UNIVERSIDAD DE BURGOS

Desarrollo de interfaces para Arduino en Visual C++

JOSÉ M. CÁMARA NEBREDA

(checam@ubu.es)

2016

CONTENIDO

INTRODUCCIÓN

La formación del ingeniero electrónico en España, en sus diferentes modalidades, suele adolecer de carencias relevantes a nivel de programación. A menudo, tras haber pasado por diversas asignaturas en las que se ha trabajado con dispositivos programables (microprocesadores, microcontroladores, robots, entornos integrados, etc) el ingeniero es capaz de llevar a cabo tareas de programación sencillas utilizando lenguajes estructurados como C o similares a C.

Una capacidad tan limitada en la programación hace que un dispositivo tan potente y habitual como el PC y sus derivados (laptop, Tablet, transformables, etc) supongan una barrera insalvable para el electrónico. Considerados como dispositivos electrónicos, estas herramientas presentan unas prestaciones extraordinarias:

- Una interfaz de usuario de gran amplitud y calidad.
- Gran espacio de almacenamiento.
- Elevada potencia de cálculo.
- Múltiples posibilidades de comunicación, tanto cableada (USB, Firewire…) como inalámbrica (Wifi, Bluetooth…)
- Un coste muy razonable.

La aparición en los últimos años de entornos de desarrollo integrados de bajo coste y gran versatilidad, de los cuales Arduino es sin duda el más extendido, ha proporcionado al electrónico una salida asequible hacia múltiples aplicaciones con un esfuerzo de aprendizaje mínimo. La modularidad del sistema Arduino le permite incorporar accesorios de bajo coste para acceder a las funciones más diversas, a nivel de almacenamiento, control de dispositivos, comunicación, etc. Una de las grandes ventajas de este entorno es la sencillez de programación. Los denominados “sketches” de Arduino son programas estructurados con una sintaxis similar a C, con funcionalidades típicas de entornos de control como el watchdog en tiempo real y en los que resulta fácil acceder a los diferentes dispositivos externos.

El principal problema de estos entornos, no obstante, es la escalabilidad. No es problema incorporar en ellos cualquier funcionalidad adicional, ya sea mediante los denominados “shields” o mediante módulos de comunicación serie, típicamente SPI o I2C. La dificultad aparece cuando lo que se necesita es incorporar múltiples funcionalidades. Entonces los “shields” comienzan a apilarse, los puertos, interrupciones, e/s y demás elementos de expansión empiezan a escasear, el coste ya no es tan reducido como al principio y, finalmente, el “sketch” empieza a parecer ingobernable. Un software excesivamente complejo está a menudo sujeto a errores de programación y resulta difícil de mantener y ampliar.

Existen alternativas más potentes que las distintas versiones de Arduino, como puede ser el caso de Raspberrry Pi, que se acerca más al esquema de un computador tradicional que al de un dispositivo de control embebido. En estos casos, la inclusión de un sistema operativo y/o la aparición de entornos de programación orientados a objeto, reproduce el “escalón” con el que se encuentra el profesional de la electrónica a la hora de enfrentarse a la programación.

¿Por qué Visual C++? … sería la pregunta pertinente en este momento. Dicen los estudiosos del software que C++ surge como evolución natural de C para dotarle de las capacidades de un lenguaje orientado a objeto. Dicen también que otros lenguajes orientados a objeto, como

Java, surgen a partir de él. Parece entonces que es el lenguaje de programación orientado a objeto de referencia. Una ventaja es que la sintaxis es heredada de C, por lo que su comprensión puede ser más sencilla. Otra ventaja, para la electrónica, es que se trata de un lenguaje compilado, en lugar de interpretado, como Java. Esto le hace más eficiente, pero menos portable. Visual C++ es la solución de Microsoft para la programación en C++ en entornos gráficos y utilizando las herramientas de este fabricante. Se trata por tanto de una solución de pago pero Microsoft, primer a través de la edición Express del compilador y, más recientemente, a través de la versión Community de Visual Studio, proporciona una herramienta gratuita y completa para entornos educativos y otros, como se puede ver en el sitio web.

El presente documento pretende ser una guía rápida para que un estudiante o titulado en ingeniería electrónica pueda superar la barrera que le impone la programación y desarrollar aplicaciones sencillas que le permitan explotar la funcionalidad del ordenador como herramienta electrónica en sí mismo, para intercambiar información con otros dispositivos y para ofrecer al usuario una interfaz potente y amigable con el conjunto del sistema. Aspectos no cubiertos de forma consciente en el documento son aquellos que permitirían al propio ordenador convertirse en una herramienta de control de procesos; básicamente todo lo que tiene que ver con el concepto de tiempo real.

El objetivo es que, con unos conocimientos muy básicos de programación, se puedan desarrollar aplicaciones sencillas en entorno gráfico en aproximadamente una semana de trabajo.

Finalmente cabe destacar que en la redacción de la guía se ha perseguido la exposición clara de las diferentes soluciones planteadas. Para ello habrá sido necesario en muchos casos penalizar la calidad del código en aspectos tales como el encapsulamiento o el tratamiento de excepciones. El usuario deberá ir buscando su camino hacia un código de calidad a medida que vaya adquiriendo conocimientos y experiencia.

CONCEPTOS PREVIOS

La programación orientada a objeto difiere sustancialmente del paradigma de programación tradicional. El concepto fundamental, por supuesto, es el de **objeto**. La programación basada en objetos u orientada a objetos como es más frecuente decir, se justifica en que es más próxima a la realidad. Los escenarios reales están integrados por múltiples objetos, muchos de ellos concretos, pero también pueden ser abstractos o inmateriales. Mi coche particular es un objeto que pertenece a una **clase** junto con otros coches similares. La clase es un concepto abstracto que sirve para representar y eventualmente crear objetos basados en ella. La clase a la que pertenece mi coche puede ser perfectamente la clase "coche", pero también podría ser la clase "vehículo a motor" o la clase "medio de transporte". Cada una de ellas es más general que la anterior, por lo que abarcarían a muchos objetos de naturaleza diferente. Por ejemplo, la clase "medio de transporte incluiría trenes, aviones, etc. Igual no es razonable para nuestra aplicación tratarlos de manera común. Pero es posible por otra parte que nuestra aplicación sí trate con otros vehículos o medios de transporte, por lo que habrá que llegar a una solución para todos ellos. Afortunadamente para eso existe el concepto de **herencia**. Podemos generar la clase "medio de transporte", para posteriormente crear la clase "vehículo a motor" como heredera de ella. Luego podemos crear la clase "coche" como heredera de la anterior. Cada clase hereda las características de la anterior y les añade las suyas propias.

Más que características en abstracto, lo que incluyen las clases son, por un lado las propiedades de los objetos que se deriven de ellas, y a las que llamamos **atributos**, y por otro lado, las operaciones que se pueden realizar sobre los propios objetos, llamadas **métodos**. Un atributo que podemos asignar a la clase "medio de transporte" es la "capacidad". Este atributo sigue teniendo sentido cuando lo heredan los "vehículos a motor" y los "coches" posteriormente. Un atributo que tiene sentido para los "vehículos a motor" y por lo tanto para los "cohes", pero no para los medios de transporte en general es la "cilindrada". Un atributo que tiene sentido solamente para los coches puede ser el "número de plazas en la fila trasera", por ejemplo, ya que no tiene sentido en motos y camiones. Una operación que mi coche me permite realizar es "arrancar", que puede venir heredada desde la clase "medios de transporte". Bajar la ventanilla trasera izquierda es un método que, ni los "medios de transporte" en general, ni otros "vehículos a motor" pueden implementar, por lo que se añadirá al crear la clase "coche" como heredera de "vehículo a motor".

Cuando yo decido "arrancar" mi coche, suceden una serie de eventos, implementables mediante nuevos métodos, de los que yo no acabo de ser consciente y sobre los que no tengo control. Estos métodos, como la puesta en marcha del motor de arranque, la regulación de la entrada de combustible y aire al motor etc, no son accesibles desde fuera del objeto. Se dice que se encuentran encapsulados. El **encapsulamiento** es una propiedad de la programación orientada a objeto que proporciona una robustez importante al código. No solo los métodos pueden encapsularse, sino también los atributos. De esta manera se impide que otras entidades del software puedan acceder a ellos de una manera incorrecta. Un atributo que está encapsulado en mi coche es el estado de encendido de las luces. Yo no puedo acceder directamente a un botón que las encienda y apague de una forma individual. El vehículo me ofrece para ello una interfaz controlada que me permite encender o apagar las luces de ambos

lados de forma conjunta, no individual. Esta interfaz no me permite encender las luces de cruce y de carretera de forma simultánea; me obliga a elegir entre unas u otras. De no ser así, mi coche podría circular en un momento dado con la luz de cruce izquierda y la de carretera derecha, mientras que en la parte posterior podría tener una luz encendida y otra apagada. Es evidente que no es bueno que el usuario pueda hacer eso, por lo que se le encapsulan los atributos correspondientes y se le permite acceder a ellos mediante un método (el mando de accionamiento de luces) que le permite un acceso indirecto, limitado y controlado.

El último concepto básico relacionado con la programación orientada a objeto, que nos resta es el de **polimorfismo**. Diremos que consiste en que un mismo objeto pueden presentar un comportamiento diferente en función de la forma en que se le invoque. No es probable que vayamos a necesitar hacer uso de esta propiedad al menos en estas etapas tempranas del trabajo, por lo que no insistiremos más en él.

PREPARACIÓN DEL ENTORNO DE TRABAJO

El proceso comienza con la instalación de Visual Studio Community 2015. Obviamente la versión profesional es válida también, junto con versiones anteriores de la edición Express o profesional. Las figuras que aparecerán a lo largo del documento pueden estar generadas mediante versiones diferentes. En la web de Microsoft podemos descargar esta potente herramienta. El proceso consiste en la descarga de la herramienta de instalación como se puede ver en la Figura 1.

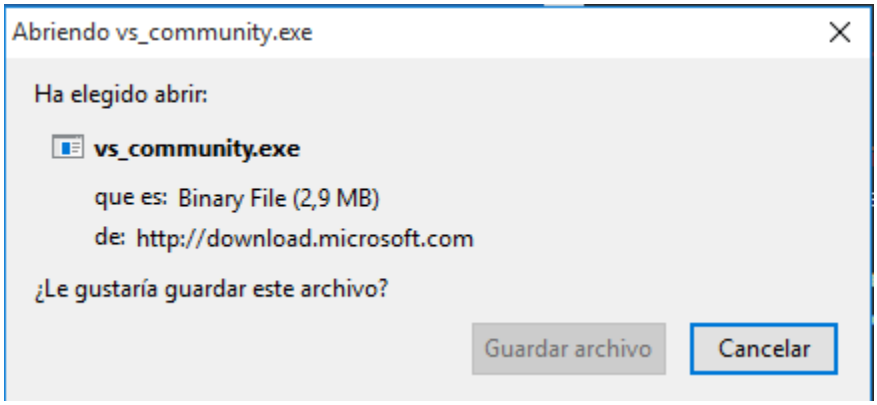


FIGURA 1

La instalación deja a elección del usuario el instalar la herramienta completa o solamente el entorno que se va a emplear, en este caso Visual C++. Ambas opciones son válidas para nuestro trabajo. El resto de opciones por defecto que aparecen durante la instalación son correctas, por lo que no daremos más detalles acerca de este proceso.

Una vez finalizada la instalación tendremos acceso al entorno de trabajo. Se trata de un trabajo orientado a la generación de proyectos, por lo que nuestro primer paso será la creación de uno nuevo (Figura 2).

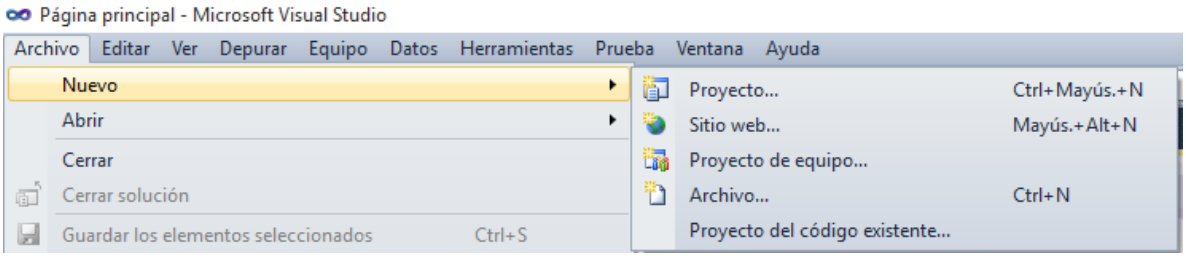


FIGURA 2

El entorno de programación nos proporciona 3 opciones para desarrollar proyectos en Visual C++:

- Proyecto Win32.
- Aplicación MFC.
- Aplicación de Windows Forms.

Las 3 opciones nos permiten alcanzar los resultados deseados, pero su nivel de abstracción es diferente. La opción que nos permite trabajar a un nivel más alto, es decir, la que nos proporciona una mayor ayuda para el desarrollo de aplicaciones gráficas es la tercera. Por este motivo, en la ventana que se abre a continuación seleccionaremos la opción "Aplicación de Windows Forms" y configuraremos el nombre y la ruta deseada para albergar el proyecto (Figura 3).

