inversión de prioridades** al bloqueo que puede ocurrir cuando una tarea de alta prioridad está esperando un recurso bloqueado por una tarea de prioridad menor. Al ser de baja prioridad, esta segunda tarea puede tardar en ser ejecutada y, al final, lo que ocurre es que se invierten las prioridades efectivas, ya que la de mayor prioridad está dependiendo de una tarea de menor prioridad. La manera más habitual de resolver este problema es tener **herencia de prioridades** para casos puntuales modificando temporalmente la prioridad de la tarea de menor prioridad para que herede la prioridad de la tarea bloqueada. Una vez que se desbloquea, la tarea de menor prioridad vuelve a tener su prioridad original.

7.1. Tareas

Para codificar una tarea en un SO normalmente debe usarse una función con un formato especial. Debido a que una tarea, en principio, no finaliza nunca, el código para implementarla debe basarse en un bucle infinito.

Así,lo habitual es definir una función que implementará la tarea, con ciertos parámetros según el SO concreto, una inicialización de variables o de recursos, y un bucle infinito donde está el código de la tarea propiamente dicho (véase el código 7.1).

Código 7.1: Esquema de una tarea

```
/* esta tarea puede recibir un parámetro */
/* en este caso no se usa */
void tarea_1(void *parametro) {

   int variables_a_inicializar;

   llamadas_a_funciones_inicio();

   while(1) {
      instrucción_1;
      instrucción_2;
   }

   /* Nunca se llega aquí */
}
```

Con esta estructura de tareas, la función *main* (el punto de entrada del programa) debe encargarse de inicializar el *board support package* (BSP) y todas las librarías que lo necesiten.

A continuación, es habitual crear e inicializar los mecanismos de comunicación necesarios para seguir con la creación de las tareas, y por último iniciar el *scheduler* del SO (véase el código 7.2).

Código 7.2: Esquema de la función main

```
void main(void) {
  int variables_a_inicializar;

  /* Inicialización BSP */
  BSP_init();

  /* Inicialización librerías */
  Libreria_1_init();
  Libreria_2_init();

  ...

  /* Creación de semáforos y colas */
  SemaphoreHandle_t semaforo_1 = xSemaphoreCreateBinary();
  QueueHandle_t cola_1 = xQueueCreate(20, sizeof(int) );

  /* Crear tareas */
  xTaskCreate(tarea_1, "Tarea1", 256, NULL, 4, NULL);
  xTaskCreate(tarea_2, "Tarea2", 256, NULL, 3, NULL);

  /* Iniciar scheduler */
  vTaskStartScheduler();

  while(1) {
    /* Nunca se llega aquí */
  }

  /* Nunca se llega aquí */
}
```

7.2. Comunicación entre tareas

Lo más habitual es que el SO incorpore un mínimo conjunto de funciones para la comunicación de tareas. Así, lo más normal es tener disponible funciones para el manejo de semáforos, mutex y colas (ver módulo sobre sistemas operativos). Otros sistemas operativos pueden tener más funciones de comunicación, como pueden ser memoria compartida o señales.

Como se ha visto en el apartado anterior, hay que instanciar cada mecanismo que se quiera usar, de modo que queda prefijado antes de empezar la ejecución del sistema completo. Luego, las tareas que lo requieran acceden a usar el recurso de forma sencilla.

8. *Drivers* para periféricos

Actualmente la mayoría de los fabricantes de microcontroladores proporciona una serie de librerías para facilitar el desarrollo de aplicaciones con sus dispositivos. Como es lógico, cada fabricante diseña estas librerías según su criterio y buenas prácticas.

Por regla general, estas librerías son modulares, de modo que se puede incorporar al proyecto en desarrollo solo las partes que van a usarse. Normalmente se dispone de una librería o módulo del fabricante para cada periférico de su microcontrolador, y se añaden al proyecto en desarrollo.

Aunque hay bastante variedad en cómo los fabricantes diseñan estas librerías, es habitual que el fabricante proporcione dos tipos de librerías, unas de bajo nivel donde se accede a los periféricos y se interactúa con ellos y otro de más alto nivel donde se accede a funciones más complejas.

8.1. *Drivers* de bajo nivel

Aquí se incluyen los *drivers* o librerías que proporcionan acceso a la funcionalidad básica de cada uno de los periféricos de un microcontrolador. De esta manera será más sencillo y rápido (y libre de errores) para el desarrollador trabajar con los periféricos del microcontrolador.

