

Departamento de Arquitectura y Tecnología de Computadores



UNIVERSIDAD DE SEVILLA

# Comunicación Hardware – Software: desde CoOs a una aplicación en .Net

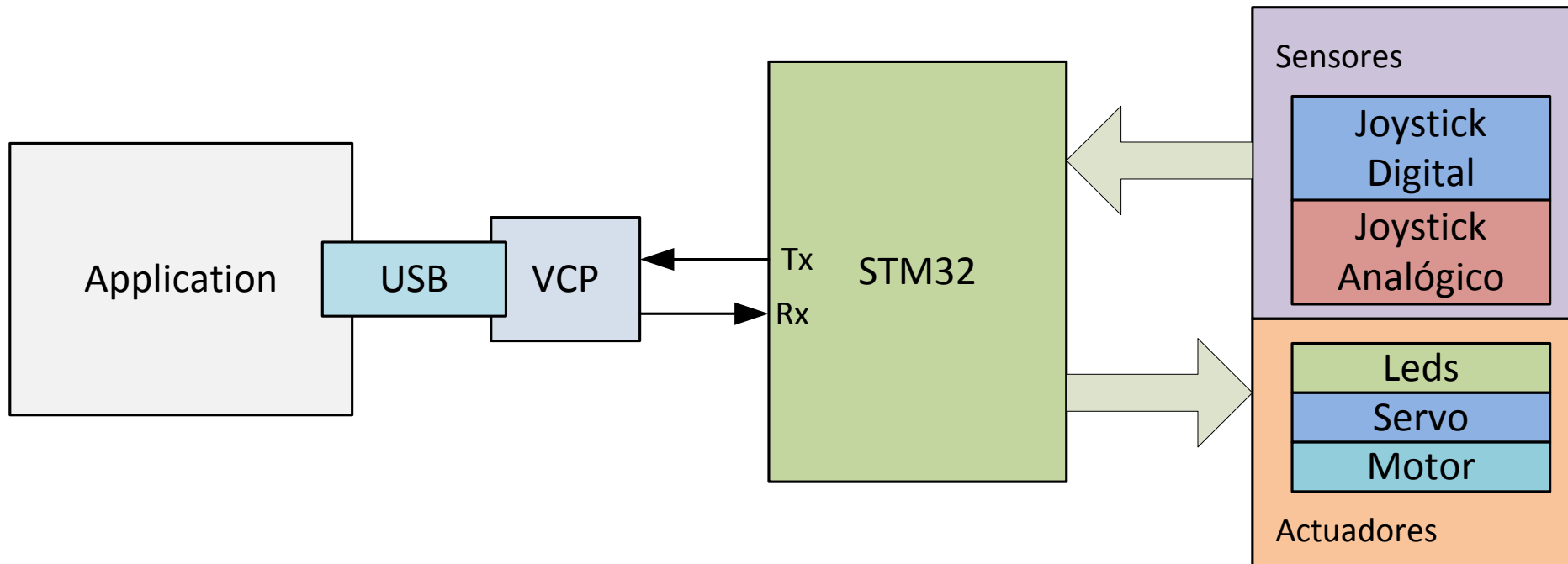
E.T.S. de Ingeniería Informática  
Avda. Reina Mercedes, S/N.  
41012 Sevilla, SPAIN

Escuela Universitaria Politécnica  
C/ Virgen de África, 7.  
41011 Sevilla, SPAIN

# Objetivos

- Ilustrar el manejo de los puertos series asíncronos de un PC desde C#.
- Implementar y desarrollar un mecanismo de comunicación de alto nivel.
- Empaquetado de la información:
  - Enviar al PC paquetes con el estado de los sensores:
    - Joystick digital y analógico.
  - Recibir comandos desde el PC s:
    - Mover servo / motor.
    - Lanzar animaciones de Leds.
- Implementar una aplicación en C# usando Windows Form:
  - Mostrar la información recibida a través del puerto serie al usuario.
  - Generar y enviar comandos por el puerto serie al sistema.

# Visión a nivel de sistema

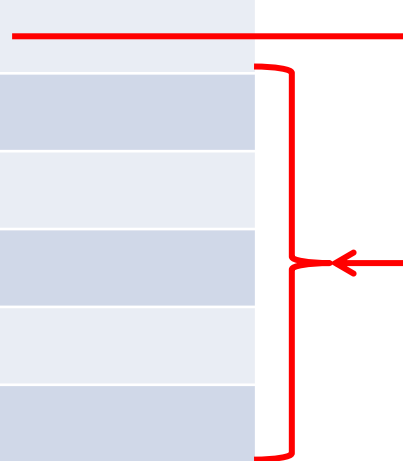


# Empaquetando la información

- Con anterioridad hemos transmitido cadenas de texto a través del puerto serie:
  - Muy práctico para observar la información.
  - Muy difícil la decodificación de los datos e interpretación de comandos.
  - Usa una gran número de bytes, baja densidad de información.
- Para comunicar sistemas a un nivel más alto es común el usar mecanismos de empaquetamiento de la información.
  - La información se envía encapsulada entre una cabecera (header) y una cola (trailing).
  - Los datos se transmiten tal cuales en binario, no una representación en texto.
    - Un número de 16 bits en dos bytes VS. Una cadena de texto que representa un número de 16 como 5 dígitos ( 5 bytes).

# Empaquetando la información

Off-set	Descripción	Dato
+0	Inicio de trama	0x7E
+1	Longitud de la trama	0x05
+2	Dato [0]	0x01
+3	Dato [1]	0x55
+4	Dato [2]	0x00
+5	Dato [3]	0xAA
+6	Dato [4]	0x04
+8	Fin de trama	0x0D



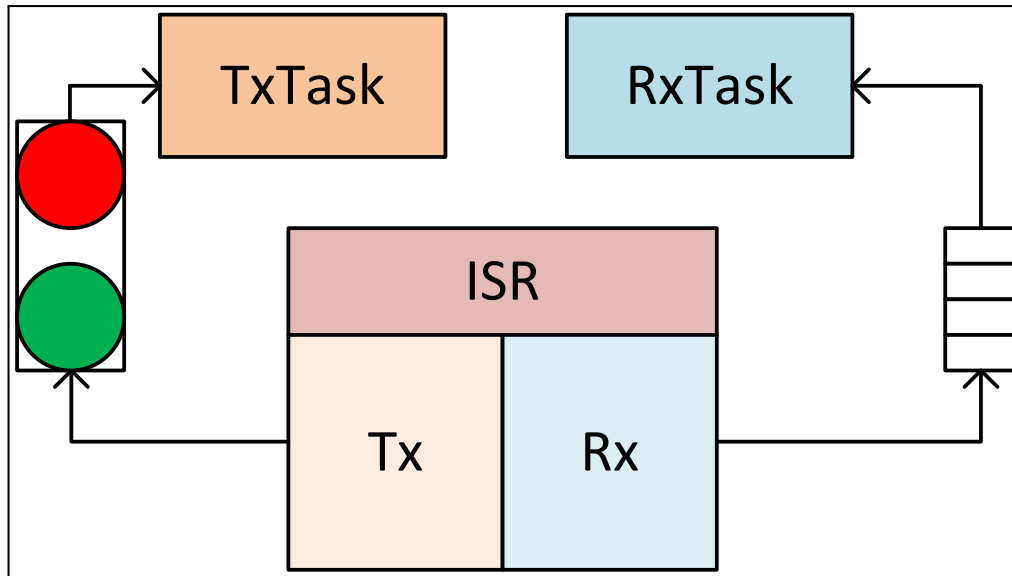
# Interpretando el paquete

- En este ejemplo enviamos los dos ejes del joystick analógico (16 bits / 2 bytes), así como el del joystick digital (1 byte).
- En total se envían 5 bytes .
  - $X = 256 * \text{Dato}[0] + \text{Dato}[1]$
  - $Y = 256 * \text{Dato}[2] + \text{Dato}[3]$
  - $\text{Dig.} = \text{Dato}[4]$

Off-set	Descripción	Dato	Interpretación
+0	Inicio de trama	0x7E	Header
+1	Longitud de la trama	0x05	
+2	Dato [0]	0x01	A. Joy X MSB
+3	Dato [1]	0x55	A. Joy X LSB
+4	Dato [2]	0x00	A. Joy Y MSB
+5	Dato [3]	0xAA	A. Joy Y LSB
+6	Dato [4]	0x04	Joy Digital
+8	Fin de trama	0x0D	Trailing

# Recordando la transmisión serie

- **2 tareas:** Transmisión y recepción.
- **Para transmitir** se usa un semáforo encapsulado en una función: Transmisión bloqueante.
  - `void SerialSendByte(char data)`
- **Para recibir** una cola de mensajes: Recepción bloqueante.
  - `c=CoPendQueueMail(queueRxId, 0, &err);`



# **Parte 1: Transmitiendo datos desde el STM32 - Recibiendo en C#**



# Ejercicio 1

- Modificar **serialTxTask** para que:
  - Lea cada 100mSeg los valores de los dos ejes del joystick analógico y el joystick digital.
    - a = getAnalogJoy(axxis);
    - b = Read\_Joy();
  - Envíe un paquete con la estructura expuesta anteriormente usando la función **SendSerialByte** de manera iterativa.

