Internet de las Cosas

Práctica 1.B

Introducción

Este segundo bloque de la primera práctica pretende introducir los elementos principales de los núcleos de tiempo real, a veces llamados también sistemas operativos, para microcontroladores. Estos sistemas permiten desarrollar una programación mucho más flexible y modular basada en tareas (hebras) y aportan, como mínimo, un interesante conjunto de primitivas de sincronización, comunicación y priorización de tareas que facilitan el desarrollo de aplicaciones complejas con restricciones de tiempo real.

Objetivos

- Introducir los componentes básicos de un sistema de tiempo real para microcontroladores.
- Conocer las características principales de ChibiOS y algunos de los recursos disponibles para sincronizar, comunicar o priorizar tareas.
- Explorar el uso de ChibiOS para implementar firmware multienhebrado basado en un planificador de tipo Round-Robin con prioridades.

Sistemas de tiempo real para microcontroladores

Todos los sistemas de tiempo real tienen como objetivo la definición de aplicación que **garantice** ciertas respuestas dentro de unos márgenes de tiempo (restricciones de tiempo real, blando o duro). Es muy importante que el sistema resultante sea predecible y determinista (responderá de la misma forma ante las mismas condiciones).

Estos sistemas se programan en base a pequeñas tareas que desarrollan funciones muy concretas, bien de forma periódica o tras la generación de eventos específicos. Desde el punto de vista técnico, estas tareas se articula como hebras que se ejecutan de forma concurrente. Esto implica que es necesario garantizar que las funciones que se incluyan en el código debe ser “thread-safe” y reentrante. Una función es segura desde el punto de la *programación multienhebrada o multi-hilo* (thead-safe) si el acceso a estructuras de datos y recursos compartidos entre varias hebras están protegidos por mecanismos de exclusión (mutexs, semáforos, etc) que garanticen la coherencia e integridad de estas operaciones. Complementariamente, una función verifica ser reentrante está preparada para que pueda ser invocada de forma concurrente desde varias hebras. Esto implica que no puede conservar variables estáticas entre llamadas y que tampoco devuelve un puntero a una variable estática (global). Evidentemente, una función reentrante no puede llamar a funciones no reentrantes.

Las funciones malloc() y free(), que gestionan la asignación dinámica de memoria, no son “thread-safe”, como tampoco lo serán las librerías que las utilicen, como es el caso de las librerías SD y String de Arduino. La gestión de la memoria es un aspecto esencial en cualquier RTOS, y todos disponen de primitivas para asignar a cada hebra el uso exclusivo de una parte de la RAM para ser usado como pila de llamadas o *stack* de esa hebra. Adicionalmente, cada RTOS puede aportar diferentes soluciones para permitir la creación un memoria *heap* (montículo libre, memoria global reservada, ...) donde sea posible gestionar cierta cantidad de memoria de forma dinámica.

Un problema central a cualquier RTOS es la forma en la que se gestiona el reparto del tiempo de procesador entre las diferentes hebras. La solución básica, presente en cualquier RTOS, consiste en ordenar la precedencia o importancia de cada tarea en base a un nivel de prioridad. Cuando una tarea de prioridad superior está lista para ejecutarse, la tarea de prioridad inferior que se esté ejecutando será desplazada. Cuando la competencia por el procesador se presente entre hebras de idéntica prioridad, hay dos soluciones o esquemas básicos:

- **Round-robin cooperativo**: la tarea en ejecución debe contemplar esta posibilidad en su diseño y estar preparada para ejecutarse de manera ágil durante periodos de tiempo de unos pocos milisegundos y ceder el control del procesador entre estos periodos.
- **Round-robin con desplazamiento**: Se establece en la configuración del RTOS el cuanto o rodaja de tiempo máximo que una tarea puede conservar el control del procesador. La hebra puede ceder el control del procesador de forma voluntaria si ha terminado, pero si agota su cuota de tiempo, será desplazada por el planificador de tareas para permitir la ejecución de cualquier otra tarea en su mismo nivel de prioridad que esté lista para ejecutarse.

A la hora valorar la conveniencia de elegir un sistema operativo de tiempo real (Real-Time Operative System, RTOS) para sistemas empotrados son varios los factores a tener en cuenta:

- Nivel de soporte para el microcontrolador elegido (y otras arquitecturas)
- Cuánta memoria FLASH/RAM extra implica la utilización del RTOS y qué sobrecarga temporal implica en tiempo de ejecución
- Tipo de planificador de tareas (scheduler)
- Cuánto tiempo tarda en conmutar el control del microcontrolador entre dos tareas (hard real-time context switching time)
- Existencia de una capa de abstracción del hardware (Hardware Abstraction Layer, HAL)
- Disponibilidad de documentación sobre el RTOS
- Nivel de actividad/desarrollo actual y existencia de algún tipo de soporte
- Tipo de licencia que implica el uso del software

Aunque existen numerosas alternativas comerciales, nos focalizamos en esta introducción en un conjunto reducido de RTOS de código abierto bien conocidas. La mayoría de estos núcleos de tiempo real asumen el uso de C/C++ como lenguaje de programación del propio RTOS y de las aplicaciones.

ChibiOS

ChibiOS es un proyecto liderado por Giovanni Di Sirio y se encuentra fuertemente vinculado a la serie de procesadores STM32 de la empresa ST Microelectronics, aunque incluye un soporte básico para otras arquitecturas. Se caracteriza por la optimización del código del RTOS, tanto en tamaño como en sobrecarga en tiempo de ejecución.

Se organiza en dos capas o niveles principales (ver figura 1). La capa de abstracción del hardware o HAL oculta los detalles de acceso y programación de periféricos de cada arquitectura mediante la introducción de modelos genéricos de manejadores de dispositivos o drivers. La otra capa es la que corresponde al núcleo de tiempo real (RT) e implementa todos los elementos de despacho de tareas y mecanismos de sincronización y/o comunicación de hebras del RTOS. Las dos capas son independientes y es posible usar el núcleo de tiempo real excluyendo la capa HAL, justo la aproximación usada en la adaptación de Bill Greiman de ChibiOS, como veremos a continuación.

