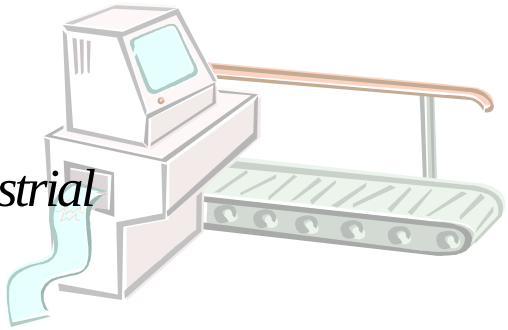
Página Web: https://web08.uah.es/
e-mail: eliseo.garcia@uah.es
Despacho: N-245
Departamento de Automática

Práctica 5

Sistemas Operativos de Tiempo Real

Laboratorio de Informática Industrial



Sistemas Operativos de Tiempo Real

Introducción

A menudo la Informática Industrial trata de la programación de pequeños sistemas informáticos en los que realizando un código con un conjunto moderado de sentencias se da solución a los requisitos que se solicitaban. Debido al éxito de los microcontroladores, en la actualidad se busca resolver mediante ellos problemas que precisan una algoritmia más complicada, o incluso que estos dispositivos sean capaces de gestionar un hardware más complejo.

Para dar solución a estos requerimientos, el desarrollador se enfrenta a la problemática de realizar la aplicación, incluyendo elementos de alto y bajo nivel. Frecuentemente la máquina no dispone de más software que el que escriba el propio programador; en otras ocasiones, se puede disponer de un conjunto de bibliotecas de funciones específicas suministradas por el fabricante de alguno de los dispositivos. Si la aplicación es demasiado compleja, necesitaremos con casi toda seguridad la ayuda de otro programa que nos ofrezca ciertas funciones que facilite el desarrollo de la aplicación. Estas funciones son realizadas por un programa llamado Sistema Operativo.

Un sistema Operativo es un conjunto de programas que dan soporte para la ejecución de aplicaciones. Dentro de los Sistemas Operativos se encuentran los Sistemas Operativos de Tiempo Real (RTOS), que añaden a los Sistemas Operativos de propósito general una restricción temporal. Así, un RTOS debe garantizar la respuesta en un estricto margen de tiempo.

Un RTOS permite separar las funciones del programa de usuario en tareas auto-contenidas e implementar una planificación bajo demanda de la ejecución de cada una de ellas. Un RTOS avanzado ofrece muchos beneficios, como:

- Planificación de tareas. Las tareas son llamadas cuando se necesitan, asegurando así un mejor flujo de programa y una mejor respuesta a eventos.
- Multitarea. La planificación de tareas ofrece la ilusión de que se están ejecutando un número de tareas simultáneamente.
- Comportamiento determinístico. Los eventos y las interrupciones se manejan en un tiempo definido
- ISR más cortas. Lo que permite un comportamiento de las interrupciones más determinístico
- Comunicación entre tareas. Gestiona la compartición de datos, memoria y recursos hardware entre distintas tareas.
- Uso definido de la pila. Cada tarea posee un espacio de pila definido, permitiendo un uso de memoria predecible.
- Gestión del sistema. Permite al programador poner toda su atención en el desarrollo de la aplicación más que en la gestión de recursos.

En la presente práctica vamos a utilizar el RTOS llamado Keil RTX. Este es un RTOS libre y determinístico diseñado para dispositivos ARM y Cortex-M. En concreto, nosotros vamos a utilizar una versión de RTX que es compatible con el estándar CMSIS, con lo que podremos utilizar sus funciones, asegurando portabilidad entre distintos procesadores Cortex-M.

RTX permite la creación de programas que simultáneamente llevan a cabo múltiples funciones (o tareas) y ayuda a crear aplicaciones que están mejor estructurados y se puedan mantener de manera más sencilla. Se pueden asignar distintas prioridades a las tareas. El núcleo RTX utiliza esas prioridades a la hora de seleccionar la siguiente tarea a ejecutar (planificación por preferencia). Además proporciona funciones adicionales para la comunicación entre tareas, la gestión de memoria y de periféricos.

Configuración de un proyecto con RTX

Antes de crear un proyecto nuevo, se debe obtener el Sistema Operativo. Keil ofrece el sistema operativo a través del enlace https://www.keil.com/demo/eval/rtx.htm.

Una vez descargado e instalado en una carpeta (nosotros, en el desarrollo de la práctica vamos a referirnos a que la localización del sistema operativo se encuentra en la carpeta C:\Keil\cmsis_rtos_rtx_v4p70\), se crea un proyecto nuevo.