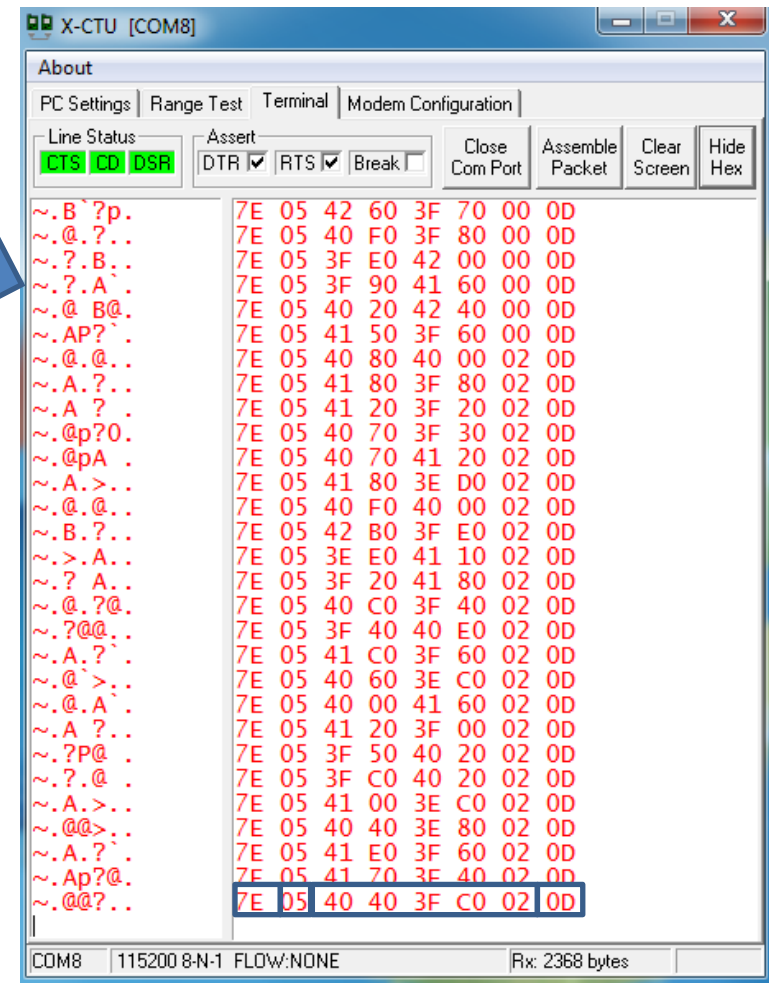
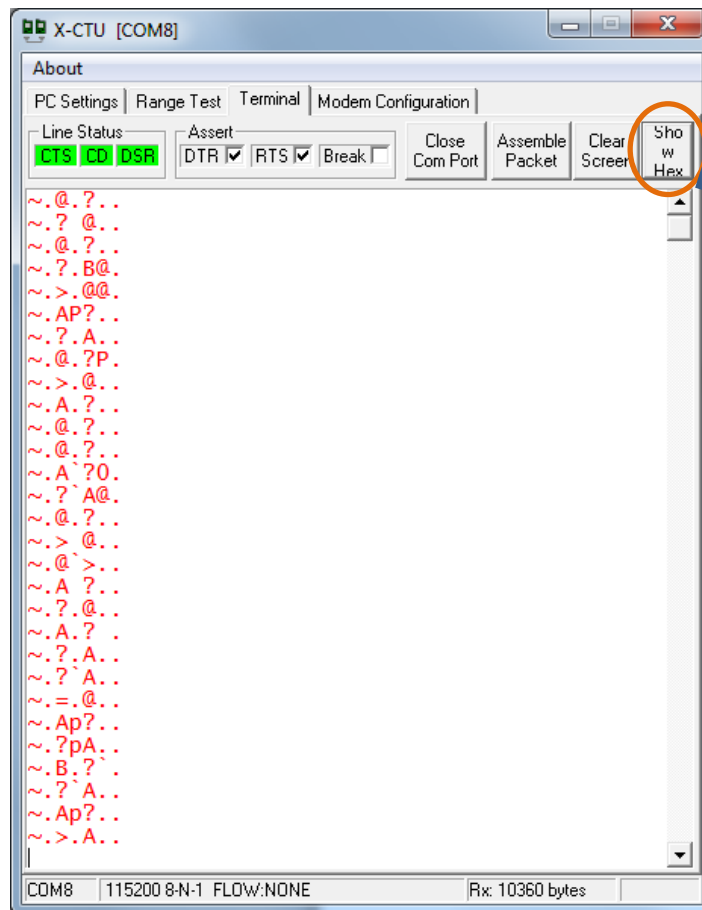
```
void serialTxTask(void * parg){
    int i;
    for(;;){
        //Actualizar estado
        x = getAnalogJoy(0);
        //...

        SerialSendByte(0x7E); //Inicio de trama
        SerialSendByte(5);    //Longitud
        //SerialSendByte(...) //Byte 0 -> MSB del eje x
        //SerialSendByte(...) //Byte 1 -> LSB del eje x
        // ..
        //SerialSendByte(..)   //Byte 5 -> Unico byte del joystick
        SerialSendByte(0x0D); //fin de trama

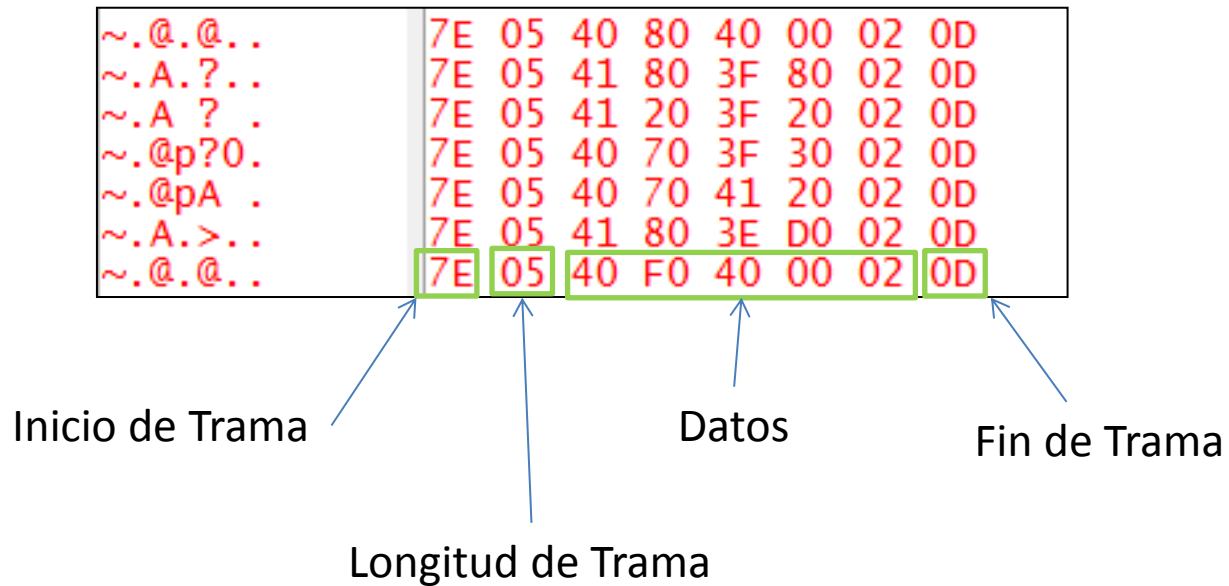
        CoTimeDelay(0,0,0,100); //Espèra
    }
}
```

# Ejercicio 1

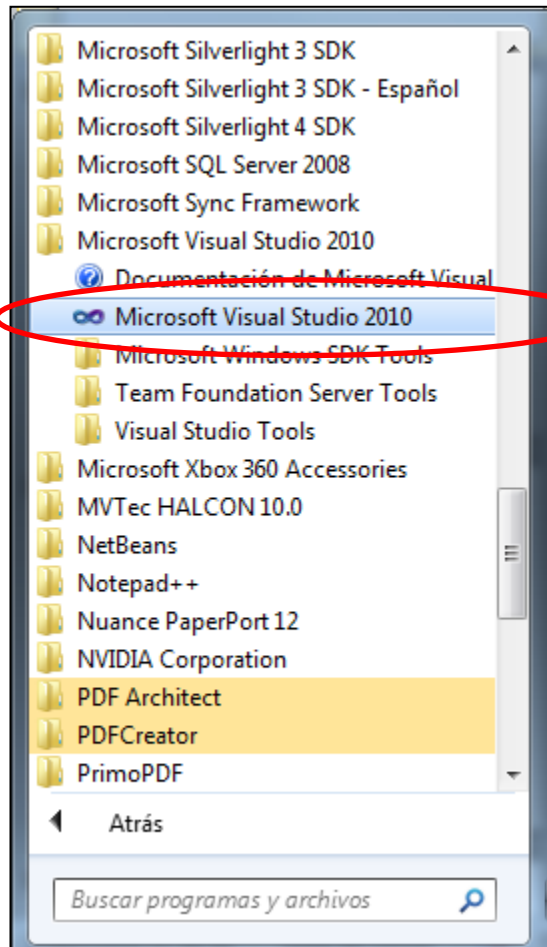
- Comprobar el funcionamiento en el XCT-U.
- Usar vista en hexadecimal.



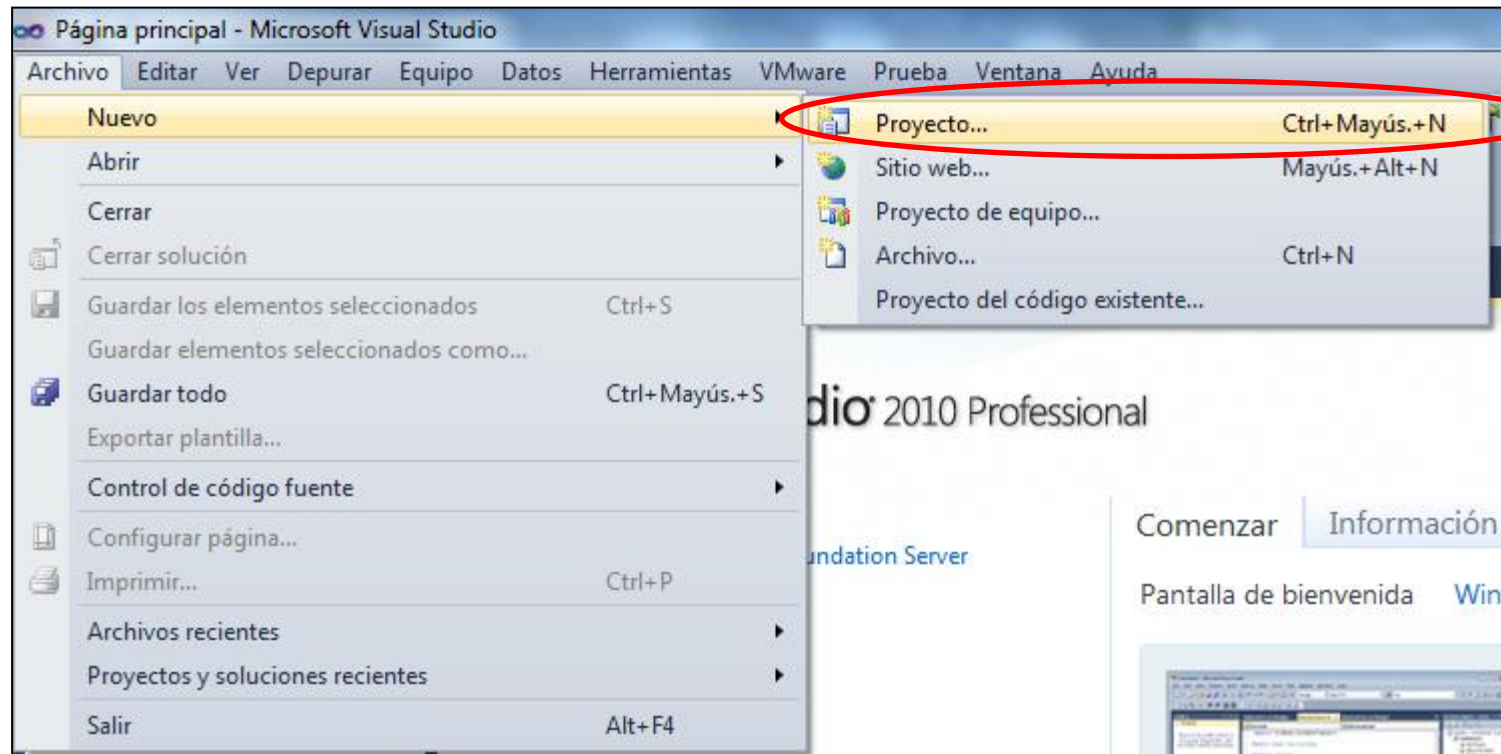
# Paquetes de datos



# Creando un proyecto en VS2010



# Creando un proyecto en VS2010



Nuevo proyecto

Plantillas recientes

Plantillas instaladas

Visual C#

Windows

Web

Office

Cloud

Prueba

Reporting

SharePoint

Silverlight

WCF

Workflow

Otros lenguajes

Otros tipos de proyectos

Base de datos

Proyectos de prueba

Plantillas en línea

.NET Framework 4

Ordenar por: Predeterminado

Buscar Plantillas instaladas

	Aplicación de Windows Forms	Visual C#
	Aplicación WPF	Visual C#
	Aplicación de consola	Visual C#
	Aplicación web ASP.NET	Visual C#
	Biblioteca de clases	Visual C#
	Aplicación web de ASP.NET MVC 2	Visual C#
	Aplicación de Silverlight	Visual C#
	Biblioteca de clases de Silverlight	Visual C#
	Aplicación de servicios WCF	Visual C#
	Aplicación web (entidades de datos dinámicos ASP.NET)	Visual C#
	Habilitar Windows Azure Tools	Visual C#

Tipo: Visual C#

Proyecto para crear una aplicación con una interfaz de usuario de Windows Forms

Nombre: Communication App

Ubicación: C:\Users\Angel\Desktop\Angel\docencia\SETR2\Practicas\Practica 6 - Comunicacion HW-SW\

Nombre de la solución: Communication App

Examinar...