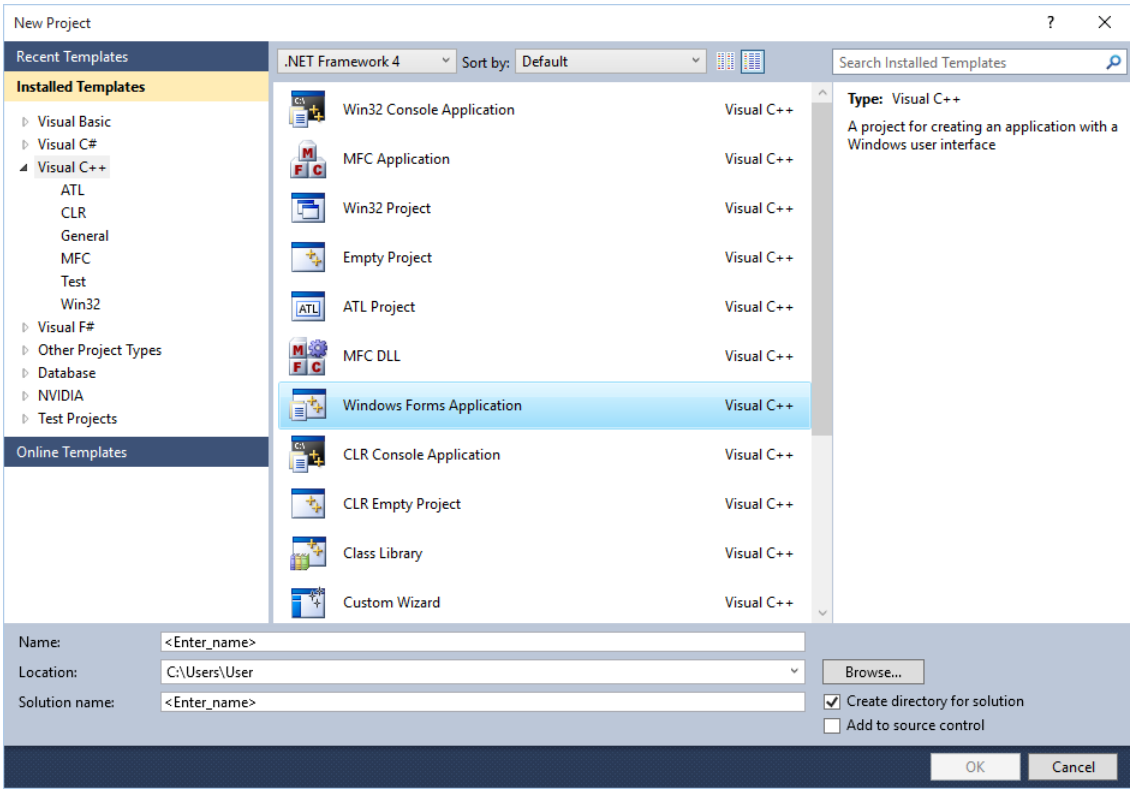


FIGURA 3

Inmediatamente se nos creará el esqueleto de una aplicación de Windows Forms típica (Figura 4). En ella se puede ver un primer formulario (Form1) y la estructura general del proyecto en la que se incluyen:

- Ficheros de código C++ (.cpp)
- Ficheros de cabecera (.h)
- Ficheros de recursos

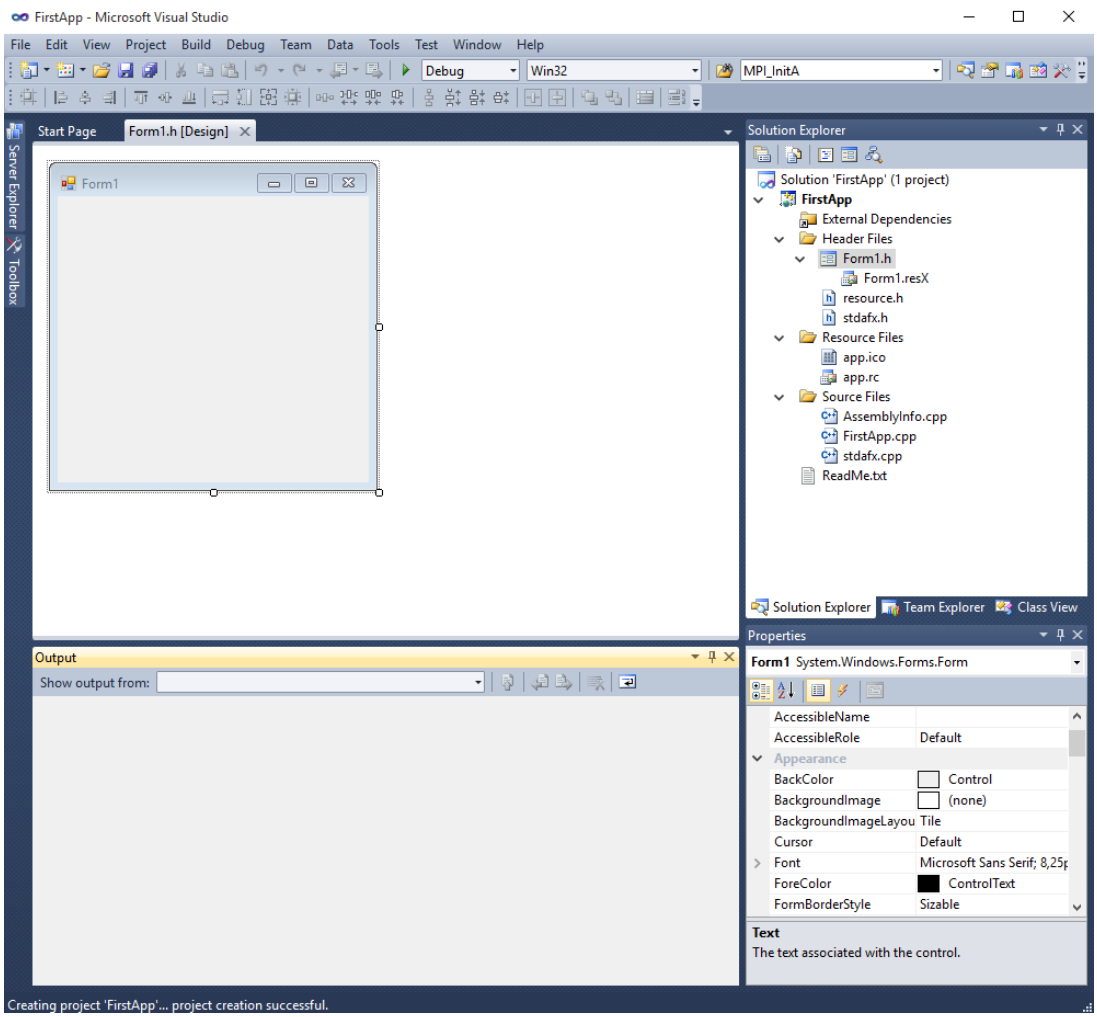


FIGURA 4

El fichero de cabecera “Form1.h” contiene la descripción de un primer formulario y la presenta de forma gráfica. Para trabajar sobre este formulario manteniendo la representación gráfica, necesitaremos la ventana “Cuadro de herramientas”, que podemos hacer visible a través del menú “Ver” (Figura 5).

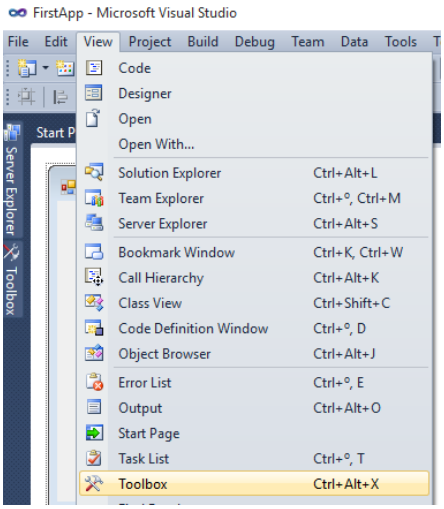


FIGURA 5

A modo ilustrativo vamos a colocar algunos controles en el formulario. Empezaremos arrastrando una etiqueta al área del formulario (Figura 6).

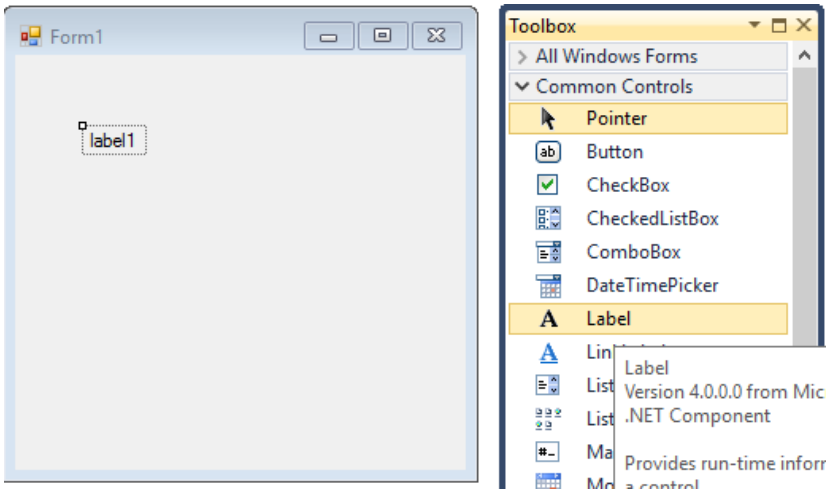


FIGURA 6

Siguiendo el mismo procedimiento, insertaremos otra etiqueta, dos botones y un cuadro de texto. La ventana de propiedades nos permite acceder tanto a las propiedades de los elementos introducidos como a las del propio formulario (Figura 7).

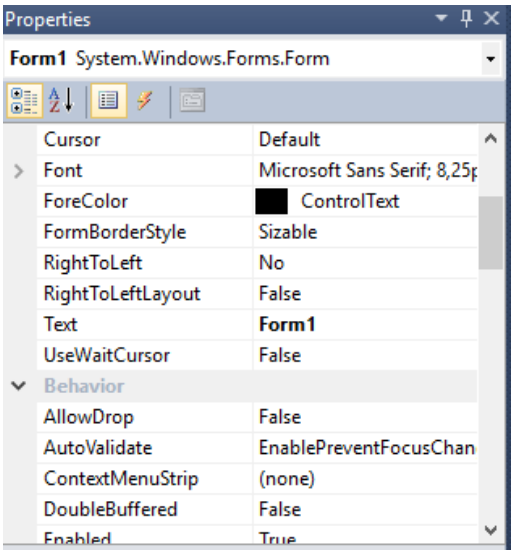


FIGURA 7

Las etiquetas son variables de texto a mostrar. Se les puede inicializar mediante la propiedad "Text" y posteriormente se puede modificar esta propiedad en tiempo de ejecución. En nuestro caso vamos a situar una de las etiquetas en la parte superior de la ventana y la vamos a inicializar al valor "Introduce texto". Inmediatamente debajo de ella situaremos el cuadro de texto cuya propiedad "Text" dejaremos vacía y debajo de él colocaremos la segunda etiqueta, también con la propiedad "Text" vacía. En este caso no se va a mostrar nada por lo que parecerá que la etiqueta no existe. Debajo del conjunto situaremos uno de los dos botones y ajustaremos su texto a "Aceptar". El botón restante lo posicionaremos en la esquina inferior derecha de la ventana y le daremos un valor "Salir" a su propiedad "Text". De esta forma, lo que veremos en la descripción de la ventana será lo que se muestra en la Figura 8.

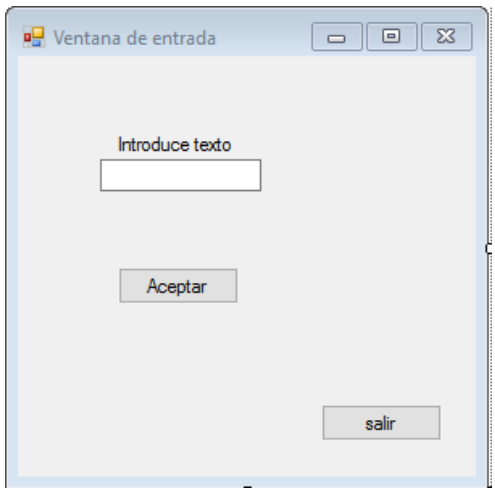


FIGURA 8

Todo parece indicar que le vamos a pedir al usuario que introduzca un texto en la caja y que vamos a realizar alguna operación con él cuando pulse el botón "Aceptar", sin embargo, lo que

vamos a hacer no está especificado aún. Para ello tendremos que realizar unas mínimas labores de programación. Un doble click en un botón nos abre una ventana de código en la que podemos programar la acción asociada a la pulsación sobre el mismo. En este caso, al pulsar el botón "Aceptar", queremos que se muestre el texto introducido en la hasta ahora invisible "label2". Para ello introduciremos la siguiente línea de código:

```
label2->Text=String::Format("Texto introducido: {0}", textBox1->Text);
```

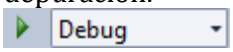
El significado de esta línea es bastante evidente: queremos que el contenido de la propiedad "Text" de "textBox1", es decir, lo que escriba el usuario en la caja de texto, sea mostrado en la etiqueta "label1". De esta forma, el método privado asociado al click del botón queda programado de la siguiente manera:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
        label2->Text=String::Format("Texto introducido: {0}", textBox1->Text);
 }
```

De forma similar, lo que pretendemos hacer con el segundo botón es salir de la aplicación, por lo que el método correspondiente quedará como:

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e)
{
        Application::Exit();
 }
```

Para poder disfrutar de esta primera aplicación tendremos que construir la solución mediante el menú "Generar -> Generar solución". Si nos encontramos bajo la configuración "Debug" podremos ejecutarla desde Visual Studio en modo depuración mediante el icono de depuración.



DESCRIPCIÓN DE LA APLICACIÓN

Hasta aquí, hemos conseguido llevar a cabo una pequeña aplicación gráfica sin esfuerzo aparente. Sin embargo, si editamos el código que hay detrás de "Form1.h", al que acabamos de acceder para añadir la pequeña funcionalidad de nuestras etiquetas y botones, veremos que nos resulta bastante extraño. Vamos a tratar de explicar cómo funciona para que sea luego más fácil construir sobre él. Para una referencia más completa sobre la forma de trabajar de Windows Forms, se recomienda visitar este enlace.

En primer lugar nos encontramos con la directiva: `#pragma once`. Esta directiva es habitual en los ficheros de cabecera en C++; sirve para indicar que lo que viene a continuación se ha de incluir una sola vez en la aplicación.

A continuación viene el código más relevante para la aplicación declarado dentro del espacio de nombres `namespace AplicacionWindowsForms`. Dentro de él lo primero es declarar los espacios de nombres que se van a utilizar, como: `using namespace System::Windows::Forms;`

Los espacios de nombre permiten especificar dónde se encuentran los elementos que vamos a referenciar: clases, funciones, variables, etc. Al declarar nuestro código como espacio de nombre, también será posible referenciarlo en otras unidades de software.

