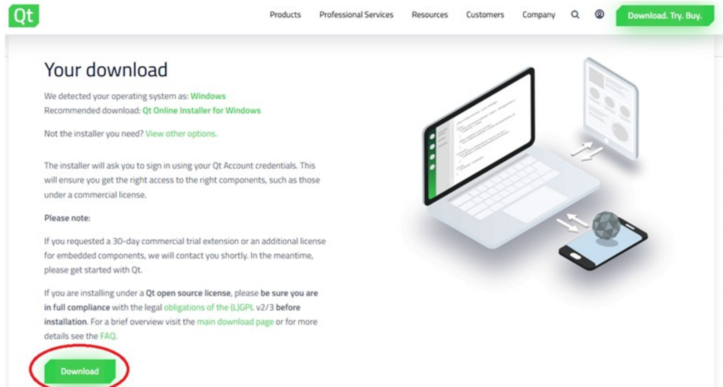
Sumario

Instalar Qt..5
Que es Qt..6
  Manejo de memoria..7
  Señales y Ranuras...8
Qt Creator...9
  Porque usar Qt...9
  Estructura de un proyecto Qt...12
Las clases de Qt...16
Hola mundo con Qt..16
  Asignando un icono a la ventana principal.......................................18
Agregar imagen a la ventana principal...20
Proyecto con dos ventanas..24
Intercambio de datos entre ventanas...28
  Las ventanas propietarias de sus objetos...28
  Señales y Slot personalizados...29
Alineación de objetos en ventanas...32
Trabajando con una barra de menú..33
  Trabajando con un archivo de texto desde un menú...........................37
  Iconos en la barra de menú...38
  Iconos en la barra de Herramientas..40
Sensor BMP280 + Arduino + Qt..40
  Análisis del proyecto BMP280 parte I...44
Análisis del proyecto BMP280 parte II...49
BMP280 dos ventanas y barra de herramientas.....................................59
La macro Q_OBJECT...65
Sensor DHT22 con Qt + Socket de red..67
  Conectando Arduino por Socket UDP...68
  Análisis del proyecto Qt + DHT22 + Socket UDP.................................72

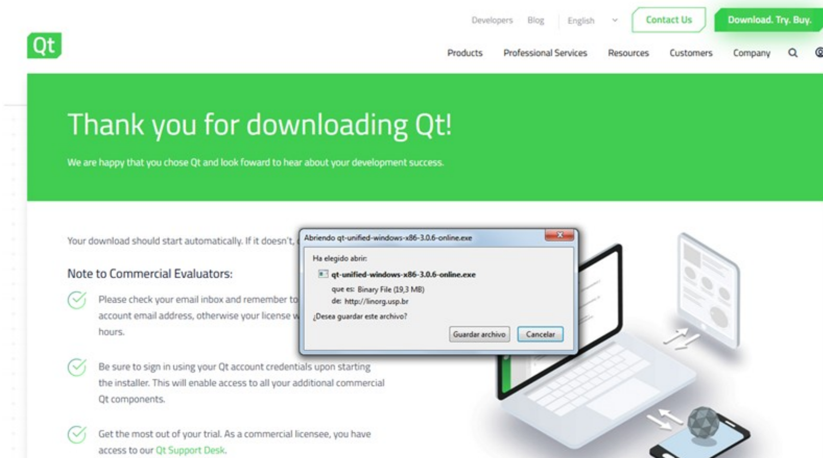
Control de pines Arduino + Qt + Socket de red..79
Control PID con Qt..95
Funcionamiento general de un PID..96
Interfaz Qt para un PID...97
Control PWM con Qt y Arduino...98
Electrónica del proyecto PID..101
Programa Arduino del PID...102
Programa Qt del PID..106

Instalar Qt.

El proceso de instalación de Qt es muy simple, se accede al sitio oficial de Qt y se descarga la versión *Open Source*.



Una vez iniciado el proceso de descarga guardamos el archivo y lo ejecutamos en nuestra computadora.



El sitio web detecta el sistema operativo e indica la versión adecuada a instalar, la configuración inicial del entorno de trabajo es automática y no requiere atención del usuario, en Windows eventualmente será necesario agregar al *Path* algunas rutas para las herramientas de Qt sobre todo si en el futuro se decide instalar *QWT* para sumar los *Widgets* de ese modulo. Los *Widgets* no son otra cosa que cada uno de los objetos que podemos desplegar en un formulario o ventana principal de un proyecto Qt.

Que es Qt.

Es un *framework,* un marco de trabajo que funciona tanto en Windows, Linux o Mac. Básicamente es programación en C++ y es ampliamente usado para desarrollar programas que utilicen interfaz gráfica de usuario (GUI), agrega también diferentes tipos de herramientas para la línea de comandos y consolas de trabajo para sistemas que no necesitan una GUI.
Qt es software libre y de código abierto, originalmente era un desarrollado de la empresa noruega Trolltech que al ser comprada por Nokia en el 2008 pasó a ser un desarrollo de la división de software de Nokia,
Qt es utilizado en el escritorio KDE para sistemas como GNU/Linux o FreeBSD, productos de Adobe y una gran cantidad de aplicaciones comerciales utilizan Qt como enlace con el usuario.
Utiliza el lenguaje de programación C++ de forma nativa, también puede ser utilizado en varios otros lenguajes de programación a través de adaptaciones o *biding*. Ejemplos de estos bindings son Qt Jambi (Java), PyQt (Python), PHP-Qt (PHP) o Qyoto (C#), entre otros muchos.
Encontramos Qt en sistemas informáticos embebidos para automoción, aeronavegación y aparatos domésticos diversos, etc.
El API de la biblioteca Qt cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API unificada para la manipulación de archivos y una gran cantidad de otros métodos para el manejo de estructuras de datos tradicionales.
Lo que diferencia a Qt de un programa C++ cualquiera es que añade muchísimas funcionalidades a C++, cambiándolo de tal forma, que prácticamente crea un nuevo lenguaje de programación. Además, facilita la tarea de programar en C++, que en casos como en la programación de entornos gráficos, puede ser bastante pesada.
Sin embargo Qt no deja de ser C++, es decir, siempre se pueden usar librerías estándar o cualquier otra librería y la sintaxis de C++ normal y corriente, por lo cual, es muy versátil.
Qt se basa en la programación orienta a objetos (POO) y es esto lo que hace que Qt sea tan potente y fácil de usar.
Los dos pilares de Qt son la clase *QObject (objeto Qt)* y la herramienta *MOC (compilador de metaobjetos).*
La programación mediante objetos Qt permite derivar nuevas clases de objetos QObject.
Al crear nuevos objetos derivados se heredan una serie de propiedades que caracterizan a Qt y que marcan diferencias con el C++ estándar por ejemplo.

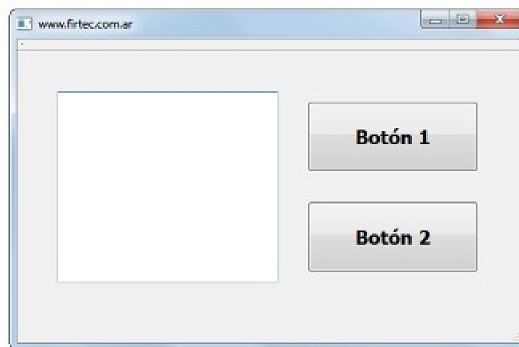
- Una gestión simple del manejo de memoria.

- El concepto de Señales y Ranuras.

Veamos un poco más sobre estos dos puntos.

Manejo de memoria.

Cuando se crea una instancia de una clase derivada de un QObject es posible pasar al constructor un puntero al objeto padre siendo esta la base de la gestión de memoria. Al borrar un objeto padre, se borran todos sus objetos hijos asociados, esto quiere decir que una clase derivada de QObject puede crear instancias de objetos pasándole el puntero *this* como padre sin preocuparse de la destrucción de estos hijos.
Esto es una gran ventaja frente al C++ estándar, ya que en C++ estándar cuando se crea un objeto siempre hay que tener cuidado de destruir los objetos uno a uno y de liberar memoria con *delete*.
Un ejemplo de aplicación podría ser por ejemplo crear un ventana que sería el objeto padre, agregar dos botones *QPushButton* y una línea de texto editable *QPlainTextEdit*, estos componentes son hijos de esta ventana.
En la imagen siguiente se puede ver el aspecto del ejemplo propuesto. La ventana principal es una clase derivada de QObject que permite representar objetos gráficos como ventanas, botones, etc. Por lo tanto *QPushButton* y *QPlainTextEdit* derivan de la ventana principal.
Es importante entender el concepto de que en Qt todo lo que vemos en una ventana (incluso la propia ventana) es un objeto.