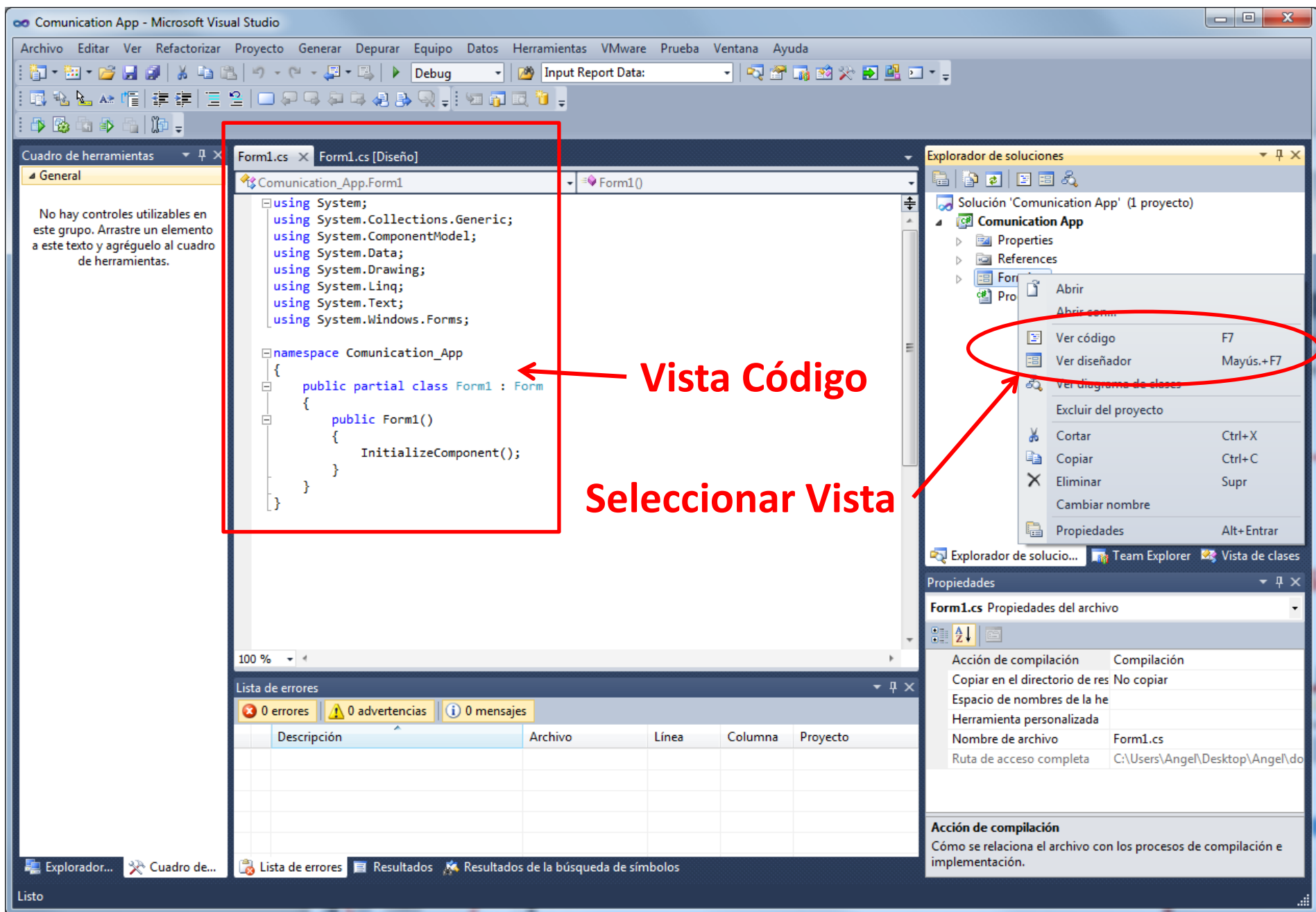
☒ Crear directorio para la solución

☐ Agregar al control de código fuente

Aceptar

Cancelar

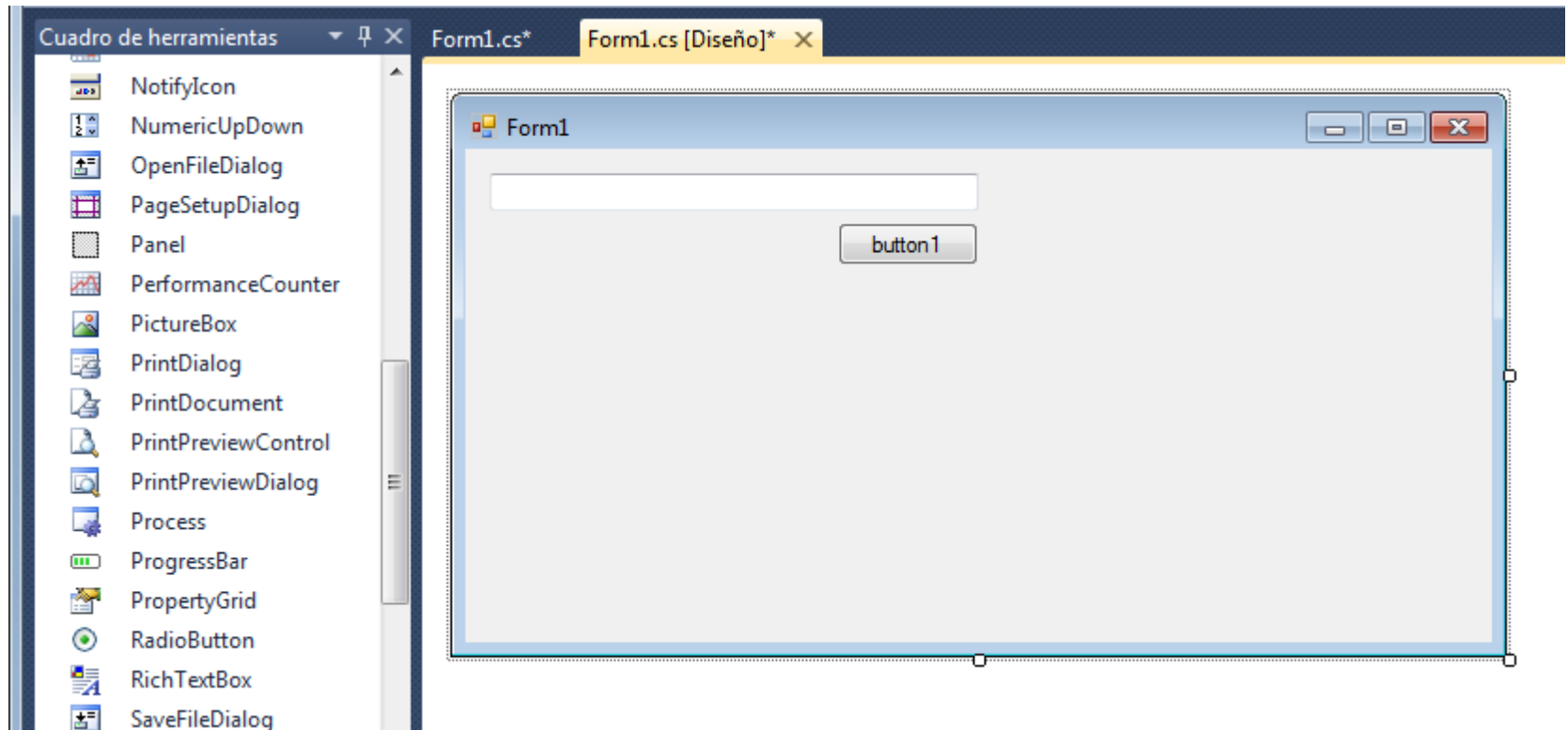






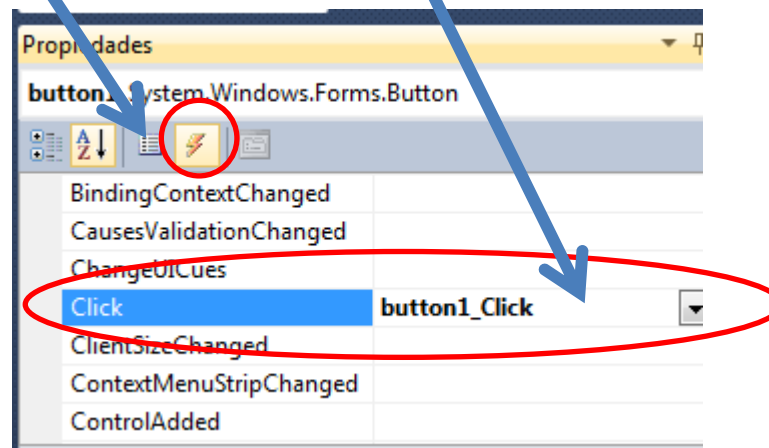
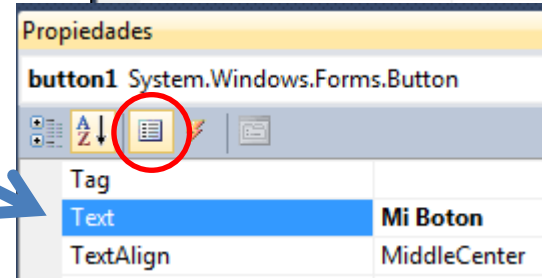
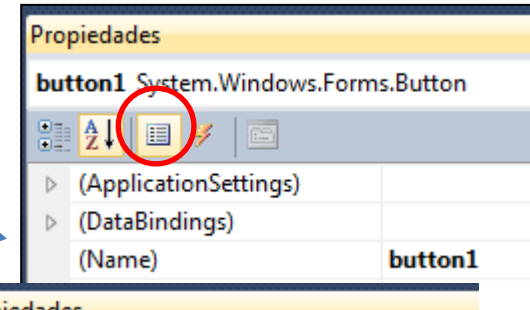
# Añadiendo elementos al formulario

- Arrastrar desde la barra de herramientas un **“Text Box”** y un **“Button”**.

