Tutorial de programación para Pocket PC

Para empezar…

Notas sobre la implementación

El código presentado en este tutorial solo ha sido probado usando el .NET Framework 2.0 y .NET Compact Framework 2.0 y en la plataforma Pocket PC con Windows Mobile 2003 y 2005, no se garantiza el correcto funcionamiento en ambientes anteriores a estas versiones o plataformas diferentes.

Información sobre este tutorial

Última modificación: 18 de abril de 2007

Tabla de contenido

Resumen

¿Por qué la existencia de este tutorial?

Este tutorial fue creado debido a la poca difusión del desarrollo para la plataforma Windows Mobile específicamente usando la tecnología Microsoft .NET, la cual, debido al gran auge de Java se ha visto un poco opacado, es por eso que aquí presentamos algunas de las características de .NET 2.0 y C# 2.0 y como puede ser usado para desarrollar aplicaciones útiles que exploten el uso de la conectividad inalámbrica de las Pocket PC y Smartphones.

¿Para quién va dirigido?

Desarrolladores que han tenido poca experiencia trabajando con Microsoft C#.

¿Cuál es el objetivo?

Servir de introducción al desarrollador que le interese crear aplicaciones para dispositivos móviles que soporten la tecnología .NET Compact Framework 2.0 usando el lenguaje C# en su versión 2.0

Primer paso
Preparando el entorno de desarrollo

Para empezar a desarrollar aplicaciones para dispositivos con Windows Mobile primeramente necesitamos tener instalado el IDE de Microsoft Visual Studio 2005, el .NET Compact Framework 2.0 y el Windows Mobile SDK 5.0, que es la herramienta que nos permitirá crear proyectos de aplicaciones para dispositivos móviles.

Instalando Visual Studio 2005

Si aun no tienes experiencia usando esta herramienta, es recomendable que descargues la versión de prueba de la página de Microsoft para poder proseguir con el resto del tutorial, este es el link que te llevara a la descarga de la versión de prueba por 90 días.
Visual Studio ofrece todo lo necesario para empezar a desarrollar aplicaciones para Windows con C#, Visual Basic, J#, C++ y ASP .NET. Este tutorial se enfoca en C# porque es el lenguaje que mejores características ofrece para desarrollar este tipo de aplicaciones.
Visual Studio ya trae incorporado en la instalación .NET Framework 2.0

Visual Studio 2005 Professional

Instalando .NET Compact Framework 2.0 SP2 (2.0.2)

.NET Compact Framework es la tecnolog{ia con la cual es possible desarrollar y correr aplicaciones creadas con Visual Studio para dispositivos móviles con Windows Mobile.
Esta versión es compatible con dispositivos Pocket PC, Pocket PC Phone Edition, Smartphone y otros con sistema Windows CE.

.NET Compact Framework 2.0 SP2

Este paquete usa ActiveSync 4.0 o posterior para instalar el paquete en tu dispositivo móvil, si no cuentas con ActiveSync, descargalo de aquí:

ActiveSync 4.5

Instalando Windows Mobile 5.0 SDK para Pocket PC

Para desarrollar aplicaciones para Windows Mobile con Visual Studio es necesario instalar el kit de desarrollo (SDK) para esta plataforma. Con esto se agregaran varios tipos de proyecto en Visual Studio que podrán usar las clases del .NET Compact Framework.

Windows Mobile 5.0 SDK para Pocket PC

Ahora que tenemos instaladas estas herramientas ya podemos empezar a programar nuestras aplicaciones móviles, en el siguiente paso veremos como hacer una aplicación sencilla para Windows Mobile.