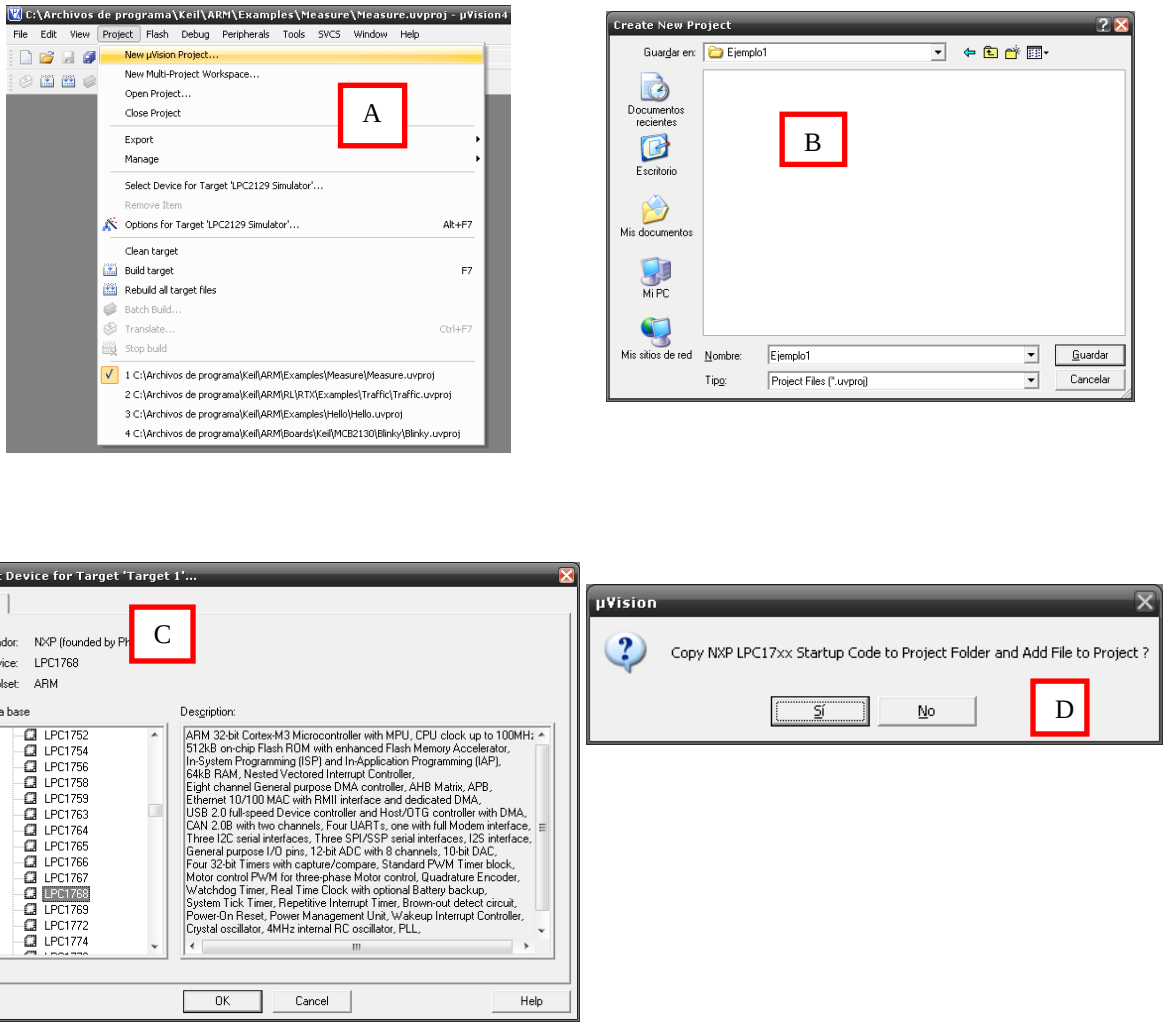


Figura 1. Ventanas para crear un nuevo proyecto, seleccionar el fabricante y el dispositivo.

Renombramos la entrada Target 1 como RTX-LPC1768 (haciendo click sobre ella, tras estar previamente seleccionada). Cambiamos también la entrada del árbol de proyecto *Source Group 1* como *StartUp*. En esta entrada de proyecto, además de contener el fichero *startup_LPC17xx.s*, incluiremos otro fichero llamado *system_LPC17xx.c* necesario para la configuración inicial del microcontrolador. Este fichero se encuentra en la ruta C:\Keil\ARM\Startup\NXP\LPC17xx\, pero es muy importante copiar este fichero desde este directorio hacia el mismo directorio en que se encuentre nuestro proyecto e incluirlo desde ahí, para así no

correr el riesgo de modificar el original. Una vez copiado, lo incluiremos situándonos sobre la carpeta *StartUp*, pulsando el botón derecho y seleccionando la opción *Add Existing Files To Group*.

A continuación vamos a crear los grupos de objetos relacionados con el Sistema Operativo. Para eso, nos situamos sobre la entrada RTX-LPC1768 del árbol del proyecto, hacemos click con el botón derecho y seleccionamos *Add Group*…. Se creará una entrada llamada *New Group* que renombraremos como *RTX Configuration*. A continuación realizamos una copia del fichero RTX_Conf_CM.c, que se encuentra en la carpeta C:\Keil\cmsis_rtos_rtx_v4p70\Templates\ al directorio donde se encuentra nuestro proyecto, y lo incluimos dentro del grupo *RTX Configuration*. Este fichero contiene los parámetros configurables del RTOS. Se recomienda visualizar el contenido de este fichero con el objetivo de conocer las posibilidades de configuración que nos ofrece, o bien consultar su edición en el manual del Sistema Operativo.