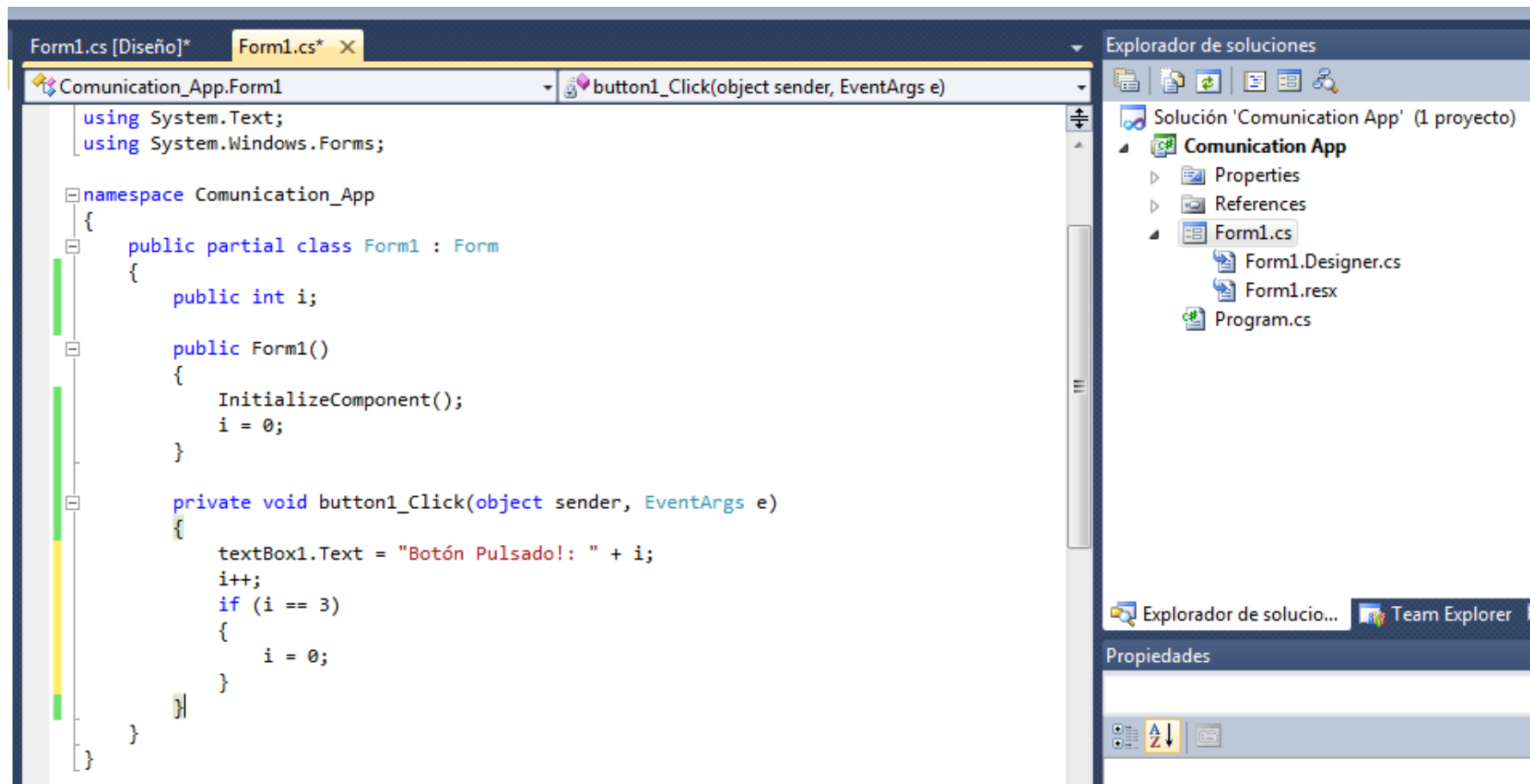


# Cambiando las propiedades y añadiendo eventos

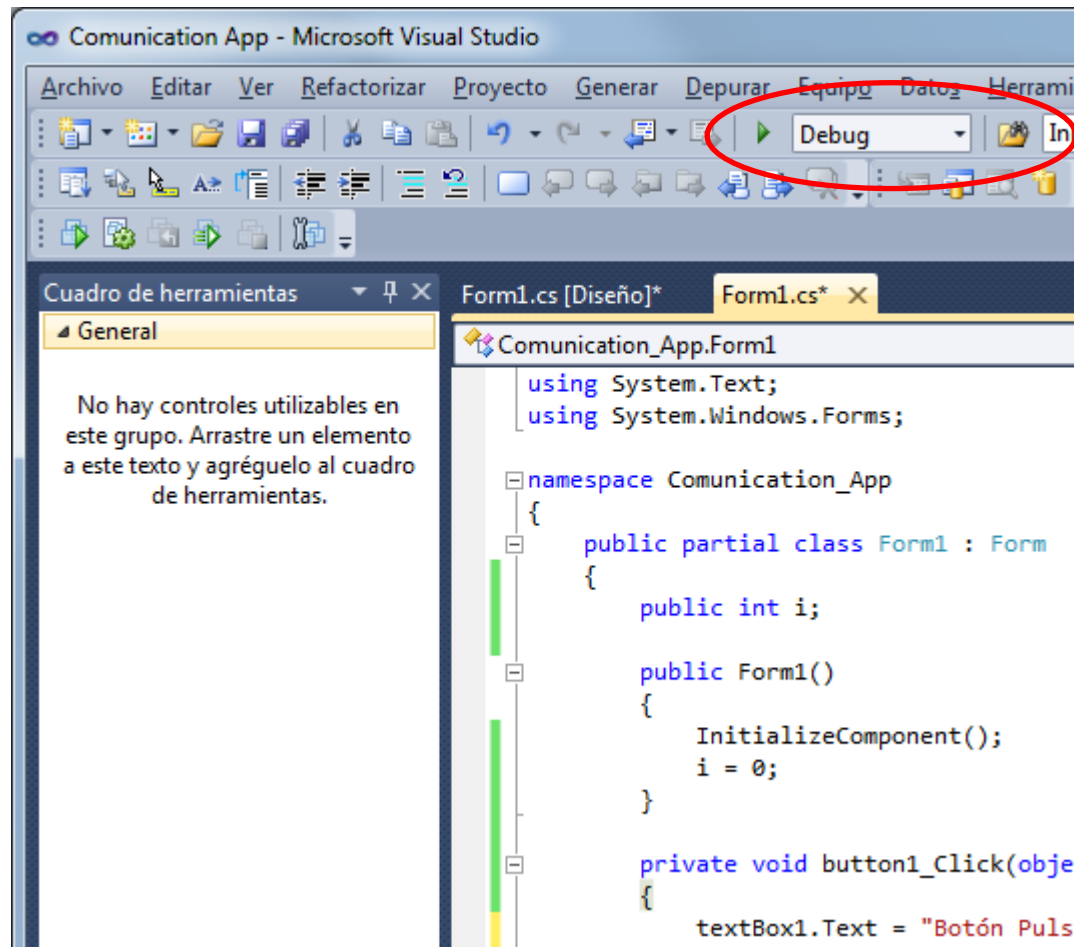
- Seleccionar el botón:
  - Nombre del objeto.
  - Texto del botón.
  - Eventos:
    - Hacer doble click para añadir.



# Implementando la funcionalidad del formulario

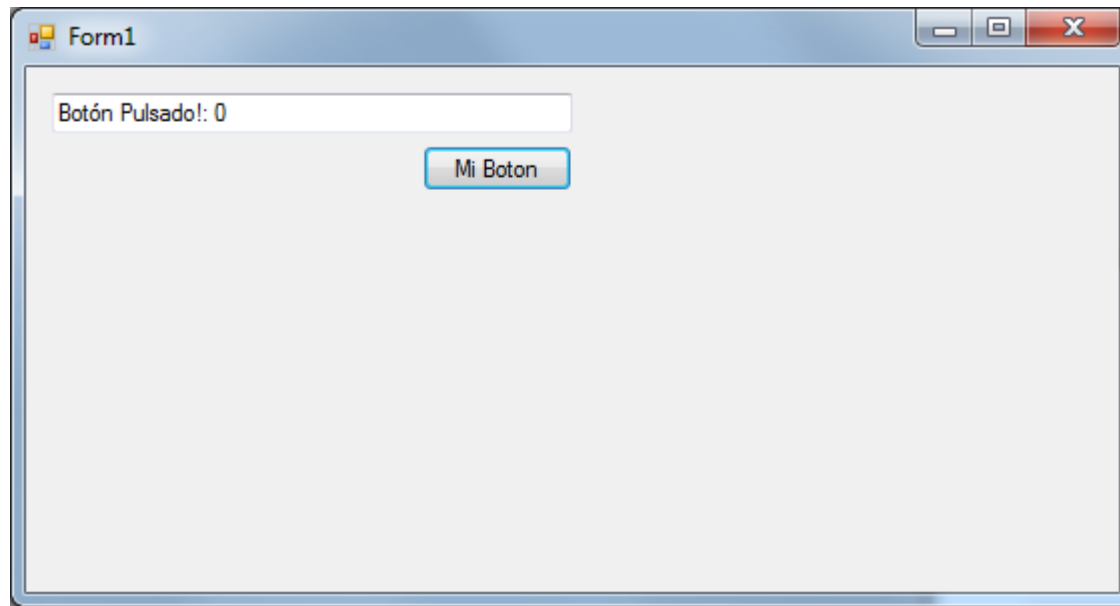


# Ejecutando el proyecto



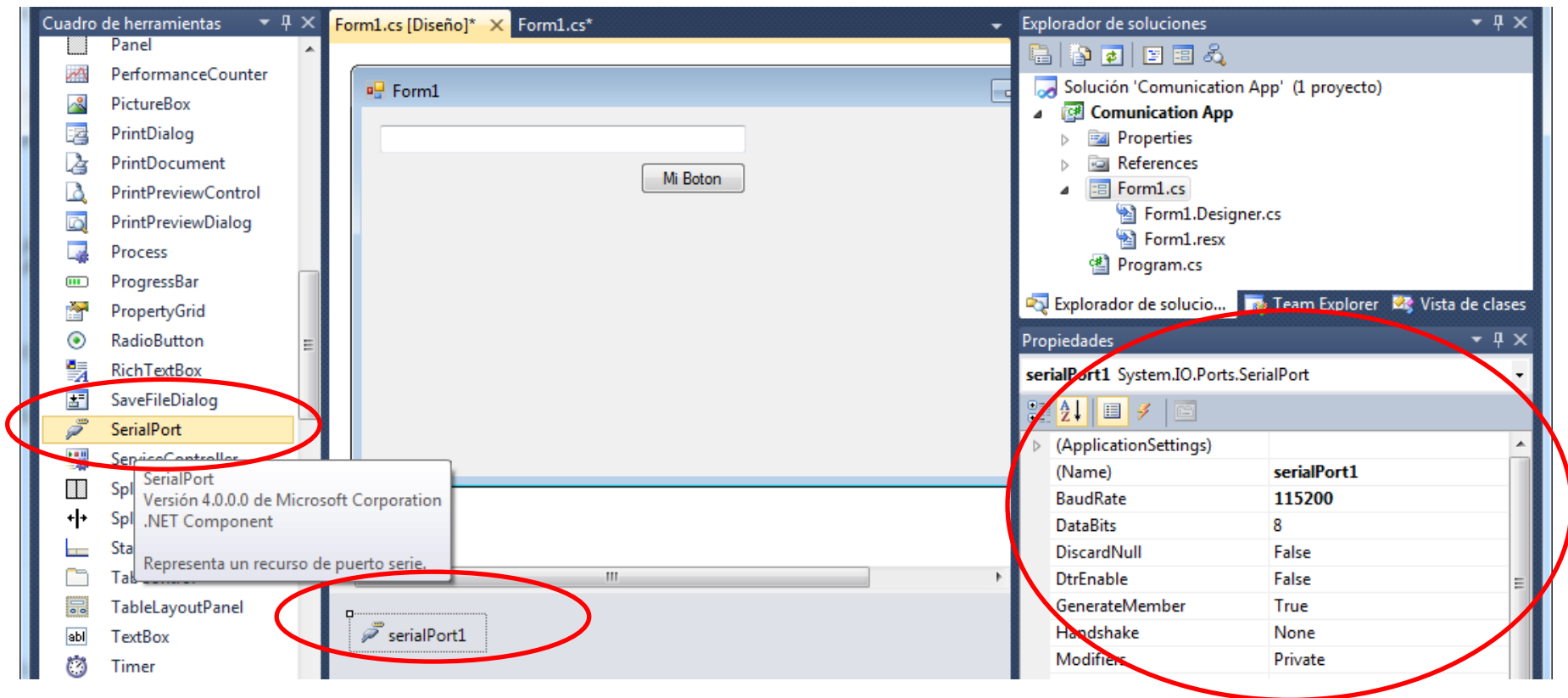
# Aplicación en ejecución

- Comprobar su funcionamiento.



