Province de Liège
Enseignement
Haute Ecole de la Province de Liège

Embedded systems programming.
Basic concepts.

Ir. Philippe CAMUS.
Chargé de cours.

CONTENTS.

FOREWORD.

The purpose of this text book is to review modern programming methods applied to embedded systems.

As we will see in the next chapter, embedded systems have specific characteristics related to their interactions with the outside world.

These characteristics require special programming methods.

This book is only an introduction, a "teaser" to the world of embedded system programming. It is intended for readers with a basic knowledge in embedded systems hardware, a beginner level in Python and an intermediate level in C/C++.

Apart from essential theoretical bases, this book is built on a "by example" teaching method.

Many examples are given, followed by exercises whose purpose is to allow the student to assess himself.

Most examples use one of the two following hardware platforms: Arduino Mega and Raspberry Pi Pico.

Given the time available for this course, we had to make choices. Working with a Raspberry Pi Zero or 4 would have been great too. For those who would be interested, starting kits are available. The software libraries for Python on Raspbian can be found here : https://gpiozero.readthedocs.io/

For the same reason, we will not work with coroutines. The coroutines allow multitasking in a cooperative way and seem to be an emerging and useful method, especially in MicroPyhton.

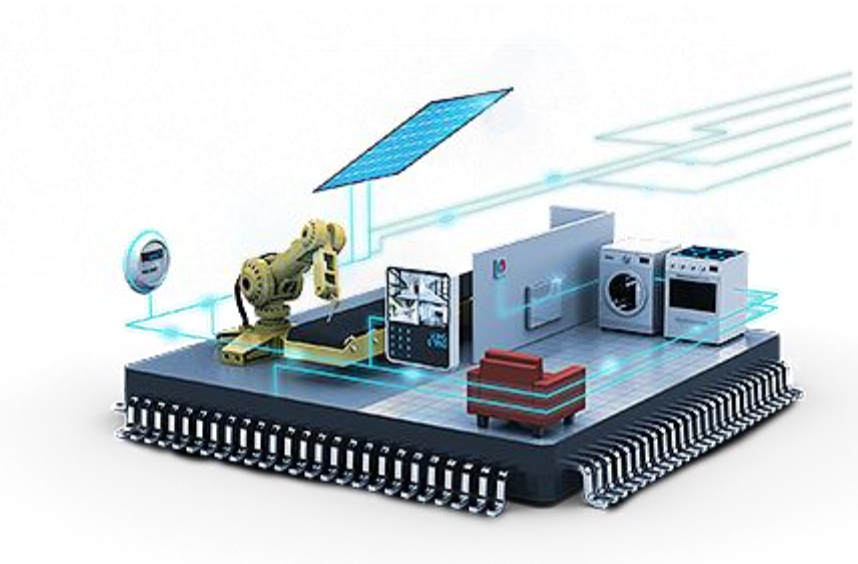
For further information, see:
https://en.wikipedia.org/wiki/Coroutine
https://docs.micropython.org/en/latest/library/uasyncio.html

Embedded systems.

An embedded system is a combination of hardware and software, connected to sensors, displays and actuators. It autonomously controls systems by collecting information in its environment with sensors.

It is most often built around a microcontroller, a hardware device made up of a microprocessor, RAM and flash memories and numerous peripheral controllers.

These systems are very popular and used in many fields: transportation, space missions, robotics, telecommunication, household appliances, home automation, medicine, power management and distribution, lighting, chemical or biochemical processes, industrial systems, games…



Picture from freepikpsd.com

Hardware and software architecture requirements.

The fact that embedded systems must be autonomous and sometimes control vital processes implies that they must have specific characteristics:

- have response time compliant with the process to control;
- have a low or even very low power consumption if they use batteries;
- be very reliable.

Response time.

When data arrive, the system must be able to take and process them without any loss.

It implies that the hardware and the software routines used to take and process the data must be fast enough to handle the flow of information without compromising the execution of the other tasks of the system.

It also implies being able to test your system to prove that these requirements are met.

These characteristics can be summarized in one expression: "**to be real time**".

Power consumption.

If your system is powered by batteries, a low power consumption is essential. That requires low power hardware with a sleep mode and adapted software architecture.

Reliability.

Reliability is "the quality of being able to be trusted or believed because of working or behaving well" *(Cambridge dictionary)*

Reliability can be divided in two parts: compliance with specifications and robustness.

Compliance with specifications.

Compliance is "the act of obeying a request". *(Cambridge dictionary)*
In this case, the request is a specification file given to describe how the system must perform.

Hardware compliance: the hardware must meet the requirements in terms of power consumption, operating temperature range, form factor, electromagnetic compatibility, security regulation, electrical standards…

Software compliance: the designer has the responsibility that his(her) software meets the requirements of the customer.

Developing a test workbench is mandatory and you have to think about testing from the start of the project.

See: *https://www.embedded.com/the-basics-of-embedded-software-testing-part-1/* for a comprehensive review of this problem.

A way to achieve software reliability is to choose the good computing platform with the good tools (language, libraries…) and the more appropriate programming paradigm.

Robustness.

Robustness is "the quality of being strong, and healthy or unlikely to break or fail". *(Cambridge dictionary)*

There is something more in robustness compared to compliance: a robust system should not fail even in unexpected situations, therefore not mentioned in the specification.

The unexpected situations arise from disturbances coming from the outside world.

Here is a brief list of problems which can compromise the proper functioning of the system:

- Vibrations or shocks damaging sensors.
- Electromagnetic perturbations (from high-power devices, from the lightning…) leading to bit value changes in variable or in the code.

- Radiation (charged particles from the sun, ionizing radiation, nuclear reactions…) leading to the destruction of crystalline structure of semiconductors.

Another problem is simply the fact that we have failed to meet the user requirements or failed to find bugs in our code…

Hardware robustness: to increase the robustness the first step is doing a good design and layout of the system with decoupling, filtering, ground planes… In extreme situations, we will have to use electromagnetic shields or radiation hardened components.

Software robustness: to increase software robustness we can implement a run-time mechanism to detect abnormal situations and to correct them or to switch to a fail-safe mode.

Here are some methods we can use in our code:

- Checking data from sensors to detect abnormal values.
- Checking for unexpected states of the system.
- Checking for run-time memory leaks (this problem must be addressed first in code testing and goes through the use of static data structures)
- Checking for data lost due to overrun.
- Checking for flash memory wear…

Checking means here detecting a potentially dangerous situation and correcting it or going to a fail-safe mode.

To give a higher level of robustness the following techniques can be used:

- A watchdog timer (a timer that resets the CPU if the program lost control).

- A dedicated system for controlling the main system

- Redundancy (two or more identical systems and an arbitration mechanism to detect differences in operation and to switch in a fail-safe mode)

PROGRAMMING PARADIGMS.

Ten shades of programming.

As stated previously, programs running in embedded systems are made of infinite loops because such systems never stopped, otherwise we will lose control of your device!

In C/C++ it leads to code like:

```
while (1)
  {
  //
  // do embbeded system stuff
  //
  }
```

In Python :

```
while True:
   #
   # do embbeded system stuff
   #
```

These loops can be used in different ways, called "programming paradigms":

- Simple loop.
- Procedural vs. object oriented.
- Loop with timing control.
- Simple loop, interrupt and FPGA blocks.
- I/O dedicated processors.
- Finite-State Machine.
- Event driven.
- Independent tasks.
    - Multiprocessing.
    - Processors sharing.
- Distributed computing.

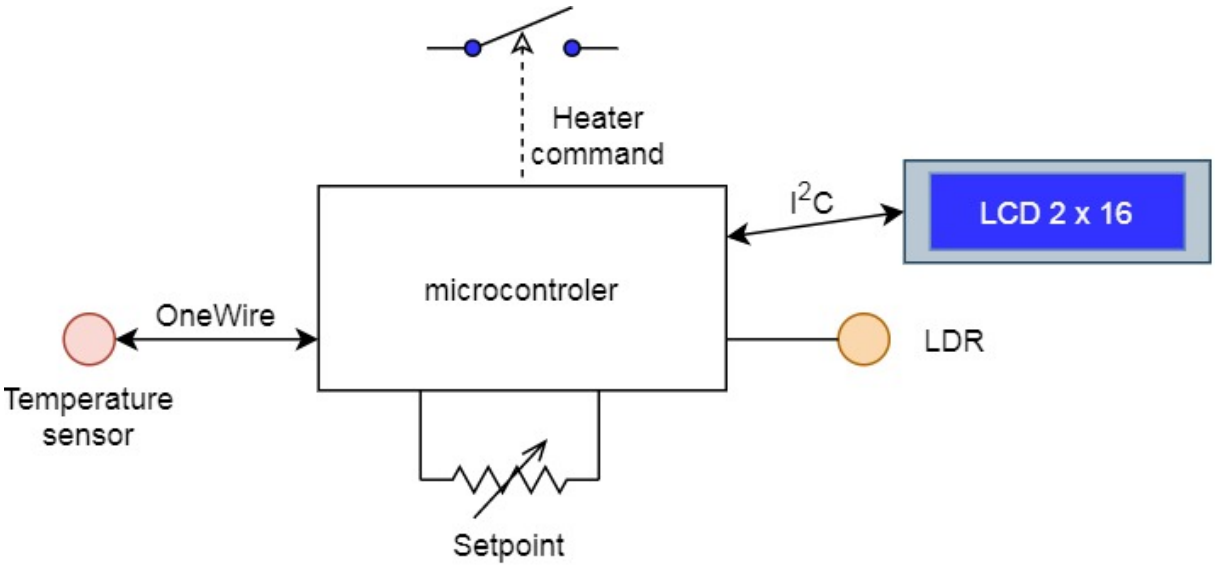
We will discover these paradigms in the following sections.

Illustration of each paradigm.

To illustrate each paradigm we will use examples of real devices and we will implement them on an Arduino Mega board (ATmega 2560) and/or a Raspberry Pi Pico board (RP 2040).

We will concentrate on two approaches: simple loop and RTOS. The other methods will be discussed briefly, they will be developed in other courses (for instance Linux Programming and ARM CPU programming).

A recurring example used through this book will be a simple thermostatic device whose block diagram is the following:

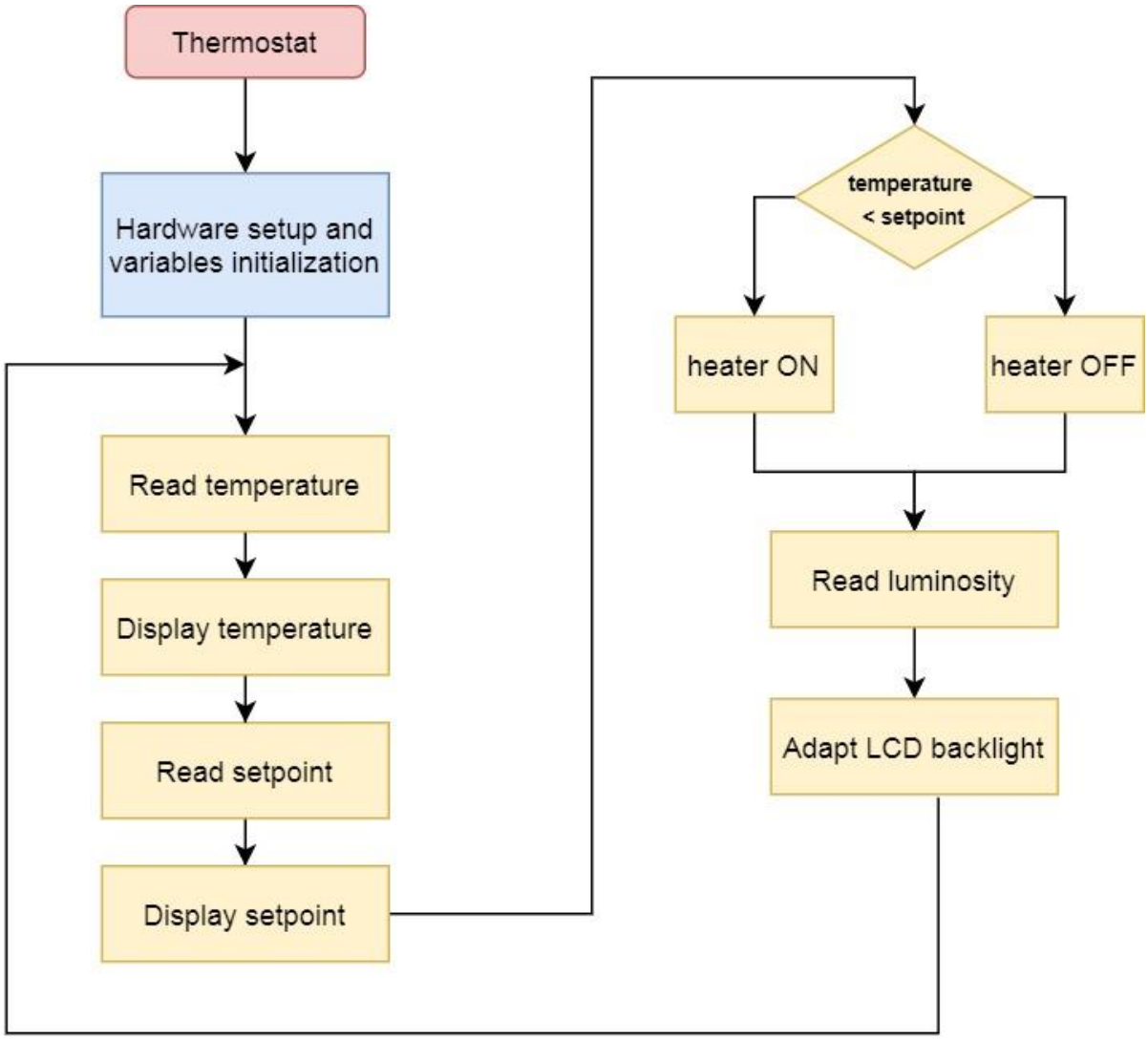


Its operating principle is straightforward: read the temperature, read the setpoint, compare them and switch the heater (via a relay) on or off accordingly.

The temperature and setpoint will be displayed on an LCD, whose brightness will be adjusted according to the ambient light measured by an LDR.

Regardless of the paradigm used to implement the software, we will need routines to read the sensors, to switch the heater, to display data on the LCD and to control its backlight. Some "glue logic" will be necessary to control these routine and to make decisions on how to control the whole system.

Flowchart.


Thermostat
Hardware setup and variables initialization
Read temperature
Display temperature
Read setpoint
Display setpoint
temperature < setpoint
heater ON
heater OFF
Read luminosity
Adapt LCD backlight

Simulated I/O thermostat version 1, in C/C++.

This first version follows the above flowchart and doesn't use any real sensors nor display. All I/O are simulated in a static way. We will see just after how to use stimuli that are more dynamic.

This example uses the Arduino Mega board, the software is made of three parts:

- the main program `therm_loop_sim.ino;`
- the header file for the utility routines `therm_loop_sim_util.h;`
- utility routines `therm_loop_sim_util.ino;`

Inputs are simulated by constants in the header file, output and display are simulated by messages send to the console.

To test different behaviour of the program you have to modify the header file, then recompile…

```
/**
 * @file therm_loop_sim.ino
 * @brief  simulation of thermostat
 * Uses therm_loop_sim_util.ino for functions code
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

#include "therm_loop_sim_util.h"

// the setup function runs once when you press reset or power the board
void setup()
{
// initialize console serial communication at 9600 bits per second:
Serial.begin(9600);

//hardware initialization sensors, display, output controler...
hardware_setup();
Serial.println("");
}

// run continuously
void loop()
{
uint16_t temperature;
uint16_t setpoint;
uint16_t luminosity;

temperature = read_temperature();
display_temperature(temperature);

setpoint = read_setpoint();
display_setpoint(setpoint);
```

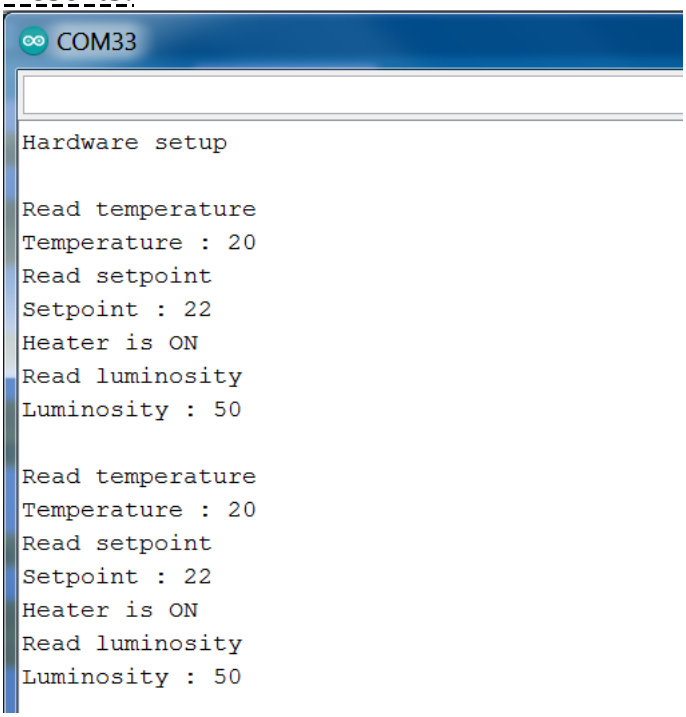
```
if (temperature < setpoint)
  set_heater(1);
 else
  set_heater(0);

luminosity = read_luminosity();
display_luminosity(luminosity);

delay(2500);
Serial.println("");
}
```

```
/**
 * @file therm_loop_sim_util.h
 * @brief  declaration of functions prototypes
 * and constants
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

#define DEFAULT_TEMPERATURE 20
#define DEFAULT_SETPOINT 22
#define DEFAULT_LUMINOSITY 50

void hardware_setup(void);
uint16_t read_temperature(void);
void display_temperature(uint16_t);
uint16_t read_setpoint(void);
void display_set_point(uint16_t);
uint16_t read_luminosity(void);
void display_luminosity(uint16_t);
void set_heater(uint8_t);
```

```
/**
 * @file therm_loop_sim_util.ino
 * @brief  simulation of thermostat
 * I/O access routines
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

void hardware_setup(void)
{
Serial.println("Hardware setup");
}

uint16_t read_temperature(void)
{
Serial.println("Read temperature");
return DEFAULT_TEMPERATURE;
}

void display_temperature(uint16_t temperature)
{
Serial.print("Temperature : ");
Serial.println(temperature);
```

```
}

uint16_t read_setpoint(void)
{
Serial.println("Read setpoint");
return DEFAULT_SETPOINT;
}

void display_setpoint(uint16_t setpoint)
{
Serial.print("Setpoint : ");
Serial.println(setpoint);
}

uint16_t read_luminosity(void)
{
Serial.println("Read luminosity");
return DEFAULT_LUMINOSITY;
}

void display_luminosity(uint16_t luminosity)
{
Serial.print("Luminosity : ");
Serial.println(luminosity);
}

void set_heater(uint8_t state)
{
if (state==1)
  Serial.println("Heater is ON");
 else
  Serial.println("Heater is OFF");
}
```

Results:



Simulated I/O thermostat version 1, in Python.

Same example as the previous one, but in Python running on Thonny.

The software is made of two parts:

- the main program: `therm_loop_sim.py;`
- module with utility routines: `therm_loop_sim_util.py;`

```
# file therm_loop_sim.py
# simulation of thermostat
# author philippe.camus@hepl.be
# date 15/8/2021

from therm_loop_sim_util import *
import time

hardware_setup()
print()

while(True):
    temperature = read_temperature()
    display_temperature(temperature)

    setpoint = read_setpoint()
    display_setpoint(setpoint)

    if (temperature < setpoint):
        set_heater(1)
    else:
        set_heater(0)

    luminosity = read_luminosity()
    display_luminosity(luminosity)

    time.sleep(2.5) # delay of 2.5 seconds
    print()
```

```
# file therm_loop_sim_util.py
# simulation of thermostat
# I/O access routines
# author philippe.camus@hepl.be
# date 15/8/2021

def hardware_setup():
    print("Hardware setup")

def read_temperature():
    print("Read temperature")
    return 20


def display_temperature(temp):
    print("Temperature : ",temp)
```

```
def read_setpoint():
    print("Read setpoint")
    return 22

def display_setpoint(setp):
    print("Setpoint : ",setp)

def read_luminosity():
    print("Read luminosity")
    return 50

def display_luminosity(lum):
    print("Luminosity : ",lum)

def set_heater(state):
    if (state==0):
        print("Heater is OFF")
    else:
        print("Heater is ON")

def check_lib():
    hardware_setup()
    temperature = read_temperature()
    display_temperature(temperature)
    setpoint = read_setpoint()
    display_setpoint(setpoint)
    luminosity = read_luminosity()
    display_luminosity(luminosity)
    set_heater(0)
    set_heater(1)

if __name__ == "__main__":
    # execute only if run as the main module (i.e. not an import module)
    check_lib()
```

Results:

Shell

```
>>> %Run therm_loop_sim.py
 Hardware setup

 Read temperature
 Temperature :  20
 Read setpoint
 Setpoint :  22
 Heater is ON
 Read luminosity
 Luminosity :  50

 Read temperature
 Temperature :  20
 Read setpoint
 Setpoint :  22
 Heater is ON
 Read luminosity
 Luminosity :  50
```

The Python corner.

Modules.

To make your code easier to read and maintain, it is useful to split it in several files. You will have a "main" file where the execution starts and other files called "modules" containing additional parts of your code.

The modules, also called packages or libraries, can be developed by other computer scientists, as numpy, or by yourself.

A module is simply a Python file. To use the contents of a module you have to import it first.

You can import it and keep the function and variable names used in the module:

```
from therm_loop_sim_util import *

# call a function of the module :
hardware_setup()
```

You can import a module and use its name as a prefix for the function and variable names used in the module:

```
import time

# call a function of the module :
time.sleep(2.5)  # function is prefixed by module name
```

You can import a module and use your own name as a prefix for the function and variable names used in the module:

```
import therm_loop_sim_util as therm_util

# call a function of the module :
therm_util.hardware_setup() # function is prefixed by your own name
```

You can import only a few functions of the module:

```
from time import sleep

# call a function of the module :
sleep(2.5)  # function is not prefixed
```

As a module is not different from regular Python code, you can run it standalone. By comparing the value of the internal variable **__name__** with the constant **"__main__"**, it is possible to detect if you are running the code in a standalone way or not.

This allows you to add some code to test your module. This code will not be run if the module is imported.

```
def check_lib():
    hardware_setup()
    temperature = read_temperature()
    display_temperature(temperature)
    setpoint = read_setpoint()
    display_setpoint(setpoint)
    luminosity = read_luminosity()
    display_luminosity(luminosity)
    set_heater(0)
    set_heater(1)

if __name__ == "__main__":
    # execute only if run as the main module (i.e. not an import module)
    check_lib()
```

Simulated I/O thermostat version 2, in C/C++.

Improvement of the version 1, we are going to use stimuli vectors instead of a constant stimulus.

This example uses the Arduino Mega board, the software is made of three parts:

- the main program `therm_loop_sim2.ino;`
- the header file for the utility routines `therm_loop_sim_util2.h`*(this header will contain the stimuli vectors);*
- utility routines `therm_loop_sim_util2.ino;`

```
/**
 * @file therm_loop_sim2.ino
 * @brief  simulation of thermostat
 * Uses therm_loop_sim_util.ino for functions code
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

#include "therm_loop_sim_util2.h"

// the setup function runs once when you press reset or power the board
void setup()
{
// initialize console serial communication at 9600 bits per second:
Serial.begin(9600);

//hardware initialization sensors, display, output controller...
hardware_setup();
Serial.println("");
}

// run continuously
void loop()
{
uint16_t temperature;
uint16_t setpoint;
uint16_t luminosity;

temperature = read_temperature();
display_temperature(temperature);

setpoint = read_setpoint();
display_setpoint(setpoint);

if (temperature < setpoint)
  set_heater(1);
 else
  set_heater(0);

luminosity = read_luminosity();
display_luminosity(luminosity);
```

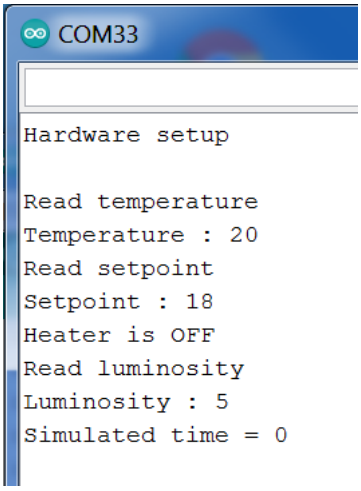
```
inc_time();
delay(2500);
Serial.println();
}
```

```
/**
 * @file therm_loop_sim_util2.h
 * @brief  declaration of functions prototypes
 * and constants
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

#define SIZE_OF_SIM 12
const uint16_t temp_list[SIZE_OF_SIM]={20,20,20,21,21,21,22,22,22,23,23,23};
const uint16_t setp_list[SIZE_OF_SIM]={18,20,22,24,21,20,22,21,20,20,20,20};
const uint16_t lum_list[SIZE_OF_SIM]={5,10,35,50,75,100,5,10,35,50,75,100};

uint8_t sim_time=0;

void hardware_setup(void);
uint16_t read_temperature(void);
void display_temperature(uint16_t);
uint16_t read_setpoint(void);
void display_set_point(uint16_t);
uint16_t read_luminosity(void);
void display_luminosity(uint16_t);
void set_heater(uint8_t);
void inc_time(void);
```

```
/**
 * @file therm_loop_sim_util2.ino
 * @brief  simulation of thermostat
 * I/O access routines
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

void hardware_setup(void)
{
Serial.println("Hardware setup");
}

uint16_t read_temperature(void)
{
Serial.println("Read temperature");
return temp_list[sim_time];
}

void display_temperature(uint16_t temperature)
{
Serial.print("Temperature : ");
Serial.println(temperature);
}
```

```
uint16_t read_setpoint(void)
{
Serial.println("Read setpoint");
return setp_list[sim_time];
}

void display_setpoint(uint16_t setpoint)
{
Serial.print("Setpoint : ");
Serial.println(setpoint);
}

uint16_t read_luminosity(void)
{
Serial.println("Read luminosity");
return lum_list[sim_time];
}

void display_luminosity(uint16_t luminosity)
{
Serial.print("Luminosity : ");
Serial.println(luminosity);
}

void set_heater(uint8_t state)
{
if (state==1)
  Serial.println("Heater is ON");
 else
  Serial.println("Heater is OFF");
}

void inc_time(void)
{
Serial.print("Simulated time = ");Serial.println(sim_time);
sim_time++;
if (sim_time>=SIZE_OF_SIM)
  sim_time=0;
}
```

Results:




```
Read temperature
Temperature : 20
Read setpoint
Setpoint : 20
Heater is OFF
Read luminosity
Luminosity : 10
Simulated time = 1

Read temperature
Temperature : 20
Read setpoint
Setpoint : 22
Heater is ON
```

Simulated I/O thermostat version 2, in Python.

Improvement of the Python version 1, running on Thonny. It uses lists of stimuli instead of a constant stimulus.

The software is made of two parts:

- the main program: `therm_loop_sim2.py;`
- module with utility routines: `therm_loop_sim_util2.py;`

```
# file therm_loop_sim2.py
# simulation of thermostat
# author philippe.camus@hepl.be
# date 15/8/2021

from therm_loop_sim_util2 import *
#import time
from time import sleep

hardware_setup()
print()

while(True):
    temperature = read_temperature()
    display_temperature(temperature)

    setpoint = read_setpoint()
    display_setpoint(setpoint)

    if (temperature < setpoint):
        set_heater(1)
    else:
        set_heater(0)

    luminosity = read_luminosity()
    display_luminosity(luminosity)

    inc_time()
    sleep(2.5) # delay of 2.5 seconds
    print();
```

```
# file therm_loop_sim_util2.py
# simulation of thermostat
# I/O access routines
# author philippe.camus@hepl.be
# date 15/8/2021

temp_list=[20,20,20,21,21,21,22,22,22,23,23,23]
setp_list=[18,20,22,24,21,20,22,21,20,20,20,20]
lum_list=[5,10,35,50,75,100,5,10,35,50,75,100]
length_list=len(temp_list)
sim_time=0
```

```
def hardware_setup():
    print("Hardware setup")

def read_temperature():
    print("Read temperature")
    return temp_list[sim_time]

def display_temperature(temp):
    print("Temperature : ",temp)

def read_setpoint():
    print("Read setpoint")
    return setp_list[sim_time]

def display_setpoint(setp):
    print("Setpoint : ",setp)

def read_luminosity():
    print("Read luminosity")
    return lum_list[sim_time]

def display_luminosity(lum):
    print("Luminosity : ",lum)

def set_heater(state):
    if (state==0):
        print("Heater is OFF")
    else:
        print("Heater is ON")

def inc_time():
    global sim_time
    print("Simulated time = ", sim_time)
    sim_time = sim_time+1
    if (sim_time>=length_list):
        sim_time=0


def check_lib():
    hardware_setup()
    temperature = read_temperature()
    display_temperature(temperature)
    setpoint = read_setpoint()
    display_setpoint(setpoint)
    luminosity = read_luminosity()
    display_luminosity(luminosity)
    set_heater(0)
    set_heater(1)


if __name__ == "__main__":
    # execute only if run as the main module (i.e. not an import module)
    check_lib()
```

Results:

```
Shell
>>> %Run therm_loop_sim2.py
 Hardware setup

 Read temperature
 Temperature :  20
 Read setpoint
 Setpoint :  18
 Heater is OFF
 Read luminosity
 Luminosity :  5
 Simulated time =  0

 Read temperature
 Temperature :  20
 Read setpoint
 Setpoint :  20
 Heater is OFF
 Read luminosity
 Luminosity :  10
 Simulated time =  1
```

Modifying code to get stimuli from an Excel spreadsheet.

Replace :

```
temp_list=[20,20,20,21,21,21,22,22,22,23,23,23]
setp_list=[18,20,22,24,21,20,22,21,20,20,20,20]
lum_list=[5,10,35,50,75,100,5,10,35,50,75,100]
```

with:

```
import openpyxl

# Extract column data from excel file and store them in 3 lists
wb = openpyxl.load_workbook("thermsim.xlsx")
sheet = wb["sim1"]

temp=sheet["A"]
temp_list = [temp[x].value for x in range(1,len(temp))]

setp=sheet["B"]
setp_list = [setp[x].value for x in range(1,len(setp))]

lum=sheet["C"]
lum_list = [lum[x].value for x in range(1,len(lum))]

wb.close()
```

In this example, we use the following sheet:

	A	B	C	D	E
1	Temperature	Setpoint	Luminosity		
2	20	18	5		
3	20	20	10		
4	20	22	35		
5	21	24	50		
6	21	21	75		
7	21	20	100		
8	22	22	5		
9	22	21	10		
10	22	20	35		
11	23	20	50		
12	23	20	75		
13	23	20	100		
14					
15					

The Python corner.

Lists.

Lists are very important data structure in Python. The lists are used to store several data in a single variable, in an ordered way. Different types of data can be stored in the same list.

