Lab Assignment 4. Basic Timing Services.

1. Objective

The objective of this practice *is* that the student learns to design, implement and use a set of basic timing services, directly configuring the *hardware* that supports this type of services, called *Timer Unit*. In order to carry out this practice, work will be done on the *Timer Unit* integrated in the TSIM2 simulator of the LEON3 processor used in previous practices, and the data structures and basic functions that allow its management will be implemented.

2. Introduction

The *Timer Unit* integrated in the TSIM2 simulator of the LEON3 processor has two *timers* that can be programmed to work either independently or in cascade (the cascade configuration is disabled for the evaluation version used in the laboratory).

A clock signal (*clk*) controls the pre-scaling stage of the *Timer Unit* designed to set the base frequency at which the *timers* will work. Each pulse of this clock will decrease the value of the *prescaler* register by one unit. When this register reaches the zero value and a new clock pulse arrives (*underflow* situation) an output pulse will be generated and the value stored in *prescaler reload will be* reloaded. The output pulse of the *prescaler* controls the *timers*, decreasing their *timer value* registers. In this way, the value stored in *Prescaler Reload* allows you to set the base frequency at which the *timers* work as a division of the clock signal frequency, according to the following equation:

$$base\_frequency\_timers = clk\_frequency / (prescaler\ reload + 1)$$

The following figure shows a schematic of the *Timer Unit*, where you can see the function of the pre-scaling stage and its connection to the two *timers*.

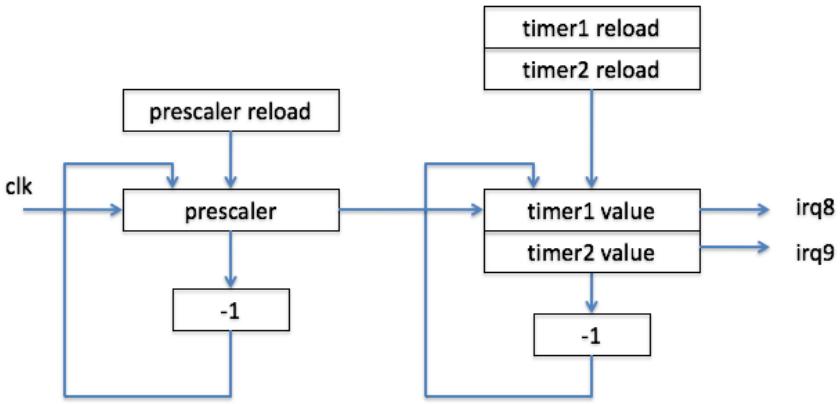

Figure 1. General schematic of the *Timer Unit*

The two *timers* have, in their default configuration, a behavior analogous to that of the pre-scaler stage: the *timer value* register decreases its value each time it receives an input pulse. Once it reaches the zero value, and after receiving a new pulse, the *underflow* situation is produced which generates an output pulse. The *timers* can be configured so that this output pulse triggers an interruption. In this way, and by means of the installation of a handler for this interruption, it is possible to **program** the **execution of an action associated with an interval of time,** and therefore define **timing services**.

Also, as with the *prescaler*, the *timers* can be configured so that, after the *undeflow situation*, a value is **automatically reloaded** in the *timer1 value* and *timer2 value* registers. The *timer1 reload* and *timer2 reload* registers respectively store the reload value for each of the *timers*. In this way, and in combination with the installation of the corresponding interrupt handler, the **periodic execution** of an action can be **programmed**, and from this action, different **timing services** can be defined.

The *Timer Unit* integrated in the LEON3-based SoC can be configured to generate independent interrupts for each *timer*, or share the same interrupt (the latter option is disabled in the evaluation version of TSIM2). The external interrupts associated with *timer1* and *timer2* are, respectively, those of level 8 and level 9.

Finally, the *Timer Unit* also allows the two *timers* to be chained, so that the input pulse of *timer2* is provided by the output pulse of *timer1*. As mentioned above, this option, available in the commercial configuration, is disabled in the evaluation version of TSIM2 used in these practices.

3. Design of a Driver for the Timer Unit

The following table shows the set of registers that constitute the *Timer Unit* controller.

