Politecnico di Torino

Master’s degree in Mechatronic Engineering

2025/2026

Operating systems for embedded systems

Report Lab 01: Tasks and alarms

Date 13/10/2025

Group

Matteo Morfino – 345791

Matteo Vecchia – 348346

Exercise 1

Regarding the first exercise, the objective was to implement two basic periodic tasks within the system.

Specifically, Task A was required to operate with a period of 500 ms and a priority level of 2, while Task B had to execute every 750 ms with a different assigned priority.

For this exercise, the corresponding .oil configuration file was already provided and therefore did not require any modification. Instead, the implementation focused on adapting the .c source file, where the necessary task definitions and periodic behavior were introduced according to the given specifications.

```c
#include <stdio.h>
#include <math.h>
#include "tpl_os.h"

int main(void)
{
    printf("lab01\n");
    printf("Task A: period = 500 ms, priority = 2, starts at t=0 ms\n");
    printf("Task B: period = 750 ms, priority = 1, starts at t=1500 ms\n");
    printf("Starting OS...\n\n");
    StartOS(OSDEFAULTAPPMODE);
    return 0;
}

DeclareAlarm(a500msec);
DeclareAlarm(a750msec);

static int counterA = 0;
static int counterB = 1500;

TASK(TaskA)
{
    counterA += 500;
    printf("Task A executed. CounterA = %d\r\n", counterA);
    TerminateTask();
}

TASK(TaskB)
{
    counterB += 750;
    printf("Task B executed. CounterB = %d\r\n", counterB);
    TerminateTask();
}

TASK(stop)
{
    CancelAlarm(a500msec);
    CancelAlarm(a750msec);
    printf("\nStopping periodic tasks...\r\n");
    printf("Final CounterA = %d\r\n", counterA);
    printf("Final CounterB = %d\r\n", counterB);
    printf("Shutdown\r\n");
    ShutdownOS(E_OK);
    TerminateTask();
}
```

Figure 1:c code for the ex 1

Basically, a simple counter mechanism was implemented within each task. The counter variable increments its value each time the corresponding task is executed, and its current value is printed to the console. When the program terminates, the final value of the counter is displayed, providing a clear indication of how many times each task was executed during the runtime.

The resulting output is shown in Figure 2.

```
lab01
Task A: period = 500 ms, priority = 2, starts at t=0 ms
Task B: period = 750 ms, priority = 1, starts at t=1500 ms
Starting OS...

Task A executed. CounterA = 500
Task A executed. CounterA = 1000
Task A executed. CounterA = 1500
Task A executed. CounterA = 2000
Task B executed. CounterB = 2250
Task A executed. CounterA = 2500
Task B executed. CounterB = 3000
Task A executed. CounterA = 3000
Task A executed. CounterA = 3500
Task B executed. CounterB = 3750
Task A executed. CounterA = 4000
Task B executed. CounterB = 4500
Task A executed. CounterA = 4500
Task A executed. CounterA = 5000
Task B executed. CounterB = 5250
Task A executed. CounterA = 5500
Task B executed. CounterB = 6000
Task A executed. CounterA = 6000

Stopping periodic tasks...
Final CounterA = 6000
Final CounterB = 6000
Shutdown
Exiting virtual platform.
```

Figure 2:Output for ex1

The experiment confirms the correct configuration and execution of both periodic tasks according to their defined timing and priority parameters.

Exercise 2

For the second exercise, the requirement was to maintain the same task specifications but execute them on an Arduino Uno platform.

To achieve this, only the initial section of the .oil configuration file was modified to adapt the system to the Arduino target as outlined in the reference blink file, while the section defining the Tasks remained unchanged.

```
OIL_VERSION = "2.5" : "test" ;

CPU test {
    OS config {
        STATUS = STANDARD;
        BUILD = TRUE {
            TRAMPOLINE_BASE_PATH = "../../..";
            APP_NAME = "ex2_arduino";
            APP_SRC = "ex2.c";
            CPPCOMPILER = "avr-g++";
            COMPILER = "avr-gcc";
            LINKER = "avr-gcc";
            ASSEMBLER = "avr-gcc";
            COPIER = "avr-objcopy";
            SYSTEM = PYTHON;
        };
    SYSTEM_CALL = TRUE;
    };
```

Figure 3: .oil file for ex 2

As for the .c source file, the task functions were modified using Arduino's commands, in the setup() we declare the PinMode of pin 13 as OUTPUT, while the loop() remains empty because the program works using the tasks. Each task contains the digitalWrite() instruction, implementing the behavior according to exercise 2, turning on led 13 with TaskA and turning it off with TaskB.

The generated .hex file was then uploaded to a simulated Arduino Uno board using SimulIDE. The LED blinking behavior was observed to match the expected timing and priority specifications.

The modified source code can be seen in Figure 4.

```c
#include <Arduino.h>
#include "tpl_os.h"

#define LED_PIN 13

DeclareAlarm(a500msec);
DeclareAlarm(a750msec);

void setup(void)
{
    init();
    pinMode(LED_PIN, OUTPUT);
    StartOS(OSDEFAULTAPPMODE);
    // return 0;
}

void loop(void)
{
    while(1)
}

TASK(TaskA)
{
    digitalWrite(LED_PIN, HIGH);
    TerminateTask();
}

TASK(TaskB)
{
    digitalWrite(LED_PIN, LOW);
    TerminateTask();
}

TASK(stop)
{
    CancelAlarm(a500msec);
    CancelAlarm(a750msec);
    ShutdownOS(E_OK);
    TerminateTask();
}
```

Figure 4:c code for the ex 2

In conclusion, both exercises successfully demonstrated the implementation and execution of periodic tasks using the OSEK environment, first on a simulated platform and then on an Arduino Uno target. The obtained results confirmed the correct configuration, timing behavior, and portability of the developed code.