Ecole Centrale de Nantes

EMARO1-CORO1 Master

R E T S Y

(**RE**al **T**ime **SY**stems)

Lab #2: PERIODIC TASKS AND ALARMS

Report of:

Hryshaienko Olena

(Group C)

Tartari Michele

(Group C)

N A N T E S

2 0 1 7

INTRODUCTION

Real-Time systems are reactive systems which have to do processing as a result of events.

In this lab we will trigger processing as a result of time passing (expiration of an Alarm), using the following concepts: alarm and counter.

On the TP ECN board, the Systick timer is used as interrupt source for alarms. The interrupt is sent every 1ms.

We have implemented the following applications, answering to the next questions:

1) **FIRST** application

This application implements a periodic task that reads push button *P0_8* of the board every 100ms by using the function *readbutton()*. We have then coded a *t_process* task that does different action every time the button is pushed:

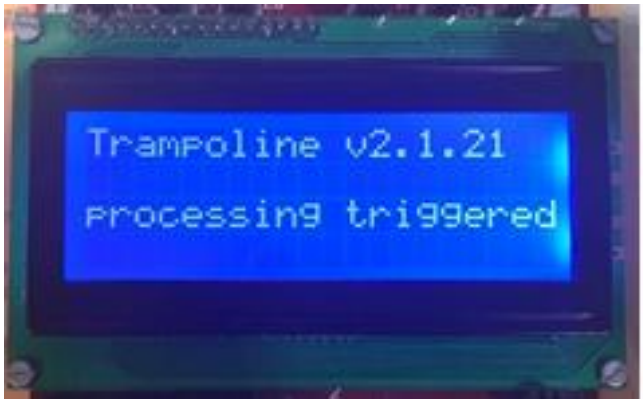
- on *odd* execution displays "*processing triggered*";
- on *even* executions clears the LCD.

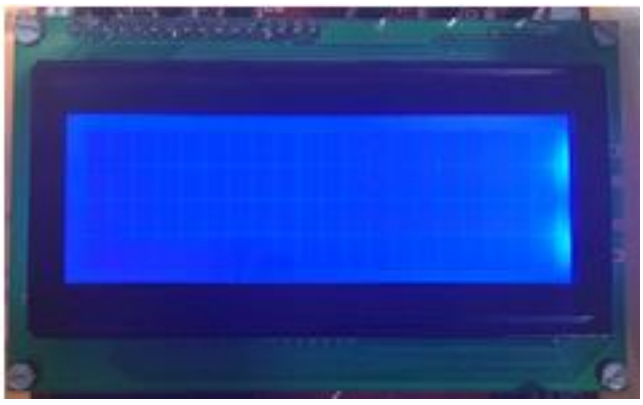
Snippet of .cpp file

```
TASK(button_scanner)
{
  ButtonState button = readButton();
  if (button == BUTTON_PUSH) {
      digitalWrite(3, HIGH);
      ActivateTask(t_process);
  }
  else if (button == BUTTON_RELEASE)
{
    digitalWrite(3, LOW);
  }
  TerminateTask();
}

TASK(t_process)
{
    static int i=0;
    i = 1 - i;
    if(i==0)
        lcd.clear();
    else
        lcd.print("Proc triggered");
    TerminateTask();
}
```

Snippet of .oil file

```
TASK button_scanner {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK t_process {
        PRIORITY = 2;
        AUTOSTART = FALSE;
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };

  ALARM run_button_scanner {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
      TASK = button_scanner;
    };
    AUTOSTART = TRUE {
      APPMODE = stdMode;
      ALARMTIME = 100;
      CYCLETIME = 100;
    };
  };
```



