Práctica 4. Servicios Básicos de Temporización.

1. Objetivo

El objetivo de esta práctica es que el alumno aprenda a diseñar, implementar y utilizar un conjunto de servicios básicos de temporización, configurando directamente el *hardware* que da soporte a este tipo de servicios, denominado *Timer Unit*. Para poder realizar esta práctica se trabajará sobre la *Timer Unit* integrada en el simulador TSIM2 del procesador LEON3 utilizado en prácticas anteriores, y se implementarán las estructuras de datos y funciones básicas que permiten su gestión.

2. Introducción

La *Timer Unit* integrada en el simulador TSIM2 del procesador LEON3 cuenta con dos *timers* que pueden programarse para trabajar, bien de forma independiente, o en cascada (la configuración en cascada está deshabilitada para la versión de evaluación utilizada en el laboratorio).

Una señal de reloj (*clk*) controla la etapa de pre-escalado de la *Timer Unit* destinada a fijar la frecuencia base a la que los *timers* van a trabajar. Cada pulso de este reloj decrementará en una unidad el valor del registro *prescaler*. Cuando este registro alcanza el valor cero y llega un nuevo pulso de reloj (situación de *underflow*) se generá un pulso de salida y se carga de nuevo en *prescaler* el valor almacenado en *prescaler reload*. El pulso de salida del *prescaler* es tomado como entrada por los *timers*, decrementando sus registros *timer value*. De esta forma, el valor almacenado en *prescaler reload* permite configurar la frecuencia base a la que trabajan los *timers* como una división de la frecuencia de la señal de reloj, según la siguiente ecuación:

$$frecuencia\_base\_timers = frecuencia\_clk / (prescaler\ reload + 1)$$

La siguiente figura muestra un esquema de la *Timer Unit*, donde se puede ver la función de la etapa de pre-escalado y su conexión con los dos *timers*.

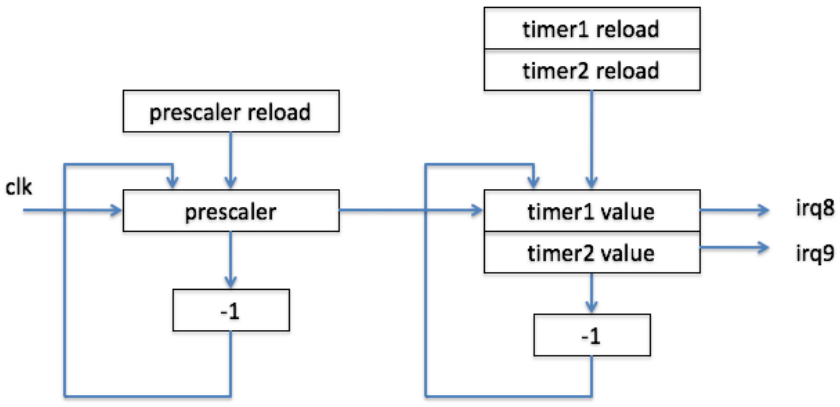


Figura 1. Esquema general de la *Timer Unit*

Los dos *timers* presentan, en su configuración por defecto, un comportamiento análogo al de la etapa de pre-escaler: el registro *timer value* decrementa su valor cada vez que recibe un pulso de entrada. Una vez que alcanza el valor cero, y tras recibir un nuevo pulso, se produce la situación de *underflow* que genera un pulso de salida. Los *timers* pueden ser configurados para que este pulso de salida dispare una interrupción. De esta forma, y mediante la instalación de un manejador de dicha interrupción, es posible **programar** la **ejecución** de una acción asociada al paso del **tiempo**.

También, y al igual que ocurría con el *prescaler*, los *timers* puede configurarse para que, tras la situación de *undeflow*, se **recargue automáticamente** un valor en los registros *timer1 value* y *timer2 value*. Los registros *timer1 reload* y *timer2 reload*, son los que, respectivamente, almacenan el valor de recarga para cada uno de los *timers*. De esta manera, y en combinación con la instalación del manejador de interrupción correspondiente, se puede **programar** la **ejecución periódica** de una acción, y a partir de esta acción, definir diferentes **servicios de temporización**.

La *Timer Unit* integrada en el SoC basado en LEON3 puede ser configurada para generar interrupciones independientes para cada *timer*, o para que ambos *timers* compartan la misma interrupción (opción deshabilitada en la versión de evaluación de TSIM2). Las interrupciones externas asociadas a los *timer1* y *timer2*, son, respectivamente, las de los niveles 8 y la 9.

Finalmente, la *Timer Unit* también permite configurar en cascada los dos *timers*, con el fin de que el pulso de entrada del *timer2* sea proporcionado por el de salida del *timer1*. Como ya se ha indicado, esta opción, disponible en la configuración comercial, está deshabilitada en la versión de evaluación de TSIM2 que se utiliza en estas prácticas.