Otro elemento relacionado con el Sistema Operativo es la librería RTX_CM3.lib, que se encuentra en el directorio C:\Keil\cmsis_rtos_rtx_v4p70\lib\ARM. Este fichero contiene el código compilado del Sistema Operativo, es decir, el código que se ejecutara cuando solicitemos algún servicio del RTOS. Para añadir esta librería crearemos un nuevo grupo llamado *RTX Library*.

Finalmente, se creará una entrada llamada *New Group* que renombraremos como *SourceFiles*. En este grupo incluiremos nuestros ficheros fuente.

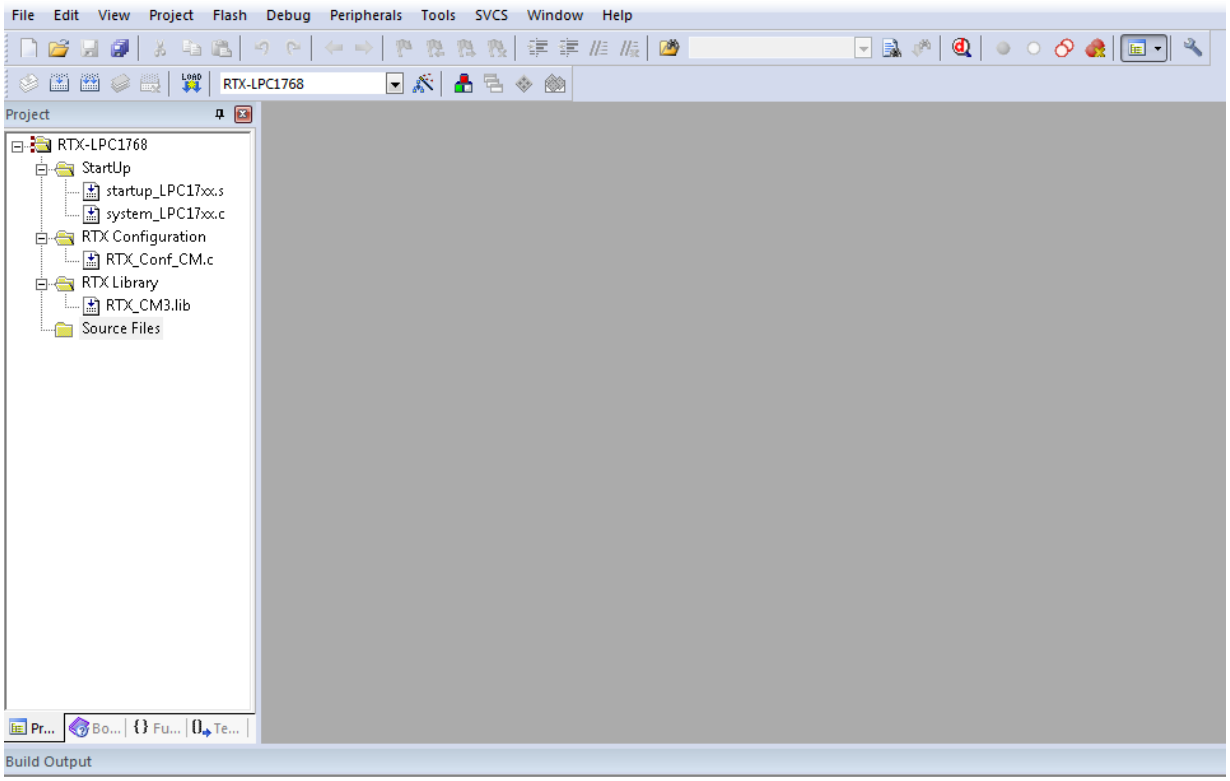


Figura 2. Configuración de los grupos del proyecto

Una vez creados los grupos, deberemos configurar el proyecto para indicarle que vamos a utilizar como Sistema Operativo el sistema RTX. Para eso, nos situamos sobre el nombre del proyecto en el árbol del proyecto y haciendo click con el botón derecho, pulsamos las Opciones del proyecto

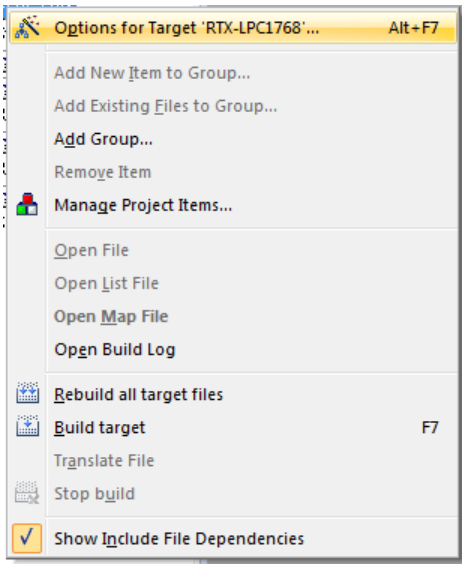


Figura 3. Menú de Opciones

En la pestaña Target, en la lista desplegable de Operating System, seleccionar la Opción RTX Kernel. Asimismo, se debe seleccionar la opción Use Microlib.