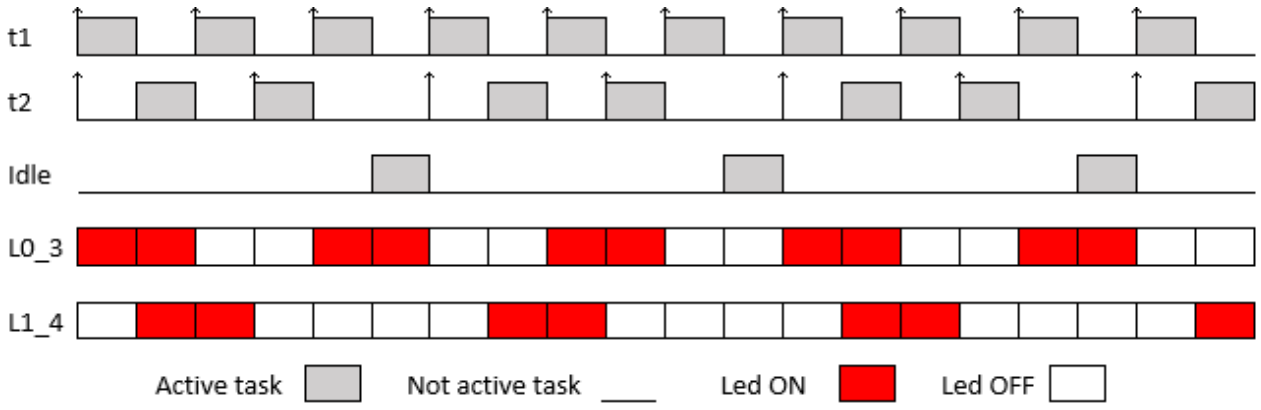


2) **SECOND** application

2 periodic tasks:

- *t1* (priority 2, period 1s);
  - *t1* toggles *LED L0_3* each time it executes;
- *t2* (priority 1, period 1.5s);
  - *t2* toggles *LED L1_4* each time it executes.

Question 2.1 To give the 20 fist states of the LED (by hand), given by the execution of the application with the display date of each state (0 being the application startup date). Is the whole system periodic? If yes, what is the period and the behavior?



The system is indeed periodic and has a period of 6s

Question 2.2 The application needs a counter and 2 alarms. In Trampoline/ARM a counter is connected to a timer with a 1ms cycle time. What maximum *TICKSPERBASE* do you use to fulfill the application requirements?

We set the *TIKPERBASE* variable of the *SystemCounter* to be equal to 500, being 500ms the maximum common divisor of between the cycle times of *t1* and *t2*.

Question 2.3 How are the alarms configured to fulfill the application requirements? Declare the counter and both alarms and write the application. Verify it works.

Snippets of .oil file

```
COUNTER SystemCounter {
  TICKSPERBASE = 500;
  MAXALLOWEDVALUE = 65535;
  MINCYCLE = 1;
  };


TASK t1 {
  PRIORITY = 3;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
```

```
  ALARM run_t1 {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
      TASK = t1;
    };
    AUTOSTART = TRUE {
      APPMODE = stdMode;
      ALARMTIME = 2;
      CYCLETIME = 2;
    };
};
```

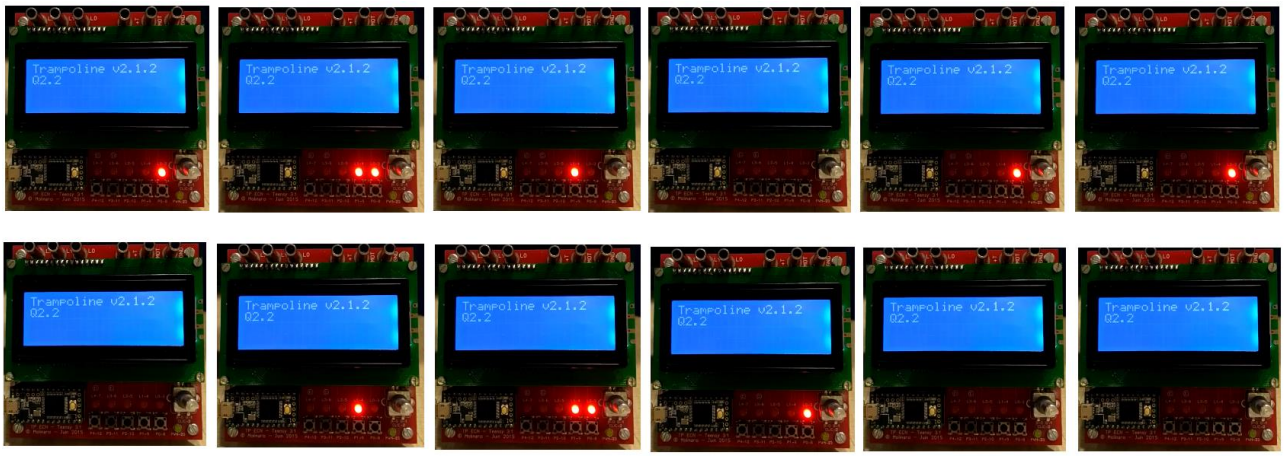
```
    SCHEDULE = FULL;
  };


  TASK t2 {
        PRIORITY = 1;
        AUTOSTART = FALSE;
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };
```

```
ALARM run_t2 {
  COUNTER = SystemCounter;
  ACTION = ACTIVATETASK {
  TASK = t2;
  };
  AUTOSTART = TRUE {
  APPMODE = stdMode;
  ALARMTIME = 3;
  CYCLETIME = 3;
  };
};
```

