Politecnico Di Torino

Department of CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Mechatronic Engineering

2024/2025

Operating Systems for Embedded Systems - Prof. Violante

Laboratory 02 – messages

Wednesday, October 30, 2024

Fornasier Francesca – 338688

Exercise 1

1.1) OIL file description

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

CPU test {
  OS config {
    STATUS = STANDARD;
    BUILD = TRUE {
      TRAMPOLINE_BASE_PATH = "/Users/francescafornasier/github/Operating_Systems_for_Embedded_Systems/trampoline";
      APP_NAME = "lab02ex1";
      APP_SRC = "ex1.cpp";
      CPPCOMPILER = "avr-g++";
      COMPILER = "avr-gcc";
      LINKER = "avr-gcc";
      ASSEMBLER = "avr-gcc";
      COPIER = "avr-objcopy";
      SYSTEM = PYTHON;
      LIBRARY = serial;
    };
    SYSTEM_CALL = TRUE;
  };

  APPMODE stdAppmode {};
```

The first part of the oil file contains directives for the oil compiler in order to create the proper configuration of the operating system for the specific configuration we need in our application.
We include the library “serial” so we can use the serial monitor of the ArduinoUno.

```
  ALARM aTaskC_100ms {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK { TASK = TaskC; };
    AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 100; CYCLETIME = 100; };
  };

  ALARM aTaskM_500ms {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK { TASK = TaskM; };
    AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 500; CYCLETIME = 500; };
  };

  ALARM aTaskV_250ms {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK { TASK = TaskV; };
    AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 250; CYCLETIME = 250; };
  };
```

For this exercise we need three periodic tasks so we define three periodic alarms that expires every 100, 500 and 250 milliseconds and the action associated with them is to activate TaskC, TaskM and TaskV respectively.
Cycle time of the TaskV is chosen to meet the most stringent requirement, that is making the led blink fast at 4Hz.
Since later in the oil file all the tasks have the property autostart equal to true (so they will be placed in the ready queue at the starting of the operating system), the alarmtime is set equal to the cycletime.

Since we want the communication between the tasks to be implemented using the message passing mechanism provided by OSEK, we have to define the message objects we are going to need in the application.

```
41  MESSAGE from_C {
42    MESSAGEPROPERTY = SEND_STATIC_INTERNAL {
43      CDATATYPE = "int";
44    };
45  };
46
47  MESSAGE M_receive {
48    MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL {
49      SENDINGMESSAGE = from_C;
50    };
51  };
```

We have two message objects for the communication between TaskC and TaskM: message from_C through which TaskC send an integer value to TaskM (the data type is specified in CDATATYPE parameter) and M_receive used by M to get the data sent by C; in SENDINGMESSAGE parameter we specify the unique identifier of the message used by the producer of the data.

```
53  MESSAGE from_M {
54    MESSAGEPROPERTY = SEND_STATIC_INTERNAL {
55      CDATATYPE = "int";
56    };
57  };
58
59  MESSAGE V_receive {
60    MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL {
61      SENDINGMESSAGE = from_M;
62    };
63  };
```

In the same way we implement the communication between TaskM and TaskV, with form_M used to send the message and V_receive used to receive it.

All messages are defined either as SEND_STATIC_INTERNAL or RECEIVE_UNQUEUED_INTERNAL since they are all sent and received within the same processor.

Description of the tasks:

```
65    TASK TaskV {
66      PRIORITY = 1;
67      AUTOSTART = TRUE { APPMODE = stdAppmode; };
68      ACTIVATION = 1;
69      SCHEDULE = FULL;
70      MESSAGE = V_receive;
71    };