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

Table 1. *Timer Unit* Controller Registers

To manage the *Timer Unit* controller, and manipulate its registers, a set of routines will be defined that will constitute the driver of this device. The routines are the following:

`leon3_timerunit_set_configuration`

A routine that configures the *Timer Unit*, setting the value of the *Configuration Register* as well as initialising the *Scaler Value and Scaler Reload Value* registers. Its prototype is as follows:

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

The `scalerValue` parameter is used to initialise the values of the *Scaler Value and Scaler Reload Value* registers, while the `freeze_during_debug` and `separate_interrupts` parameters are used to initialise the *DF* and *SI* fields of the *Configuration Register* (see Figure 2). The `freeze_during_debug` field determines whether during processor debug mode the timer should be *freeze* or not, while the `separate_interrupts` parameter determines whether the *timers* use separate interrupts.

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.

Figure 2. Fields of the *Configuration Register* to be configured by the driver

`leon3_timer_config`

Routine that allows to configure the two *timers* of the *Timer Unit*, setting the value of its registers *Timer Control Register*, *Timer Counter Value Register and Timer Counter Reload Value Register*. Its prototype is as follows:

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

- The `timerId` parameter indicates whether *timer1* (`timerId==0`) or *timer2* (`timerId==1`) is to be configured.
- The `timerValue` parameter initialises the value of the *Timer Counter Value and Timer Counter Reload Value* registers.

- The parameter `chain_with_prec_timer` is used to initialise the *CH* field of the *Timer Control Register (*see figure 3). This field determines whether the *timer* takes as input pulse the output pulse of the previous timer (chained mode) or the output pulse of the *prescaler* (In the evaluation version of TSIM2 the chained mode is disabled).
- The `restart_timer` parameter is used to initialise the *RX* field of the *Timer Control Register (*see Figure 3) which determines whether after the *undeflow situation* the *Timer Counter Value* register is reset (`restart_timer !=0`) or not (`restart_timer ==0`) with the value of the *Timer Counter Reload Value*.

To modify the *Timer Counter Value* register, first set the value of the *Timer Counter Reload Value* and then activate the *LD* field of the *Timer Control Register.*

Next figure shows the fields of the *Timer Control Register*. In order to control the value of these fields the following set of functions will be defined.

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

Figure 3. *Timer Configuration Register* fields

leon3_timer_enable_irq and leon3_timer_disable_irq

Routines that enable and disable the interrupt associated with the *underflow* condition of a timer.

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

- The `timerId` parameter indicates whether enabling the interrupt affects *timer1* (`timerId==0`) or *timer2* (`timerId==1`). Both functions modify the *IE* field of the *Timer Control Register*.

leon3_timer_enable and leon3_timer_disable

Routines that enable and disable timers.

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

- The `timerId` parameter indicates whether the enable affects *timer1* (`timerId==0`) or *timer2* (`timerId==1`). Both functions modify the *EN* field of the *Timer Control Register*.

leon3_timer_clear_irq

Routine that clears the *IP* bit of the *Timer Control Register*. This bit is automatically set to 1 when an interrupt occurs due to an *underflow* situation. It must be cleared in the interrupt handler to indicate that the interrupt has already been serviced.

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

4. Monotonic clock design and universal time management.

One of the most basic timing services that can be implemented by the *Timer Unit* is a universal time manager via a monotonic clock. A monotonic clock is one that is incremented periodically during the execution of a system. The value taken by the clock can thus be associated to the universal time by means of a reference. For this practice we will use the following routines **(already coded)** that facilitate the implementation of a monotonic clock over a universal time encoding known as Y2K. This encoding uses a 32-bit integer to set the seconds that have passed since 1 January 2000.

init_monotonic_clock

A routine that initialises the variables that manage the monotonic clock and sets a reference value in Y2K format as the initial universal time. Its prototype is as follows:

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

irq_handler_update_monotonic_clock

An interrupt attention routine that updates the monotonic clock. This routine shall be invoked periodically by scheduling a timer from the Timer Unit. The invocation period shall be defined by the macro `TIMING_SERVICE_TICKS_PER_SECOND`. The function prototype is as follows:

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

update_universal_time_Y2K

Function for updating the Y2K coded reference for calculating the universal time. The prototype of the function is as follows:

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

get_universal_time_Y2K

Function that returns the current value of the universal time encoded in Y2K. The prototype of the function is as follows:

```
uint32_t get_universal_time_Y2K ();
```

print_date_time_from_Y2K

A function that prints the value of the universal time encoded in Y2K that is passed via the `seconds_from_y2K` parameter. The prototype of the function is as follows:

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

`date_time_to_Y2K`

Function that encodes in Y2K a date and time passed by parameter using the fields `day, month, year, hour, minutes and seconds`. The prototype of the function is as follows:

```
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. Timing Services

Using the *Timer Unit driver*, it is possible to program a ***timer*** to interrupt periodically. By means of a routine to attend to this periodic interruption, called a **system tick**, it is possible to define different timing services, such as the monotonic clock mentioned in the previous section. In this practice we are going to define the following function to program one of the *timers* of the *Timer Unit* and initialise the timing services that support the monotonic clock:

`init_timing_service`

Function that configures a *Timer Unit timer* to schedule a periodic interrupt (**system tick).** The function also installs a user handler for this interrupt to support different basic timing services, such as the management of a monotonic clock. To initialise this service, the function receives the parameter `currentTime_in_Y2K`, which indicates the date and time value (universal time) at the time of the call.

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

6. Practice 4_1: LEON3 Project creation

Creation of a new project called prac4_1 whose executable is for the Sparc Bare C platform. In this project, create two subdirectories: **include** and **src**. In the **src** directory add the .c and .asm files from the previous practice (except main.c), together with the files *leon3_timer_unit_drv.c, leon3_monotonic_clock.c, leon3_timing_service.c* that you will find in the file linked as *prac4_1_sources.zip* in the web page. Similarly, add to the **include** directory the header files (.h) from the previous practice, along with the files *leon3_timer_unit_drv.h, leon3_monotonic_clock.h and leon3_timing_service.h* contained in *prac4_1_sources.zip.*

7. Practice 4_1: Task to be performed

1. Add the following code to the `leon3_types.h` file. This code defines the `NULL` macro for pointer handling and allows to define the `bool_t` type selectively (using the `bool` type if compiling a C++ file or the `unsigned char` type if compiling a C file).

```
#ifndef LEON3_TYPES_H_
#define LEON3_TYPES_H_.

#ifndef NULL
      #define NULL 0
#endif

#ifndef __cplusplusplus

      typedef unsigned char         bool_t;

      #define true      1
      #define false     0

#else

      typedef bool            bool_t;

#endif

typedef unsigned char         byte_t;
...
```

2. **Complete in the file *leon3_timer_unit_drv.c*** the code for the following defined functions as specified in section 3 of this script:

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

3. Define in the file *leon3_timing_service. h* the value of the macro `TIMING_SERVICE_TICKS_PER_SECOND` to set the periodic interruption of the clock **tick to** have a frequency of 10 Hz.

```
#define TIMING_SERVICE_TICKS_PER_SECOND        10
```

4. **Complete in the file *leon3_timing_service.c* the code of the function** `init_timing_service (uint32_t currentTime_in_Y2K)`. To complete this function**, use the following macros declared** in the same file:
   - `LEON3_FREQ_MHZ` frequency at which the processor operates in MHZ
   - `TIMER_ID` identifier of the *timer* to be used
   - `TIMER_IRQ_LEVEL` interrupt level of this *timer*

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

The function must perform the following actions:

1) Make the system call that disables all interrupts.
2) Mask the *timer* interrupt level (`TIMER_IRQ_LEVEL`)
3) Disable *timer* with `TIMER_ID` using `leon3_timer_disable`
4) Disable *timer* interrupt with `TIMER_ID` using `leon3_timer_disable_irq`
5) Configure, using `leon3_timerunit_set_configuration`, the *TimerUnit* so that the output pulse of the pre-scaler stage has a frequency of 1MHz, *freeze* is enabled during debugging, and separate interrupts are generated for the two *timers*.

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

6) Configure the *timer* with `TIMER_ID` identifier, using the `leon3_timer_config` function, to underflow every system **tick** `(1000000UL/TIMING_SERVICE_TICKS_PER_SECOND -1)`, without chaining the timers, and with *timer* reset after *underflow*.

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

7) Install `timertick_irq_handler` as the handler associated with the *timer* interrupt level, defined by the `TIMER_IRQ_LEVEL` macro.
8) Initialising the monotonic clock

```
init_monotonic_clock(currentTime_in_Y2K);
```

9) *Clear timer* interrupt with TIMER_ID using `leon3_timer_clear_irq`

10) Unmask *timer* interrupt level (`TIMER_IRQ_LEVEL`)
11) Enable *timer* interrupt with TIMER_ID identifier using the `leon3_timer_enable_irq` function.
12) Enable the *timer* with `TIMER_ID` identifier using the `leon3_timer_enable` function.
13) Perform system call to enable interrupts by completing initialisation.

5. Check the correct functioning of the *Timer Unit* driver implementation with the following main program that will be included as **main.c** file to the project.