3. Diseño de un Driver para la Timer Unit

La siguiente tabla muestra el conjunto de registros que constituyen el controlador de la *Timer Unit.*

REGISTER	APB ADDRESS
Scaler Value	0x80000300
Scaler Reload Value	0x80000304
Configuration Register	0x80000308
Timer 1 Counter Value Register	0x80000310
Timer 1 Reload Value Register	0x80000314
Timer 1 Control Register	0x80000318
Timer 2 Counter Value Register	0x80000320
Timer 2 Reload Value Register	0x80000324
Timer 2 Control Register	0x80000328

Tabla 1. Registros del controlador de la *Timer Unit*

Para manejar el controlador de la *Timer Unit*, y manipular sus registros, se van a definir un conjunto de funciones que constituirán el driver de este dispositivo. Las funciones son las siguientes:

`leon3_timerunit_set_configuration`

Función que configura la *Timer Unit*, fijando el valor del *Configuration Register* así como inicializando los registros *Scaler Value y Scaler Reload Value.* Su prototipo es el siguiente:

```
void leon3_timerunit_set_configuration (uint32_t scalerValue
                                , bool_t freeze_during_debug
                                , bool_t separate_interrupts );
```

El parámetro `scalerValue` es utilizado para inicializar los valores de los registros *Scaler Value y Scaler Reload Value*, mientras que los parámetros `freeze_during_debug y separate_interrupts` se utilizan para inicializar los campos *DF* y *SI* del registro *Configuration Register* (ver la figura 2). El campo `freeze_during_debug` determina si durante el modo depuración del procesador el timer debe de ser detenido (*freeze*) o no, mientras que el parámetro `separate_interrupts` determina si los *timers* utilizan interrupciones separadas.

9	DF		Disable Timer Freeze 0: Timer unit can be frozen during debug mode. 1: Timer unit cannot be frozen during debug mode.
8	SI	1	Separate Interrupts 0: Single interrupt for timers. 1: Each timer generates a separate interrupt. Read=1; Write=Don’t care.

Figura 2. Campos del registro *Configuration Register* a configurar por el driver

leon3_timer_config

Función que permite configurar los dos *timers* de la *Timer Unit*, fijando el valor de sus registros *Timer Control Register*, *Timer Counter Value Register y Timer Counter Reload Value Register*. Su prototipo es el siguiente:

```
void leon3_timer_config(uint8_t timerId
                        , uint32_t timerValue
                        , bool_t chain_with_prec_timer
                        , bool_t restart_timer);
```

- El parámetro `timerId` indica si se va configurar el *timer1* (`timerId==0`) o el *timer2* (`timerId==1`).
- El parametro `timerValue` inicializa el valor de los registros *Timer Counter Value y Timer Counter Reload Value*.
- El parámetro `chain_with_prec_timer` se utiliza para inicializar el campo *CH* del registro *Timer Control Register* (ver la figura 3). Este campo determina si el *timer* toma como pulso de entrada el pulso de salida del timer anterior (modo encadenado) o el pulso de salida del *prescaler* (En la version de evaluación de TSIM2 el modo encadenado está dehabilitado).
- El parámetro `restart_timer` se utilizan para inicializar el campo *RX* del registro *Timer Control Register* (ver la figura 3) que determina si tras la situación de *undeflow* el registro *Timer Counter Value* se reinicia (`restart_timer !=0`) o no (`restart_timer ==0`) con el valor de *Timer Counter Reload Value*.

Para modificar el registro *Timer Counter Value* primero se debe fijar el valor del *Timer Counter Reload Value* y después se activa el campo *LD* del *Timer Control Register*.

La siguiente figura muestra el conjunto de campos del *Timer Control Register* que controla individualmente cada Timer. Para configurar esos campos se van a utilizar un conjunto de funciones que se describen a continuación.

BIT NUMBER(S)	BIT NAME	RESET STATE	DESCRIPTION
31-7	Reserved		
6	DH		Debug Halt State of timer when DF=0. Read only. 0: Active 1: Frozen
5	CH		Chain with preceding timer. 0: Timer functions independently 1: Decrementing timer *n* begins when timer (*n*-1) underflows.
4	IP		Interrupt Pending 0: Interrupt not pending 1: Interrupt pending. Remains '1' until cleared by writing '0' to this bit.
3	IE		Interrupt Enable 0: Interrupts disabled 1: Timer underflow signals interrupt.
2	LD		Load Timer Writing a '1' to this bit loads the value from the timer reload register to the timer counter value register.
1	RS		Restart Writing a '1' to this bit reloads the timer counter value register with the value of the reload register when the timer underflows.
0	EN		Timer Enable 0: Disable 1: Enable

Figura 3. Campos del registro *Timer Configuration Register*

`leon3_timer_enable_irq y leon3_timer_disable_irq`

Funciones que habilitan y deshabilitan la interrupción asociada a la situación de *underflow* de un timer.

```
uint8_t leon3_timer_enable_irq(uint8_t timerId);
uint8_t leon3_timer_disable_irq uint8_t timerId);
```

- El parámetro `timerId` indica si la habilitación de la interrupción afecta al *timer1* (`timerId==0`) o al *timer2* (`timerId==1`). Ambas funciones modifican el campo *IE* del registro *Timer Control Register*.

`leon3_timer_enable y leon3_timer_disable`

Funciones que habilitan y deshabilitan los timers.

```
uint8_t leon3_timer_enable(uint8_t timerId);
uint8_t leon3_timer_disable(uint8_t timerId);
```

- El parámetro `timerId` indica si la habilitación afecta al *timer1* (`timerId==0`) o o al *timer2* (`timerId==1`). Ambas funciones modifican el campo *EN* del registro *Timer Control Register*.