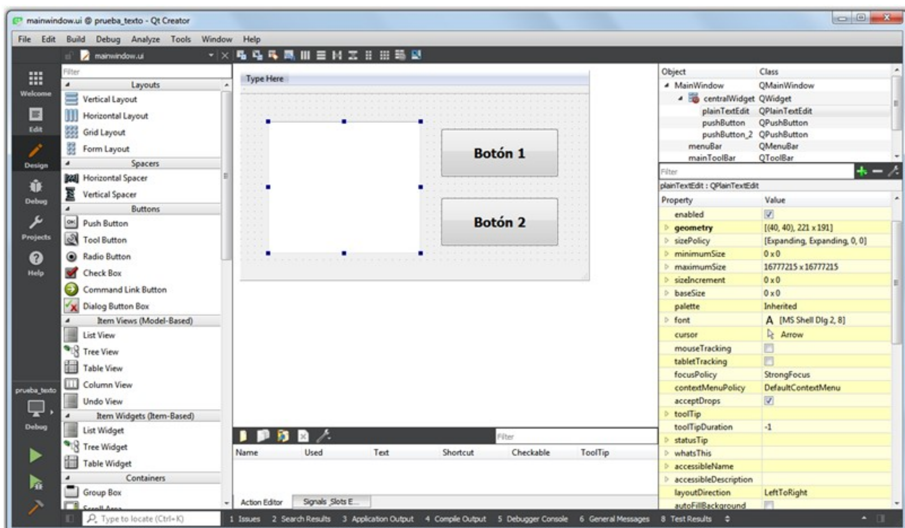
Al destruir la ventana principal se destruirían los hijos liberando la memoria correspondiente de manera transparente al programador.

Qt Creator.

Es la interfaz gráfica, el IDE (Entorno Integrado de Desarrollo) de Qt, de aspecto sencillo, intuitivo y muy simple de usar, Qt Creator es la herramienta para construir las aplicaciones que usaremos como ejemplos.
Es importante tener claro que Qt Creator es solo un IDE, en la plataforma Windows el compilador *mingw* (compilador GCC) es el encargado compilar las aplicaciones, podemos incluso generar aplicaciones Qt tipo consola que requieran una interfaz gráfica pero que también podemos crear desde Qt Creator.

Porque usar Qt.

El trabajo con electrónica y mas precisamente con microcontroladores hace que muchas veces se necesiten interfaces para relevar datos desde una computadora, tableta o móvil.



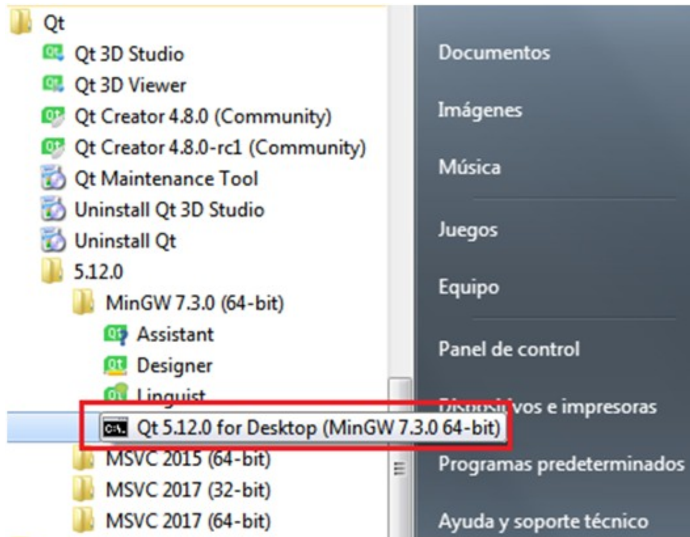
Crear interfaces gráficas con Qt es bastante sencillo, como se aprecia en la imagen anterior la interfaz se ensambla arrastrando los componentes necesarios a la ventana, escribimos el código para la acción que debe realizar el componente (botón, edición, etc) y al compilar se obtiene un archivo ejecutable similar al de cualquier programa Windows.
En un sistema Windows cuando se quiere ejecutar una aplicación en otra máquina que no tenga instalado Qt, se presenta la situación de las librerías compartidas, no solo basta con tener el archivo *exe* también hay una serie de

librerías *DLL* que serán necesarias llevar junto con el archivo ejecutable.

Para solucionar esto hay dos caminos.

- Realizar una compilación estática, esto inserta el código de las librerías en uso en el propio ejecutable, aumenta el tamaño del ejecutable, pero reduce dependencias.
- Proporcionar las DLLs necesarias para el funcionamiento de los programa junto con el ejecutable, se puede incluso utilizar un instalador del tipo .net para distribuir el programa.

windeployqt.exe es una herramienta de Qt que nos ayudará a incluir las librerías DLL que el ejecutable necesita.
Para acceder a esta herramienta abrimos la terminal de Qt (no use *CMD* de Windows!!).



Estando dentro de la carpeta que contiene el ejecutable que queremos exportar a otra computadora que no tenga instalo Qt, ejecutamos ***windeployqt.exe*** **.** <u>no olvide el punto</u>, esto desencadena un proceso en donde todos los archivos necesarios para que la aplicación funcione se agregarán a la carpeta donde está el ejecutable, luego según el caso se puede armar un instalador con alguno de los muchos programas que existen para esto.

```
#endif // MAINWINDOW_H
```

El archivo *mainWindow.ui, ui* significa *User Interface* y es el archivo que tiene la configuración de la interfaz gráfica de usuario, los objetos que desplegamos en la ventana con *Qt Creator* se encuentran en este archivo y desde luego el código de la ventana principal esta ligado a estos elementos.

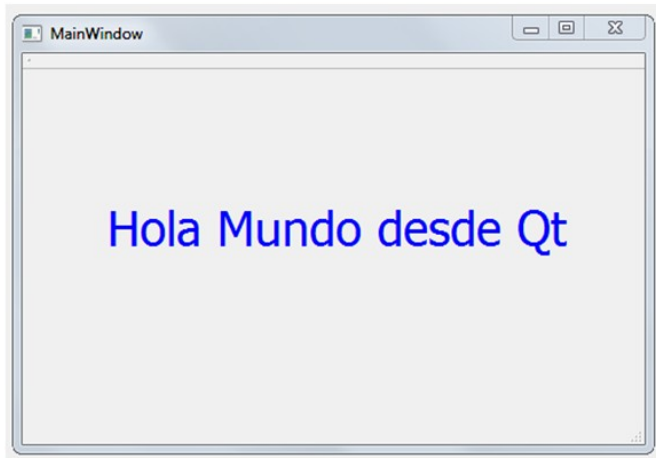
Las clases de Qt.

Recordando que una clase es una plantilla, un modelo con el cual podemos crear objetos que hereden sus características, el número de clases de Qt es realmente enorme pero podemos comentar algunas de las mas usadas en los ejemplos tratados.

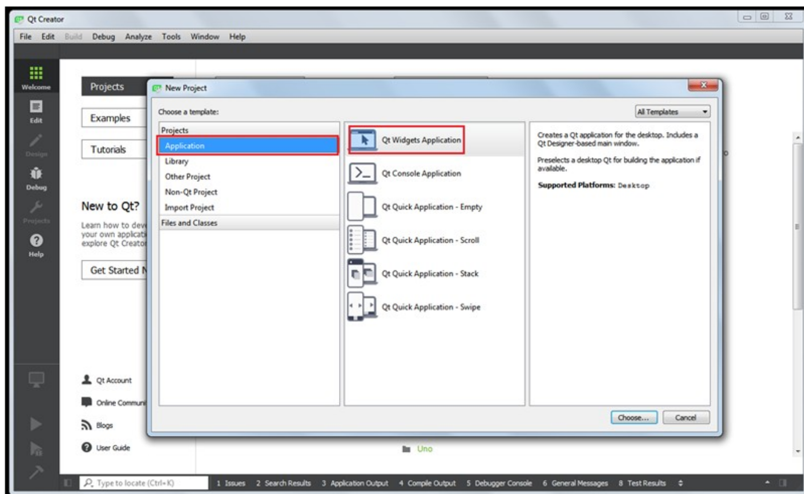
- QObject: Es la clase más importante de todas, ya que describe el objeto básico Qt, que permite el manejo de memorias y las señales y slots.
- QWidget: Es la clase que permite crear el elemento básico de interfaz de usuario.
- QMainWindow: Widget de tipo MainWindow, la ventana principal de programa.
- QDialogBox: Widget de tipo cuadro de diálogo estándar.
- QLabel: Widget Permite mostrar etiquetas de texto o incluso imágenes.
- QLineEdit: Widget Muestra una línea de texto.
- QTextEdit: Es similara QLineEdit pero permitiendo mostrar texto, incluso en formato HTML.
- QPushButton: Widget Esta clase sirve para desplegar botones.
- QFrame: Widget Sirve para generar marcos.
- QMessageBox: Widget Despliega los Message Box.
- QLayout: Es la clase que gestiona la geometría de los objetos mostrados.
- QBoxLayout: Alinea los widgets de manera vertical u horizontal.
- QThread: Clase que gestiona hilos de ejecución independientes del SO que se esté usando.
- QTimer: Clase que proporciona una interfaz de alto nivel para los temporizadores.

Hola mundo con Qt.