73    TASK TaskC {
74      PRIORITY = 3;
75      AUTOSTART = TRUE { APPMODE = stdAppmode; };
76      ACTIVATION = 1;
77      SCHEDULE = FULL;
78      MESSAGE = from_C;
79    };
80
81    TASK TaskM {
82      PRIORITY = 2;
83      AUTOSTART = TRUE { APPMODE = stdAppmode; };
84      ACTIVATION = 1;
85      SCHEDULE = FULL;
86      MESSAGE = from_M;
87      MESSAGE = M_receive;
88    };
89  };
```

All tasks have autostart set to True, so they are immediately set as ready.
For each one we define the messages it has to handle and the priority.

Priorities

TaskC samples the input and the collected data determine the behavior of TaskM, so we need it to have the highest priority (3).
TaskM processes the data it receives from TaskC and sends messages that set the behavior of TaskV, its priority must therefore be lower than TaskC and greater than the one of TaskV (2).
TaskV only manages the led based on the variable received from TaskM and so it has the lowest priority (1).

1.2) C++ file description

```
#include "math.h"
#include "tpl_os.h"
#include "tpl_com.h"
#include "Arduino.h"

DeclareAlarm(aTaskC_100ms);
DeclareAlarm(aTaskM_500ms);
DeclareAlarm(aTaskV_250ms);
DeclareMessage(from_C);
DeclareMessage(M_receive);
DeclareMessage(from_M);
DeclareMessage(V_receive);
DeclareTask(TaskC);
DeclareTask(TaskM);
DeclareTask(TaskV);

const int switchMask = 0x800;  // 0000 0000  0000 0000  0000 1000  0000 0000
const int analogMask = 0x3FF;  // 0000 0000  0000 0000  0000 0011  1111 1111

void setup(){
    Serial.begin(115200);
    pinMode(13, OUTPUT);  // led
    pinMode(12, INPUT_PULLUP);  // switch
    pinMode(A0, INPUT);  // voltage input
    digitalWrite(13, LOW);
}
```

In the cpp file we define the behavior of our application.
In the first part of the cpp file we declare all the tasks, alarms and messages we defined in the oil file and two global integer variables used later in the code as bitmasks to work separately with the part of the input related to the analog input and the one depending on the switch.
In the setup function we initialise the communication over the serial port and we initialise the pins we are going to use: 13 (connected to the led) as output, digital pin 12 and analog pin A0 as input. Pin 13 is set to LOW in order to start the execution with the led off.

```
TASK(TaskC) {
    static int switchStart = 0;
    int message = analogRead(A0) & analogMask;

    if(digitalRead(12) == LOW) {
        if(switchStart - millis() >= 1000)  // if switch has been pressed for at least 1 second
            message = message | switchMask;  // set switch bit = 1
    }
    else switchStart = millis();

    Serial.print("C to M: ");
    Serial.println(message, BIN);

    SendMessage(from_C, &message);

