UNIVERSIDAD DE OVIEDO

Departamento de Ingeniería Eléctrica,
Electrónica, de Computadores y Sistemas

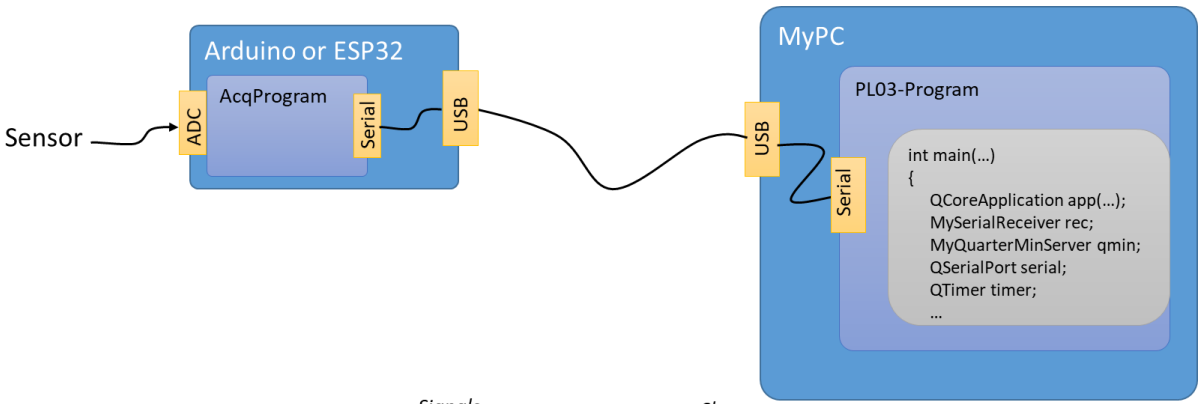
Guía de Prácticas

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 que, cada segundo, 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. Se recibirá un dato en formato real xml : <Dato>15.2</Dato>.
  - Por cada dato recibido por el puerto serie, se escribirá en un archivo de salida:
    - Dicho dato para los segundos "normales"
    - El doble de dicho dato si está activado el cuarto de minuto

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



Signals *Slots*

timer → timeout() → qmin → On1SecondElapsed()
serial → readyRead() → rec → OnDataReceived()
qmin → QuarterMin (bool) → rec → OnQuarterMin(bool)

MyQuarterMinServer
On1SecondElapsed() : get current time (use QTime object)
{ second % 15 == 0 : emit QuarterMin(true);
{ second % 15 == 1 : emit QuarterMin(false);

MySerialReceiver
OnDataReceived() : read string , decode xml , data*multiplier , write in file
OnQuarterMin(bool isStarting) : change internal multiplier

MySerialReceiver.h

```
#include <QSerialPort>
...

class MySerialReceiver : public QObject // Derivar de QObject para usar signals/slots
{
   Q OBJECT
   ...
private:
   ...
   QSerialPort serial;
   float multiplier=1;
   ...
public:
   MySerialReceiver (QObject* parent=NULL);
public slots:
   ...
   void OnDataReceived();
   void OnQuarterMin(bool isStarting);
} ;
```

MySerialReceiver.cpp

```
#include " MySerialReceiver.h"
...
MySerialReceiver:: MySerialReceiver (QObject* parent) : QObject(parent)
{
    // Initialize serial port
    serial.setPortName("COM3");  // Cambiar según puerto serie disponible
    serial.setBaudRate(115200);   // Debe ser igual al programado en el Arduino
    serial.setDataBits(QSerialPort::Data8); // Debe ser igual al programado en el Arduino
    serial.setParity(QSerialPort::NoParity); // Debe ser igual al programado en el Arduino
    serial.setStopBits(QSerialPort::OneStop); // Debe ser igual al programado en el Arduino

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

void MySerialReceiver::OnDataReceived()
{

    QString data= serial.readAll();
    Usar data
}
```

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.