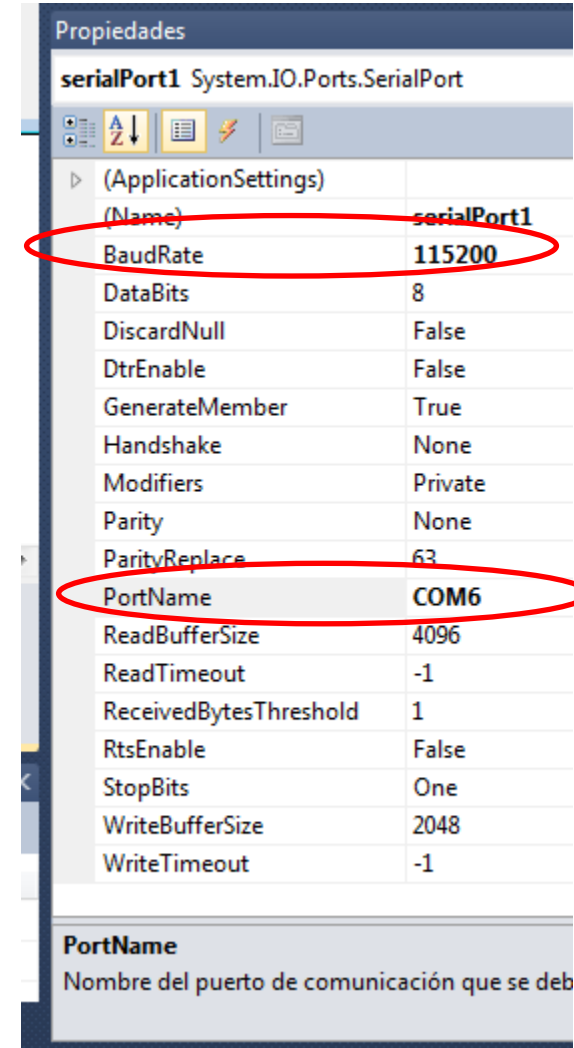
# Agregando el puerto serie

- Arrastrar un **SerialPort** sobre el formulario.
- Se añade en una barra debajo de formulario.
- Podemos editar sus propiedades/eventos



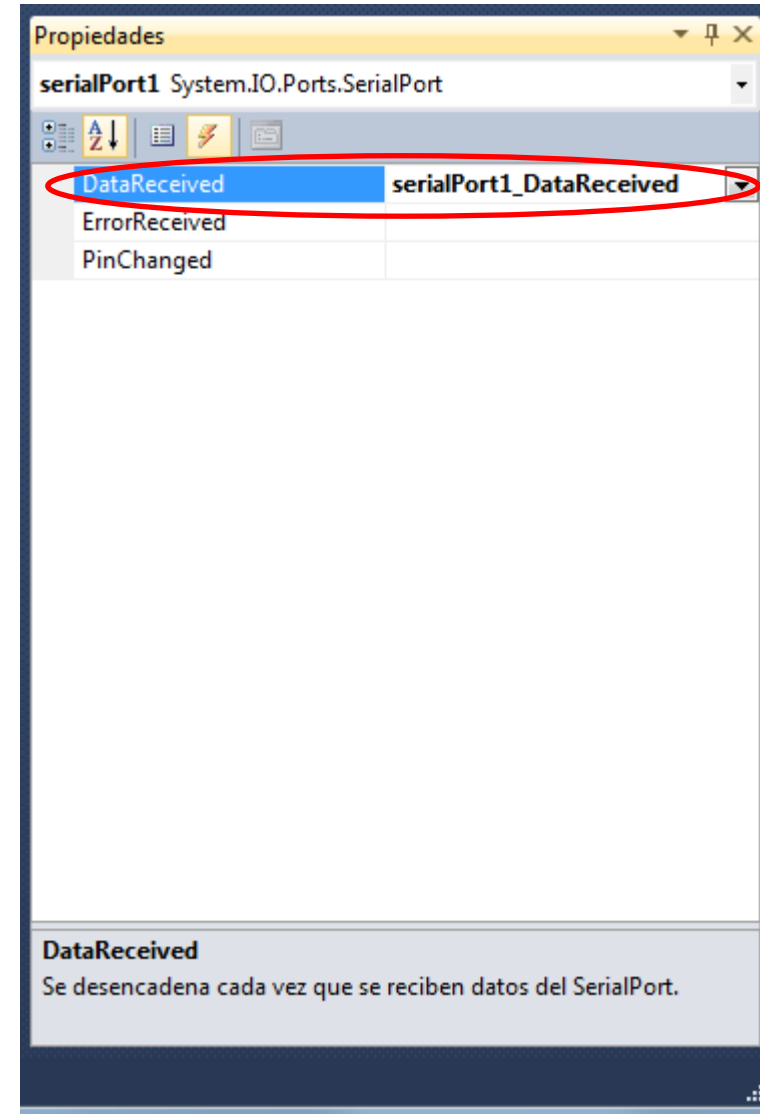
# Configurando el puerto serie

- Hay que configurar:
  - **BaudRate:** 115200
  - **PortName:** depende de cada ordenador, mirar en el X-CTU.



# Configurando el puerto serie

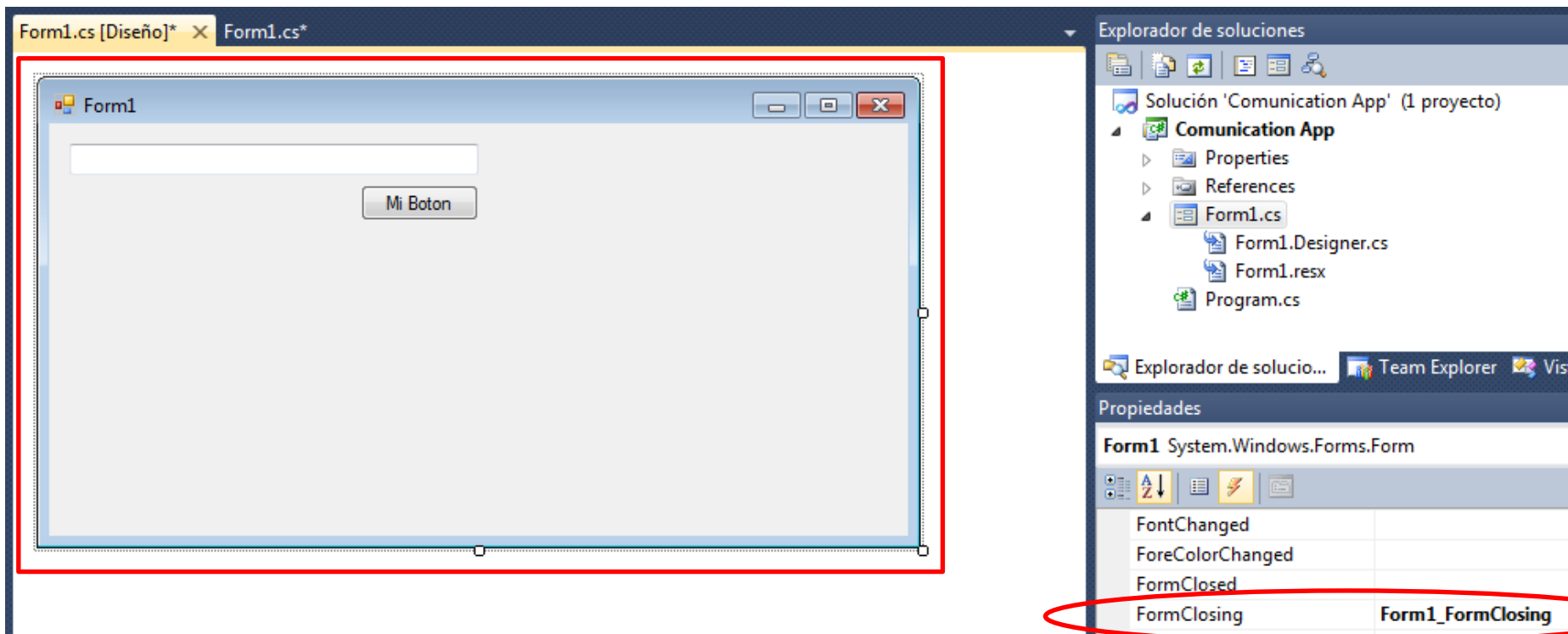
- Añadir evento de datos recibidos.





# Evento FormClosing del Formulario

- Seleccionar el formulario.
- Añadir evento **FormClosing**.



# Abriendo el puerto serie

- Abrir el puerto al iniciar el formulario.
- Cerrar el puerto al cerrar el formulario.

```
namespace Communication_App
{
    public partial class Form1 : Form
    {
        public int i;

        public Form1()
        {
            InitializeComponent();
            i = 0;
            //Abrir el puerto serie al inicio.
            serialPort1.Open();
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            //Cerrar el puerto serie al cerrar el formulario.
            serialPort1.Close();
        }
    }
}
```

# Recibiendo datos

- Completar el evento de recepción de datos para ir extrayendo uno a uno los bytes del buffer de recepción.

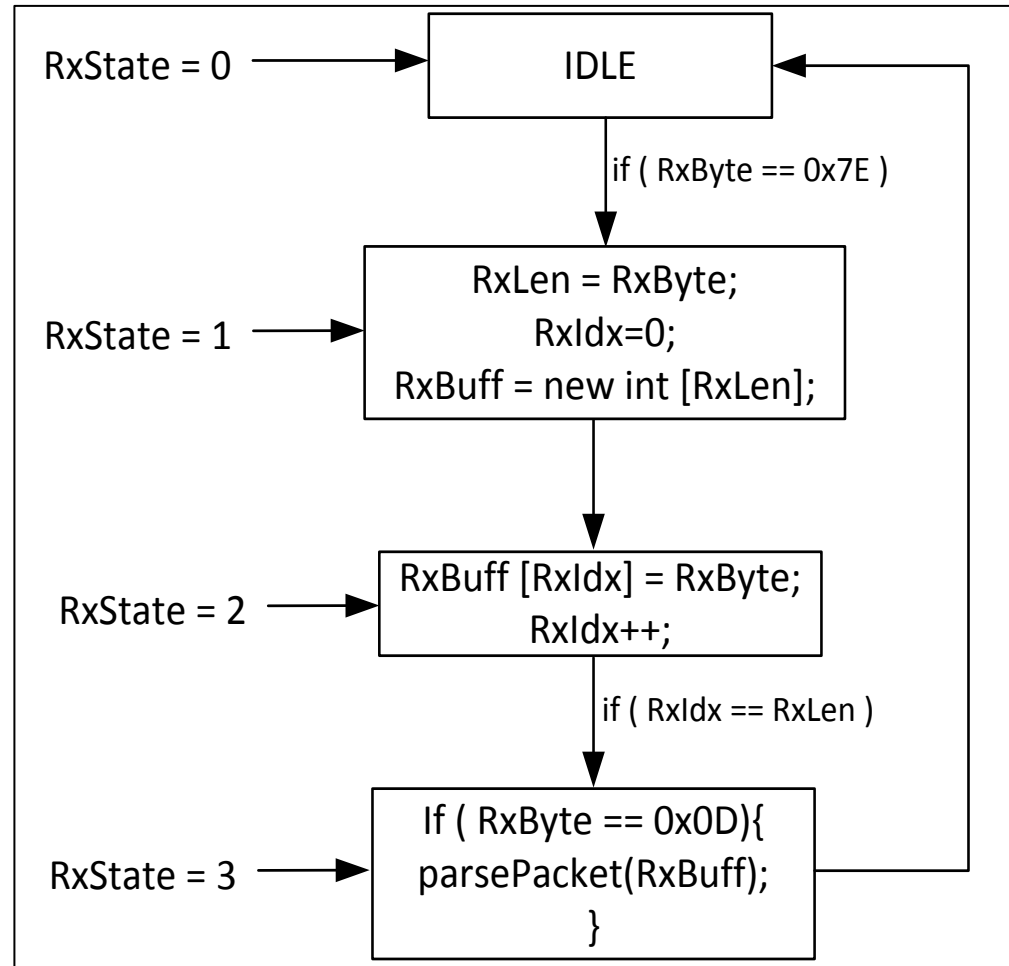
```
private void serialPort1_DataReceived(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)
{
    int b;
    while (serialPort1.BytesToRead > 0)
    {
        b = serialPort1.ReadByte();
        //Procesar el dato recibido
        rxByteStateMachine(b);
    }
}
```