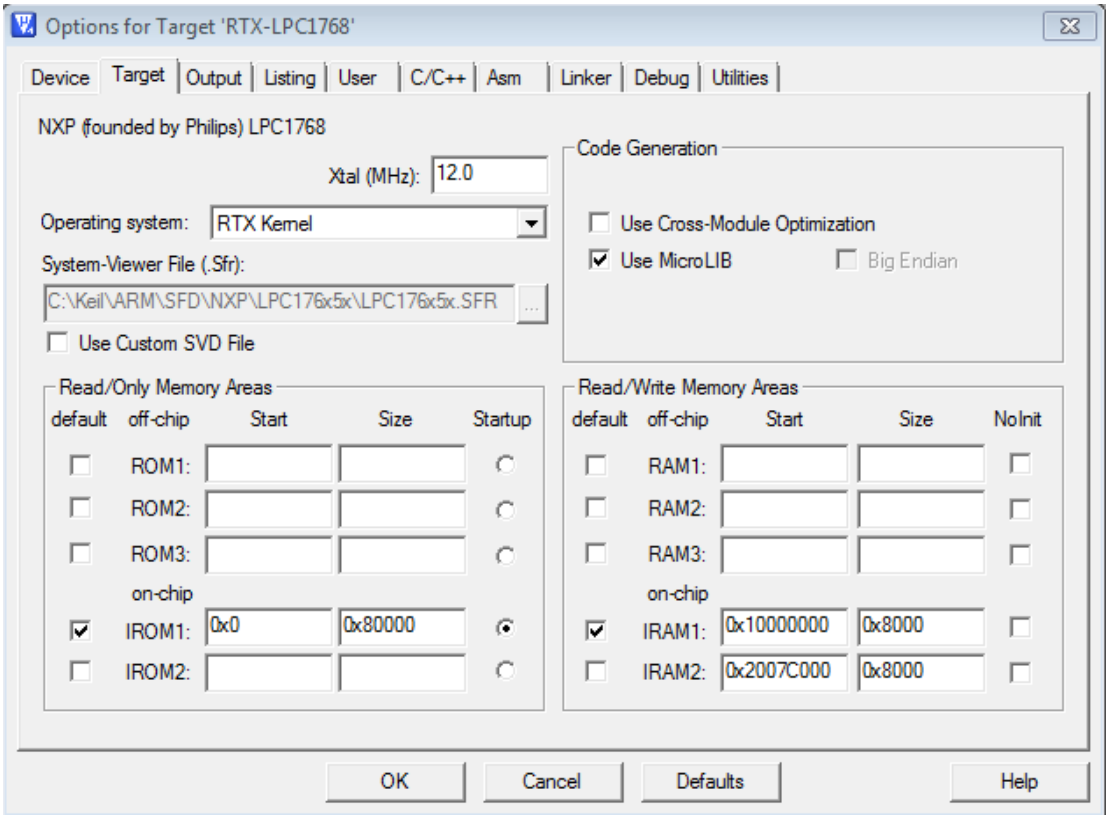


Figura 4. Configuracion de la pestaña Target

A continuación, se introduce en la pestaña C/C++, en el cuadro de dialogo Include PATHs, la ruta donde se encuentran los ficheros de inclusión d RTX (En nuestro caso, C:\Keil\cmsis_rtos_rtx_v4p70\INC)