`leon3_timer_clear_irq`

Función que borra el bit *IP* del registro *Timer Control Register*. Ese bit se pone automáticamente a 1 cuando se produce una interrupción debida a una situación de *underflow*. Debe ser borrado en el manejador de la interrupción para indicar que la interrupción ya ha sido atendida.

```
uint8_t leon3_timer_clear_irq (uint8_t timerId);
```

4. Diseño de un reloj monotónico y gestión del tiempo universal.

Uno de los servicios de temporización más básicos que se pueden implementar mediante la *Timer Unit* es un gestor del tiempo universal a través de un reloj monotónico. Un reloj monotónico es aquél que se incrementa de forma periódica durante la ejecución de un sistema. El valor que toma el reloj se puede de esta forma asociar al tiempo universal mediante una referencia. Para esta práctica se van a utilizar las siguientes funciones **(ya codificadas)** que facilitan la implementación de un reloj monotónico sobre una codificación del tiempo universal conocida como Y2K. Esta codificación emplea un entero de 32 bits para fijar los segundos que han pasado desde el 1 de enero de 2000.

`init_monotonic_clock`

Función que inicializa las variables que gestionan el reloj monotónico y fija como tiempo universal inicial una valor de referencia en formato Y2K. Su prototipo es el siguiente:

```
void init_monotonic_clock(uint32_t Y2KTimeRef);
```

`irq_handler_update_monotonic_clock`

Rutina de atención a una interrupción que actualiza el reloj monotónico. Esta rutina debe ser invocada periódicamente, para lo que se programará un timer de la Timer Unit. El periodo de invocación debe estar definido mediante la macro `TIMING_SERVICE_TICKS_PER_SECOND`. El prototipo de la función es el siguiente:

```
void irq_handler_update_monotonic_clock (void);
```

update_universal_time_Y2K

Función que permite actualizar la referencia codificada en Y2K que permite calcular el tiempo universal. El prototipo de la función es el siguiente:

```
void update_universal_time_Y2K( uint32_t Y2KTimeRef);
```

get_universal_time_Y2K

Función que devuelve el valor actual del tiempo universal codificado en Y2K. El prototipo de la función es el siguiente:

```
uint32_t get_universal_time_Y2K ();
```

print_date_time_from_Y2K

Función que imprime el valor del tiempo universal codificado en Y2K que se pasa a través del parámetro `seconds_from_y2K`. El prototipo de la función es el siguiente:

```
void print_date_time_from_Y2K(uint32_t seconds_from_y2K);
```

date_time_to_Y2K

Función que codifica en Y2K una fecha y hora pasadas por parámetro mediante los campos `day, month, year, hour, minutes y seconds`. El prototipo de la función es el siguiente:

```
uint32_t date_time_to_Y2K(uint8_t day, uint8_t month,
                          uint8_t year, uint8_t hour,
                          uint8_t minutes, uint8_t seconds);
```

5. Servicios de Temporización

Empleando el *driver* de la *Timer Unit* es posible programar un ***timer*** para que interrumpa periódicamente. Mediante una rutina de atención a esa interrupción periódica, denominada **tick del sistema**, es posible definir diferentes servicios de temporización, como el del reloj monotónico citado en el apartado anterior. En esta práctica se va a definir la siguiente función para programar uno de los *timers* de la la *Timer Unit* e inicializar los servicios de temporización que dan soporte al reloj monotónico:

init_timing_service

Función que configura un *timer* de la *Timer Unit* para programar una interrupción periódica (**tick del sistema).** La función, además, instala un manejador de usuario para dicha interrupción que permite dar soporte a diferentes servicios básicos de

temporización, como el de la gestión de un reloj monotónico. Para inicializar este servicio, la función recibe el parámetro `currentTime_in_Y2K`, que indica el valor de fecha y hora (tiempo universal) en el momento de la llamada.

```
uint8_t init_timing_service (uint32_t currentTime_in_Y2K);
```

6. Práctica 4_1: Creación de proyecto para LEON3

Creación de un nuevo proyecto denominado prac4_1 cuyo ejecutable sea para la plataforma Sparc Bare C. En ese proyecto crear dos subdirectorios **include** y **src**. En el directorio **src** añadir los archivos .c y .asm de la práctica anterior (salvo el main.c), junto con los archivos *leon3_timer_unit_drv.c*, *leon3_monotonic_clock.c*, *leon3_timing_service.c* que encontrarás en el archivo enlazado como *prac4_1_fuentes.zip* en la página web. Igualmente, añadir al directorio **include** los archivos de cabecera (.h) de la práctica anterior, junto con los archivos *leon3_timer_unit_drv.h*, *leon3_monotonic_clock.h y leon3_timing_service.h* que contiene *prac4_1_fuentes.zip*

7. Práctica 4_1: Tarea a realizar

1. Añadir el siguiente código al archivo `leon3_types.h`. Este código define la macro `NULL` para la gestión de punteros y permite definir el tipo `bool_t` de forma selectiva (utilizando el tipo `bool` si se compila un archivo C++ o el tipo `unsigned char` si se compila un archivo C).

```
#ifndef LEON3_TYPES_H_
#define LEON3_TYPES_H_

#ifndef NULL
      #define NULL  0
#endif

#ifndef __cplusplus

      typedef unsigned char         bool_t;

      #define true      1
      #define false     0

#else

      typedef bool            bool_t;

#endif

typedef unsigned char         byte_t;
...
```

2. **Completar en el archivo *leon3_timer_unit_drv.c*** el código de las siguientes funciones definidas de acuerdo a lo especificado en el apartado 3 de este guion:

   - `leon3_timer_config`
   - `leon3_timer_enable`
   - `leon3_timer_disable`
   - `leon3_timer_enable_irq`
   - `leon3_timer_disable_irq`
   - `leon3_timer_clear_irq`

3. Definir en el archivo *leon3_timing_service.h* el valor de la macro `TIMING_SERVICE_TICKS_PER_SECOND` para fijar que la interrupción periódica del **tick** de reloj tenga una frecuencia de 10 Hz

```
#define TIMING_SERVICE_TICKS_PER_SECOND        10
```

4. **Completar en el archivo *leon3_timing_service.c* el código de la función `init_timing_service`** `(uint32_t currentTime_in_Y2K)`. Para completar esta función **se deberán utilizar las siguientes macros declaradas** en ese mismo archivo:
   - `LEON3_FREQ_MHZ` frecuencia a la que trabaja el procesador en MHZ
   - `TIMER_ID` identificador del *timer* que se va a utilizar
   - `TIMER_IRQ_LEVEL` nivel de la interrupción de dicho *timer*

```
#define LEON3_FREQ_MHZ                    20
#define TIMER_ID                          0
#define TIMER_IRQ_LEVEL                   8
```

   La función debe realizar las siguientes acciones:

   1) Realizar la llamada al sistema que deshabilita todas las interrupciones.
   2) Enmascarar el nivel de interrupción del *timer* (`TIMER_IRQ_LEVEL`)
   3) Deshabilitar el *timer* con identificador `TIMER_ID` empleando `leon3_timer_disable`
   4) Deshabilitar la interrupción del *timer* con identificador `TIMER_ID` empleando `leon3_timer_disable_irq`
   5) Configurar, empleando `leon3_timerunit_set_configuration`, la *TimerUnit* para que el pulso de salida de la etapa de pre-escaler tenga una frecuencia de 1MHz, se habilite el *freeze* durante la depuración, y se generen interrupciones separadas para los dos *timers*.

```
leon3_timerunit_set_configuration(LEON3_FREQ_MHZ-1,
                                  true , true );
```

6) Configurar el *timer* con identificador `TIMER_ID`, empleando la función `leon3_timer_config`, para que se produzca un underflow cada **tick** del sistema `(1000000UL/TIMING_SERVICE_TICKS_PER_SECOND-1)`, sin encadenar los timers, y con reinicialización del *timer* tras el *underflow*.

```
leon3_timer_config(TIMER_ID,
          1000000UL/TIMING_SERVICE_TICKS_PER_SECOND-1,
            false, true);
```

7) Instalar `timertick_irq_handler` como manejador asociado al nivel de interrupción del *timer*, definido por la macro `TIMER_IRQ_LEVEL`.
8) Inicializar el reloj monotónico

```
init_monotonic_clock(currentTime_in_Y2K);
```

9) Hacer el *clear* de la interrupción del *timer* con identificador TIMER_ID usando `leon3_timer_clear_irq`
10) Desenmascarar el nivel de interrupción del *timer* (`TIMER_IRQ_LEVEL`)
11) Habilitar la interrupción del *timer* con identificador TIMER_ID empleando la función `leon3_timer_enable_irq`
12) Habilitar el *timer* con identificador `TIMER_ID` empleando la función `leon3_timer_enable`.
13) Realizar la llamada al sistema para habilitar las interrupciones finalizando la inicialización.

5. Comprobar el correcto funcionamiento de la implementación del driver de la *Timer Unit* con el siguiente programa principal que se incluirá como archivo **main.c** al proyecto.

```
#include "leon3_ev_handling.h"
#include "leon3_hw_irqs.h"

#include "leon3_uart.h"
#include "leon3_bprint.h"

#include "leon3_monotonic_clk.h"
#include "leon3_timer_unit_drv.h"
#include "leon3_timing_service.h"


int main()
{
      uint32_t aux1,aux2;
```

```
	//Instalar manejadores de traps para habilitar
	//y deshabilitar las interrupciones
	//Install trap handlers for enable and disable irqs

	leon3_set_trap_handler(0x83,  leon3_trap_handler_enable_irqs);
	leon3_set_trap_handler(0x84 , leon3_trap_handler_disable_irqs);

	//Inicializar servicio de temporización con el tiempo
	//universal actual
	//Init Timing Service with current universal time

	init_timing_service(date_time_to_Y2K(18, 3, 22, 0, 0, 0 ));

	while(1){

		//Mostrar tiempo con un intervalo de 10 segundos.
		//Print time with an interval of 10 seconds

		aux1=get_universal_time_Y2K();
		if(((aux1%10)==0)&& aux1!=aux2){

			print_date_time_from_Y2K(aux1);

			aux2=aux1;
		}

	}
	return 0;
}
```

La salida correcta del programa debe ser un mensaje de hora cada 10 segundos.

```
Fecha 18|3|2022 Hora 0:0:0
Fecha 18|3|2022 Hora 0:0:10
Fecha 18|3|2022 Hora 0:0:20
Fecha 18|3|2022 Hora 0:0:30
Fecha 18|3|2022 Hora 0:0:40
Fecha 18|3|2022 Hora 0:0:50
Fecha 18|3|2022 Hora 0:1:0
Fecha 18|3|2022 Hora 0:1:10
Fecha 18|3|2022 Hora 0:1:20
Fecha 18|3|2022 Hora 0:1:30
```