The lists are indexed, with index starting at 0. There are numerous methods that you can apply on list (adding or deleting elements, searching the list, doing conversion…), see *https://www.w3schools.com/python/python_lists.asp*.

Example :

```
setp_list=[18,20,22,24,21,20,22,21,20,20,20,25]

print("length of list is",len(setp_list),sep=" : ")

print("setp_list[3] = ", end="") # don't go to the line
print(setp_list[3],end="\n\n") # goto the line and skip one line

print() #skip one more line

new_list1 =  setp_list[:3] # extract sublist

print("setp_list[:3]")
```

```
for data in new_list1:
    print(data)

print()# skip one line
print("setp_list[9:]")
new_list1 =  setp_list[9:] # extract sublist

for data in new_list1:
    print(data)
```

Results:

```
Shell ×
>>> %Run list.py
 length of list is : 12
 setp_list[3] = 24


 setp_list[:3]
 18
 20
 22

 setp_list[9:]
 20
 20
 25
```

Access to Excel spreadsheets.

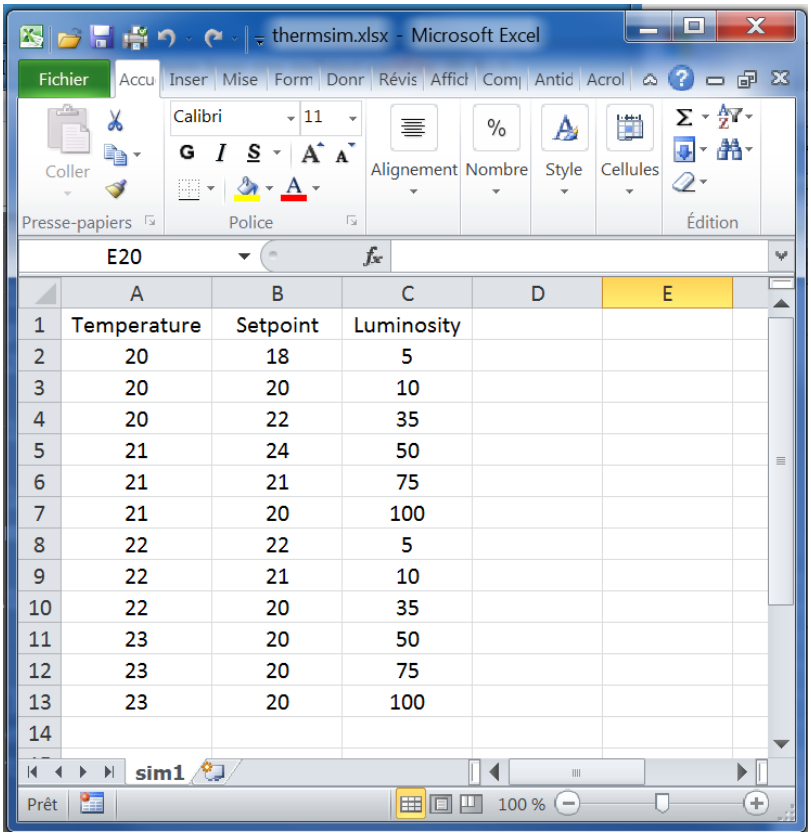
The **openpyxl** module gives you an easy way to interact with Excel spreadsheets, see *https://openpyxl.readthedocs.io/*.

Simple examples of reading and writing:

```
# file test excel.py
# Tests with openpyxl
# uses openpyxl library : https://openpyxl.readthedocs.io/
# author philippe.camus@hepl.be
# date 18/8/2021

import openpyxl

filename="thermsim.xlsx"

wb = openpyxl.load_workbook(filename) # open spreadsheet
sheet = wb["sim1"] # chose sheet sim1
```

```
print(sheet["A1"].value) # a single value (A1)
#print(sheet.cell(1,1).value) # equivalent to previous line
print()

temp1=sheet["A"] # column A
temp1_list = [temp1[x].value for x in range(1,len(temp1))]
print(temp1_list)
print()

temp2=sheet["1"] # line 1
temp2_list = [temp2[x].value for x in range(len(temp2))]
print(temp2_list)

# change a single cell:
sheet["A2"].value=-2
#sheet.cell(2,1).value=-2 # equivalent to previous line


# change a complete line
for i,data in enumerate(temp2_list):
    sheet.cell(1,i+1).value=data.upper() # i+1 because cells start at 1 not 0

wb.save(filename) # write changes into file

wb.close()
```

Spreadsheet before running code:

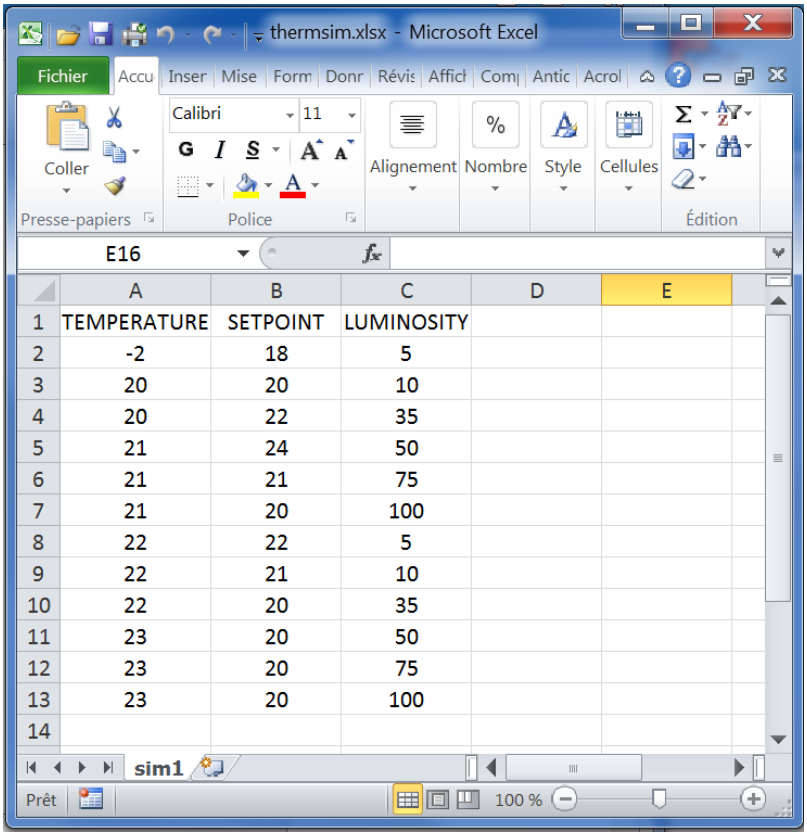


	A	B	C	D	E
1	Temperature	Setpoint	Luminosity		
2	20	18	5		
3	20	20	10		
4	20	22	35		
5	21	24	50		
6	21	21	75		
7	21	20	100		
8	22	22	5		
9	22	21	10		
10	22	20	35		
11	23	20	50		
12	23	20	75		
13	23	20	100		
14					

Results:

```
Shell
>>> %Run 'test excel.py'
 Temperature

 [20, 20, 20, 21, 21, 21, 22, 22, 22, 23, 23, 23]

 ['Temperature', 'Setpoint', 'Luminosity']
```

Spreadsheet after running code:



	A	B	C	D	E
1	TEMPERATURE	SETPOINT	LUMINOSITY		
2	-2	18	5		
3	20	20	10		
4	20	22	35		
5	21	24	50		
6	21	21	75		
7	21	20	100		
8	22	22	5		
9	22	21	10		
10	22	20	35		
11	23	20	50		
12	23	20	75		
13	23	20	100		
14					

`enumerate(temp2_list)`returns a list of tuples containing the index and value of elements in the list.

```
[(0, 'Temperature'), (1, 'Setpoint'), (2, 'Luminosity')]
```

Also, consider using Google sheets (module gspread) to create sheets in the cloud you can share with others. See *https://gspread.readthedocs.io/*.

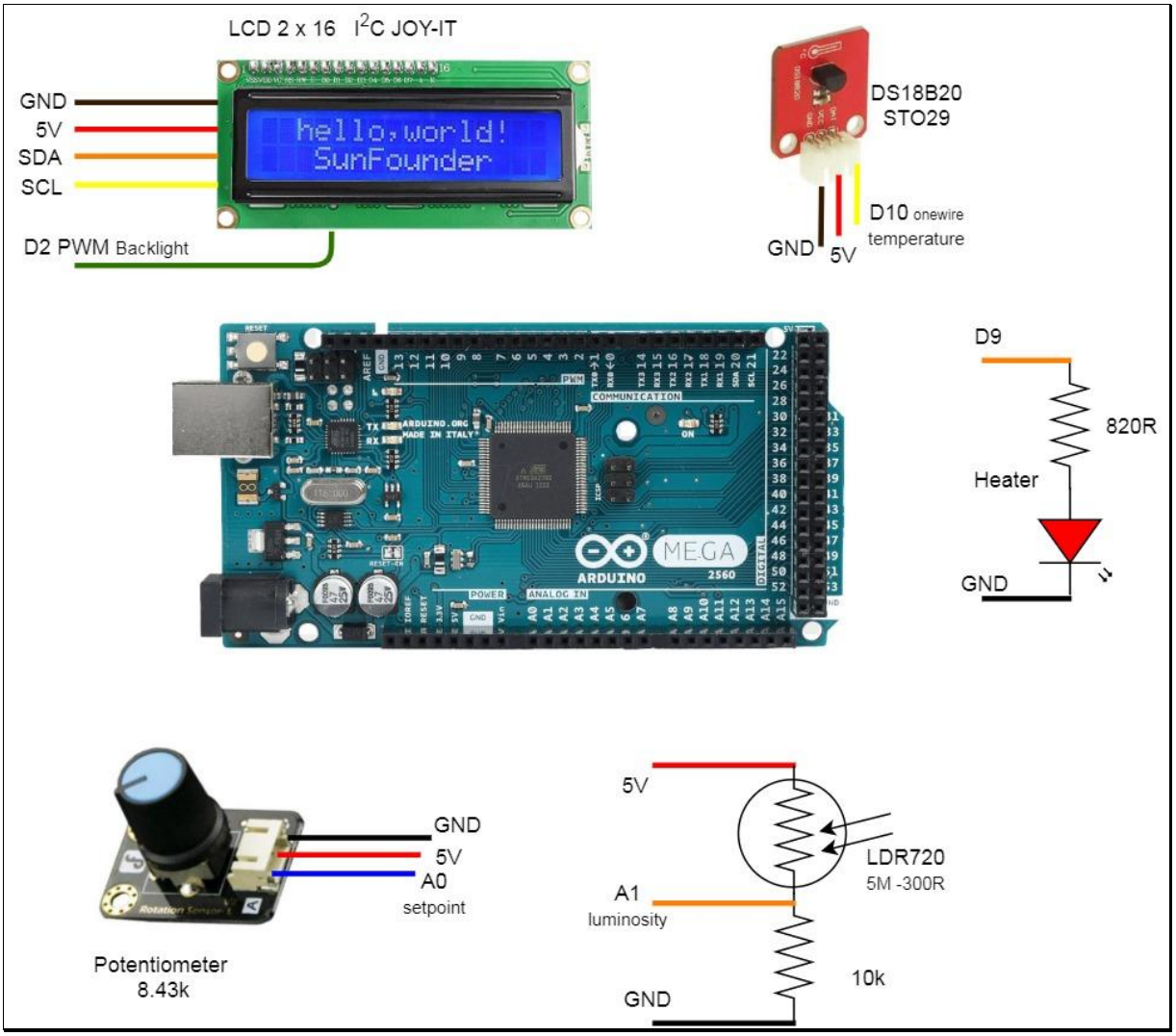
Exercises for the simulated I/O thermostat.

a) For the C/C++ version modify the code to add a temperature generator. At each call of the `read_temperature()` *function, a value is produced following a symmetric sawtooth. The sawtooth increases by one degree steps from 18 to 26 and then decreases to 18 and so on (it's more a stair case than a sawtooth).*

The setpoint will be fixed to 22 degrees and the luminosity to 50.

b) For the Python version, use global variables instead of returned values.

Thermostat with sensors and display, Arduino version.



Program for the Arduino in C/C++.

The software is made of three parts:

- the main program `therm_loop_v1.ino;`
- the header file for the utility routines `therm_loop_util_v1.h`
- utility routines `therm_loop_util_v1.ino;`

Starting with the simulated version, it is only necessary to replace stimuli with calls to appropriate I/O object or function. You can let leave the console interaction for the debug. Just put console instruction between `#ifdef DEBUG ...`

`#endif` preprocessing instruction for conditional compilation. If `DEBUG` symbol is defined the instructions will be compiled, otherwise they will be skipped.

```
/**
 * @file therm_loop_v1.ino
 * @brief  thermostat
 * Uses therm_loop_util_v1.ino for functions code
 * @author philippe.camus@hepl.be
 * @date 19/8/2021
 */

// Necessary libraries
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define HEATER_LED 9
#define DS18B20_PIN 10
#define POT A0
#define LDR A1
#define BACKLIGHT 2
#define ON 1
#define OFF 0

//#define DEBUG // in normal mode comment out this line

// create I/O objects
LiquidCrystal_I2C lcd(0x27,20,2);  // set the LCD address to 0x27 for a 16 chars
                                   //and 2 line display
OneWire oneWire(DS18B20_PIN); // set up oneWire to pin 10 (DS18B20 connection)
DallasTemperature sensors(&oneWire); // create a DS18B20 object

#include "therm_loop_util_v1.h" // header for your functions


// the setup function runs once when you press reset or power the board
void setup()
{
#ifdef DEBUG
// initialize console serial communication at 9600 bits per second:
Serial.begin(9600);
#endif

//hardware initialization sensors, display, output controler...
hardware_setup();
}

// run continuously
void loop()
{
uint16_t temperature;
uint16_t setpoint;
uint16_t luminosity;

temperature = read_temperature();
display_temperature(temperature);

setpoint = read_setpoint();
```

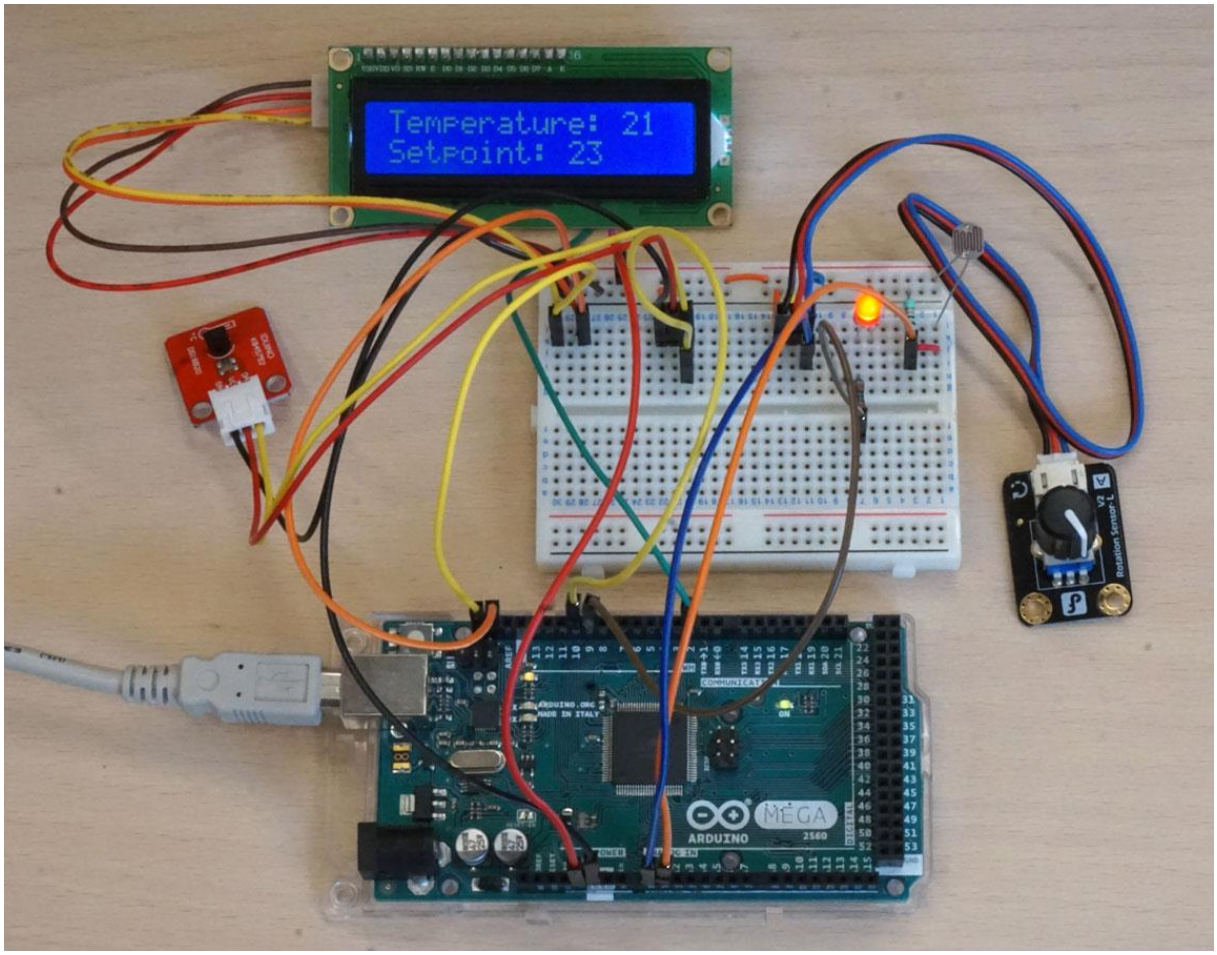
```
display_setpoint(setpoint);
if (temperature < setpoint)
  set_heater(ON);
 else
  set_heater(OFF);

luminosity = read_luminosity();
display_luminosity(luminosity);

delay(500);
#ifdef DEBUG
Serial.println("");
#endif
}
```

```
/**
 * @file therm_loop_util_v1.h
 * @brief  declaration of functions prototypes
 * and constants
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

void hardware_setup(void);
uint16_t read_temperature(void);
void display_temperature(uint16_t);
uint16_t read_setpoint(void);
void display_set_point(uint16_t);
uint16_t read_luminosity(void);
void display_luminosity(uint16_t);
void set_heater(uint8_t);
```

```
/**
 * @file therm_loop_util_v1.ino
 * @brief  simulation of thermostat
 * I/O access routines
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

void hardware_setup(void)
{
#ifdef DEBUG
Serial.println("Hardware setup");
Serial.println();
#endif
lcd.init(); // initialize the lcd
lcd.backlight();
lcd.setCursor(1,0);
lcd.print("Hardware setup");

sensors.begin();
pinMode(HEATER_LED, OUTPUT);
pinMode(BACKLIGHT, OUTPUT);
}
uint16_t read_temperature(void)
```

```
{
#ifdef DEBUG
Serial.println("Read temperature");
#endif
sensors.requestTemperatures();
return sensors.getTempCByIndex(0);
}

void display_temperature(uint16_t temperature)
{
#ifdef DEBUG
Serial.print("Temperature : ");
Serial.println(temperature);
#endif
lcd.setCursor(0,0);
lcd.print("Temperature:   "); //add spaces to erase previous value
lcd.setCursor(13,0);
lcd.print(temperature);
}

uint16_t read_setpoint(void)
{
#ifdef DEBUG
Serial.println("Read setpoint");
#endif
return (analogRead(POT)/32)+4; // between 4 and 36
}

void display_setpoint(uint16_t setpoint)
{
#ifdef DEBUG
Serial.print("Setpoint : ");
Serial.println(setpoint);
#endif
lcd.setCursor(0,1);
lcd.print("Setpoint:   "); //add spaces to erase previous value
lcd.setCursor(10,0);
lcd.print(setpoint);
}

uint16_t read_luminosity(void)
{
#ifdef DEBUG
Serial.println("Read luminosity");
#endif
return analogRead(LDR);
}

void display_luminosity(uint16_t luminosity)
{
#ifdef DEBUG
Serial.print("Luminosity : ");
Serial.println(luminosity);
#endif
analogWrite(BACKLIGHT,analogRead(LDR)/4); // 0-1023 range to 0-255 range
}



void set_heater(uint8_t state)
```

```
{
if (state==1)
  {
  #ifdef DEBUG
  Serial.println("Heater is ON");
    #endif
  digitalWrite(HEATER_LED, HIGH);
  }
 else
  {
  #ifdef DEBUG
  Serial.println("Heater is OFF");
  #endif
  digitalWrite(HEATER_LED, LOW);
  }
}
```

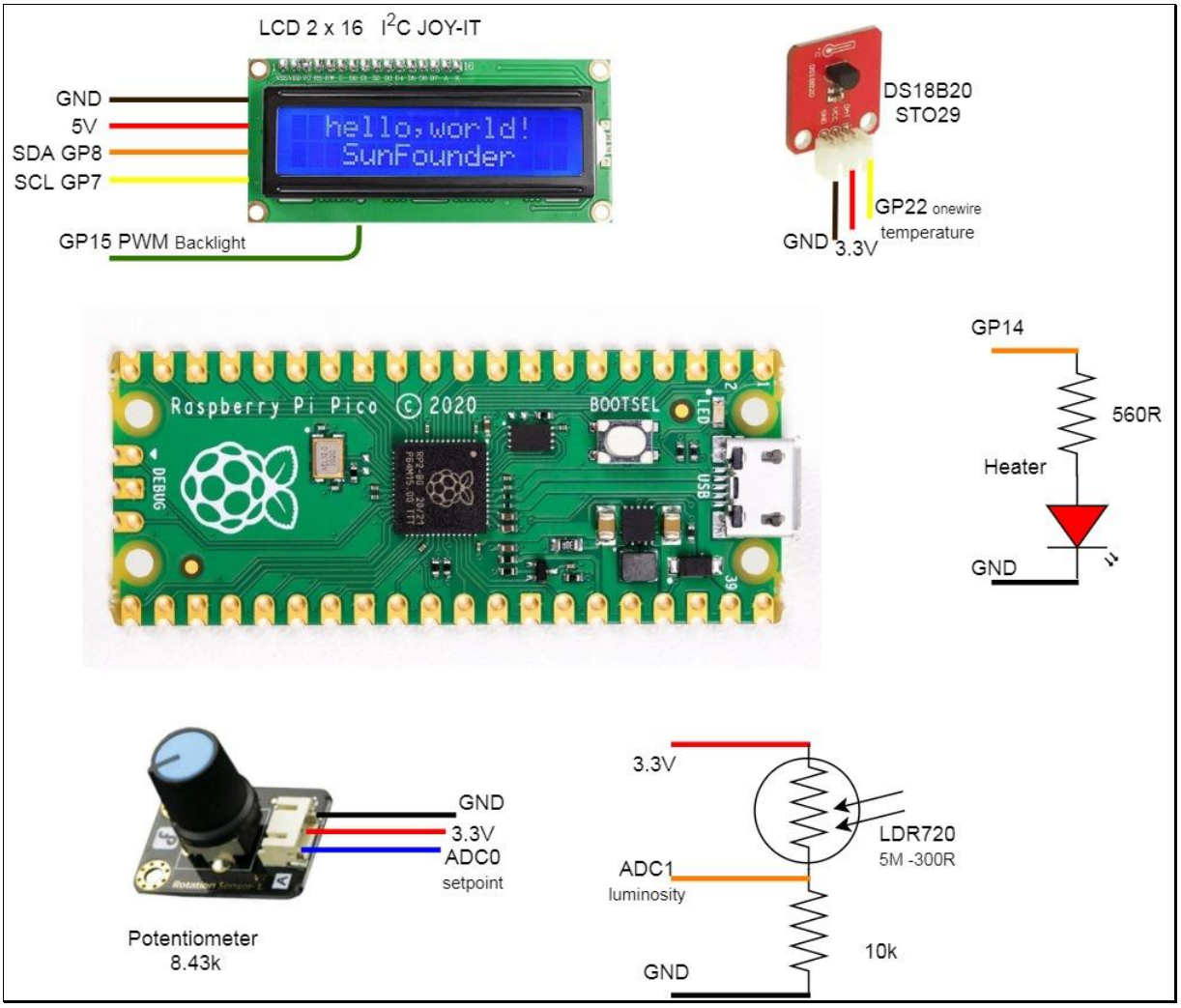


Arduino thermostat on breadboard.

Exercises for the Arduino thermostat.

Add MQTT services to the thermostat. Choose a Wifi shield and give a complete software architecture for this IoTfied thermostat.

Thermostat with sensors and display, Raspberry Pi pico version.



Program for the Raspberry pi Pico in Python.

The software is made of two parts:

- the main program `therm_loop_v1.py;`
- utility routines `therm_loop_util_v1.py.`

Starting with the simulated version, it is only necessary to replace stimuli with calls to appropriate I/O object or function. All I/O objects are created inside the `hardware_setup()` function and must be returned by this function to be available for the other functions.

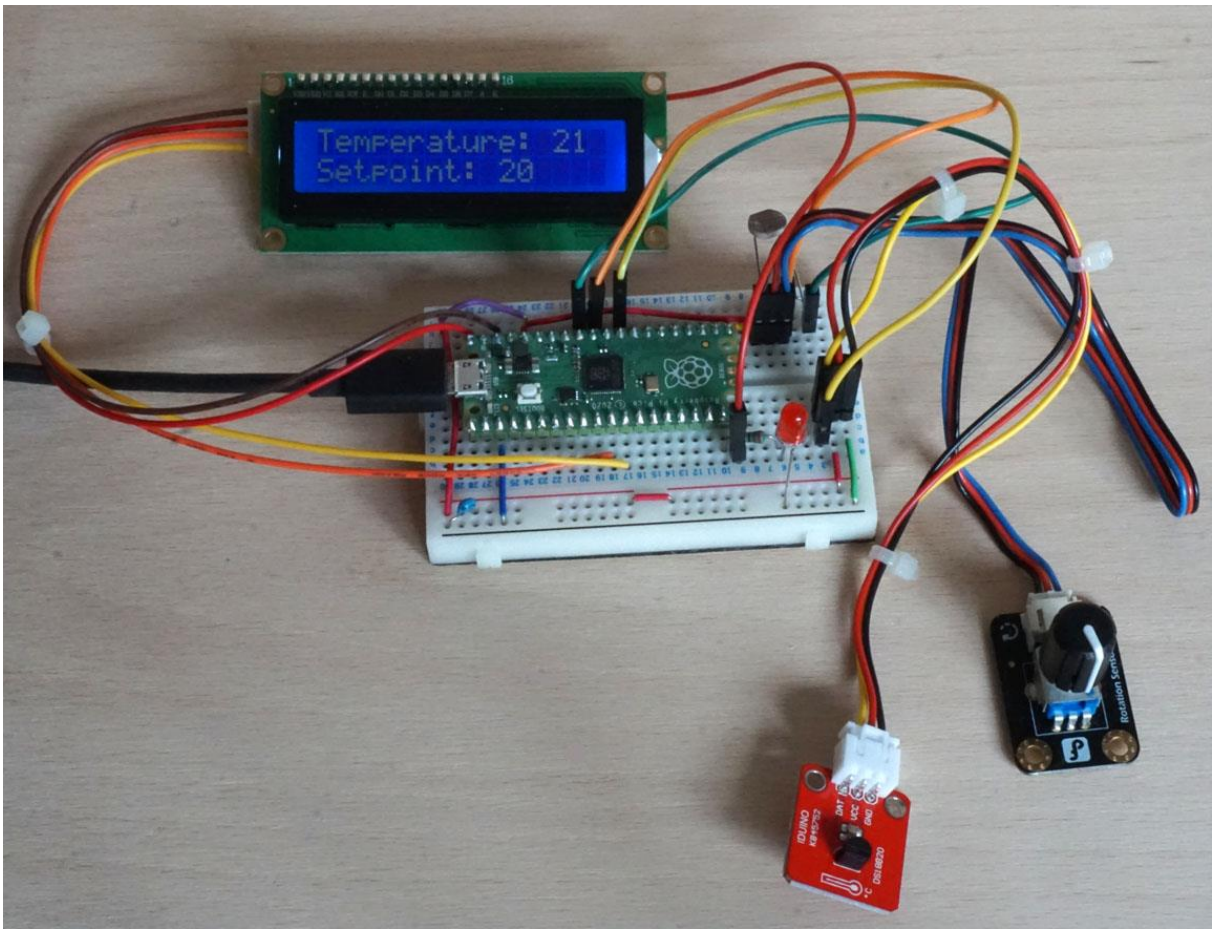
In Python, there is no counterpart to the C/C++ preprocessing instruction `#ifdef DEBUG ... #endif`. If you want to remove the console display, you have to comment out all your `print()` instructions (or use the logging module, but it is another story…).

```
# file therm_loop_v1.py
# simulation of thermostat
# author philippe.camus@hepl.be
# date 21/8/2021

import time
from therm_loop_util_v1 import *

ds_sensor, roms, pot, heater_led, ldr,lcd, bckl = hardware_setup()

print()

while(True):
    temperature = read_temperature(ds_sensor, roms)
    display_temperature(temperature, lcd)

    setpoint = read_setpoint(pot)
    display_setpoint(setpoint, lcd)

    if (temperature < setpoint):
        set_heater(1,heater_led)
    else:
        set_heater(0,heater_led)

    luminosity = read_luminosity(ldr)
    display_luminosity(luminosity, bckl)

    time.sleep(1)
    print()
```

```
# file therm_loop_util_v1.py
# simulation of thermostat
# I/O access routines
# author philippe.camus@hepl.be
# date 21/8/2021

import machine, onewire, ds18x20
from pico_i2c_lcd import I2cLcd
from machine import I2C, Pin, PWM
import utime as time

def hardware_setup():
    print("Hardware setup")
    ds_pin = machine.Pin(22)
    ds_sensor = ds18x20.DS18X20(onewire.OneWire(ds_pin))
    roms = ds_sensor.scan() # list of ds18x20 connected

    pot = machine.ADC(0)
    heater_led = machine.Pin(14, machine.Pin.OUT)
    ldr = machine.ADC(1)

    i2c = I2C(0, sda=Pin(8), scl=Pin(9), freq=400000)
    lcd = I2cLcd(i2c, 0x27, 2, 16)
    lcd.backlight_on()
```

```
    bckl = PWM(Pin(15))
    bckl.freq(1000)
    bckl.duty_u16(32768)

    lcd.move_to(1,0)
    lcd.putstr("Hardware setup")

    time.sleep(1.5)
    return ds_sensor, roms, pot, heater_led, ldr, lcd, bckl

def read_temperature(ds_sensor, roms):
    print("Read temperature")
    ds_sensor.convert_temp() # initiates temp conversion
    return int(ds_sensor.read_temp(roms[0]))

def display_temperature(temp,lcd):
    print("Temperature : ",temp)
    lcd.move_to(0,0)
    lcd.putstr("Temperature:   ")
    lcd.move_to(13,0)
    lcd.putstr(str(temp))

def read_setpoint(pot):
    print("Read setpoint")
    return int((pot.read_u16()/2047)+4)

def display_setpoint(setp,lcd):
    print("Setpoint : ",setp)
    lcd.move_to(0,1)
    lcd.putstr("Setpoint:   ")
    lcd.move_to(10,1)
    lcd.putstr(str(setp))

def read_luminosity(ldr):
    print("Read luminosity")
    return int((ldr.read_u16()/65535)*100)

def display_luminosity(lum,lcd):
    print("Luminosity : ",lum)
    bckl.duty_u16(lum*655)

def set_heater(state,heater_led):
    if (state==0):
        print("Heater is OFF")
        heater_led.value(0)
    else:
        print("Heater is ON")
        heater_led.value(1)

def check_lib():
    ds_sensor, roms,pot,heater_led, ldr,lcd , bckl = hardware_setup()
    temperature = read_temperature(ds_sensor, roms)
    display_temperature(temperature,lcd)
    setpoint = read_setpoint(pot)
    display_setpoint(setpoint,lcd)
    luminosity = read_luminosity(ldr)
    display_luminosity(luminosity,bckl)
    set_heater(0,heater_led)
    set_heater(1,heater_led)
```

```
if __name__ == "__main__":
    # execute only if run as the main module (i.e. not an import module)
    check_lib()
```



Raspberry Pi Pico thermostat on breadboard.