Snippet of .cpp file

```
TASK(t1)
{
   digitalWrite(3, !digitalRead(3));
   TerminateTask();
}

TASK(t2)
{
   digitalWrite(4, !digitalRead(4));
   TerminateTask();
}
```

When compiled and executed the application behaved as expected.



3) **THIRD** application

Question 3.1 Design and program application using Trampoline, using following requirements:

- After starting, the system waits.
- When the button is pressed, the system start a function F that is implemented using a periodic task (period = 1s), which is using a blinking *LED L0_3*.
- When the button is pressed again, function F is stopped. When the switch is pressed, the system is shutdown as quickly as possible.

A *readButton9()* function has been created, it has the same behavior as *readButton()* but operates on *P1_9* instead of *P0_8*. The *SystemCounter* will have *TICKSPERBASE* = 100.

The *button_scanner* is called cyclically by the *run_button_scanner* alarm every 100ms and keeps checking if a button is pressed, once it happens we may have 3 different behaviors:

1. The button *P0_8* is pressed for the first time: "*Start*" is printed on the screen and a relative alarm run_t1 is set with a cycletime of 1s.
2. The button *P0_8* is pressed for the second time: the alarm is cancelled and "*Stop*" is printed on the screen.
3. The button *P1_9* is pressed: the system is shutdown as quickly as possible.

Snippet of .cpp file

```
TASK(t1)
{
   digitalWrite(3, !digitalRead(3));
   TerminateTask();
}

TASK(button_scanner)
{
  ButtonState        button        =
readButton9();
  if (button == BUTTON_PUSH){
        lcd.println("I'm off :(");
        ShutdownOS(E_OK);
      }

  static int b1=0;
  button = readButton();
  if (button == BUTTON_PUSH) {
    b1 = 1 - b1;
    if(b1){
        lcd.print("Start-->");
        SetRelAlarm(run_t1, 10, 10);
    }
    else{
        lcd.println(" -->Stop");
        CancelAlarm ( run_t1 );
    }
  }
```

Snippet of .oil file

```
TASK button_scanner {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK t1 {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  COUNTER SystemCounter {
    TICKSPERBASE = 100;
    MAXALLOWEDVALUE = 65535;
    MINCYCLE = 1;
  };

  ALARM run_t1 {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
      TASK = t1;
    };
    AUTOSTART = FALSE;
  };

  ALARM run_button_scanner {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
      TASK = button_scanner;
    };
    AUTOSTART = TRUE {
      APPMODE = stdMode;
      ALARMTIME = 1;
      CYCLETIME = 1;
```

	}; };

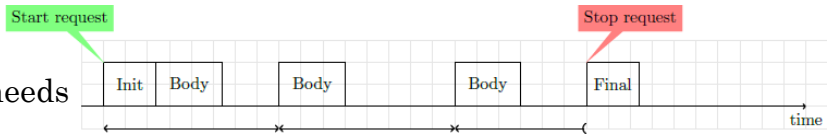
Pressed in time

Missed deadline

Switch off

Question 3.2 Modify the application.