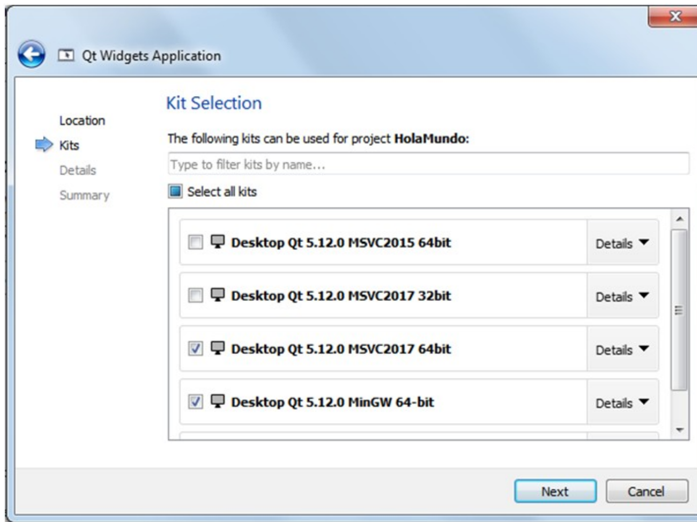
Vamos a iniciar el primer ejemplo con el clásico “Hola mundo”.



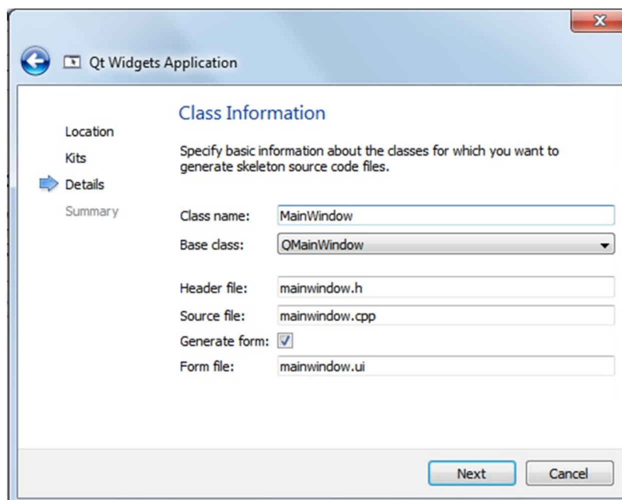
Una ventana simple con un objeto *Label* y un texto serán suficientes para este primer ejemplo.
Primero seleccionamos el tipo de programa que vamos a crear, en este caso será una aplicación Qt.



El paso siguiente determinar en que carpeta crearemos el proyecto y luego seleccionar que herramientas configuradas en el entorno Qt vamos a usar.



Es importante que la selección de herramientas sea la correcta, tendremos problemas para crear la versión *release* si no configuramos esto correctamente. El paso siguiente será definir los nombres que tendrán los archivos que formarán el proyecto, por el momento dejamos los que pone por defecto.



Y ya estamos en condiciones de generar una primera aproximación de la aplicación que vamos a generar.

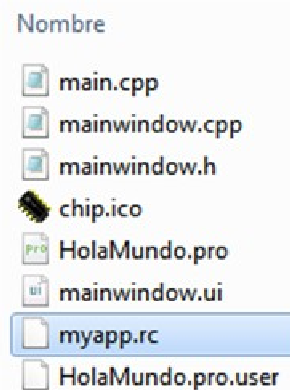
Asignando un icono a la ventana principal.

Notará que la ventana que hemos creado lleva el icono por defecto y un título de ventana también por defecto.

Para cambiar esto y personalizar nuestra ventana se seguirán los siguientes pasos.
Primero necesitamos crear un archivo que en nuestro caso hemos llamado *myapp.rc*. En este archivo se ha escrito solo una línea de código.

```
IDI_ICON1 ICON DISCARDABLE "chip.ico"
```

Siendo "*chip.ico*" el icono que vamos a usar en nuestra ventana y que debe estar en la misma carpeta donde está el proyecto de lo contrario se debe agregar la ruta completa del icono a usar.
La carpeta del proyecto se vería del siguiente modo.



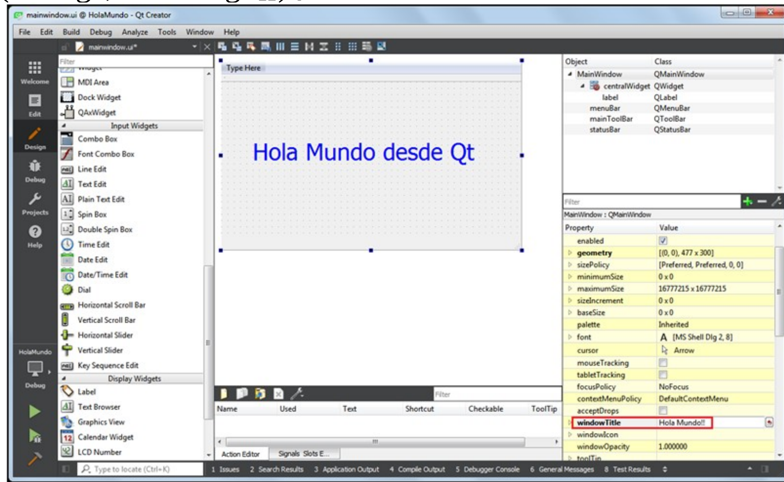
Luego editamos el archivo *HolaMundo.pro* y en su encabezado agregamos la siguiente línea resaltada.

```
QT += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
RC_FILE = myapp.rc
TARGET = HolaMundo
TEMPLATE = app
```

También vamos a cambiar el título de la ventana principal y para hacer esto solo se ha editado la propiedad *windowTitle* de la ventana principal para colocar el texto que se quiera como título de ventana.

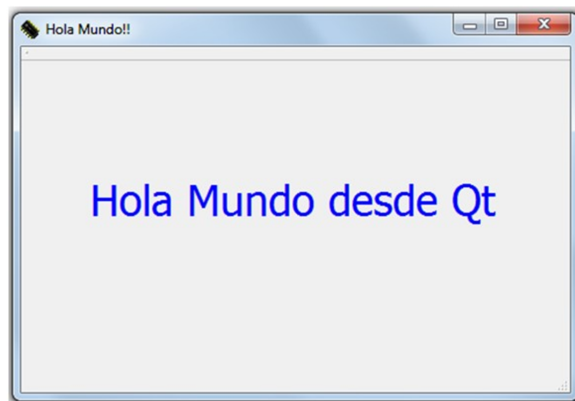
Desde las propiedades de la ventana principal podemos cambiar su aspecto, colores, fuente, texto, geometría, etc.
Si queremos que la ventana no pueda cambiar su geometría cuando el usuario ejecuta la aplicación, una forma simple de fijar el tamaño de la ventana es editar el archivo *main.cpp* del proyecto y agregar el siguiente código dentro de la función *main()* y antes de hacer la ventana visible.
En este caso se ha fijado un tamaño de ventana de 479 x 300 pixeles.

int main(int argc, char *argv[]){



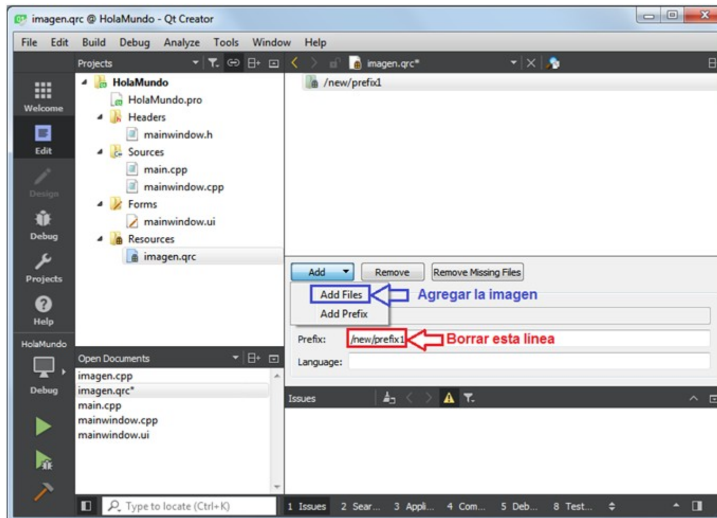

```
QApplication a(argc, argv);

MainWindow w;

w.setFixedSize(477,300);

w.show();
return a.exec();
}
```

El resultado al compilar el ejemplo con las reformas será el siguiente.

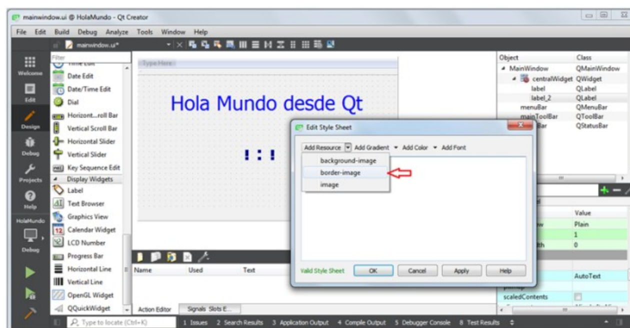


Agregar imagen a la ventana principal.

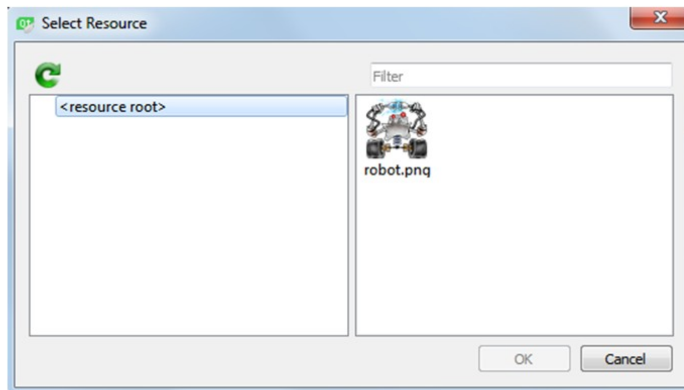
Puede ser necesario ya sea por cuestiones decorativas o funcionales el agregado de alguna imagen en la ventana que visible.