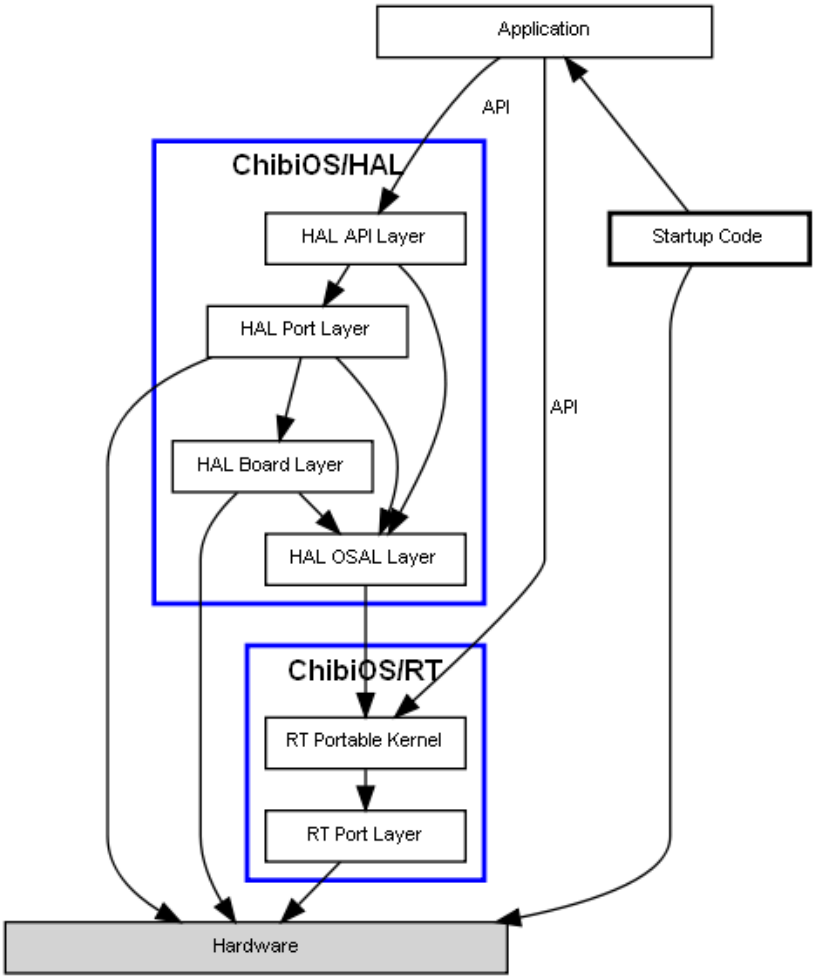

Figura 1: Arquitectura interna de ChibiOS

Como en el caso de RTOS y Zephyr, es posible usar ChibiOS de forma integral, pero esto implicaría un largo proceso de aprendizaje del entorno de programación, de las diferentes opciones de configuración vinculadas al microcontrolador, etc. Con el objetivo de poder introducir y ejercitar estos conceptos, pero esquivando estas dificultades, en esta práctica usaremos la adaptación de

ChibiOS 20.3.3 de Bill Greiman, basado en el núcleo de tiempo real ChibiOS/RT 6.1.3, disponible para procesadores ARM y AVR. Esta implementación excluye la capa de abstracción hardware (HAL) y contiene solo la parte vinculada a la gestión de hebras, incluyendo el despachador de tareas, y los mecanismos de comunicación y sincronización de las mismas. Es una implementación muy ligera que, a través de una librería, permite una integración muy sencilla con el IDE de Arduino.

FreeRTOS

FreeRTOS es un RTOS de larga tradición con una licencia tipo MIT de código abierto que soporta una amplia variedad de arquitecturas de microcontroladores. Dispone de una buena documentación y que se sigue desarrollando activamente en la actualidad. Recientemente se ha postulado su integración como capa de tiempo real en AZURE, la plataforma de computación en la nube de Microsoft.

Se organiza en base a un núcleo (kernel) y un conjunto de librerías específicas. El núcleo de tiempo real no es tan compacto como en el caso de otros RTOS y su uso implica sumar entre 6 y 12 Kbytes al firmware.

Incluye un planificador de tareas basado en prioridades. Como en muchos otros RTOS, los dos modos básicos de control en la ejecución de las tareas son por desplazamiento (preemptive) de la tarea en ejecución al agotar su rodaja de tiempo o de forma cooperativa entre hebras de idéntica prioridad. En este último modo el diseñador debe estructurar la ejecución de las diferentes tareas en etapas ágiles, de corta duración, tras las cuales la tarea cede el control del microcontrolador a otras tareas.

En principio, Bill Greiman adaptó en 2015 la versión 8.2.3 de FreeRTOS para su uso como librería desde el IDE de Arduino, pero esta adaptación no se mantiene de forma activa y puede considerarse obsoleta. Posteriormente, en 2020 la empresa china Seeed-Studio ha adaptado la versión 10.4.3 de FreeRTOS para permitir el uso de FreeRTOS desde el IDE de Arduino, pero solo sobre procesadores SAMD21 y SAMD51, sin documentación y con un conjunto de ejemplos muy reducido.

Zephyr

Zephyr es una iniciativa relativamente reciente que ha conseguido un gran respaldo por parte de empresas y usuarios. Se trata de un RTOS que comparte muchos de los elementos básicos ya descritos para ChibiOS y FreeRTOS. Adicionalmente, añade un conjunto más amplio de esquemas de despacho de tareas y pone un cierto énfasis en facilitar el uso de librerías con cumplan con el estándar POSIX o hagan uso de CMSIS (Cortex Microcontroller Software Interface Standard) para microcontroladores de la familia ARM CORTEX-M. Se trata de un software de código abierto con una licencia de tipo Apache 2.0, salvo para ciertas partes que se han tomado de otros desarrollos y que tienen sus propias licencias.