# Recibiendo datos

- Añadir el método **RxByteStateMachine(int RxByte)** que implemente una máquina de estado asíncrona para la recepción de un paquete.
- El método será invocado byte a byte, y deberá controlar la recepción de un paquete.
- Al finalizar la recepción de un paquete, este deber ser decodificado (parseado), más adelante se implementará una función que realice esta función.
- Usar atributos del formulario como variables globales:
  - **Números enteros:**
    - **RxState:** Estado de recepción.
    - **RxLen:** Longitud de los datos a recibir.
    - **RxIdx:** Índice de los datos recibidos.
  - **Array de enteros:**
    - **RxBuff:** Buffer de recepción.

# Máquina de estados de recepción (C#)

- Usar **RxState** para controlar el estado.
- Implementar el cuerpo con una estructura switch/case sobre **RxState**.



# Máquina de estados de recepción (C#)

```
public int RxStatus, RxLen, RxIdx;
public int [] RxBuff;

public void RxByteStateMachine(int RxByte)
{
    switch (RxStatus)
    {
        case 0:    //Inicio del paquete
            if (RxByte == 0x7E) //Si se recibe el inicio de trama
            {
                RxStatus = 1;    //Estado de longitud
            }
            break;
        case 1:    //Longitud
            //...
            break;
        case 2:    //Recepción de datos
            //..
            break;
        case 3:    //Fin del paquete
            if (RxByte == 0x0D) //Si se recibe el fin de trama
            {
                //Descodificar el paquete
                ParsePacket(RxBuff);
            }
            RxStatus = 0;    //Vuela al estado inicial
            break;
        default:
            break;
    }
}
```

# Comprobando la recepción datos

- Depurar la máquina de estados anterior.
- Comprobar que pasa por todos estados y almacena 5 bytes en **RxBuff**.
- Finalmente se debe invocar **ParseByte**.

# Descodificando la información

- Declarar 3 atributos de tipo entero para almacenar la información recibida:
  - **AnalogJoyX**: valor del eje X.
  - **AnalogJoyY**: valor del eje Y.
  - **DigitalJoy**: número de botón pulsado.
- Nuevo método: **ParsePacket (int [] Packet)**:
  1. Descodificar la información.
  2. Almacenarla en las variables globales.
  3. Refrescar el formulario.



# Descodificando la información

- El método **ParsePacket** recibe el paquete sin la cabecera / cola.
- Desde su punto de vista:

Off-set	Descripción	Dato	Interpretación
+0	Dato [0]	0x01	A. Joy X MSB
+1	Dato [1]	0x55	A. Joy X LSB
+2	Dato [2]	0x00	A. Joy Y MSB
+3	Dato [3]	0xAA	A. Joy Y LSB
+4	Dato [4]	0x04	Joy Digital

- Reconstruir uno a uno los valores enviados, y almacenarlos en las variables globales.
- Imprimir en **textBox1** el valor recibido como un String.

# Descodificando la información

```
int AnalogJoyX, AnalogJoyY, DigitalJoy;

public void ParsePacket(int[] Packet)
{
    //Descodificación del eje X
    AnalogJoyX = (Packet[0] << 8) | Packet[1];
    //Descodificación eje Y
    //...
    //Descodificación Joy Digital
    //...

    textBox1.Text = "Eje X: " + AnalogJoyX + "Eje Y...";
}
```

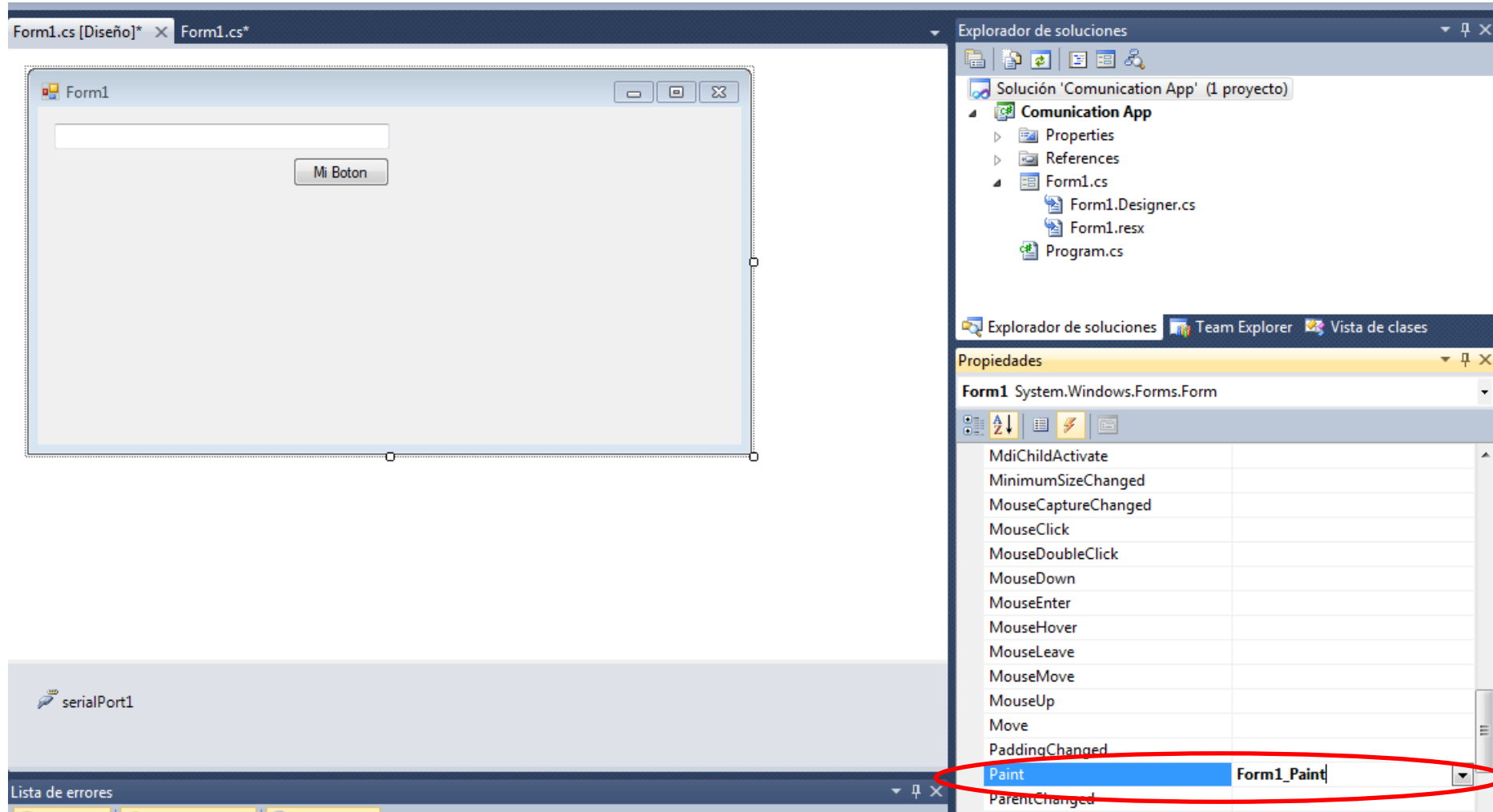
# Ejecutar y comprobar su funcionamiento

- Ejecutar el método **ParsePacket** instrucción a instrucción.
- Comprobar que almacena correctamente la información en las variables globales.
- **OJO: Excepción!!**

# Actualizando el formulario

- **El evento de recepción del puerto serie se ejecuta en un hilo distinto al del formulario principal.**
- Al intentar modificar un control (**textBox1**) desde otro hilo provocamos una excepción.
- **Solución:**
  - Forzar desde el evento el repintado del formulario.
  - Actualizar el formulario en el evento Paint.