Now function F implementation needs



- an *Init* code (runs once when the F is started);
- a *Final* code (runs once when F is stopped).

Init and *Final* print their names on the LCD.

The only difference from the previous question is the addition of two extra tasks. *Init* will be activated by *button_scanner* before setting *run_t1* while *Final* is activated button_scanner after cancelling the alarm.

Snippet of .cpp file	**Snippet of .oil file**

TASK(Init)
{
lcd.print("Init-->");
TerminateTask();
}
 	
TASK(Final)
{
lcd.println(" -->Final");
TerminateTask();
}
 	
TASK(button_scanner)
{
[…]
button = readButton();
if (button == BUTTON_PUSH) {
b1 = 1 - b1;
if(b1){
ActivateTask(Init);
SetRelAlarm(run_t1, 10, 10);
}
else{
CancelAlarm (run_t1);
	TASK Init {
	PRIORITY = 3;
	AUTOSTART = FALSE;
	ACTIVATION = 1;
	SCHEDULE = FULL;
	};
	
	TASK Final {
	PRIORITY = 3;
	AUTOSTART = FALSE;
	ACTIVATION = 1;
	SCHEDULE = FULL;
	};

```
        ActivateTask( Final );
    }
  }
  TerminateTask();
}
```

Question 3.3 Use only one extended task to implement function F.

To use a single extended task we must implement the following changes:

1. On the first press of P0_8 button_scanner will activate the Body task, on the second press it will set the *evt_f* event in Body.
2. Once activated the body task will print "*Init*" on the screen then set a relative alarm.
3. Body will enter an infinit while loop during it enter the wait state attending for an event (either *evt* or *evt_f*).
   3.1. If *evt_f* is found it will disable the event then exit the loop.
   3.2. Else,iIf the event *evt* is found (which is setted by the alarm) it will toggle the *LED L0_3*, disable *evt* and restart the loop.
4. Once out of the loop, Body will cancel the alarm, print on screen "*Final*" and terminate.

Snippet of .oil file

```
  EVENT evt {
    MASK = AUTO;
};

EVENT evt_f {
  MASK = AUTO;
};
```

```
TASK Body {
  PRIORITY = 3;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  SCHEDULE = FULL;
  EVENT = evt;
  EVENT = evt_f;
  };
```

Snippet of .cpp file

```
TASK(Body)
{
    EventMaskType received;

    lcd.print("Init-->");
    int i=1;
    SetRelAlarm ( run_t1, 10, 10);
    digitalWrite(3, HIGH);

    while (1) {
        WaitEvent(evt | evt_f);
        GetEvent(Body, &received);
        if(evt_f & received)
            break;

        ClearEvent(evt);
        digitalWrite(3, !digitalRead(3));
```

```
    }
    ClearEvent(evt_f);
    CancelAlarm ( run_t1 );
    lcd.println(" -->Final");
    TerminateTask();
}

TASK(button_scanner)
{
  […]
  static int b1=0;
  button = readButton();
  if (button == BUTTON_PUSH) {
    b1 = 1 - b1;
    if(b1){
        ActivateTask(Body);
    }
    else{
        SetEvent(Body, evt_f);
    }
  }
  TerminateTask();
}
```

4) **FOURTH** application

Watchdog - is a mechanism that allows to stop a processing or the waiting for an event when a deadline occurs.

Question 4.1

- Each time *P0_8* is pressed, *P1_9* must be pressed within 2s.
  - ➔ In such case, you print the time between the two occurrences.
  - ➔ Otherwise, an error message is displayed.

A *flag* variable is initially set to false. Then the *button_scanner* is called cyclically by the *run_button_scanner* alarm every 100ms and keeps checking if a button is pressed, once it happens we may have 3 different behaviors:

1. *P0_8* is pressed and *flag* is false: the *Body* task is activated and *flag* is set to true.
2. *P1_9* is pressed and *flag* is true: the event *evt_f* is set in *Body* and *flag* is set to false.
3. *P0_8* is pressed and *flag* is true or *P1_9* is pressed and *flag* is false: no action will be taken in this case.