Ejemplos con ChibiOS

Los siguientes ejemplos ilustran de forma simple el posible uso de algunos recursos básicos presentes en cualquier RTOS. Es esta ocasión, basaremos el desarrollo de estos ejemplos en ChibiOS, más concretamente en la adaptación de su núcleo de tiempo real realizado en la librería ChRT.

Ejemplo 1: Creación de hebras

Este primer ejemplo ilustra la creación de hebras en ChibiOS y el uso de los dos posibles esquemas de despacho de tareas: round-robin con desplazamiento y round-robin cooperativo. Se compone de tres hebras:

- Una hebra que se limita a hacer parpadear el BULTIN_LED del Arduino MKR 1310 a 200 ms (50 ms encendido, 150 ms apagado). En el modo round-robin no cooperativo esta hebra es la de mayor prioridad (NORMALPRIO + 1).
- Una segunda hebra que únicamente incrementa una variable contadora. Cuando se usa un esquema no cooperativo es la hebra de menor prioridad (NORMALPRIO – 1) que se ejecuta, por ese nivel prioridad, “siempre que puede”, es decir, cuando ninguna de las hebras de mayor prioridad está preparada para ejecutarse.
- La tercera hebra imprime el valor del contador en intervalos de 1 segundo. Esta hebra reinicia a cero el valor de la variable contadora cada vez que se ejecuta y, en el modo no cooperativo, se ejecuta con un nivel de prioridad intermedio (NORMALPRIO). Nótese que en este ejemplo, las tres hebras reciben el mismo nivel de prioridad cuando se activa el modo cooperativo.

Aspectos a notar:

- La función chSetup() verifica que la configuración de ChibiOS es compatible con la elección del esquema de despacho de tareas. Para ello verifica el valor de algunas macros definidas en el fichero chconfig correspondiente a la arquitectura del microcontrolador. Más concretamente, el uso del esquema cooperativo exige fijar la macro CH_TIME_QUANTUM a cero. En este ejemplo, el valor de la macro `COOPERATIVE_SCHEDULER` permite automatizar estas verificaciones y detener la ejecución, si la configuración es incorrecta.
- Es necesario definir un tamaño de stack para cada una de las hebras. El tamaño “correcto” debe ser aquel que, sin malgastar la siempre escasa RAM, permita la invocación de las funciones y la declaración de las variables auxiliares requeridas. Es recomendable comenzar con unos tamaños generosos e irlos ajustando a medida que la implementación del firmware se vaya estabilizando. ChibiOS dispone de la función `chUnusedThreadStack()` para consultar el tamaño de la parte no usada del stack de la hebra.
- Las variables que se actualizan y/o se consultan desde diferentes hebras, en este ejemplo la variable contadora `count`, deben declararse volatile.
- El uso de la pareja de llamadas `noInterrupts()` e `interrupts()` permite proteger un cierto bloque de sentencias frente a interrupciones, incluyendo cambios de contexto.

```
/* --------------------------------------------------------------------
 *  Ejemplo ChibiOS-1
 *    Este ejemplo ilustra cómo es posible crear varia hebras de forma
 *    estática. Permite experimentar con el uso de un esquema
 *    cooperativo de despacho de hebras de igual prioridad u otro
 *    no cooperativo (round-robin con desplazamiento)
 *
 *    Requiere el uso de la librería ChRt de Bill Greiman
 *        https://github.com/greiman/ChRt
 *
 *  Asignatura (GII-IoT)
 * --------------------------------------------------------------------
 */

#include "ChRt.h"

// El uso de un esquema ROUND_ROBIN requiere fijar esta macro a false
#define COOPERATIVE_SCHEDULER   true

volatile uint32_t count = 0;

#if COOPERATIVE_SCHEDULER

  volatile uint32_t maxDelay_i = 0;
  #define BLINKING_THD_PRIO     NORMALPRIO
  #define COUNTING_THD_PRIO     NORMALPRIO
  #define PRINTING_THD_PRIO     NORMALPRIO

#else

  #define BLINKING_THD_PRIO     (NORMALPRIO + 1)
  #define PRINTING_THD_PRIO     (NORMALPRIO)
  #define COUNTING_THD_PRIO     (NORMALPRIO - 1)

#endif

// --------------------------------------------------------------------
// BlinkingThread - Hace parpadear el LED
//                  Prioridad:  BLINKING_THD_PRIO
//                  STACK: 64 bytes
// --------------------------------------------------------------------
static THD_WORKING_AREA(wa_BlinkingThread, 64);

static THD_FUNCTION(BlinkingThread , arg)
{
```

```
  (void)arg;

  // LED_BUILTIN = 13 (predefinido)
  pinMode(LED_BUILTIN, OUTPUT);

  // Flash led every 200 ms.
  while (true) {
    // Turn LED on.
    digitalWrite(LED_BUILTIN, HIGH);

    // Sleep for 50 milliseconds.
    chThdSleepMilliseconds(50);

    // Turn LED off.
    digitalWrite(LED_BUILTIN, LOW);

    // Sleep for 150 milliseconds.
    chThdSleepMilliseconds(150);
  }
}
// ------------------------------------------------------------------------
// PrinterThread - Imprime el valor del contador
//                 Prioridad: PRINTING_THD_PRIO
//                 STACK: 256 bytes
// ------------------------------------------------------------------------
static THD_WORKING_AREA(wa_PrinterThread, 256);

static THD_FUNCTION(PrinterThread , arg)
{
  (void)arg;

  while (true) {
    // Dormimos durante un segundo
    chThdSleepMilliseconds(1000);

    // Imprime el conteo del segundo anterior
    #if COOPERATIVE_SCHEDULER == false

      SerialUSB.print("Count: ");
      SerialUSB.println(count);

      // Ponemos a cero el contador.
      count = 0;

    #else
```

```
      SerialUSB.print("Count: ");
      SerialUSB.print(count);
      SerialUSB.print(" Max delay(usec): ");
      SerialUSB.println(TIME_I2US(maxDelay_i));

      // Ponemos a cero el contador.
      count = 0;
      maxDelay_i = 0;
    #endif
  }
}

// ------------------------------------------------------------------------
// CountingThread - Hebra contadora
//                  Prioridad: COUNTING_THD_PRIO
//                  STACK: 64 bytes
// ------------------------------------------------------------------------
static THD_WORKING_AREA(wa_CountingThread, 64);

static THD_FUNCTION(CountingThread , arg)
{
  (void)arg;

  while (true) {
    noInterrupts();
    count++;
    interrupts();

    #if COOPERATIVE_SCHEDULER
      systime_t t_i = chVTGetSystemTimeX();
      chThdYield();
      t_i = chVTTimeElapsedSinceX(t_i);
      if (t_i > maxDelay_i) maxDelay_i = t_i;
    #endif
  }
}

// ------------------------------------------------------------------------
// Creamos tres hebras
// ------------------------------------------------------------------------
void chSetup()
{
  // Aquí asumimos que el valor de CH_CFG_ST_TIMEDELTA es siempre cero
  // para cualquier tarjeta basada en SAMD que están soportadas solo en
```

```
  // "tick mode"

  // Verificamos primero la configuración de ChibiOS
  if (COOPERATIVE_SCHEDULER) {
    if (CH_CFG_TIME_QUANTUM) {
      SerialUSB.println("You must set CH_CFG_TIME_QUANTUM zero in");
      #if defined(__arm__)
          SerialUSB.print("src/<board type>/chconfig<board>.h");
      #elif defined(__AVR__)
          SerialUSB.print("src/avr/chconfig_avr.h");
      #endif
      SerialUSB.println(" to enable cooperative scheduling.");
      while (true) {}
    }
  }
  else {
    if (CH_CFG_TIME_QUANTUM == 0) {
      SerialUSB.println("You must set CH_CFG_TIME_QUANTUM to a non-zero value in");
      #if defined(__arm__)
          SerialUSB.print("src/<board type>/chconfig<board>.h");
      #elif defined(__AVR__)
          SerialUSB.print("src/avr/chconfig_avr.h");
      #endif
      SerialUSB.println(" to enable round-robin scheduling.");
      while (true) {}
    }
  }

  // Lanzamos la hebra que controla el LED.
  // Prioridad: BLINKING_THD_PRIO
  chThdCreateStatic(wa_BlinkingThread, sizeof(wa_BlinkingThread),
                    BLINKING_THD_PRIO, BlinkingThread, NULL);

  // Lanzamos la hebra que imorime el conteo.
  // Prioridad: PRINTING_THD_PRIO.
  chThdCreateStatic(wa_PrinterThread, sizeof(wa_PrinterThread),
                    PRINTING_THD_PRIO, PrinterThread, NULL);

  // Lanzamos la hebra de menor prioridad (CountThread).
  // Prioridad: COUNTING_THD_PRIO
  chThdCreateStatic(wa_CountingThread, sizeof(wa_CountingThread),
                    COUNTING_THD_PRIO, CountingThread, NULL);
}

// ------------------------------------------------------------------------
```

```
// Set up puede usarse para activar los dispositivos que vayan a
// emplearse
// ------------------------------------------------------------------------
void setup()
{
  SerialUSB.begin(115200);

  // Wait for USB Serial.
  while (!SerialUSB) {}

  // chBegin() resetea el stack y nunca retorna.
  chBegin(chSetup);
}

// ------------------------------------------------------------------------
// El entorno Arduino exige la declaración de esta función que se trata
// como una hebra de prioridad normal (NORMALPRIO)
//
// NUNCA usar delay() para imponer un retardo. Si lo hacemos, impediremos
// la ejecución de las hebras de menor prioridad. Esto es fácil de comprobar
// en este ejemplo sustituyendo la llamada a chThdSleepMilliseconds() por
// delay(). Usar siempre chThdSleepMilliseconds() o alguna función
// equivalente
// ------------------------------------------------------------------------
void loop()
{
  SerialUSB.println("Loop");
  chThdSleepMilliseconds(5000);
  //delay(5000);
}
```