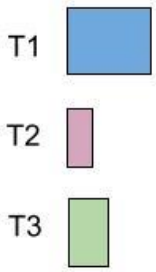
8. Servicios de temporización para la implementación de un ejecutivo cíclico

Utilizando la interrupción del **tick del sistema** es posible implementar, además del reloj monotónico visto en el apartado anterior, otros servicios de temporización. En esta parte de la práctica veremos cómo el **tick del sistema** nos va a permitir proporcionar servicios destinados a la construcción de un *ejecutivo cíclico*, es decir, un programa que periódicamente repiten una secuencia de ejecución de tareas. Los *ejecutivos cíclicos* tienen un período básico, que controla el inicio de la ejecución de una secuencia específica

de tareas, y un híper-período, que es el número de períodos básicos a partir del cual se vuelve a repetir la secuencia completa de ejecución de las tareas del *ejecutivo cíclico*.

En la siguiente figura se muestra un ejemplo de un *ejecutivo cíclico*, en el que se controla la ejecución periódica de 3 tareas cuyos periodos y tiempos de ejecución están determinados en la tabla adjunta.

Task	Execution Time (sec)	Period (sec)
T1	1	2
T2	0.3	3
T3	0.5	4



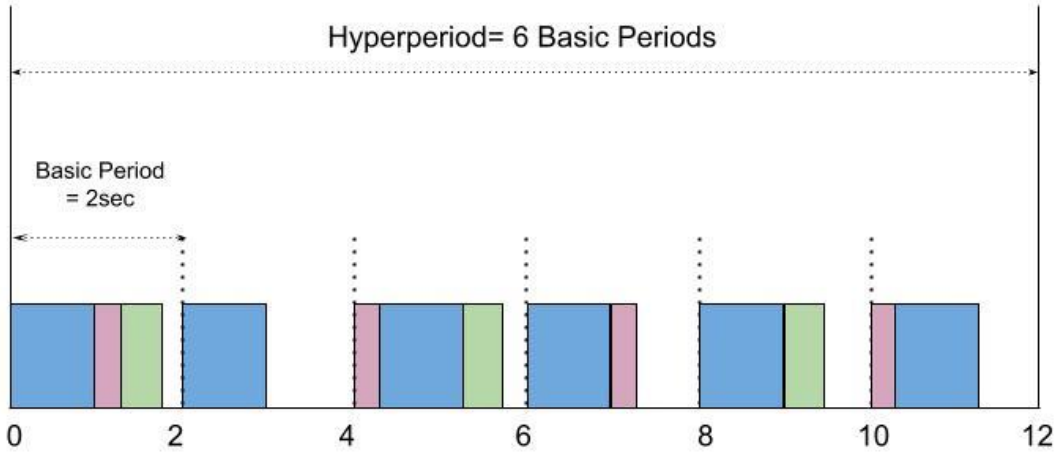


Figura 4. Ejemplo de ejecutivo cíclico

El *ejecutivo cíclico* tiene un período básico de 2 segundos, mientras que el hiperperíodo es de 6 periodos básicos.

Con el fin de poder implementar, tanto éste como otros ejecutivos cíclicos, se van a ampliar los servicios de temporización implementados en el módulo *leon3_timing_service*. Esta ampliación consta, en primer lugar, en añadir o modificar los siguientes elementos al archivo *leon3_timing_service.c*

TickCounterFromReset

La variable global `TickCounterFromReset` se añade a las variables declaradas con el fin de poder gestionar un contador de **ticks** desde el último reset del sistema.

```
static uint64_t TickCounterFromReset=0;
```

timertick_irq_handler

La función `timertick_irq_handler`, instalada como rutina de atención asociada al **tick del sistema**, se modifica, tal como se muestra a continuación, para incrementar en una unidad la variable `TickCounterFromReset` cada vez que se produzca la interrupción **del tick del sistema**.

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

void timertick_irq_handler (void){

    leon3_timer_clear_irq(TIMER_ID);

    TickCounterFromReset++;

    irq_handler_update_monotonic_clock();

}
```

get_tick_counter_from_reset

Para poder consultar el valor de esta variable, se añade, también al archivo *leon3_timing_service.c,* la definición de la función `get_tick_counter_from_reset`. Esta función, devuelve el valor de la variable global `TickCounterFromReset`, manejando adecuadamente la máscara de interrupción del *timer* con el fin de evitar posibles condiciones de carrera.

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

uint64_t get_tick_counter_from_reset(void){

    uint64_t tick_counter_from_reset;

    //Enmascaro la interrupción del timer para evitar condiciones de carrera.
    //Mask the timer interrupt to avoid race condition
     leon3_mask_irq(TIMER_IRQ_LEVEL);

     tick_counter_from_reset=TickCounterFromReset;

     //desenmascaro para retornar a la situación previa
     //unmask to restore previous situation
     leon3_unmask_irq(TIMER_IRQ_LEVEL);

    return tick_counter_from_reset;
```

```
}
```

wait_until

Finalmente, y con el fin de poder sincronizar la ejecución de las tareas con el inicio de cada período básico del ejecutivo cíclico, se implementa como parte de los servicios de temporización la función `wait_until`. Esta función utiliza el valor recibido como parámetro (`ticksFromReset`), para realizar una espera activa hasta que este valor coincide con el valor retornado por la función `get_tick_counter_from_reset` ya implementada.

```
//*****************************************************************
uint8_t wait_until(uint64_t ticks_from_reset){

    uint8_t error=0;

    uint64_t tick_counter_from_reset;

    tick_counter_from_reset=get_tick_counter_from_reset();

    if(ticks_from_reset <tick_counter_from_reset)
        error=1;
    else
        while(tick_counter_from_reset< ticks_from_reset)
              tick_counter_from_reset=get_tick_counter_from_reset();

    return error;

}
```