Exercises for the Raspberry Pi Pico thermostat.

a) Using an ESP8266 as a WiFi module, add MQTT services to the thermostat. Give a complete hardware and software architecture.

b) Using a BME280 sensor (temperature, humidity and pressure) and a dust sensor HM3301, design an environmental data logger. Data will be stored in Excel format on an SD card. Give a complete hardware and software architecture.

c) Using an ESP8266 as a WiFi module and a Unicorn PIM546 neopixel matrix, design a WiFi analyser. Each one of the 16 columns of the display will indicate the level of the corresponding WiFi channel. Give a complete hardware and software architecture. How will you address the power requirement of the display?

The MicroPython corner.

When working with a microcontroller as the Raspberry Pi Pico we will use an adapted version of Python: MicroPython (see *https://micropython.org*).

MicroPython is composed of three groups of libraries:

- The standard Python libraries with exceptions and limitations.
- MicroPython specific libraries.
- Port-specific libraries (we will use the RP2xx port).

One of the important MicroPython specific libraries is the **machine** library that contains functions related to the hardware.
(See : *https://docs.micropython.org/en/latest/library/machine.html*).

I/O Pin control: class Pin.

```
from machine import Pin

# create an output pin on pin #0
p0 = Pin(0, Pin.OUT)

# set the value low then high
p0.value(0)
p0.value(1)

# create an input pin on pin #2, with a pull up resistor
p2 = Pin(2, Pin.IN, Pin.PULL_UP)

# read and print the pin value
print(p2.value())

# reconfigure pin #0 in input mode with a pull down resistor
p0.init(p0.IN, p0.PULL_DOWN)

# configure an irq callback
p0.irq(lambda p:print(p))
```

See *https://docs.micropython.org/en/latest/library/machine.Pin.html* and samples of code from the web for explanations.

lambda is a Python keyword used to create a "noname" short function.

```
lambda p:print(p) # parameter = p, print(p) is the function body
```

Using an ADC: class ADC.

```
import machine

pot = machine.ADC(0) # ADC channel 0

val= pot.read_u16()
```

Question for the Raspberry Pi Pico thermostat MicroPython corner.

Discover how the `UART` *class works and test a communication with a terminal program (it needs a UART (3.3v) to USB cable). Describe in detail the* `irq` *method.*

Limitations of the simple loop paradigm.

The thermostat is a very simple system without critical issues and with plenty of time to do its tasks.

However it highlights the drawbacks of this method and give us clues to improve it.

The limitations are the following:

- Lack of responsiveness. A change in sensor value is only taken into account when the corresponding test function is executed. For instance, if we turn the potentiometer to change the setpoint, we have to wait until the `read_setpoint()` function is executed and then wait another time until the `set_heater()` function is executed.

- Lack of timing control. All the functions are executed with the same period.

- Lack of priorities. None of the function can override another.

- Not taking account of the system state. For instance, if no changes are made in the setpoint and the temperature is constant, you don't need to call the `set_heater()` function.

- The loop is not run continuously, a delay() is made after each iteration. If you remove the delay, the loop may run too fast, if you leave it, you have a lack of responsiveness. During this delay no code is executed (it could be an opportunity to put the system in sleep mode to decrease the power consumption).

- The readability of the code can be low if your system needs many conditions testing (spaghetti coding).

We will see below how to address partly those problems while keeping the loop architecture, but it won't be generally the best method.

Testing loop speed.

To test the thermostat loop speed, we use an output pin and set it to 1 during the loop execution.

Arduino loop speed.

```
#define PROBE 13

void setup()
{
//hardware initialization sensors, display, output controler...
hardware_setup();

pinMode(PROBE, OUTPUT);
digitalWrite(PROBE, LOW);
}


// run continuously
void loop()
{
uint16_t temperature;
uint16_t setpoint;
uint16_t luminosity;

digitalWrite(PROBE, HIGH);// 1 level = loop duration

temperature = read_temperature();
display_temperature(temperature);

setpoint = read_setpoint();
display_setpoint(setpoint);

if (temperature < setpoint)
  set_heater(ON);
 else
  set_heater(OFF);
```

```
luminosity = read_luminosity();
display_luminosity(luminosity);

digitalWrite(PROBE, LOW);

delay(500);
}
```

Measured loop duration (by scope) : 734 ms.

Raspberry Pi Pico loop speed.

```
probe_pin = machine.Pin(0,machine.Pin.OUT) # use GP0 to check loop timing
probe_pin.value(0)

print()

while(True):
    probe_pin.value(1) # // 1 level = loop duration

    temperature = read_temperature(ds_sensor, roms)
    display_temperature(temperature, lcd)

    setpoint = read_setpoint(pot)
    display_setpoint(setpoint, lcd)

    if (temperature < setpoint):
        set_heater(1,heater_led)
    else:
        set_heater(0,heater_led)

    luminosity = read_luminosity(ldr)
    display_luminosity(luminosity, lcd)

    probe_pin.value(0)

    time.sleep(1)
    print()
```

Measured loop duration (by scope) : 75 ms.

Despite the fact that Python is not compiled, the Python loop is almost 10 times faster.

One of the reasons for this situation is the fact that the Raspberry Pi Pico runs at 125 MHz and the Arduino Mega runs at 16 MHz. Another reason lies in the fact that the Cortex architecture of the Pico is more efficient than the ATmega architecture.

Bit-banging.

Bit-banging is the technique of emulating a communication bus (for instance the Onewire bus) using code that manipulates I / O instructions (setting and clearing I/O lines, testing I/O lines level and performing delay).

We use this approach when no hardware peripheral is available in the microprocessor or when we need more peripherals than available. A well-known example is the use of SoftwareSerial library in Arduino (https://www.arduino.cc/en/Reference/softwareSerial).

This approach leads to slow down a lot the CPU and even to block the program during the delays or when interrupts are disabled.

The following example shows a part of the OneWire library of Paul Soffregen (https://github.com/PaulStoffregen/OneWire/), used to write a bit on the bus.

```
void OneWire::write_bit(uint8_t v)
{
	IO_REG_TYPE mask IO_REG_MASK_ATTR = bitmask;
	volatile IO_REG_TYPE *reg IO_REG_BASE_ATTR = baseReg;

	if (v & 1) {
		noInterrupts();
		DIRECT_WRITE_LOW(reg, mask);
		DIRECT_MODE_OUTPUT(reg, mask);  // drive output low
		delayMicroseconds(10);
		DIRECT_WRITE_HIGH(reg, mask);   // drive output high
		interrupts();
		delayMicroseconds(55);
	} else {
		noInterrupts();
		DIRECT_WRITE_LOW(reg, mask);
		DIRECT_MODE_OUTPUT(reg, mask);  // drive output low
		delayMicroseconds(65);
		DIRECT_WRITE_HIGH(reg, mask);   // drive output high
		interrupts();
		delayMicroseconds(5);
	}
}
```

Notice the delays and interrupt desable blocking the processor.

Transforming hardware sensors or actuators into software objects is an important step to increase the reliability of an embedded system because it allows a better readability of the code, it improves the maintainability and it makes subsequent modifications easier.

OO on Arduino and C++.

We will transform a simple hardware device, a potentiometer, into its software counterpart. First we will use C++

```
/**
 * @file potoo.ino
 * @brief class for a potentiometer
 * @author philippe.camus@hepl.be
 * @date 11/8/2021
 */
class potentiometer
  {
  uint8_t pin; // center tap is connected to this analog pin
  uint32_t max_resistance;
  uint16_t max_angle;
  public :
  potentiometer(uint8_t p, uint32_t max_r, uint16_t max_a=300.0) //constructor
    {
    pin=p;
    max_resistance=max_r;
    max_angle=max_a;
   }

  uint32_t resistance()
    {
    return((float)analogRead(pin)/1024*max_resistance);
    }

  uint16_t angle()
    {
    return((float)analogRead(pin)/1024*max_angle);
    }

  uint8_t percent()
    {
    return((float)analogRead(pin)/1024*100);
    }
  };

potentiometer mypot(A0,8430); // Instanciate a 8430 ohms potentiometer
                              // connected to A0

// The setup function runs once when you press reset or power on the board
void setup()
{
Serial.begin(9600);
```

```
}


void loop()
// display resistance, angle and percentage value for a choosen potentiometer
{
Serial.print(mypot.resistance());Serial.print(" ohms  ");
Serial.print(mypot.angle());Serial.print("°  ");
Serial.print(mypot.percent());Serial.println("%");
delay(1000);
}
```

We will now integrate this class as well as a class for an LDR into our thermostat project.

As before, the software is made of three parts:

- the main program `therm_loop_v2.ino;`
- the header file for the utility routines `therm_loop_util_v2.h;`
- utility routines `therm_loop_util_v2.in.`

```
/**
 * @file therm_loop_v2.ino
 * @brief  thermostat
 * Uses therm_loop_util_v1.ino for functions code
 * @author philippe.camus@hepl.be
 * @date 19/8/2021
 */

// Necessary libraries
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define HEATER_LED 9
#define DS18B20_PIN 10
#define POT A0
#define LDR A1
#define BACKLIGHT 2
#define ON 1
#define OFF 0

//#define DEBUG // in normal mode comment out this line

#include "therm_loop_util_v2.h" // header for your functions and classes

// create I/O objects
// set the LCD address to 0x27 for a 16 chars and 2 line display
LiquidCrystal_I2C lcd(0x27,20,2);

OneWire oneWire(DS18B20_PIN); // set up oneWire to pin 10 (DS18B20 connection)

DallasTemperature sensors(&oneWire); // create a DS18B20 object
```

```
potentiometer setp_pot(POT,8430);

ldr disp_ldr(LDR);
// the setup function runs once when you press reset or power the board
void setup()
{
#ifdef DEBUG
// initialize console serial communication at 9600 bits per second:
Serial.begin(9600);
#endif

//hardware initialization sensors, display, output controler...
hardware_setup();
}


// run continuously
void loop()
{
uint16_t temperature;
uint16_t setpoint;
uint16_t luminosity;

temperature = read_temperature();
display_temperature(temperature);

setpoint = read_setpoint();
display_setpoint(setpoint);

if (temperature < setpoint)
  set_heater(ON);
 else
  set_heater(OFF);

luminosity = read_luminosity();
display_luminosity(luminosity);

delay(500);
#ifdef DEBUG
Serial.println("");
#endif
}
```

```
/**
 * @file therm_loop_util_v2.h
 * @brief  declaration of functions prototypes
 * and classes
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */

// Class for using potentiometer as a software object.
class potentiometer
  {
  uint8_t pin; // center tap is connected to this analog pin
  uint32_t max_resistance;
  uint16_t max_angle;
  public :
```

```
  potentiometer(uint8_t p, uint32_t max_r, uint16_t max_a=300.0) //constructor
    {
    pin=p;
    max_resistance=max_r;
    max_angle=max_a;
   }

  uint32_t resistance()
    {
    return((float)analogRead(pin)/1024*max_resistance);
    }

  uint16_t angle()
    {
    return((float)analogRead(pin)/1024*max_angle);
    }

  uint8_t percent()
    {
    return((float)analogRead(pin)/1024*100);
    }

     uint32_t range(uint32_t rstart, uint32_t rend)
    {
    return(((float)analogRead(pin)/1024*(rend-rstart))+rstart);
    }
  };

// Class for using ldr as a software object.
class ldr
  {
  uint8_t pin;
  public :
  ldr(uint8_t p) //constructor
    {
    pin=p;
    }

  uint32_t resistance()
    {
    return 5000000UL-(((float)analogRead(pin)/1024*5000000UL)+300);
    }

  uint8_t percent()
    {
    return ((float)analogRead(pin)/1024*100);
    }
  };

void hardware_setup(void);
uint16_t read_temperature(void);
void display_temperature(uint16_t);
uint16_t read_setpoint(void);
void display_set_point(uint16_t);
uint16_t read_luminosity(void);
void display_luminosity(uint16_t);
void set_heater(uint8_t);
```

```
/**
```

```
 * @file therm_loop_util_v2.ino
 * @brief  simulation of thermostat
 * I/O access routines
 * @author philippe.camus@hepl.be
 * @date 15/8/2021
 */



void hardware_setup(void)
{
#ifdef DEBUG
Serial.println("Hardware setup");
Serial.println();
#endif
lcd.init(); // initialize the lcd
lcd.backlight();
lcd.setCursor(1,0);
lcd.print("Hardware setup");

sensors.begin();
pinMode(HEATER_LED, OUTPUT);
pinMode(BACKLIGHT, OUTPUT);
}

uint16_t read_temperature(void)
{
#ifdef DEBUG
Serial.println("Read temperature");
#endif
sensors.requestTemperatures();
return sensors.getTempCByIndex(0);
}

void display_temperature(uint16_t temperature)
{
#ifdef DEBUG
Serial.print("Temperature : ");
Serial.println(temperature);
#endif
lcd.setCursor(0,0);
lcd.print("Temperature:   ");// add spaces to erase previous value
lcd.setCursor(13,0);
lcd.print(temperature);
}

uint16_t read_setpoint(void)
{
#ifdef DEBUG
Serial.println("Read setpoint");
#endif
return setp_pot.range(4,36); // between 4 and 36
}

void display_setpoint(uint16_t setpoint)
{
#ifdef DEBUG
Serial.print("Setpoint : ");
Serial.println(setpoint);
#endif
```

```
lcd.setCursor(0,1);
lcd.print("Setpoint:    "); //add spaces to erase previous value
lcd.setCursor(10,1);
lcd.print(setpoint);
}

uint16_t read_luminosity(void)
{
#ifdef DEBUG
Serial.println("Read luminosity");
#endif
return disp_ldr.percent();
}

void display_luminosity(uint16_t luminosity)
{
#ifdef DEBUG
Serial.print("Luminosity : ");
Serial.println(luminosity);
#endif

analogWrite(BACKLIGHT, luminosity*2.56);
}

void set_heater(uint8_t state)
{
if (state==1)
  {
  #ifdef DEBUG
  Serial.println("Heater is ON");
    #endif
  digitalWrite(HEATER_LED, HIGH);
  }
 else
  {
  #ifdef DEBUG
  Serial.println("Heater is OFF");
  #endif
  digitalWrite(HEATER_LED, LOW);
  }
}
```

OO on Raspberry Pi Pico and MicroPython.

Let's see how we can implement classes in Python. We choose the potentiometer as we did in the previous example.

```
# file potoo.py
# Class for a potentiometer
# author philippe.camus@hepl.be
# date 22/8/2021

import time
from machine import ADC

class potentiometer:
    def __init__(self, id, max_r, max_a=300):
         self.ad_chan = ADC(id)
         self.max_resistance = max_r
         self.max_angle = max_a

    def resistance(self):
        return self.ad_chan.read_u16()/65535*self.max_resistance

    def angle(self):
        return self.ad_chan.read_u16()/65535*self.max_angle

    def percent(self):
        return self.ad_chan.read_u16()/65535*100


mypot = potentiometer(0,8430)

while(True):
    print(int(mypot.resistance()))
    print(int(mypot.angle()))
    print(int(mypot.percent()))
    print()
    time.sleep(1)
```

Exercices (in Python).

a) Design and test a class for the LDR.

b) Add potentiometer and LDR classes in the `therm_loop_v1.py` *program.*

What does the Object-Oriented paradigm bring to us?

The main interest of object orientation for embedded systems is to improve code readability and to simplify software design. But this paradigm doesn't improve responsiveness and other problems highlighted in the loop architecture.

LOOP WITH TIMING CONTROL.

In our thermostat, not all the functions should be performed at the same pace.

Each action has its own requirement:

- Reading temperature can be done each minute because temperature doesn't change fast.
- Turning the potentiometer to choose the setpoint must result in a fast change on the display, so it needs a faster sampling rate, for instance of 500 ms or less.
- Getting luminosity can be done within one or two seconds.
- Switching the heater should not occur too quickly to avoid untimely switching while setpoint is modified.

The solution is to control, for each function, the most appropriate moment to call it.

For this purpose we will use a timer for scheduling each function.

Arduino and C/C++.

As before, the software is made of three parts:

- the main program `therm_loop_v3.ino;`
- the header file for the utility routines `therm_loop_util_v3.h;`
- utility routines `therm_loop_util_v3.ino.`

The last two files are the same as their counterparts in version v2 and will not be shown here.

We use the `millis()` function to keep track of time. Each function will be controlled by a test between its last calling time, the present moment and its recommended execution rate.

```
/**
 * @file therm_loop_v3.ino
 * @brief  thermostat
 * Uses therm_loop_util_v1.ino for functions code
 * @author philippe.camus@hepl.be
 * @date 19/8/2021
 */

// Necessary libraries
```

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define HEATER_LED 9
#define DS18B20_PIN 10
#define POT A0
#define LDR A1
#define BACKLIGHT 2
#define ON 1
#define OFF 0

//#define DEBUG // in normal mode comment out this line

#include "therm_loop_util_v3.h" // header for your functions and classes

// create I/O objects
LiquidCrystal_I2C lcd(0x27,20,2);  // set the LCD address to 0x27 for a 16 chars
and 2 line display
OneWire oneWire(DS18B20_PIN); // set up oneWire to pin 10 (DS18B20 connection)
DallasTemperature sensors(&oneWire); // create a DS18B20 object

potentiometer setp_pot(POT,8430);
ldr disp_ldr(LDR);

// execution rate and last execution time
#define EXR_TEMP 60000
uint32_t lastex_temp = millis();

#define EXR_SETP 250
uint32_t lastex_setp = millis();

#define EXR_HEAT 2000
uint32_t lastex_heat = millis();

#define EXR_LUM 1000
uint32_t lastex_lum = millis();

// the setup function runs once when you press reset or power the board
void setup()
{
#ifdef DEBUG
// initialize console serial communication at 9600 bits per second:
Serial.begin(9600);
#endif

//hardware initialization sensors, display, output controler...
hardware_setup();
}


// run continuously
void loop()
{
uint16_t temperature;
uint16_t setpoint;
uint16_t luminosity;

if ((unsigned long)(millis() - lastex_temp) >= EXR_TEMP)
```

```
  {
  lastex_temp = millis();
  temperature = read_temperature();
  display_temperature(temperature);
  }

if ((unsigned long)(millis() - lastex_setp) >= EXR_SETP)
  {
  lastex_setp = millis();
  setpoint = read_setpoint();
  display_setpoint(setpoint);
  }

if ((unsigned long)(millis() - lastex_heat) >= EXR_HEAT)
  {
  lastex_heat = millis();
  if (temperature < setpoint)
    set_heater(ON);
   else
    set_heater(OFF);
  }

if ((unsigned long)(millis() - lastex_lum) >= EXR_LUM)
  {
  lastex_lum = millis();
  luminosity = read_luminosity();
  display_luminosity(luminosity);
  }

#ifdef DEBUG
Serial.println("");
#endif
}
```

Raspberry Pi Pico and MicroPython.

The software is made of two parts:

- the main program `therm_loop_v3.py`;
- utility routines `therm_loop_util_v3.py` (same as v1).

```
# file therm_loop_v3.py
# simulation of thermostat
# author philippe.camus@hepl.be
# date 21/8/2021

import time
from machine import Timer
from therm_loop_util_v1 import *

ds_sensor, roms, pot, heater_led, ldr, lcd, bckl = hardware_setup()

print()
```

```
timer1 = Timer()
timer2 = Timer()
timer3 = Timer()
timer4 = Timer()

# group read or check and display into callback functions
# a timer object parameter is mandatory (but not used here)
def temp_read_display(timer):
    global temperature # must be global to be used by heater_test
    temperature = read_temperature(ds_sensor, roms)
    display_temperature(temperature, lcd)

def setp_read_display(timer):
    global setpoint # must be global to be used by heater_test
    setpoint = read_setpoint(pot)
    display_setpoint(setpoint, lcd)

def heater_test(timer):
    if (temperature < setpoint):
        set_heater(1,heater_led)
    else:
        set_heater(0,heater_led)

def lum_read_display(timer):
    luminosity = read_luminosity(ldr)
    display_luminosity(luminosity, lcd)

# read and display data once before installing timer callbacks
temperature = read_temperature(ds_sensor, roms)
display_temperature(temperature, lcd)

setpoint = read_setpoint(pot)
display_setpoint(setpoint, lcd)

if (temperature < setpoint):
    set_heater(1,heater_led)
else:
    set_heater(0,heater_led)

luminosity = read_luminosity(ldr)
display_luminosity(luminosity, bckl)

# install callbacks. These functions will be called by the timer interrupt code
# each function is called with the more appropriate rate
timer1.init(freq=0.5, mode=Timer.PERIODIC, callback=temp_read_display)
timer2.init(freq=2, mode=Timer.PERIODIC, callback=setp_read_display)
timer3.init(freq=1, mode=Timer.PERIODIC, callback=heater_test)
timer4.init(freq=0.25, mode=Timer.PERIODIC, callback=lum_read_display)

while(True):
    # when no callback is executing you can do useful stuff here.
    # for instance log in a file the operating time of the heater
    # or put the system in sleep mode to save energy
```

Results (for the MicroPython code).

In the following shell window, you can see the different occurrence of each function.

Shell

```
>>> %Run -c $EDITOR_CONTENT
 Hardware setup

 Read temperature
 Temperature :  23
 Read setpoint
 Setpoint :  19
 Heater is OFF
 Read luminosity
 Luminosity :  20

>>> Read setpoint
 Setpoint :  19
 Read setpoint
 Setpoint :  19
 Heater is OFF
 Read setpoint
 Setpoint :  19
 Read temperature
 Temperature :  23
 Read setpoint
 Setpoint :  19
 Heater is OFF
 Read setpoint
 Setpoint :  19
 Read setpoint
 Setpoint :  19
 Heater is OFF
 Read setpoint
 Setpoint :  19
 Read temperature
 Temperature :  23
 Read setpoint
 Setpoint :  19
 Heater is OFF
 Read luminosity
 Luminosity :  20
```

Using text files to log data.

You get a file object in using the `open( )` build-in function. This function needs two parameters: the name of the file we want to work with and the mode of connection to the file : "`r`" for read, "`w`" for write (and overwrite if the file still exists), "`a`" to add to an existing file (or create a new one if it doesn't exist). Default mode for a file is a text file, i.e. line of characters separated by crlf (`\r\n`). Other modes exist.

The `read()` method returns a line (formatted as a string) from the file and skips to the next line. If we have reached the end of the file, `read()` returns an empty string.

The `write()` method writes a line to the file and goes to next position.

The `close()` method ends the access to the file and saves buffered data to it.

To add a time stamp we use the time method of the `time()` module. It returns the number of seconds since a starting date and time, usually 1/1/1970 00:00:00

The `localtime()` method return an 8tuple of numbers : year, month, day, hour, minute, second, day of week (0 = Monday), number of days since the beginning of the year.

The following program adds a time stamp and the temperature in a file, each time it is called.

```
# file log_temp.py
# log temperature from RPI pico sensor
# with date stamp in a text file
# author philippe.camus@hepl.be
# date 25/8/2021

import time
from machine import ADC

now=time.localtime(time.time())
#   yyyy mm dd  hh  mm  ss  wd  wy
# wd : weekday (0..6) wy number of day in the year

# now date
str_dated_temp=str(now[2])+"/"+str(now[1])+"/"+str(now[0])+" " \
               +str(now[3])+":"+str(now[4])+":"+str(now[5])
```

```
# now temperature
temp_sensor = ADC(4) # from RPi Pico sensor with conversion formula
temperature = temp_sensor.read_u16()*3.3 / 65535
celsius_degrees = 27 - (temperature - 0.706) / 0.001721

# date + temperature + end of line
str_dated_temp=str_dated_temp + "  "+str(round(celsius_degrees,1)) +"\r\n"

# write to file
f = open("temp_log.txt", "a")
f.write(str_dated_temp)
f.close()

#read file
f = open("temp_log.txt", "r")
cur_line=f.readline()
while (cur_line !=""): # end of file means an empty line
    print(cur_line, end="")
    cur_line=f.readline()