```
#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_timer_unit_drv.h"
#include "leon3_timing_service.h"


int main()
{
       uint32_t aux1,aux2;

      //Install traps handlers to enable
      //and disable interrupts
      //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);

      //Initialise timing service with time
      //universal current
      //Init Timing Service with current universal time

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

      while(1){

            //Display time with a 10-second interval.
            //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;
}
```

The correct output of the programme must be a time message every 10 seconds.

```
Date 18|3|2022 Time 0:0:0
Date 18|3|2022 Time 0:0:10
Date 18|3|2022 Time 0:0:20
Date 18|3|2022 Time 0:0:30
Date 18|3|2022 Time 0:0:40
Date 18|3|2022 Time 0:0:50
Date 18|3|2022 Time 0:1:0
Date 18|3|2022 Time 0:1:10
Date 18|3|2022 Time 0:1:20
Date 18|3|2022 Time 0:1:30
```

8. Timing services for the implementation of a cyclic executive

Using the **system tick** interrupt, it is possible to implement, in addition to the monotonic clock seen in the previous section, other timing services. In this part of the practice we will see how the **system tick** will allow us to provide services destined to the construction of a *cyclic executive*, that is, a programme that periodically repeats a sequence of task execution. *Cyclic executives* have a basic period, which controls the start of the execution of a specific sequence of tasks, and a hyper-period, which is the number of basic periods from which the complete sequence of execution of the tasks of the *cyclic executive* is repeated.

The following figure shows an example of a *cyclic executive*, in which the periodic execution of 3 tasks is controlled, whose periods and execution times are determined in the attached table.

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

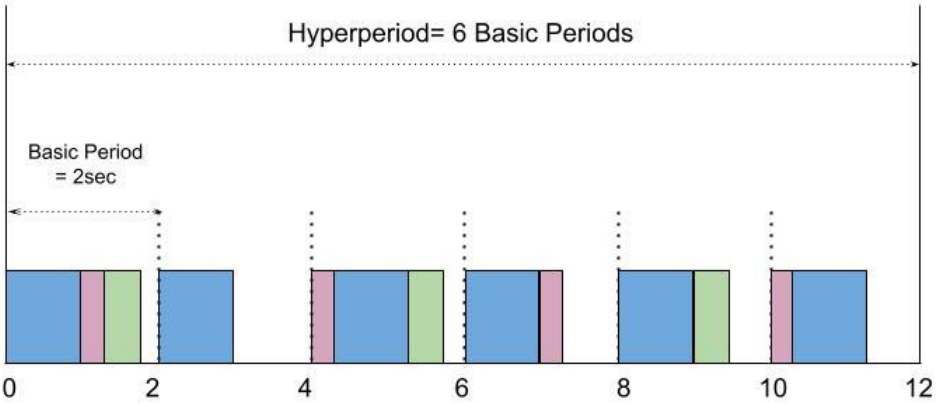

Figure 4. Example of a cyclic executive

The *cyclic executive* has a basic period of 2 seconds, while the hyperperiod is 6 basic periods.

In order to be able to implement this and other cyclic executives, the timing services implemented in the *leon3_timing_service* module are going to be extended. This extension consists, first of all, in adding or modifying the following elements to the *leon3_timing_service.*c file

`TickCounterFromReset`

The global variable `TickCounterFromReset` is added to the declared variables in order to be able to manage a **tick** counter since the last system reset.

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

`timertick_irq_handler`

The `timertick_irq_handler` function, installed as an attention routine associated with the **system tick**, is modified, as shown below, to increment the `TickCounterFromReset` variable by one each time **the system tick is** interrupted.

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

void timertick_irq_handler (void){

    leon3_timer_clear_irq(TIMER_ID);

    TickCounterFromReset++;

    irq_handler_update_monotonic_clock();

}
```

get_tick_counter_from_reset

In order to obtain the value of this variable, the definition of the function `get_tick_counter_from_reset` is also added to the *leon3_timing_service. c* file. This function returns the value of the global variable `TickCounterFromReset`, properly handling the *timer* interrupt mask in order to avoid possible race conditions.

