UNIVERSIDAD DE OVIEDO

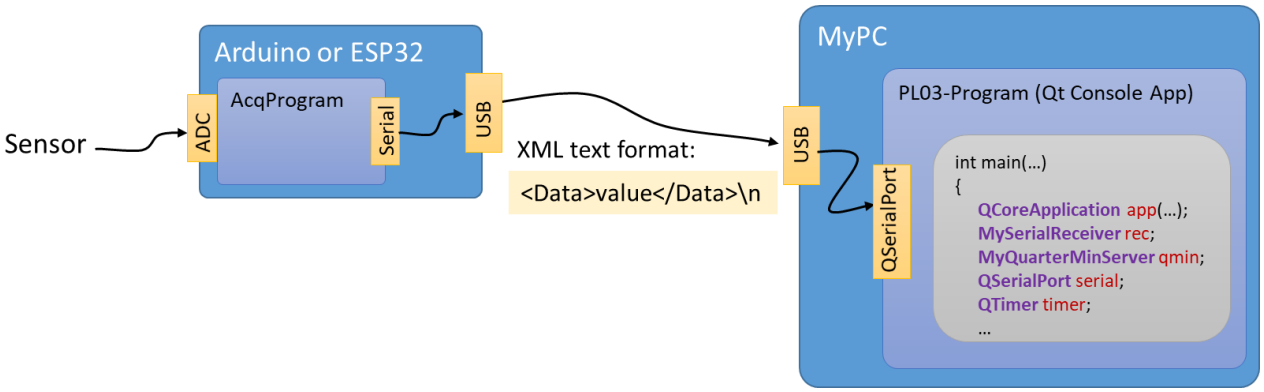
Departamento de Ingeniería Eléctrica,
Electrónica, de Computadores y Sistemas

# Guía de Prácticas

| | | | |
|---|---|---|---|
| ASIGNATURA: | Implementación de Sistemas de Control | | |
| CENTRO: | Centro Internacional de Postgrado | | |
| ESTUDIOS: | Master en Ingeniería Mecatrónica | | |
| CURSO: | 2º | CUATRIMESTRE: | 1 |
| CARÁCTER: | Obligatoria | CRÉDITOS ECTS: | 6 |
| PROFESORADO: | Ignacio Alvarez, Fernando Briz | | |

PRACTICA 03: Programación orientada a eventos con Qt-SDK y C++

- Realizar un programa orientado a eventos que atienda a dos fuentes de eventos:
  - Un temporizador: cada segundo, el servidor del evento obtiene la hora actual y comprueba si estamos en un cuarto de minuto (el segundo es 00, 15, 30 ó 45). En caso de cambio (transiciones NO/SI y SI/NO), envía una señal indicando este hecho.
  - La entrada de datos por un puerto serie desde un equipo de adquisición: cuando se reciba un texto en formato xml, se escribe en pantalla y en un archivo dicho dato, o el doble del mismo si está activado el cuarto de minuto.




El resultado esperado es:

Console output

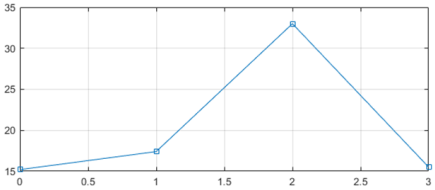
```
App started, receiving from COM4
11:02:13 --> Data received 15.2 (x1)
11:02:14 --> Data received 17.4 (x1)
11:02:15 --> Quarter Min Started
11:02:15 --> Data received 16.5 (x2)
11:02:16 --> Quarter Min Finished
11:02:16 --> Data received 15.5 (x1)
...
```

File output (results.txt)