La ampliación, se completa, además, con la declaración de las funciones `get_tick_counter_from_reset y wait_until` en el archivo *leon3_timing_service.h,*

```
uint64_t get_tick_counter_from_reset(void);

uint8_t wait_until(uint64_t ticks_from_reset);
```

9. Práctica 4_2: Creación de proyecto para LEON3

Creación de un nuevo proyecto denominado prac4_2 cuyo ejecutable sea para la plataforma Sparc Bare C. En ese proyecto crear dos subdirectorios **include** y **src** donde se deben añadir todos los archivos del proyecto prac4_1 **salvo el que contiene la función `main`.**

10. Práctica 4_2: Tarea a realizar

1. Modificar los archivos *leon3_timing_service.h* y *leon3_timing_service.c* de acuerdo a lo expuesto en el apartado 8.

2. Comprobar el funcionamiento de estas funciones mediante el siguiente programa principal que se incluirá como archivo **main.c** al proyecto. Este programa implementa el ejecutivo cíclico descrito en la figura 4. Las tres tareas Task1, Task2 y Task3 utilizan la función `emu_execution_time` (ya implementada) para emular su tiempo de ejecución, en milisegundos, de acuerdo a lo especificado en la tabla de dicha figura.

```
#include "leon3_ev_handling.h"
#include "leon3_hw_irqs.h"

#include "leon3_uart.h"
#include "leon3_bprint.h"

#include "leon3_monotonic_clk.h"
#include "leon3_timer_unit_drv.h"
#include "leon3_timing_service.h"


// Código de la tarea T1
void Task1(void){

    leon3_print_string(" Start T1\n");

    emu_execution_time(1000); //execution time 1000 ms

    leon3_putchar('\n');

    leon3_print_string(" End T1\n");

}

// Código de la tarea T2
void Task2(void){


      leon3_print_string(" Start T2\n");

      emu_execution_time(300);  //execution time 300 ms

       leon3_putchar('\n');

       leon3_print_string(" End T2\n");
}

// Código de la tarea T3
void Task3(void){

       leon3_print_string(" Start T3\n");
```

```
        emu_execution_time(500);  //execution time 500 ms

        leon3_putchar('\n');

        leon3_print_string(" End T3\n");

}

// Definición de las macros de configuración
#define CYClIC_EXECUTIVE_PERIOD_IN_TICKS            20
#define CYClIC_EXECUTIVE_HYPER_PERIOD               6
#define CYClIC_EXECUTIVE_TASKS_NUMBER               3

// Array bidimensional para definir la secuencia de tareas del ejecutivo cíclico
void (*cyclic_executive [CYClIC_EXECUTIVE_HYPER_PERIOD]
                                        [CYClIC_EXECUTIVE_TASKS_NUMBER+1])(void)={
                                                {Task1,Task2,Task3,NULL},

                                                {Task1,NULL,NULL,NULL},

                                                {Task2,Task1,Task3,NULL},

                                                {Task1,Task2,NULL,NULL},

                                                {Task1,Task3,NULL,NULL},

                                                {Task2,Task1,NULL,NULL}
};

// Función principal
int main()
{

    //Install handlers for enable and disable irqs
    leon3_set_trap_handler(0x83,  leon3_trap_handler_enable_irqs);
    leon3_set_trap_handler(0x84 , leon3_trap_handler_disable_irqs);

    //Declaración de variables de control del ejecutivo cíclico
    uint8_t current_period=0;
    int task_index=0;
    uint64_t next_period_in_ticks_from_reset;

    //Inicialización del servicio de temporización y toma de referencia absoluta
    //del número de ticks desde el reset del sistema

    //Init Timing Service
    init_timing_service( date_time_to_Y2K(18, 3, 22, 0, 0, 0 ));

    //Get Absolute reference
    next_period_in_ticks_from_reset=get_tick_counter_from_reset();

    while(1){
        task_index=0; //poner a 0 al inicio de cada periodo básico

        leon3_print_string("\nStart period\n");
        print_date_time_from_Y2K(get_universal_time_Y2K());
```

```
        // Control de la ejecución de las tareas de cada período básico
        while(cyclic_executive[current_period][task_index]){
                cyclic_executive[current_period][task_index]();
                task_index++;
        }

        //Sincronización con el inicio del siguiente período básico
        //Update Absolute reference with next period
        next_period_in_ticks_from_reset+=CYClIC_EXECUTIVE_PERIOD_IN_TICKS;
        wait_until(next_period_in_ticks_from_reset);

        //Next basic period
        current_period++;
        if(current_period==CYClIC_EXECUTIVE_HYPER_PERIOD){

                current_period=0;
                leon3_print_string("\n****************\n");
                leon3_print_string("\nNext hyperperiod\n");
                leon3_print_string("\n****************\n");
        }

    }
    return 0;
}
```