# Añadir el evento Paint al formulario principal



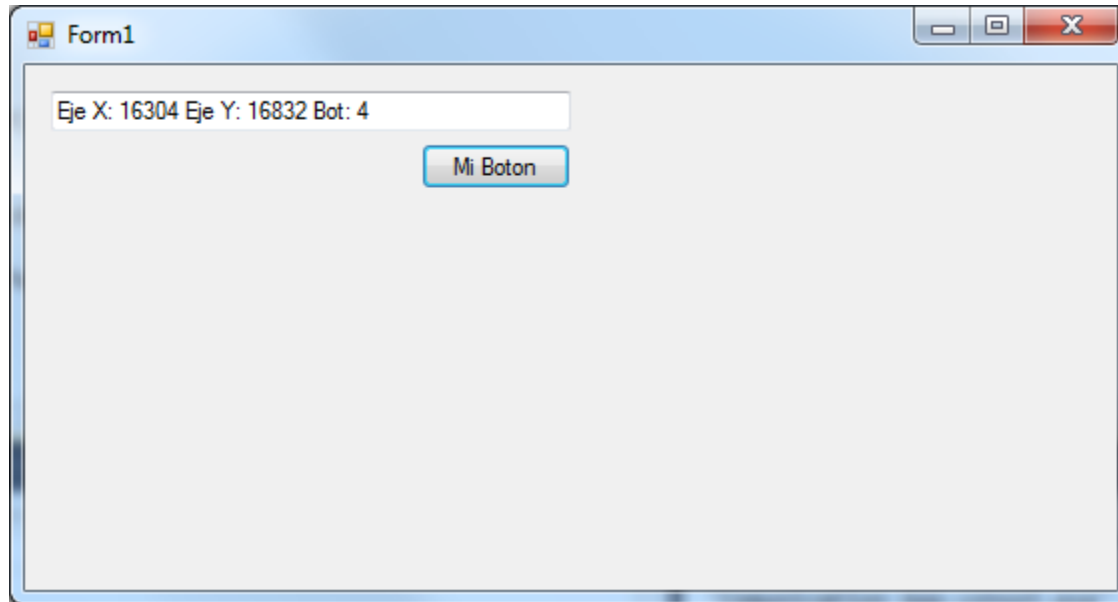
# Invalidando y refrescando el formulario

```
public void ParsePacktet(int[] Packet)
{
    //Descodificación del eje X
    AnalogJoyX = (Packet[0] << 8) | Packet[1];
    //Descodificación eje Y
    //...
    //Descodificaicón Joy Digital
    //...

    this.Invalidate();
}

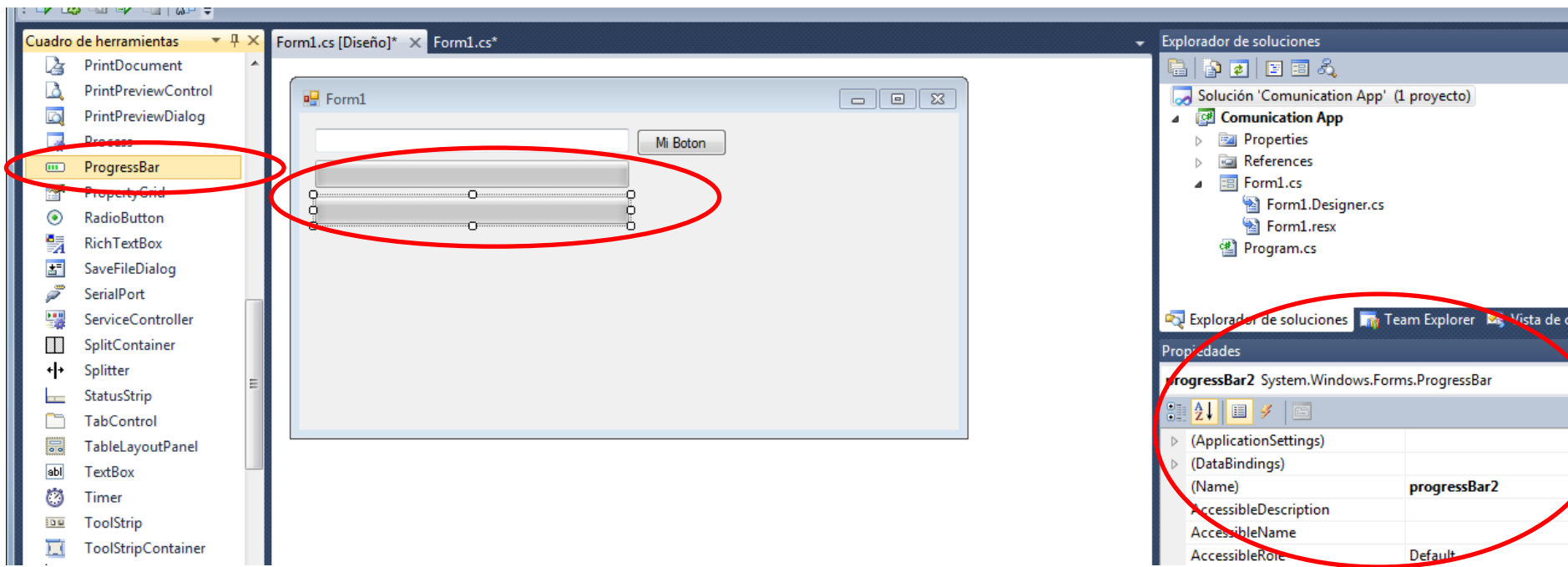
private void Form1_Paint(object sender, PaintEventArgs e)
{
    textBox1.Text = "Eje X: " + AnalogJoyX + "Eje Y...";
}
```

# Ejecutando la Aplicación



# Añadiendo barras de progreso

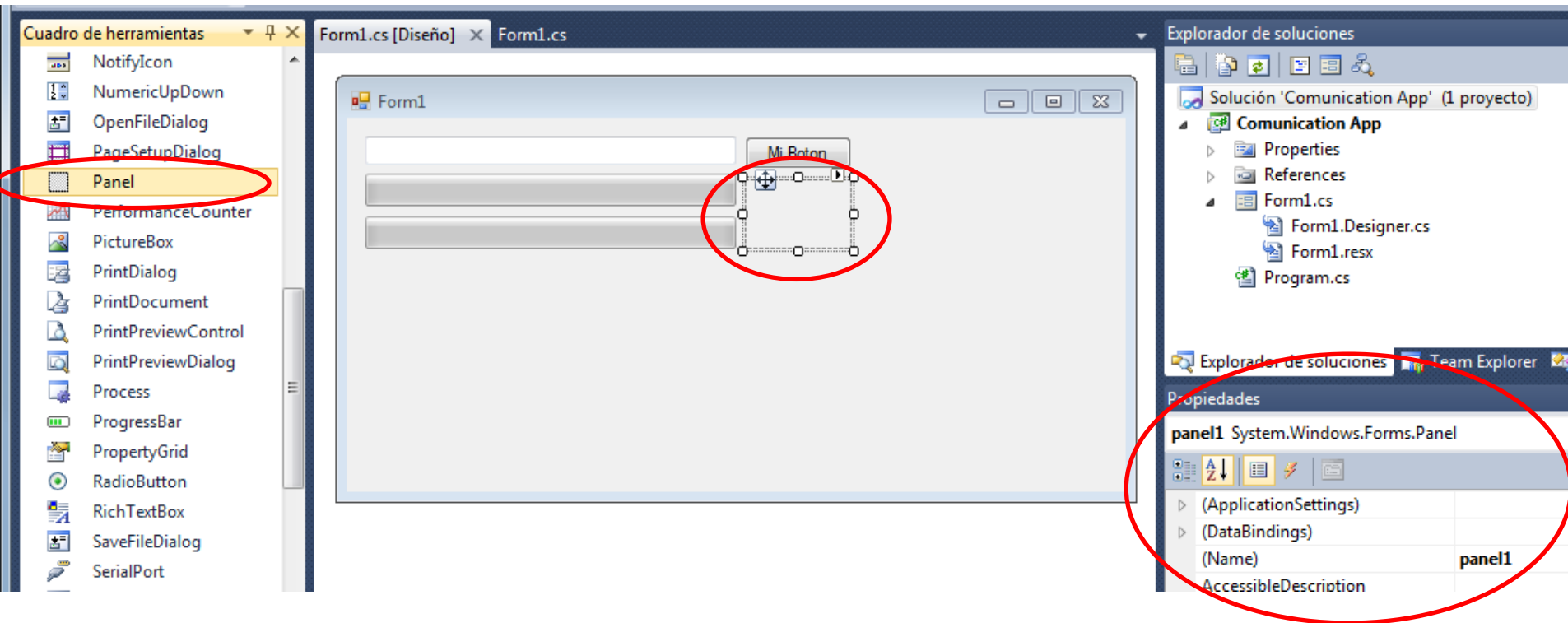
- Añadir dos barras de progreso a la interfaz para representar el joystick analógico.
- Modificar sus propiedades de ambas barras.:
  - Maximum = 4096
- Modificar el evento Paint del formulario:
  - Refrescar el valor de cada ProgressBar con el valor de cada uno de ejes del joystick.
  - Ejemplo: **progressBar1.Value = Valor;**



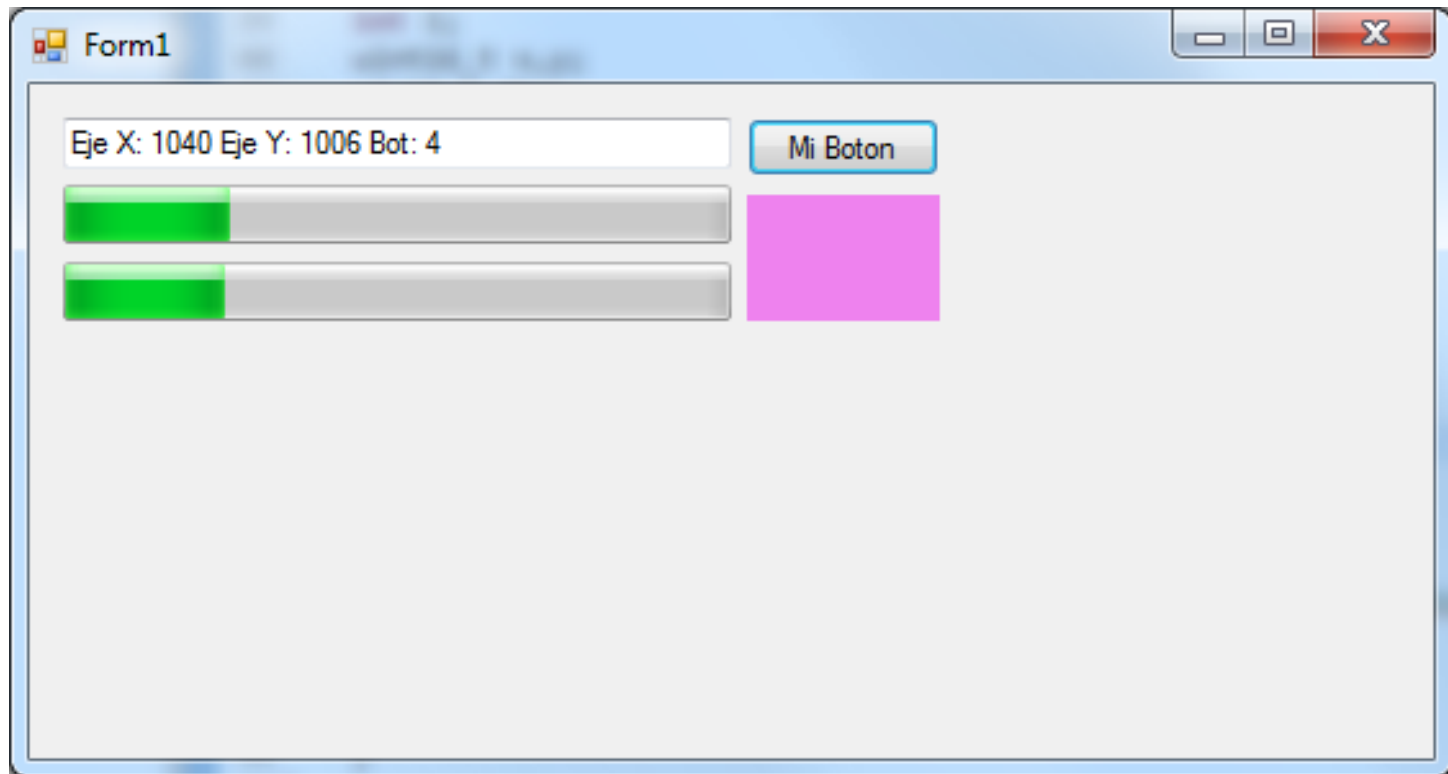


# Añadiendo un Panel

- Añadir un Panel a la interfaz.
- Cambiar el color del panel dependiendo del botón pulsado en el joystick digital.
- Ampliar el evento Paint del formulario para cambiar el color de fondo del panel dependiendo del botón pulsado:
  - Ejemplo: **panel1.BackColor = Color.Red;**



# En Ejecución



## **Parte 2: Transmitiendo comandos desde C# - Recibiendo en el STM32**

# Enviando comandos al CoOs

- La idea es enviar comandos desde la aplicación en C# al STM32.
- Los comandos que vamos a enviar van a estar destinado a los “actuadores” incorporados en la placa:
  - Lanzar animaciones de leds.
  - Posición servo.
  - Velocidad / Dirección del motor.
- Definiremos un formato de paquete para cada elemento.