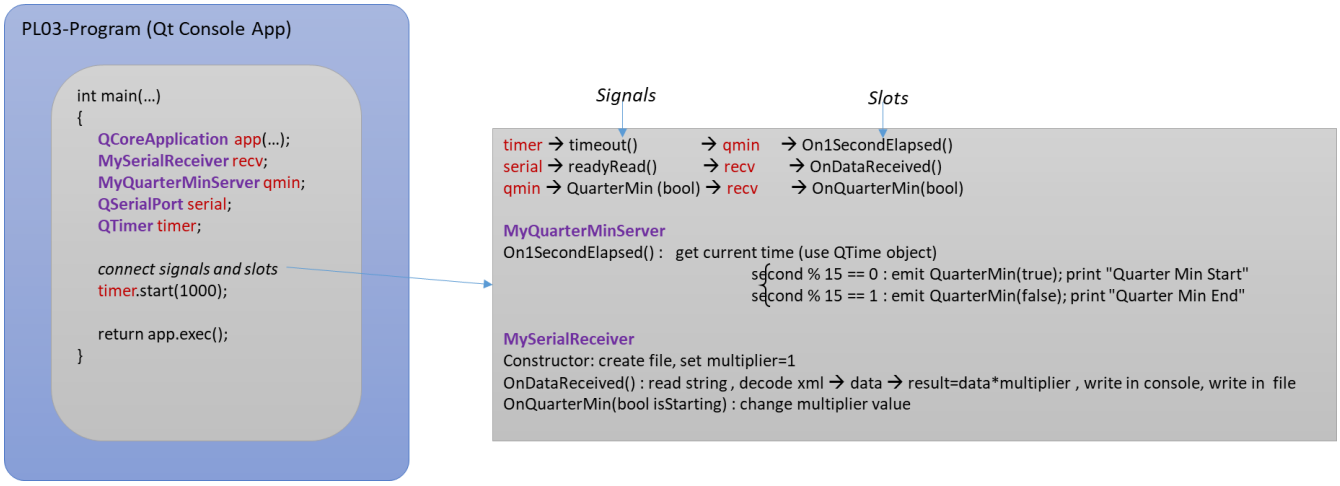
```
0 , 15.2
1 , 17.4
2 , 33.0
3 , 15.5
...
```

Matlab

```
>> data=importdata('results.txt');
>> figure; plot(data(:,1),data(:,2),'-s');
```



La organización de clases, señales y slots será la siguiente:




**Puerto serie:** Para la lectura de datos del puerto serie se utilizará QSerialPort (necesario añadir línea **QT += serialport** en archivo .pro).

main.cpp

```
...
int main(...)
{
   ...
   QSerialPort serial;
   ...
    // Initialize serial port
    serial.setPortName("COM4");  // Cambiar según puerto serie disponible
    serial.setBaudRate(115200);   // Debe ser igual al programado en el Arduino/ESP32
    serial.setDataBits(QSerialPort::Data8); // Debe ser igual al programado en el Arduino/ESP32
    serial.setParity(QSerialPort::NoParity); // Debe ser igual al programado en el Arduino/ESP32
    serial.setStopBits(QSerialPort::OneStop); // Debe ser igual al programado en el Arduino/ESP32

    if (serial.open(QIODevice::ReadWrite))     {
        connect(&serial , SIGNAL(readyRead()), &recv , SLOT(OnDataReceived()));
    }
    ...
}
```

## MySerialReceiver

MySerialReceiver.h

```
...
class MySerialReceiver : public QObject
// Derivar de QObject para usar signals/slots
{
   Q_OBJECT
   ...
private:
   QFile output;
   float multiplier=1;  // Si se inicializa aquí,
                        // no es necesario hacer
                        // en el constructor
public:
   MySerialReceiver (QObject* parent=NULL);
public slots:
   ...
   void OnDataReceived();
   void OnQuarterMin(bool isStarting);
} ;
```

MySerialReceiver.cpp

```
#include " MySerialReceiver.h"
...
MySerialReceiver::MySerialReceiver (QObject* parent) :
                                        QObject(parent) {
    multiplier=1; // No es necesario si se ha
                  // inicializado en la declaración
    output.open("results.txt",QIODevice::WriteOnly);
}

void MySerialReceiver::OnDataReceived() {
    QSerialPort* from=(QSerialPort*) sender();
    // sender(): función de QObject, devuelve QObject*

    QString data= from->readAll();
    QTime now=QTime::currentTime();

    Decode data , calc with multiplier,
    write to console and to output
}

void MySerialReceiver:: OnQuarterMin(bool isStarting)
{
    multiplier = ...;
}
```

**MyQuarterMinServer** → realizar con una de las 2 opciones:

- **QTimer** timer se declara, se conecta a su slot y se arranca en main( )
- **QTimer** timer está en MyQuarterMinServer ; se arranca en el constructor de MyQuarterMinServer y se conecta desde main( ) (se requiere una nueva señal conectada internamente a timer.timeout( ) )

### Generador de datos

La generación de datos para el puerto serie se puede realizar mediante un Arduino (o equivalente con un sensor analógico), con un programa similar al ejemplo siguiente (para uso de sensor DHT22 que mide temperatura). El anexo al final de esta propuesta explica brevemente la programación para Arduino.