Ejemplo 2: Comunicación de hebras mediante mailboxes

ChibiOS dispone de varias abstracciones para intercomunicar hebras. Una de ellas, basada en mensajes (ver chmsg.h), permite enviar un mensaje directamente a una hebra. Alternativamente, es posible definir eventos y usarlos entre hebras. Un evento es equivalente a un mensaje con tipo, pero sin contenido. Se puede ver un ejemplo del uso de eventos en este enlace.

Otra abstracción disponible en ChibiOS se basa en el uso de buzones de correo o mailboxes. Este recurso de comunicación entre hebras es asíncrono y no dirigido, de manera que cualquier hebra puede alojar mensajes (“echar una carta al buzón”) o extraerlos de él. De esta forma, un mailbox es un mecanismo flexible que puede servir para que varias hebras envíen mensajes a otra hebra que los despacha (nuestro ejemplo) o justo al revés. En general, los mensajes se despachan (se leen) en el orden en el que se insertan en el mailbox, pero también es posible insertar mensajes de alta prioridad que serán los primeros en ser extraídos del buzón.

En este ejemplo se crean tres hebras idénticas (WorkThread) cuyo cometido es crear mensajes, que incluyen el nombre de la hebra y el tiempo de creación en **ticks** (unidades internas de conteo de tiempo), e insertarlos en un buzón o mailbox (printer_mailbox) a razón de un mensaje por segundo. En paralelo, otra hebra (PrinterThread) monitoriza la recepción de estos mensajes y cuando ve que el mailbox tiene "correo" lo extrae e imprime el mensaje en pantalla.

El código se ha organizado en tres ficheros, un fichero de configuración (config.h), dos ficheros, printer.ino y worker.ino, que contienen, respectivamente, el código de la hebra "printer" y de la hebra "worker" y el sketch principal, ChibiOS_EJ2_MKR1310_ArduinoIDE.ino.