Comprobar que la salida esperada es la siguiente:

```
Start period
Fecha 18|3|2022 Hora 0:0:0
 Start T1
\
 End T1
 Start T2
\
 End T2
 Start T3
\
 End T3

Start period
Fecha 18|3|2022 Hora 0:0:2
 Start T1
\
 End T1

Start period
Fecha 18|3|2022 Hora 0:0:4
 Start T2
\
 End T2
 Start T1
\
 End T1
 Start T3
\
```

```
 End T3

Start period
Fecha 18|3|2022 Hora 0:0:6
 Start T1
\
 End T1
 Start T2
\
 End T2

Start period
Fecha 18|3|2022 Hora 0:0:8
 Start T1
\
 End T1
 Start T3
\
 End T3

Start period
Fecha 18|3|2022 Hora 0:0:10
 Start T2
\
 End T2
 Start T1
\
 End T1

***************

Next hyperperiod

***************
```

Los elementos que se han utilizado en el programa para la implementación del ejecutivo cíclico, y que deben ser analizados en detalle, son los siguientes:

Definición de las macros de configuración

Para implementar el ejecutivo cíclico, en primer lugar, se han definido un conjunto de macros con el siguiente significado:

- `CYCl_EXEC_PERIOD_IN_TICKS:` duración del período básico en ticks
- `CYCl_EXEC_HYPER_PERIOD:` número de períodos básicos del hiperperiodo
- `CYCl_EXEC_TASKS_NUMBER:` número total de tareas del ejecutivo cíclico

Para implementar el ejecutivo de la figura 4 estas macros han tomado los siguientes valores:

```
//MACROS QUE DEFINEN EL EJECUTIVO CÍCLICO
#define CYCl_EXEC_PERIOD_IN_TICKS             20
#define CYCl_EXEC_HYPER_PERIOD                6
```

```
#define CYCl_EXEC_TASKS_NUMBER                3
```

Array bidimensional para definir la secuencia de tareas del ejecutivo cíclico

Para dar soporte a la ejecución de tareas, se define una array de dos dimensiones de punteros a función, siendo las dos dimensiones: el número de períodos básicos de que consta el hiperperiodo (`CYCl_EXEC_HYPER_PERIOD`) y el número de tareas distintas a ejecutar durante el ejecutivo cíclico incrementado en una unidad (`CYCl_EXEC_TASKS_NUMBER` + 1). Cada fila de este array se inicializa con la secuencia de tareas a ejecutar en el período básico correspondiente, tomando valor NULL los elementos finales de cada fila que no son utilizados en ese período. El hecho de definir la segunda dimensión del array como CYCl_EXEC_TASKS_NUMBER + 1 permite asegurar que la secuencia de ejecución definida en cada fila siempre termina, al menos, con un elemento NULL, siendo esta una forma sencilla de indicar que ya no hay más tareas a ejecutar en ese período básico facilitando así el control de la ejecución.

```
// Array bidimensional para definir la secuencia de tareas del ejecutivo cíclico
void (*cyclic_executive [CYCl_EXEC_HYPER_PERIOD][CYCl_EXEC_TASKS_NUMBER+1])(void)={
                                                    {Task1,Task2,Task3,NULL},

                                                    {T1,NULL,NULL,NULL},

                                                    {Task2,Task1,Task3,NULL},

                                                    {Task1,Task2,NULL,NULL},

                                                    {Task1,Task3,NULL,NULL},

                                                    {Task2,Task1,NULL,NULL}
};
```

Declaración de variables de control del ejecutivo cíclico

Para controlar el ejecutivo cíclico se declaran las siguientes tres variables en la función `main`:

```
   //Declaración de variables de control del ejecutivo cíclico
   uint8_t  current_period=0;
   uint8_t  task_index=0;
   uint64_t next_period_in_ticks_from_reset;
```

Cada una de estas variables tiene el siguiente significado:

- `current_period`: Variable que determina el período básico actual que se está ejecutando.
- `task_index`: Variable que determina la posición de la tarea actual a ejecutar dentro de la secuencia de tareas de cada período básico.

- `next_period_in_ticks_from_reset`: Variable que contiene el número de ticks desde el reset del sistema en el que debe empezar el siguiente período básico.

Inicialización del servicio de temporización y toma de referencia absoluta del número de ticks desde el reset del sistema.

Antes de iniciar el bucle de control del ejecutivo cíclico en el `main`, se inicializa el servicio de temporización, y se toma como referencia absoluta el número de ticks desde el reset:

```
//Init Timing Service
 init_timing_service( date_time_to_Y2K(18, 3, 22, 0, 0, 0 ));

 //Get Absolute reference
 next_period_in_ticks_from_reset=get_tick_counter_from_reset();
```

Control de la ejecución de las tareas de cada período básico

Una vez en el bucle de control global del ejecutivo cíclico, se utiliza el siguiente código para controlar la ejecución de la secuencia de tareas de cada período básico.

```
  // Control de la ejecución de las tareas de cada período básico
 while(cyclic_executive[current_period][task_index]){
        cyclic_executive[current_period][task_index]();
        task_index++;
 }
```

Sincronización con el inicio del siguiente período básico

Con el siguiente código se realiza una espera activa que controla la sincronización con el comienzo del siguiente período básico.

```
  //Sincronización con el inicio del siguiente período básico

  //Update Absolute reference with next period
  next_period_in_ticks_from_reset+=CYClIC_EXECUTIVE_PERIOD_IN_TICKS;
  //Wait until next period starts
  wait_until(next_period_in_ticks_from_reset);
```

11. Práctica 4_3: Creación de proyecto para LEON3

Creación de un nuevo proyecto denominado prac4_3 cuyo ejecutable sea para la plataforma Sparc Bare C. En ese proyecto crear dos subdirectorios **include** y **src,** y añadir todos los archivos del proyecto prac4_2.