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

uint64_t get_tick_counter_from_reset(void){

    uint64_t tick_counter_from_reset;

    // I mask the timer interrupt to avoid race conditions.
    //Mask the timer interrupt to avoid race condition
     leon3_mask_irq(TIMER_IRQ_LEVEL);

     tick_counter_from_reset=TickCounterFromReset;

     //unmasking to return to the previous situation
     //unmask to restore previous situation
     leon3_unmask_irq(TIMER_IRQ_LEVEL);

    return tick_counter_from_reset;

}
```

wait_until

Finally, in order to synchronise the execution of the tasks with the start of each basic period of the cyclic executive, the `wait_until` function is implemented as part of the timing services. This function uses the value received as a parameter (`ticksFromReset`), to perform an active wait until this value matches the value returned by the `get_tick_counter_from_reset`.

```
//*****************************************************************
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;

}
```

The extension is further completed by the declaration of the functions `get_tick_counter_from_reset and wait_until` in the file *leon3_timing_service. h,*

```
uint64_t get_tick_counter_from_reset(void);

uint8_t wait_until(uint64_t ticks_from_reset);
```

9. Practice 4_2: LEON3 Project creation

Creation of a new project called prac4_2 whose executable is for the Sparc Bare C platform. In this project create two subdirectories **include** and **src** where all the files of the prac4_1 project must be added **except the one containing the `main` function**.

10. Practice 4_2: Task to be performed

1. Modify the files *leon3_timing_service.h* and *leon3_timing_service.c according to* section 8.

2. Test the operation of these functions by means of the following main program to be included as **main.c** file to the project. This program implements the cyclic execution described in Figure 4. The three tasks Task1, Task2 and Task3 use the `emu_execution_time` function (already implemented) to emulate their execution time, in milliseconds, as specified in the table in that figure.

```
#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_timer_unit_drv.h"
#include "leon3_timing_service.h"


// Task code T1
void Task1(void){

    leon3_print_string(" Start T1");

    emu_execution_time(1000); //execution time 1000 ms

    leon3_putchar('\n');

    leon3_print_string(" End T1");

}

// Task code T2
void Task2(void){


       leon3_print_string(" Start T2");

       emu_execution_time(300); //execution time 300 ms

           leon3_putchar('\n');

           leon3_print_string(" End T2");
}

// Task code T3
void Task3(void){

        leon3_print_string(" Start T3");

    emu_execution_time(500); //execution time 500 ms

           leon3_putchar('\n');

           leon3_print_string(" End T3");

}

// Definition of the configuration macros
#define CYClIC_EXECUTIVE_PERIOD_IN_TICKS            20
```

```
#define CYClIC_EXECUTIVE_HYPER_PERIOD               6
#define CYClIC_EXECUTIVE_TASKS_NUMBER               3

// Two-dimensional array to define the sequence of tasks of the cyclic executive
void (*cyclic_executive [CYClIC_EXECUTIVE_HYPER_PERIOD]

      [CYClIC_EXECUTIVE_EXECUTIVE_TASKS_NUMBER+1])(void)={
                                                  {Task1,Task2,Task3,NULL},

                                                  {Task1,NULL,NULL,NULL,NULL},

                                                  {Task2,Task1,Task3,NULL},

                                                  {Task1,Task2,NULL,NULL},

                                                  {Task1,Task3,NULL,NULL},

                                                  {Task2,Task1,NULL,NULL}
};

// Main function
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);

    //Declaration of control variables of the cyclic executive
    uint8_t current_period=0;
    int task_index=0;
    uint64_t next_period_in_ticks_from_reset;

    //Initialisation of the timing service and taking of absolute reference
    // of the number of ticks since system reset

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

    //Get Absolute reference
    next_period_in_ticks_from_reset=get_tick_counter_from_reset();

    while(1){
       task_index=0; //set to 0 at the start of each basic period

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

       // Control the execution of the tasks of each basic period
    while(cyclic_executive[current_period][task_index]){
       cyclic_executive[current_period][task_index]();
       task_index++;
          }

    //Synchronisation with the start of the next basic period
    //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("\next hyperperiod");
               leon3_print_string("\n****************\n");
    }

    }
    return 0;
}
```

Check that the expected output is as follows:

```
Start period
Date 18|3|2022 Time 0:0:0
 Start T1
\
 End T1
 Start T2
\
 End T2
 Start T3
\
 End T3

Start period
Date 18|3|2022 Time 0:0:2
 Start T1
\
 End T1

Start period
Date 18|3|2022 Time 0:0:4
 Start T2
\
 End T2
 Start T1
\
 End T1
 Start T3
\
 End T3

Start period
Date 18|3|2022 Time 0:0:6
 Start T1
\
 End T1
 Start T2
\
 End T2
```

```
Start period
Date 18|3|2022 Time 0:0:8
 Start T1
\
 End T1
 Start T3
\
 End T3

Start period
Date 18|3|2022 Time 0:0:10
 Start T2
\
 End T2
 Start T1
\
 End T1

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

Next hyperperiod

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

The elements that have been used in the programme for the implementation of the cyclic executive, and which need to be analysed in detail, are the following:

Definition of configuration macros

To implement the cyclic executive, first of all, a set of macros have been defined with the following meaning:

- `CYCl_EXEC_PERIOD_IN_TICKS`: duration of the basic period in ticks
- `CYCl_EXEC_HYPER_PERIOD`: number of basic hyperperiod periods
- `CYCl_EXEC_TASKS_NUMBER`: total number of tasks of the cyclic executive

To implement the executive in figure 4 these macros have taken the following values:

```
//MACROS DEFINING THE CYCLIC EXECUTIVE
#define CYCl_EXEC_PERIOD_IN_TICKS             20
#define CYCl_EXEC_HYPER_PERIOD                6
#define CYCl_EXEC_TASKS_NUMBER                3
```

Two-dimensional array for defining the task sequence of the cyclic executive

To support the execution of tasks, a two-dimensional array of function pointers is defined, the two dimensions being: the number of basic periods of which the hyperperiod consists (`CYCl_EXEC_HYPER_PERIOD`) and the number of different tasks to be executed during the cyclic executive increased by one unit (`CYCl_EXEC_TASKS_NUMBER` + 1). Each row of this array is initialised with the sequence of tasks to be executed in the corresponding basic

period, taking NULL value the final elements of each row that are not used in that period. Defining the second dimension of the array as CYCl_EXEC_TASKS_NUMBER + 1 ensures that the execution sequence defined in each row always ends with at least one NULL element, which is a simple way of indicating that there are no more tasks to be executed in that basic period, thus facilitating execution control.

```
// Two-dimensional array to define the sequence of tasks of the cyclic executive
void (*cyclic_executive [CYCl_EXEC_HYPER_PERIOD][CYCl_EXEC_TASKS_NUMBER+1])(void)={
                                                    {Task1,Task2,Task3,NULL},

                                                    {T1,NULL,NULL,NULL,NULL},

                                                    {Task2,Task1,Task3,NULL},

                                                    {Task1,Task2,NULL,NULL},

                                                    {Task1,Task3,NULL,NULL},

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

Declaration of cyclic executive control variables

To control the cyclic executive, the following three variables are declared in the `main` function:

```
   //Declaration of control variables of the cyclic executive
   uint8_t current_period=0;
   uint8_t task_index=0;
   uint64_t next_period_in_ticks_from_reset;
```

Each of these variables has the following meaning:

- `current_period`: Variable that determines the current basic period being executed.
- `task_index`: Variable that determines the position of the current task to be executed within the sequence of tasks of each basic period.
- `next_period_in_ticks_from_reset`: Variable containing the number of ticks from the system reset at which the next basic period is to start.

Initialisation of the timing service and taking absolute reference of the number of ticks since system reset.

Before starting the cyclic executive control loop in `main`, the timing service is initialised, and the number of ticks since reset is taken as the absolute reference:

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

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

Monitoring of the execution of the tasks of each basic period

Once in the global control loop of the cyclic executive, the following code is used to control the execution of the sequence of tasks for each basic period.

```
  // Monitoring of the execution of tasks for each basic period
 while(cyclic_executive[current_period][task_index]){
       cyclic_executive[current_period][task_index]();
       task_index++;
  }
```

Synchronisation with the start of the next basic period

With the following code an active wait is performed which controls the synchronisation with the start of the next basic period.

```
  //Synchronisation with the start of the next basic period

  //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. Practice 4_3: LEON3 Project creation

Creation of a new project called prac4_3 whose executable is for the Sparc Bare C platform. In this project, create two subdirectories, **include** and **src,** and add all the files from the prac4_2 project.