Por ejemplo, una librería de un fabricante para acceder al ADC (conversor analógico-digital) interno de un microcontrolador puede tener las siguientes funciones [Silicon Labs (b)]:

- adc_init(): inicia el ADC, configurándolo según un conjunto de parámetros elegibles por el desarrollador (*sampling time*, bits de trabajo, filtro previo, etc.).

- adc_start(): inicia la conversión. Con esta llamada el periférico ADC empezará una conversión de la entrada analógica según los parámetros de funcionamiento configurados previamente.

- adc_get_latest_value(): devuelve el valor una vez que el ADC ha realizado la conversión.

Otro ejemplo distinto, para el manejo de un periférico USART, un fabricante nos da estas funciones [Silicon Labs (c)]:

- USART_Init(): para inicializar y configurar el módulo con los parámetros de funcionamiento deseados (*baudrate*, número de bits, número de bits de stop, etc.)

- USART_Enable(): activa el módulo USART para que empiece a recibir y pueda enviar datos.

- USART_Rx(): recibe un dato del exterior siguiendo los parámetros previamente configurados.

- USART_Tx(): transmite un dato hacia el exterior siguiendo los parámetros previamente configurados.

Este tipo de librerías acceden directamente a los registros de configuración y datos de los periféricos, por lo que son totalmente específicas para cada fabricante, familia y modelo de microcontrolador.

Hay que destacar que, habitualmente, estas librerías no están preparadas para multitarea (véase el apartado 7) y por tanto es responsabilidad del desarrollador añadir los mecanismos de protección necesarios.

8.2. Librerías de alto nivel

Aparte de las librerías básicas de bajo nivel para acceder a los distintos periféricos, los fabricantes de microcontroladores van ofreciendo mayores facilidades a los desarrolladores.

Así pues, es cada vez más habitual encontrarse con librerías de alto nivel y de mayor complejidad proporcionadas por los fabricantes.

Por ejemplo, una tarea tediosa y compleja de hacer correctamente trabajando con librerías de bajo nivel es el manejo del DMA (*direct memory access*). Cada vez más, los fabricantes ofrecen librerías que integran el uso de DMA de forma que los desarrolladores no tienen por qué conocer los detalles de implementación de dicho sistema.

Siguiendo con el ejemplo de la USART del apartado anterior, una librería de mayor nivel ofrecerá funciones tales como:

- UART_Trasmit(): envía todo un *buffer* de datos de un tamaño determinado por el módulo USART y usando el DMA. Así, la CPU puede continuar trabajando mientras el periférico va enviando los datos.

- UART_Receive(): recibe un número de datos y los almacena en un *buffer*. Esta función también trabaja usando el DMA de modo que la CPU puede estar ocupada en otra tarea o activar un modo de bajo consumo (véase el apartado 9).

De esta manera, y como vemos en el ejemplo, el uso de funciones complejas como el manejo del DMA u otras quedan integradas en librerías proporcionadas por los propios fabricantes.

Por último, cabe decir que hay fabricantes que ofrecen también librerías de terceros (véase el apartado 10) integradas en sus entornos de trabajo y con sus librerías de bajo nivel, lo que facilita el uso de estas librerías de terceros para proyectos complejos.

9. Programación para bajo consumo

En todo lo visto hasta ahora sobre cómo programar sistemas empotrados no se ha tenido en cuenta el factor de tener un sistema que deba funcionar en bajo consumo.

En casos de dispositivos que deban minimizar la energía que usan para su funcionamiento, ya sea porque funcionan alimentados por baterías o en entornos donde no hay una fuente de energía fiable, se suelen seguir una serie de estrategias para minimizar en lo posible el consumo del sistema.

Estas estrategias se basan en tener desconectados todo el tiempo que sea posible la mayor parte del sistema. El manejo de qué partes están conectadas (alimentadas) en cada momento se realiza por el software que corre en el microcontrolador como dispositivo principal. A su vez, dentro del microcontrolador se sigue la misma estrategia con los periféricos que se usan y están alimentados en cada momento.

Así, si por ejemplo un puerto serie se usa solamente cuando el sistema debe comunicarse con un dispositivo externo, la USART correspondiente dentro del microcontrolador se desconecta (se le retira la alimentación y/o el reloj del sistema) para que no esté funcionando y, por tanto, no esté consumiendo energía.

Asimismo, esta estrategia se lleva a cabo incluso con la CPU del propio microcontrolador. Así, en dispositivos que no usan la capacidad de cómputo del microcontrolador durante periodos largos de tiempo, es posible desactivar la CPU y así reducir el consumo de manera considerable (ver tabla 1 y figura 2) .