Segundo paso
Nuestra primera aplicación móvil

Ahora veremos algunas de los aspectos fundamentales para crear una aplicación móvil. Empezaremos con una sencilla aplicación que lo que hará es dibujar las tres funciones trigonometricas básicas seno, coseno y tangente mediante una interfaz sencilla. Para esto usaremos Windows Forms y GDI+ que explicare en un momento para quienes no han usado estas clases. Cabe mencionar que .NET Compact Framework (.NET CF desde ahora) no contiene todas las clases base que .NET Framework (.NET F desde ahora) debido claramente a que tiene que ser portable a dispositivos móviles, sin embargo tiene toda la funcionalidad que puedas requerir para desarrollar todas las aplicaciones que puedas requerir.

GDI+

En Windows, el acceso al subsistema de gráficos se hacia en un principio usando las APIs GDI disponibles desde Windows 3.1, GDI ofrecía a los desarrolladores la capacidad de controlar cualquier tipo de elemento de una interfaz, y esta característica ha sido reconstruida totalmente en .NET Framework. GDI+ ha remplazado a GDI como el API para acceder a los subsistemas gráficos de Windows. Con GDI+, puedes acceder a fuentes, manipular cualquier tipo de imagen, y trabajar con shapes (usaremos algunas palabras técnicas para evitar ambigüedades) en tus aplicaciones de C#. Para tener una idea clara de cómo usar GDI+ necesitas un buen entendimiento de los objetos Graphics, Pen, Brush y Color. Con estos cuatro objetos, puedes hacer casi cualquier cosa que necesites con el GUI e imágenes en .NET. Esta sección explora algunos de estos objetos y te familiariza con el uso de GDI+ en C#.

Windows Forms

Para poder crear aplicaciones para Windows se necesita hacer uso de las clases que se encuentran dentro del namespace System.Windows.Forms, el cual no esta disponible por default en un nuevo proyecto por lo que se tiene que referenciar con el menú de Project.
System.Windows.Forms contiene todas las clases que necesitas para construir aplicaciones de escritorio en el caso del .NET F y en el caso del .NET CF para aplicaciones móviles. Estas clases te permiten trabajar con formas, botones, controles de edición, cajas de selección, listas y muchos otros elementos de interfaz de usuario.

Las aplicaciones Windows Forms hacen uso de dos clases fundamentales del .NET F: la clase Form, que maneja las formas de las aplicaciones y los controles puestos en la forma, y la clase

Application, que maneja el control de mensajes de Windows de la aplicación enviados y recibidos de las formas de la aplicación.

Creando la aplicación inicial

Para empezar a programar nuestra aplicación necesitamos primero crear un proyecto en Visual Studio de tipo **Device Application**, para hacerlo en **Visual Studio** ve al menú File -> New -> Project y selecciona **Device Application** como tipo de proyecto y de nombre Ejemplo1 como se muestra en la Figura 1.
Como podremos ver al presionar OK que **Visual Studio** nos genera varios archivos como vemos en el **Solution Explorer** y una vista de una Forma que se asemeja al de una Pocket PC en el **Designer** como se muestra en la Figura 2.