f.close()
```

Exercices.

a) Design a data logger storing data stamp and temperature each minute.

b) On the Raspberry Pi Pico, the date is initialized when you connect the board to your computer. There is no battery backup, so if you use the board as a standalone system the time information is lost when you disconnect it. Devise a system with an external battery to power the board and to maintain the clock running.

Interrupts.

To get a faster response to external events, one of the method is to use interrupts.

The Wikipedia definition of an interrupt is:

In digital computers, an interrupt is a response by the processor to an event that needs attention from the software. An interrupt condition alerts the processor and serves as a request for the processor to interrupt the currently executing code when permitted, so that the event can be processed in a timely manner. If the request is accepted, the processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, unless the interrupt indicates a fatal error, the processor resumes normal activities after the interrupt handler finishes. (*https://en.wikipedia.org/wiki/Interrupt*)

We are not going into details here, this subject will be discussed more deeply in other courses.

One simple way to deal with interrupt is to use callback functions.

Example for Arduino:

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

Allows attaching a callback function (`ISR`) to a GPIO input pin.
See : *https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/*

Example for Micropython:

In the machine module, you will find several methods to attach interrupts to pins, timer, uart…

- `pin.irq(handler= , trigger= )`
- `timer.init(freq= , mode=Timer.PERIODIC, callback= )`
- `uart.irq(trigger=UART.RX_ANY, handler= )`

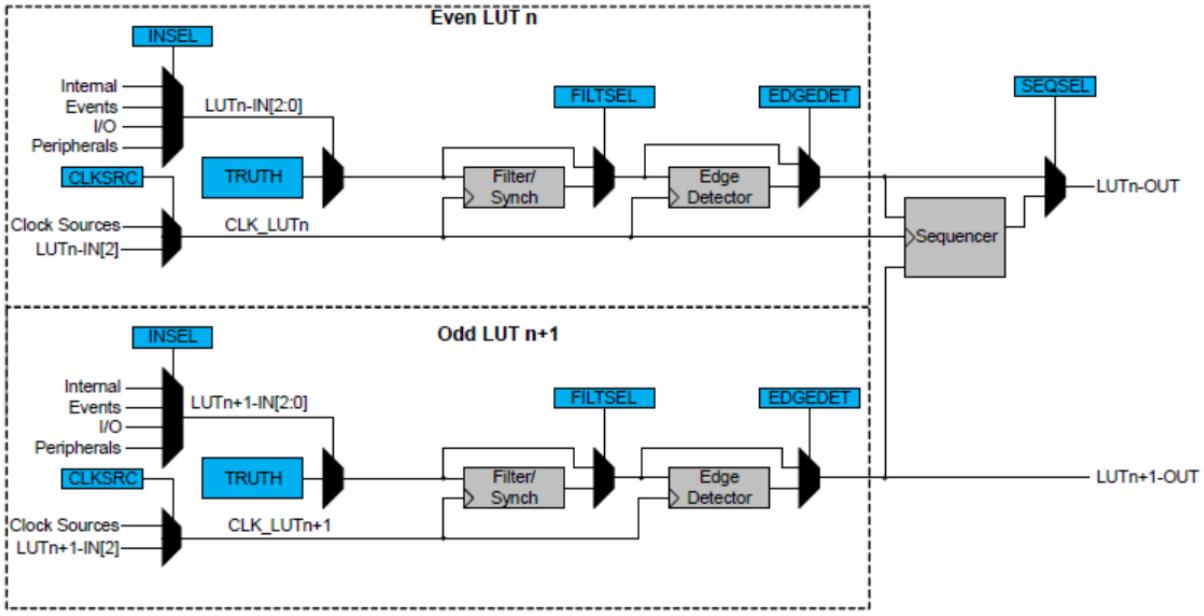
See : *https://docs.micropython.org/en/latest/library/machine.html*

FPGA block: the ATmega4809 CCL.

When we need high speed in our applications, for instance in video, the solution is to design a specific circuit in an FPGA.

Sometimes we need computing power too, and we can integrate a CPU core in our FPGA.

Another possibility is to use a microcontroller with embedded FPGA blocks. This type of architecture is now available on some microcontrollers, for instance on the ATmega 4809. These are simple blocks here, named Configurable Custom Logic (CCL)



The Configurable Custom Logic (CCL) is a programmable logic peripheral which can be connected to the device pins, to events, or to other internal peripherals. The CCL can serve as 'glue logic' between the device peripherals and external devices. The CCL can eliminate the need for external logic components, and can also help the designer to overcome real-time constraints by combining Core Independent Peripherals (CIPs) to handle the most time-critical parts of the application independent of the CPU.

(Extract from ATmega 4809 dataSheet).

In this kind of architecture, the high-speed tasks are assigned to the hardware blocks instead of being controlled by software.

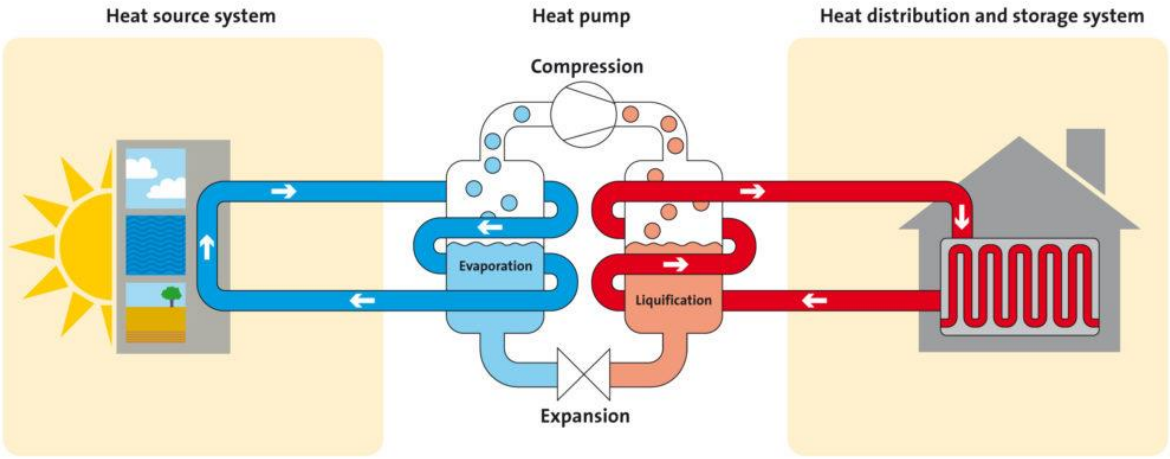
DMA AND I/O DEDICATED PROCESSORS.

So far, we have worked with a very simple example: the thermostat. It is time now to address more complex problems. Say we need to design a coefficient of performance (COP) logger for a heat pump.

The COP is the ratio of the useful heat produced to the electrical energy consumed.

First, let us define the requirements of our system.

The heat pump and heating system are the following:



Picture from https://www.rhc-platform.org/

The heat source can be water, ground or air. The heating is achieved through hot water circulating in the floor or in radiators.

We need the following sensors:

- Voltage (V_{in}) and current (I_{in}) sensors connected to the mains to compute electrical energy.
- Temperature sensors at both end of the heating circuit (measuring T_{in} and T_{out}) and a flowmeter (measuring F_m – mass flow) to compute the produced heat.

Our device will also have a display (touch screen) and a Bluetooth transceiver to communicate with a smartphone.

The COP will be computed every minute which implies to integrate the data (Vin, Iin, Tin, Tout and Fm) from the sensor during this period. Let's say that we need to sample V_{in} and I_{in} every 100µs and T_{in}, T_{out} and F_m every 0.5s.

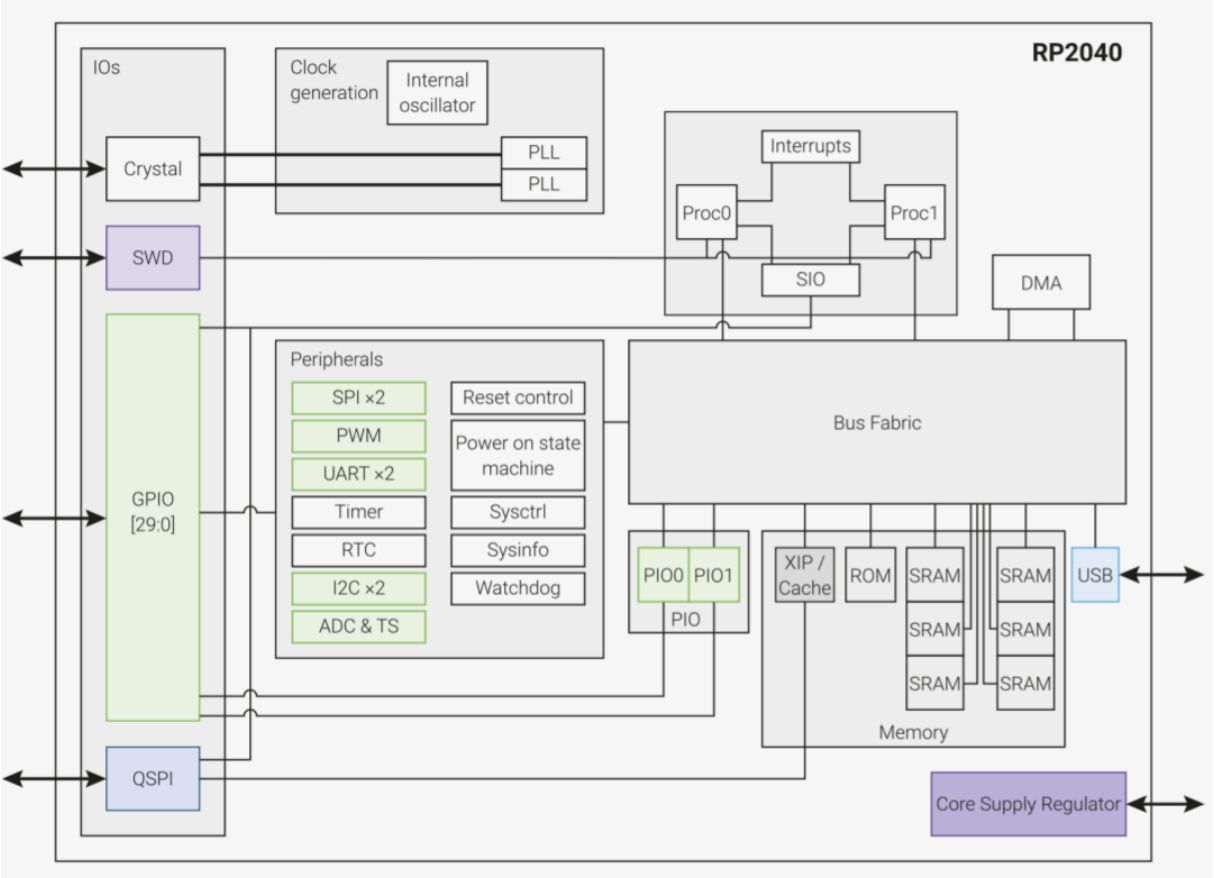
In addition to these tasks, we have to display data on the touchscreen, get inputs from it and manage the Bluetooth connection.

It won't be easy, maybe even impossible, to get information on time from all these sensors if we use the simple loop architecture.

DMA.

DMA stand for Direct Memory Access. It means getting data from the sensors without using the CPU and therefore without executing code.

A special microcontroller architecture is needed to implement this function. It uses an interconnection matrix called a bus fabric. We will find this kind of system in the RP2040 used in the Raspberry Pi Pico.



RP2040 block diagram from the datasheet.

Once configured, the DMA block is able to transfer data from a peripheral to a block of RAM memory. For the COP logger, it means that the program can receive all the sensor data without any intervention of its own and that the data will be present in realtime. Other tasks of the program can be done in the loop without any problems.

I/O dedicated processors.

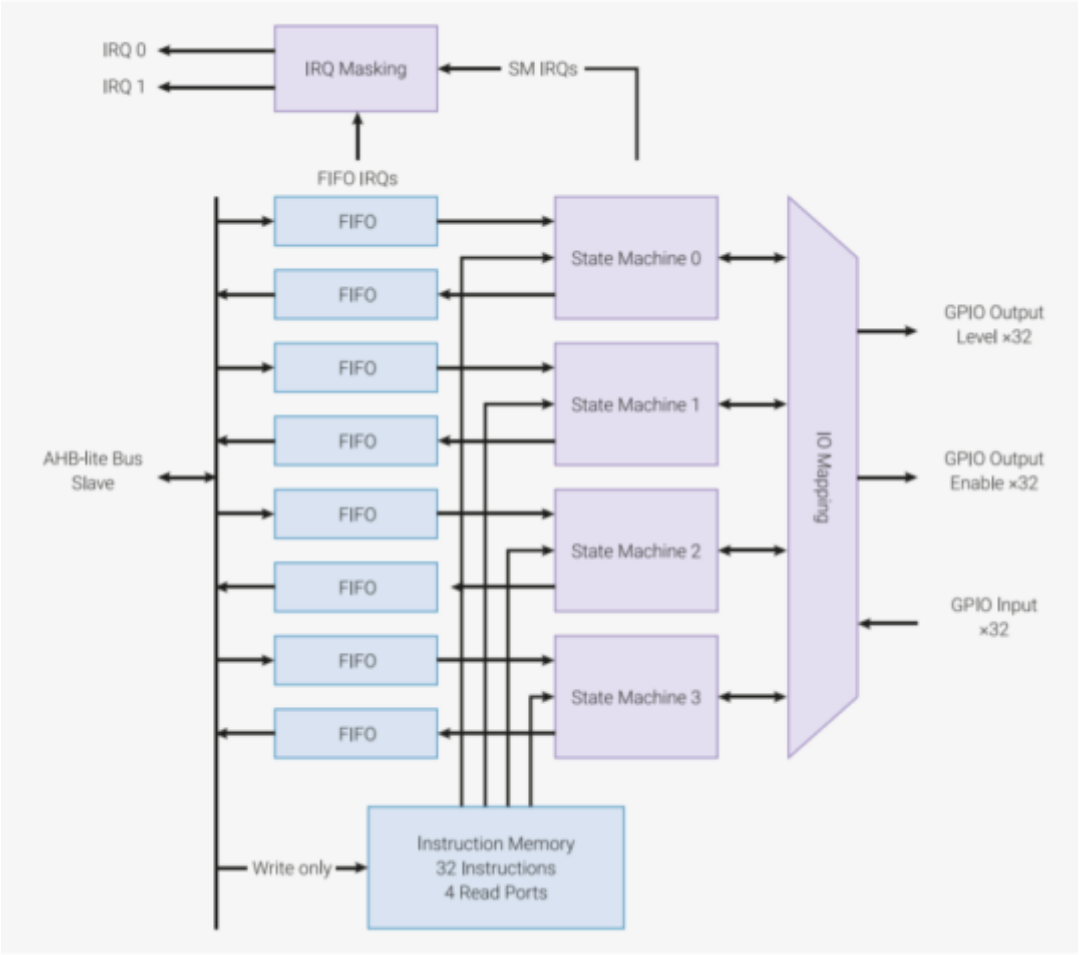
I/O dedicated processors are specialized processors especially designed to control I/O communication and connected through FIFO to the microcontroller.

We will illustrate this with the RP2040 PIO (Programmable Input Output block).

The RP2040 contains 8 I/O dedicated processors which are actually 8 small 16-bit RISC processors, with a set of 9 very powerful instructions.

The 8 PIOs are connected to the system bus (CPU and DMA access) each via two FIFOs of 4 words of 32 bits (one on input and one on output).

They can directly interact with the GPIO lines.



Each instruction is executed in exactly one machine cycle (the value of which can be chosen by a predivisor and can reach the speed of the CPU clock). PIOs were designed to perform bit-banging in an efficient way and give us the

opportunity to build custom peripherals almost as effective as their counterparts (if they exist).

The following example shows the use of PIO to control the Neopixel bus.

```
# file neo_bar_graph.py
# bargarph on Neopixel Adafruit 1426
# author philippe.camus@hepl.be
# date 21/8/2021
# Adaptated from :
#http://www.pibits.net/code/raspberry-pi-pico-and-neopixel-example-in-micropython.php#codesyntax_1
import array, time
from machine import Pin
import rp2

# Configure the number of WS2812 LEDs, pins and brightness.
NUM_LEDS = 8
PIN_NUM = 2
brightness = 0.05

#Neopixel protocol is controlled by a pio machine
@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT,
autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]
    wrap()


# Create the StateMachine with the ws2812 program, outputting on Pin(PIN_NUM).
sm = rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(PIN_NUM))

# Start the StateMachine, it will wait for data on its FIFO.
sm.active(1)

# Display a pattern on the LEDs via an array of LED RGB values.
ar = array.array("I", [0 for _ in range(NUM_LEDS)])

def pixels_show():
    dimmer_ar = array.array("I", [0 for _ in range(NUM_LEDS)])
    for i,c in enumerate(ar):
        r = int(((c >> 8) & 0xFF) * brightness)
        g = int(((c >> 16) & 0xFF) * brightness)
        b = int((c & 0xFF) * brightness)
        dimmer_ar[i] = (g<<16) + (r<<8) + b
    sm.put(dimmer_ar, 8) # to the PIO FIFO

def pixels_set(i, color):
    ar[i] = (color[1]<<16) + (color[0]<<8) + color[2]
```

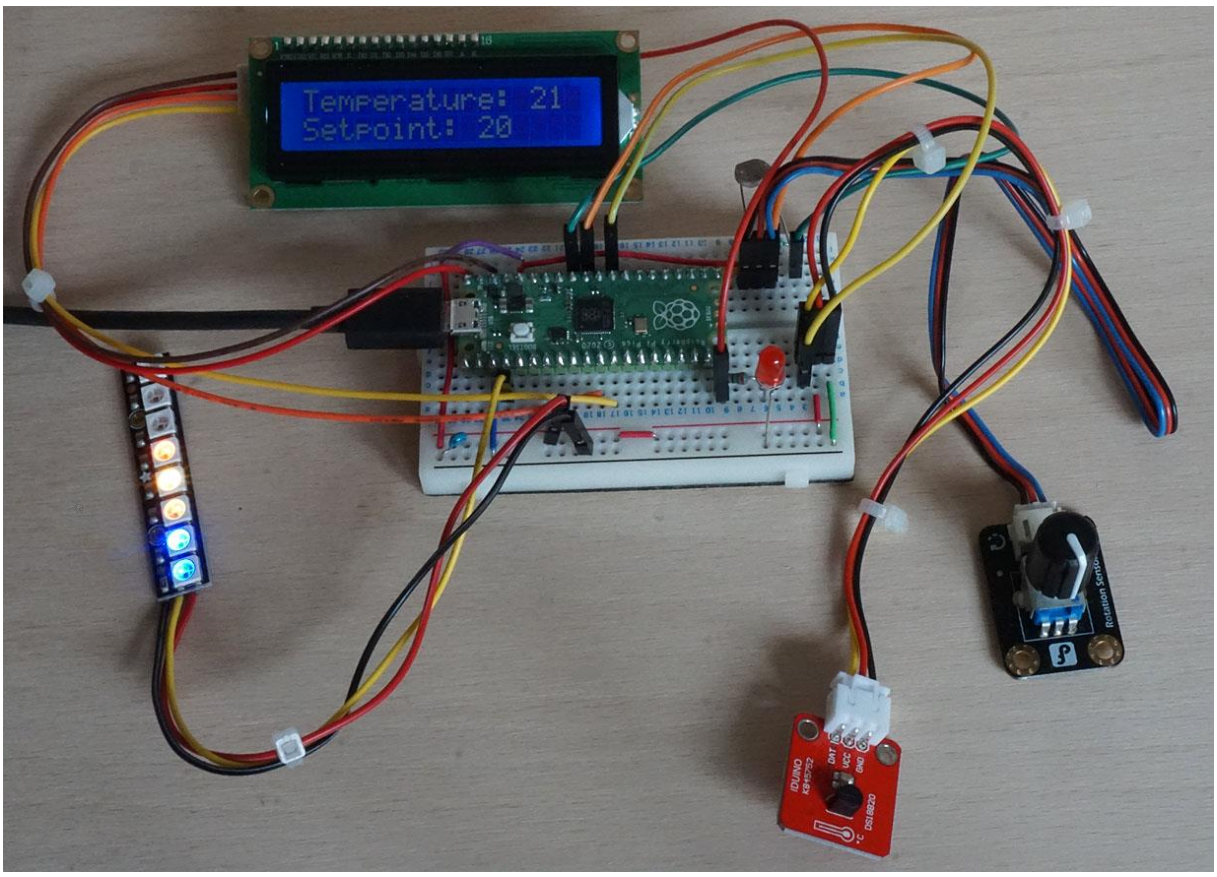
```
BLACK = (0, 0, 0)
RED1 = (63, 0, 0)
RED2 = (255, 0, 0)
ORANGE1 = (127, 31,0)
ORANGE2 = (255, 65,0)
YELLOW1 = (63, 37, 0)
YELLOW2 = (255, 150, 0)
BLUE1 = (0, 0, 63)
BLUE2 = (0, 0, 255)

# 8 kinds of bagraph
BG0 = (BLUE1, BLACK, BLACK ,BLACK ,BLACK ,BLACK ,BLACK ,BLACK)
BG1 = (BLUE1, BLUE2, BLACK ,BLACK ,BLACK ,BLACK ,BLACK ,BLACK)
BG2 = (BLUE1, BLUE2, YELLOW1 ,BLACK ,BLACK ,BLACK ,BLACK ,BLACK)
BG3 = (BLUE1, BLUE2, YELLOW1 ,YELLOW2 ,BLACK ,BLACK ,BLACK ,BLACK)
BG4 = (BLUE1, BLUE2, YELLOW1 ,YELLOW2 ,ORANGE1 ,BLACK ,BLACK ,BLACK)
BG5 = (BLUE1, BLUE2, YELLOW1 ,YELLOW2 ,ORANGE1 ,ORANGE2 ,BLACK ,BLACK)
BG6 = (BLUE1, BLUE2, YELLOW1 ,YELLOW2 ,ORANGE1 ,ORANGE2 ,RED1 ,BLACK)
BG7 = (BLUE1, BLUE2, YELLOW1 ,YELLOW2 ,ORANGE1 ,ORANGE2 ,RED1 ,RED2)

BG_LIST= [BG0, BG1, BG2, BG3, BG4, BG5, BG6, BG7]

# used to test library
def draw_bar_graph(level):
    BG_DISP= BG_LIST[int(level/4)]

    for i, color in enumerate(BG_DISP):
        pixels_set(i, color)

    pixels_show()


if __name__ == "__main__":
    # execute only if run as the main module (i.e. not an import module)
    for i in range(8):
        draw_bar_graph((i*4)+3)
        time.sleep(1)
```

We can import this module in our thermostat and add a call to `draw_bar_graph(level)` when needed.

If we measure the loop duration (by scope), the value is 77 ms.

That's 2 ms more than without the bargraph. If we analyse in detail the code, we can see that a great part of this time is spent in formating the value to display.

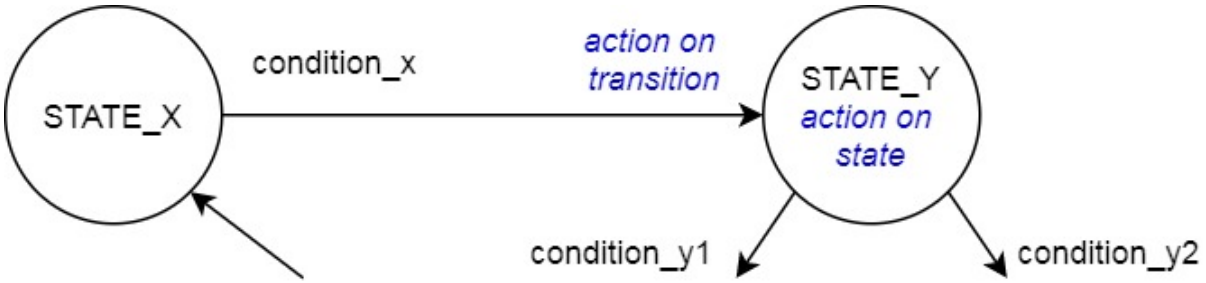
The PIO language is powerful, but, as it is assembly, a bit cryptic.



Raspberry Pi Pico thermostat on breadboard with neopixel bargraph.

FINITE-STATE MACHINE (F.S.M.).

A state machine, aka a "finite state machine" or F.S.M., is a kind of digital machine that can be described by a finite number of states. Each state describes the peculiar behaviour of the machine. Passing from state to state depends on conditions. Actions can occur either when the transition is done, or when a state is entered (actions bound to a state).



To program this kind of machine, for instance in C, you need a state variable aimed to contain the name of the current state and an infinite loop with a switch… case… instruction to select the code to execute in the current state.

The code of each state will check for conditions, performs actions and will change the state of the system (by writing in the state variable) if necessary.

```
enum state_names {STATE_X, STATE_Y} current_state;

while (1)
  {
  switch(current_state)
   {
    case STATE_X:
      if (condition_x)
        {
        action_on_transition();
        current_state = STATE_Y;
        }
      break;

     case STATE_Y:
       action_on_state();
       if (condition_y1)
         {
         // go to another state and possibly do some action;
         }

       if (condition_y2)
```

```
        {
        // go to another state and possibly do some action;
        }
      break;

    // continuing for other states
}
```

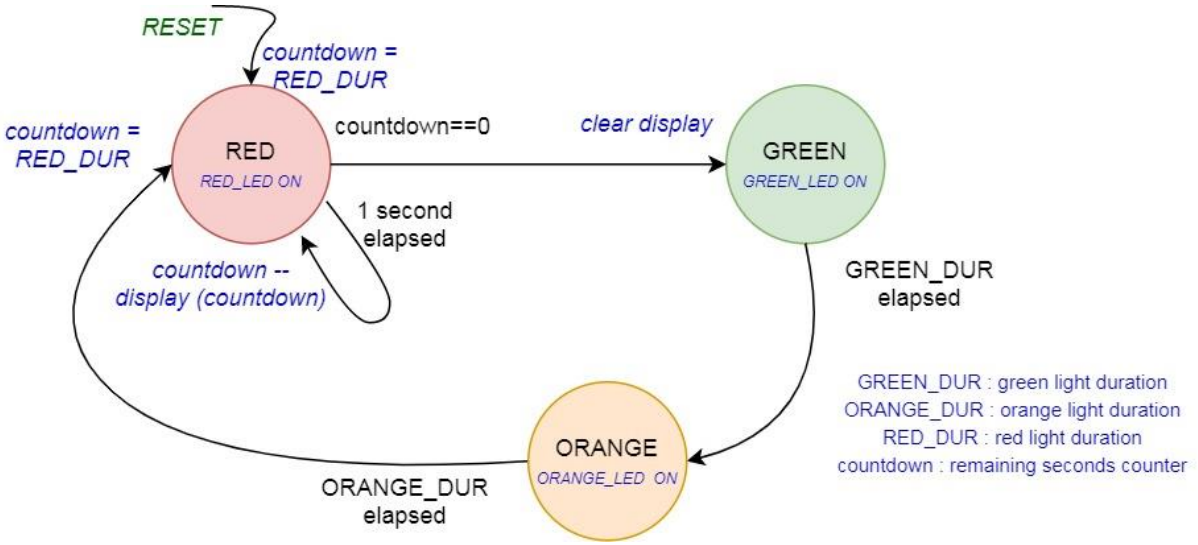
Example.

Let us design alternating traffic lights for a road repair.

We will use an Arduino UNO with a Seeed Studio Starter Shield. It should be noted that this shield doesn't work with Arduino Mega board due to I2C pin position.

3 LEDs will simulate the traffic light and a seven segments display the countdown counter.

State diagram:



Note that there are three actions bound to the states (`RED_LED ON`, `GREEN_LED ON`, `ORANGE_LED ON`) and two actions bound to the transitions (`countdown = RED_DUR`, `countdown-- display(countdown)`)

Program :

```
/**
 * @file alt_traffic4.ino
 * @brief FSM to control alternate traffic lights
 * use millis function to control duration of ligths
 * implemented on UNO board with Seeed Studio Starter Shield
 * @author philippe.camus@hepl.be
 * @date 5/8/2021
 */

#include <TTSDisplay.h> // library for 7 segments display using TM1636 IC

// LEDS on Starter Shield
#define RED_LED 5
#define ORANGE_LED 4
#define GREEN_LED 3

// duration for each light
#define SECOND_TICK 1000UL // 1 second tick = 1000 ms

// duration in seconds
#define RED_DUR 11
#define ORANGE_DUR 1
#define GREEN_DUR 10


enum LIGHT_STATES {RED, ORANGE, GREEN};// State names
enum LIGHT_STATES light_state; //State variable

uint32_t previous_millis; // to remember previous end of duration time
uint16_t countdown; // second countdown for red waiting period

TTSDisplay disp; // instanciate display object


void clear_all_leds(void)
{
digitalWrite(RED_LED,0);
digitalWrite(ORANGE_LED,0);
digitalWrite(GREEN_LED,0);
}


void setup()
{
pinMode(RED_LED,OUTPUT);
pinMode(ORANGE_LED,OUTPUT);
pinMode(GREEN_LED,OUTPUT);
clear_all_leds();
light_state = RED;
digitalWrite(RED_LED,1);
previous_millis = millis();
countdown = RED_DUR;
disp.num(countdown);
}
```

```
void loop()
{
switch (light_state) //state machine
  {
  case RED:
   clear_all_leds();
   digitalWrite(RED_LED,1);
   if ((unsigned long)(millis() - previous_millis) >= SECOND_TICK)
     {
     countdown--;
     disp.num(countdown);
     previous_millis = millis();
     if (countdown == 0)
       {
       light_state = GREEN;
       disp.clear();
       }
     }
   break;
  case ORANGE:
   clear_all_leds();
   digitalWrite(ORANGE_LED,1);
   if ((unsigned long)(millis() - previous_millis) >= (ORANGE_DUR*SECOND_TICK))
     {
     light_state = RED;
     countdown = RED_DUR;
     disp.num(countdown);
     previous_millis = millis();
     }
   break;
  case GREEN:
   clear_all_leds();
   digitalWrite(GREEN_LED,1);
   if ((unsigned long)(millis() - previous_millis) >= (GREEN_DUR*SECOND_TICK))
     {
     light_state = ORANGE;
     previous_millis = millis();
     }
   break;
  }
// Note : the statement
// if ((unsigned long)(millis() - previousMillis) >= RED_DUR)
// is correct even in case of millis rollover after 49 days
// see https://www.baldengineer.com/arduino-how-do-you-reset-millis.html
}
```

The code is rather straightforward and self-explanatory. The three states are highlighted in yellow. `TTSDisplay` class is used to access the LED display. The time duration of the different phases of the light can be easily modified to match a real system.

Exercice (in Python).

Design and test the traffic lights system in MicroPython on the RPi Pico using a neo pixel bargraph for the lights and the LCD for the countdown.

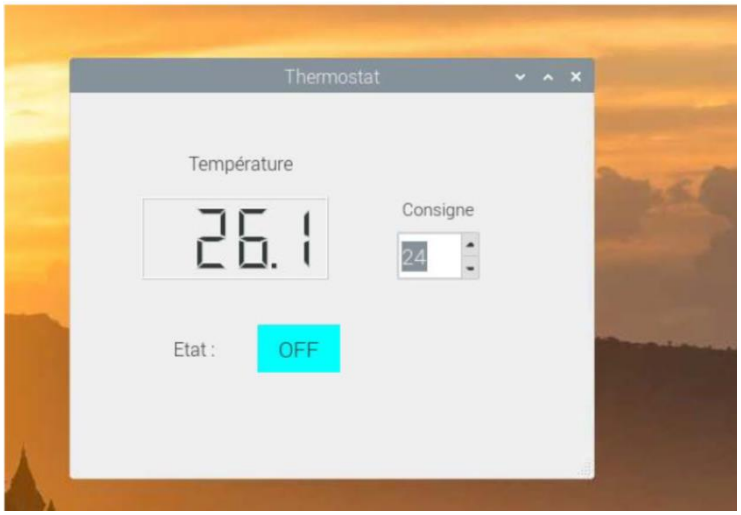
EVENT-DRIVEN.

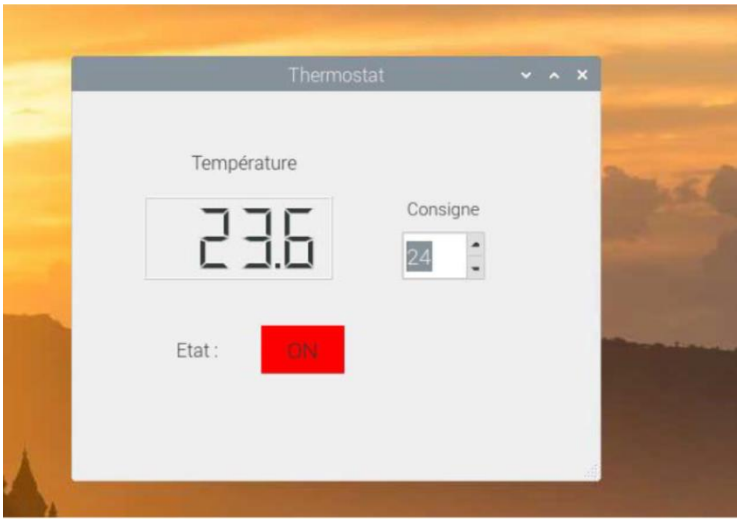
Event-driven programming is mainly used in Graphical User Interface (GUI). A GUI is composed of objects such as Windows, Button, Menu, Edit boxes… The user can interact with these objects with a mouse, keyboard or other entry device. A program is composed of an event dispatcher and handlers. Events may be generated by the user or by objects like timers.

A widespread software tool to design GUI is Qt, a cross-platform development tool for embedded systems.

The GUI can be designed by a specific tool like QtDesigner and integrated in your software easily. For Python, the PyQt tool generates all you need to write your own application.

In the following thermostat example, a DS18B20 temperature sensor is connected to a Raspberry pi 4. The user enters the setpoint via a spinBox.





The program consists of two parts:

- a Python file with the description of the GUI elements (QT_thermostat_1.py generated by PyQT5 in this example);
- the user code (given below).