En estos casos, se reanuda el funcionamiento de la CPU cuando ocurre algún evento, ya sea interno o externo, como puede ser que se genere una interrupción debido a un cambio en una señal que llega por un GPIO o un Timer genere una interrupción porque ha llegado a término su cuenta.

Tabla 1. Ejemplo de consumos y periféricos disponibles

Nivel de funcionamiento	CPU	Periféricos	Consumo
0	Funcionando	Todos disponibles	5.6 mA
1	Parada	Todos disponibles	1.71 mA
2	Parada	UART, ADC, DAC, DMA, RTC y GPIO	1 μA
3	Parada	RTC y GPIO	0.59 μA
4	Apagada	GPIO	0.02 μA

Elaboración propia a partir de mediciones propias y Silicon Labs (2015b)].

Figura 2. Medición de consumo. Se observan al menos tres modos de bajo consumo en la gráfica

Elaboración propia.

Así, hay que recordar que el consumo total de un dispositivo es una función del consumo instantáneo y del tiempo que está en ese consumo. Por tanto, interesa que un dispositivo esté la mayor parte del tiempo en muy bajo consumo y es aceptable que por periodos cortos de tiempo aumente considerablemente su consumo para realizar su tarea.

En el caso de usar un SO, si este no incluye ya funciones de bajo consumo para el microcontrolador que estemos trabajando, normalmente se ofrecen ayudas para que el desarrollador pueda añadir sus estrategias de bajo consumo.

Así, en el caso de FreeRTOS, que no da soporte directo a bajo consumo, sí que da las siguientes funcionalidades:

- *IdleHook*: permite que la tarea *Idle* llame a una función propia (*hook*). Como la tarea *Idle* se ejecuta cuando el sistema operativo no tiene disponible ninguna otra tarea para ser ejecutada, pueden activarse entonces algunas estrategias de bajo consumo, como, por ejemplo, poner en modo *sleep* el microcontrolador por un cierto periodo de tiempo.

- *Tickless*: en este modo de funcionamiento, el SO no ejecuta siempre el *tick* de sistema si no es necesario. Así, si todas las tareas están bloqueadas y la siguiente tarea a despertarse está durmiendo por un cierto periodo de tiempo, el microcontrolador puede ponerse en modo *sleep* todo ese tiempo mientras un *timer* cuenta el tiempo para despertarse.

- *Low Power Tick*: en el caso de microcontroladores Cortex cabe la posibilidad de generar el *tick* de sistema con un *timer* de bajo consumo en lugar del dispositivo estándar SysTick Timer.

10. Librerías habituales

Aparte de las librerías que los fabricantes de microcontrolador suelen ofrecer de forma libre para el manejo de sus periféricos, existen librerías que no están ligadas a un fabricante y realizan tareas complejas.

10.1. Lightweight IP

Lightweight IP (abreviada lwIP) es una librería de código abierto que implementa todo el *stack* TCP/IP y diseñada para sistemas empotrados. La librería esta diseñada para ocupar el mínimo espacio de memoria posible tanto en RAM como en ROM/FLASH [Lightweight (2017)].

La lista de protocolos implementados en esta librería es enorme: empezando por TCP y UDP, pasando por IP e IPv6 y terminando en PPP o ICMP. Además, como *addons* hay servidores de HTTP y TFTP, clientes MQTT etc.

Usando esta librería en un sistema empotrado, el desarrollador se ahorra la tarea de escribir el código necesario para soportar todos estos protocolos, lo que acelera la implementación y evita todos los *bugs* asociados a todo desarrollo.

10.2. FatFS

FatFS es una librería para sistemas empotrados que maneja el sistema de ficheros FAT [ChaN (2013)]. Esta librería está diseñada para ser totalmente independiente de los *drivers* de bajo nivel, de modo que es fácilmente portable a distintas plataformas y dispositivos.

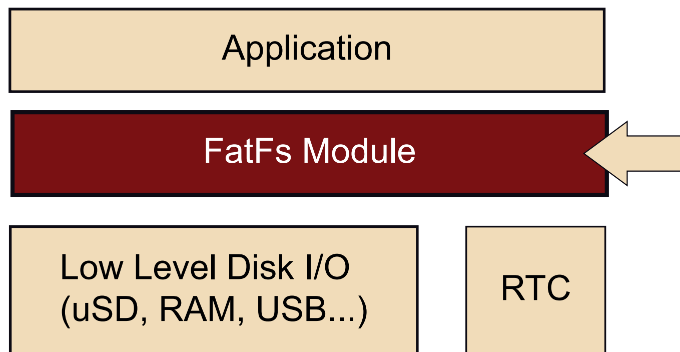
Así, por un lado esta librería ofrece las funciones habituales para trabajar con ficheros (abrir o cerrar fichero, escribir o leer datos, moverse a una posición determinada, renombrar fichero, etc.) y trabajar con directorios (crear directorio, listar contenido de un directorio).