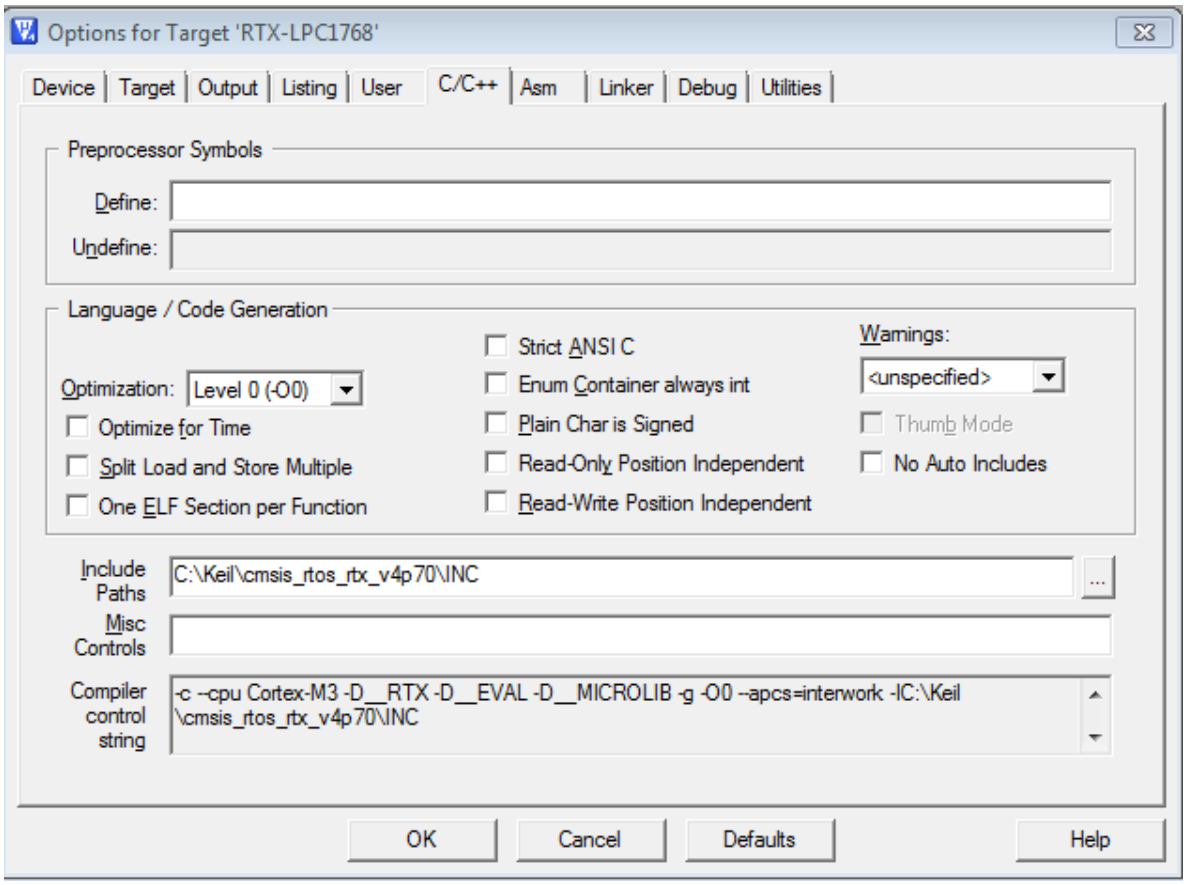


Figura 5. Configuracion de la pestaña C/C++

Con la selección de estas opciones, ya el proyecto se encuentra preparado para trabajar con el Sistema Operativo RTX. Según el resto de opciones que se quiera dotar al proyecto, se realiza la configuración pertinente.

Ejemplo de Proyecto con RTX

A continuación se muestra un ejemplo de una aplicación que se ejecuta con el Sistema Operativo RTX. En este ejemplo, se crean dos tareas, que se encargan de encender y apagar dos diodos D7 y D8 de la placa Blueboard con una frecuencia distinta. Estos diodos se encuentran conectados a los pines 1.28 y 1.29 del dispositivo LPC1768, con lo que actuando sobre ellos se consigue el parpadeo de los diodos.

El código desarrollado es el que se muestra a continuación. En él se observa que se crea de forma explícita un thread, que se encarga de gestionar el pin 1.29, mientras que el thread main gestiona el pin 1.18. La conmutación entre ambos threads la lleva a cabo el Sistema Operativo RTX.

Para añadir este código al proyecto, se debe crear un archivo nuevo main.c dentro del grupo Source Files, donde insertar las sentencias que se muestran.

```
#include "cmsis_os.h"
#include "lpc17xx.h"


/* Forward reference */
void threadX (void const *argument);

/* Thread IDs */
osThreadId main_id;
osThreadId threadX_id;

/* Thread definitions */
osThreadDef(threadX, osPriorityNormal, 1, 0);

/*----------------------------------------------------------------------------
 *   Funcion Delay
 *----------------------------------------------------------------------------*/
void delay(uint32_t n)
{
  int32_t i;
  for(i=0;i<n;i++);
}

/*----------------------------------------------------------------------------
 *   Thread X
 *----------------------------------------------------------------------------*/
void threadX (void const *argument) {
                        while(1){
                                        LPC_GPIO1->FIOPIN |= (1<<29);  // Enciendo LED
                                        delay (1000000);
                                        LPC_GPIO1->FIOPIN &= ~(1<<29); // Apago LED
                                        delay (1000000);
                        }
}

/*----------------------------------------------------------------------------
 *   Main Thread
 *----------------------------------------------------------------------------*/
int main (void) {

  /* Get main thread ID */
  main_id = osThreadGetId();

              LPC_GPIO1->FIODIR |= (1<<18);             // P1.18 definido como salida
              LPC_GPIO1->FIODIR |= (1<<29);             // P1.29 definido como salida
  LPC_GPIO1->FIOCLR |= (1<<18);          // P1.18 apagado
  LPC_GPIO1->FIOCLR |= (1<<29);          // P1.29 apagado

                /* Create thread X */
  threadX_id = osThreadCreate(osThread(threadX), NULL);

  while(1) {
     LPC_GPIO1->FIOSET = (1<<18);   // Enciendo LED
     delay (5000000);
     LPC_GPIO1->FIOCLR = (1<<18);   // Apago LED
     delay (5000000);
                                  }
}
/*----------------------------------------------------------------------------
 * end of file
 *----------------------------------------------------------------------------*/
```

Sincronización

Uno de los elementos fundamentales en los Sistemas Operativos que ofrecen multitarea son los mecanismos para gestionar los distintos threads que están activos en el sistema, comúnmente conocido como mecanismos de sincronización. El sistema Operativo RTX ofrece tres elementos de sincronización entre hilos, que son mutexes, semáforos y señales.