Once activated - will print "*You have 2s.*." on screen, set a relative alarm (that will trigger *evt*) and waits for an event on of the following two will occur:

1. *evt_f* is received: *Body* will print on screen the time between the two occurrences.
2. *evt* is found an error message is displayed.

The alarm is then cancelled, the events reset, and *Body* terminates.

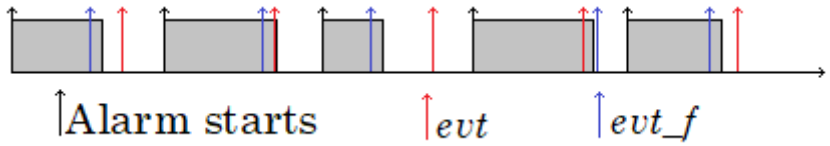
Snippet of .cpp file

```
TASK(Body)
{
    lcd.println("You have 2s..");
    EventMaskType received;
    TickType left;
    SetRelAlarm ( run_t1, 2000, 0);

// wait for button press or time runs
// out, check which event occurred.
    WaitEvent(evt | evt_f);
    GetEvent(Body, &received);

// if button print time elapsed
    if(evt_f & received) {
            GetAlarm(run_t1,&left);
        float time= (2000-
float(left))/1000;
        lcd.print("You wasted ");
        lcd.print(time);
        lcd.println("s of your life
here");
        ClearEvent(evt_f);
        ClearEvent(evt);
    }

// else print a message
    else {
        lcd.println("Toooo slow!");
        ClearEvent(evt);
    }

    CancelAlarm ( run_t1 );
    TerminateTask();
}

TASK(button_scanner)
{
    static bool flag=false; //setted
true when countdown starts
    ButtonState button = readButton();
if ((button == BUTTON_PUSH) & !flag) {
        flag=true;
        ActivateTask(Body);
    }

    button = readButton9();
    if ((button == BUTTON_PUSH) &
flag){
        SetEvent(Body, evt_f);
        flag=false;
    }
TerminateTask();
}
```

Snippet of .oil file

```
EVENT evt {
    MASK = AUTO;
};
EVENT evt_f {
  MASK = AUTO;
};

TASK button_scanner {
  PRIORITY = 3;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  SCHEDULE = FULL;
};

TASK Body {
  PRIORITY = 3;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  SCHEDULE = FULL;
  EVENT = evt;
  EVENT = evt_f;
  };

COUNTER SystemCounter {
  TICKSPERBASE = 1;
  MAXALLOWEDVALUE = 65535;
  MINCYCLE = 1;
};

ALARM run_t1 {
  COUNTER = SystemCounter;
  ACTION = SETEVENT{
  TASK= Body;
  EVENT=evt;
  };
  AUTOSTART = FALSE;
};

ALARM run_button_scanner {
  COUNTER = SystemCounter;
  ACTION = ACTIVATETASK {
    TASK = button_scanner;
  };
  AUTOSTART = TRUE {
    APPMODE = stdMode;
    ALARMTIME = 100;
    CYCLETIME = 100;
  };
};
```

Question 4.2 What is happening if the timeout occurs just after P1_9 has been pressed, but before the waiting task got the event? If your application does not handle correctly this scenario, modify it.



The Gantt diagram above shows all the possible situations that the application might have to deal with. The 1st, 3rd and 5th cycles represents the ideal situations: where P1_9 is pressed before the deadline expires. The 2nd cycle displays the situation in which the *evt* occurs before the program registers *evt_f*: here the output is still a time within 2s printed on screen. The 4th cycle displays the scenario where the second button isn't pressed and the error message is printed on the screen.

5) **FIFTH** application

In this application, we programmed a chase using 4 tasks that manages the leds and a button scanner that does the following:

- when *P0_8* is pressed, the chase stops;
- when *P1_9* is pressed, the chase continues;
- when *P2_10* is pressed, the chase direction changes (even if it is stopped).