    TerminateTask();
}
```

TaskC:
We define a static variable that will contain the time instant when we first sample the switch in the low state.
The message is initialised with the value we read from the analog pin connected with the input voltage. The value is put in a bit-wise ‘and’ condition with the proper bit mask to ensure the result is on 10 bits.
We read the switch state and if it is low we verify if it has been pressed for al least 1 second, if it is true we set the twelfth bit to 1 using a bitwise ‘or’ between the message itself and the bit mask for the switch, otherwise we update the switchStart variable.
We use the serial output to print which are the tasks involved in the communication and what is the value that iOS being transmitted (displayed in binary encoding for simplicity).

The message is sent using SendMessage system call.
The last operation is always a TerminateTask().

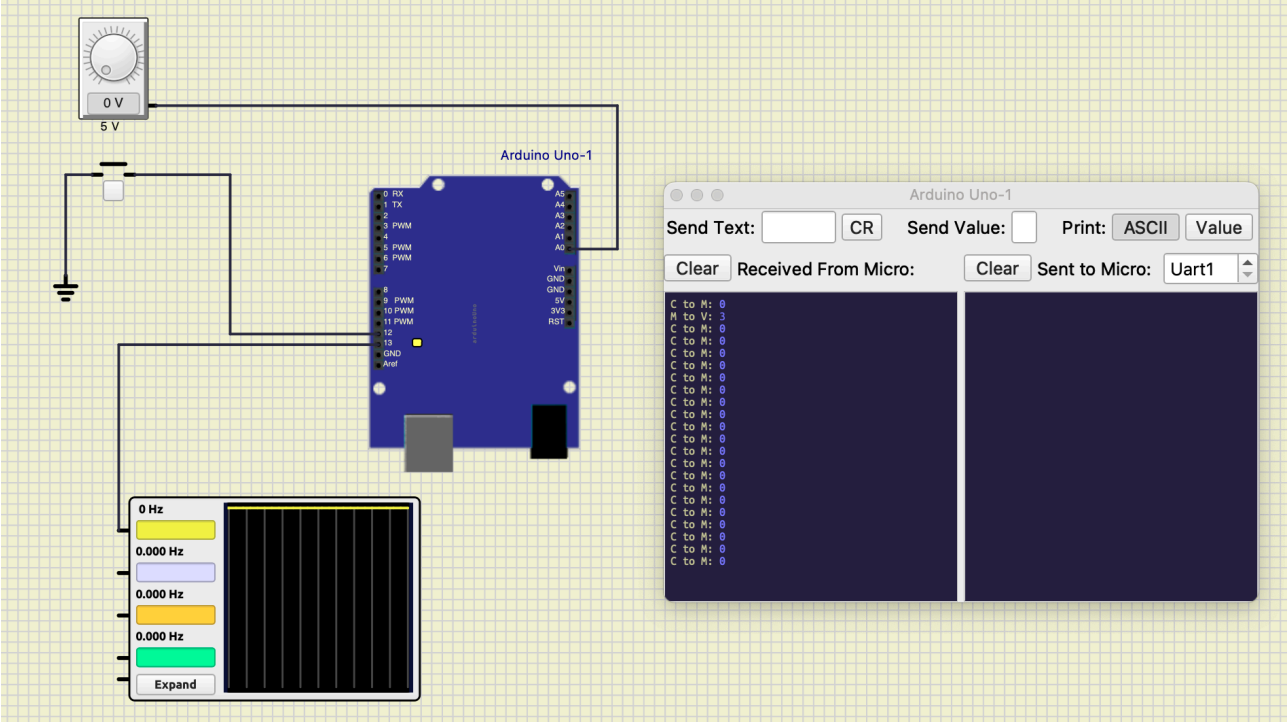
```
46  TASK(TaskM) {
47      int received_msg;
48      static int new_message = -1;
49      static int prev_received_msg = -1;
50      int analog_value;
51      static int referenceVal = -1;
52      int x;
53
54      if( ReceiveMessage(M_receive, &received_msg) != E_OK )
55          TerminateTask();
56
57      if (received_msg != prev_received_msg){  // don't compute again the new message if M has received a message = previous one
58          analog_value = received_msg & analogMask;
59
60          if(received_msg & switchMask) {  // if switch bit is set to 1, update reference value
61              referenceVal = analog_value;