Los hilos de una aplicación pueden trabajar para conseguir un objetivo común. En tales circunstancias deben ser sincronizados, para obtener los resultados de forma ordenada, compartir recursos, evitar condiciones de carrera, etc.

Los objetos de sincronización son los mecanismos que se ofrecen por el núcleo o por una biblioteca de funciones para permitir la sincronización entre hilos. Tales objetos permiten controlar el estado de ejecución de los hilos los cuales podrán ser detenidos y reiniciados, comenzar y/o terminar a la par, excluir sus ejecuciones entre sí, etc.

Como herramienta de sincronización clásica contamos con el *semáforo*, que permite el control de la ejecución en exclusión mutua, es decir restringe la ejecución de ciertos hilos cuando se encuentren en conflicto, y les permite continuar cuando desaparezca el peligro. Se trata de una variable que se maneja como un entero que implementa un contador numérico sobre el que se pueden realizar al menos dos operaciones: incrementar y decrementar. Por razones históricas también se conoce a estas operaciones como *V* y *P,* respectivamente.

Para cualquier valor de cuenta mayor que 0, la operación de incremento suma 1, y la de decremento resta 1. Si estando el contador en 0 un hilo intentara decrementarlo, sería bloqueado. Para poder desbloquearlo otro hilo debería realizar la operación de incremento. Por tanto cuando existan hilos bloqueados, la operación de incremento tan solo desbloqueará a uno de ellos.

Una aplicación básica es la limitación del número de hilos que pueden estar accediendo simultáneamente a un mismo *recurso compartido*. Para ello se cargaría un valor inicial en el semáforo con el número máximo de hilos que pueden acceder al recurso simultáneamente. Cada hilo que quisiera utilizar el recurso previamente haría la operación de decremento. Cuando hubieran terminado de usarlo, incrementarían el contador.

Los tipos de datos asociados a los semáforos que ofrece el RTOS RTX son las siguientes:

- osSemaphoreDef(osSemaphoreDef) //Crea una definición de tipo Semáforo
- osSemaphoreId semaphore_id // Declara un semáforo

Las funciones que RTX proporciona para manejar semáforos son las mostradas a continuación:

- osSemaphoreId osSemaphoreCreate (const osSemaphoreDef_t *semaphore_def, int32_t count); // Crea e inicializa un objeto semáforo para la gestión de recursos
- int32_t osSemaphoreWait (osSemaphoreId semaphore_id, uint32_t millisec); // Espera hasta que el semáforo se encuentre disponible.
- osStatus osSemaphoreRelease (osSemaphoreId semaphore_id); // Libera un semaforo.
- osStatus osSemaphoreDelete (osSemaphoreId semaphore_id); //Borra un semáforo que fue creado mediante la función osSemaphoreCreate.

Un *mutex* es asimilable a un semáforo binario. Se usan habitualmente para proteger la entrada a una sección crítica. En la mayoría de las ocasiones, es suficiente con este tipo de objeto de sincronización, y dado que su uso es muy sencillo, es uno de los más utilizados. Los tipos de datos asociados a los mutex que ofrece el RTOS RTX son las siguientes:

```
osMutexId  mutex;   //Declara una variable de tipo mutex
```

OsMutexDef(mutex_def); //Crea una definición de tipo mutex.

Las funciones que RTX proporciona para manejar mutexes son las mostradas a continuación:

- osMutexId osMutexCreate (const osMutexDef_t *mutex_def); // Crea e Inicializa un mutex.
- osStatus osMutexWait (osMutexId mutex_id, uint32_t millisec); // Espera hasta que el mutex se encuentre disponible.
- osStatus osMutexRelease (osMutexId mutex_id); // Libera un mutex que fue obtenido mediante una llamada a osMutexWait.
- osStatus osMutexDelete (osMutexId mutex_id); // Elimina un mutex que fue creado mediante osMutexCreate.

Otro método de sincronización entre threads que ofrece RTX son las señales. Mediante este objeto de sincronización, se permite que un thread se encuentre detenido hasta que otro thread le envié una señal que lo haga reanudar su ejecución. Las funciones que RTX proporciona para manejar mutexes son las mostradas a continuación:

- int32_t osSignalSet (osThreadId thread_id, int32_t signals); //Establece los flags de señal específicos de un thread active.
- int32_t osSignalClear (osThreadId thread_id, int32_t signals); //Borra los flags de señal específicos de un thread activo.
- os_InRegs osEvent osSignalWait (int32_t signals, uint32_t millisec); //Espera a que uno o más flags de señal sean señalizados por el thread que se encuentra en ejecución actualmente

.

Buffer FIFO Circular

La programación de sistemas obliga a mantener diferentes tipos de datos agrupados por sus características. Una de estas agrupaciones de datos muy utilizada son los buffers FIFO (First In First Out), que pueden consistir en un simple array de datos, al que se le agregan unas reglas de acceso, de tal forma que el primer dato que se extraiga del array sea el primer dato que se introdujo. Es decir, se respeta el orden de entrada. Esas reglas de acceso se pueden implementar mediante funciones que deben ser las únicas que permiten el acceso al array. Tanto para la inserción como para la extracción de los elementos de un buffer FIFO resulta necesario mantener un índice que apunte al siguiente elemento a ser escrito, y al siguiente elemento a ser leído respectivamente. Además, resulta útil conocer la cantidad de elementos que se encuentran disponibles en el array.