# Empaquetando los comandos

- Todos los comandos van a ser paquetes de 2 bytes:
  - Comando: Carácter Ascii.
  - Valor: Entero de 8 bits

Comando	Valor	Descripción
'L'	0 a 3	Lanza la animación de leds indicada.
'S'	0 a 180	Fija la posición del servo.
'M'	-128 a 128	Establece la velocidad y sentido del motor.

# Creación y transmisión de comandos

- En VisualStudio 2010.
- Crear un nuevo método en el formulario:
  - **public void TxCmd (char cmd, byte value)**
    - Declaración de un array de 5 bytes.
    - Rellenar el array con los valores adecuados.
    - Escribir en el puerto serie:
      - **serialPort1.Write(buffer, offset , length);**

Off-set	Descripción	Dato	Interpretación
+0	Inicio de trama	0x7E	
+1	Longitud de la trama	0x02	
+2	Comando	'L'	Comando para los Leds
+3	Valor	0x02	Número de animación (de 0 a 3).
+4	Fin de trama	0x0D	

# Comenzando con los Leds

- Llamar al método **TxCmd** cada vez que se pulse el botón con el comando 'L'.
- Incrementar el valor del parámetro cada vez que se pulse el botón.
- Mantener el parámetro en el rango adecuado: de 1 a 4.

# Enviado comandos a los Leds

```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = "Botón Pulsado!: " + i;
    i++;
    if (i == 4)
    {
        i = 0;
    }
    TxCmd('L', (byte) i);
}

public void TxCmd(char cmd, byte value)
{
    byte[] buffer = new Byte[5];

    buffer[0] = 0x7E;
    buffer[1] = 2;
    buffer[2] = (byte) cmd;
    buffer[3] = value;
    buffer[4] = 0x0D;

    serialPort1.Write(buffer, 0, 5);
}
```



# Recibiendo datos en el STM32

- Volviendo al CooCox, en **SerialTask.c**, modificar la tarea **serialRxTask**.
- Declarar en la tarea las variables necesarias para implementar recepción: RxLen, RxIdx, RxBuff
- No es necesario un buffer de recepción grande, con 8 elementos es suficiente.
- Implementar la máquina de estados de recepción asíncrona, pero en versión iterativa.
- Esperar a recibir bytes en la cola de recepción:
  - **c=CoPendQueueMail(queueRxId,0,&err);**
- Se puede usar un simple bucle **for** para rellenar **RxBuffer** con los datos recibidos.
- Al finalizar la recepción de un paquete, llamar a la función:
  - **parsePacket ( RxBuffer);**

# Recepción bloqueante en CoOos

```
void serialRxTask(void * parg){
    uint8_t c = 0x00;
    uint8_t RxLen = 0;
    uint8_t RxIndex = 0;
    uint8_t RxBuffer[8];
    StatusType err;

    for(;;){
        c = 0;
        //1- Esperar recepcion inicio trama
        while(c != 0x7E){
            c=CoPendQueueMail(queueRxId,0,&err);
        }
        //2- Longitud del paquete
        RxLen = CoPendQueueMail(queueRxId,0,&err);
        //3- Recepcion de datos
        //for (.....)
        //...
        //4- Fin de tranmision
        //...
        if(c == 0x0D){
            parsePacket(RxBuffer);
        }
    }
}

void parsePacket (uint8_t * buff){
    CoSetFlag(1);
}
```

# Decodificando los comandos

- La función **parsePacket** se encargará de interpretar los comandos.
- El primer byte de **RxBuff** contiene el comando.
  - Implementar una estructura **switch/case** en la función **parsePacket** sobre **RxBuff[0]** que seleccione entre los comandos: 'L', 'S', 'M'.
- Recodar función:
  - **SetFlagKey (nLed)**

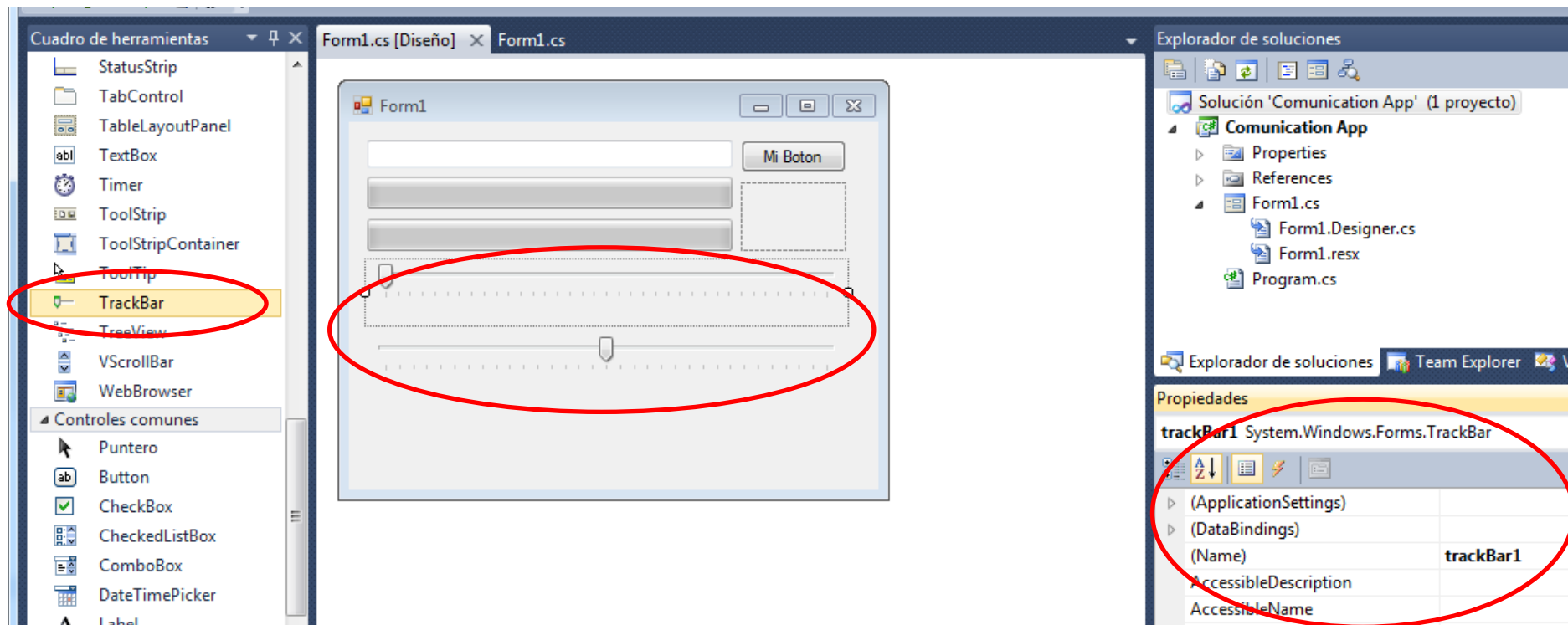
```
void parsePacket (uint8_t * buff){  
    switch (buff[0]){  
        case 'L':    //Animacion Leds  
            SetFlagKey(buff[1]);  
            break;  
        case 'S':    //Posición Servo  
            //...  
            break;  
        case 'M':    //Velocidad/Sentido Motor  
            //...  
            break;  
    }  
}
```

# Comprobando la recepción de los comandos

- Compilar y lanzar el código anterior en el STM32.
- Volver a la aplicación en C#.
- Enviar el comando 'L' usando el botón de la aplicación.
- Comprobar que se lanzan las animaciones y la comunicación funciona correctamente.

# Enviando comandos al Servo/Motor

- Añadir dos TrackBars.
- Representarán:
  - Ángulo del Servo.
  - Velocidad del Motor.



# Enviando comandos al Servo/Motor

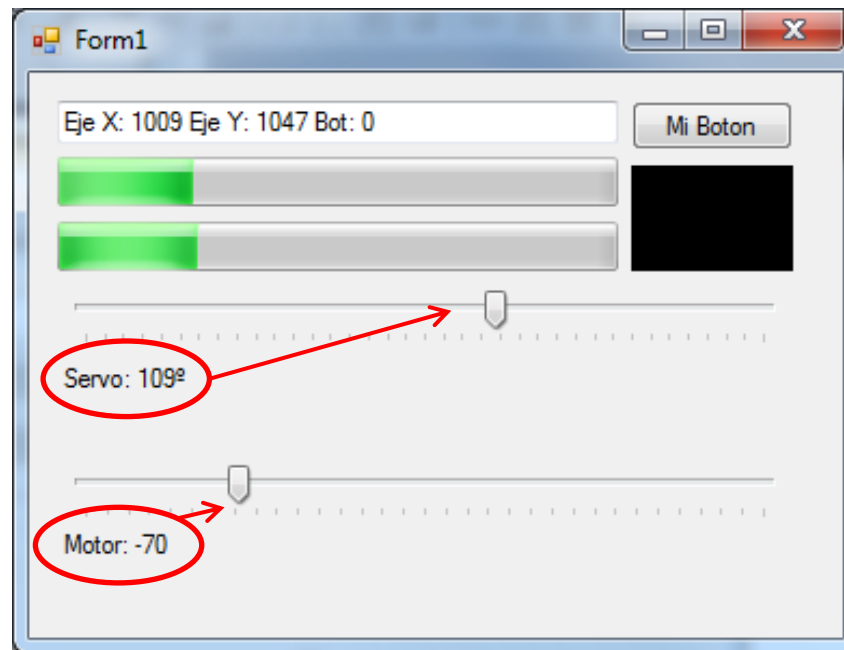
- Modificar las propiedades de cada TrackBar:
  - **trackBar1:**
    - Maximum = 180
    - Minimun = 0
  - **trackBar2:**
    - Maximum = 127
    - Minimun = -127
- Añadir para cada barra el evento **Scroll**. Se disparará al mover la barra.
  - **trackBar1:** enviar el comando 'S' junto con el valor de la barra.
  - **trackBar2:** enviar el comando 'M' junto con el valor de la barra.

# Recibiendo comandos de Servo/Motor

- Volviendo al **CooCox**, en **SerialTask.c**, completar la función **parsePacket** para que ejecute los comandos 'S' y 'M'.
- Usar las funciones proporcionadas:
  - **void setServoPos(uint16\_t grados);**
  - **void setMotorSpeed(int8\_t speed);**

# Terminando la aplicación en C#

- Añadir dos etiquetas de texto (**Label**).
  - Grados del servo.
  - Velocidad del motor.
- Modificar los eventos **Scroll** de las **TrackBars** para que actualicen el texto de las **Labels** al cambiar de valor.
  - `label1.Text = "... " + var + "... "`;
- Modificar el constructor del formulario para que proporcione a los **Labels** de un texto inicial al lanzarse el formulario.





# Terminando la aplicación en C#

- Añadir dos nuevos botones.
  - Centrar servo.
  - Parar motor.
- Modificar el nombre de cada botón.
- Añadir eventos Click a cada botón:
  - **Servo:** Enviar posición 90º, actualizar el valor del **TrackBar** del servo.
  - **Motor:** Enviar velocidad 0, actualizar el valor del **TrackBar** del motor.

Form1

Eje X: 1016 Eje Y: 1047 Bot: 0

Mi Boton

Servo: 90º

Center Servo

Motor: -70

Stop Motor

Form1

Eje X: 1019 Eje Y: 1058 Bot: 0

Mi Boton

Servo: 90º

Center Servo

Motor: 0

Stop Motor