Luego agregamos a nuestra ventana principal un nuevo *Label* y desde sus propiedades borramos el texto de tal forma que solo quede el *Label* vacío.



Estando el nuevo *Label* seleccionado con el botón derecho del *mouse* seleccionamos *Change style Sheet* y agregamos el recurso que este caso será *border-image*.
Seguidamente tendremos el listado de las imágenes disponibles para asignar a *border-image*.
Esta selección completará todo el *Label* con la imagen, solo tenemos que achicar/agrandar el *Label* con el *mouse* para adecuarlo al tamaño y ubicación que se desee).
En este ejemplo solo hay disponible una imagen para asignar a este recurso.



Una vez cumplido todos los pasos el resultado final al compilar el proyecto es el que se aprecia en la siguiente imagen.

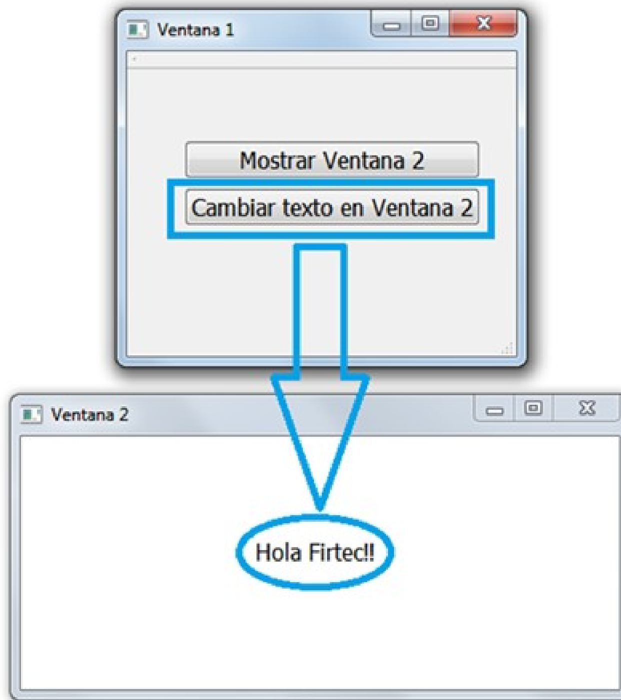
Proyecto con dos ventanas.

Es común que muchas veces necesitamos más de una ventana en el proyecto, por ejemplo una ventana de configuración o una ventana específica para mostrar los datos también puede ser una ventana donde el usuario se registra etc.
El procedimiento para crear otro formulario *(ventana)* es bastante simple, Qt tiene un mecanismo para asociar la nueva ventana al proyecto de manera "*casi*" automática. El programador es quien finalmente incluye los correspondientes archivos de cabecera necesarios.
El siguiente ejemplo crea una nueva ventana (*Ventana 2*) que se hace visible al actuar sobre un botón colocado en la primer ventana (*Ventana 1*).

ventana. Si es necesario cerrar Ventana 1 escribimos ***this->close()*** y Ventana 1 se cierra dejando solo Ventana 2.

Intercambio de datos entre ventanas.

Como vimos Ventana 2 tiene un *Label* donde se lee "*Texto original!!*", vamos agregar un segundo botón para cambiar el contenido de ese *Label* desde la primer ventana.



Para hacer esto debemos tener muy claro algunos conceptos de C++.

Las ventanas propietarias de sus objetos.

Como se dijo antes, Qt es C++ y esto significa que aquí todo tiene "*dueño*", todo tiene un campo de visibilidad y privacidad.

Partiendo de eso y simplificando mucho las cosas, podemos decir que todos los objetos desplegados en una ventana pertenecen a esa ventana y no tienen conexión alguna con otras ventanas.

Por lo tanto para que un objeto de Ventana 1 interactúe con un objeto de Ventana 2, primero este debe saber la dirección del objeto destino.

Para hacer esto hay dos caminos.

- *Crear un puntero al objeto buscado. (Simple pero no aplica en todos los casos).*
- *Conectar los objetos mediante señal/slot personalizado para la acción que se busca.*

Considerando que los objetos propios de Qt tienen slot definidos, estos también tienen distintos tipos de señales que podemos usar.
Por ejemplo el un botón tiene sus slot pero también emite señales.

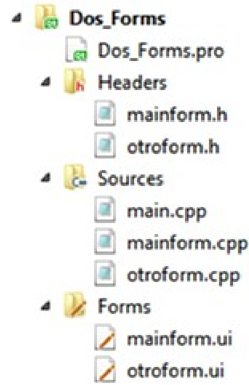
- ***clicked()*** Esta señal se emite cuando se hace *click* sobre el botón.
- ***toggled()*** Cuando el botón cambia de estado se emite esta señal.
- ***pressed()*** Señal que se emite cuando el botón es apretado.
- ***release()*** Señal cuando el botón es liberado.

En nuestro ejemplo vamos a usar la señal *clicked()* para conectarnos con el objeto en Ventana 2.

Señales y Slot personalizados.

Para poder "*encontrar*" el objeto *Label* de Ventana 2 vamos a crear un slot en esa ventana que reciba la señal del botón en Ventana 1, el slot lo hemos llamado *datosRemotos()*.
La conexión la hacemos en el *main()* de Ventana 1 de la siguiente forma.

```
MainForm::MainForm(QWidget *parent) :

QMainWindow(parent),

ui(new Ui::MainForm)

{

ui->setupUi(this);

      connect(ui->pushButton, SIGNAL(clicked()), Window2,
                     SLOT(datosRemotos()));
```

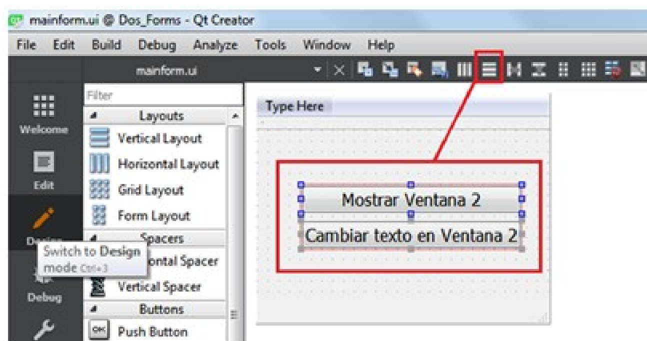


Estos son todos los archivos que forman el ejemplo, no olvide agregar en las cabeceras los correspondientes archivos *H* de cada uno de los formularios.

Recordar.
En Qt todo lo que desplegamos en una ventana son objetos y pertenecen a esa ventana y solo pueden tener acceso a ellos los objetos miembros de esa “Clase”. Por defecto no hay forma de interactuar con objetos en ventanas distintas si no es por medio de un puntero al objeto o mediante conexiones entre señales y slot's personalizados.

Alineación de objetos en ventanas.

Cuando se despliegan objetos en una ventana suele ser tedioso y complicado una correcta alineación de estos objetos en el formulario.

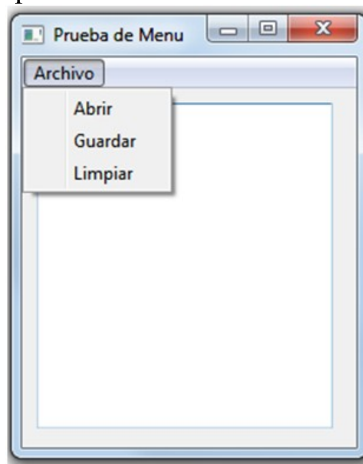


Qt tiene una herramienta para ajustar esto, esta alineación puede ser tanto vertical como horizontal y se encuentra en la parte superior del menú general de Qt. (Ver imagen siguiente).
Solo basta con seleccionar todos los objetos que se busca alinear y hacer click

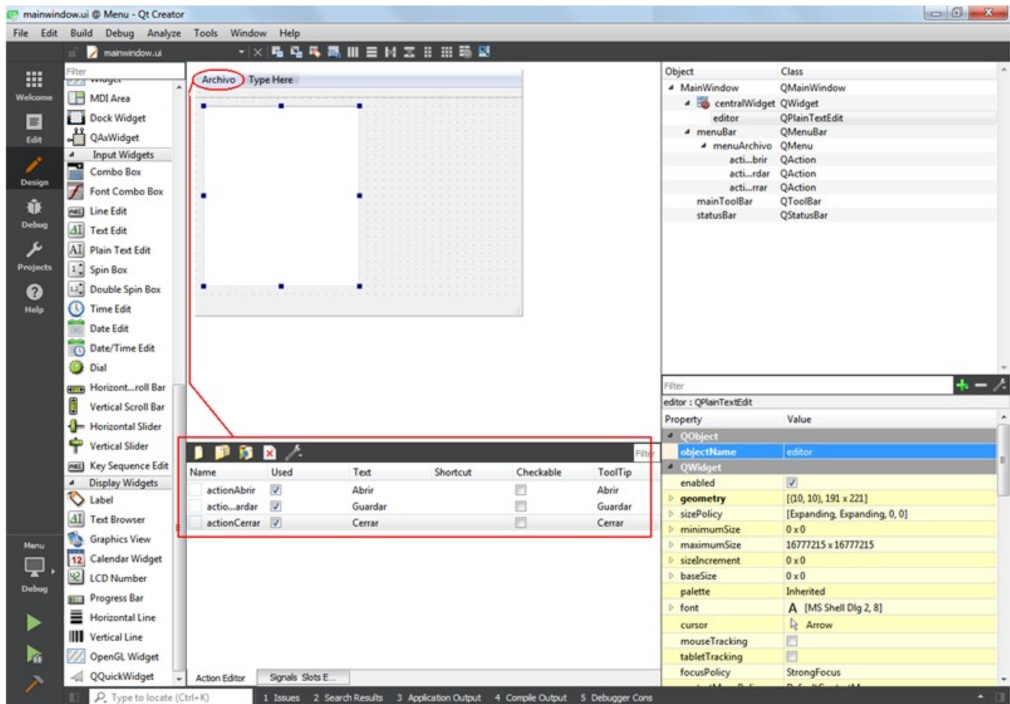
sobre el botón correspondiente para que se ajusten automáticamente.
En la siguiente imagen se puede ver como realizar un ajuste vertical de los objetos seleccionados.