Es importante llamar la atención sobre el fichero **config.h,** que detalla la secuencia de pasos que implica la declaración de un mailbox en ChibiOS, y que son básicamente cuatro:

1. decidir la capacidad del mailbox (`MB_PRINT_SLOTS`), o el número de mensajes que puede acumular.
2. declarar un array de objetos del tipo que se quiere comunicar,
3. declarar un "memory pool" sobre este array,
4. declarar el mailbox

Veamos primero el código del módulo **config.h**

```
//-------------------------------------------------------------------------
// Número de slots disponibles en el Mailbox/memory pool.
const size_t MB_PRINT_SLOTS = 10;

// Tipo para el los slots del memory pool.
typedef struct
{
  char* name;
  int msg;
  systime_t time_i;
} printerMsg_t;

// Array de objetos de tipo printerMsg_t
printerMsg_t printerPool[MB_PRINT_SLOTS];

// Declaramos el banco de memoria o Memory pool.
MEMORYPOOL_DECL(printerMemPool, sizeof(printerMsg_t), PORT_NATURAL_ALIGN, NULL);

//-------------------------------------------------------------------------
// Slots de mensajes del mailbox.  Aloja punteros a los objetos del banco
// de memoria.
msg_t letter[MB_PRINT_SLOTS];

// Mailbox queue structure.
MAILBOX_DECL(printer_mailbox, &letter, MB_PRINT_SLOTS);
```

```
//------------------------------------------------------------------------------
// Prioridades de las hebras
#define WORK_THD_PRIO       (NORMALPRIO + 1)
#define PRINTER_THD_PRIO    (NORMALPRIO)

//------------------------------------------------------------------------------
// Declaramos el tamaño de los stacks de las hebras
// Estas macros deben declararse aquí o en la pestaña principal
static THD_WORKING_AREA(wa_PrinterThread, 512);
static THD_WORKING_AREA(wa_WorkThread_1, 512);
static THD_WORKING_AREA(wa_WorkThread_2, 512);
static THD_WORKING_AREA(wa_WorkThread_3, 512);

//------------------------------------------------------------------------------
// Puerto de depuración
#define debugPort SerialUSB
```

El módulo **worker.ino** incluye la declaración de las hebras de trabajo o "worker". Nótese que la creación de un mensaje implica obtener primero un objeto tipo `printerMsg_t` del pool de memoria `printerMemPool`, vinculado al mailbox de la hebra printer. El envío del mensaje se realiza mediante la función `chMBPostTimeout()`, que se invoca de forma no bloqueante (TIME_IMMEDIATE). El éxito de esta operación se verifica inspeccionando el valor devuelto. Típicamente, esta operación solo puede fallar si el mailbox está lleno, es decir, si se han consumido todos los slots y no dispone de slots libres. Si el tercer parámetro de `chMBPostTimeout()`se cambia a TIME_INFINITE la llamada se vuelve bloqueante, lo que en general implica esperar a que se libere algún slot del mailbox.

```
// ------------------------------------------------------------------------
// WorkThread - Elabora mensaje que envía a la hebra Printer usando un
//              mailbox.
//              Prioridad: WORK_THD_PRIO
//              STACK: 512 bytes
// ------------------------------------------------------------------------
static THD_FUNCTION(WorkThread, name)
{
  int msg = 0;

  while (true) {

    // Get object from memory pool.
    printerMsg_t* p = (printerMsg_t *)chPoolAlloc(&printerMemPool);
    if (!p) {
      debugPort.print((char*)name);
      debugPort.println(": chPoolAlloc failed");
```

```
    }
    else {
      // Form message.
      p->name = (char*)name;
      p->msg = msg++;
      p->time_i = chVTGetSystemTime();

      // Send message.
      msg_t s = chMBPostTimeout(&printer_mailbox, (msg_t)p, TIME_IMMEDIATE);
      if (s != MSG_OK) {
        debugPort.print((char*)name);
        debugPort.println(": chMBPost failed");
      }
    }
    chThdSleepMilliseconds(1000);

  }
}
```

El módulo printer.ino contiene la definición de la hebra “printer”. Nótese que esta hebra ejecuta una llamada bloqueante a `chMBFetchTimeout()` para obtener un nuevo mensaje de tipo `printerMsg_t`. Si tiene éxito, organiza su impresión por pantalla para finalmente liberar el objeto al pool de memoria. No realizar este último paso impedirá que se puedan alojar nuevos mensajes una vez se consuma la capacidad inicial. Nótese también el uso de la macro `TIME_I2MS(ticks)`para convertir ticks (unidades internas de conteo de tiempo) a milisegundos (ver otras macros similares en chtime.h).

```
// ------------------------------------------------------------------------
// PrinterThread - Imprime los mensajes recibidos en el mailbox
//                 Prioridad: PRINTING_THD_PRIO
//                 STACK:256 bytes
// ------------------------------------------------------------------------
static THD_FUNCTION(PrinterThread , arg)
{
  (void)arg;

  while (true) {
    printerMsg_t *p = 0;

    // Get mail.
    msg_t res = chMBFetchTimeout(&printer_mailbox, (msg_t*)&p, TIME_INFINITE);
    if (res == MSG_OK) {
      debugPort.print(p->name);
      debugPort.write(' ');
      debugPort.print(TIME_I2MS(p->time_i));
      debugPort.print(" ms ");
      debugPort.println(p->msg);

      // Put memory back into pool.
      chPoolFree(&printerMemPool, p);
    }
    else {
      debugPort.println("printer: chMBFetchTimeout() failed");
    }
  }
}
```