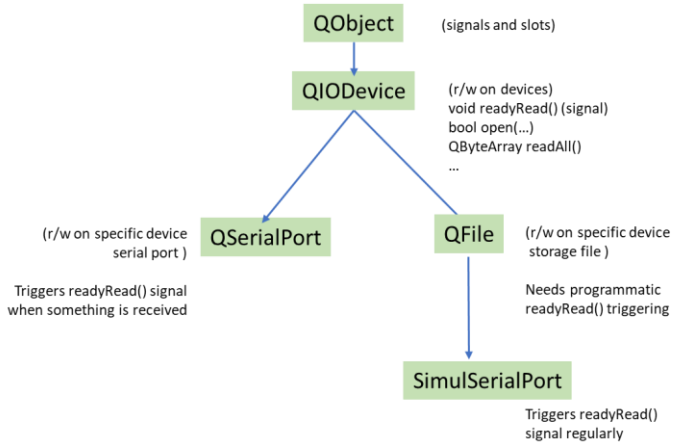
LeeTemperatura.ino

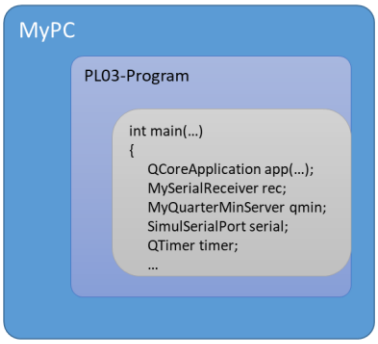
```
#include "DHT.h"
#define DHTPIN  5
#define DHTTYPE DHT22
#define TM ms   1000

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");
    }
  }
}
```

Si no se dispone de Arduino, realizar una tercera clase (SimulSerialPort) que simula el puerto serie mediante lectura temporizada de datos de un archivo usando un segundo temporizador, y generando la señal readyRead( ) cuando se dispone de una nueva línea; sustituir QSerialPort por esta nueva clase.

Simulación de puerto serie mediante recepción de archivo:

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 QIODevice), por lo que la interacción con cualquiera de ellas es muy parecida





SimulSerialPort.h

```
#ifndef SIMULSERIALPORT_H
#define SIMULSERIALPORT_H

#include <QFile>
#include <QTimer>

class SimulSerialPort : public QFile
{
    Q_OBJECT
private:
    QTimer simulTimer;
public:
    explicit SimulSerialPort(QObject *parent = nullptr);
    void setPortName(const QString& name) { setFileName(name); }
    void setBaudRate(int) { return; }
    void setDataBits(int) { return; }
    void setParity(int) { return; }
    void setStopBits(int) { return; }
signals:

public slots:
    void OnIsTimeToSend();
};

#endif // SIMULSERIALPORT_H
```

Bajo el temporizador, que se arranca en el constructor, simplemente se emite readyRead( ) como haría el puerto serie real cuando se reciben datos:

SimulSerialPort.cpp

```
#include "SimulSerialPort.h"

SimulSerialPort::SimulSerialPort(QObject *parent) : QFile(parent)
{
    connect(&simulTimer,SIGNAL(timeout()),this,SLOT(OnIsTimeToSend()));
    simulTimer.start(2000);
}

void SimulSerialPort::OnIsTimeToSend()
{
    emit readyRead();
}
```

Los cambios en MySerialReceiver son mínimos

MySerialReceiver.h

```
#include "SimulSerialPort.h" // En lugar de QSerialPort
...

class MySerialReceiver : public QObject
{
   ...
   SimulSerialPort serial;  // En lugar de QSerialPort
   ...
} ;
```

MySerialReceiver.cpp

```
#include " MySerialReceiver.h"
...
MySerialReceiver:: MySerialReceiver (QObject* parent) : QObject(parent)
{
    serial.setPortName("C:/User/.../MyData.txt");  // Cambiar según archivo de datos
    serial.setBaudRate(115200);   // La hemos programado para no hacer nada, no hay baudRate para archivo
    ...
    if (serial.open(QIODevice::ReadWrite))
    {
        connect(&serial,SIGNAL(readyRead()),this,SLOT(OnDataReceived()));
    }
}

void MySerialReceiver::OnDataReceived()

{

    QString data= serial.readLine();  // No leer el archivo de golpe sino línea por línea
    Usar data
}
```

- Ampliaciones:

Utilizar para ir obteniendo datos reales de los movimientos de la mano/joystick y guardarlos en archivos, para tener disponibles cuando se vea cómo tratar y filtrar este tipo de señales.

Unir a PL01 y PL02 para leer datos en tiempo real y filtrar de acuerdo con los criterios que se verán en las clases siguientes de procesamiento de señal.

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.