Por otro lado, necesita que el desarrollador implemente unas pocas funciones de bajo nivel y específicas para el dispositivo de almacenamiento que se esté usando (figura 3).

Sistema de ficheros FAT

El sistema de ficheros FAT fue diseñado en 1977 y usado por Microsoft en sus primeros sistemas operativos. Aún se usan variantes y mejoras del original en discos duros, tarjetas SD, dispositivos USB, etc.

Figura 3. Diagrama de capas de la librería FatFS [ChaN (2013)]



De esta forma, si el dispositivo de almacenamiento es una tarjeta SD habrá que implementar la funcionalidad para leer o escribir en los sectores de dicha tarjeta, obtener el estado de la tarjeta (espacio libre, etc.), etc. En el caso de un dispositivo USB de almacenamiento (*USB dongle*), habrá que usar las librerías de bajo nivel USB para implementar las mismas funciones.

Como muestra, un proyecto de ejemplo de ST microelectronics permite configurar una placa de evaluación con una interfaz SD y una interfaz USB para que actúe como un dispositivo de almacenamiento masivo USB. Así, usando la librería FatFS y una librería USB propia de ST, el desarrollador tan solo tiene que escribir unas pocas líneas de código y unos pocos minutos para conseguir que el desarrollo funcione.

10.3. Librerías gráficas

Otro tipo de librerías que suelen ofrecer los fabricantes de microcontroladores son las librerías gráficas para los dispositivos que integran controladores para pantallas. Debido a la complejidad de controlar y manipular gráficos en una pantalla, se suelen usar librerías escritas por terceras partes para tal fin [STMicroelectronics (2017a)].

Habitualmente estas librerías pueden trabajar con un gran número de pantallas o modelos, ya que pueden configurarse para cada caso los principales parámetros de funcionamiento (como número de píxeles, profundidad de color, tipo de comunicación, velocidad de refresco, etc. Véase figura 4).

Figura 4. Diagrama de capas de la librería STemWin

© STMicroelectronics. [STMicroelectronics (2017a)].

Existen diversas librerías de este tipo y cada fabricante suele proporcionar una distinta. No obstante, la mayoría de ellas tienen las características parecidas:

- Bajo nivel: normalmente ya implementado por el fabricante para sus microcontroladores, permite el manejo del controlador de pantalla.
- Impresión de texto: una parte de la librería permite insertar texto en distintos tamaños y fuentes en la pantalla.
- Primitivas de dibujo básicas: permite dibujar formas básicas en la pantalla, como líneas, círculos, áreas, etc.
- Manejo de ratón y punteros: permite situar un puntero de ratón y moverlo por la pantalla.
- Manejo de GUI: permite dibujar GUI (*graphial user interface*) mediante *widgets* avanzados y el manejo de ventanas.
- Primitivas de dibujo avanzadas: en algunos casos se incluyen funciones de gráficos avanzadas, como animaciones, manejo de 3D o presentación de fotografías en la pantalla.

Con el uso de librerías de este tipo se simplifica el tiempo y la dificultad de desarrollar una aplicación que use una interfaz gráfica para comunicarse con el usuario.

Widget

Los *widgets* son pequeños gráficos con cierta funcionalidad e interacción. Por ejemplo: barras laterales de desplazamiento, botones, *checkbox*, etc.

10.4. CMSIS

La librería CMSIS (sigla de *cortex microcontroller software interface standard* es una propuesta de ARM [ARM (2017a)].

Esta librería está diseñada para tener funciones comunes a cualquier sistema empotrado que funcione con microcontroladores Cortex de ARM, sea cual sea el fabricante. De esta manera, ARM propone un conjunto de funciones para que los fabricantes las implementen para sus microprocesadores concretos.

Así, para un desarrollador, cambiar de fabricante sin salirse de la familia Cortex le supondrá un coste de desarrollo menor, ya que parte del código se podrá reutilizar.

Esta librería consta de los siguientes módulos:

10.4.1. CMSIS-CORE

Se incluyen funciones para el manejo de las interioridades de las distintas CPU de la familia Cortex, incluyendo el manejo de las interrupciones, configuración del *timer* del sistema SysTick, distribución de los relojes de sistema, etc.