Por último, el sketch principal es el encargado de organizar el lanzamiento del núcleo de ChibiOS y crear las cuatro hebras del ejemplo, tres hebras de tipo “worker” y una hebra “printer”.

```
/* ------------------------------------------------------------------------
 *  Ejemplo ChibiOS-2
 *    Este ejemplo muestra cómo usar los mecanismos de comunicación
 *    entre hebras (mailboxes) disponibles en ChibiOS.
 *
 *    Requiere el uso de la librería ChRt de Bill Greiman
 *      https://github.com/greiman/ChRt
 *  Asignatura (GII-IoT)
 * ------------------------------------------------------------------------
 */

#include <ChRt.h>
#include "config.h"
```

```
/ ------------------------------------------------------------------------
// Creamos tres hebras
// ------------------------------------------------------------------------
void chSetup()
{
  // Aquí asumimos que el valor de CH_CFG_ST_TIMEDELTA es siempre cero
  // para cualquier tarjeta basada en SAMD que están soportadas solo en
  // "tick mode"

  // Verificamos primero la configuración de ChibiOS es compatible
  // con un esquema no cooperativo, verificando el valor de CH_CFG_TIME_QUANTUM
  if (CH_CFG_TIME_QUANTUM == 0) {
    debugPort.println("You must set CH_CFG_TIME_QUANTUM to a non-zero value in");
    #if defined(__arm__)
        debugPort.print("src/<board type>/chconfig<board>.h");
    #elif defined(__AVR__)
        debugPort.print("src/avr/chconfig_avr.h");
    #endif
    debugPort.println(" to enable round-robin scheduling.");
    while (true) {}
  }

  // Vinculamos memPool con el array de objetos printerPool_t.
  chPoolLoadArray(&printerMemPool, printerPool, MB_PRINT_SLOTS);

  // Lanzamos la hebra Printer
  // Prioridad: PRINTER_THD_PRIO.
  chThdCreateStatic(wa_PrinterThread, sizeof(wa_PrinterThread),
                    PRINTER_THD_PRIO, PrinterThread, NULL);

  // Lanzamos tres hebras de trabajo (Worker)
  // Prioridad: WORK_THD_PRIO
  char name[3][10] = {"Worker_1", "Worker_2", "Worker_3"};

  chThdCreateStatic(wa_WorkThread_1, sizeof(wa_WorkThread_1),
                    WORK_THD_PRIO, WorkThread, name[0]);

  chThdCreateStatic(wa_WorkThread_2, sizeof(wa_WorkThread_2),
                    WORK_THD_PRIO, WorkThread, name[1]);

  chThdCreateStatic(wa_WorkThread_3, sizeof(wa_WorkThread_3),
                       WORK_THD_PRIO, WorkThread, name[2]);
}
```

```
// ------------------------------------------------------------------
// La función setup puede usarse para activar los dispositivos que vayan
// a emplearse
// ------------------------------------------------------------------
void setup()
{
  debugPort.begin(115200);

  // Wait for USB Serial.
  while (!debugPort) {}

  // chBegin() resetea el stack y nunca retorna.
  chBegin(chSetup);
}

// ------------------------------------------------------------------
// El entorno Arduino exige la declaración de esta función que se trata
// como una hebra de prioridad normal (NORMALPRIO)
//
// NUNCA usar delay() para imponer un retardo. Si lo hacemos, impediremos
// la ejecución de las hebras de menor prioridad. Esto es fácil de comprobar
// en este ejemplo sustituyendo la llamada a chThdSleepMilliseconds() por
// delay(). Usar siempre chThdSleepMilliseconds() o alguna función
// equivalente
// ------------------------------------------------------------------
void loop()
{
  debugPort.println("Loop");
  chThdSleepMilliseconds(5000);
  //delay(5000);
}
```

Ejemplo 3: Estimación de la carga computacional

El siguiente ejemplo muestra cómo es posible monitorizar en ChibiOS la carga computacional de una aplicación multienhebrada. El contexto de la aplicación que sirve de contexto para este ejemplo se compone de cinco hebras operando en base a un esquema round-robin con prioridades basado en rodajas de tiempo o “ticks”. La hebra “top” es la de máxima prioridad (NORMALPRIO + 2), que se ejecuta con un periodo de CYCLE_MS milisegundos, es la encargada de evaluar la carga de la aplicación. La carga computacional se origina, en su gran mayoría, por la ejecución de tres hebras idénticas denominadas “workers”. La carga computacional se puede ajustar para cada una de estas hebras ajustando los valores intermedios (1, 2 y 3) que se incluyen en el vector threadLoad[], de inicio fijado a 50. Este número indica las iteraciones que se realizan en cada ciclo en las hebras “workers”. Evidentemente, si aumentamos estos números incrementaremos la carga computacional inducida por estas hebras.

El segundo aspecto configurable es el periodo (en milisegundos) con el que se ejecutan estas hebras, ajustable a través de los valores declarados, posiciones 1, 2 y 3, en el vector threadPeriod_ms[]. En su configuración inicial la hebra 1 y 3 se ejecutan a 5Hz, cada 200 ms, mientras que la segunda hebra tiene una tasa de repetición de 10 veces por segundo (10 Hz o un período de 100 ms).

Algunos detalles a notar:

- Es fundamental cambiar los valores de dos parámetros de configuración de ChibiOS editando el fichero ChRt/src/rt/templates/chconf.h para fijar TRUE los parámetros: CH_DBG_THREADS_PROFILING y CH_CFG_NO_IDLE_THREAD. De otra forma, el código proporcionado no compilará.
- La primera opción activa las opciones que permiten medir la carga computacional de cada hebra mediante la función chThdGetTicksX(), que devuelve el número de ticks, como medida interna de tiempo, consumidos durante la ejecución de cada hebra.
- La segunda opción, CH_CFG_NO_IDLE_THREAD, impide que se lance la hebra “Idle”, una hebra de la mínima pripridad que, en general, no realiza ninguna tarea. Por esta hebra, por ser la de mínima prioridad, sólo se ejecuta si ninguna otra hebra está demandando tiempo de CPU. Es consecuencia, el número de ticks que acumule será una medida directa del tiempo que la CPU está ociosa o sin carga. En nuestro caso, al impedir que se lance esta hebra, pasamos a usar la función loop() como hebra idle.
- Usar reales de precisión simple o doble tiene un impacto importante en este micro, como puede verificarse ajustando la macro USE_DOUBLE. Este es detalle a tener en cuenta si el microcontrolador dispone de una FPU, Floating Processing Unit, que soporte tipos de doble precisión real. La librería matemática incorpora variantes de tipo float, de simple precisión, para todas las funciones matemáticas incluidas en la librería. Son fácilmente identificables por el sufijo f en el nombre de la función.
- Nótese el uso de un semáforo binario como temporizador, chBSemWaitTimeout(), en la hebra top. Perfectamente podría haber empleado una llamada a chThdSleepMilliseconds() o similar. Sin embargo, usar chBSemWaitTimeout() permite abortar desde otra hebra la espera en el semáforo y así “despertar” esta hebra.

```
/* ========================================================================== *\
 *  Ejemplo ChibiOS-3
 *  Este ejemplo muestra cómo estimar la carga computacional de cada hebra
 *
 *  IMPORTANTE: en ChRt/src/rt/templates/chconf.h
 *    - CH_DBG_THREADS_PROFILING debe activarse (TRUE)
 *    - CH_CFG_NO_IDLE_THREAD debe activarse (TRUE)
 *
 *  Requiere el uso de la librería ChRt de Bill Greiman
 *    https://github.com/greiman/ChRt
```

```
 *  Asignatura (GII-IoT)
\* ================================================================== */
#include <ChRt.h>
#include <math.h>

//--------------------------------------------------------------------------
// Parametrization
//--------------------------------------------------------------------------
#define USE_DOUBLE    FALSE   // Change to TRUE to use double precision (heavier)

#define CYCLE_MS      1000
#define NUM_THREADS   5  // Three working threads + loadEstimator (top) +
                         // loop (as the idle thread)
                         // TOP thread is thread with id 0

char thread_name[NUM_THREADS][15] = { "top",
                                      "worker_1", "worker_2", "worker_3",
                                      "idle" };

volatile uint32_t threadPeriod_ms[NUM_THREADS] = { CYCLE_MS, 200, 100, 200, 0 };
volatile int threadLoad[NUM_THREADS] = {0, 50, 50, 50, 0};

volatile uint32_t threadEffectivePeriod_ms[NUM_THREADS] = { 0, 0, 0, 0, 0 };
volatile uint32_t threadCycle_ms[NUM_THREADS] = { 0, 0, 0, 0, 0 };

// Struct to measure the cpu load using the ticks consumed by each thread
typedef struct {
  thread_t * thd;
  systime_t lastSampleTime_i;
  sysinterval_t lastPeriod_i;
  sysinterval_t ticksTotal;
  sysinterval_t ticksPerCycle;
  float loadPerCycle_per;
} threadLoad_t;

typedef struct {
  threadLoad_t threadLoad[NUM_THREADS];
  uint32_t idling_per;
} systemLoad_t;
```

```
systemLoad_t sysLoad;


//-------------------------------------------------------------------------
// Load estimator (top)
// High priority thread that executes periodically
//-------------------------------------------------------------------------
BSEMAPHORE_DECL(top_sem, true);
static THD_WORKING_AREA(waTop, 256);

static THD_FUNCTION(top, arg)
{
  (void)arg;
  bool ledState = LOW;

  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, ledState);

  // Initialize sysLoad struct
  memset(&sysLoad, 0, sizeof(sysLoad));

  systime_t lastTime_i = 0;
  systime_t period_i = TIME_MS2I(CYCLE_MS);

  // Reset top_sem as "taken"
  chBSemReset(&top_sem, true);

  while (!chThdShouldTerminateX()) {

    // Wait a certain amount of time
    // chThdSleepMilliseconds(CYCLE_MS);
    systime_t deadline_i = lastTime_i + period_i;
    if (deadline_i > chVTGetSystemTimeX()) {
     chBSemWaitTimeout(&top_sem, sysinterval_t(deadline_i - chVTGetSystemTimeX()));
    }

    // Accumulated ticks for this cycle
    uint32_t accumTicks = 0;
```

```
// This assumes that no other thread will accumulate ticks during this sampling
// so we can use this timestamp for all threads
lastTime_i = chVTGetSystemTimeX();
```

```
    // tid starts at 1 because we do not include this thread (top)
    for (int tid = 1; tid < NUM_THREADS; tid++) {
      threadLoad_t * thdLoad = &(sysLoad.threadLoad[tid]);
      thdLoad->lastSampleTime_i = lastTime_i;
      systime_t ticks = chThdGetTicksX(thdLoad->thd);

      thdLoad->ticksPerCycle = ticks - thdLoad->ticksTotal;
      thdLoad->ticksTotal = ticks;
      accumTicks += thdLoad->ticksPerCycle;
    }

    for (int tid = 1; tid < NUM_THREADS; tid++) {
      threadLoad_t * thdLoad = &sysLoad.threadLoad[tid];
      thdLoad->loadPerCycle_per = (100 * (float)thdLoad->ticksPerCycle)/accumTicks;

      SerialUSB.print(thread_name[tid]);
      SerialUSB.print("  ticks(last cycle): ");
      SerialUSB.print(thdLoad->ticksPerCycle);
      SerialUSB.print("  CPU(%): ");
      SerialUSB.print(thdLoad->loadPerCycle_per);
      SerialUSB.print("   Cycle duration(ms): ");
      SerialUSB.print(threadCycle_ms[tid]);
      SerialUSB.print("  period(ms): ");
      SerialUSB.println(threadEffectivePeriod_ms[tid]);
    }
    SerialUSB.println();

    // Switch the led state
    ledState = (ledState == HIGH) ? LOW : HIGH;
    digitalWrite(LED_BUILTIN, ledState);
  }
}

//------------------------------------------------------------------------------
// Worker thread executes periodically
//------------------------------------------------------------------------------
static THD_WORKING_AREA(waWorker1, 256);
static THD_WORKING_AREA(waWorker2, 256);
static THD_WORKING_AREA(waWorker3, 256);
```

```
static THD_FUNCTION(worker, arg)
{
  int worker_ID = (int)arg;
  sysinterval_t period_i = TIME_MS2I(threadPeriod_ms[worker_ID]);
  systime_t deadline_i = chVTGetSystemTimeX();
  systime_t lastBeginTime_i = 0;

  while (!chThdShouldTerminateX()) {
    systime_t beginTime_i = chVTGetSystemTimeX();
    threadEffectivePeriod_ms[worker_ID] = TIME_I2MS(beginTime_i - lastBeginTime_i);

    int niter = threadLoad[worker_ID];
    #if USE_DOUBLE
      double num = 10;
    #else
      float num = 10;
    #endif

    for (int iter = 0; iter < niter; iter++) {
      #if USE_DOUBLE
        num = exp(num) / (1 + exp(num));
      #else
        num = expf(num) / (1 + expf(num));
      #endif
    }

    deadline_i += period_i;
    lastBeginTime_i = beginTime_i;
    threadCycle_ms[worker_ID] = TIME_I2MS(chVTGetSystemTimeX() - beginTime_i);
    if (deadline_i > chVTGetSystemTimeX()) {
      chThdSleepUntil(deadline_i);
    }
  }
}

//------------------------------------------------------------------------------
// Continue setup() after chBegin() and create the two threads
//------------------------------------------------------------------------------
void chSetup()
```

```
{
  // Here we assume that CH_CFG_ST_TIMEDELTA is set to zero
  // All SAMD-based boards are only supported in "tick mode"
  // Check first if ChibiOS configuration is compatible
  // with a non-cooperative scheme checking the value of CH_CFG_TIME_QUANTUM
  if (CH_CFG_TIME_QUANTUM == 0) {
    SerialUSB.println("You must set CH_CFG_TIME_QUANTUM to a non-zero value in");
    #if defined(__arm__)
        SerialUSB.print("src/<board type>/chconfig<board>.h");
    #elif defined(__AVR__)
        SerialUSB.print("src/avr/chconfig_avr.h");
    #endif
    SerialUSB.println(" to enable round-robin scheduling.");
    while (true) {}
  }
  SerialUSB.print("CH_CFG_TIME_QUANTUM: ");
  SerialUSB.println(CH_CFG_TIME_QUANTUM);

  // Check we do not spawn the idle thread
  if (CH_CFG_NO_IDLE_THREAD == FALSE) {
    SerialUSB.println("You must set CH_CFG_NO_IDLE_THREAD to TRUE");
  }

  // Start top thread
  sysLoad.threadLoad[0].thd = chThdCreateStatic(waTop, sizeof(waTop),
    NORMALPRIO + 2, top, (void *)threadPeriod_ms[0]);

  // Start working threads.
  sysLoad.threadLoad[1].thd = chThdCreateStatic(waWorker1, sizeof(waWorker1),
    NORMALPRIO + 1, worker, (void *)1);

  sysLoad.threadLoad[2].thd = chThdCreateStatic(waWorker2, sizeof(waWorker2),
    NORMALPRIO + 1, worker, (void *)2);

  sysLoad.threadLoad[3].thd = chThdCreateStatic(waWorker3, sizeof(waWorker3),
    NORMALPRIO + 1, worker, (void *)3);

  // This thread ID
  sysLoad.threadLoad[4].thd = chThdGetSelfX();
```

```
}
```

```
//---------------------------------------------------------------------------
// setup() function
//---------------------------------------------------------------------------
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);

  SerialUSB.begin(115200);
  while(!SerialUSB) { ; }

  SerialUSB.println("Hit any key + ENTER to start ...");
  while(!SerialUSB.available()) { delay(10); }

  // Initialize OS and then call chSetup.
  // chBegin() never returns. Loop() is invoked directly from chBegin()
  chBegin(chSetup);
}

//---------------------------------------------------------------------------
// loop() function. It is considered here as the idle thread
//---------------------------------------------------------------------------
void loop() { }
```