Tras las oportunas declaraciones de espacios de nombres utilizados, nos encontramos con una pieza de código como:

```
public ref class Form1 : public System::Windows::Forms::Form
	{
	public:
		Form1(void)
		{
			InitializeComponent();
			//
			//TODO: agregar código de constructor aquí
			//
		}

	protected:
		~Form1()
		{
			if (components)
			{
				delete components;
			}
		}
```

Se trata de la creación de la clase Form1 como heredera de `System::Windows::Forms::Form`. En ella se incluyen los métodos constructor y destructor: `Form1(void)` y `~Form1()`. En estos métodos, el primero declarado como público y el segundo como protegido, podemos incluir el código que deseemos, pero por defecto se incluye la inicialización y borrado de los componentes que vamos incluyendo.

Si introducimos un botón de salida, veremos cómo automáticamente se incluye su declaración como:

```
private: System::Windows::Forms::Button^  button1;
```

para luego crearlo e inicializarlo a partir del código generado por el diseñador (caja de herramientas):

```
void InitializeComponent(void)
              {
                     this->components = (gcnew System::ComponentModel::Container());
                     this->button1 = (gcnew System::Windows::Forms::Button());
                     this->SuspendLayout();
                     //
                     // button1
                     //
                     this->button1->Location = System::Drawing::Point(197, 226);
                     this->button1->Name = L"button1";
                     this->button1->Size = System::Drawing::Size(75, 23);
                     this->button1->TabIndex = 0;
                     this->button1->Text = L"salir";
                     this->button1->UseVisualStyleBackColor = true;
```

En este caso se crea el objeto y se establecen sus propiedades, entre las cuales se encuentra el texto “salir” que hemos podido introducir en la ventana de propiedades, pero que también podríamos establecer directamente en esta zona de código.

Tras esta zona de código, que no se debe alterar y que se encuentra delimitada por las directivas:

```
#pragma region Windows Form Designer generated code
/// Método necesario para admitir el Diseñador. No se puede modificar
/// el contenido del método con el editor de código.
……..
#pragma endregion
```

Aparecen la parte de código relacionada con los objetos y acciones que decidamos incluir, como es el caso de la pulsación del botón de salida:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
       Application::Exit();
}
```

Antes de finalizar este repaso de la estructura de la aplicación conviene señalar que el punto de partida no se encuentra en el archivo Form1.h, sino en el archivo de código con el mismo nombre que el proyecto, en nuestro caso: FirstApp.cpp. En él se encuentra el método “main” que es el origen de toda aplicación. En él se invoca el inicio de la misma como:

```
Application::Run(gcnew Form1());
```

INCORPORACIÓN DE EVENTOS GENERADOS POR EL RATÓN

El trabajo del ingeniero electrónico exige habitualmente la comunicación con dispositivos periféricos. Esto es una necesidad que a un programador estándar le llega habitualmente en un estadio más avanzado de su trabajo, pero aquí tenemos que contemplarla desde el principio. El periférico más habitual y que nos vamos a encontrar disponible en un PC es el ratón. Aparte de su utilidad en el control del ordenador, el ratón se utiliza en ocasiones en electrónica como entrada de información proveniente de dispositivos que simulan ser un ratón para que equipos externos o usuarios con características especiales puedan hacerse con el control de la máquina. Solemos llamar a estos dispositivos "emuladores de ratón". Veamos cómo podemos trabajar con ellos.

En la vista de diseño de Form1.h vamos a seleccionar la ventana completa. En la ventana de Propiedades, pulsaremos el icono de eventos  . En la larga lista que se abre disponemos de una sección denominada "Mouse", en la que tenemos acceso a los eventos generados por el ratón.

Si hacemos doble click sobre el evento "Mouse down", se abrirá su propia ventana de propiedades, al tiempo que en la parte final del código de Form1.h se crea un método para la gestión de este evento. Vamos a introducir la siguiente línea de código en él:

```
MessageBox::Show(L"Mouse down!");
```

Una vez compilada y ejecutada la aplicación veremos que al pulsar el botón izquierdo del ratón, aparece una nueva ventana señalándonos la acción que se ha llevado a cabo. También podremos comprobar que esto solo ocurre cuando la pulsación se realiza en la parte interior de la ventana de la aplicación, no en el encabezamiento ni en cualquier otra zona de la pantalla. El evento está asociado a Form1 exclusivamente. También es interesante observar que el mensaje se muestra al pulsar el botón del ratón que es precisamente el evento que se ha capturado. No se está realizando la acción "Click" por lo tanto, ya que ésta está formada por dos eventos: "MouseDown" + "MouseUp" (pulsación + liberación).

UNA PRIMERA APLICACIÓN PRÁCTICA

Vamos a llevar a cabo una aplicación que resuelva un problema concreto. Necesitaremos algunos conocimientos adicionales para llevarla a cabo, pero los iremos introduciendo por el camino. El enunciado sería el siguiente: vamos a crear un sencillo comunicador para una persona que no puede expresarse de manera oral y que además tiene dificultades de visión. Este comunicador le va a permitir responder sí o no a cualquier pregunta que se le haga. Para ello le vamos a proporcionar sendos botones que, para vencer sus dificultades visuales se harán de gran tamaño, tendrán colores de fondo llamativos y además, emitirán un sonido cuando el ratón los sobrevuele. De esta forma el usuario sabrá que está sobre el botón y de qué botón se trata. Si se corresponde con la respuesta que quiere dar, lo pulsará. Añadiremos también un botón de salida de la aplicación.

Empezaremos por crear el contexto mencionado, incorporando 3 botones al formulario. Modificaremos su propiedad "Text" para introducir los valores: "Sí", "No" y "salir" respectivamente. Colocaremos los botones "Sí" y "No" en posición central y en tamaño grande. El botón "salir" estará en la esquina inferior derecha. Para modificar el tamaño de letra podemos ajustar la propiedad "Font" de cada botón. También podemos ajustar el color de fondo mediante la propiedad "BackColor" como se ve en la Figura 9.

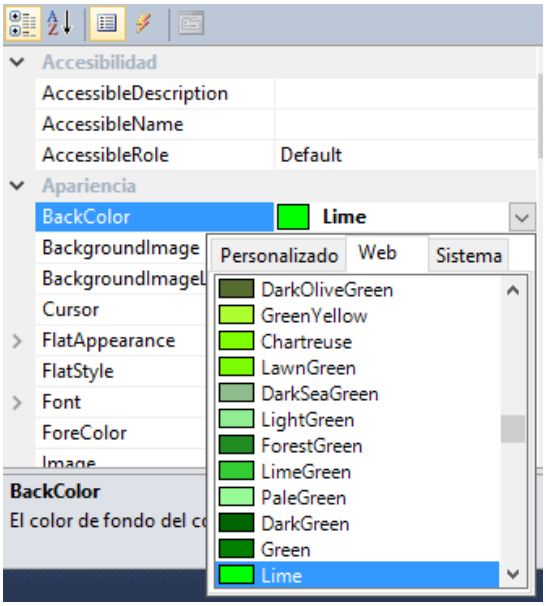


FIGURA 9

Modificamos estas propiedades de los botones y el título de la ventana para obtener una apariencia final como la que se muestra en la Figura 10.

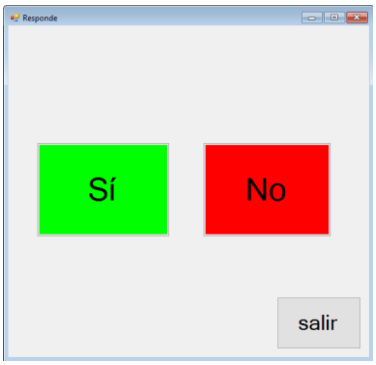


FIGURA 10

Una vez que tenemos los objetos, vamos a introducir las acciones. Éstas se van a producir como respuesta a eventos provocados por el ratón. Para poder activar los eventos asociados a cada botón, vamos a seleccionarlos y a pulsar el icono de eventos  . Para cada botón vamos a activar dos eventos:

- Button_MouseHover
- Button_Click.

El evento "MouseHover" nos va a permitir que el texto introducido en cada botón suene cuando el ratón lo sobrevuele. Así el usuario sabrá que está sobre el botón en cuestión. Si es el botón adecuado, el evento "Click" le permitirá confirmarlo y hará que el texto suene de nuevo como respuesta a la pregunta que verbalmente se le ha planteado al usuario.

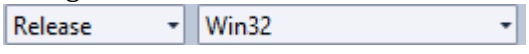
Al activar estos eventos se crearán los métodos correspondientes en el código de "Form1.h". Es tarea nuestra hacer que se conviertan en los sonidos correspondientes. El bloque de código que nos permite invocar un sonido es el siguiente:

```
System::Media::SoundPlayer^ player=gcnew System::Media::SoundPlayer();
player->SoundLocation = (".\\Si.wav");
player->Load();
player->PlaySync();
```

Donde "Si.wav" es un fichero de sonido que se habrá creado previamente con la pronunciación de la palabra "Sí" y que en este caso está en la misma ruta que la propia aplicación. Existen muchas herramientas para grabar sonido y almacenarlo en formato .wav. Una de ellas, gratuita es RecordPad. Con la herramienta crearemos los tres ficheros de sonido y para finalizar la aplicación, en el método correspondiente al click del botón "salir", en lugar de repetir este sonido, provocaremos la finalización de la aplicación:

```
Application::Exit();
```

Podemos depurar la aplicación y, cuando todo esté correcto, generar el ejecutable mediante la configuración "Release":



Se creará dentro del proyecto una carpeta denominada "Release" en la que encontraremos el ejecutable. Lo podemos llevar a la carpeta que nos apetezca pero debemos recordar que los ficheros de audio deben acompañarlo ya que en el código hemos establecido que su ruta sea la de la propia aplicación.

OTRAS PROPIEDADES DEL RATÓN

Vamos a terminar este recorrido por las posibilidades que nos da el ratón realizando una sencilla aplicación en la que capturemos y mostremos sus coordenadas en la pantalla. Esto nos puede servir para desarrollar aplicaciones más complejas que reaccionen a movimientos de un dispositivo externo.

Para ponerlo en práctica vamos a incluir en nuestra aplicación un botón de Inicio/Parada y dos etiquetas en las que se refleje la lectura de la posición del cursor y otras dos en las que se explique lo que representan las anteriores. Finalmente, un botón de salida. El aspecto de nuestra ventana será similar al que vemos en la Figura 11.

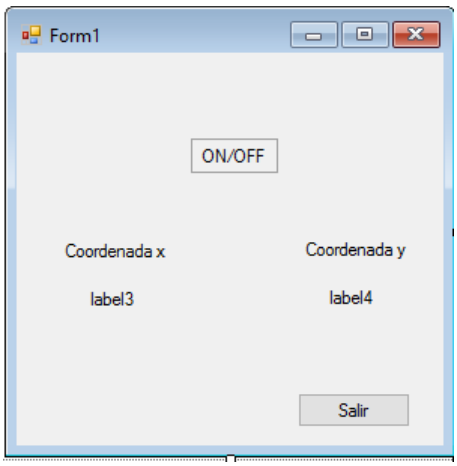


FIGURA 11

Una forma rápida de crear un botón de Inicio/Parada es utilizar un "CheckBox" y cambiar su apariencia a "Button". Ajustaremos su propiedad "Text" a "ON/OFF" y su propiedad FlatStyle a "Popup". De esta forma aparecerá como hundido cuando se pulse.

En la vista de diseño de Form1.h vamos a seleccionar la ventana completa. En la ventana de Propiedades, pulsaremos el icono de eventos  . En la lista de eventos generados por el ratón seleccionaremos "MouseMove".

Se nos creará un método vacío en el que nosotros podremos insertar el código necesario para que las coordenadas *x* e *y* de la posición del cursor se reflejen en las etiquetas "label3" y "label4" respectivamente. El código sería el siguiente:

```
private: System::Void Form1_MouseMove(System::Object^  sender,
System::Windows::Forms::MouseEventArgs^  e) {
                        int x=0, y=0;
                        x=e->X;
                        y=e->Y;
                        label3->Text = System::Convert::ToString(x);
                        label4->Text = System::Convert::ToString(y);
                }
```

Como ya hemos visto anteriormente cómo trabajar con el botón de salida, nos quedaría únicamente programar el botón de Inicio/Parada. Para ello bastaría con condicionar la actualización de las coordenadas del cursor en pantalla al estado del mencionado botón. Podemos conseguirlo modificando en código vinculado al evento de movimiento del ratón de la siguiente manera:

```
private: System::Void Form1_MouseMove(System::Object^  sender,
System::Windows::Forms::MouseEventArgs^  e) {
                    int x=0, y=0;
                    x=e->X;
                    y=e->Y;
                    if(checkBox1->Checked){
                    label3->Text = System::Convert::ToString(x);
                    label4->Text = System::Convert::ToString(y);
                    }
             }
```

COMUNICACIÓN SERIE

Al igual que el ratón, la mayoría de los dispositivos que se conectan al PC hoy en día lo hacen a través de un puerto serie. Para ilustrarlo vamos a centrarnos en el caso concreto de las placas Arduino. Vamos a ver cómo podemos interactuar con ellas desde Visual C++ utilizando el Framework .NET. Comenzaremos un proyecto nuevo para ello.

En este enlace encontramos una referencia inmejorable para el objetivo que tenemos en este caso, por lo que vamos a reproducir este proyecto muy aproximadamente. En la ventana principal vamos a introducir los siguientes elementos:

- Dos controles de cuadro combinado (ComboBox) que permiten visualizar una lista de opciones (ListBox) con un cuadro de texto. De esta forma el usuario puede seleccionar un valor de la lista o introducir uno nuevo. A estos controles les llamaremos Port y Baud respectivamente y los configuraremos la propiedad "DropDownStyle" como "DropDownList". En el cuadro Baud, mediante la opción, "Items (collection)" introduciremos una serie de valores estándar de velocidad de transmisión, entre los cuales estará el valor 9600 que es la que emplea Arduino por defecto.
- Junto a los controles anteriores ubicaremos sendas etiquetas con los textos: "COM Port" y "Baud rate" respectivamente.
- Dos botones a los que llamaremos "Abrir puerto" y "Cerrar puerto".
- Dos cuadros de texto para introducir los datos enviados y recibidos. En el cuadro de datos recibidos activaremos la propiedad "Read only".
- Dos botones a los que llamaremos "Enviar" y "Recibir".
- Un puerto serie que podemos encontrar como componente en la caja de herramientas. El puerto serie no aparecerá como icono dentro de la ventana, pero sí su código.
- Un botón denominado "Salir", para abandonar la aplicación.

Vamos a modificar su nombre por defecto "serialPort1" para pasar a llamarlo "arduino":