Trabajando con una barra de menú.

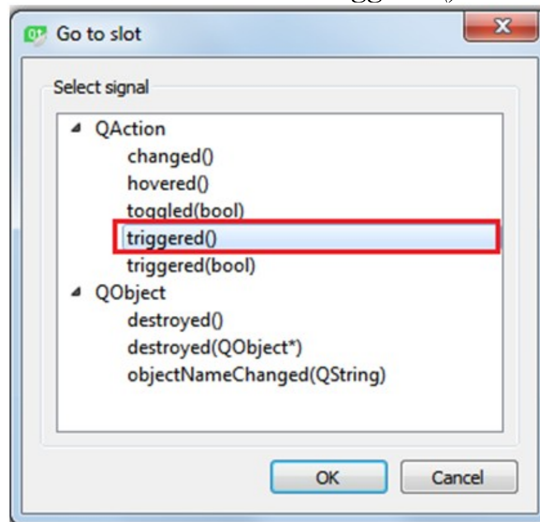
Para ver el funcionamiento de un menú vamos a crear una simple aplicación que muestra en un componente Text Edit el contenido de un archivo de texto que llamamos “prueba.txt”.
Nuestro simple menú permite abrir el archivo y ver su contenido, agregar nuevos datos y también limpiar el contenido del editor (no del archivo).
Cuando se crea un nuevo formulario (*ventana*), de manera automática se genera la barra de menú, la barra de herramientas y la barra de estado al pié de la ventana, lo único que debemos hacer es agregar los rótulos (*Abrir, Guardar, Limpiar*) y la acción que necesitamos hacer con cada rótulo que coloquemos en el menú.
En la imagen siguiente se aprecia como vemos en Qt Creator los correspondientes rótulos que tiene el menú.



Cada uno de estos rótulos tendrá asignada una acción como se aprecia en el recuadro de la imagen, cada una de estas acciones puede ser un slot disponibles para el menú.
En nuestro caso nos resulta de interés el slot *triggered()*, esta señal se emite cada vez que hacemos click sobre un rótulo determinado.



Cuando nos posicionamos sobre un rótulo o item del menú, botón derecho del ratón y vamos a *Go to slot,* seleccionamos *triggered()*.



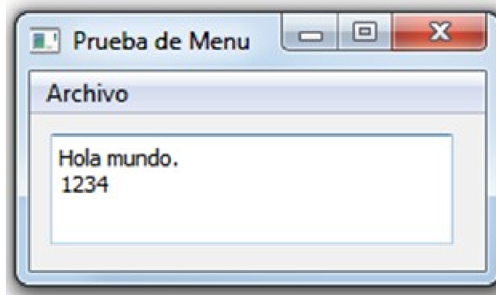
Esta selección será la misma para todos los rótulos del menú. En el archivo *mainwindow.h* se puede ver la declaración automática de los slot correspondientes.

El archivo *mainwindow.cpp* contiene todo lo necesario para el

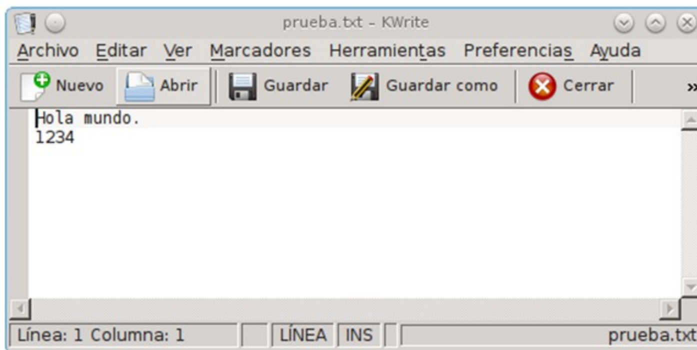
```
void MainWindow::on_actionCerrar_triggered(){
ui->editor->setPlainText("");
}
```

Trabajando con un archivo de texto desde un menú.

En nuestro menú la acción "*Abrir*" muestra el contenido del archivo en el componente de edición de texto, resultado sera como se aprecia en la siguiente imagen.



El contenido real del archivo *prueba.txt* es el siguiente y como se puede ver la aplicación refleja el contenido de este archivo.



Lo primero que encontramos en el slot "*Abrir*" es la definición de una variable tipo archivo vinculado con el archivo que nos interesa. (La ruta del archivo corresponde a la computadora donde se verificó el funcionamiento del ejemplo).

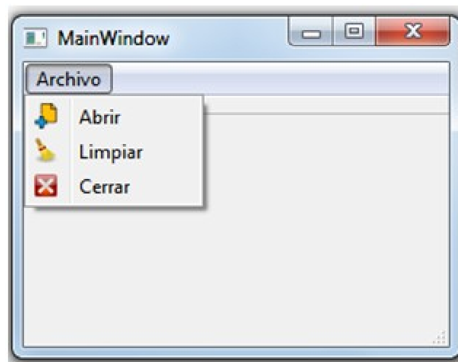
QFile archivo("C:/Electrónica/Programas QT/Menu/ejemplo.txt"). Este archivo se abre en modo lectura y escritura, ***archivo.open(QIODevice::ReadWrite)*** y el editor de texto se configura para leer texto plano desde la variable archivo

ui→editor→setPlainText(archivo.readAll()).

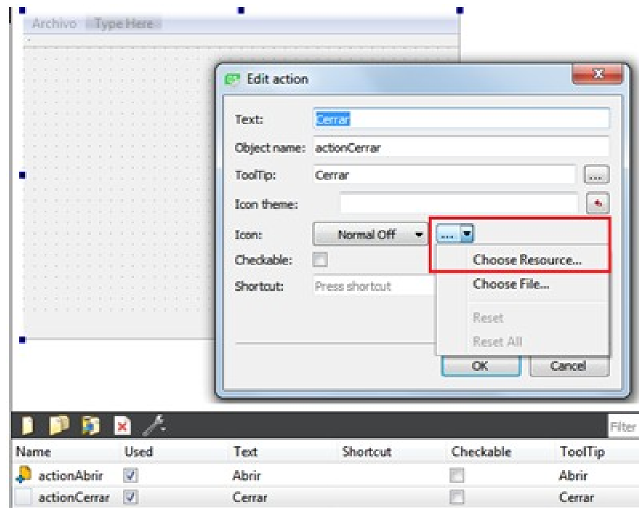
El componente **QIODevice** es una interfaz que permite el acceso a distintos componentes de hardware entre ellos a los discos.
Para guardar datos en el archivo nuestro menú tiene el rótulo "*Guardar*" que dispara una señal capturada por el correspondiente slot que realiza el trabajo en el archivo.
Igual que en el slot anterior se crea una variable tipo archivo, ***QFile archivo("C:/Electrónica/Programas QT/Menu/ejemplo.txt")*** pero en este caso hay una diferencia en la forma que abrimos el archivo, ***archivo.open(QIODevice::ReadWrite | QIODevice::Text)*** indicando que abrimos el archivo en modo lectura y escritura y que trabajamos en modo texto.
Definimos una variable tipo cadena ***QTextStream texto(&archivo)*** esto para guardar en texto lo que enviaremos al archivo en disco.
Luego pasamos lo que esté en el editor a esta variable ***texto<<ui->editor->toPlainText()*** que será lo que finalmente termine en el archivo del disco. (*Observe los signos angulares*).
Finalmente cerramos el archivo ***archivo.close()***.
Para limpiar el contenido del editor que se encuentra en el slot que dispara la señal "Limpiar" solo basta con enviar al editor una cadena vacía ui→editor→setPlainText("").

Iconos en la barra de menú.