```
import sys
from QT_thermostat_1 import *
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from w1thermsensor import W1ThermSensor  # DS18B20 library

sensor = W1ThermSensor() # create software object for sensor


class mywindow(QtWidgets.QMainWindow): # this class is your GUI

    def __init__(self):  # constructor with some inheritance
        super(mywindow, self).__init__()
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)

        self.timer = QTimer()  # timer initialization
        self.timer.timeout.connect(self.handleTimer)
        self.timer.start(1000)

    def handleTimer(self):  # the only event driver in this example
        # get temperature and display it
        temperature=sensor.get_temperature()
        self.ui.lcdNumber.display(str(round(temperature,1)))

        # read setpoint from spinBox
        consigne=self.ui.spinBox.value()

        # display heater state in the label element
        if consigne > temperature :
            self.ui.lbl_state_val.setText("ON")
            self.ui.lbl_state_val.setStyleSheet("background-color: red")
        else :
            self.ui.lbl_state_val.setText("OFF")
            self.ui.lbl_state_val.setStyleSheet("background-color: cyan")

app = QtWidgets.QApplication([])
application = mywindow()
application.show()
sys.exit(app.exec())
```

You notice that the loop structure is found in the timer handler.

INDEPENDENT TASKS.

In the previous paradigms, we used routines (i.e. pieces of code) running in a sequential way inside an infinite loop. The execution of these routines was sometime dependant of the state of the system.

Another approach consists in dividing a program in tasks to be carried out simultaneously.

In the thermostat example, you will have four tasks: one to read temperature and display it, one to read the setup and display it, one to command the heater and one to adapt the luminosity of the display.

Two questions arise with this approach: how to run task concurrently and how to exchange data between them.

Running tasks concurrently may be done in two ways: allocating a processor for each task (multiprocessing) or sharing a unique processor with all the tasks (processor sharing).

Data exchange can be done by using shared memories. It implies to have a synchronization mechanism between the tasks. For instance, to display a sensor value you have to wait until this value is given by the task which collect it. This leads to a programming model called "producer consumer pattern".

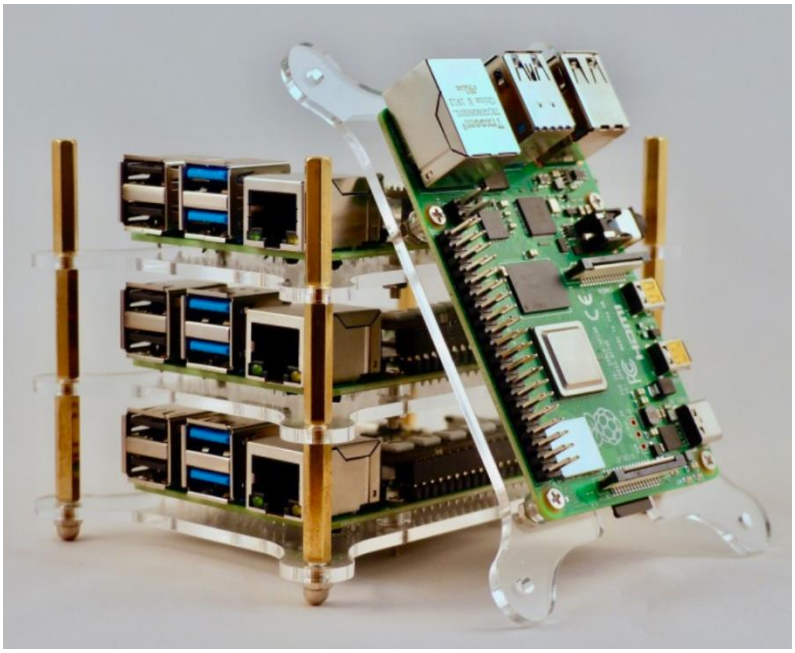
Multiprocessing.

There are few multi-core microcontrollers aimed to the low-end embedded systems market. The Propeller from Parallax Inc. is an example of such a device (https://www.parallax.com/propeller-multicore-concept/). The RPi 2040 is another example, but it has only two cores.

Another approach of multiprocessing is using several computers in a cluster. The word supercomputer is often used for this kind of configuration.

In the past, that was a synonym of high speed very expensive configuration, but nowadays supercomputing with Raspberry Pi is affordable and can be effective for certain classes of applications.

In these configurations, one computer must play the role of a coordinator.



Clustering Raspberry PI.
Photo https://magpi.raspberrypi.org/articles/build-a-raspberry-pi-cluster-computer,
see also Hackspace magazine 41 p79.

Processor sharing.

Processor sharing refers to the fact that each task can receive the processor for a slice of time. Several strategies can be used to give the processor to the tasks with or without real time constraints.

To share your processor, you need at least two extra software components:

- a **scheduler** to decide whose turn is next among the tasks;
- a **context switch** to do all the mechanics of switching CPU from one piece of code to another.

How and when the scheduler chooses to switch to another task will define the reactivity of the system.

This leads to three types of systems:

- not real-time systems, with not defined deadlines;
- soft real-time systems, deadlines are important but you can miss one;
- hard real-time systems, with strict deadlines that cannot be exceeded.

In hard real-time systems, each task is allocated a priority and a mechanism called pre-emption occurs. Pre-emption is the fact to give the CPU to the higher priority task as soon as it is able to run.

Soft real-time systems use a more collaborative approach. Each task tries to return the processor as fast as possible but will not be pre-empted.

This article gives a good comparison between soft and hard real-time systems: http://digitalthinkerhelp.com/15-differences-between-hard-real-time-and-soft-real-time-system-with-examples/

Windows or Linux, for example, does not allow for real-time behaviours because these operating systems can block the processor for an undefined period.

By configuring Linux correctly, it is, however, possible to develop soft real-time applications.

Linux programming (often on Raspbian) or using threads in Python will not be discussed here.

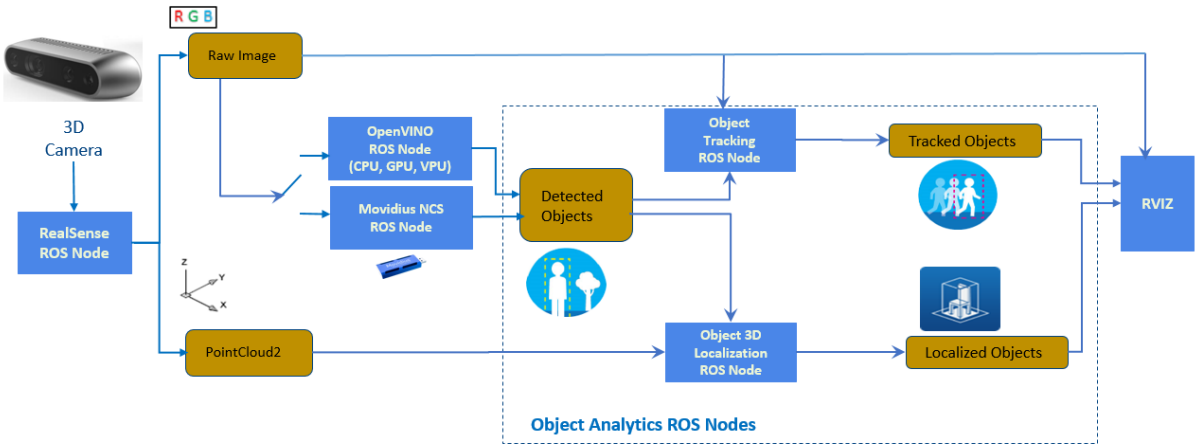
We will concentrate in the following sections on multitasking with a hard real-time system for simple (but efficient) configurations, named FreeRTOS.

DISTRIBUTED COMPUTING.

In a distributed computing architecture (in the context of embedded systems), several embedded systems take responsibility for different tasks and communicate with each other through data buses. To manage all the system activities you need a conductor: a specific operating system.

One example of such a system is ROS (Robotic OS). It is a meta-operating system aimed to the field of robotics.

See : *https://en.wikipedia.org/wiki/Robot_Operating_System*



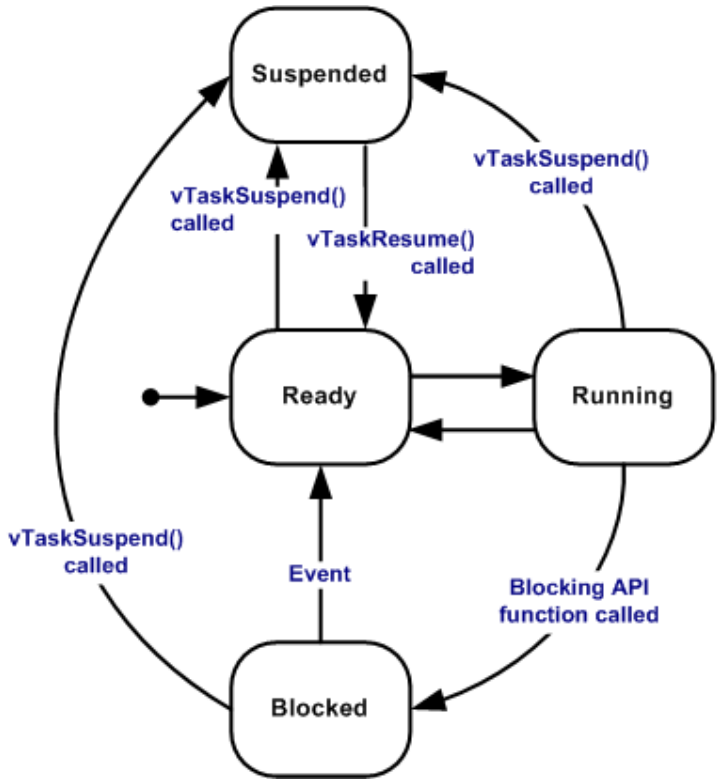
Tasks and scheduling

In the multitasking paradigm, an application is made up of chunks of code called tasks, each one responsible for a part of the application.

Each task, in FreeRTOS, is implemented as a function and must be known by the scheduler through a proper declaration. This declaration binds the function to a Task Control Block (TCB) structure and to a stack memory block specific to each task.

Each task is given a specific priority level and can be in one of the following states:

- **Running** (the task uses the CPU).
- **Ready** (the task is ready to use the CPU but a task with a higher priority is currently using it).
- **Blocked** (the task is waiting for an event: a timing event, a data arrival or synchronization).
- **Suspended** (it's a specific case of blocking. The task can switch to Ready state only if another task gives it the permission).



Tasks states and possible transitions (from freertos.org)

The scheduler manages the TCBs and decides, depending on system events or system calls, in which state a task must be.

In FreeRTOS the scheduler uses the following rule:

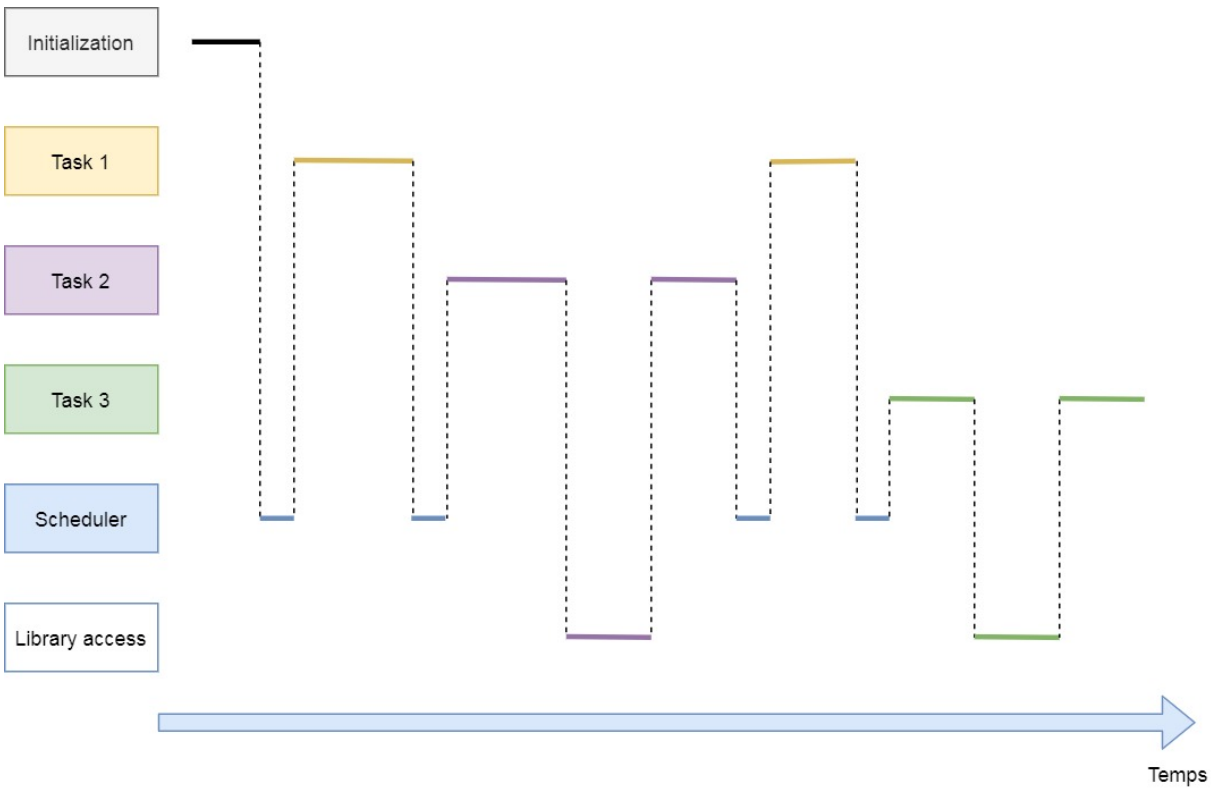
"The highest priority task, neither blocked nor suspended, must have the CPU."

If a switch of the CPU between two tasks is necessary, the scheduler calls the context switch routine to perform this operation.

TCB structure contains pointers and data necessary to the control of the tasks.

The task stack contains the local variables of the task and its context. The context is the set of data necessary to store and retrieve the state of the CPU when a switch is performed (address of next instruction and CPU's internal registers).

The switching between tasks can be visualized by a timing diagram:

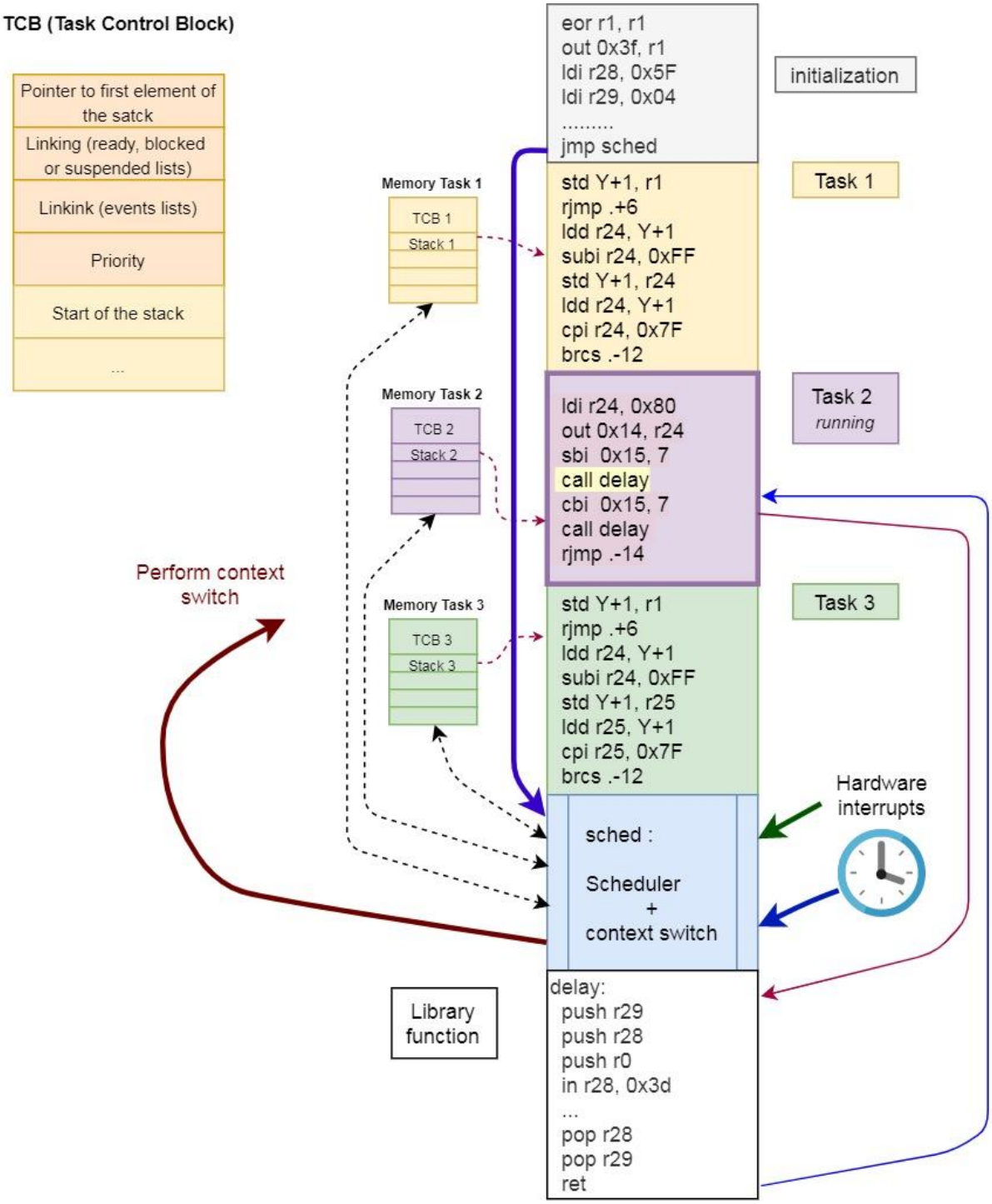


Samples of task switching in an RTOS.

Note: if several tasks have the same priority, while they are not blocked or suspended, they receive, each in turn, an equal amount of CPU time.

Atomicity.

An operation is atomic if it cannot be interrupted by any means. In a CPU, the level of atomicity is the machine instruction. So an instruction of a high-level language, for instance a simple addition, or a call to a library function can be interrupted before its completion. It can lead to serious problems if we are not aware of this situation (see below: reentrancy)



Low-level view of a multitasking program.

The semaphores and mutexes.

In the early days of railroad operation, a semaphore was a mechanical signal with a rotating arm controlling access to a track and preventing train collision.

Open – train can proceed

Closed – train must stop

From https://fr.wikipedia.org/wiki/Sémaphore_(signalisation_ferroviaire)

In an operating system, a semaphore is a software element (variable and associated functions) used to protect execution of critical pieces of code.

The two most common semaphores are mutexes and binary semaphores.

Mutexes.

The purpose of a mutex is to guarantee atomicity for a block of code (called a resource). This block of code is most often a peripheral access routine or a non-reentrant library function.

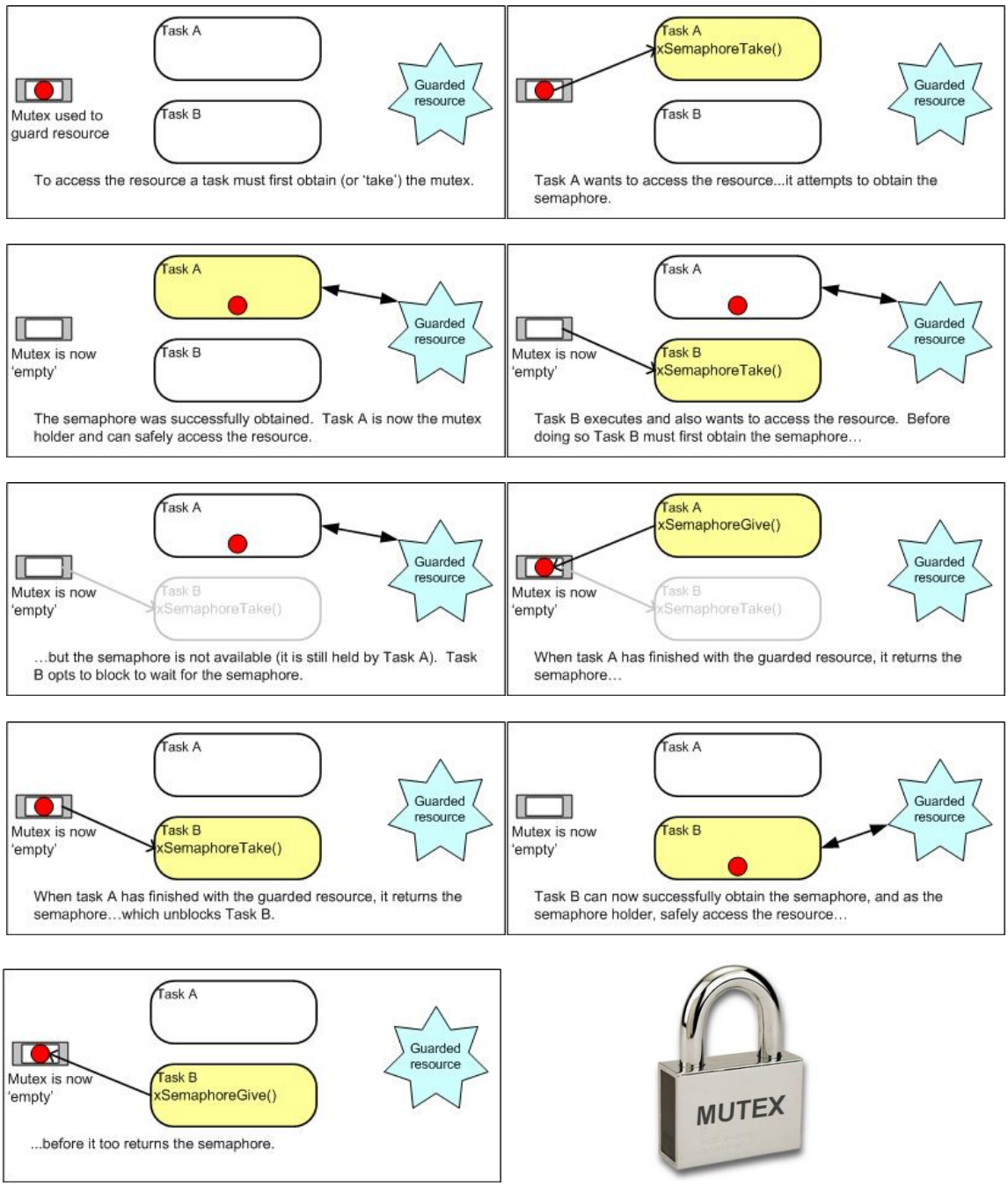
When a task wants to access the mutex protected block, it must ask for the mutex state.

If it is free (open in the railroad analogy), the task can execute the code and the mutex is set unavailable (closed in the railroad analogy).

If the mutex is not available, the requesting task is blocked.

When a task leaves the protected code section, it returns the semaphore (set it open in the railroad analogy).

In the following figures, we use the analogy of token passing to illustrate the mutex operation.



From freertos.org

Mutexes are generally associated with a mechanism of priority rising.

The task obtaining the mutex has its priority raised to a high level so it is likely not to be preempted and can leave the critical section of code as soon as possible.

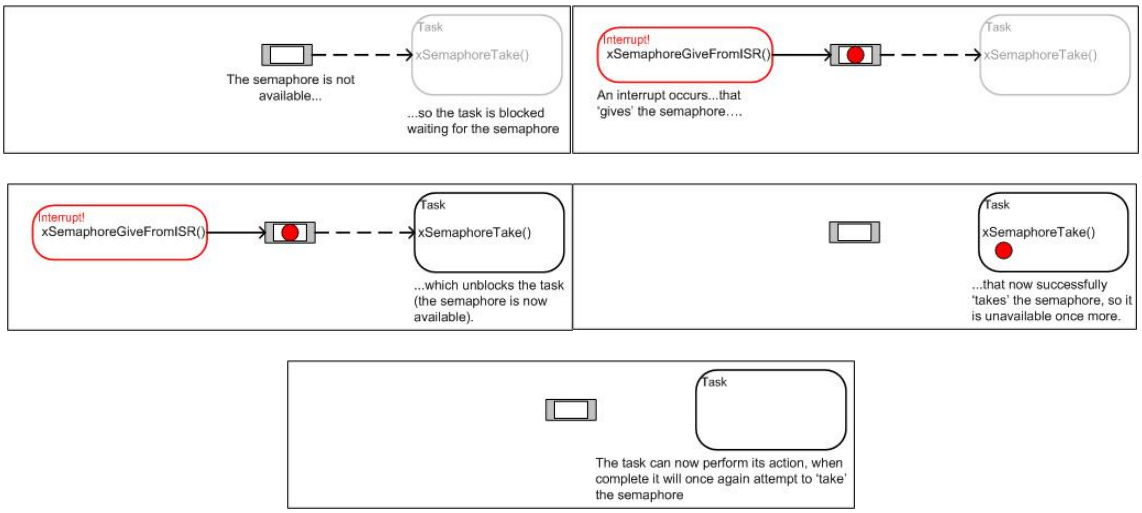
Binary semaphores.

A binary semaphore is used to synchronize tasks. It blocks a task until an event occurs, for instance data arrival signalled by a hardware interrupt.

Unlike mutexes, binary semaphores do not imply a priority rising.

When initialized, a binary semaphore is generally set as not available (closed).

In the following figures, we use the same analogy than previously to illustrate the binary semaphore operation.

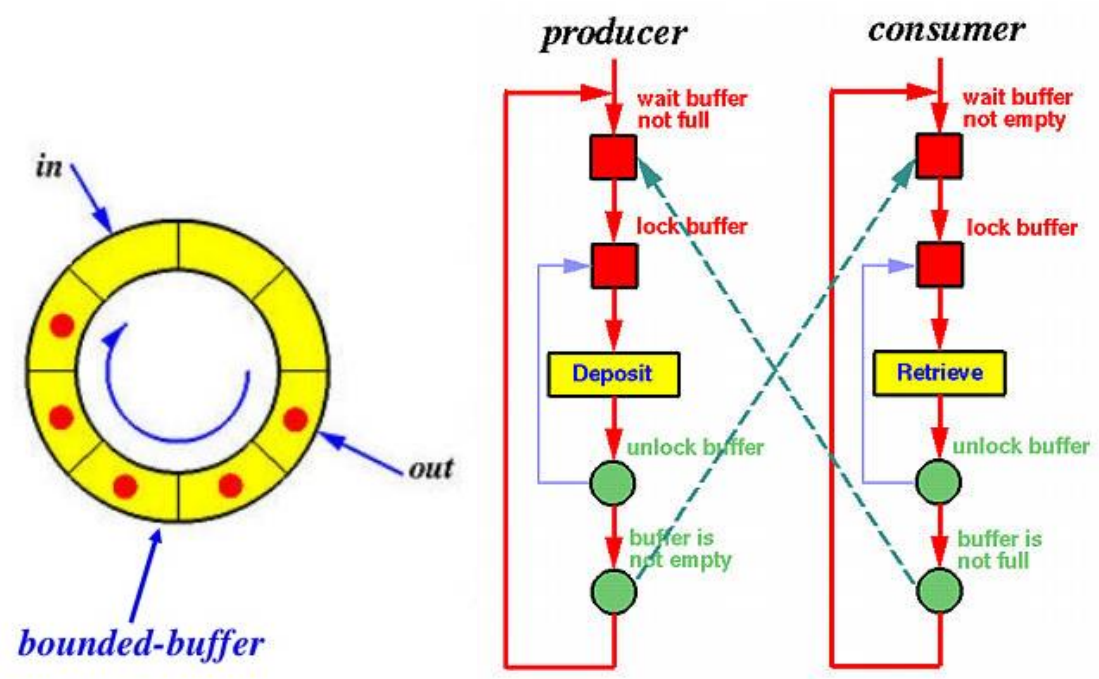


From freertos.org

Communication between tasks.

Communication between tasks (i.e. exchange of data) generally uses a technique called "producer – consumer method". Shared memories are used to deposit and retrieve data. These memories take the form of a FIFO implemented as a circular buffer.

The producer task and the consumer task use a mutex to protect the read and write operation within the FIFO, and two binary semaphores to block tasks, one if the FIFO is empty, one if the FIFO is full.



https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html

We may encounter models that are more elaborate with several producers or several consumers.

Buffer, mutex and semaphores are very often embedded in one software object called a **queue**.

Interrupts management.

When an interrupt occurs, it occurs within the scope of the current task. This requires special actions that will not be developed in this first approach to RTOS. This case will be discussed a little bit when we study FreeRTOS.

Problems to be aware of to get a reliable program.

Problems with semaphores.

In a multitasking, problems may occur when semaphores are used.

The first one is priority inversion:

- A low priority task takes a semaphore.
- A high priority task asks for the semaphore and is blocked because the semaphore is owned by the low priority task.
- A medium priority task preempts the low priority task preventing the low priority task to finish its critical block and release the semaphore.

It is therefore as if the medium priority task blocked the high priority task, hence the name "priority inversion".

In most systems, as explained earlier for the mutex, when a task take a mutex its priority is raised to avoid any preemption. This mechanism is not implemented for the binary semaphore.

Exercice.

Draw a timing diagram for the priority inversion situation.

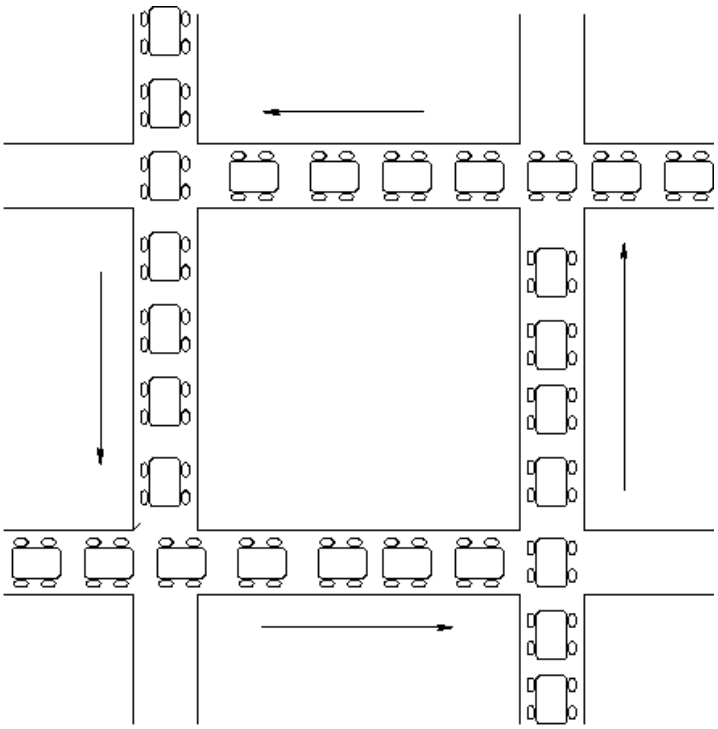
A second one is deadlock also called deadly embrace.

It involves the simultaneous use of two semaphores by two tasks.

Imagine we have two semaphores S1 and S2.

- S1 is owned by Task1.
- S2 is owned by Task2.
- Task1 request S2 before releasing S1.
- Task2 request S1 before releasing S2.

At this time the system is completely stuck.



Source : http://www.cs.rpi.edu

This picture illustrates a deadlock situation.

Exercice.

Draw a timing diagram for a deadlock situation.

The sharing of resources.

The two main types of resources are peripherals and software utility routines.

Peripherals: a hardware output or input device controlled by a piece of software called a driver. For instance, a remote terminal connected by a USB link and acting as a console for our program `Serial.print( )` instruction.

Software utility routines: your own routines or library functions like `sprintf()` or `strcpy()`.

Problems with peripherals: when you send data to a peripheral from a task, preemption can occur within the time interval needed to transfer all your data to the output buffer. If another task starts a data transfer for the same peripheral, the data of both tasks will be mixed up and therefore unreadable.

Problems with software utility routines: if a task calls a library function, for instance `sprintf( )`, and preemption occurs and then another task call the

same function, the first execution of the function is not finished while the function is called again. It could be a problem depending on the fact that the function is reentrant or not.

Reentrancy: *if multiple invocations can safely run concurrently on a single processor system, where a reentrant procedure can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution.* (from *https://en.wikipedia.org/wiki/Reentrancy_(computing)*)

Several conditions are necessary to make a function reentrant. One of the most important is to use local variable for internal buffering (or no variable at all), so at each call a set of new variable will be created.

For example, the `sprintf( )` is not reentrant because it uses an internal global buffer (the same at each call), the `strcpy(char * dest, const char* src)` is re-entrant because it has no internal variable at all and works only with addresses to external variables. These addresses are stored at each call in the task's stack.

Here are some solutions to peripherals sharing and reentrancy problems (from the worst to the best).

- Disable all interrupts.
- Suspend scheduler.
- Use mutexes.
- Use a special task named a gatekeeper. This a task dedicated to access the resource, all the tasks must use the gatekeeper to access the resource.

The first two solutions prevent all interactions with the environment. A bad idea for a realtime system aimed at quick reactions from external conditions.

Schedulability Analysis.

If our program is composed of a list of tasks, the question arises to know if we will have enough time to make our program meet its requirements.

In real time programing, we will encounter several types of tasks:

- Background tasks with no deadline.
- Periodic tasks, i.e. with fixed and recurring deadlines.
- Tasks with varying deadline depending on external conditions.

If a task has a deadline, the rule is very simple: it must meet its deadline.

For periodic tasks the first condition is to have enough CPU time to perform all the tasks in respecting their timing requirements. It means that the CPU usage (U) must be lower than 100%.

If for each task i we have :

- C_i = task i duration
- T_i = task i period

Then we must have (n=number of tasks) :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} < 1$$

This condition is necessary, not sufficient.

The *Liu and Layland* formula gives us a least upper bound condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} < n(2^{\frac{1}{n}} - 1)$$

If this condition is met, it exists a feasible schedule that will always meet the deadlines.

If the system is schedulable, the Rate Monotonic Scheduling (R.M.S.) algorithm gives us the method to choose priority:

If

- T_i = task i period.
- P_i = task i priority.

Then you must choose $P_i > P_j$ if $T_i < T_j$

In one word: the smaller the period (i.e. the higher the repetition rate), the greater the priority.

For task having deadlines depending on external conditions, you must consider another method called Earliest Deadline First (E.D.F.).
See: https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling

FreeRTOS is distributed freely under the MIT open source license.

The API (Application programming Interface), i.e. the list of functions to control FreeRTOS, can be found on https://www.freertos.org/.

Tasks management.

List of some useful functions.

For each function, you have to check its complete definition and meaning in the FreeRTOS API documentation.