```
private: System::IO::Ports::SerialPort^  arduino;
.
.
this->arduino = (gcnew System::IO::Ports::SerialPort(this->components));
```

En este momento, nuestra ventana Form1, a la que hemos renombrado como "Comunicación serie" debería ser como la que aparece en la Figura 12.

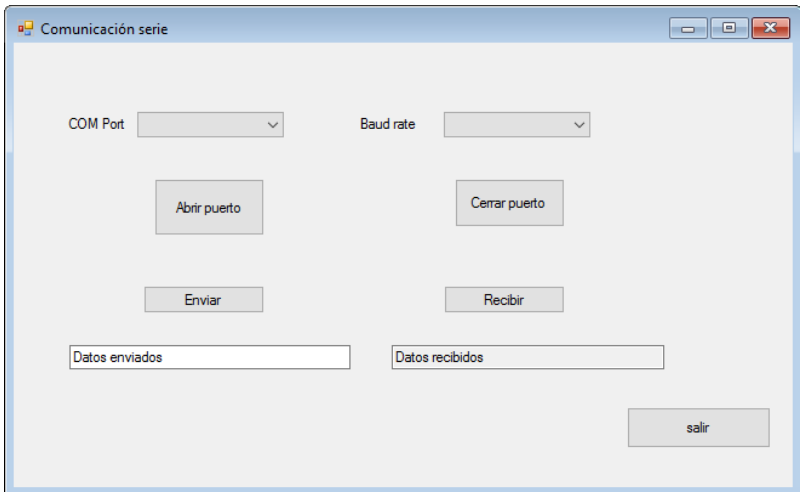


FIGURA 12

La primera operación que deberá realizar nuestra aplicación es localizar los puertos disponibles para que el usuario pueda seleccionar el adecuado. Para ello, dentro de "Initailize componentes", en el constructor de Form1, vamos a incluir una nueva función llamada "buscaPuertos()". La implementaremos en la zona de introducción de código (bajo #pragma endregion) con el siguiente código:

```
//Método de búsqueda de puertos disponibles
private: void buscaPuertos(void)
	{
	//Búsqueda de puertos disponibles mediante GetPortNames
	array<Object^>^objectArray = SerialPort::GetPortNames();
	//El array devuelto lo introducimos en el cuadro combinado
	this->Port->Items->AddRange(objectArray);
}
```

Si ejecutamos la aplicación veremos que la lista desplegable de puertos aparece inicialmente vacía. No obstante, si conectamos a continuación la placa Arduino, veremos que se añade el puerto COM en que se haya instalado. Podremos comprobar que coincide con el que nos indica el Arduino IDE so lo iniciamos.

Asociado al click del botón "Abrir puerto" introduciremos el siguiente código:

```
textBox1->Text=String::Empty;	//Vaciar el texto de la caja
textBox2->Text=String::Empty;	//Vaciar el texto de la caja
//Si no se ha seleccionado puerto o baudios...
if(Port->Text==String::Empty || Baud->Text==String::Empty)
	textBox1->Text="Por favor seleccione ajustes de puerto";
else{
	try{
		if(!arduino->IsOpen){
			arduino->PortName=Port->Text;
			arduino->BaudRate=Int32::Parse(Baud->Text);
			textBox1->Text="Introducir mensaje aquí";
			arduino->Open();
		}
		else
```

```
                        textBox1->Text="El puerto ya está abierto";
        }
        catch(UnauthorizedAccessException^){
                textBox1->Text="Acceso denegado";
        }
}
```

Si todo ha sido correcto el puerto quedará listo para la comunicación.

En respuesta al click del botón "Cerrar puerto" introduciremos:

```
arduino->Close();
```

En respuesta al click del botón "Enviar" introduciremos:

```
String^ mensaje = textBox1->Text;
if(arduino->IsOpen)
        arduino->WriteLine(mensaje);
else
        textBox2->Text = "El puerto no está abierto";
```

De esta forma se envía a la placa Arduino el contenido del cuadro de texto.

En respuesta al click del botón "Recibir" introduciremos:

```
if(arduino->IsOpen){
        textBox2->Text=String::Empty;      //Vaciar el texto de la caja
        try{
                textBox2->Text=arduino->ReadLine();
        }
        catch(TimeoutException^){
                textBox2->Text="Timeout";
        }
}
else
        textBox2->Text = "El puerto no está abierto";
```

Para evitar que la aplicación se quede en espera infinita a recibir datos o enviarlos, conviene ajustar las propiedades de Timeout de envío y recepción del puerto serie a un valor razonable. Para ello sobre-escribiremos el valor por defecto (-1) con por ejemplo 500 (milisegundos).

Es interesante conocer que la comunicación serie conlleva el ajuste de más parámetros que el nombre del puerto y la velocidad de transmisión. En la tabla siguiente vemos los ajustes por defecto del puerto serie que son coincidentes sin más modificación en Visual C++ y en Arduino.

Parámetro	Valor por defecto
Bits de datos	8
Handshake	Ninguno
DTR enable	No
Paridad	Sin paridad
Bits de parada	1

La aplicación realizada hasta ahora permite una comunicación serie con otro dispositivo, pero si tratamos de conectar con Arduino seguramente nos encontraremos con un problema en la recepción de datos: el "timeout" no deja de "saltar" sin que se reciba ninguna información. Jugando con el timeout podemos llegar a recibir datos, pero siempre será una operación insegura. El motivo es que, a diferencia del resto de acciones programadas, el usuario no tiene control sobre cuándo se produce la llegada de datos, sino que depende del dispositivo externo que los genera. Esto nos obligaría a estar en nuestro programa constantemente comprobando esa llegada, lo cual resulta muy ineficiente. Por otro lado, si la llegada es muy rápida, se podría llenar el buffer y perder información. La forma de solucionar estos problemas es recurrir a la concurrencia. Esto lo vemos en el siguiente apartado.

RECEPCIÓN DE DATOS CONCURRENTES

Cuando intentamos gestionar una comunicación en la que se incluye recepción de datos estamos situándonos en un entorno concurrente. Esto quiere decir que se dan dos procesos distintos de forma simultánea en el tiempo: el cuerpo de nuestra aplicación y la llegada de mensajes a través del puerto.

La forma adecuada de manejar estos eventos simultáneos es mediante el lanzamiento de hilos de ejecución concurrentes. Si por debajo disponemos de un hardware paralelo, como es un procesador multihilo, los hilos se ejecutarán de forma simultánea. Si no es así, lo harán de forma alternativa, pero es tarea del sistema operativo su planificación; el programador no se ha de ocupar de ello. Esta funcionalidad nos la facilita .net mediante el uso de "backgroundworker". Se trata de un hilo de ejecución que se lanza en segundo plano para descargar al hilo principal (la interfaz de usuario) de tareas pesadas que ralentizarían su ejecución. Podemos incluirlo a través del cuadro de herramientas. Automáticamente se generará su declaración, inicialización y el método por defecto "DoWork", en el que se incluirá el código a ejecutar por este hilo:

```
private: System::ComponentModel::BackgroundWorker^  backgroundWorker1;
this->backgroundWorker1 = (gcnew System::ComponentModel::BackgroundWorker());
private: System::Void backgroundWorker1_DoWork(System::Object^  sender,
System::ComponentModel::DoWorkEventArgs^  e) {
       }
```

Su uso en principio es simple: el evento que nos parezca adecuado, por ejemplo el click en el botón "Abrir puerto", lanza la ejecución del hilo:

```
backgroundWorker1->RunWorkerAsync();
```

Es posible pasarle un argumento en el paréntesis, pero en nuestro caso no va a ser necesario. Lo que sí va a ser necesario es establecer una vía segura para pasar los datos leídos por el puerto desde el hilo secundario al principal. El acceso a la caja de texto directamente desde el hilo de lectura del puerto no lo es. Para ello en primer lugar vamos a declarar un delegado que regule el acceso asíncrono a la caja de texto:

```
delegate void SetTextDelegate(String^ texto); //Delegado para acceso seguro a
textBox2
```

Para gestionar correctamente la impresión del mensaje recibido vamos a modificar la respuesta al click en el botón "Recibir" de manera que éste simplemente nos sirva para borrar la caja de texto:

```
textBox2->Text=String::Empty;
```

El trabajo del hilo de ejecución consistirá en estar permanentemente leyendo la información del puerto serie e imprimiéndola en la caja de texto, pero de manera segura:

```
private: System::Void backgroundWorker1_DoWork(System::Object^  sender,
System::ComponentModel::DoWorkEventArgs^  e) {
       while (true)
       {
         try{
              tempVal = arduino->ReadLine();
              this->SetText(tempVal); //Llama al método de escritura segura en
textBox2
              arduino->DiscardInBuffer();
         }
         catch(TimeoutException^){
         }
       }
}
```

Este método provoca que el hilo de ejecución se encuentre constantemente examinando el puerto, por lo que su uso de CPU es máximo. Si la cantidad de eventos que se prevén en el puerto es baja podemos intercalar una llamada a `Sleep(n)`, para detener la ejecución durante "n" milisegundos antes de intentar una nueva lectura.

El método de escritura segura es:

```
private: void SetText(String^ texto)
        {
            if (this->textBox2->InvokeRequired)
            {
                SetTextDelegate^ d = gcnew SetTextDelegate(this, &Form1::SetText);
                this->Invoke(d, gcnew array<Object^> { texto });
            }
            else
            {
                this->textBox2->Text = texto;
            }
        }
```

La aplicación está terminada. Simplemente nos quedaría probarla conectando una placa Arduino UNO. Utilizaremos el siguiente sketch:

```
String palabra;
void setup () {
   Serial.begin(9600);
   while(!Serial) { ; }
}
void loop () {
   if( Serial.available()> 0) {
      palabra=Serial.readString();
      Serial.print(palabra);
   }
   delay(20);
}
```

Se trata de un ejemplo muy sencillo en el que Arduino se hace eco del mensaje que le llega por el puerto serie y se lo reenvía al PC. A pesar de su simplicidad, el ejemplo merece una discusión. Como ya se ha comentado en el caso de la interfaz de usuario, la recepción de datos por un puerto siempre implica un desafío, al tratarse de un evento asíncrono. Los microcontroladores y sistemas embebidos derivados de ellos como Arduino, no son multihilo, por lo que la solución propuesta para el PC no es aplicable. Este tipo de entornos se valen habitualmente de interrupciones asociadas a eventos en el puerto serie para abordar el problema.

Una interrupción asociada a la recepción de un dato por el puerto detendría la ejecución normal del programa, llevándolo a una rutina de servicio predefinida que, una vez concluida devuelve el control al lugar en que se interrumpió. De esta manera, si llega un dato por el puerto, la rutina de servicio lo podría copiar en un elemento de memoria para luego ser tratado por el cuerpo principal del programa.

Arduino no dispone de interrupciones asociadas al puerto serie, por lo que es inevitable acudir a un procedimiento de muestreo para la recepción de datos. Esto implica explorar constantemente la entrada el puerto. El problema es que si se hace demasiado a menudo, el uso de CPU es excesivo; si la situación es la contraria, el buffer en el que se almacenan los datos que llegan por el puerto se podría desbordar con la consiguiente pérdida de información. La función Serial.available() viene a suavizar la situación, ya que permite saber si hay datos pendientes en el puerto antes de proceder a una costosa lectura que podría ser innecesaria. Conviene aclarar que los datos leídos son automáticamente eliminados del buffer para dejar hueco a otros nuevos. El buffer funciona como una pila FIFO en este sentido.

La llamada a “delay(20)” simplemente pretende limitar el uso de CPU ya que no se prevé una llegada continua de datos por el puerto.

Es habitual que la comunicación serie produzca resultados inesperados por problemas de formato de los datos enviados y recibidos. Se recomienda siempre consultar el funcionamiento de las funciones de envío y recepción tanto en C++ como en Arduino para comprobar que los datos se envían en el formato esperado por el receptor y que no se añaden elementos no deseados como retornos de carro, etc.

ENTRADA / SALIDA DE DATOS

Hasta ahora hemos visto cómo comunicarnos desde la interfaz de usuario con un dispositivo conectado al puerto serie/USB como es el caso de Arduino. El programa ejemplo utilizado, en el que Arduino simplemente se hace eco de la información que le llega, aunque simple, ya nos ha dado pie a una pequeña discusión de cómo debe ser la relación de los diferentes tipos de sistemas con la comunicación serie. Recordar que el origen del problema es que la llegada de información es asíncrona con el resto del funcionamiento del sistema.

En este apartado vamos a tratar el intercambio de información significativa entre dispositivos. Seguiremos comunicando vía serie el PC con Arduino UNO. Para poder trabajar con información real, vamos a dotar a Arduino de una shield comercial con múltiples entradas y salidas, tanto analógicas como digitales. Se trata de la "Arduino Basic I/O" de Microsystems Engineering que vemos en la Figura 13.

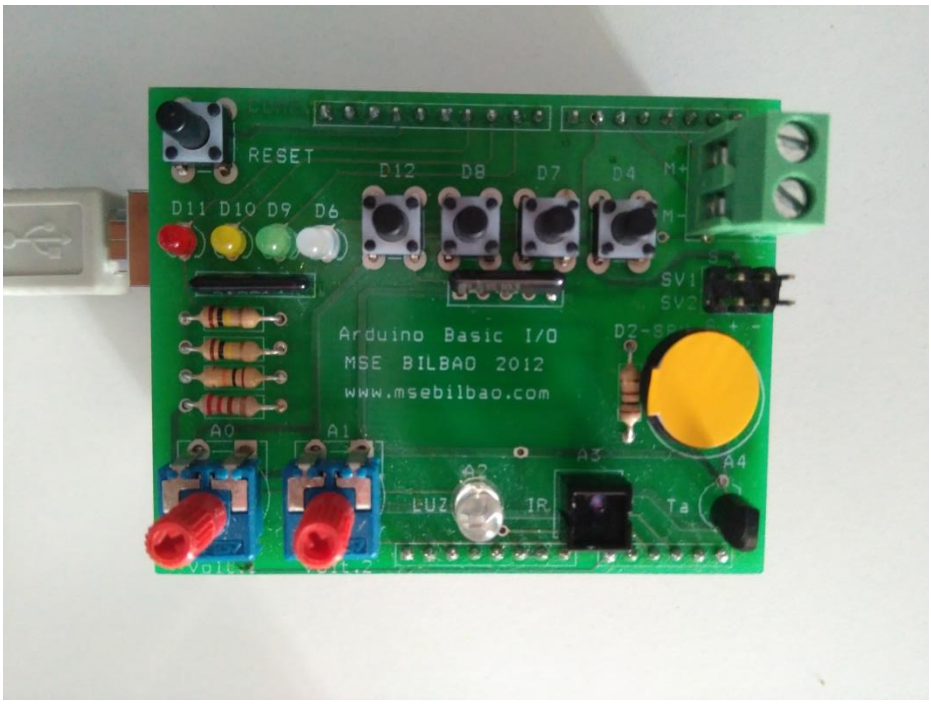


FIGURA 13

Vamos a tratar de dotar al usuario del PC del acceso a las diferentes señales de la placa. Cada tipo de señales presenta una problemática distinta, por lo que vamos a abordarlas por separado. La interfaz básica que ya teníamos diseñada y en funcionamiento, la vamos a mantener, ya que nos permite comprobar qué recibe Arduino e incluso introducir información en formato texto a través de la caja de envío.

SALIDAS DIGITALES

Vamos a empezar con este tipo de señales ya que son las más fáciles de manejar. Las salidas digitales nos permiten activar/desactivar elementos o dispositivos externos. Presentan dos estados que podemos llamar ON/OFF.

El objetivo de nuestra aplicación, en lo que afecta a las salidas digitales, será permitir al usuario enviar la orden de encendido/apagado a cada una de las salidas disponibles. En nuestra placa de trabajo disponemos de 4 salidas conectadas a otros tantos diodos LED de diferentes colores y conectadas a los pines 6, 9, 10 y 11 de Arduino.

Para poder gobernar su estado de una manera intuitiva vamos a proporcionar al usuario unos controles simples que permitan encender y apagar los LEDs. El control que hemos seleccionado para ello es la denominada "CheckBox" que podemos encontrar en el cuadro de herramientas de Visual Studio. Su apariencia por defecto no es la más adecuada de manera que la cambiaremos a través de su propiedad "Appearance" a la forma de botón como se muestra en la Figura 14.

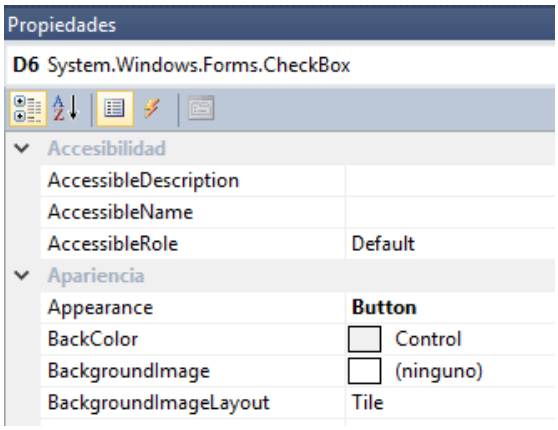


FIGURA 14

Su comportamiento va a ser el de un pulsador con enclavamiento que nos mantiene estable tanto la posición de encendido como la de apagado. Para hacerlo más intuitivo podemos hacer que el texto se corresponda con la salida digital que gobierna que se puede ver también en la serigrafía de la placa junto a cada diodo: D11, D10, D9, D6.

Si hacemos doble click en el botón, se abrirá el método por defecto, que responde al evento de un cambio en el estado de este control. Así, cada vez que lo activemos o desactivemos, se ejecutará este método. Veamos entonces el código que vamos a introducir como respuesta al click:

```
private: System::Void D6_CheckedChanged(System::Object^  sender, System::EventArgs^
e) {
       if(arduino->IsOpen){
         if ( D6 ->Checked ){
              arduino->WriteLine("D6ON");
              D6->BackColor=Color::White;
         }
         else{
```

```
                arduino->WriteLine("D6OFF");
                D6->BackColor=BackColor;
            }
        }
        else
                textBox2->Text = "El puerto no está abierto";
}
```

Fundamentalmente lo que se hace cada vez que se pulsa el control es, previa comprobación de que el puerto está abierto, distinguir si se ha seleccionado (que va a equivaler al encendido) o deseleccionado el control (equivalente a apagado). En caso de encendido se envía a Arduino el mensaje que ordena encender la salida correspondiente; en caso contrario se envía el mensaje de apagado. Localmente se cambia el color de fondo del control cuando se encuentra seleccionado para que coincida con el del diodo que se ha ordenado encender. La Figura 15 muestra el resultado con dos de los diodos encendidos.

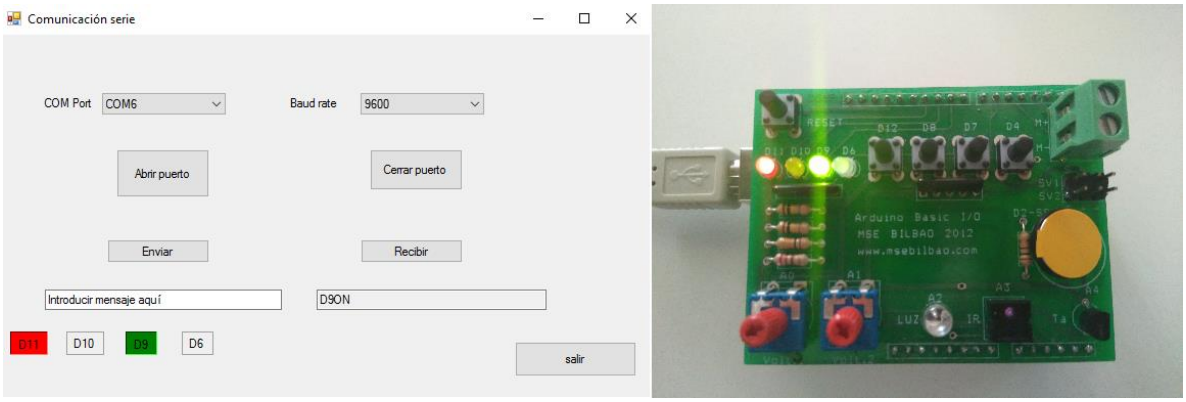


FIGURA 15

En el lado de Arduino, vamos a mantener la funcionalidad básica de la aplicación que veníamos utilizando, es decir la transmisión del eco del puerto serie y vamos a añadir el código necesario para identificar y ejecutar las órdenes que llegan del PC. El código resultante es el siguiente:

```
String palabra;
void setup () {
    //Salidas digitales en la shield
    pinMode(10, OUTPUT);
    pinMode(11, OUTPUT);
    pinMode(9, OUTPUT);
    pinMode(6, OUTPUT);

   Serial.begin(9600);
   while(!Serial) { ; }
}
void loop () {
   if( Serial.available()> 0) {
      palabra=Serial.readString();
      Serial.print(palabra);
```

```
      //Activación de salidas digitales
      if(palabra == "D11ON\n")
       digitalWrite(11, HIGH);
      else if(palabra == "D11OFF\n")
       digitalWrite(11, LOW);
       else if(palabra == "D10ON\n")
        digitalWrite(10, HIGH);
        else if(palabra == "D10OFF\n")
         digitalWrite(10, LOW);
         else if(palabra == "D9ON\n")
          digitalWrite(9, HIGH);
          else if(palabra == "D9OFF\n")
           digitalWrite(9, LOW);
           else if(palabra == "D6ON\n")
            digitalWrite(6, HIGH);
            else if(palabra == "D6OFF\n")
             digitalWrite(6, LOW);
  }
  delay(20);
}
```

En la función setup() damos de alta las salidas digitales de la placa. En el bucle principal se va comparando la información que llega por el puerto con las órdenes que hemos acordado que sirven para encender y apagar cada dispositivo.

Una vez puesto en funcionamiento observamos que las órdenes se transmiten y ejecutan sin ningún problema. A pesar de ello notamos que existe un cierto retardo entre la pulsación del ratón sobre el botón en pantalla y el encendido/apagado del diodo. Hay varios factores que intervienen en este retardo; el más evidente es el retardo que voluntariamente estamos introduciendo al final del bucle en Arduino con la línea "delay(20)". Si eliminamos esta línea el retardo será menor, pero seguirá existiendo. En todo caso hay que tener en cuenta que este retardo viene a representar el código que Arduino ejecutaría en una situación real. Tenemos que pensar que, en esta gestión muestreada del puerto, Arduino puede llegar a tardar en adquirir el mensaje entrante tanto como la duración de su ciclo de programa completo. Un fenómeno que podemos observar si prestamos atención a cómo se produce el encendido del diodo y el retorno del mensaje al PC, es que ambos se producen de forma prácticamente simultánea. Esto implica que, a la escala que podemos apreciar con la vista, fenómenos como el retardo de propagación o la velocidad de transmisión son poco importantes. En el caso del mensaje de retorno, Arduino lo envía de forma inmediata ya que se encuentra justo detrás de la lectura del puerto en el código. Se recomienda probar diferentes valores de retardos y ubicaciones del envío y recepción por el puerto para entender mejor cómo influye cada aspecto. En cualquier caso lo más importante es darse cuenta de la existencia de estos tiempos y las consecuencias que pueden tener en algunas aplicaciones de control de procesos o de seguridad.

SALIDAS ANALÓGICAS

Las salidas analógicas, a diferencia de las digitales, van a tener valores cualesquiera dentro de un rango. En el caso de Arduino UNO las salidas analógicas están implementadas como PWM (modulación de anchura de impulsos). Esto implica que la salida toma valores de cero o de 5V. La relación entre el tiempo que se encuentra a nivel bajo y a nivel alto determina el valor medio de la señal que es el que se considera asociado a la salida analógica. Este valor se determina por programa mediante un entero que puede variar de 0 (señal nula) a 255 (señal continua de 5V). En caso de ser necesaria una verdadera señal analógica, se deberá generar mediante hardware externo. Una interesante discusión al respecto se puede encontrar aquí. La frecuencia de la señal pulsatoria según el fabricante es de "aproximadamente" 980 Hz.

En nuestra placa de prueba podemos utilizar por ejemplo la salida digital 6, que corresponde al LED de color blanco, como salida analógica y regular de esta manera su luminosidad. A los LED este tipo de excitación les va como anillo al dedo y, a la mencionada frecuencia de señal, nuestro ojo no nota el parpadeo.

Conceptualmente, el trabajo con salidas analógicas no difiere del uso de salidas digitales, especialmente cuando se trata de estas salidas "simuladas" mediante PWM ya que no vamos a plantearnos cuestiones como la resolución o el tiempo de conversión.

En primer lugar vamos a seleccionar un control en Windows Forms que nos permita establecer el valor de la salida. Lo más apropiado que encontramos es el denominado "TrackBar" que nos permite desplazar un cursor a lo largo de una barra adquiriendo valores comprendidos entre dos propiedades del objeto:

- "Minimum": que por defecto es cero y así lo dejaremos.
- "Maximum": que ajustaremos a 255.

El evento por defecto cuya gestión se abre si hacemos doble click en el control es el correspondiente a la acción "ValueChanged". Parece lógico aprovecharlo para informar a la placa Arduino de que debe cambiar el valor de la salida. Sin embargo esto no va a funcionar correctamente ya que el evento se lanza continuamente mientras estamos desplazando el cursos por la barra, no al dar por finalizado el recorrido. De esta forma se generan múltiples eventos de comunicación que llegan a saturar el buffer de recepción de Arduino. Utilizaremos en su lugar el evento "MouseUp", que se lanzará cuando liberemos el ratón dentro del control. Esto es lo que hacemos al terminar de arrastrar el cursos por la barra; lo ideal para poder informar solamente del valor final. Haremos doble click en este evento, como se ve en la Figura 16, para que sea incorporado el código correspondiente al método que lo gestiona.

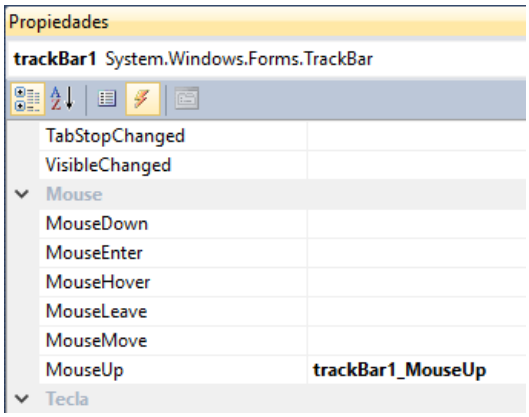


FIGURA 16

El método quedará programado de la siguiente manera:

```
private: System::Void trackBar1_MouseUp(System::Object^  sender,
System::Windows::Forms::MouseEventArgs^  e) {
       String^ a;
       a = trackBar1->Value.ToString(); //Transforma la posición del cursor en
string
       a = "A" +a; //Se añade al principio del string la letra A (salida analógica)
       arduino->WriteLine(a);
}
```

En la interfaz sustituimos el botón de encendido/apagado D6 por el nuevo control. El resultado se muestra en la Figura 17.

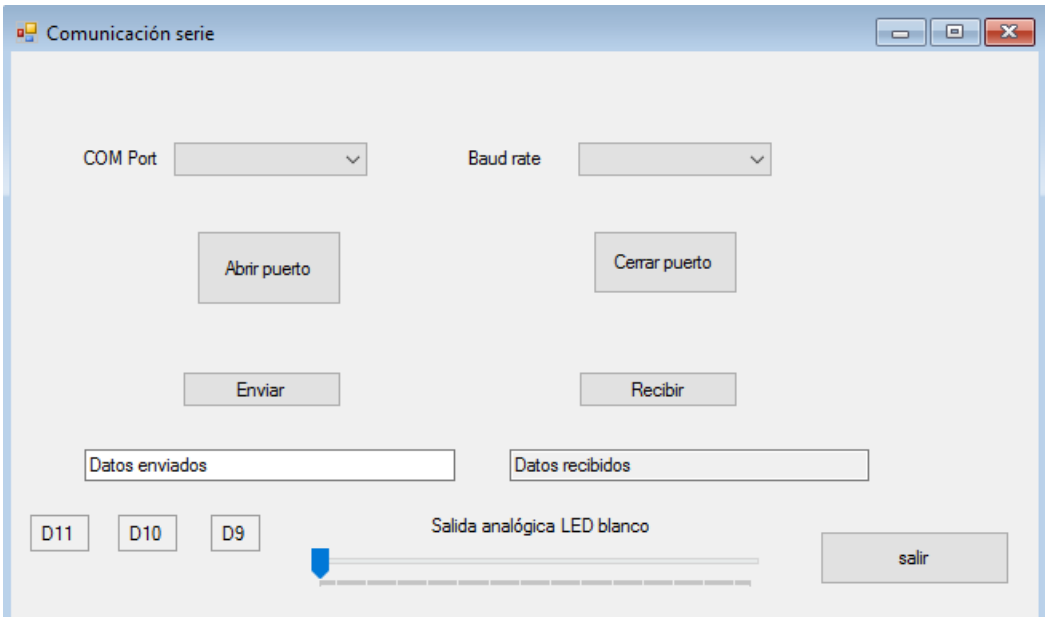


FIGURA 17

Arduino deberá establecer que esta información, al empezar por el carácter “A” hace referencia a la salida analógica. A continuación deberá transformar el resto de la cadena en valor entero para lanzarlo por la salida 6 (el diodo blanco). Se ha modificado el código de Arduino para que en lugar de esperar una orden de encendido/apagado en la salida 6, compruebe si se ha recibido el valor de la salida analógica. El código queda como sigue:

```
String palabra;
String valor;
int valorEntero;
void setup () {
    //Salidas digitales en la shield
    pinMode(10, OUTPUT);
    pinMode(11, OUTPUT);
    pinMode(9, OUTPUT);
    pinMode(6, OUTPUT);

   Serial.begin(9600);
   while(!Serial) { ; }
}
void loop () {
   if( Serial.available()> 0) {
       palabra=Serial.readString();
       Serial.print(palabra);
       //Activación de salidas digitales
       if(palabra == "D11ON\n")
        digitalWrite(11, HIGH);
       else if(palabra == "D11OFF\n")
        digitalWrite(11, LOW);
        else if(palabra == "D10ON\n")
          digitalWrite(10, HIGH);
          else if(palabra == "D10OFF\n")
            digitalWrite(10, LOW);
            else if(palabra == "D9ON\n")
              digitalWrite(9, HIGH);
              else if(palabra == "D9OFF\n")
                digitalWrite(9, LOW);

       //Activación de salida analógica
        if (palabra.charAt(0) == 'A'){ //Comprueba el código de salida analógica
         valor = palabra.substring(1); //Extrae el valor numérico
         valorEntero = valor.toInt();// Lo convierte a entero
         analogWrite(6,valorEntero);
        }

   }
   delay(20);
}
```

ENTRADAS DIGITALES

Con este tipo de señales comienza la verdadera complejidad en el procesamiento de información. Al igual que suceden en el caso de la comunicación serie, las dificultades se encuentran en la llegada de datos, ya que se produce de forma asíncrona al programa que se está ejecutando.

Recordar que en el caso de la comunicación serie contemplábamos tres formas de gestionar esta cuestión:

- Lanzamiento de hilos de ejecución concurrentes con la misión de "escuchar" la llegada de información. Una opción viable para entornos multihilo como los PCs pero no para dispositivos embebidos monohilo como Arduino.
- Uso de interrupciones que derivan el flujo de programa a rutinas de servicio cuando se produce la llegada de datos. Opción habitualmente disponible en sistemas embebidos pero no en Arduino.
- Muestreo del puerto para comprobar periódicamente la posible llegada de información. Se trata de una opción que genera el riesgo de perder información si el muestreo es poco frecuente o sobreutilizar la CPU.

A diferencia de lo que sucedía con la comunicación serie, en Arduino sí tenemos la posibilidad de configurar interrupciones asociadas a las entradas digitales (también a las analógicas, por cierto). Lamentablemente no disponemos de tantas interrupciones como entradas, por lo que alguna tendrá que ser muestreada. Abundando en esta situación, Arduino UNO dispone de interrupciones asociadas a los pines 2 y 3, que no coinciden con las entradas digitales disponibles en nuestra placa de prueba. Nos vemos obligados por tanto a muestrear todas las señales en nuestra aplicación.

El estado de estas señales se enviará como es lógico al PC, que las mostrará en la interfaz de usuario. Utilizaremos dos de estas señales para generar órdenes hacia sendas salidas, concretamente vamos a hacer que los pulsadores asociados a las entradas D12 y D8 activen las salidas D11 y D10 (led rojo y naranja respectivamente). En este momento se abre una interesante discusión. ¿Se encargará Arduino de ejecutar las órdenes y el PC será simplemente informado? ¿o se enviará la información al PC para que él sea quien interprete su significado y la convierta en la orden correspondiente? A favor de la primera opción se encuentra un argumento fundamental: el control lo debe ejecutar un dispositivo de tiempo real, es decir, que tenga tiempos de respuesta acotados y Arduino lo es, ya que tiene la posibilidad de configurar un watchdog que lo garantice. A favor de la segunda opción está el hecho de descargar a un dispositivo de potencia limitada, como es Arduino, del procesamiento de la información. Dicho de otra forma, la decisión está vinculada a la interpretación del concepto de "tiempo real".

En el ámbito de control de procesos, el concepto de tiempo real va íntimamente ligado a garantizar tiempos de respuesta acotados. En procesamiento de señal, va más vinculado a una elevada capacidad de procesamiento. En nuestro caso no estamos condicionados por una situación real concreta y, como tenemos la posibilidad, pues vamos a implementar ambas opciones. Así, la funcionalidad de nuestros 4 diodos LED va a quedar como sigue:

- El LED rojo será operado exclusivamente desde Arduino; el PC será informado para que pueda reflejar su estado en la interfaz.

- El LED naranja será operado por el PC, bien al recibir una pulsación desde D8 o bien al ser accionado el botón correspondiente en la interfaz de usuario. Dado que el funcionamiento del control en el PC es biestable y el de la shield es monoestable, tendremos que unificarlos.
- El LED verde será operado exclusivamente por el PC.
- El LED blanco quedará como salida analógica.

Otra decisión que deberemos tomar es cómo interpretamos las pulsaciones en las entradas digitales. Mecánicamente la shield lleva instalados pulsadores, los cuales tienen un único estado estable, el nivel bajo. Por software podemos mantener esa funcionalidad, con lo que el LED correspondiente se apagará nada más soltar el pulsador, o convertirlos en biestables, cambiando el estado del LED cada vez que se produzca una pulsación. Teniendo en cuenta nuestra comunicación con el PC nos podemos plantear en este caso, si la conversión se realiza en Arduino o en el propio PC.

Como disponemos de 4 entradas digitales vamos a ir combinando estas opciones para ver cómo llevarlas a cabo y las ventajas/inconvenientes de cada una.

En primer lugar, vamos a preparar la interfaz de usuario para albergar los nuevos elementos. Se trata de 4 pulsadores, para los cuales vamos a seleccionar el control de CheckBox habitual. La idea es que no sea el usuario el que modifique su estado, sino el programa, por lo que cambiaremos su propiedad "Enable" a "False".

Esto es muy sencillo y rápido de hacer, pero en el interior del código, para operar estos controles, debemos ver cómo interpretar la información que llega por el puerto correctamente. Debemos codificar las órdenes de entrada al PC de la misma forma que el PC codifica las que manda a Arduino. La interpretación se podría hacer en el hilo que atiende la escucha del puerto, pero no lo vamos a hacer así ya que este hilo tiene una misión muy concreta a la que debe dedicarse con devoción. Como viene siendo habitual, vamos a dejar que la información entrante se traslade a la caja de texto para ser allí analizada por el hilo principal. A continuación vemos cómo queda el método incluyendo un primer control; el resto sería análogo:

```
private: System::Void textBox2_TextChanged(System::Object^  sender,
System::EventArgs^  e) {
       if(textBox2->Text =="D12ON")
              this->D12->BackColor=Color::Black;
       else if(textBox2->Text == "D12OFF")
              this->D12->BackColor=BackColor;
 }
```

Nos surge ya una primera duda. ¿Actualizamos ya el estado de D11 (LED rojo), dado que sabemos que Arduino lo habrá hecho ya en respuesta a la activación de la entrada digital que nos reporta? Se podría hacer, pero vamos a preferir no hacerlo. Por dos motivos principales:

- Queremos asegurarnos de que la orden se ejecuta. Si la programación no es correcta en Arduino en algún caso, estaríamos informando de forma errónea al usuario.
- En previsión de que se pueda estar haciendo en Arduino una conversión del pulsador a biestable por software, no vamos a asumir que la pulsación se ejecuta tal cual es efectuada por el usuario.

Esto nos lleva a añadir otro "trozo" de código análogo al anterior para D11. Además, deberemos deshabilitar el botón D11 ya que hemos decidido que el usuario no va a poder actuar sobre el LED desde la interfaz.

En el lado de Arduino, vamos a comprobar el estado del pulsador y actuar en consecuencia, esto es, encendiendo o apagando el LED e informando al PC. A continuación se plantean dos propuestas de solución para ello:

```
if(digitalRead(12)== HIGH){
 Serial.print("D12ON\n");
 digitalWrite(11, HIGH);
 Serial.print("D11ON\n");
}
else{
 Serial.print("D12OFF\n");
 digitalWrite(11, LOW);
 Serial.print("D11OFF\n");
}
```

```
if(digitalRead(12)== HIGH && testigo12 == false){
    Serial.print("D12ON\n");
    digitalWrite(11, HIGH);
    delay(10);
    Serial.print("D11ON\n");
    testigo12 = true;
    delay(10);
   }
else if(digitalRead(12)== LOW && testigo12 == true){
    Serial.print("D12OFF\n");
    digitalWrite(11, LOW);
    delay(10);
    Serial.print("D11OFF\n");
    testigo12 = false;
    delay(10);
   }
```

A la izquierda tenemos una solución simple y, que si la probamos directamente veremos que funciona. ¿Por qué complicarla entonces con la propuesta de la derecha? Hay dos motivos:

- La propuesta de la izquierda transmite en cada ciclo de programa el estado de la entrada digital. Si ésta no cambia, cosa que sucede el 99% de las ocasiones, se está haciendo un uso intensivo del puerto sin aportar ninguna información relevante. Unas pocas líneas de código permiten optimizar el uso del puerto enormemente. La gestión de "testigo12" (que se declara como global y se inicializa a "false" en la función (setup()) hace que solamente se envíe información cuando cambia el estado de la entrada digital 12.
- Si a continuación enviamos el estado de otras señales, aunque la comunicación y la velocidad a la que el hilo secundario del PC lee el puerto sean suficientes, nuestra caja de texto se convierte en un cuello de botella, haciendo que las informaciones se atropellen y el PC pierda datos. Para evitarlo, introducimos un pequeño retardo después de cada envío. Si la velocidad de ejecución del programa fuera un problema, tendríamos que plantearnos actualizar la información en el PC de otro modo, por ejemplo, introduciendo un buffer o empleando más cajas de texto.

Dado que el objetivo es seguir añadiendo señales, vamos a trabajar con el modelo de la derecha. Lo siguiente va a ser incluir la gestión de la salida D10 (diodo naranja), desde la entrada D8. Vamos a implementar en este caso una operativa distinta. Vamos a convertir la entrada digital en biestable por software, es decir, vamos a hacer que el diodo se encienda con una pulsación y se apague con la siguiente, no al soltar el pulsador. Como se ha comentado con anterioridad, esta funcionalidad la podemos implementar en Arduino o en el PC. Vamos a hacerlo en Arduino:

```
    if(digitalRead(8)== HIGH && testigo8 == false){
      if(orden == false){
       Serial.print("D8ON\n");
       orden = true;
       delay(10);
      }
      else {
       Serial.print("D8OFF\n");
       orden = false;
       delay(10);
      }
      testigo8 = true;
    }
     else if(digitalRead(8)== LOW && testigo8 == true)
      testigo8 = false;
```

```
    else if(textBox2->Text == "D8ON"){
        this->D8->BackColor=Color::Black;
        arduino->WriteLine("D10ON");
        this->D10-
    >BackColor=Color::Orange;
    }
        else if(textBox2->Text ==
    "D8OFF"){
                this->D8-
    >BackColor=BackColor;
                arduino-
    >WriteLine("D10OFF");
                this->D10-
    >BackColor=BackColor;
        }
```

En este caso necesitamos una segunda variable auxiliar, "orden", que también se declara como global y se inicializa a "false" en "setup()". Esta variable representa el estado de la orden que se ha dado (encender = true; apagar = false) para cambiarla cada vez que se acciona el pulsador.

A la derecha vemos la forma de gestionarlo en Visual C++. Como hemos decidido que sea el PC el que dé la orden de activar/desactivar la salida digital, cuando se recibe la información, además de actualizar la interfaz, tanto en la entrada (D8) como en la salida (D10), se debe enviar a Arduino la orden de encender el diodo naranja.

La entrada digital 7 no va a producir ningún efecto en la shield, ya que no nos quedan diodos libres; únicamente se va a informar al PC para que los muestre en la interfaz. A continuación vemos el código de Arduino y el del PC:

```
   if(digitalRead(7)== HIGH && testigo7 == false){
    Serial.print("D7ON\n");
    delay(10);
    testigo7 = true;
    delay(10);
   }
   else if(digitalRead(7)== LOW && testigo7 == true){
    Serial.print("D7OFF\n");
    delay(10);
    Serial.print("D7OFF\n");
    testigo7 = false;
    delay(10);
   }
```

```
   else if(textBox2->Text == "D7ON")
       this->D7-
    >BackColor=Color::Red;
       else if(textBox2->Text ==
    "D7OFF")
                this->D7-
    >BackColor=BackColor;
```

La entrada restante, D4, tampoco va a producir efecto visible en la placa. Con el fin de aportar una nueva funcionalidad, vamos a hacer que funcione como biestable, pero esta vez va a ser el PC el que se encargue de implementar esta funcionalidad. Va a ser sencillo entender cómo

funciona el código añadido ya que va a ser análogo al asociado al diodo naranja. La diferencia es que la funcionalidad de Arduino y el PC están intercambiadas:

```
   if(digitalRead(4)==  HIGH  &&
testigo4 == false){
    Serial.print("D4ON\n");
    delay(10);
    testigo4 = true;
    delay(10);
   }
   else  if(digitalRead(4)==  LOW  &&
testigo4 == true){
    Serial.print("D4OFF\n");
    delay(10);
    Serial.print("D4OFF\n");
    testigo4 = false;
    delay(10);
   }
```

```
else if(textBox2->Text == "D4ON" && testigo4
== false){
      if(orden == false){
            this->D4->BackColor=Color::Black;
            orden = true;
      }
      else{
            this->D4->BackColor=BackColor;
            orden = false;
      }
      testigo4 = true;
}
else if(textBox2->Text == "D4OFF" &&
testigo4== true)
      testigo4 = false;
```

Se puede observar que en Visual C++ aparecen las variables auxiliares análogas a las que teníamos en Arduino para la gestión de D8. Estas variables se declaran como globales junto con el resto de elementos privados de la clase Form1 y se inicializan en el constructor de la misma a “false”.

ENTRADAS ANALÓGICAS

Se trata sin duda de la modalidad de señal más compleja. Al igual que las entradas analógicas, pueden tomar valores cualesquiera dentro de un rango (0 – 5V). Al igual que las entradas digitales, su evolución es asíncrona respecto del programa. A diferencia de las salidas analógicas, las señales de entrada sí son realmente analógicas. Para un procesador, que es un dispositivo digital, no es posible trabajar con señales analógicas directamente. Estas señales se han de digitalizar. Para ello Arduino dispone de un conversor analógico – digital de 10 bits. Esto implica que los niveles analógicos de 0 a 5V se transforman en valores numéricos de 0 a 1023. Esto puede ser suficiente, insuficiente o excesivo según la aplicación.

Otro aspecto importante, quizá el más importante, es el ritmo al que se deben realizar estas conversiones o, dicho de otro modo, la cantidad de muestras de la señal que debemos tomar por unidad de tiempo. El conversor analógico digital emplea un tiempo determinado en realizar la conversión. El que tenemos en Arduino está en torno a 100 µs. Esto es el tiempo que tarda en completarse la instrucción analogRead(pin). Sin embargo, no podemos presuponer que será esa nuestra frecuencia de toma de muestras, ya que normalmente tomaremos una por ciclo de programa de manera que va a estar condicionada por la duración del ciclo de programa, que será muy superior.

La discusión sobre cuál debe ser la frecuencia de toma de muestras es densa. Una cosa que sí está clara es que debe ser constante y conocida sea cual sea el destino de las muestras. ¿Cómo garantizar que esto es así? Deberemos fijar el tiempo de ciclo de nuestro programa. Esto es más fácil decirlo que hacerlo ya que, antes de empezar, no sabemos cuánto va a durar el programa. Si necesitamos, enseguida vamos a ver cómo saberlo, que la toma de muestras se haga a intervalos de 1s (periodo de muestreo), será imprescindible que nuestro ciclo de programa sea más rápido. De otro modo nuestro problema no tendrá solución. Para poder implementar este ajuste del tiempo de ciclo tendremos que introducir una espera al final de loop(). Hemos de averiguar cuánto tiempo lleva consumido el ciclo de programa y esperar el tiempo restante. Para poder medir el tiempo podemos utilizar tiempo = millis(), que nos devuelve en tiempo transcurrido en milisegundos desde que se inició el programa. Si lo medimos al principio y al final de loop(), la diferencia será el tiempo transcurrido y podremos introducir un retardo hasta consumir el tiempo de ciclo prefijado:

```
tiempo = millis();
.
.
.
delay (1000-(millis()-tiempo));
```

Previamente habremos declarado “tiempo” como “unsigned long”.

No es un método ortodoxo ni eficiente de solventar el problema. Más adelante volveremos sobre esta cuestión para proponer un método mejor.

Es el momento de volver a la discusión sobre cuál debe ser la frecuencia de muestreo. La solución nos la da un valor derivado del llamado criterio de Nyquist, Shanon o Nyquist-Shanon, según la bibliografía. El criterio establece que para poder reconstruir una señal analógica sin perder información, es necesario que la frecuencia a la que se ha muestreado sea

superior al doble de la frecuencia máxima de la señal. No vamos a entrar en demostraciones al respecto.

La frecuencia máxima de una señal es difícil de estimar en algunos casos, en otros es conocida. Por ejemplo el sonido sabemos que no es perceptible por encima de 20 KHz, por lo que no tiene interés tratar de detectar armónicos de frecuencia superior. El sonido típicamente se muestrea a 44.1 kHz. La discusión realmente no termina aquí, ya que la señal puede contener y de hecho contiene, armónicos a frecuencias superiores. El hecho de no cumplir el criterio de Nyquist para ellos no implica que los perdamos sin más, sino que pueden generarnos errores denominados de "aliasing" en la reproducción de la señal. Para trabajar de forma rigurosa con entradas analógicas se hace necesario instalar filtros paso-bajo analógicos, denominados "anti-aliasing" que eliminen aquellos armónicos que nuestra frecuencia de muestreo hace imposible captar correctamente.

Esto nos indica que Arduino se nos va a quedar un poco corto, tanto por frecuencia de muestreo, como por carencia de filtros analógicos para trabajar con señales analógicas relativamente exigentes. Señales como temperatura o luminosidad que varían lentamente son perfectamente utilizables, no obstante.

Cuestión aparte es quién realiza el procesamiento de la señal analógica de entrada. La discusión es la misma que hemos planteado en el caso de las entradas digitales, es decir, en aplicaciones de control de procesos, el dispositivo de tiempo real (Arduino) debe tomar las decisiones. En procesamiento de señal, cuando sobre la información recibida se realizan pesados cálculos, el dispositivo más potente (el PC), debe ser el encargado.

Nuestra shield dispone de entradas analógicas de diferente naturaleza, entre las que se incluyen potenciómetros accionables por el usuario. Como hicimos con las entradas digitales vamos a aprovechar esta circunstancia para plantear formas distintas de trabajar con las entradas analógicas.

SENSOR DE TEMPERATURA

Una de las entradas que proporciona la shield está conectada a un sensor de temperatura. No vamos a entrar en las características de este sensor, pero por la naturaleza de la magnitud física sí que podemos determinar que se tratará de una señal de variación lenta por lo que Arduino será perfectamente capaz de seguir su evolución.

El tratamiento que vamos a hacer de esta señal será únicamente su monitorización. Para ello emplearemos el control "chart" disponible en la ventana de herramientas de Windows Forms. Vamos por lo tanto a representar la evolución de la temperatura ambiente a lo largo del tiempo.

En primer lugar vamos a abordar la lectura y envío de la temperatura por el puerto serie. Para ello construiremos en Arduino un string que comience por la letra "T" seguida del valor leído. Declaramos una variable tipo "String" llamada temperatura y añadimos el siguiente trozo de código:

```
temperatura = String(analogRead(A4));
```

```
temperatura = "T" + temperatura;
temperatura = temperatura + "\n";
Serial.print(temperatura);
```

En el lado del PC deberemos leer e interpretar esta información, para luego enviar los datos a una representación gráfica. No resulta simple la manipulación de las cadenas, por lo que lo vamos a explicar con cierto detalle. En el método de respuesta a evento de cambio en el contenido de textBox2 vamos a seguir añadiendo código para la nueva señal:

Al inicio del método vamos a declarar dos variables tipo "String" para ayudarnos en el proceso:

```
String^ cadena;
String^ cadenaAux;
cadena = textBox2->Text;
```

A continuación el código de interpretación del contenido:

```
else if (cadena != ""){
	cadenaAux=cadena;
	cadena=cadena->Substring(0,1);
	if (cadena == "T"){
		cadenaAux=cadenaAux->Substring(1);
		DatoNuevoChart(Int32::Parse(cadenaAux));

	}
}
```

El procedimiento consiste en copiar el contenido de la caja de texto a un string que vamos a manipular posteriormente. En primer lugar, nos aseguramos que la cadena no está vacía; de lo contrario se lanzaría una excepción por acceso a posiciones inexistentes. A continuación comprobamos si la cadena empieza por la letra "T" que sería la que corresponde con la información de temperatura. Al extraer la subcadena que empieza en la posición 0 y tiene una longitud de un carácter (Substring(0,1)), modificamos la propia cadena, por lo que previamente la habremos salvado en la variable auxiliar "cadenaAux". Una vez comprobado que es la información de temperatura, extraemos el dato numérico que aparece en el resto del string desde la posición un hasta el final (Substring(1)), es decir, ignorando la "T". Este dato lo vamos a pasar a la gráfica como entero (Int32::Parse(cadenaAux), a través del método "DatoNuevoChart" que explicaremos luego.

La representación gráfica de los datos de temperatura se puede realizar a través del control "chart" disponible en el cuadro de herramientas como se ve en la Figura 18.

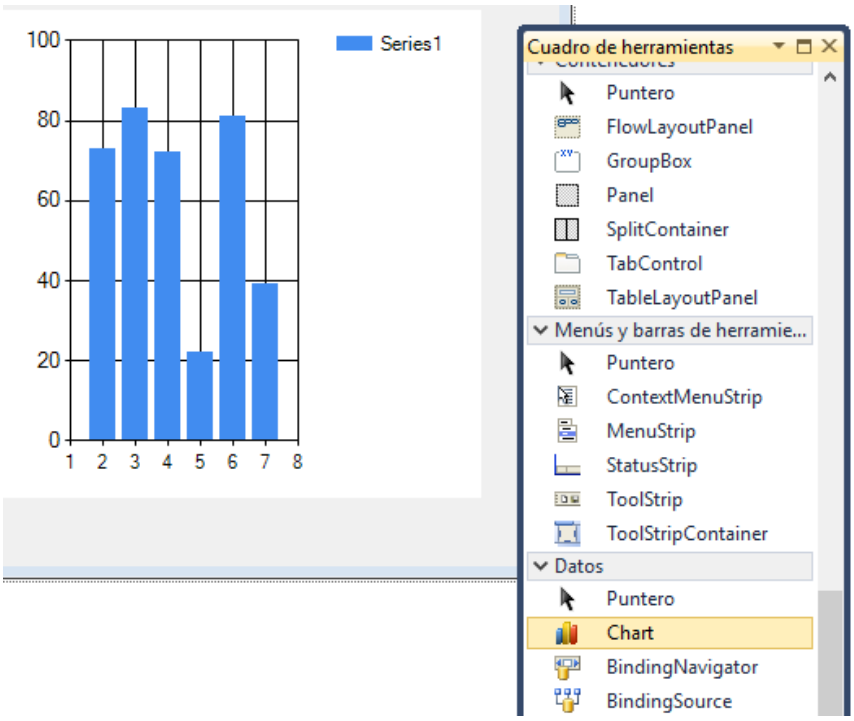


FIGURA 18

Es un control complejo, integrado por objetos agrupados en colecciones como son las series de datos. A través de la propiedad Series (Colección)->Name podemos modificar el nombre por defecto de la serie "Series1" para pasar a llamarle "Temperatura". También podemos cambiar el tipo de gráfico asociado a la serie Series (Colección)->ChartType, para que en lugar del gráfico de barras, nos muestre una representación en forma de línea. El resultado se muestra en la Figura 19.

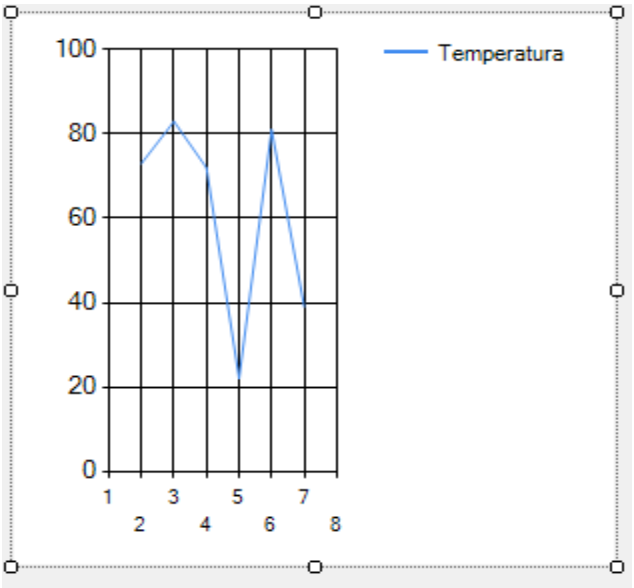


FIGURA 19

Existe la posibilidad de introducir una lista de puntos en la serie. También se pueden añadir más series e inicializarlas. Nada de esto vamos a hacer. En su lugar, iremos añadiendo puntos a la serie en tiempo de ejecución. También para ello tenemos diferentes alternativas. Nos decantaremos por lo siguiente:

1. En el constructor de la clase Form1(), introduciremos el número de puntos que pretendamos visualizar de forma simultánea, por ejemplo 30, correspondientes a 30 segundos de monitorización. Los inicializamos a un valor cualquiera, por ejemplo 50:

```
for(int i=1; i<=8; i++){
                    this->chart1->Series["Temperatura"]->Points->AddY(50);}
```

2. Construimos el método DatoNuevoChart para introducir nuevos datos. Si nada más los introducimos, el gráfico se re-escalará constantemente para poder mostrar más datos en el mismo espacio. Preferimos que por cada dato que entre salga el más antiguo, por lo que iremos eliminando el dato de la posición 0:

```
private: void DatoNuevoChart(int y)
 {
      this->chart1->Series["Temperatura"]->Points->AddY(y);
      this->chart1->Series["Temperatura"]->Points->RemoveAt(0);
}
```

Si ejecutamos lo que tenemos hasta ahora, veremos dos cuestiones relevantes.

1. La temperatura se actualiza a intervalos de un segundo como estaba previsto, pero los valores que muestra la gráfica no se corresponden con valores de temperatura reales. Esto es debido a que lo que recibimos son los valores de tensión medidos en la entrada analógica en la que 0V se corresponden con un cero y 5V con 255. Será necesario calibrar el sensor y hacer la conversión correspondiente. Dejamos esta tarea al usuario interesado en ello ya que no tiene mayor relevancia a nivel de programación.
2. La temperatura evoluciona como era de esperar, pero si hemos mantenido en nuestra interfaz el resto de señales, veremos que su actualización se ha retrasado al mismo intervalo de un segundo que hemos configurado para la entrada analógica. Esto no resulta admisible.

¿Cómo podemos entonces ajustar el periodo de muestreo de la entrada analógica sin que afecte al ciclo de programa? La respuesta está en el uso de interrupciones. En Arduino, como en cualquier entorno basado en un controlador digital, existe la posibilidad de programar interrupciones. Se trata de eventos externos o internos que interrumpen el flujo de programa para ejecutar una rutina especial. A continuación devuelven en control al lugar en que se había detenido la ejecución. La idea es entonces asociar la lectura y envío de la señal a una interrupción asociada a un temporizador interno pre-programado a un segundo.

A nivel de configuración va a ser necesario habilitar la interrupción y ajustar su funcionamiento. Para ello, en la función setup() introducimos el siguiente código:

```
noInterrupts();
TCCR1A = 0; //Borrado del registro de configuración del temporizador
TCCR1B = 0; //Borrado del registro de configuración del temporizador
TCNT1  = 0; //Inicialización de la cuenta del temporizador a cero
```

```
OCR1A = 62500; //Valor de comparación para la interrupción
TCCR1B |= (1 << WGM12);  // modo CTC (la interrupción se produce cuando la cuenta llega a OCR1A)
TCCR1B |= (1 << CS12); // 256 pre-escala de la frecuencia respecto del reloj principal (16MHz/256 = 62500Hz)
TIMSK1 |= (1 << OCIE1A);  // Habilita la interrupción
interrupts();
```

Como vemos, las interrupciones se deshabilitan temporalmente para realizar su configuración. Los registros de configuración se borran completamente para luego introducir los valores relevantes:

- La pre-escala, esto es, la frecuencia a la que va a trabajar el temporizador y que se deriva del reloj principal de la CPU a través de un divisor de frecuencia que en este caso ajustamos a 256.
- El valor de comparación, que va a hacer que se produzca la interrupción cuando la cuenta del temporizador lo alcance.

La rutina de servicio a la interrupción tiene un nombre concreto y su contenido es programable. En este caso lo que procede hacer es introducir en ella la parte de código que lee la entrada analógica, la codifica y la envía:

```
ISR(TIMER1_COMPA_vect)
{
 temperatura = String(analogRead(A4));
 temperatura = "T" + temperatura;
 temperatura = temperatura + "\n";
 Serial.print(temperatura);
}
```

Vemos que los tiempos de respuesta del resto de señales vuelven a ser normales. Arduino no dispone de muchas interrupciones, por lo que incrementar el número de entradas y hacerlas compatibles con el resto de señales cada vez se torna más complicado.

Es posible que percibamos que nuestra gráfica no se actualiza siempre en tiempo real. Esto puede depender de la planificación del hilo secundario que escucha el puerto serie. Tenemos la certeza, no obstante, de que Arduino manda la información regularmente como está previsto. Esta información no se pierde, al menos mientras mantengamos un volumen de datos bajo, el buffer del puerto en el PC no se va a llenar, aún así podríamos configurarlo con un tamaño mayor. Como consecuencia de todo ello podemos concluir que la información en la gráfica es correcta.

Todo ello está muy bien, pero a veces no es suficiente con ver la información en una gráfica, sino que debe quedar constancia de la evolución que ha tenido la señal a lo largo del tiempo. Esto nos obliga a almacenar el histórico de datos, para lo cual podemos emplear un fichero. Para ello deberemos incorporar el espacio de nombres correspondiente:

```
using namespace System::IO;
```

Como hicimos con la visualización gráfica, vamos a crear un sencillo método que permita almacenar datos en el fichero. Los datos serán añadidos al final del mismo:

```
private: void guardar(String^ dato)
{
     StreamWriter^ fichero = gcnew StreamWriter("temperatura.txt",true);
     fichero->WriteLine(dato);
     fichero->Close();
}
```

Precisamente el parámetro "true" tras el nombre del fichero provoca que el contenido anterior no se borre sino que se añada el nuevo tras él.

El guardado del dato se hará justo delante de su envío a la gráfica:

```
else if (cadena != ""){
       cadenaAux=cadena;
       cadena=cadena->Substring(0,1);
       if (cadena == "T"){
              cadenaAux=cadenaAux->Substring(1);
              guardar(cadenaAux);
              DatoNuevoChart(Int32::Parse(cadenaAux));
       }
}
```

La discusión sobre el salvado de datos en ficheros se puede alargar tanto como se quiera: cuadros de diálogo para seleccionar el fichero y la ruta, botones de inicio y fin de almacenamiento de datos, etc. No lo vamos a abordar en este momento, pero sin duda tiene interés profundizar en esta cuestión.

LECTOR DE INFRARROJOS

En la shield que tenemos conectada se nos presenta un fotodiodo como entrada analógica. Habitualmente esto sería una entrada digital, que se activaría si llega luz infrarroja y quedaría a cero en caso contrario. En este caso la entrada es capaz de medir la intensidad de la luz infrarroja incidente.

No tendría mucho interés monitorizar sin más la señal, ya que es lo que acabamos de hacer con la entrada de temperatura, así que vamos a buscarle una utilidad distinta.

En una primera aproximación podemos incluir en el bucle principal de Arduino, la lectura de la entrada analógica A3, que es la que corresponde al sensor infrarrojo, con un código análogo al que empleamos para la entrada de temperatura antes de optar por temporizarla mediante interrupción:

```
infrarrojo = String(analogRead(A3));
infrarrojo = "I" + infrarrojo;
infrarrojo = infrarrojo + "\n";
Serial.print(infrarrojo);
```

Podremos ver en la caja de texto del PC la información correspondiente al sensor y cómo varía constantemente. Se trata de una señal que varía muy deprisa, por lo que merece la pena muestrearla muy rápidamente, pero la información que proporciona es de escasa utilidad en muchos casos. Si lo que queremos es, por ejemplo, leer una trama transmitida por infrarrojos, lo que necesitamos es traducir esta señal a digital y muestrearla a intervalos regulares. La primera parte es sencilla: simplemente establecemos un umbral y, por encima de él consideramos que la señal es uno y por debajo, cero.

La segunda parte es más complicada. Primero debemos saber a qué frecuencia esperamos que lleguen los pulsos de luz. Después tendremos que habilitar un mecanismo para muestrear a esa misma velocidad.

Arduino dispone de la posibilidad de generar una interrupción condicionada a la comparación entre el valor de una entrada analógica y un valor de referencia. Esta sería una solución ideal ya que podríamos registrar cada pulso de luz que llegue al detector de infrarrojos. Lamentablemente esta opción está asociada a la entrada analógica A1, que en nuestra placa está conectada a un potenciómetro, no al sensor de infrarrojos. Habrá que recurrir por tanto a muestrear la señal. Esta decisión plantea también dos alternativas:

- Muestrear tan rápido como sea posible (en cada ciclo del bucle principal), esperando no perder ninguna transición.
- Recurrir a una interrupción asociada a un temporizador que provoque la lectura de la entrada a la velocidad que sea necesaria.

Si conocemos la velocidad de variación de la señal luminosa, la segunda opción parece la más eficiente y segura. Esto es así por ejemplo en las señales que emiten los mandos a distancia comerciales. El problema es que estos dispositivos trabajan sobre una portadora cuya frecuencia es de, dependiendo casos, 30 – 40 kHz. No parece una frecuencia alcanzable para nuestro bucle principal, que tendría que completarse en unos 10µs.

Dejaremos la lectura de datos de un mando a distancia para una versión posterior de esta guía y de momento nos limitaremos a un funcionamiento más simple: vamos a retransmitir la señal incidente por el led infrarrojo que tenemos conectado a la salida analógica 2. Si el bucle es suficientemente rápido, podríamos estar construyendo un repetidor para un mando a distancia. También podríamos monitorizar la señal en el PC, pero no añadiría nada nuevo a lo visto con la señal de temperatura.

Únicamente vamos a añadir un botón de permiso para habilitar o deshabilitar la retransmisión de infrarrojos. Será un botón basado en “checkbox” como los utilizados hasta ahora y cuya señal codificaremos como IHON, IHOFF (habilitación/deshabilitación de infrarrojos).

En Arduino declararemos una variable booleana que tomará el valor “true” cuando llegue IHON y false con IHOFF. Cuando la variable sea “true”, haremos que la salida analógica A2 sea igual a la entrada analógica A3.

Una vez realizada la programación, hemos podido observar que habitualmente la retransmisión de la señal infrarroja se hace correctamente.

ENTRADAS POTENCIOMÉTRICAS

Nuestra shield dispone de dos potenciómetros conectados a las entradas analógicas A0 y A1. En este apartado vamos simplemente a monitorizar su estado mediante el mismo tipo de control que utilizamos para establecer el estado del diodo led blanco. Este control se denomina "trackBar". La diferencia va a estar en que lo vamos a hacer inaccesible al usuario externo ajustando su propiedad "Enable" a "False. De esta forma, la posición del cursor se modificará por código en base a la información que llegue de Arduino. Esta información la vamos a codificar mediante, como es habitual, una letra ("P" para el potenciómetro conectado a A0 y "F" para el conectado a A1), seguida del valor devuelto por la entrada analógica. Una vez separada la letra, enviamos el valor a la propiedad "Value" de la barra de desplazamiento:

```
else if(cadena == "P"){
              cadenaAux=cadenaAux->Substring(1);
              textBox1->Text =cadenaAux;
              trackBar2->Value=Int32::Parse(cadenaAux);
       }
else if(cadena == "F"){
              cadenaAux=cadenaAux->Substring(1);
              textBox1->Text =cadenaAux;
              trackBar3->Value=Int32::Parse(cadenaAux);
       }
```

Hemos enlazado las comprobaciones con la que hacíamos para la entrada de temperatura. La dinámica es análoga e incluso aprovechamos las mismas variables auxiliares, ya que los códigos son mutuamente excluyentes.

En el lado de Arduino simplemente enviamos la lectura de las entradas dentro del bucle principal:

```
//Entradas potenciómetros
  pot1 = String(analogRead(A0));
  pot1 = "P" + pot1;
  pot1 = pot1 + "\n";
  Serial.print(pot1);

  pot2 = String(analogRead(A1));
  pot2 = "F" + pot2;
  pot2 = pot2 + "\n";
  Serial.print(pot2);
```

Una vez ejecutado el código, veremos como la posición del cursor en las barras de desplazamiento, se actualiza de forma inmediata al mover el cursor en la shield. Incluso podemos observar pequeñas oscilaciones debido a que la lectura no es muy estable. Podemos utilizar una temporización para reducir en número de envíos, ya que es una señal que no varía durante la mayor parte del tiempo. También se pueden establecer "saltos" de variación, de manera que no se realicen envíos si no se han producido variaciones significativas del valor. Todo ello con el fin de reducir la utilización del puerto serie.

Con los controles introducidos en esta sección, la interfaz adquiere el aspecto que vemos en la Figura 20.

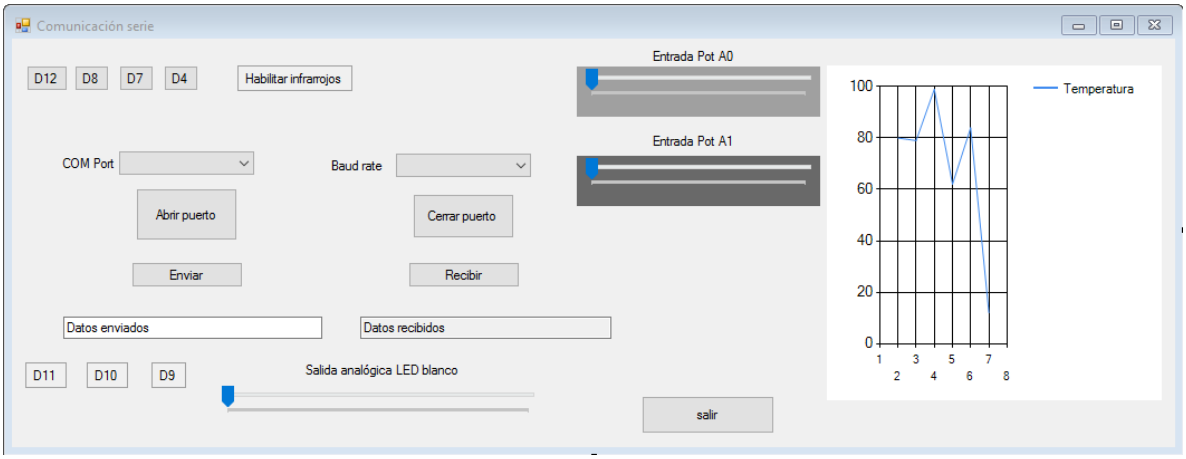


FIGURA 20

REFERENCIAS

Paseo guiado por Visual C++

Getting started with Windows forms

Crear una aplicación de Windows Forms

How mouse input Works on Windows Forms

Guía de Microsoft para comunicaciones por puerto serie en Windows

Desarrollo de aplicación de comunicación por puerto serie (vídeos de youtube):

- Primera parte.
- Segunda parte.
- Tercera parte.

Guía de trabajo con gráficas en Windows Forms