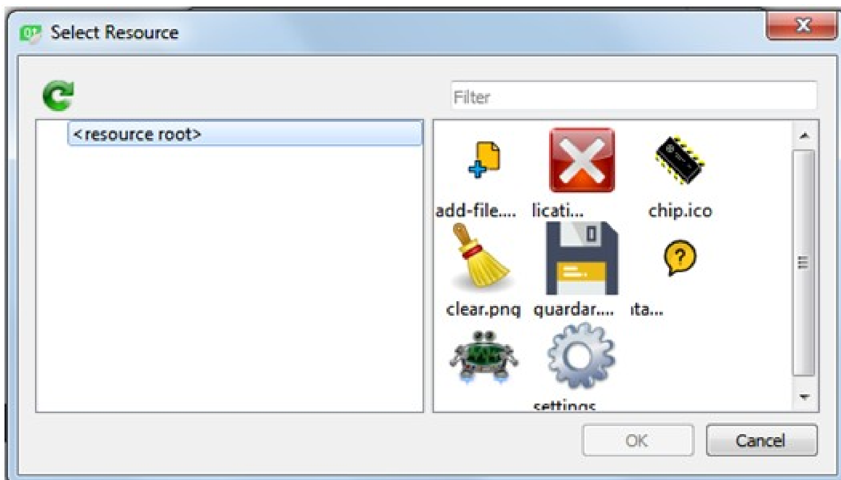
El menú puede quedar mucho mas profesional si agregamos iconos en cada uno de los items que el menú tenga.



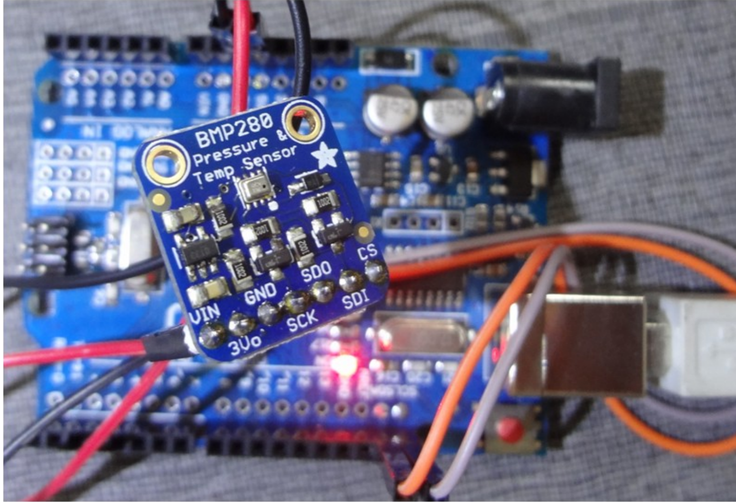
Para lograr esto necesitamos de un archivo de recursos para contener las imágenes y el procedimiento para crear ese archivo y agregarlo al proyecto es exactamente igual a cuando agregamos una imagen a la ventana.



Para agregar una imagen a un item del menú simplemente lo seleccionamos desde la lista de acciones o items y con el botón derecho del ratón lo editamos, luego seleccionamos agregar un recurso y elegimos la imagen de las disponibles en los recursos que hemos cargado.
Todos los pasos son muy parecidos a cargar una imagen en el formulario solo que aquí es para el menú y siempre que se trabaje con este tipo de recursos se deberá de contar con un archivo de recursos previamente asociado al proyecto.



- *Ofrecer la posibilidad de seleccionar la velocidad en baudios que se requiera.*
- *Mostrar en la barra de estado si estamos conectados y bajo que condiciones.*
- *Mostrar el valor de temperatura y presión leídos desde el sensor BMP280.*



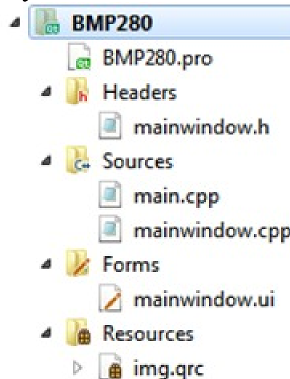
Una placa Arduino Uno se conecta a un sensor BMP280 mediante el puerto I2C de Arduino.
El *Sketch* o programa Arduino es el siguiente.

```
/***************************************************

* Protocolo: Si recibe el carácter "a" envía la
* temperatura.

* Si recibe el carácter "b" envía la presión.

* Si recibe el carácter "c" limpia los buffer de
* comunicaciones.
* Placa Arduino: UNO
* Arduino IDE: 1.8.13
* www.firtec.com.ar

***************************************************/

#include <Wire.h>

#include <SPI.h>
```

Para el intercambio de información entre Qt y Arduino se ha diseñado un simple protocolo que consiste en el envío de caracteres de control.
Cuando desde la ventana Qt se envía el carácter “a” Arduino responde enviando la temperatura leída desde el sensor BMP280.
Cuando Qt envía el carácter “b” Arduino responde enviando la presión barométrica leída desde el sensor BMP280.
La recepción del carácter “c” limpia los *buffers* de recepción en Arduino.
Desde Qt el control dinámico de este protocolo de comunicaciones está comandando por un temporizador que cada un segundo envía una “*signal”* que es recibida por un “*slot”* (método que atiende el evento del temporizador) y es en este método donde se decide que comando enviar.

Análisis del proyecto BMP280 parte I.

El árbol de archivos que conforman el proyecto contiene el archivo *PRO* con la descripción general del proyecto.



Los archivos *CPP* y *H* del método principal como de la ventana y un archivo de recursos para una imagen que se agrega a la ventana principal.
El resultado final de la interfaz que vamos a diseñar es la siguiente.

Lo primero que vamos hacer es editar el archivo *BMP280.pro* y en su encabezado agregar las siguientes líneas resaltadas en rojo.

```
QT += core gui

QT += widgets

QT += core gui serialport
# Necesario para el control del puerto serial
RC_FILE = myapp.rc
 # Carga el icono de la aplicación

.

.
```

Con esto tendremos control del puerto serial como también agregamos un icono a la ventana.
En el archivo *main.cpp* se fijado un tamaño de pantalla para que el usuario no pueda cambiar la geometría de la ventana principal.

```
#include "mainwindow.h"
```

- QByteArray serialData2
- bool bandera = false bandera de uso general.
- bool error = false bandera para reportar errores.

Importante II.
Para poder usar los objetos en la ventana de trabajo tendremos siempre que cargar su correspondiente archivo de cabecera, si lo olvidamos cuando compilamos el proyecto se genera un error al desconocer la naturaleza del objeto que pretendemos usar.

Análisis del proyecto BMP280 parte II.

El archivo *mainwindow.cpp* contiene todo el código de funcionamiento de la aplicación, y dentro del método principal de *mainwindow.cpp* encontramos la siguiente líneas de código.

```
Q_FOREACH(QSerialPortInfo port,
QSerialPortInfo::availablePorts()) {
ui->puerto->addItem(port.portName());
}
```

Aquí es donde se analizan los posibles puertos COM que el sistema tiene disponible.
Estos puertos se enumeran en un *QComboBox* que en este ejemplo se llama "*puerto*", aquí se van agregando de tal forma que al desplegar el *QComboBox* se pueda seleccionar uno de los puertos disponibles en el sistema.
En la imagen siguiente puede ver la ubicación de este *QComboBox* .

También se ha dispuesto otro *QComboBox* que se ha definido como *baudBox* y cuya lista son las posibles velocidades en baudios que se han programado como posibles en este ejemplo.

En la imagen anterior puede ver la ubicación del *QComboBox* para seleccionar la velocidad en baudios de la comunicación.

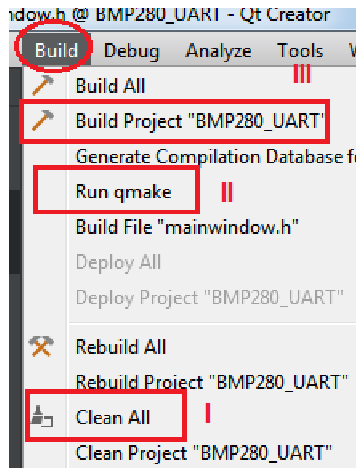
```
if(bandera){

QMessageBox::warning(this, "CONECTADO!!", "NO PUEDE CAMBIAR LA
CONFIGURACIÓN");

}

else

Window2->show(); // Hace visible la segunda ventana

}
```

En la siguiente imagen se puede ver como funciona el bloqueo de la segunda ventana si el usuario intenta configurar el puerto COM al mismo tiempo que se encuentra conectado.

La variable bandera, igual que en el ejemplo anterior, indica si esta conectado a un puerto COM y cambia su estado cada vez que se acciona el botón *Conectar – Desconectar*.
El código en este slot es el siguiente, se han resaltado dos líneas encargadas de solicitar los datos a la ventana de configuración.

```
void MainWindow::on_pushButton_clicked(){

if(bandera == false){

arduino->setPortName(Window2->COMx);
```

```
namespace Ui {

class Form;

}

class Form : public QWidget

{

Q_OBJECT

public:
explicit Form(QWidget *parent = nullptr);
~Form();
public slots:
void Datos_Sensor();
public:
Ui::Form *ui;
private slots:
void on_config_clicked();
};
#endif // FORM_H
```



Si **no** agregamos la macro Q_OBJECT **no se pueden usar señales/slot** en el proyecto, y si agregamos o quitamos esta macro en un proyecto ya ensamblado será necesario cumplir con tres pasos para volver a compilar el proyecto y que los cambios surtan efecto.
Quitar o agregar Q_OBJECT en un proyecto implican grandes cambios en la forma en que el proyecto se construye y por esto es que se debe reconstruir

toda la estructura del proyecto.

Para recordar.