```
void vTaskStartScheduler( void );
```
Starts the RTOS scheduler

```
void vTaskSuspendAll( void );
```
Suspends the scheduler without disabling interrupts:

```
BaseType_t xTaskResumeAll( void );
```
Resumes the scheduler after it was suspended:

```
TaskHandle_t
```
Type of the TCB.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        configSTACK_DEPTH_TYPE usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask
                        );
```
Task creation: creates a new task and add it to list of tasks ready to run.
See: https://www.freertos.org/FAQMem.html#StackSize for an interesting discussion about stack size and memory requirements.

```
void vTaskDelete( TaskHandle_t xTask );
```
Remove a task from the the FreeRTOS kernel.

Note : a function name starting with v indicates that the function returns no value, with an x it indicates that a status or error code is returned.

Example.

For each piece of code, you have to:

- Read it carefully and understand it.
- Test it on the Arduino Mega Board with the FreeRTOS libraries 10.4.4. installed.
- Answer questions under the code (by making the code or searching the answer on the web).

1) Two tasks with a software delay.

In this example, we are going to create two tasks, with the same priority, which send a message to the console, enter a loop to make a delay, and then start again forever.

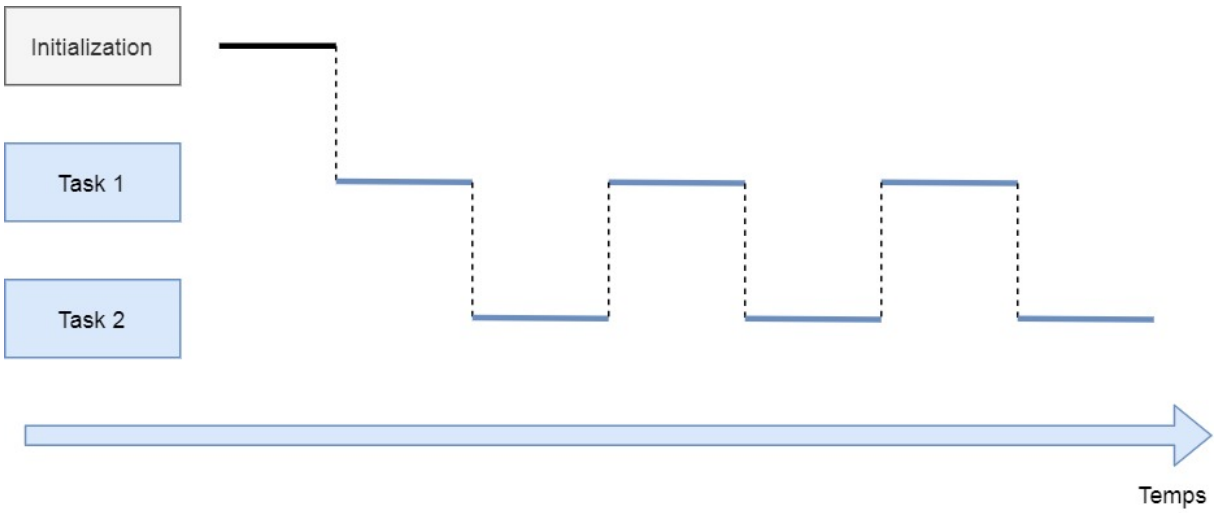
This is not a good way to use a multitasking system, at all! However, it will bring out to your attention some important problems.

```
/**
 * @file ex01_2tsk_soft_del.ino
 * @brief 2 tasks with same priority and a software delay loop
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 11/8/2021
 */
#include <Arduino_FreeRTOS.h>

/* Used as a loop counter to create a very crude delay. */
#define mainDELAY_LOOP_COUNT  400000UL //

/* The task functions. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );


// The setup function runs once when you press reset or power on the board
void setup()
{
// initialize serial communication at 9600 bits/s:
Serial.begin(9600);

// Now set up two tasks, with same priority to run independently.
//Create one of the two tasks.
 xTaskCreate(  vTask1,   // Pointer to the function that implements the task.
    "Task 1", // Text name for the task.  For debugging purpose.
    200,      // This stack size could be adjusted by reading the Stack Highwater
```

```
    NULL,     // We are not using the task parameter.
    1,        // Priority from 0 (min) to (configMAX_PRIORITIES - 1) (max)
    NULL );   // We are not using the task handle.

// for Higwater see : uxTaskGetStackHighWaterMark in FreeRTOS documentation

//Create the other task in exactly the same way.
xTaskCreate( vTask2, "Task 2", 200, NULL, 1, NULL );


//Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                 // present in the initVariant() function of the Free RTOS library

}

void loop()
{
// Empty. Things are done in tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

void vTask1( void *pvParameters )
{
const char pcTaskName[] = "Task 1 is running";
volatile uint32_t ul;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
  //Print out the name of this task.
  Serial.println( pcTaskName );

  //Delay for a period.
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
      /* This loop is just a very crude delay implementation.  There is
      nothing to do in here.  Later exercises will replace this crude
      loop with a more appropiate delay/sleep function. */
    }
  }
}
/*-----------------------------------------------------------*/

void vTask2( void *pvParameters )
{
const char pcTaskName[] = "Task 2 is running";
volatile uint32_t ul;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
  //Print out the name of this task.
  Serial.println( pcTaskName );
```

```
    //Delay for a period.
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
      {
      /* This loop is just a very crude delay implementation.  There is
      nothing to do in here.  Later exercises will replace this crude
      loop with a more appropiatedelay/sleep function. */
      }
  }
}
```

Timing diagram.



Exercises for example 1.

a) What happens if you change `mainDELAY_LOOP_COUNT` *to 400. Why ?*

b) Devise a method to check the duration of the time slice allocated to each task.

c) What happens if you change the priority of one of the tasks ?

d) Why is "volatile" necessary in the declaration of the `ul` *variable ?*

e) Explain how you can more precisely choose the stack size for your tasks.

Task management functions (continued).

```
void vTaskDelay( const TickType_t xTicksToDelay );
```
Delay a task for a given number of ticks.

```
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime,
                      const TickType_t xTimeIncrement );
```
Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution rate.

```
volatile TickType_t xTaskGetTickCount( void );
```
Returns the count of ticks since vTaskStartScheduler was called.

Examples.

2) Two tasks with a delay using `vTaskDelay()`.

In this example, we are going to create two tasks, with different priorities, which send a message to the console and then delayed themselves.

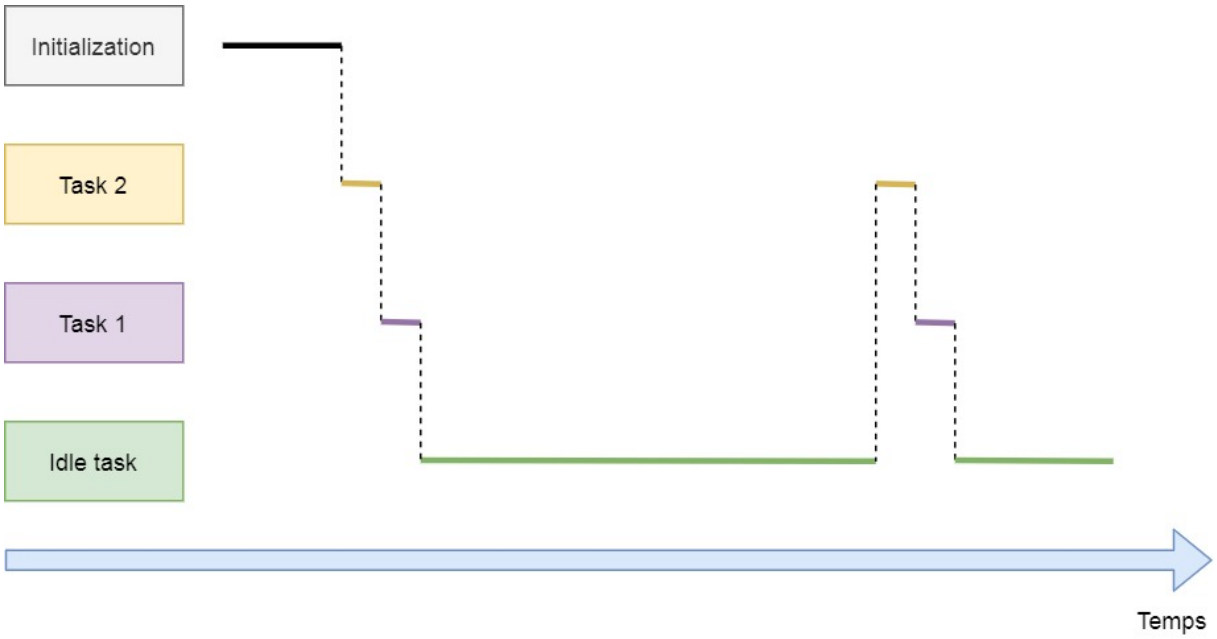
That's one of the methods to use when you want to implement periodic tasks (the simplest but not the more accurate). When the task has finished its current work, it is preempted and goes to the blocked state until the delay is finished.

```
/**
 * @file ex04_2tsk_tskdel.ino
 * @brief 2 tasks with different priority and a delay by FreeRTOS
 * delay start at end of task
 *FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 11/8/2021
 */
#include <Arduino_FreeRTOS.h>

const char pcTextForTask1[] = "Task 1 is running";
const char pcTextForTask2[] = "Task 2 is running";

/* The task functions. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );


// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

// Now set up two tasks to run independently.
```

```
//Create one of the two tasks.
 xTaskCreate(  vTask1,   // Pointer to the function that implements the task.
    "Task 1", // Text name for the task.  For debugging purpose.
    200,      // This stack size could be adjusted by reading the Stack Highwater
    (void*)pcTextForTask1,   // Pass the text to be printed as the task parameter.
    1,        // Priority from 0 (min) to (configMAX_PRIORITIES - 1) (max)
    NULL );   // We are not using the task handle.

  /* Create the other task in exactly the same way. */
  xTaskCreate( vTask2, "Task 2", 200, (void*)pcTextForTask2, 2, NULL );

//Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                  // present in the initVariant() function of the Free RTOS library
}


void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

void vTask1( void *pvParameters )
{
const char* pcTaskName;

/* The string to print out is passed in via the parameter.
 *  Cast this to a  character pointer. */
 pcTaskName = (char * ) pvParameters;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    //Print out the name of this task.
     Serial.println( pcTaskName );
    /* Delay for some time.  This time we use a call to vTaskDelay() which
    puts the task into the Blocked state until the delay period has expired.
    The delay period is specified in 'ticks'. */
    vTaskDelay( 1000 / portTICK_PERIOD_MS );
  }
}
/*-----------------------------------------------------------*/

void vTask2( void *pvParameters )
{
const char *pcTaskName ;

/* The string to print out is passed in via the parameter.
 *  Cast this to a  character pointer. */
pcTaskName = (char *) pvParameters;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    //Print out the name of this task.
     Serial.println( pcTaskName );
```

```
    /* Delay for some time.  This time we use a call to vTaskDelay() which
    puts the task into the Blocked state until the delay period has expired.
    The delay period is specified in 'ticks'. */
    vTaskDelay( 1000 / portTICK_PERIOD_MS );
  }
}
```

Timing diagram.



The diagram lets appear a new task : idle. This task is provided by FreeRTOS and do some background stuff. When all tasks of a RTOS system are blocked, this task is called. Its priority is the lowest of the system.

Exercises for example 2.

a) What happens if you change the delay of task 2 to 500 ms?

b) Explain the tick mechanism. What is its interaction with the scheduler.

c) Find the tick duration. How is implemented, in hardware, the tick timer in your system.

d) Draw a timing diagram including the state of the task. Use colours to denote states (red = blocked, orange = ready, green = running).

e) When starts precisely the delay of each task? Is it appropriate if you need a regular and accurate sampling?

3) Two tasks with a delay using `vTaskDelay()` *with only one function for both tasks.*

```
/**
 * @file ex04b_2tsk_tskdel.ino
 * @brief 2 tasks with different priority and a delay by FreeRTOS
 * uses the same code for both tasks
 * delay start at end of task
 *FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 11/8/2021
 */
#include <Arduino_FreeRTOS.h>

const char pcTextForTask1[] = "Task 1 is running";
const char pcTextForTask2[] = "Task 2 is running";

/* The task function. */
void vTask( void *pvParameters );


// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

// Now set up two tasks to run independently.
//Create one of the two tasks.
 xTaskCreate(  vTask,   // Pointer to the function that implements the task.
    "Task 1", // Text name for the task.  For debugging purpose.
    200,      // This stack size could be adjusted by reading the Stack Highwater
    (void*)pcTextForTask1,   // Pass the text to be printed as the task parameter.
    1,        // Priority from 0 (min) to (configMAX_PRIORITIES - 1) (max)
    NULL );   // We are not using the task handle.

  /* Create the other task in exactly the same way. */
  xTaskCreate( vTask, "Task 2", 200, (void*)pcTextForTask2, 2, NULL );

//Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                 // present in the initVariant() function of the Free RTOS library
}


void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks --------------------*/
/*-----------------------------------------------------*/

void vTask( void *pvParameters )
{
const char* pcTaskName;

/* The string to print out is passed in via the parameter.
```

```
 *  Cast this to a  character pointer. */
 pcTaskName = (char * ) pvParameters;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    //Print out the name of this task.
     Serial.println( pcTaskName );
    /* Delay for some time.  This time we use a call to vTaskDelay() which
    puts the task into the Blocked state until the delay period has expired.
    The delay period is specified in 'ticks'. */
    vTaskDelay( 1000 / portTICK_PERIOD_MS );
  }
}
```

Exercises for example 3.

a) How could you have two tasks with different delays, using just one function? (Hint: instead of a single string, pass a structure to the tasks).

b) Using the "Low-level view of a multitasking program" diagram given earlier in the text, make modifications to represent the current program.

4) Two tasks with a delay using `vTaskDelayUntil()`.

In this example, we are going to create two tasks, with different priorities, which send a message to the console and then delayed themselves.

Here, we keep track of time and compensate the code execution delays to perform a precise sampling rate.

```
/**
 * @file ex05_2tsk_tskdelabs.ino
 * @brief 2 tasks with different priority and a delay by FreeRTOS
 * delay is absolute, it is a kind of periodic sampling rate
 * *FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 12/8/2021
 */
#include <Arduino_FreeRTOS.h>

const char pcTextForTask1[] = "Task 1 is running";
const char pcTextForTask2[] = "Task 2 is running";

/* The task functions. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );
```

```
// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

// Now set up two tasks to run independently.
// Create one of the two tasks.
  xTaskCreate(  vTask1,   // Pointer to the function that implements the task.
    "Task 1", // Text name for the task.  This is to facilitate debugging only.
    200,      // This stack size could be adjusted by reading the Stack Highwater
    (void*)pcTextForTask1,   // Pass the text to be printed as the task parameter.
    1,        // Priority from 0 (min) to (configMAX_PRIORITIES - 1) (max)
    NULL );   // We are not using the task handle.

  /* Create the other task in exactly the same way. */
  xTaskCreate( vTask2, "Task 2", 200, (void*)pcTextForTask2, 2, NULL );

//Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                  // present in the initVariant() function of the Free RTOS library
}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

void vTask1( void *pvParameters )
{
const char *pcTaskName;
TickType_t xLastWakeTime;

 /* The string to print out is passed in via the parameter.  Cast this to a
  character pointer. */
pcTaskName = ( char * ) pvParameters;

   /* The xLastWakeTime variable needs to be initialized with the current tick
  count.  Note that this is the only time we access this variable.  From this
  point on xLastWakeTime is managed automatically by the vTaskDelayUntil()
  API function. */
xLastWakeTime = xTaskGetTickCount();

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* Print out the name of this task. */
    Serial.println( pcTaskName );

    /* We want this task to execute exactly every 1000 milliseconds.  As per
    the vTaskDelay() function, time is measured in ticks, and the
    portTICK_PERIOD_MS constant is used to convert this to milliseconds.
    xLastWakeTime is automatically updated within vTaskDelayUntil() so does not
    have to be updated by this task code. */

    vTaskDelayUntil( &xLastWakeTime, ( 1000 / portTICK_PERIOD_MS ) );
```

```
    //  you have to give the address of xLastWakeTime
  }
}


void vTask2( void *pvParameters )
{
const char *pcTaskName;
TickType_t xLastWakeTime;

 /* The string to print out is passed in via the parameter.  Cast this to a
  character pointer. */
pcTaskName = ( char * ) pvParameters;

   /* The xLastWakeTime variable needs to be initialized with the current tick
  count.  Note that this is the only time we access this variable.  From this
  point on, xLastWakeTime is managed automatically by the vTaskDelayUntil()
  API function. */
xLastWakeTime = xTaskGetTickCount();

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* Print out the name of this task. */
    Serial.println( pcTaskName );

    /* We want this task to execute exactly every 1000 milliseconds.  As per
    the vTaskDelay() function, time is measured in ticks, and the
    portTICK_PERIOD_MS constant is used to convert this to milliseconds.
    xLastWakeTime is automatically updated within vTaskDelayUntil() so does not
    have to be updated by this task code. */

    vTaskDelayUntil( &xLastWakeTime, ( 1000 / portTICK_PERIOD_MS ) );
    // you have to give the address of xLastWakeTime
  }
}
```

Exercises for example 4.

a) What happens if a higher priority task is using CPU when delay deadline occurs?

b) What the difference between `vTaskDelayUntil()` *and* `xTaskDelayUntil()`.

Task management functions (continued).

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t xTask );
```
Returns the priority of task **xTask**.

```
void vTaskPrioritySet( TaskHandle_t xTask,
                       UBaseType_t uxNewPriority );
```
Set the priotiy of task **xTask.**

Example.

4) Two tasks with different priorities and no delay.

Changing priority enables ready task to become running.

```
/**
 * @file ex08_chg_prio.ino
 * @brief 2 tasks running continuously and changing their priorities
 * each other to obtain CPU
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 12/08/2021
 */
#include <Arduino_FreeRTOS.h>

/* The two task functions. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Used to hold the handle of Task2. */
TaskHandle_t xTask2Handle;


// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

 /* Create the first task at priority 2.  This time the task parameter is
  not used and is set to NULL.  The task handle is also not used so likewise
  is also set to NULL. */
xTaskCreate( vTask1, "Task 1", 200, NULL, 2, NULL ); // Priority = 2

  /* Create the second task with priority = 1 - which is lower than the priority
  given to Task1.  Again the task parameter is not used so is set to NULL -
  BUT this time we want to obtain a handle to the task so pass in the address
  of the xTask2Handle variable. */
xTaskCreate( vTask2, "Task 2", 200, NULL, 1, &xTask2Handle );
     /* The task handle is the last parameter ^^^^^^^^^^^^^ */

//Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                 // present in the initVariant() function of the Free RTOS library
}
```

```
void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

void vTask1( void *pvParameters )
{
UBaseType_t uxPriority;

  /* This task will always run before Task2 as it has the higher priority.
  Neither Task1 nor Task2 ever block so both will always be in either the
  Running or the Ready state.

  Query the priority at which this task is running - passing in NULL means
  "return our own priority". */
  uxPriority = uxTaskPriorityGet( NULL );

while(1)
  {
    /* Print out the name of this task. */
    Serial.println( "Task1 is running" );

    /* Setting the Task2 priority above the Task1 priority will cause
    Task2 to immediately start running (as then Task2 will have the higher
    priority of the    two created tasks). */
    Serial.println( "About to raise the Task2 priority" );

    // set to priority of present task + 1
    vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

    /* Task1 will only run when it has a priority higher than Task2.
    Therefore, for this task to reach this point Task2 must already have
    executed and set its priority back down to 0. */
  }
}

/*-----------------------------------------------------------*/

void vTask2( void *pvParameters )
{
UBaseType_t uxPriority;

  /* Task1 will always run before this task as Task1 has the higher priority.
  Neither Task1 nor Task2 ever block so will always be in either the
  Running or the Ready state.

  Query the priority at which this task is running - passing in NULL means
  "return our own priority". */
  uxPriority = uxTaskPriorityGet( NULL );

  for( ;; )
  {
    /* For this task to reach this point Task1 must have already run and
    set the priority of this task higher than its own.
```

```
    Print out the name of this task. */
    Serial.println( "Task2 is running" );

    /* Set our priority back down to its original value.  Passing in NULL
    as the task handle means "change our own priority".  Setting the
    priority below that of Task1 will cause Task1 to immediately start
    running again. */
    Serial.println( "About to lower the Task2 priority" );
    vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
  }
}
```

Exercises for example 5.

a) Draw a timing diagram including the state of the task.

b) Which parameter allows changing the maximum priority value?

Hooks.

A hook is a method used to add you own code to some FreeRTOS functions.

Each time the Idle task is running, it calls the **vApplicationIdleHook()** FreeRTOS function.

The tick interrupt routine does the same with **vApplicationTickHook()**

These FreeRTOS functions have the **weak** attribute and do nothing.

If you provide a function with the same name, yours will be used instead of the empty kernel function.

Idle Hook Function.

```
void vApplicationIdleHook( void );
```

Tick Hook Function.

```
void vApplicationTickHook( void );
```

Sharing resources.

In our programming context, resources are either hardware peripherals or pieces of code (for instance a library function).

In the following example, we will use the console access, via serial port, within 3 tasks and see what's happen.

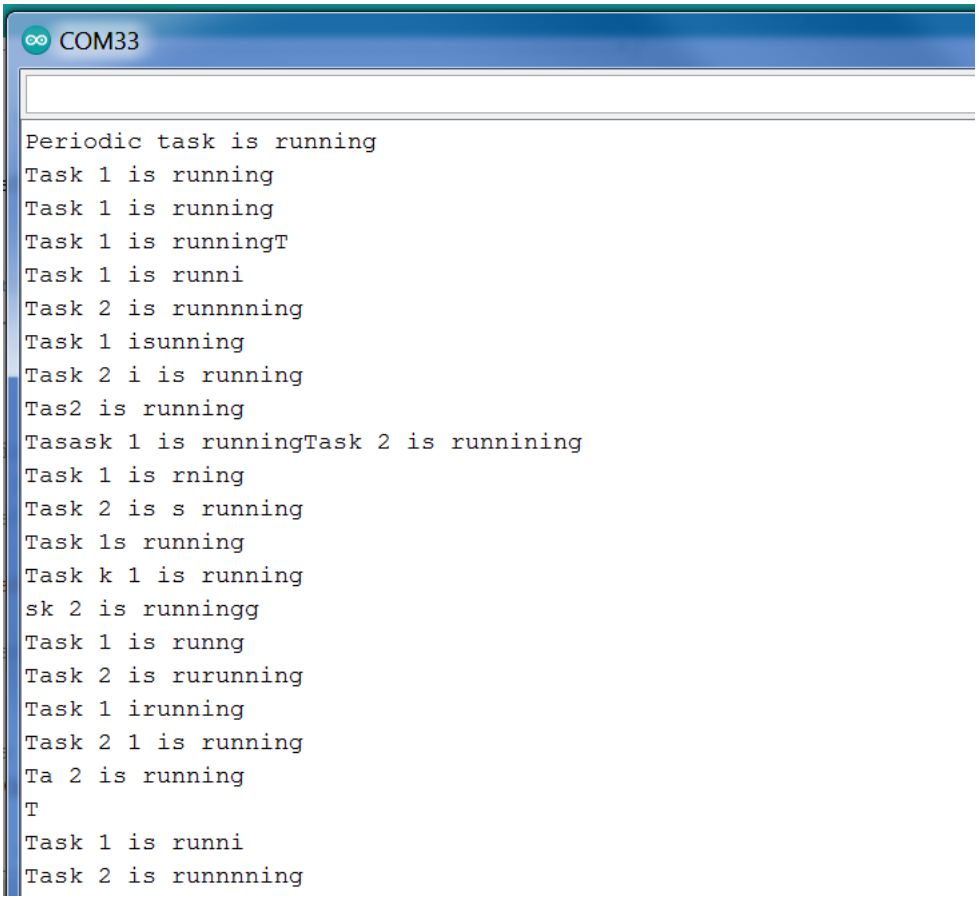
Two tasks (`vContinuousProcessingTask`), with the same code are running continuously and sending messages to the console.

One task is (`vPeriodicTask`) is running periodically with a 500 ms period.

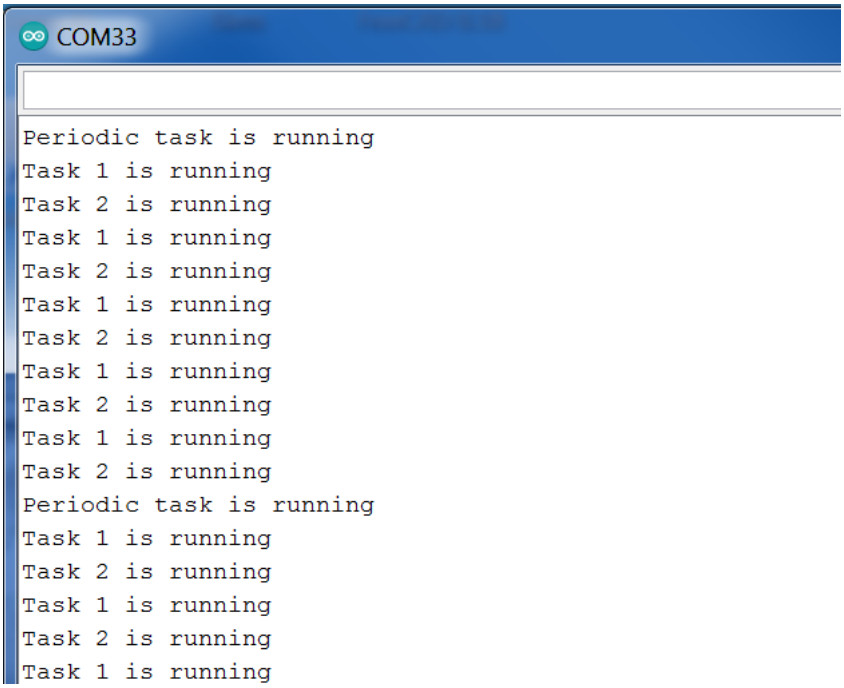
```
/**
 * @file ex06_2tskcont_1tskper.ino
 * @brief 2 continous tasks with same priority
 * and a periodic task (with higher priority)
 * * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 13/8/2021
 */
#include <Arduino_FreeRTOS.h>

const char pcTextForTask1[] = "Task 1 is running";
const char pcTextForTask2[] = "Task 2 is running";
const char pcTextForPeriodicTask[] = "Periodic task is running";


/* The task functions. */
void vContinuousProcessingTask( void *pvParameters );
void vPeriodicTask( void *pvParameters );


// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

// Create two instances of the continuous processing task, both at priority 1.
xTaskCreate( vContinuousProcessingTask, "Task 1", 200, (void*)pcTextForTask1, 1,
NULL );
xTaskCreate( vContinuousProcessingTask, "Task 2", 200, (void*)pcTextForTask2, 1,
NULL );

// Create one instance of the periodic task at priority 2.
xTaskCreate( vPeriodicTask, "Task 3", 200, (void*)pcTextForPeriodicTask, 2, NULL
);

//Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                  // present in the initVariant() function of the Free RTOS library
}
```

```
void loop()
{
  // Empty. Things are done in Tasks.
}

/*--------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*--------------------------------------------------*/


void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

/* The string to print out is passed in via the parameter.  Cast this to a
  character pointer. */
pcTaskName = ( char * ) pvParameters;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1) // continous until preemption occurs
  {
    /* Print out the name of this task.  This task just does this repeatedly
    without ever blocking or delaying. */
    Serial.println(pcTaskName);
   }
}
/*-----------------------------------------------------------*/


void vPeriodicTask( void *pvParameters )
{
TickType_t xLastWakeTime;
char *pcTaskName;

 /* The string to print out is passed in via the parameter.  Cast this to a
  character pointer. */
pcTaskName = ( char * ) pvParameters;


  /* The xLastWakeTime variable needs to be initialized with the current tick
  count.  Note that this is the only time we access this variable.  From this
  point on xLastWakeTime is managed automatically by the vTaskDelayUntil()
  API function. */
xLastWakeTime = xTaskGetTickCount();

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* Print out the name of this task. */
     Serial.println(pcTaskName);

    /* We want this task to execute exactly every 500 milliseconds. */
    vTaskDelayUntil( &xLastWakeTime, ( 500 / portTICK_PERIOD_MS ) );
  }
}
```

The result displayed on the console is the following:



Ugly, isn't it? We would have expected something like this:



Where is the root of the problem?

The problem lies in the fact that the 3 tasks try to use the same USB port COM hardware resource through the function `Serial.println()` and are interrupted by the preemption process before finishing sending their string.

In addition the next running task may overrun the FIFO buffer used by the `Serial.println()` function if there is not enough time to empty it.

In the next program, we are going to look for a solution. This will be a raw one and we will have to improve it.

Let's write a print utility called `Safe_Print_String`.

```
void Safe_Print_String( const char *pcString )
{
/* Print the string, suspending the scheduler as method of mutual
  exclusion. */

vTaskSuspendAll(); // not the best method at all. We try to do better next time!

Serial.println(pcString);
Serial.flush(); // wait until sending buffer is empty

xTaskResumeAll();
}
```

- First we block the scheduler.
- Then we send the string with the regular function.
- After what we use a blocking function `Serial.flush()` to wait until the complete string is sent.
- Finally, we unblock the scheduler.