62              Serial.print("NewR = ");
63              Serial.println(analog_value);
64          }
65          if(referenceVal == -1){
66              new_message = 3;  // led ON
67              Serial.println("M to V: 3");
68          }
69          else {
70              x = abs(analog_value-referenceVal);
71              if(x<100){
72                  new_message = 0;  // led OFF
73                  Serial.println("M to V: 0");
74              }
75              else if(x>=200){
76                  new_message = 2;  // blink fast (4 Hz)
77                  Serial.println("M to V: 2");
78              }
79              else{
80                  new_message = 1;  // blink slow (1 Hz)
81                  Serial.println("M to V: 1");
82              }
83          }
84      }
85      SendMessage(from_M, &new_message);
86      prev_received_msg = received_msg;
87      TerminateTask();
88  }
```

TaskM:
The task receives the message from C and check if it is different from the last one it received to avoid useless computations.
It extracts the analog value and a boolean indicating the switch state with a bitwise ‘&’, If this second value is true we update the reference value substituting it with the current analog value.
The integer value that the task must send to TaskV is is decided by a comparison between the current analog value and the reference value, if it is <100 it will send 0, if <200 it will send 1 and if it is >=200 it will send 2. In the case reference value has not been set yet, the task will send 3.
As for TaskC we use the serial monitor to show what are the messages exchanged between TaskM and TaskV.
As last operation before terminating, we update the value of the previous message with the value of the last received message.
For the reference value, current message and previously received message we use static variables because we need them to maintain their value between different executions of the task.

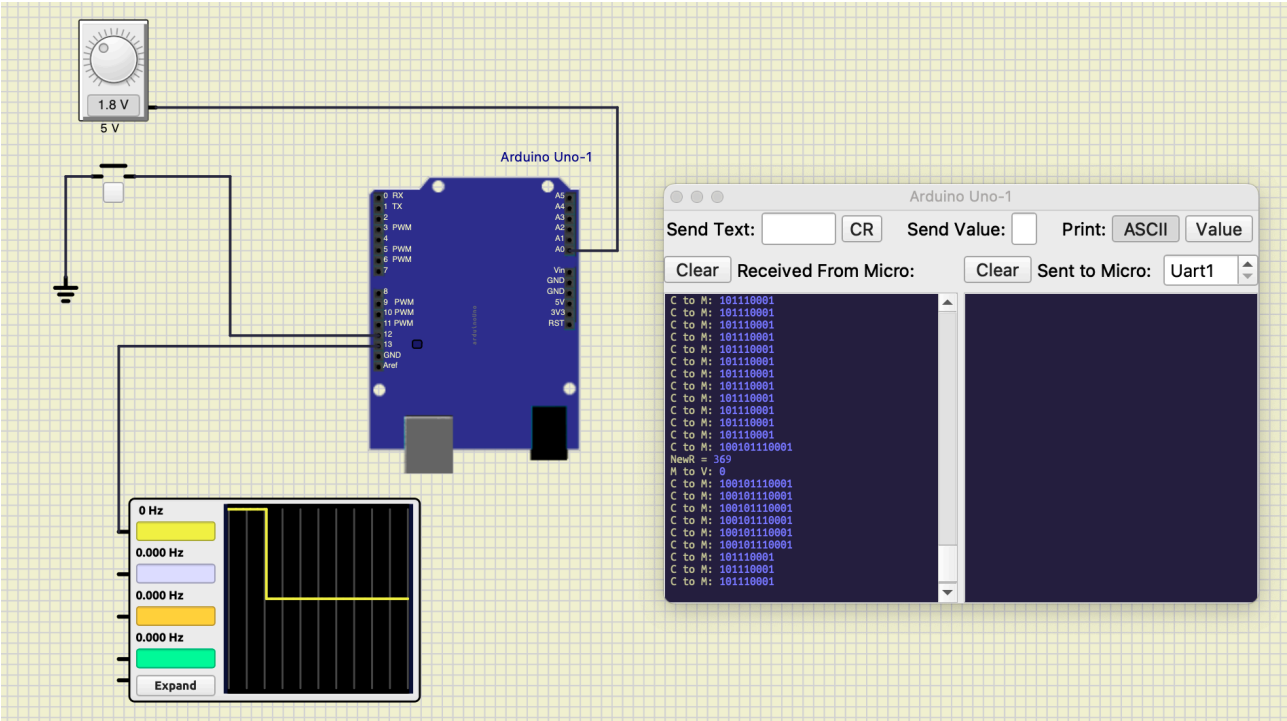
```
90  TASK(TaskV) {
91      int state = digitalRead(13);
92      int received;
93      static int Vcnt = 0;
94
95      if( ReceiveMessage(V_receive, &received) != E_OK )
96          TerminateTask();
97
98      if(received == 0){  // led is off
99          Vcnt = 0;  // reset value for next time we have to make the led blink slowly
100         digitalWrite(13, LOW);
101     }
102     else if(received == 1){  // blink slow: change led state every 4 executions
103         if(Vcnt == 0){
104             state = !state;
105             digitalWrite(13, state);
106         }
107         Vcnt++;
108         Vcnt = (Vcnt)%4;
109     }
110     else if(received == 2){  // blink fast: change led state at every execution
111         Vcnt = 0;  // reset value for next time we have to make the led blink slowly
112         state = !state;
113         digitalWrite(13, state);
114     }
115     else {  // received == 3, led is always on
116         digitalWrite(13, HIGH);
117     }
118
119     TerminateTask();
120 }
```

TaskV:
The task controls the led state according to the value of the message it receives from TaskM.