Cuando se usan señales y slots en un proyecto será necesario incluir en la definición de la clase principal la macro Q_OBJECT que aparecer en la sección privada de una clase que declara sus propias señales y ranuras o que utiliza otros servicios proporcionados por el sistema de metaobjetos de Qt.
El MOC genera las definiciones de las variables miembro y los métodos que la macro Q_OBJECT ha declarado y se requiere este código de metaobjeto para el mecanismo que controla el funcionamiento de las señales y ranuras, el sistema dinámico de propiedades y algunas funciones internas de Qt. Si la aplicación no utiliza señales y ranuras esta macro puede no estar aumentando ligeramente la velocidad de compilación.

Sensor DHT22 con Qt + Socket de red.

El siguiente ejemplo propone leer la temperatura y humedad desde un sensor *DHT22* por medio de un *Socket UDP* y seguramente el lector encontrará variadas definiciones de lo que es un socket en redes informáticas, sin embargo desde el punto de vista de la electrónica con microcontroladores, podríamos simplemente decir que es el RS-232 de TCP-IP.
Una de las formas mas simples de conectar un microcontrolador, PLC o electrónica en general a una computadora es a través de un puerto serial con el conocido protocolo RS-232.
Un socket hace eso mismo, establece una conexión entre dos puntos, sin importar donde se encuentren estos puntos y esto si es una gran diferencia con el viejo RS-232.
Podemos hacer una medición de voltaje, temperatura, humedad o lo que fuera necesario verificar y transmitir estos datos a cualquier parte del mundo por TCP-IP para esto solo necesitamos dos cosas:

1. *Una dirección IP donde enviar los datos.*
2. *Un puerto virtual donde los datos será recogidos.*

Hay varios tipos de socket pero dos son de uso mas común.

- **Los socket de flujo** *(*`SOCK_STREAM`) *que son transportados por TCP (Protocolo de Control de Transmisión) y asegura que los mensajes*

```
uint16_t port;

QByteArray datagram;

QString dato1;

QString dato2;

datagram.resize(socket->pendingDatagramSize());
// Tamaño del datagrama a leer.

while (socket->hasPendingDatagrams()) {

socket-
>readDatagram(datagram.data(),datagram.size(),&sender,&port);

ui->IP->setText(sender.toString());
// Mustra la IP a la que se conecta

QString recibido_socket(datagram);
// Convierte QByteArray a QString

QStringList datos = recibido_socket.split(",");
// Busca la marca separadora de campos ","

ui->l1->setText(datos[0]);
ui->l2->setText(datos[1]);
statusBar()->showMessage(tr(" Conectado en puerto
%1").arg(port));
}}
```

Para el agregado de las imágenes a la ventana principal se usó el mecanismo explicado en casos anteriores.

Control de pines Arduino + Qt + Socket de red.

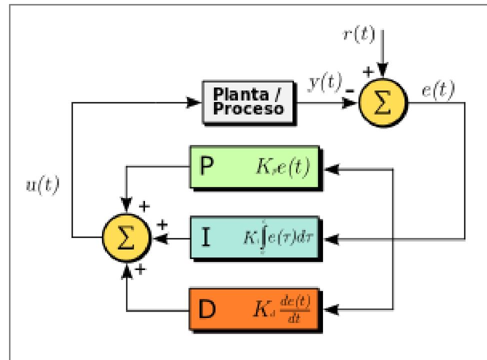
En el ejemplo anterior la placa *Arduino* envía los datos de un sensor *DHT22* a un programa Qt mediante un socket UDP, es decir que el programa Qt solo recibe datagramas, en este ejemplo vamos a recibir el voltaje de un canal analógico pero también vamos a cambiar el estado de pines de *Arduino* enviado comando por un socket UDP y para eso vamos a crear la siguiente interfaz Qt. (*En nuestro canal de* ***youtube*** *puede ver un vídeo de la aplicación funcionando*).

Cuatro botones serán los encargados de cambiar el estado de los pines en la placa Arduino, también tenemos un *Label* que muestra el voltaje leído en A0, como en el ejemplo anterior los datos de conexión del socket son también mostrados en sus correspondientes *Label's*.
Del lado de *Arduino*, tenemos un programa relativamente simple que solo decodifica los datagramas enviados desde Qt y actuando según lo que programado para cada comando.
En este caso *Arduino* configura su propia IP usando el servidor DHCP disponible en la red de prueba, también se puede ajustar para una IP fija como en ejemplos anteriores.
Para la interpretación de los datos enviados por Qt se ha usado un simple *switch( )* que según sea el caso pone un pin a nivel alto o bajo.

```
switch(packetBuffer[0]){

case '1':{

digitalWrite(5, HIGH);

break;

}

case '2':{

digitalWrite(5, LOW);
```

Control PID con Qt.

Primero un poco de teoría. PID es un sistema de control por realimentación ampliamente usado en sistemas de control industrial, calcula la desviación o error entre un valor medido y un valor deseado. El algoritmo del control PID consiste de tres parámetros distintos: el *proporcional*, el *integral*, y el *derivativo*.



El valor proporcional depende del error actual, el integral depende de los errores pasados y el derivativo es una predicción de los errores futuros. La suma de estas tres acciones es usada para ajustar al proceso por medio de un elemento de control como la posición de una válvula de control o la potencia suministrada a un calentador o a un motor.
Ajustando las tres variables en el algoritmo de control del PID, el controlador puede proveer una acción de control diseñado para los requerimientos del proceso en específico. La respuesta del controlador puede describirse en términos de la respuesta del control ante un error, el grado el cual el controlador sobrepasa el punto de ajuste, y el grado de variación del sistema.
Está claro que el uso del PID no garantiza control óptimo de un sistema o la estabilidad del mismo, sin embargo el PID permite un ajuste muy fino optimizando el uso de los recursos asignados a ese control.
Algunas aplicaciones pueden solo requerir de uno o dos modos de los que provee este sistema de control. Un controlador PID puede ser llamado también PI, PD, P o I en la ausencia de las acciones de control respectivas.
Los controladores PI son particularmente comunes, ya que la acción derivativa es muy sensible al ruido, y la ausencia del proceso integral puede evitar que se alcance al valor deseado debido a la acción de control.

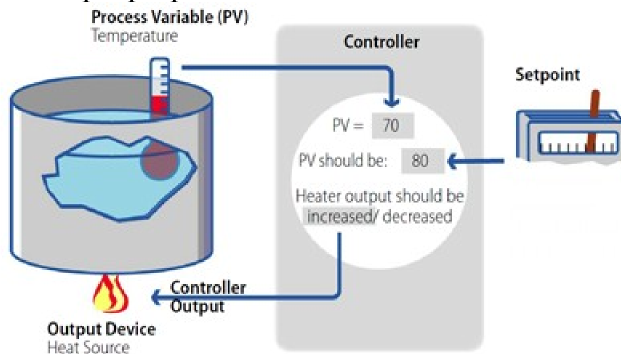
Funcionamiento general de un PID.

Para el correcto funcionamiento de un sistema PID que controle un proceso se necesita al menos tres elementos básicos.

1. Un sensor, que determine el estado del sistema.
2. Un controlador, que genere la señal que gobierna al actuador.
3. Un actuador, que modifique al sistema de manera controlada.

El sensor proporciona la información al controlador, la cual representa el punto actual en el que se encuentra el proceso o sistema.

El controlador lee una señal externa que representa el valor que se desea alcanzar. Esta señal recibe el nombre de punto de consigna (*o punto de referencia*), la cual es de la misma naturaleza y tiene el mismo rango de valores que la señal que proporciona el sensor.



El controlador resta la señal de punto actual a la señal de punto de consigna, obteniendo así la señal de error, que determina en cada instante la diferencia que hay entre el valor deseado (*consigna*) y el valor medido.

La señal de error es utilizada por cada uno de los tres componentes del controlador PID. Las tres señales sumadas, componen la señal de salida que el controlador va a utilizar para la acción final de control. La señal resultante de la suma de estas tres se llama variable manipulada que será transformada para ser compatible con el actuador utilizado en el sistema por ejemplo un PWM.

Básicamente un control PID se puede reducir al siguiente algoritmo de control:

$$u(t) = K_p \left[e(t) + \frac{1}{T_i} \int e(t)dt + T_d \frac{de(t)}{dt} \right] = P + I + D$$

Por ejemplo imagine el lector que intenta controlar la temperatura de un recipiente con agua mediante un calefactor eléctrico.

Se ha fijado el ajuste a 40° grados. Con el sistema clásico SI/NO cuando la masa liquida llegue a los cuarenta grados el sistema corta la potencia del

calefactor sin embargo la inercia térmica de la propia masa hace que el valor de temperatura sea superado ampliamente, lo mismo ocurre cuando la masa líquida empieza a enfriarse, cuando está por debajo del punto de ajuste el sistema se activa sin embargo la propia inercia hace que transcurre un tiempo en que sigue perdiendo temperatura hasta que el efecto del calentador se hace notar y la temperatura sube.
Está claro que se desperdicia mucha energía permitiendo que el valor de temperatura suba por encima del valor fijado lo mismo cuando el valor de temperatura baja, todo esto debido a que un sistema SI/NO aplica toda la potencia en el momento que se activa y corta esta potencia cuando se desactiva.
Un PID modula esta potencia según la necesidad, es decir que si la masa líquida tiene un valor de temperatura alejado de su punto de ajuste aplicará el 100% de potencia al calefactor y a medida que la temperatura de ajuste se acerque al valor real irá bajando esta potencia hasta que la temperatura real coincida con la de ajuste, si la masa líquida baja su temperatura el PID aplica solo la potencia necesaria para recuperar el estado de equilibrio.
Una forma simple de controlar potencia con un sistema electrónico es con un *PWM* (*Control de potencia por modulación de pulso*). Es común que un sistema PID incorpore en su estructura de control un PWM.