```
/**
 * @file ex06b_2tskcont_1tskper.ino
 * @brief  2 continous tasks with same priority
 * and a periodic task (with higher priority).
 * First try to solve sharing problem with a safe print function
 * *FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 13/8/2021
 */
#include <Arduino_FreeRTOS.h>

const char pcTextForTask1[] = "Task 1 is running";
const char pcTextForTask2[] = "Task 2 is running";
const char pcTextForPeriodicTask[] = "Periodic task is running";

// safe print function - this is a function, not a task
void Safe_Print_String( const char *pcString );

/* The task functions. */
void vContinuousProcessingTask( void *pvParameters );
```

```
void vPeriodicTask( void *pvParameters );


// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

// Create two instances of the continuous processing task, both at priority 1.
xTaskCreate( vContinuousProcessingTask, "Task 1", 200, (void*)pcTextForTask1, 1,
NULL );
xTaskCreate( vContinuousProcessingTask, "Task 2", 200, (void*)pcTextForTask2, 1,
NULL );

// Create one instance of the periodic task at priority 2.
xTaskCreate( vPeriodicTask, "Task 3", 200, (void*)pcTextForPeriodicTask, 2, NULL
);

 //Start the scheduler so our tasks start executing.
vTaskStartScheduler(); // not necessary in last implementation
                  // present in the initVariant() function of the Free RTOS library
}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

/* The string to print out is passed in via the parameter.  Cast this to a
  character pointer. */
pcTaskName = ( char * ) pvParameters;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1) // continous until preemption occurs
  {
    /* Print out the name of this task.  This task just does this repeatedly
    without ever blocking or delaying. */
    Safe_Print_String( pcTaskName ); // use the safe print function
  }
}
/*-----------------------------------------------------------*/

void vPeriodicTask( void *pvParameters )
{
TickType_t xLastWakeTime;
char *pcTaskName;

 /* The string to print out is passed in via the parameter.  Cast this to a
  character pointer. */
pcTaskName = ( char * ) pvParameters;
```

```
  /* The xLastWakeTime variable needs to be initialized with the current tick
  count.  Note that this is the only time we access this variable.  From this
  point on xLastWakeTime is managed automatically by the vTaskDelayUntil()
  API function. */
xLastWakeTime = xTaskGetTickCount();

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* Print out the name of this task. */
     Safe_Print_String( pcTaskName ); // use the safe print function
    /* We want this task to execute exactly every 500 milliseconds. */
    vTaskDelayUntil( &xLastWakeTime, ( 500 / portTICK_PERIOD_MS ) );
  }
}

/*-----------------------------------------------------------*/

void Safe_Print_String( const char *pcString )
{
/* Print the string, suspending the scheduler as method of mutual
  exclusion. */

vTaskSuspendAll(); // not the best method at all. We try to do better next time!

Serial.println(pcString);
Serial.flush(); // wait until sending buffer is empty

xTaskResumeAll();
}
```

The result is what we expected but at the expense of two **inappropriate** actions in a real-time context:

- Suspending the scheduler, this action must be avoided at all costs.

- Using the blocking function `Serial.flush()` in the context of a high priority task (the task calling the function `Safe_Print_String()`) which slow down the system.

In the next chapters, we will develop the tools to overcome these problems using mutexes and gatekeeper tasks.

Queues.

A Queue is a FreeRTOS object consisting in a FIFO sharable with several tasks along with an embedded mutex for access protection and two binary semaphores one for empty condition indication and one for full condition indication. The queue is the foundation of the producer consumer model.

Queues management functions.

All functions need the inclusion of **queue.h**

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

Creates a new queue and returns a handle by which the queue can be referenced.

- Number of elements: `uxQueueLength`
- Size of each element: `uxItemSize`

QueueHandle_t
Type by which queues are referenced.

```
BaseType_t xQueueSendToBack(QueueHandle_t xQueue,
                            const void * pvItemToQueue,
                            TickType_t xTicksToWait);
```

Post an item to the back of a queue. The item is queued by copy, not by reference. Return **pdTRUE** if queue not full, else **errQUEUE_FULL**.

- Queue object to interact with : **xQueue**
- Pointer to the dada to add: **pvItemToQueue**
- Timeout in ticks: **xTicksToWait** (**portMAX_DELAY** = undefined delay)

```
BaseType_t xQueueSendToFront(QueueHandle_t xQueue,
                             const void * pvItemToQueue,
                             TickType_t xTicksToWait);
```

Post an item to the front of a queue. The item is queued by copy, not by reference. Return **pdTRUE** if queue not full, else **errQUEUE_FULL.**

Both function block the task that tries to write to the queue until timeout occurs if the queue if full.

```
BaseType_t xQueueReceive(QueueHandle_t xQueue,
                         void *pvBuffer,
                         TickType_t xTicksToWait);
```

Receive an item from a queue and remove it from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created. Return **pdTRUE** if queue not empty, else **pdFALSE**.

- Queue object to interact with : **xQueue**
- Pointer to the dada to read: **pvItemToQueue**
- Timeout in ticks: **xTicksToWait** (**portMAX_DELAY** = undefined delay)

```
BaseType_t xQueuePeek(QueueHandle_t xQueue,
                      void *pvBuffer,
                      TickType_t xTicksToWait);
```

Receive an item from a queue without removing it from the queue.

Both function block the task that tries to read from the queue until timeout occurs if the queue if empty.

```
UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

Return the number of messages stored in a queue.

taskYIELD() Macro for forcing a context switch, useful for tasks with same priority.

Exercices.

a) Data are stored dynamically (at execution time) in the queue and can produce a memory overflow. How can we avoid this problem?

b) What happens if several tasks are blocked on the same queue and that the blocking condition is removed?

c) If several producer tasks use the same queue to send data to one consumer task, how can the consumer distinguish where the data come from?

Examples.

1) One producer task with priority 1 sends data each 500 ms to a consumer task with priority 2. Consumer task is blocked while queue is empty or if a timeout of 1000 ms occurs.

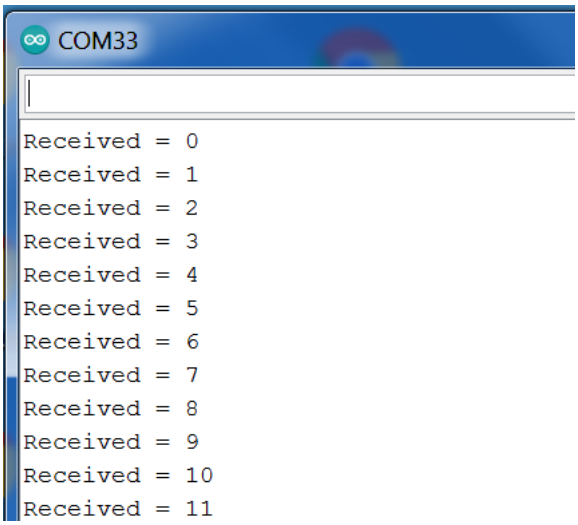
The conditional compilation code is added to create an overflow condition.

```
/**
 * @file ex10_queue_1P_1C_empty_test.ino
 * @brief 1 producer (Sender)tasks with priority=1
 * and 1 customer (Receiver) task with priority=2
 * Producer writes periodically numeric data into queue
 * Customer wait until something is in the queue and displays it
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 31/8/2021
 */
#include <Arduino_FreeRTOS.h>
#include <queue.h>

//#define CHECK_TIMEOUT  //Uncomment to check for timeout

/* The tasks to be created.  Two instances are created of the sender task while
only a single instance is created of the receiver task. */
static void vSenderTask( void *pvParameters );
static void vReceiverTask( void *pvParameters );

/* Declare a variable of type QueueHandle_t.  This is used to store
 a reference to the queue that is accessed by all three tasks. */
QueueHandle_t xQueue;


// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

xQueue = xQueueCreate( 5, sizeof( uint32_t ) ); // 5 uint32_t elements
if( xQueue != NULL ) // if queue object is created
  {
   /* Create the task that will write to the queue with priority 1. */
   xTaskCreate( vProducerTask, "Producer", 200, NULL, 1, NULL );

   /* Create the task that will read from the queue.  The task is created with
   priority 2, so above the priority of the sender task. */
   xTaskCreate( vCustomerTask, "Consumer", 200, NULL, 2, NULL );

  //Start the scheduler so our tasks start executing.
  vTaskStartScheduler(); // not necessary in last implementation
                // present in the initVariant() function of the Free RTOS library
  }
  else
   Serial.println( "Could not create the queue." );
}
```

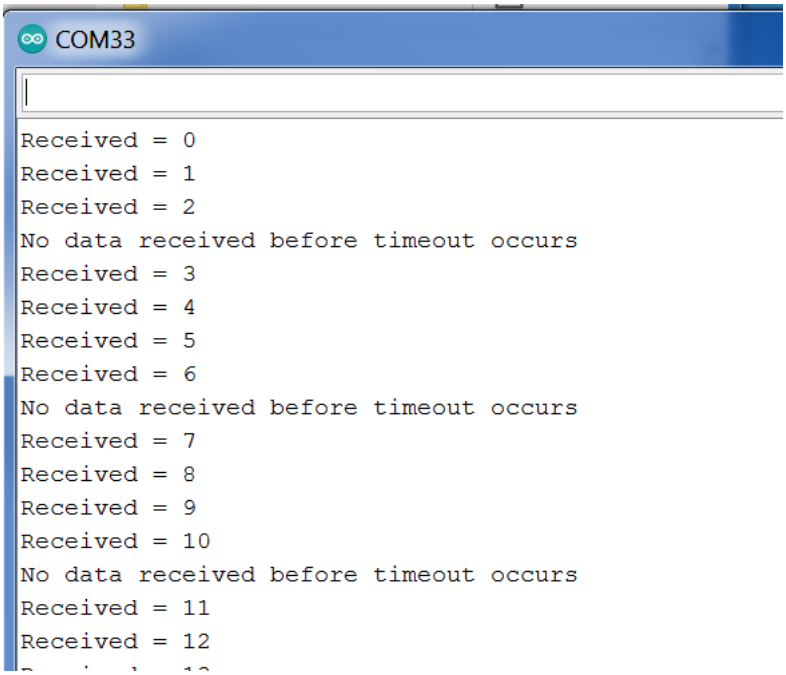
```
void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ----------------------*/
/*-----------------------------------------------------*/
static void vProducerTask( void *pvParameters )
{
uint32_t lValueToSend = 0UL;
uint32_t delay_send = 500 / portTICK_PERIOD_MS;


//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* The first parameter is the queue to which data is being sent.  The
    queue was created before the scheduler was started, so before this task
    started to execute.

    The second parameter is the address of the data to be sent.

    The third parameter is the Block time, the time the task should be kept
    in the Blocked state to wait for space to become available on the queue
    should the queue already be full.  In this case we will block indefinitively
    until the customer frees some data */
   xQueueSendToBack( xQueue, &lValueToSend, portMAX_DELAY );

   lValueToSend++;

   #ifdef CHECK_TIMEOUT
   delay_send+=200/portTICK_PERIOD_MS;
   if (delay_send>= 1100/portTICK_PERIOD_MS)
     delay_send=500 / portTICK_PERIOD_MS;;

   vTaskDelay(delay_send);
   #endif

   #ifndef CHECK_TIMEOUT
   vTaskDelay(delay_send);
   #endif
  }
}

/*-----------------------------------------------------------*/

static void vCustomerTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
uint32_t lReceivedValue;
BaseType_t xStatus;


//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* The first parameter is the queue from which data is to be received.  The
    queue is created before the scheduler is started, and therefore before this
```

```
    task runs for the first time.

    The second parameter is the buffer into which the received data will be
    placed.  In this case the buffer is simply the address of a variable that
    has the required size to hold the received data.

    the last parameter is the block time, the maximum amount of time that the
    task should remain in the Blocked state to wait for data to be available
    should the queue already be empty. */
   xStatus = xQueueReceive( xQueue, (uint32_t *)&lReceivedValue, 1000 /
                                                          portTICK_PERIOD_MS );

   if( xStatus == pdPASS )
     {
      /* Data was successfully received from the queue, print out the received
      value. */
      Serial.print( "Received = ");
      Serial.println( lReceivedValue );
     }
    else
     {
      /* We did not receive anything from the queue even after waiting for 1000ms.
      This must be an error as the sending task will be writing to the queue each
      500 ms. */
      Serial.println( "No data received before timeout occurs" );
     }
  }
}
```

Results :



With code enabled to check for timeout:



Exercise for example 1.

Analyse the timeout testing code and draw a timing diagram of the program with this test.

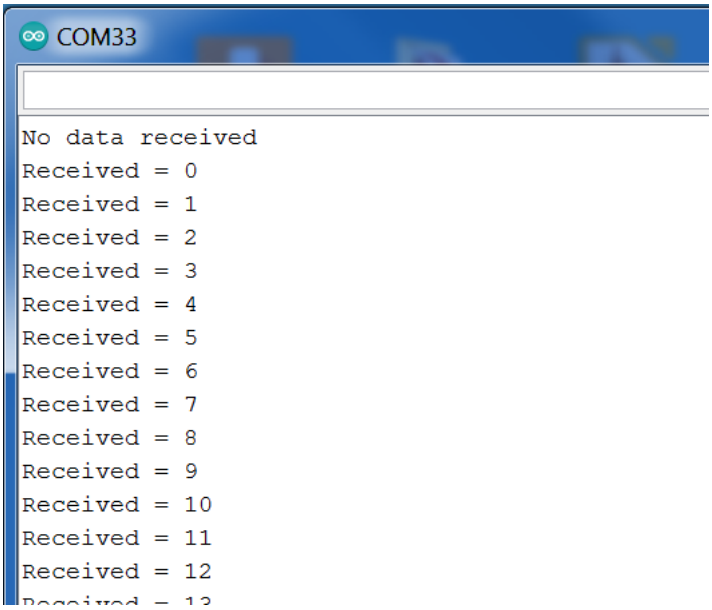
2) One producer task with priority 1 sends data continuously (until it is blocked by a full queue condition) to a consumer task with priority 2. Consumer task reads queue each 500 ms with no timeout.

```
/**
 * @file ex10_queue_1P_1C_full_test.ino
 * @brief one producer (Sender) tasks with priority=1 and one customer (Receiver)
 * task with priority=2
 * Producers write numeric data into queue until queue is full and continue
 * writing when some place is available again
 * Customer reads the queue periodically and displays what it receives
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date31
 */
#include <Arduino_FreeRTOS.h>
#include <queue.h>

/* The tasks to be created.  Two instances are created of the sender task while
only a single instance is created of the receiver task. */
static void vSenderTask( void *pvParameters );
static void vReceiverTask( void *pvParameters );
```

```
/*-----------------------------------------------------------*/

/* Declare a variable of type QueueHandle_t.  This is used to store the queue
that is accessed by all three tasks. */
QueueHandle_t xQueue;


// the setup function runs once when you press reset or power the board
void setup()
{

// initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

xQueue = xQueueCreate( 5, sizeof( uint32_t ) ); // 5 uint32_t elements

if( xQueue != NULL )// if queue object is created
  {
 /* Create two instances of the task that will write to the queue.  The
    parameter is used to pass the value that the task should write to the queue,
    so one task will continuously write 100 to the queue while the other task
    will continuously write 200 to the queue.  Both tasks are created at
    priority 1. */
    xTaskCreate( vProducerTask, "Producer", 200, NULL, 1, NULL );


    /* Create the task that will read from the queue.  The task is created with
    priority 2, so above the priority of the sender tasks. */
    xTaskCreate( vCustomerTask, "Consumer", 200, NULL, 2, NULL );


    //Start the scheduler so our tasks start executing.
   vTaskStartScheduler(); // not necessary in last implementation
                // present in the initVariant() function of the Free RTOS library
   }
  else
   Serial.println( "Could not create the queue." );
}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/
static void vProducerTask( void *pvParameters )
{

uint32_t lValueToSend = 0UL;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* The first parameter is the queue to which data is being sent.  The
    queue was created before the scheduler was started, so before this task
    started to execute.

    The second parameter is the address of the data to be sent.
```

```
    The third parameter is the Block time, the time the task should be kept
    in the Blocked state to wait for space to become available on the queue
    should the queue already be full.  In this case we will block until the
    customer frees some data */
    xQueueSendToBack( xQueue, &lValueToSend, portMAX_DELAY );

    lValueToSend++;

    // no delay here, we send until queue is full
  }
}
/*-----------------------------------------------------------*/

static void vCustomerTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
uint32_t lReceivedValue;
BaseType_t xStatus;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {

    /* The first parameter is the queue from which data is to be received.  The
    queue is created before the scheduler is started, and therefore before this
    task runs for the first time.

    The second parameter is the buffer into which the received data will be
    placed.  In this case the buffer is simply the address of a variable that
    has the required size to hold the received data.

    the last parameter is the block time, the maximum amount of time that the
    task should remain in the Blocked state to wait for data to be available
    should the queue already be empty. 0 means don't block at all */
    xStatus = xQueueReceive( xQueue, (uint32_t *)&lReceivedValue, 0 );

    if( xStatus == pdPASS )
    {
      /* Data was successfully received from the queue, print out the received
      value. */
      Serial.print( "Received = ");
      Serial.println( lReceivedValue );
    }
    else
    {
      /* We did not receive anything immediatly from the queue.
      Except for the first data, thThis must be an error as the sending task
      is free running and will be continuously writing to the queue. */
      Serial.println( "No data received" );
    }

    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}
```

Results :



Exercise for example 2.

Add testing code for the timeout.

3) Two producer tasks with priority 1 send data continuously (until they are blocked by a full queue condition) to a consumer task with priority 2. Consumer task reads queue each 500 ms with no timeout and detect the origin of the data.

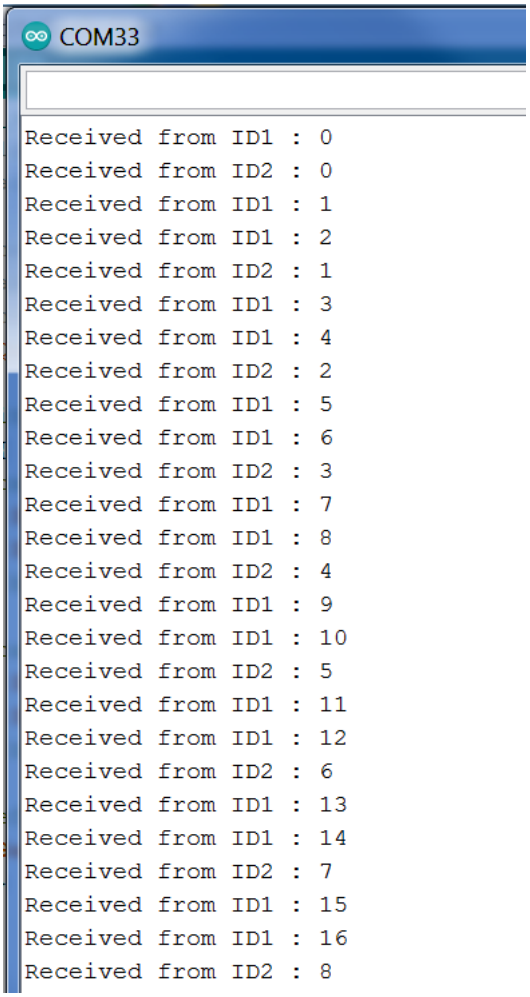
```
/**
 * @file ex10_queue_21P_1C.ino
 * @brief two producers (Sender) tasks with priority=1
 * and one customer (Receiver) task with priority=2
 * Producers writes periodically numeric data into queue
 * Customer wait until something is in the queue and displays it
 * indicating the origin
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 31/8/2021
 */

#include <Arduino_FreeRTOS.h>
#include <queue.h>

#define ID1 1
#define ID2 2

/* The tasks to be created.  Two instances are created of the sender task while
only a single instance is created of the receiver task. */
static void vSenderTask( void *pvParameters );
static void vReceiverTask( void *pvParameters );

struct data_for_cust
```

```
  {
  uint8_t id;
  uint16_t value;
  };

/* Declare a variable of type QueueHandle_t.  This is used to store
 a reference to the queue that is accessed by all three tasks. */
QueueHandle_t xQueue;

// the setup function runs once when you press reset or power the board
void setup()
{
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);

xQueue = xQueueCreate( 5, sizeof( data_for_cust ) ); // 5 data_for_cust elements
if( xQueue != NULL ) // if queue object is created
  {
   /* Create the tasks that will write to the queue with priority 1. */
   xTaskCreate( vProducerTask1, "Producer1", 200, NULL, 1, NULL );
   xTaskCreate( vProducerTask2, "Producer2", 200, NULL, 1, NULL );

   /* Create the task that will read from the queue.  The task is created with
   priority 2, so above the priority of the sender task. */
   xTaskCreate( vCustomerTask, "Consumer", 200, NULL, 2, NULL );

  //Start the scheduler so our tasks start executing.
  vTaskStartScheduler(); // not necessary in last implementation
                // present in the initVariant() function of the Free RTOS library
  }
  else
   Serial.println( "Could not create the queue." );
}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/
static void vProducerTask1( void *pvParameters )
{
data_for_cust data_to_send;
uint32_t delay1_send = 250 / portTICK_PERIOD_MS;

data_to_send.id = ID1;
data_to_send.value = 0;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* The first parameter is the queue to which data is being sent.  The
    queue was created before the scheduler was started, so before this task
    started to execute.

    The second parameter is the address of the data to be sent.

    The third parameter is the Block time, the time the task should be kept
```

```
    in the Blocked state to wait for space to become available on the queue
    should the queue already be full.  In this case we will block indefinitively
    until the customer frees some data */
   xQueueSendToBack( xQueue, &data_to_send, portMAX_DELAY );

   data_to_send.value++;

   vTaskDelay(delay1_send);
  }
}
/*-----------------------------------------------------------*/

static void vProducerTask2( void *pvParameters )
{
data_for_cust data_to_send;
uint32_t delay2_send = 500 / portTICK_PERIOD_MS;

data_to_send.id = ID2;
data_to_send.value = 0;

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* The first parameter is the queue to which data is being sent.  The
    queue was created before the scheduler was started, so before this task
    started to execute.

    The second parameter is the address of the data to be sent.

    The third parameter is the Block time, the time the task should be kept
    in the Blocked state to wait for space to become available on the queue
    should the queue already be full.  In this case we will block indefinitively
    until the customer frees some data */
   xQueueSendToBack( xQueue, &data_to_send, portMAX_DELAY );

   data_to_send.value++;

   vTaskDelay(delay2_send);
  }
}
/*-----------------------------------------------------------*/


static void vCustomerTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
data_for_cust data_received;


//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* The first parameter is the queue from which data is to be received.  The
    queue is created before the scheduler is started, and therefore before this
    task runs for the first time.

    The second parameter is the buffer into which the received data will be
    placed.  In this case the buffer is simply the address of a variable that
    has the required size to hold the received data.
```

```
    the last parameter is the block time, the maximum amount of time that the
    task should remain in the Blocked state to wait for data to be available
    should the queue already be empty. */
   xQueueReceive( xQueue, (data_for_cust*)&data_received,portMAX_DELAY);

   switch (data_received.id)
     {
     case ID1:
      Serial.print( "Received from ID1 : ");
      Serial.println( data_received.value);
      break;

     case ID2:
      Serial.print( "Received from ID2 : ");
      Serial.println( data_received.value);
      break;

    default:
     Serial.println( "Incorrect ID" );
     }
  }
}
```

Results :



Binary semaphores and mutexes.

Binary semaphores are used to synchronize tasks. Mutex semaphores are used to protect critical code sections related to the sharing of resources and to solve the reentrancy issues.

Mutexes include a priority elevation mechanism to avoid priority inversion problems.

Semaphores management functions.

All functions need the inclusion of **semphr.h**

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Creates a binary semaphore, and returns a handle by which the semaphore can be referenced. The semaphore is created in the 'empty' state, meaning the semaphore must first be given using the **xSemaphoreGive()** API function before it can subsequently be taken (obtained) using the **xSemaphoreTake()** function.

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Creates a mutex, and returns a handle by which the created mutex can be referenced. Mutexes cannot be used in interrupt service routines.

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Macro to release a semaphore.

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                TickType_t xTicksToWait );
```

Macro to obtain a semaphore.

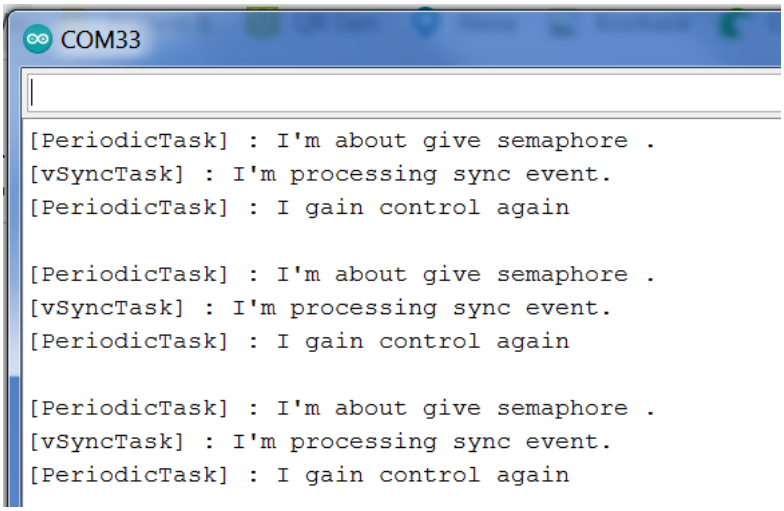
Example.

1) Use a binary semaphore to synchronize a task with another.

```
/**
 * @file ex11_sem_bin.ino
 * @brief use binary semaphore from a periodic task (vPeriodicTask)
 * to synchronize execution of another task (vSyncTask)
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 2/9/2021
 */
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

/* The tasks to be created. */
static void vSyncTask( void *pvParameters );
static void vPeriodicTask( void *pvParameters );

/*-----------------------------------------------------------*/
/* Declare a variable of type SemaphoreHandle_t.  This is used to reference the
semaphore that is used to synchronize a task with another task. */
SemaphoreHandle_t xBinarySemaphore;

// the setup function runs once when you press reset or power the board
void setup()
{
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

  /* Before a semaphore is used it must be explicitly created.
  In this example a binary semaphore is created. */
  xBinarySemaphore = xSemaphoreCreateBinary();

  /* Check the semaphore was created successfully. */
  if( xBinarySemaphore != NULL )
  {
    /* Create the sync task.  This is the task that will be synchronized
    by the periodic task.  The sync task is created with a high priority to
    ensure it runs immediately after the sync event.  In this case a
    priority of 3 is chosen. */
    xTaskCreate( vSyncTask, "Synchronized", 200, NULL, 3, NULL );

    /* Create the task that will periodically generate synchronization.
    This is created with a priority below the sync task to ensure it will
    get preempted each time the sync task exits the Blocked state. */
    xTaskCreate( vPeriodicTask, "Periodic", 200, NULL, 1, NULL );

  //Start the scheduler so our tasks start executing.
  vTaskStartScheduler(); // not necessary in last implementation
                  // present in the initVariant() function of the Free RTOS library
  }
}

void loop()
{
  // Empty. Things are done in Tasks.
}
```

```
/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/

static void vSyncTask( void *pvParameters )
{
 //As always in embedded systems, this task is implemented in an infinite loop.
 while(1)
  {
    /* Use the semaphore to wait for the event.  The semaphore was created
    before the scheduler was started so before this task ran for the first
    time.  The task blocks indefinitely meaning this function call will only
    return once the semaphore has been successfully obtained - so there is no
    need to check the returned value. */
    xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

    /* To get here the sync event must have occurred.  Process the event (in this
    case we just print out a message). */
    Serial.println( "[vSyncTask] : I'm processing sync event." );
  }
}
/*-----------------------------------------------------------*/

static void vPeriodicTask( void *pvParameters )
{

//As always in embedded systems, this task is implemented in an infinite loop.
while(1)
  {
    /* This task unblock the SyncTask periodically */
    vTaskDelay( 1000 / portTICK_PERIOD_MS );

    /* Generate the synchronization, printing a message both before hand and
    afterwards so the sequence of execution is evident from the output. */
    Serial.println( "[PeriodicTask] : I'm about give semaphore ." );
    xSemaphoreGive( xBinarySemaphore);
    Serial.println( "[PeriodicTask] : I gain control again" );
    Serial.println("");
  }
}
```

Results :



2) Use a mutex to share a resource safely. Send data to the console from within two tasks with the different priorities ans with random delays.

```
/**
 * @file ex12_saveprint_mutex.ino
 * @brief safe Serial.print using a mutex - solving rentrancy issue
 * safe print is used by two tasks with the same code.
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 2/9/2021
 */
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

/* The task to be created.  Two instances of this task are created. */
static void prvPrintTask( void *pvParameters );

/* The function that uses a mutex to control access to standard out. */
static void prvNewPrintString( const char *pcString );

/*-----------------------------------------------------------*/

/* Declare a variable of type SemaphoreHandle_t.  This is used to reference the
mutex type semaphore that is used to ensure mutual exclusive access to stdout. */
SemaphoreHandle_t xMutex;


// the setup function runs once when you press reset or power the board
void setup() {

  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

   /* Before a semaphore is used it must be explicitly created.  In this example
  a mutex type semaphore is created. */
    xMutex = xSemaphoreCreateMutex();

  /* The tasks are going to use a pseudo random delay,
   *  the generator is seeded here. */
  srand( 567 );

  /* Check if the semaphore was created successfully. */
  if( xMutex != NULL )
  {
    /* Create two instances of the tasks that attempt to write stdout.  The
    string they attempt to write is passed in as the task parameter.  The tasks
    are created at different priorities so some pre-emption will occur. */
    xTaskCreate( prvPrintTask, "Print1", 200, (void*)"Task 1 *****", 1, NULL );
    xTaskCreate( prvPrintTask, "Print2", 200, (void*)"Task 2 -----", 2, NULL );
   }

//Start the scheduler so our tasks start executing.
  vTaskStartScheduler(); // not necessary in last implementation
                // present in the initVariant() function of the Free RTOS library

}
```

```
void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/


/*-----------------------------------------------------------*/

static void prvNewPrintString( const char *pcString ) // this is a function, not a
task.
{
  /* The semaphore is created before the scheduler is started so already
  exists by the time this task executes.

  Attempt to take the semaphore, blocking indefinitely if the mutex is not
  available immediately.  The call to xSemaphoreTake() will only return when
  the semaphore has been successfully obtained so there is no need to check the
  return value.  If any other delay period was used then the code must check
  that xSemaphoreTake() returns pdTRUE before accessing the resource (in this
  case standard out. */
  xSemaphoreTake( xMutex, portMAX_DELAY );
  {
  /* The following line will only execute once the semaphore has been
  obtained, so standard output can be accessed without any problems */

    Serial.println(pcString);
    Serial.flush();
  }
  xSemaphoreGive( xMutex );

 }

/*-----------------------------------------------------------*/

static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;

  /* Two instances of this task are created so the string the task will send
  to prvNewPrintString() is passed in the task parameter.  Cast this to the
  required type. */
  pcStringToPrint = ( char * ) pvParameters;

while(1)
  {
    /* Print out the string using the newly defined function. */
    prvNewPrintString( pcStringToPrint );

    /* Wait a pseudo random time.  Note that rand() is not necessarily
    re-entrant, but in this case it does not really matter as the code does
    not care what value is returned.  In a more secure application a version
    of rand() that is known to be re-entrant should be used - or calls to
    rand() should be protected using a critical section. */
    vTaskDelay( ( rand() & 0x1FF ) );
  }
}
```

Results :

Limitations of the mutex method.

As the printing task inherits from the mutex a high priority (to avoid priority inversion) it will retain the CPU until the printing task is ended.

If this task takes a long time, the whole system is blocked by the printing task. Such a task should receive a low priority, but it this case we are back to the priority inversion problem.

When we use a mutex the rule is to keep the protected code **as short as possible**.

With the gatekeeper, we will be able to use a low priority task to control the shared resource.

Resource sharing with a gatekeeper.

A gatekeeper is a task dedicated to the resource to be shared. That's the only task which can access the resource, thus we won't have conflicts. Communication with the gatekeeper is made via a queue.

Example.

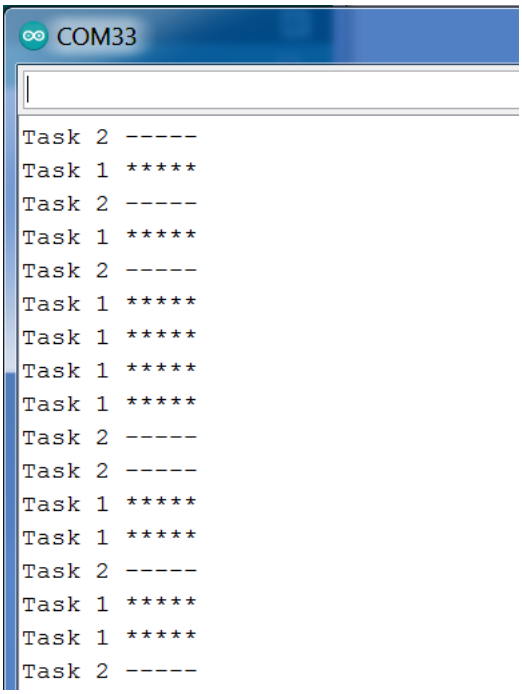
Use a gatekeeper to share a resource safely. Send data to the console from within two tasks with the different priorities and with random delays.