Dado el tamaño finito que deben tener los arrays, muchas veces resulta interesante que cuando se completen todos sus elementos, el siguiente elemento a introducir se almacene en la primera posición del array, siempre que este elemento ya haya sido extraído. Consiguiendo este tipo de acceso, tendríamos la ilusión de que el recorrido del array tiene una forma circular, ya que al llegar al último elemento volvemos al inicio y se continúa recorriendo.

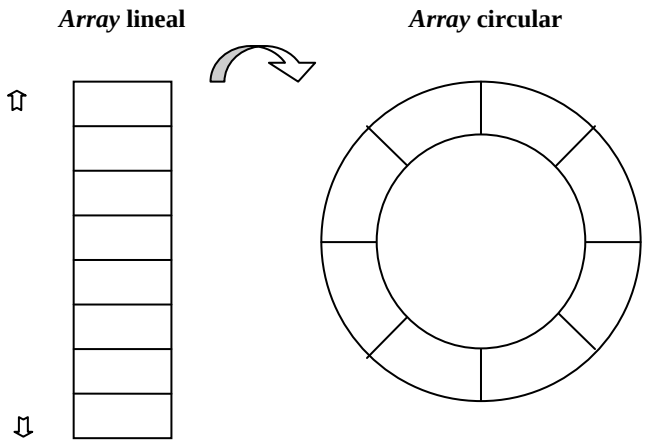


Figura 6 *– Correspondencia entre un array lineal y un array*

En la siguiente figura se muestra un ejemplo de acceso a un array FIFO circular, en el cual se introducen tres elementos y luego se extraen dos de ellos según el orden de llegada. Nótese que la flecha sólida apunta a la posición en la que se va a escribir el siguiente elemento y la flecha hueca a la posición que se va a leer.