12. Práctica 4_3: Tarea a realizar

Teniendo en cuenta el ejecutivo cíclico descrito en la siguiente figura.

Task	Execution Time	Period
TAvoidObstacles	2	10
TPathTracking	3	15
TSensorFusion	5	30
TCalculatePath	6	30

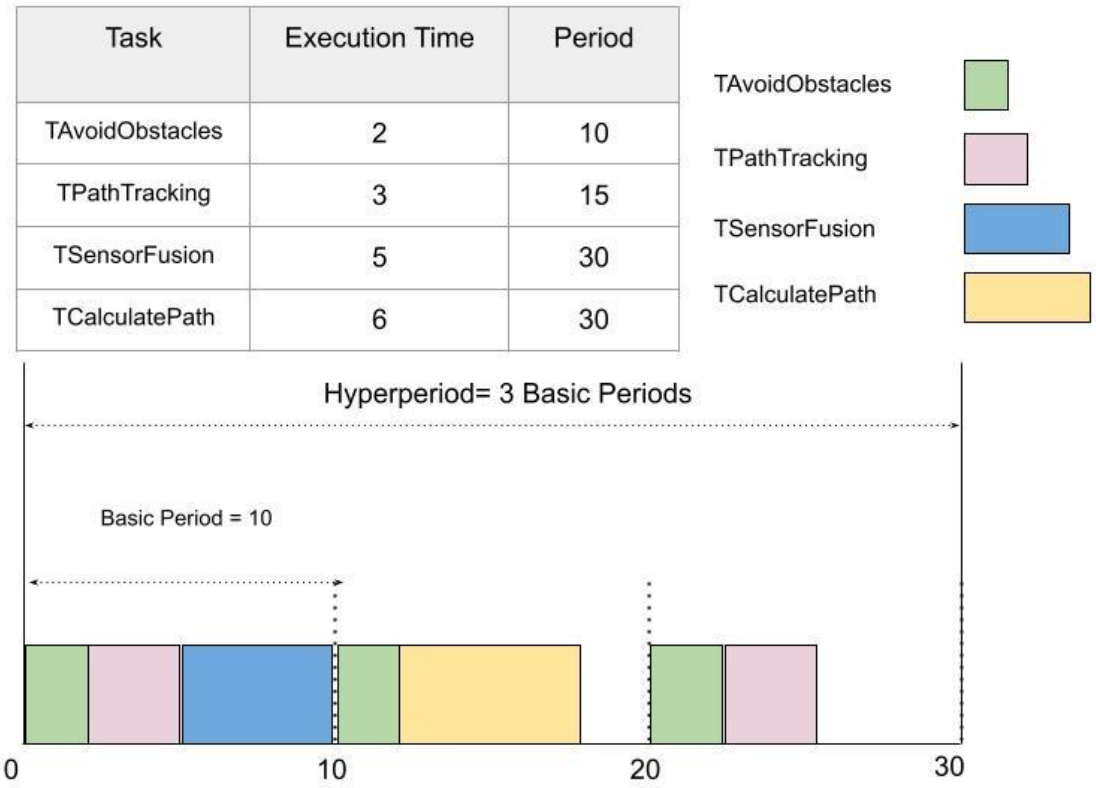


Figura 5. Ejecutivo cíclico a implementar

Modificar el programa, para que, utilizando el siguiente código de implementación de cada una de las cuatro tareas, se implemente el ejecutivo cíclico descrito en la figura 5.

```
void TAvoidObstacle(void){

    leon3_print_string(" Start Avoid Obstacles\n");

    emu_execution_time(2000);

    leon3_putchar('\n');

    leon3_print_string(" End Avoid Obstacles\n");
}

void TPathTracking(void){

    leon3_print_string(" Start Path Tracking\n");
```

```
    emu_execution_time(3000);

    leon3_putchar('\n');

    leon3_print_string(" End Path Tracking\n");
}


void TSensorFusion(void){

       leon3_print_string(" Start Sensor Fusion\n");

       emu_execution_time(5000);

       leon3_putchar('\n');

       leon3_print_string(" End Sensor Fusion\n");

}

void TCalculatePath(void){

       leon3_print_string(" Start Calculate Path\n");

       emu_execution_time(6000);

       leon3_putchar('\n');

       leon3_print_string(" End Calculate Path\n");

}
```

La salida esperada del programa debe ser la siguiente:

```
Start period
Fecha 18|3|2022 Hora 0:0:0
 Start Avoid Obstacles
\
 End Avoid Obstacles
 Start Path Tracking
\
 End Path Tracking
 Start Sensor Fusion
\
 End Sensor Fusion

Start period
Fecha 18|3|2022 Hora 0:0:10
 Start Avoid Obstacles
\
```

```
 End Avoid Obstacles
 Start Calculate Path
\
 End Calculate Path

Start period
Fecha 18|3|2022 Hora 0:0:20
 Start Avoid Obstacles
\
 End Avoid Obstacles
 Start Path Tracking
\
 End Path Tracking

***************

Next hyperperiod

***************
```