12. Practice 4_3: Task to be performed

Taking into account the cyclic executive described in the figure below.

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

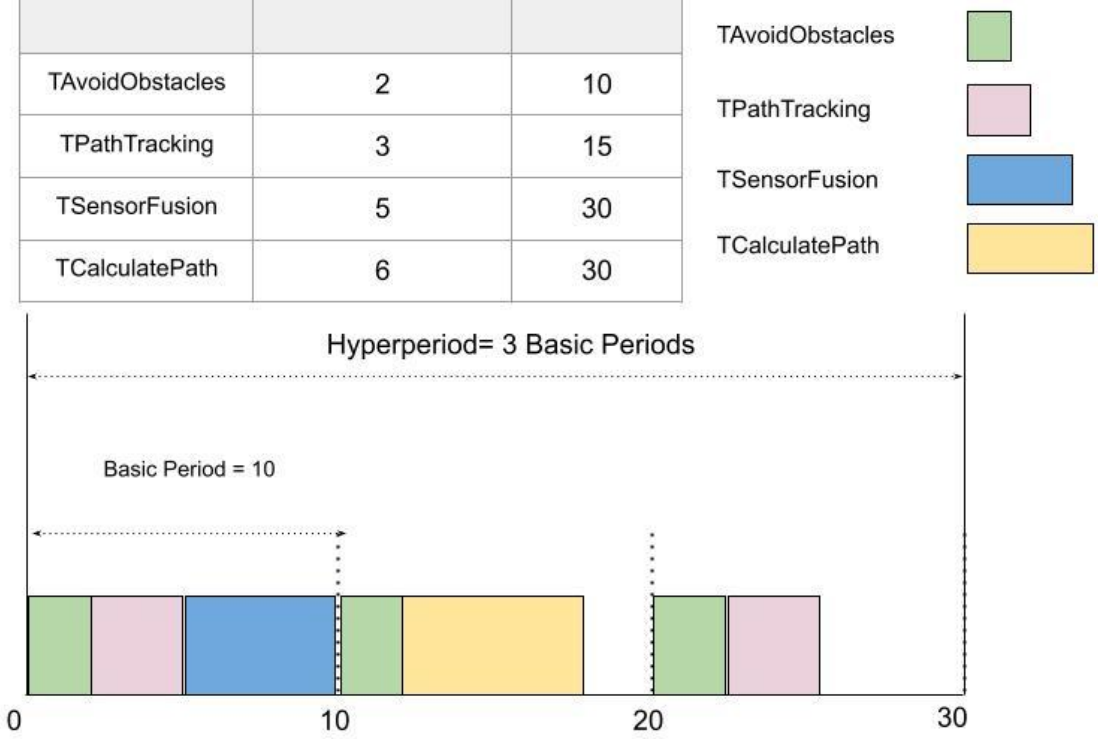

Figure 5. Cyclic executive to be implemented

Modify the program so that, using the following implementation code for each of the four tasks, the cyclic executive described in Figure 5 is implemented.

```
void TAvoidObstacle(void){

    leon3_print_string(" Start Avoid Obstacles");

    emu_execution_time(2000);

    leon3_putchar('\n');

    leon3_print_string(" End Avoid Obstacles");
}
```

```
void TPathTracking(void){

    leon3_print_string(" Start Path Tracking");

    emu_execution_time(3000);

    leon3_putchar('\n');

    leon3_print_string(" End Path Tracking");
}


void TSensorFusion(void){

          leon3_print_string(" Start Sensor Fusion");

          emu_execution_time(5000);

          leon3_putchar('\n');

          leon3_print_string(" End Sensor Fusion");

}

void TCalculatePath(void){

          leon3_print_string(" Start Calculate Path");

    emu_execution_time(6000);

          leon3_putchar('\n');

          leon3_print_string(" End Calculate Path");

}
```

The expected output of the programme should be as follows:

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

Start period
```

```
Date 18|3|2022 Time 0:0:10
 Start Avoid Obstacles
\
 End Avoid Obstacles
 Start Calculate Path
\
 End Calculate Path

Start period
Date 18|3|2022 Time 0:0:20
 Start Avoid Obstacles
\
 End Avoid Obstacles
 Start Path Tracking
\
 End Path Tracking

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

Next hyperperiod

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