To implement it we defined three global variable: *flag* (bool, initially false), *direction* (int, initially 0), and *current_LED* (int initially 3).

When started the *button_scanner* is called cyclically by the *run_button_scanner* alarm every 100ms and keeps checking if a button is pressed, once it happens we may have 4 different behaviors:

1. *P0_8* is pressed and *flag* is true: *flag* is set to false.
2. *P0_8* is pressed and *flag* is false or *P1_9* is pressed and *flag* is true: no action will be taken in this case.
3. *P1_9* is pressed and *flag* is false: *flag* is set to true, and using a switch that look at the *current_LED* value and activate the respective task *LED_i* with $i \in \{3,4,5,6\}$.
4. *P2_10* is pressed: the value of *direction* is increased of 1.

Once activated the *LED_i* task will check the value of *flag*. If *flag* is false it will just terminate itself. Differently, if *flag* is true it will switch off the led number *current_LED*, switch on the *i* led and set *current_LED* = *i*. It will then set a relative alarm that will ativated the *LED_j* task, with $j \in \{3,4,5,6\}$, $j \neq i$. This last decision will be taken by looking if *direction* is eve or odd.

Snippet of .cpp file

```
TASK(LED_3){
  if(flag){
    digitalWrite(current_LED, LOW);
    digitalWrite(3, HIGH);
    current_LED =3;
    if (direction%2 == 0)
      SetRelAlarm(A_4,100,0);
    else
      SetRelAlarm(A_6,100,0);
    };
  };
  TerminateTask();
}

TASK(LED_4){
  if(flag){
    digitalWrite(current_LED, LOW);
    digitalWrite(4, HIGH);
    current_LED=2;
    if (direction%2 == 0)
      SetRelAlarm(A_5,100,0);
    else
      SetRelAlarm(A_3,100,0);
  };
  TerminateTask();
}

TASK(LED_5){
  if(flag){
    digitalWrite(current_LED, LOW);
    digitalWrite(5, HIGH);
    current_LED=3;
    if (direction%2 == 0)
      SetRelAlarm(A_6,100,0);
    else
      SetRelAlarm(A_4,100,0);
  };
  TerminateTask();
}

TASK(LED_6){
  if(flag){
    digitalWrite(current_LED, LOW);
    digitalWrite(6, HIGH);
    current_LED=6;
  if (counter_direction%2 == 0)
      SetRelAlarm(A_3,100,0);
  else
      SetRelAlarm(A_5,100,0);
  };
  TerminateTask();
}

TASK(button_scanner)
{
  ButtonState button = readButton8();
```

Snippet of .oil file

```
TASK LED_3  {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK LED_4  {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK LED_5  {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK LED_6  {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK button_scanner  {
    PRIORITY = 2;
    AUTOSTART = TRUE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  ALARM run_button_scanner  {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK  {
      TASK = button_scanner;
    };
    AUTOSTART = TRUE  {
      APPMODE = stdMode;
      ALARMTIME = 100;
      CYCLETIME = 100;
    };
  };

  ALARM A_3  {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK  {
      TASK = LED_3;
    };
    AUTOSTART = FALSE;
  };

  ALARM A_4  {
```

```
  if (button == BUTTON_PUSH && flag){
    flag=false;
  };
  button = readButton9();
  if (button == BUTTON_PUSH && !flag){
      flag=true;
       switch(current_LED){
         case 3 :
           ActivateTask(LED_3);break;
         case 4 :
           ActivateTask(LED_4);break;
         case 5 :
           ActivateTask(LED_5);break;
         case 6 :
           ActivateTask(LED_6);break;
      };
   };
  button = readButton10();
  if (button == BUTTON_PUSH){
      direction++;
  };
 TerminateTask();
}
```

```
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK  {
      TASK = LED_4;
    };
    AUTOSTART = FALSE;
  };

  ALARM A_5 {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
      TASK = LED_5;
    };
    AUTOSTART = FALSE;
  };

  ALARM A_6 {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
      TASK = LED_6;
    };
    AUTOSTART = FALSE;
  };
```