```
/**
 * @file ex14_saveprint_gatekeeper.ino
 * @brief safe Serial.print using a gatekeeper task - solving rentrancy issue
 * Only the gate keeper task is allowed to access shared ressource.
 * Gate keeper task is a background task, so its priority is low.
 * A queue is used to receive data from two other tasks.
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 4/11/2019
 */
#include <Arduino_FreeRTOS.h>
#include <queue.h>

/* The task that sends messages to the stdio gatekeeper.  Two instances of this
task will be created. */
static void prvPrintTask( void *pvParameters );

/* The gatekeeper task. */
static void prvStdioGatekeeperTask( void *pvParameters );

/*-----------------------------------------------------------*/

/* Declare a variable of type QueueHandle_t.  This is used to send messages from
the print tasks to the gatekeeper task. */
QueueHandle_t xPrintQueue;


// the setup function runs once when you press reset or power the board
void setup() {

  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

     /* Before a queue is used it must be explicitly created.  The queue is
       created  to hold a maximum of 5 character pointers. */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

  /* The tasks are going to use a pseudo random delay,
   *  the generator is seeded here. */
  srand( 678 );

  /* Check the queue was created successfully. */
  if( xPrintQueue != NULL )
```

```
  {
  /* Create two instances of the tasks that send messages to the gatekeeper.
  The string they attempt to write is passed in as the task parameter.
  The tasks are created at different priorities so some pre-emption will occur. */
    xTaskCreate( prvPrintTask, "Print1", 200, (void*)"Task 1 *****", 1, NULL );
    xTaskCreate( prvPrintTask, "Print2", 200, (void*)"Task 2 -----", 2, NULL );

    /* Create the gatekeeper task.  This is the only task that is permitted
    to access standard out. Its priority i slow to allow other tasks to run*/
    xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 200, NULL, 0, NULL );
  }


  //Start the scheduler so our tasks start executing.
  vTaskStartScheduler(); // not necessary in last implementation
                 // present in the initVariant() function of the Free RTOS library
}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ----------------------*/
/*-----------------------------------------------------*/

static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;

  /* This is the only task that is allowed to write to the terminal output.
  Any other task wanting to write to the output does not access the terminal
  directly, but instead sends the output to this task.  As only one task
  writes to standard out there are no mutual exclusion or serialization issues
  to consider within this task itself. */
while(1)
  {
    /* Wait for a message to arrive. */
    xQueueReceive( xPrintQueue, (char *) &pcMessageToPrint, portMAX_DELAY );

    /* There is no need to check the return value as the task will block
    indefinitely and only run again when a message has arrived.  When the
    next line is executed there will be a message to be output. */
    Serial.println(pcMessageToPrint );
    Serial.flush();
  /* Now simply go back to wait for the next message. */
  }
}
/*-----------------------------------------------------------*/

static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;

/* Two instances of this task are created so the string the task will send
  to prvNewPrintString() is passed in the task parameter.  Cast this to the
  required type. */
pcStringToPrint = ( char * ) pvParameters;
```

```
while(1)
  {
    /* Print out the string, not directly but by passing the string to the
    gatekeeper task on the queue.  The queue is created before the scheduler is
    started so will already exist by the time this task executes.  A block time
    is not specified as there should always be space in the queue. */
    xQueueSendToBack( xPrintQueue, (void *) &pcStringToPrint, 0 );

    /* Wait a pseudo random time.  Note that rand() is not necessarily
    re-entrant, but in this case it does not really matter as the code does
    not care what value is returned.  In a more secure application a version
    of rand() that is known to be re-entrant should be used - or calls to
    rand() should be protected using a critical section. */
    vTaskDelay( ( rand() & 0x1FF ) );
  }
}
```

Results:



Interrupt management.

If data arrival is notified by an interrupt, as interrupt routine must be as short as possible, the best method is using the interrupt routine only to send the data through a queue to a dedicated task which will process the data.

Queue functions called from interrupt.

```
BaseType_t xQueueSendToBackFromISR
                    (
                        QueueHandle_t xQueue,
                        const void *pvItemToQueue,
                        BaseType_t *pxHigherPriorityTaskWoken
                    );
```

Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine.

`xQueueSendTobackFromISR()` will set `*pxHigherPriorityTaskWoken` to `pdTRUE` if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If `xQueueSendToBackFromISR()` sets this value to `pdTRUE` then a context switch should be requested before the interrupt is exited otherwise the context switch will take place after the next tick.

`vPortYield();`
request a context switch from an interrupt routine.

Example.

We will create a task (`vInterruptGenerator()`) to simulate an external interrupt with a software interrupt to pin 3 of the Arduino Mega. The interrupt routine (`vExampleInterruptHandler()`) attach to this interruption will send data through a queue to the task (`vStringPrinter()`)

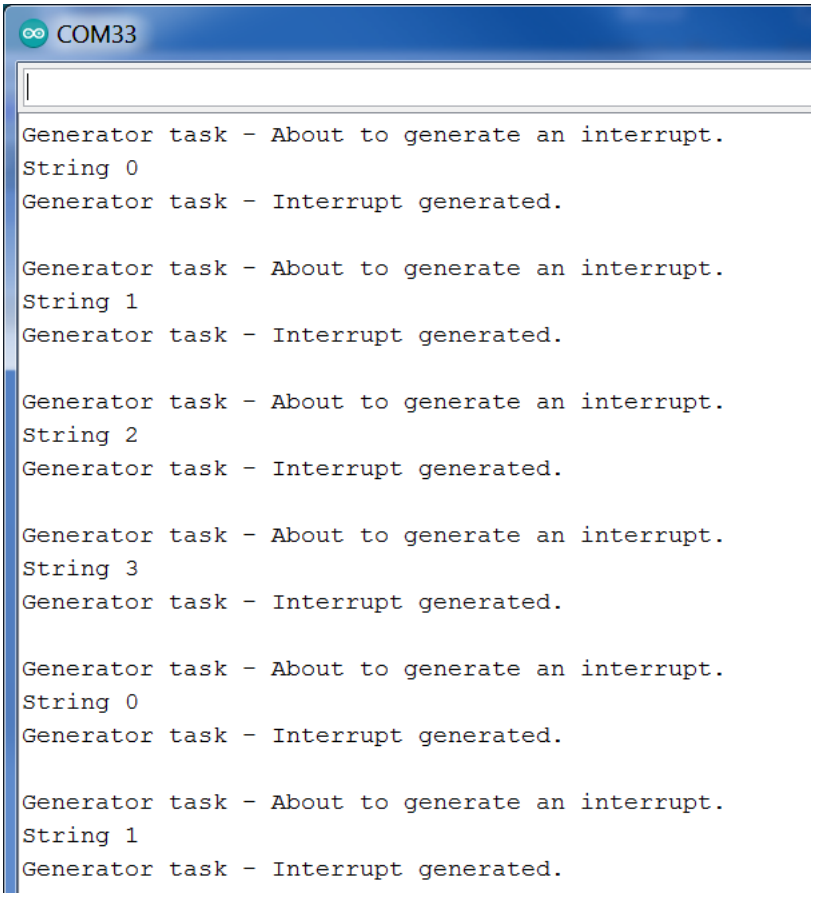
```
/**
 * @file ex13_queue_int.ino
 * @brief use a queue from interrupt to give data to a task and to awake it
 * Use a software interrupt (pin 3 of CPU).
 * This interrupt will be raised by a periodic task
 * FreeRTOS API reference : https://www.freertos.org/a00106.html
 * @author philippe.camus@hepl.be from examples suite of FreeRTOS
 * @date 2/11/2019
 */
#include <Arduino_FreeRTOS.h>
#include <queue.h>
```

```
/* The tasks to be created. */
static void vInterruptGenerator( void *pvParameters );
static void vStringPrinter( void *pvParameters );

/*-----------------------------------------------------------*/

/* The service routine for the interrupt.  This is the interrupt that the task
will be synchronized with. */
static void vExampleInterruptHandler( void );

/* Declare two variables of type QueueHandle_t.  One queue will be read from
within an ISR, the other will be written to from within an ISR. */
QueueHandle_t xStringQueue;

// pin to generate interrupts
const uint8_t interruptPin = 3;

// the setup function runs once when you press reset or power the board
void setup()
{
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

/* Install the interrupt handler. */
  pinMode(interruptPin, OUTPUT);
  // pin 3 interruption rising edge
  attachInterrupt(digitalPinToInterrupt(interruptPin),
                  vExampleInterruptHandler, RISING);

/* Before a queue is used it must be explicitly created.  The queue is created
  to hold a maximum of 5 character pointers. */
  xStringQueue = xQueueCreate( 5, sizeof( char * ) );

  /* Create the task that simulates the external interrupt.
  The task is created at priority 1. */
  xTaskCreate( vInterruptGenerator, "IntGen", 200, NULL, 1, NULL );

  /* Create the task that prints out the strings sent to it from the interrupt
  service routine.  This task is created at the higher priority of 2. */
  xTaskCreate( vStringPrinter, "String", 200, NULL, 3, NULL );

 //Start the scheduler so our tasks start executing.
  vTaskStartScheduler(); // not necessary in last implementation
                 // present in the initVariant() function of the Free RTOS library

}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*-----------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*-----------------------------------------------------*/
static void vInterruptGenerator( void *pvParameters )
{
TickType_t xLastExecutionTime;

  /* Initialize the variable used by the call to vTaskDelayUntil(). */
```

```
  xLastExecutionTime = xTaskGetTickCount();

while(1)
  {
    /* This is a periodic task.  Block until it is time to run again.
    The task will execute every 200ms. */
    vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_PERIOD_MS );

    /* Send an incrementing number to the queue five times.  These will be
    read from the queue by the interrupt service routine.  A block time is
    not specified. */

    /* Force an interrupt so the interrupt service routine can read the
    values from the queue. */
    Serial.println( "Generator task - About to generate an interrupt." );

    digitalWrite(interruptPin, LOW);
    digitalWrite(interruptPin, HIGH);// rising edge transition creates interrupt

    Serial.println( "Generator task - Interrupt generated." );
    Serial.println( " ");
  }
}
/*-----------------------------------------------------------*/

static void vStringPrinter( void *pvParameters )
{
char *pcString;

while(1)
  {
    /* Block on the queue to wait for data to arrive. */
    xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

    /* Print out the string received. */
    Serial.println( pcString );
  }
}
/*-----------------------------------------------------------*/

static void vExampleInterruptHandler( void )
{
static BaseType_t xHigherPriorityTaskWoken;
static uint8_t select=0;

/* The strings are declared static const to ensure they are not allocated to the
interrupt service routine stack, and exist even when the interrupt service routine
is not executing. */
static const char *pcStrings[] =
{
  "String 0",
  "String 1",
  "String 2",
  "String 3"
};

  xHigherPriorityTaskWoken = pdFALSE;

  xQueueSendToBackFromISR( xStringQueue, &pcStrings[ select ],
(BaseType_t*)&xHigherPriorityTaskWoken );
```

```
  select++;
  if (select>=4) select=0;

  /* Did receiving on a queue or sending on a queue unblock a task that has a
  priority higher than the currently executing task?
  If so, force a context switch here. */
  if( xHigherPriorityTaskWoken == pdTRUE )
  {
    /* NOTE: The syntax for forcing a context switch is different depending
    on the port being used.  Refer to the examples for the port you are
    using for the correct method to use! */
    vPortYield();
  }
}
```

Results:



COROUTINES IN MICROPYTHON : A PRIMER.

Coroutines are the collaborative way to do multitasking.

Each task is defined as a special kind of function called an asynchronous function. It's the responsibility of each task to give back control to the scheduler. It's done by a "`await`" instruction.

For instance, the `await uasyncio.sleep_ms(1000)` instruction gives the control to the scheduler and remains blocked for at least 1000 ms.

To start the multitasking process, you must first set up an event loop with the `uasyncio.get_event_loop()` instruction.

Whereupon you can add your tasks to the loop object with the `create_task` method. And then you can start the multitasking with the `run_forever` method.

There are no priorities for the tasks, each task receives the processor in a round robin way as soon as another task yield control to the scheduler with an await instruction (or an explicit yield).

```
# file therm_coro_v2.py
# Thermostat with coroutines for Raspberry Pi Pico
# queue.py must be installed in the raspberry flash memory
# https://github.com/peterhinch/micropython-async/blob/master/v3/primitives/queue.py
# author philippe.camus@hepl.be
# date 12/11/2021

import time
from therm_loop_util_v1 import *
import uasyncio
import queue

q=queue.Queue()# create queue

ds_sensor, roms, pot, heater_led, ldr,lcd, bckl = hardware_setup()


async def temp():
    while True:
        temperature = read_temperature(ds_sensor, roms)
        display_temperature(temperature, lcd)
        await q.put(["T",temperature]) # pass a list with source identifier "T"
        await uasyncio.sleep_ms(1000)

async def setp():
    while True:
        setpoint = read_setpoint(pot)
        display_setpoint(setpoint, lcd)
```

```
        await q.put(["S",setpoint]) # pass a list with source identifier "S"
        await uasyncio.sleep_ms(250)

async def heater():
    temperature = 20
    setpoint=25

    while True:
        message=await q.get()
        if message[0]=="T":
            temperature = message[1]
        else:
            setpoint = message[1]

        if (temperature < setpoint):
            set_heater(1,heater_led)
        else:
            set_heater(0,heater_led)

async def lum():
    while True:
        luminosity = read_luminosity(ldr)
        display_luminosity(luminosity, bckl)
        await uasyncio.sleep_ms(500)

loop = uasyncio.get_event_loop()
loop.create_task(temp())
loop.create_task(setp())
loop.create_task(heater())
loop.create_task(lum())
loop.run_forever()
```

Queues are not yet implemented "officialy" in MicroPython, the queue.py module used here is based on Paul Sokolovsky's work implemented by Peter Hinch. *https://github.com/peterhinch/micropython-async/blob/master/v3/primitives/queue.py*

The queue is implemented as a Python list. By default it's an unlimited queue. To limit it you must provide a maxsize parameter, for instance

```
q=queue.Queue(maxsize=5)# create queue of 5 elements
```

It should be noted then the await give back control to the calling task if no blocking condition occurs (for instance a blocking condition could be an empty queue on `get` method).

The `therm_loop_util_v1` module is the same as the one used in the loop architecture (p 38).

MULTITASKING THE THERMOSTAT PROJECT.

This last exercise is a synthesis exercise. It consists of designing and implementing a thermostat using the multitasking paradigm.

The hardware platform will be the Arduino Mega and you will use five tasks.

Three for data acquisition:

- Temperature reading (DS18B20).
- Setpoint reading (potentiometer).
- Luminosity reading (LDR).

One for the control of the LCD (I^2C 2 x 16):

- Temperature and setpoint value display and backlight control.

One for the heater control:

- Switching on or off the LED (symbolizing the heater).

You will use two queues: one to exchange data between the temperature and setpoint reading task and the heater task (two producers and one customer) and one to exchange data between the temperature, setpoint and luminosity reading tasks and the LCD task (three producers and one customer).

C++ classes will be designed for the potentiometer and LDR.

BIBLIOGRAPHY.

Foreword to the bibliography.

This bibliography does not contain many entries as the author does most of the examples and figures himself.

An engineering textbook differs from other textbooks.

If it is to be compared with a history textbook, this one is a compilation of sources gathered together by the author and should therefore be cited in detail.

Whereas, for a programming engineering textbook, it is completely the opposite. Each element is built from scratch and examples are devised, encoded and thoroughly checked. The classical method is to search reference materials and to spend endless hours on the websites.

References.

Python:

- Al. Sweigart, invent Your Own Computer Games with Python, No starch press, ISBN 978-1593277956.

- Eric Matthes, Python Crash Course, No starch press, ISBN 978-1593279288

Raspberry Pi Pico: Dogan Ibrahim, Raspberry Pi Pico essential, Elektor press, ISBN 978-3895764271

Arduino site: https://www.arduino.cc/

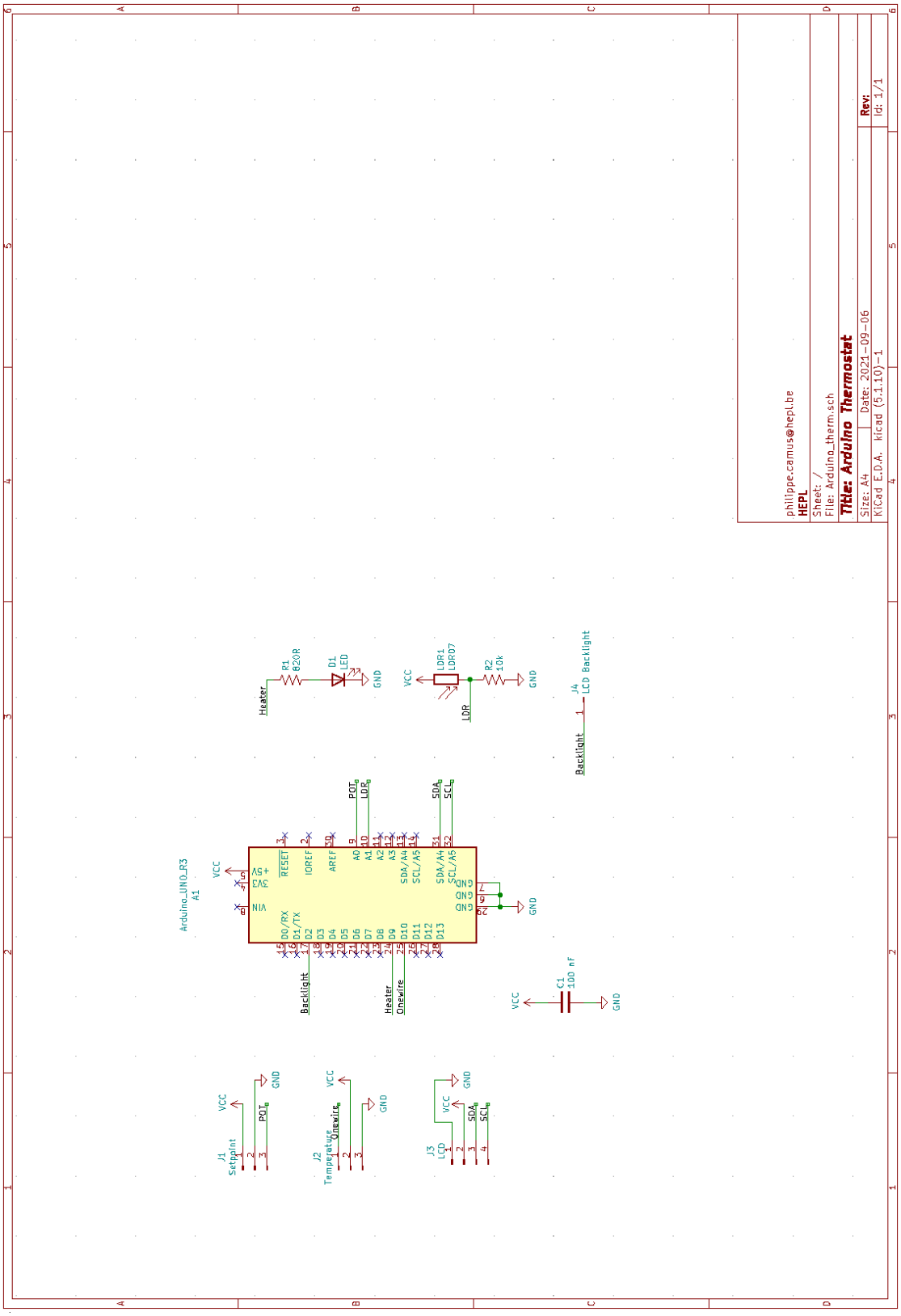
Raspberry Pi Pico site: https://www.raspberrypi.org/products/raspberry-pi-pico/

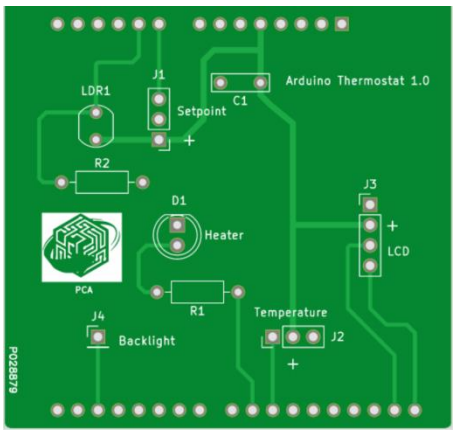
MicroPython: https://micropython.org/

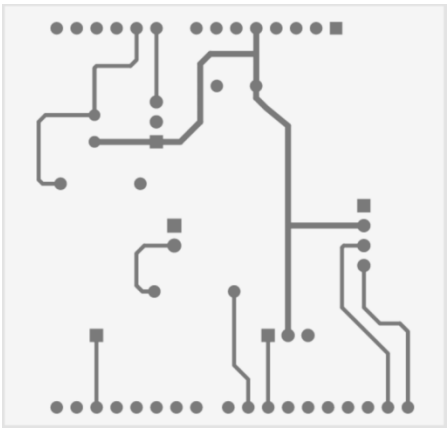
For interesting tutorials about FreeRTOS, RPi Pico (and more) watch Shawn Hymel channel on YouTube: https://www.youtube.com/c/ShawnHymel

All other references are within the text.

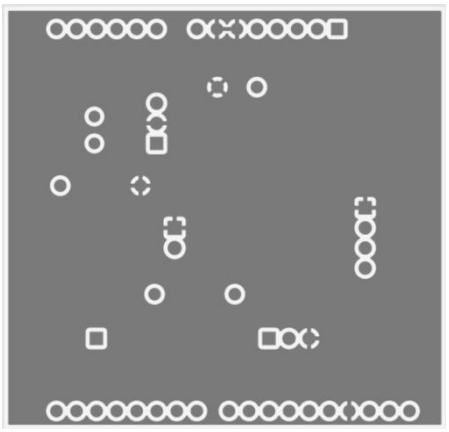
Arduino Mega thermostat shield.



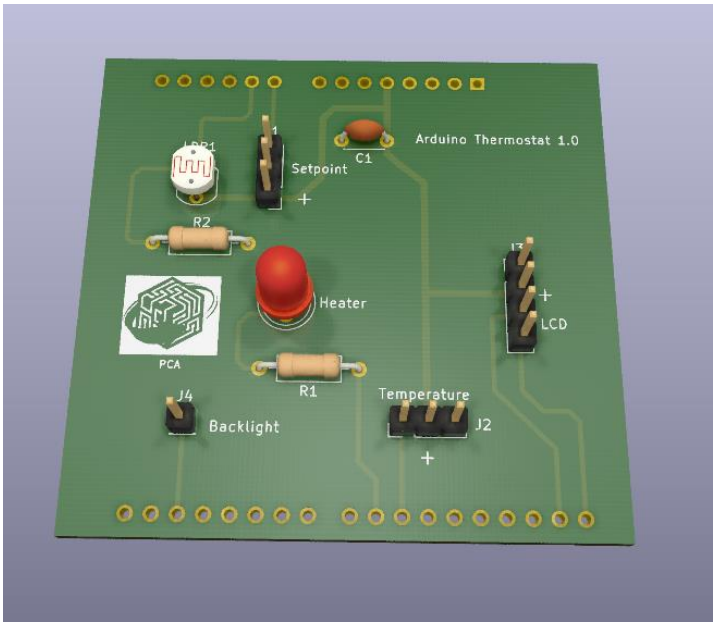




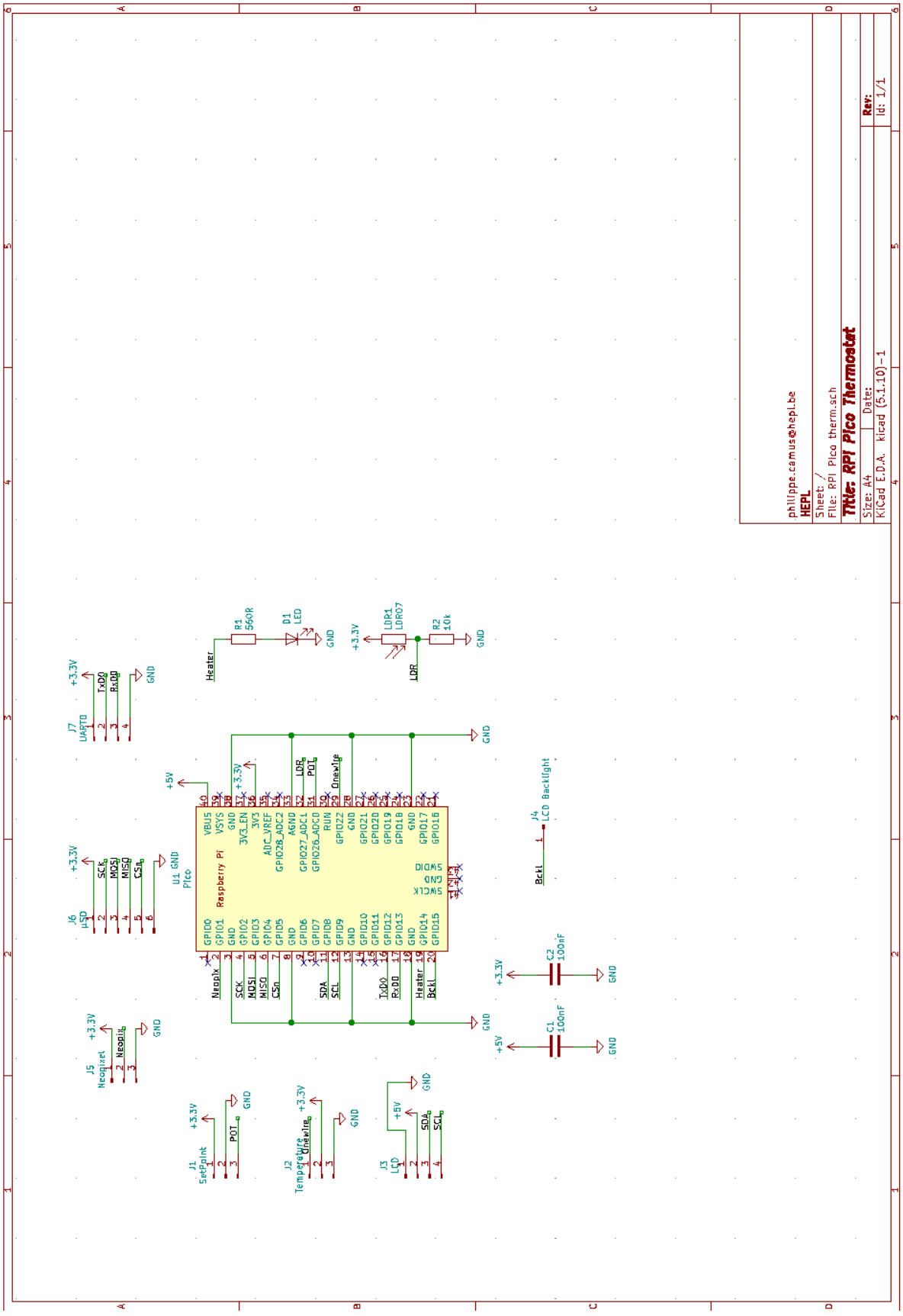
Top copper

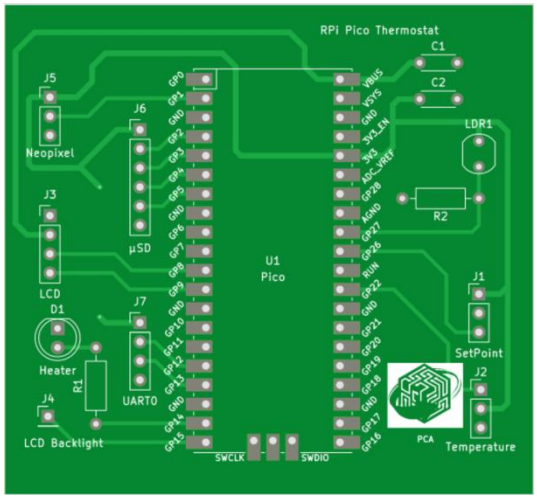


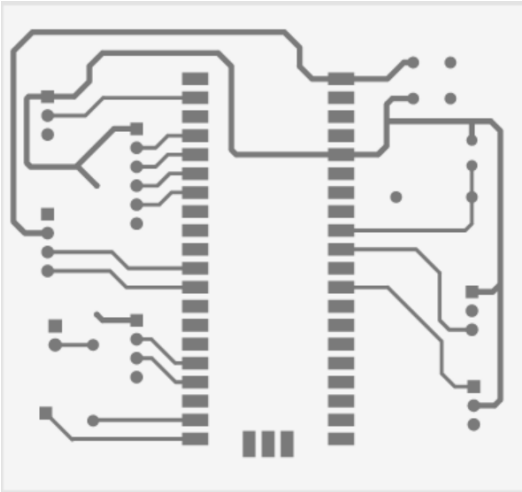
Bottom copper.



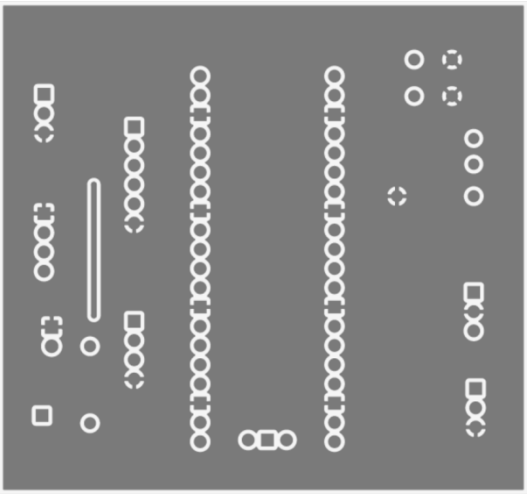
Raspberry Pi Pico thermostat board.







Top copper



Bottom copper