We use a variable to store the led state and a static variable as a counter
If the received value is 0 or 3 we respectively turn off or on the led writing low/high on the output.
If the message value is 2 the led must be blinking fast so at every execution of the task (every 250 ms, so 4 Hz) the led state must change.
If 1 is received the lest must blink slow, so we make the led state change every 1 second, corresponding to 4 executions of the task; for this purpose we use a static variable counter, that will be reset every time we get a message to impose a behavior that is different from this.

1.3) Output analysis



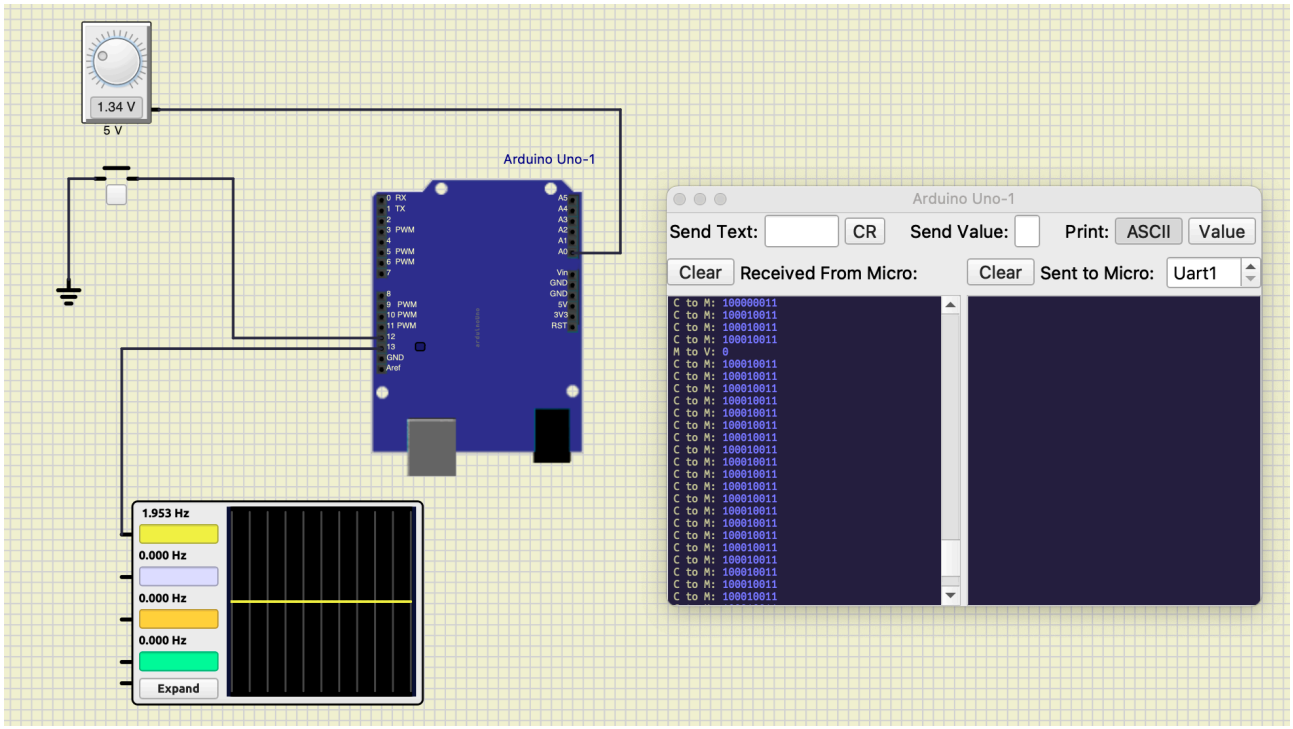
The reference has not been set yet, and the analog value is 0 so TaskM receives from TaskC an integer where the first 10 bit are 0 (analog value) and the also the twelfth bit is 0 because the button has not been pressed.
Since the reference value is still -1, TaskM will send 3 to TaskV and the output is high



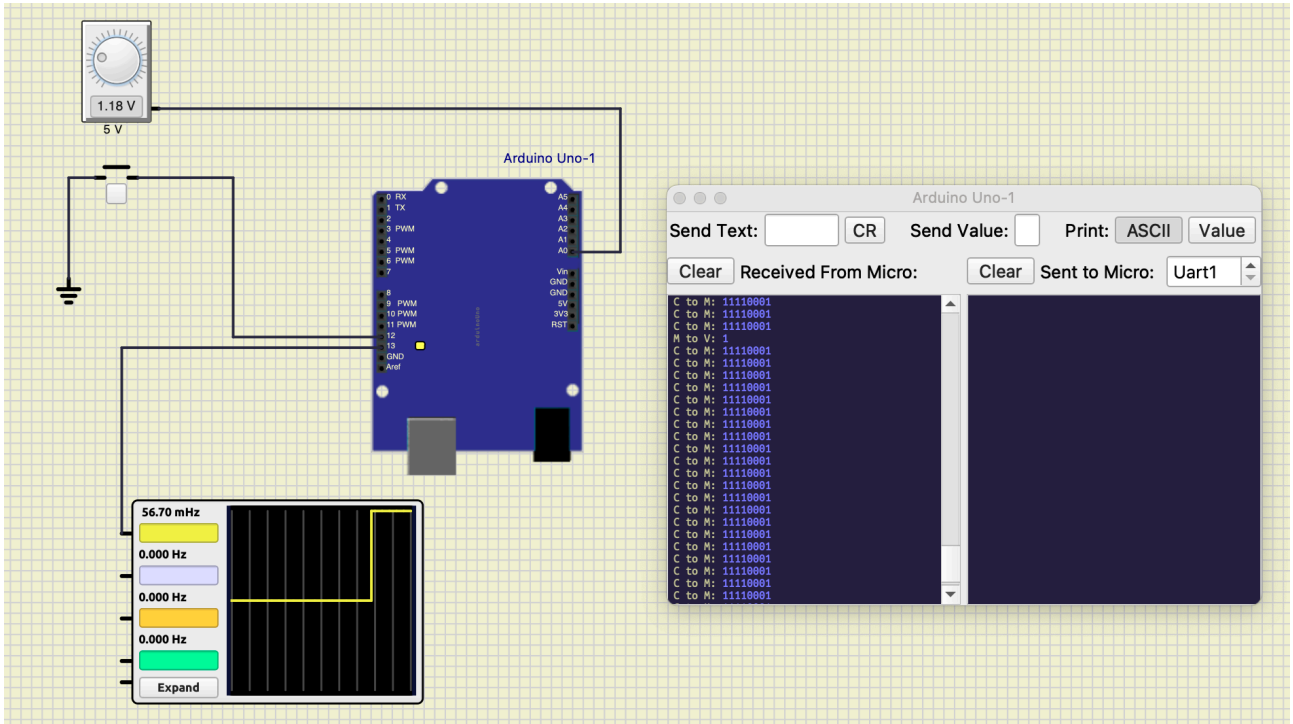
We changed the value of the analog input and then pressed the button for one second, we can see that bit 12 is set to 1 in the messages from C to M. The reference value is now updated with the value displayed on the serial monitor and at the led state is now set to LOW because the

comparison between the current value and the reference is now 0 (case x<100 of task M, value 0 will be sent to V)
After we release the button we can see in the messages from C to M that bit 12 is set again to 0.

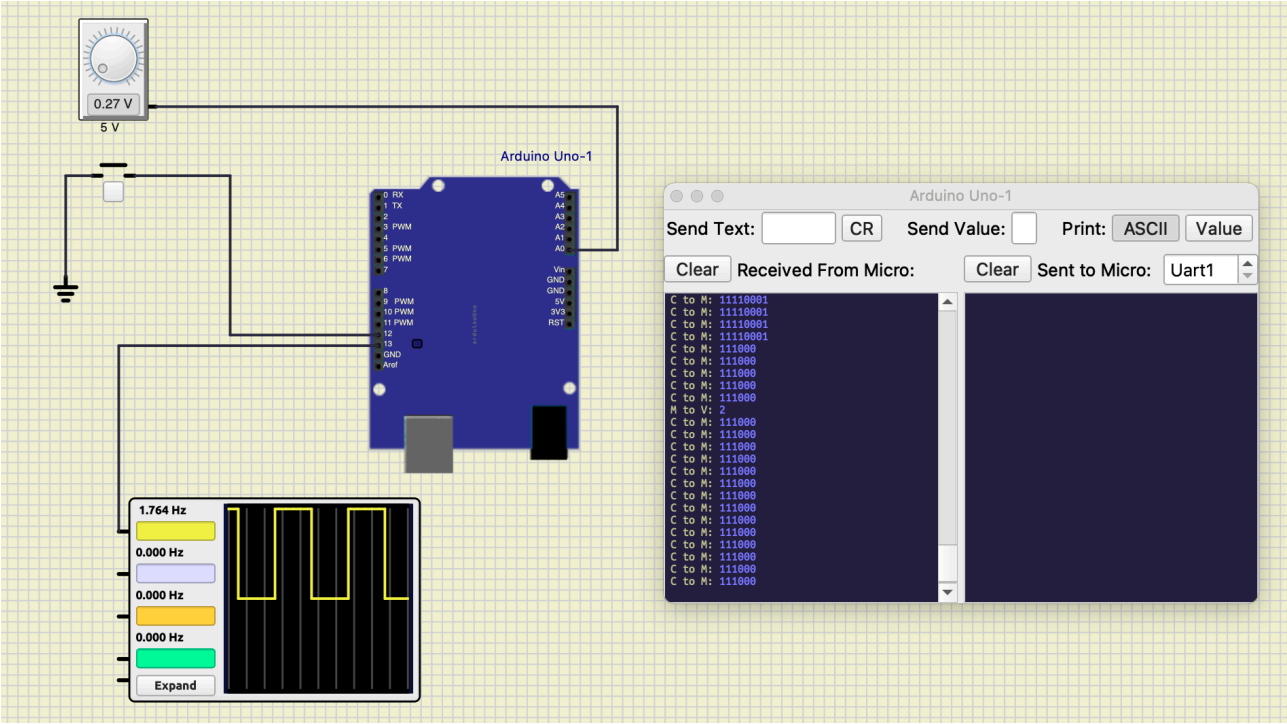
With the new reference value, we gradually change the analog input voltage.



We are still in the case x<100 of task M, so the message that will be sent is still 0 and the led will remain off



We modify the value more, the messages between C and M contain the new sampled analog voltage. The difference with the reference is now greater than 100, so M will send a message with 1 and task V make the led blink slowly according to the received value.



If we continue changing the value, we reach the condition x ≥ 200 in task M, that will send 2 to V as we can see in the serial monitor and TaskV will make the led blink faster.

1.4) Worst-case latency analysis

Since we set autostart property equal to 1 for all the tasks, at time instants 0, 500, 100 an so on all the tasks will be in the ready queue at the same time and then executed according to their priority. We have the worst case latency between a change in A0 value and the corresponding state of the led if the input variation happens right after TaskC sample it.
In this case the new value of the input will be sampled at the next execution of TaskC, at time 100ms, and elaborated at time 500 when the next instance of TaskM is executed. The corresponding TaskV is then executed right after TaskM.
Supposing the execution time of the tasks is negligible with respect to the smallest cycletime, the worst case latency will be equal to 500 milliseconds.

Exercise 2

2.1) OIL file description

Since the second exercise is very similar to the previous one, only a few modifications are required in the oil file, while the cpp file did not need any changes.