Interfaz Qt para un PID.

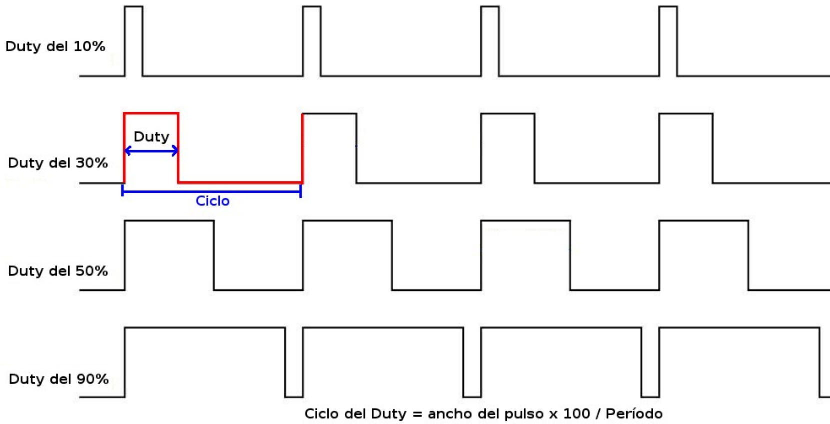
En el ejemplo que proponemos usaremos un PWM para controlar la temperatura dentro de una caja de cartón mediante una lámpara que calentará el aire alrededor de un sensor de temperatura (*LM35*) y un sistema PID será responsable de modular la potencia aplicada a esta lámpara calefactora según la diferencia entre el valor real y el valor de ajuste.

En este caso la interfaz Qt se comunica con la parte electrónica mediante una conexión serial pero podría ser Bluetooth, Red, etc.
Desde la interfaz se ajusta el PID enviando no solo el valor de referencia, también se puede realizar un ajuste fino en tiempo real cambiando las tres variables básicas del PID.
Desde la electrónica se recibe el valor actual de temperatura pudiendo así ver en todo momento como el PID realiza su trabajo. Para esta interfaz se han agregado varios objetos nuevos a la ventana, potenciómetros, visores, controles deslizantes, paneles, etc.
La electrónica que implementa el PID bajo el control de Qt será una placa Arduino que incorpora una biblioteca completa para el manejo básico de un PID mediante PWM.

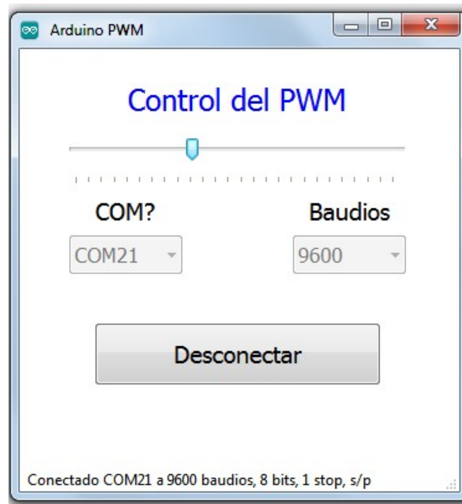
Control PWM con Qt y Arduino.

El ciclo útil de una señal PWM es la proporción de tiempo que el pin está e nivel alto respecto al total del ciclo, este tiempo se denomina “*Duty cycle*”, y generalmente se expresa en un porcentaje del ciclo total.

Para

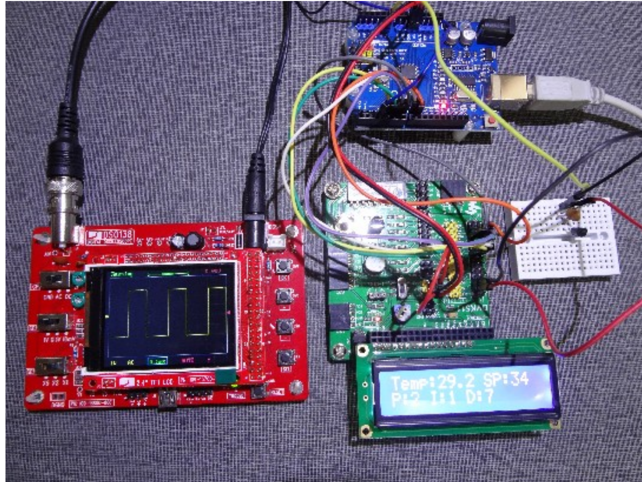


Entender un poco mas como funciona el PWM base de nuestro PID, vamos primero a realizar una simple prueba con una ventana Qt que interactúa con el modulo PWM de una placa Arduino.



Esta simple aplicación enviará por el puerto serial los datos para cambiar el ciclo de trabajo de un PWM con salida por el pin tres de la placa Arduino. La información se envía cada vez que el valor del objeto *horizontalSlider* cambie. De hecho el código completo de su método slot es el siguiente.

```
void MainWindow::on_horizontalSlider_valueChanged(int value)

{
```



Ajustando el funcionamiento del PID.

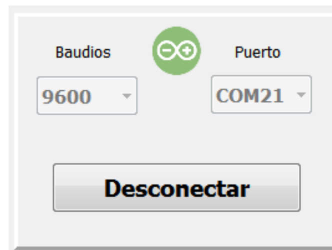
Programa Qt del PID.

Siendo la conexión con Arduino mediante un puerto COM, los datos de conexión y configuración son los mismos que en los ejemplos anteriores con algunos pequeños cambios, por ejemplo en el archivo *mainwndow.cpp* en su método *main()* se han cargado los valores por defecto en la cadena a transmitir como se puede ver en las siguientes líneas.

```
MainWindow::MainWindow(QWidget *parent) :
QMainWindow(parent),
ui(new Ui::MainWindow){
ui->setupUi(this);
dato.resize(6);
dato[0]=25;
dato[1] = 8;
dato [2] = 2;
dato [3] = 1;

Q_FOREACH(QSerialPortInfo port,
QSerialPortInfo::availablePorts()) {
ui->puerto->addItem(port.portName());
}
ui->baudBox->addItem(QStringLiteral("1200"),
QSerialPort::Baud1200);
ui->baudBox->addItem(QStringLiteral("2400"),
QSerialPort::Baud2400);
ui->baudBox->addItem(QStringLiteral("4800"),
QSerialPort::Baud4800);
ui->baudBox->addItem(QStringLiteral("9600"),
QSerialPort::Baud9600);
```

```
ui->baudBox->addItem(QStringLiteral("2400"),
QSerialPort::Baud19200);
ui->baudBox->addItem(QStringLiteral("38400"),
QSerialPort::Baud38400);
ui->baudBox->addItem(QStringLiteral("57600"),
QSerialPort::Baud57600);
ui->baudBox->addItem(QStringLiteral("115200"),
QSerialPort::Baud115200);
statusBar()->showMessage(tr("Desconectado!"));
}
```

Luego todo lo referente a la comunicación, puerto, baudios y botón Conectar/Desconectar es igual a los ejemplos anteriores.



Control de comunicaciones.

El contenido del archivo *mainwindow.h* contiene las declaraciones de los slot individuales de cada uno de los objetos de control desplegados en la ventana.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QtSerialPort/QSerialPort>
#include <QtSerialPort/QSerialPortInfo>
#include <QMessageBox>
namespace Ui {
class MainWindow;
}
class MainWindow : public QMainWindow
{
Q_OBJECT
public:
explicit MainWindow(QWidget *parent = nullptr);
~MainWindow();
QSerialPort *arduino;
QByteArray dato; // Datos para enviar
QByteArray bufferRX; // Datos para recibir
private slots:
void on_horizontalSlider_valueChanged(int value);
void LeerSerial();
void on_pushButton_2_clicked();
void on_ajustarPID_clicked();
```

```
void on_dial_kp_valueChanged(int value);
void on_dial_ki_valueChanged(int value);
void on_dial_kd_valueChanged(int value);
private:
Ui::MainWindow *ui;
bool bandera = false;
bool error = false;
};
#endif
```

El módulo de control del PID involucra varios objetos nuevos en la ventana.

La clase *QDial* dispone de estos controles que son como diales o potenciómetros circulares de ajuste, son absolutamente configurables y dan una buena presentación a la interfaz.
La clase *QLCDNumber* dispone de una pantalla tipo *display* de siete segmentos también con una configuración muy flexible, tipo de dígito, aspecto, etc.
Observe que cada vez que el valor en alguno de los objetos de ajuste para el PID cambia bajo la acción del usuario, este nuevo valor se carga en la cadena *dato[ ]* en el slot que corresponde a cada elemento o control de ajuste.

```
void MainWindow::on_dial_kp_valueChanged(int value){

ui->lcdkp->display(value);

dato[1] = value & 0xFF;

}
```