Ejercicio propuesto

Usando el código presentado en el último ejemplo como punto de partida, se pide desarrollar un esquema de monitorización que permita balancear la carga computacional en el microprocesador y llevarla a una condición “óptima”, definida como aquella en la que la utilización del procesador no supera el 85% (15% del tiempo en la tarea idle/loop) y las tareas se ejecutan respetando el periodo establecido, pero con el máximo posible de intensidad computacional.

Veamos algunos detalles e ideas:

- A diferencia del ejemplo anterior, en el que usábamos una intensidad fija para cada hebra o tarea, definida mediante el vector threadLoad[NUM_THREADS], ahora cada tarea dispondrá de un rango ajustable de intensidad o carga computacional y será necesario declarar un valor mínimo y otro máximo.
- Se recomienda comenzar verificando que se respetan las frecuencias de operación de cada hebra de trabajo con las hebras configuradas al valor mínimo de intensidad o carga (mínimo consumo de tiempo de procesador).

- Una vez “estabilizado” el periodo/frecuencia de repetición de las hebras, la hebra top puede ajustar de forma cíclica la intensidad de carga de cada hebra y evaluar la nueva configuración verificando que:

  a) Se respetan las frecuencias de repetición de las hebras

  b) No se supera el umbral de carga del sistema

  En caso contrario, deberá reducirse la intensidad computacional de las hebras y revaluar la situación.

Extra:

- Contemplar la posibilidad de que el tiempo de procesador de cada hebra pueda incluir un cierto porcentaje de carga variable (aleatoria). En este supuesto, el margen que se ha reservado de tiempo de CPU debería servir para absorber estos “picos” momentáneos de carga. Es evidente que este supuesto exige cierta tolerancia o histéresis para distinguir sobrecargas puntuales de otras situaciones de sobrecarga permanente, entendiendo como sobrecarga aquella situación en la que el uso de procesador sobrepasa el umbral establecido (p.e. del 85%).