```
ALARM aTaskC_100ms {
  COUNTER = SystemCounter;
  ACTION = ACTIVATETASK { TASK = TaskC; };
  AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 100; CYCLETIME = 100; };
};

ALARM aTaskV_250ms {
  COUNTER = SystemCounter;
  ACTION = ACTIVATETASK { TASK = TaskV; };
  AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 250; CYCLETIME = 250; };
};
```

TaskM is no longer periodic so we don't need to set an alarm for it. We still need the alarms for TaskC and TaskV

```
MESSAGE from_C {
  MESSAGEPROPERTY = SEND_STATIC_INTERNAL {
    CDATATYPE = "int";
  };
};

MESSAGE M_receive {
  MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL {
    SENDINGMESSAGE = from_C;
  };
  NOTIFICATION = ACTIVATETASK{
    TASK = TaskM;
  };
};

MESSAGE from_M {
  MESSAGEPROPERTY = SEND_STATIC_INTERNAL {
    CDATATYPE = "int";
  };
};

MESSAGE V_receive {
  MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL {
    SENDINGMESSAGE = from_M;
  };
};
```

Message definition: TaskM has to be activated when there a message is sent, we can do it using the notification property of the receiving message object and specifying the action to be performed. In this case it is to activate TaskM.
All other messages are equal to those used in exercise 1.

```
62    TASK TaskV {
63      PRIORITY = 1;
64      AUTOSTART = TRUE { APPMODE = stdAppmode; };
65      ACTIVATION = 1;
66      SCHEDULE = FULL;
67      MESSAGE = V_receive;
68    };
69
70    TASK TaskC {
71      PRIORITY = 3;
72      AUTOSTART = TRUE { APPMODE = stdAppmode; };
73      ACTIVATION = 1;
74      SCHEDULE = FULL;
75      MESSAGE = from_C;
76    };
77
78    TASK TaskM {
79      PRIORITY = 2;
80      AUTOSTART = FALSE { APPMODE = stdAppmode; };
81      ACTIVATION = 1;
82      SCHEDULE = FULL;
83      MESSAGE = from_M;
84      MESSAGE = M_receive;
85    };
86  };
```

In the tasks definition we set the autostart property equal to false since it has to be activated only when a message is sent
The priorities of the tasks are the same of the exercise 1, the relative weight between different tasks has not changed.

2.2) Worst-case latency analysis

TaskM is activated every time TaskC sends a new message, so the sampled input is elaborated every 100 ms, right after the execution of C because of its priority.
The worst case latency happens when A0 changes after TaskC sampled it. The new value of the input will be sampled at the next execution of TaskC, at time 100ms, and elaborated immediately after with the execution of TaskM. However, to make the led change its state we have to wait until the new instance of TaskV is executed and it happens at time instant 250 ms.
Supposing the execution time of the tasks is negligible with respect to the smallest cycletime as in the previous exercise, the worst case latency will be equal to 250 milliseconds.

In conclusion, with the second approach to the problem (using an event driven TaskM) we improved our application reducing the worst case latency significantly reduced.