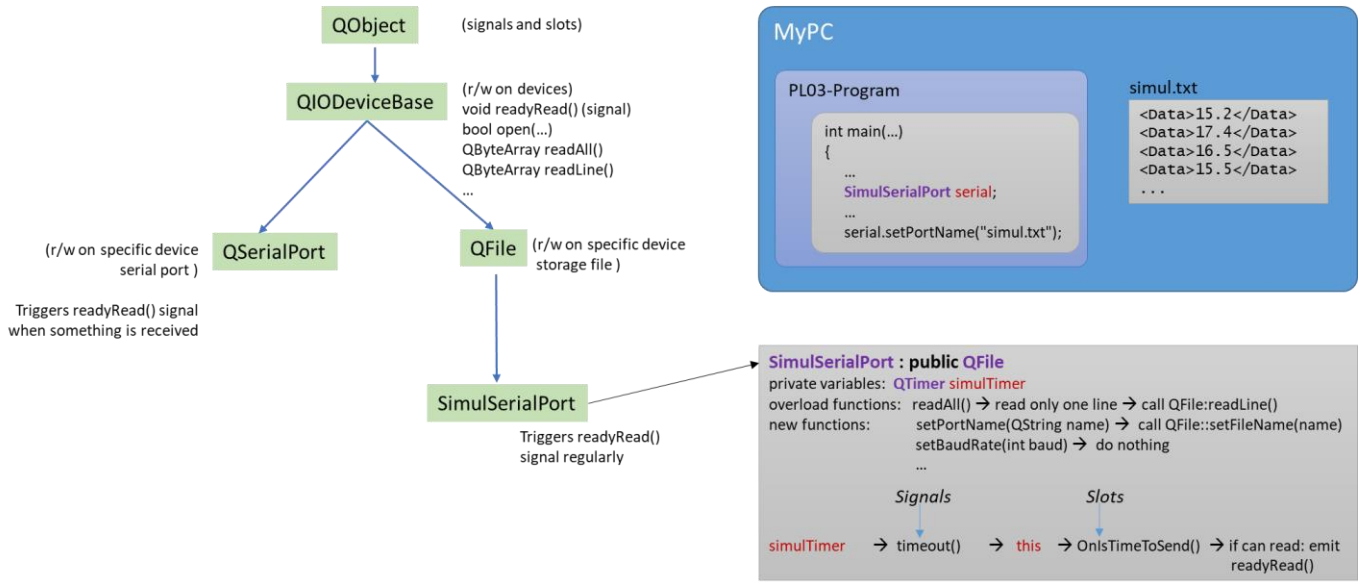
LeeTemperatura.ino

```
#include "DHT.h"
#define DHTPIN  5
#define DHTTYPE DHT22
#define TM_ms   1000

DHT  dht(DHTPIN, DHTTYPE);
long  timePrev us, timeNow us;

void setup() {
  Serial.begin(115200);
  _timePrev_us=0;
   dht.begin();
}

void loop() {
  _timeNow_us=micros();
  if (_timeNow_us-_timePrev_us >= TM_ms*1000L)
  {
     timePrev us= timeNow us;
    int16_t temp=_dht.readTemperature();
    if (true)
    {
        Serial.print("<Temp>");
        Serial.print(temp);
        Serial.print("</Temp>\r\n");
    }
  }
}
```

### Simulación de puerto serie mediante archivo:

Si no se dispone de Arduino, o se quieren hacer pruebas de manera más controlada, se puede simular la recepción de datos mediante una nueva clase en nuestro programa, SimulSerialPort, que sustituirá a QSerialPort simulando la recepción mediante lectura temporizada de datos de un archivo, para lo cual necesitará un temporizador interno (simulTimer).

