Università
della
Svizzera
italiana

Facoltà
di
scienze
informatiche

Edge Computing in the IoT

RTOS: Real Time Operating System

Alberto Ferrante
TA: Luca Butera

Threads

```
#include "mbed.h"
using namespace mbed;
using namespace rtos;

Thread thread; // or thread(osPriorityNormal); if you want to explicitly set the priority
DigitalOut led(LED1);

void led_on() {
    while(true) {
        led.write(1);
        ThisThread::sleep_for(100);
    }
}

void setup() {
    thread.start(led_on);
}
void loop() {
        led.write(0);
        ThisThread::sleep_for(100);
}
```

Threads are instantiated with normal priority by default.

The Mbed scheduler uses a **round robin** approach.

"sleep_for" is a non-locking wait. Without it the thread never frees the processor.

Threads - callbacks

```
#include “mbed.h”
using namespace mbed;
using namespace rtos;

Thread thread; // or thread(osPriorityNormal); if you want to explicitly set the priority
DigitalOut led(LED1);

void set_led(int value) {
     while(true) {
          led.write(value);
          ThisThread::sleep_for(100);
     }
}

void setup() {
     thread.start(callback(set_led, 1));
}
void loop() {
     led.write(0);
     ThisThread::sleep_for(100);
}
```

callback allows to pass parameters to the thread function.

Only one parameter can be passed, so if you need more create a class.

Threads also support **joins** if you need to wait for a thread to end.

Semaphores

```
Thread thread;
DigitalOut led(LED1);
Semaphore s1(1, 1);
Semaphore s2(0, 1);

void led_on() {
    while(true) {
        s1.acquire();
        led.write(1);
        s2.release();
    }
}

void setup() {
    thread.start(led_on);
}
void loop() {
    s2.acquire();
    led.write(0);
    s1.release();
}
```

Semaphores are declared with a number of associated resources that each thread can acquire and release in order to control critical executions.

Interrupts

```
DigitalOut led(LED1);

void flip() {
    led = !led;
}

void setup() {
    attachInterrupt(digitalPinToInterrupt(D6), &flip, PinStatus::RISING);
}
void loop() {
    ThisThread::sleep_for(250);
}
```

Interrupts allow for the execution of functions right away.

They block the code that is being executed.

Tickers can be used to schedule time dependant interrupts:
https://os.mbed.com/docs/mbed-os/v6.15/apis/ticker.html

Events

Events are used to schedule function calls into and **EventQueue**, where those can be dispatched to handlers.

This is useful as during Interrupts we don't want to execute blocking or long running code, hence we can run the critical part of an interrupt inside its context and then schedule the non critical part to be executed later on.

Here is a guide on how to use EventQueues**:**
https://os.mbed.com/blog/entry/Simplify-your-code-with-mbed-events/

Here find the Mbed EventQueue API:
https://os.mbed.com/docs/mbed-os/v6.15/apis/eventqueue.html

Queues

```
Thread thread;
DigitalOut led(LED1);
Queue<bool, 32> queue; //type and size

void send_message() {
    while(true) {
        bool value = !led.read();
        queue.put(&value);
        ThisThread::sleep_for(1000);
    }
}
void setup() {
    thread.start(send_message);
}
void loop() {
    while(true) {
        osEvent evt = queue.get();
        if (evt.status == osEventMessage) {
            led.write(*((bool*) evt.value));
        }
        ThisThread::sleep_for(1000);
    }
}
```

Queues can be used to send messages between threads.

This allows, for instance, to separate data collection and processing between different threads.

Multi-core

```
int myLED;

void setup() {
      #ifdef CORE_CM7
            bootM4();
            myLED = LEDB;
      #endif
      #ifdef CORE_CM4
            myLED = LEDG;
      #endif
      pinMode(myLED, OUTPUT);
}
void loop() {
      digitalWrite(myLED, LOW);
      delay(200);
      digitalWrite(myLED, HIGH);
      delay(rand() % 2000 + 1000);
}
```

We program **both cores** in the same sketch using **conditional compilation macros**.

For more complex programs is preferable to **write two sketches**, one for each core.

Multi-core communication

We can make the M7 and M4 cores of the Portenta **communicate** as we did with threads.

To do so we use **Remote Procedure Calls (RPCs)**, which are supported in Arduino through the **RPC.h library**.
Mind that **only the M7 core can expose RPCs.**

Open your Arduino IDE and go to
Files -> Examples -> RPC -> RPC_m4

Excercise 1

Write an Arduino program that uses **threads**, **interrupts**, **semaphores** and **queues** to do the following:

- Wait for the Pulse Oximeter sensor for a signal that new data is present.
- Read the data and put them into a queue.
- Dequeue new data and plot them to serial.

Use different threads to handle reading and plotting the data.

Excercise 1 - Libraries

Use the SparkFun library at:
https://github.com/sparkfun/SparkFun_MAX3010x_Sensor_Library/blob/master/examples/Example7_Basic_Readings_Interrupts/Example7_Basic_Readings_Interrupts.ino
as the DFRobot library does not support interrupts.

You can directly install it from the Arduino IDE Library Manager as **SparkFun_MAX3010x.**

The library provides an example of interrupt handling under
File -> Examples -> SparkFun MAX3010x Pulse and Proximity Sensor Library -> Example7_Basic_Readings_Interrupts
This example, however, does not use threads nor Mbed OS functionalities.

On iCorsi you will find a minimal working example on how to read the sensor when you get an interrupt, using what we saw today.

Excercise 2

Program the M4 and M7 cores to do the following:

- M4 waits for the Pulse Oximeter sensor for a signal that new data is present.
- M4 reads the new data.
- M4 uses RPC to make M7 print the data to serial.
- M7 prints to serial the amount of time passed from the last received data.