Para ciertos *cores* (Cortex-M7), esta librería incluye el manejo de las cachés, tanto de datos como de instrucciones.

10.4.2. CMSIS-Driver

En este módulo se definen llamadas estándar para distintos periféricos, como por ejemplo: CAN, Ethernet, I2C, memorias flash, USART, USB, etc.

10.4.3. CMSIS-DSP

Librería con funciones de tipo DSP (*digital signal processing*) optimizadas para cada una de las distintas CPU de la arquitectura Cortex.

Así, esta librería ofrece funciones para cálculo vectorial y matricial, trigonométricas, cálculo con números complejos, implementación de filtros (FIR, IIR, etc.), funciones tipo FFT (*fast fourier transform*) y funciones estadísticas.

10.4.4. CMSIS-RTOS

Este módulo sirve para implementar distintos RTOS manteniendo las mismas llamadas genéricas. De esta manera, la tarea de cambiar de RTOS se simplifica enormemente.

Esta librería implementa llamadas a gestión de tareas (crear, destruir, suspender, etc.), comunicación entre tareas (mutex, semáforos, colas, etc.), manejo de tiempo, etc.

Así, usando esta librería, es posible desarrollar un código que crea una tarea *task* de forma estandarizada. Para una aplicación determinada, este código se ejecutará encima de, por ejemplo, FreeRTOS, y será la librería CMSIS-RTOS la que indique al SO la creación de la tarea. En otra aplicación, el mismo código funcionando sobre RTX de Keil creará la misma tarea usando la misma función de la librería CMSIS-RTOS.

10.4.5. CMSIS-Pack

Módulo para las herramientas de programación que les permite acceder a cada fichero de un proyecto, así como configuraciones y demás.

De este modo, es posible portar un proyecto entero, conteniendo la documentación, la descripción de la plataforma usada, las librerías, el código fuente, etc., de una forma estandarizada.

Asimismo, usando CMSIS-Pack una empresa desarrolladora de software empotrado puede distribuir su producto de una manera sencilla para sus clientes.

10.4.6. CMSIS-SVD

Descripción en XML de los periféricos de un sistema. Este módulo está orientado a las herramientas para programadores o desarrolladores.

Con el uso de esta descripción, se permite que la misma librería pueda trabajar con distintos microcontroladores y distintas configuraciones de este, ya que la descripción explicita qué dispositivos y en qué direcciones de memoria están mapeados.

10.4.7. CMSIS-DAP

Módulo para la simplificación y ayuda en las tareas de *debug*, tanto para las herramientas de desarrollo como para los desarrolladores.

Con esta librería es posible funciones de *debug* avanzadas, como son el perfilado energético de la aplicación empotrada o tener una consola de *debug*. Estas funciones debe implementarlas el dispositivo *debugger* siguiendo las especificaciones de este módulo.

Bibliografía

ARM (2017a). «CMSIS - Cortex Microcontroller Software Interface Standard».
URL: https://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php.

ChaN (2013). «FatFs - Generic FAT Filesystem Module». URL: http://elm-chan.org/ fsw/ff/00index_e.html.

ISO (2006). «IEC 62304:2006 Medical device software — Software life cycle processes».

Lightweight (2017). «LwIP - A Lightweight TCP/IP stack - Summary». URL: https: //savannah.nongnu.org/projects/lwip/.

Lockheed Martin Corporation (2005). «Joint Strike Fighter Air Vehicle C++ Coding Standards».

MISRA (a). «MISRA Buy onlin». URL: https://www.misra.org.uk/Buyonline/ tabid/58/Default.aspx.

MISRA (b). «MISRA homepage». URL: https://www.misra.org.uk/.

Silicon Labs (b). «Emlib - ADC». URL: https://siliconlabs.github.io/Gecko_ SDK_Doc/efm32gg/html/group__ADC.html.

Silicon Labs (c). «Emlib - UART». URL: https://siliconlabs.github.io/Gecko_ SDK_Doc/efm32gg/html/group__USART.html.

Silicon Labs (2015b). *EFM32TG108 DATASHEET*. Silicon Labs.

STMicroelectronics (2017c). «LCD-TFT display controller (LTDC) on STM32 MCUs».
URL: http://www.st.com/resource/en/application_note/dm00287603.pdf.

Wikipedia (). «DO-178C». URL: https://en.wikipedia.org/wiki/DO-178C.