La nueva clase deriva de QFile y debe tener las mismas funciones de QSerialPort, aunque no hagan nada. Tanto QFile como QSerialPort obtienen la mayoría de sus funcionalidades (señales, slots y funciones de r/w de QIODeviceBase), por lo que la interacción con cualquiera de ellas es muy parecida.




- **Ampliaciones propuestas:**

Cambiar la recepción por puerto serie a recepción por WiFi UDP/IP (QUdpSocket)

Cambiar la recepción por puerto serie a recepción por WiFi TCP/IP (QTcpServer, QTcpSocket)

## ANEXO: Programación desde Arduino IDE

La programación en entorno Arduino sigue los mismos criterios de la programación orientada a objetos, con algunas pequeñas diferencias:

1) El entorno de programación (Arduino IDE) proporciona un archivo de texto inicial "NombreProyecto.ino" sobre el que escribir. Es equivalente a un archivo .cpp, con la estructura siguiente:

```
NombreProyecto.ino
#include "Arduino.h"

void setup() {
  Serial.begin(115200);
  Serial.print("Starting in setup\n");
}

void loop() {
  Serial.print("Running loop\n");
  delay(1000);
}
```

2) No existe la función main( ). Realmente, main( ) sí existe pero el Arduino IDE la hace inaccesible al programador. La función main( ) interna equivale a:

```
#include <Arduino.h>
int main(void)
{
    init();

#if defined(USBCON)
    USBDevice.attach();
#endif

    setup();  // Calls user defined function

    for (;;) {
        loop();  // Calls user defined function
        if (serialEventRun) serialEventRun();
    }

    return 0;
}
```

3) Al no tener acceso a main( ), las variables con tiempo de vida "todo el programa", que normalmente se deberían emplazar en main( ), ahora serán variables globales (accesibles por todas las funciones).

4) Las clases y funciones definidas en la librería por defecto de Arduino se encuentran declaradas en Arduino.h y otros archivos de cabecera. La documentación es bastante "pobre" (https://www.arduino.cc/reference/en/libraries/ ). La mayoría de clases tienen una función begin( ) y otra end( ) para inicializar/terminar sus objetos.

5) Algunas de las funciones más utilizadas:
   - pinMode( ), digitalRead( ), digitalWrite( ) : para E/S digital en los pines GPIO
   - pinMode( ), analogRead( ): para entrada ADC en los pines AD
   - pinMode( ), analogWrite( ): para salida PWM en los pines PWM
   - delay( ), delayMicroseconds( ): para retardar la ejecución del programa (evitar en lo posible)
   - millis( ), micros( ): obtener el tiempo transcurrido desde el inicio.

6) Algunas de las clases más utilizadas:
   - String: para manejo de cadenas de caracteres.
   - Serial: para comunicación serie. Por defecto, Arduino.h crea un objeto de esa clase con el mismo nombre Serial, que permite la comunicación con el ordenador de desarrollo (u otro cuando el programa ya está cargado) vía conversor USB ←→ RS-232 incorporado.

- SPI: para comunicación por bus SPI (incluir "SPI.h"). Por defecto, SPI.h crea un objeto de esa clase con el mismo nombre SPI, que permite la comunicación mediante los pines SPI por defecto de la tarjeta utilizada.
- Wire: para comunicación por bus I2C (incluir "Wire.h"). Por defecto, Wire.h crea un objeto de esa clase con el mismo nombre Wire, que permite la comunicación mediante los pines I2C por defecto de la tarjeta utilizada.
- Servo: para control de servomotores de CC.
- Stepper: para control de motores stepper.
- Otras (SD, EEPROM, LiquidCrystal, TFT, …)

7) Otras clases: muchos otros programadores han realizado clases para Arduino, las hay para casi todo, pero la mayoría adolecen de documentación muy deficiente. Es muy difícil saber cuáles son buenas y malas.
8) Organización de nuestros programas. En los programas Arduino, se pueden añadir archivos .h y .cpp en los que desarrollemos clases y funcionalidades, igual que en cualquier otro programa .cpp. Hay que evitar escribir todo el código en el .ino suministrado por defecto.
9) Cuando se sube un programa al Arduino/ESP32 a través de la conexión USB, éste programa se ejecutará cada vez que se encienda el equipo, no hace falta subirlo de nuevo salvo modificaciones.