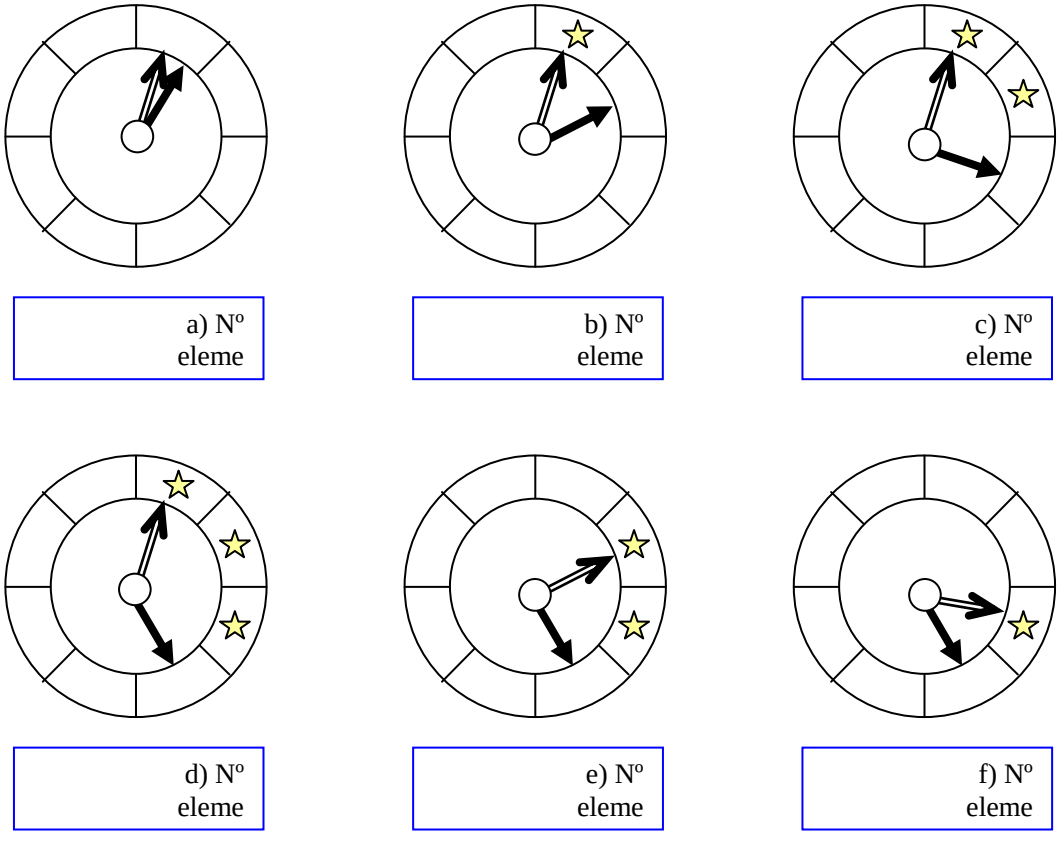


Figura 7*– Buffer FIFO mediante array*

A continuación se muestra unas funciones implementadas para la gestión del array buffer como si fuera un buffer FIFO circular. Si se persigue este fin, cualquier acceso al array buffer se debería hacer exclusivamente utilizando estas funciones. Analícese el código para comprobar que se cumplen los requisitos.

```
#ifndef TAM_FIFO
#define TAM_FIFO 25
#endif
#ifndef TIPO
#define TIPO char
#endif

static TIPO *pMete;    /* Puntero a donde meter el siguiente */
static TIPO *pSaca;    /* Puntero a donde sacar el siguiente */
                  /* FIFO estará vacío si pMete == pSaca */
                  /* FIFO estará lleno si pMete+1 == pSaca */
static TIPO buffer[TAM_FIFO];     /* Los datos del FIFO alojados estáticamente */

/*-----------------------IniciaFifo----------------------------
  Vaciar el buffer fifo
  Entradas: ninguna
  Salidas: ninguna */
void IniciaFifo(void)
{
 pMete = pSaca = &buffer[0];   /* Vacío si pMete == pSaca */
}

/*-----------------------MeteFifo----------------------------
  Introducir un dato al fifo
  Entradas: dato
  Salidas: cierto si quedan más elementos libres */
int MeteFifo(TIPO dato)
{
   TIPO *pTemp;

   pTemp = pMete + 1;
   if (pTemp == &buffer[TAM_FIFO]) /* se necesita envolver */
      pTemp = &buffer[0];

   if (pTemp == pSaca ){
      /* El fifo estaba lleno */
      return(0);    /* No quedan más elementos libres*/
   }
   else{
     *pMete = dato;  /* Meter el dato en el fifo */
      pMete = pTemp; /* Se actualiza el puntero */
      return(1);     /* Quedan más elementos libres */
   }
}

/*-----------------------SacaFifo----------------------------
  Extrae un dato del fifo
  Entradas: puntero a la variable donde almacenar el dato.
            NULL para desperdiciar el dato.
  Salidas: cierto si quedan más datos               */
int SacaFifo(TIPO *pDato)
{
   if (pMete == pSaca)
      return(0);     /* Se ha vaciado el fifo */

    if(pDato)
        *pDato = *pSaca;

   if (++pSaca == &buffer[TAM_FIFO])
      pSaca = &buffer[0];

   return(1); /* Quedan datos */
}
```

Practica Propuesta

En la práctica anterior el sistema diseñado adolece de una determinada falta de eficiencia. Cada vez que el sistema tiene que conmutar el estado de un diodo, o de un digito en el display de 7 segmentos, el sistema llama a la función Delay() o Retardo(), en la cual se ejecuta un bucle en el que no se hace ninguna operación. En otras palabras, se hace que el microcontrolador "pierda un tiempo" para poder conseguir los tiempos de espera adecuados. Esta situación es lo que se conoce como "espera activa", y consiste en hacer que el sistema ejecute código que no hace nada (es decir, la CPU se encuentra trabajando para hacer nada).

Durante ese tiempo que se ejecuta la función Delay() o Retardo(), el sistema seria más productivo si estuviera ejecutando código que sí tiene utilidad, por ejemplo, calculando los siguientes números primos. El problema que se nos plantea es que no sabemos a priori cuanto tiempo se puede tardar en calcular uno de ellos y, lanzar la ejecución su cálculo en lugar de a la función Delay(), puede hacer que perdamos más tiempo que el que se necesitaría esperar.

Existe una solución, y se basa en el uso de un Sistema Operativo multitarea. Así, podemos definir un thread cuya función sea calcular números primos (thread productor), y otro que se encargue de sacar los resultados (thread consumidor). La conmutación entre un thread y otro será realizada mediante el Sistema Operativo. De esta forma, se permite que el thread productor calcule números primos durante el tiempo en el que el thread consumidor no tenga que hacer ninguna variación en los datos de salida.

Aparecen una serie de problemas al adoptar esta solución. Uno de ellos es que el ritmo al que el thread productor obtiene los números primos sea superior al que el thread consumidor los va mostrando (2 segundos por número primo). En este caso, se puede utilizar un buffer FIFO circular donde el productor ira depositando temporalmente los valores y el thread consumidor los irá extrayendo a su ritmo. En esta práctica, considérese un buffer FIFO circular de 8 elementos enteros de 16 bits.

Se plantean situaciones que se deberán solucionar, como qué debe hacer el sistema cuando el consumidor va a extraer un dato y el buffer se encuentra vacío (el productor todavía no ha calculado el siguiente numero primo a mostrar). En este caso, la solución es que el consumidor se detenga hasta que haya un dato nuevo en el buffer. Otra situación es que el productor vaya a insertar un dato y en ese momento el buffer se encuentre lleno. En esta situación, la solución más adecuada es que el productor se detenga hasta que el consumidor extraiga un dato y, como consecuencia, deje un hueco libre en el array FIFO circular.

En la presente práctica se solicita la realización de una aplicación para el microcontrolador Cortex-M3 donde se implemente la funcionalidad expuesta en los párrafos anteriores.