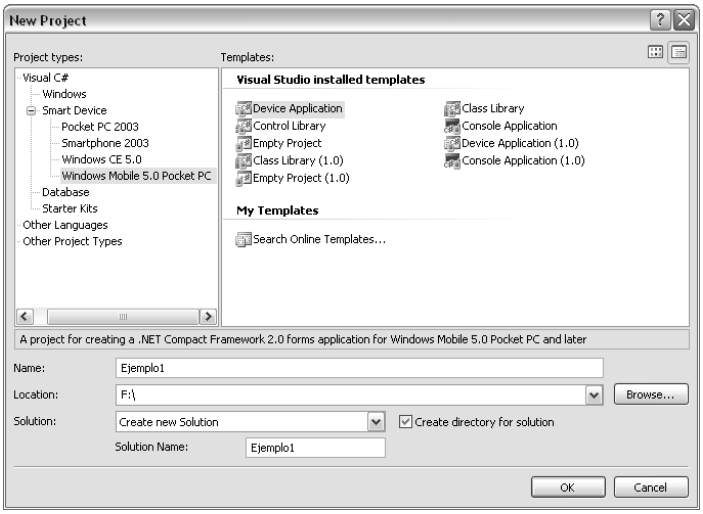

Figura 1. Creando un Device Application Project

Figura 2. Vista del Designer de un nuevo proyecto

Creando la interfaz de usuario

Lo que sigue ahora es crear la interfaz de usuario (**GUI** desde ahora) que queremos para nuestra aplicación, podremos darnos cuenta en nuestro **Toolbox** que podemos hacer uso de una gran variedad de controles de Windows predefinidos para crear una GUI. Como ya había mencionado el **.NET CF** tiene menos características que **.NET F** y eso también se refleja en la variedad de controles predefinidos que se pueden usar, sin embargo se encuentran los controles mas usados por lo que no será problema crear un **GUI** funcional. En la Figura 3 podemos ver los controles disponibles.

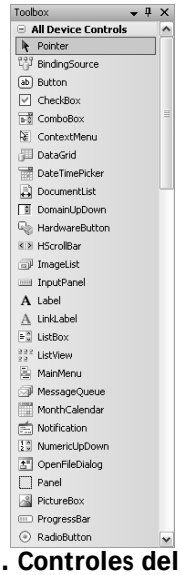

Figura 3. Controles del Toolbox

Primeramente agregaremos 3 **RadioButton** simplemente seleccionando el control correspondiente y arrastrándolo a la **Forma** que se usaran para seleccionar que función queremos dibujar. En la ventana de **Properties** podemos asignarle una etiqueta con **Text** y un nombre para identificar al control además a uno de los 3 controles le asignaremos a la propiedad **Checked** el valor **true** como se muestra en la Figura 4. Al nombre de la Forma en la que estamos trabajando cambia la propiedad **Name** por **MainForm**.

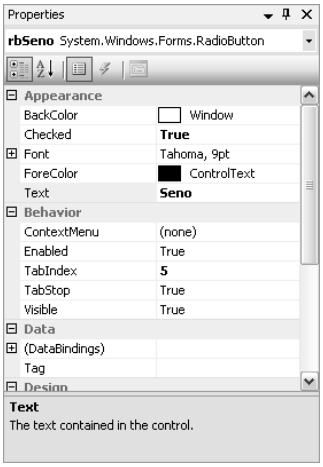

Figura 4. Asignación de propiedades

Ahora agregamos dos botones normales con el control **Button** uno servirá para **Graficar** la función y otro para **Limpiar** el dibujo.

Para graficar las funciones no necesitaremos ningún control predefinido, en su defecto programaremos un área de dibujo mediante **GDI+**. Al final la interfaz debe parecerse a la de la Figura 5.

Figura 5. Interfaz final

Programación de la aplicación

Ahora q tenemos terminada la **GUI** podemos continuar programando los comportamientos y los gráficos. Antes que nada usaremos una clase que definiremos en un archivo aparte de preferencia que llamaremos **GraphicsVars** que contendrá información relacionada con el área de dibujo además es importante que incluyas en el encabezado declaremos que usaremos las clases del **namespace System.Drawing** y el cuál también tienes que incluir una referencia en las propiedades del proyecto, en el Listado 1 veremos los campos de la clase que servirán para definir el área de dibujo.

Listado 1. Campos de la Clase GraphicsVars

```
class GraphicsVars
{
    // VARIABLES GEOMETRICAS
    // Ancho del plano de dibujo
    const int _planeWidth = 226;
    // Alto del plano de dibujo
    const int _planeHeight = 96;
    // Coordenada X del Origen del plano
    const int _planeX0 = _planeLeft + (_planeWidth / 2);
    // Coordenada Y del Origen del plano
    const int _planeY0 = _planeTop + (_planeHeight / 2);
    // Coordenada X del borde izquierdo del plano
    const int _planeLeft = 4;
```

```
// Coordenada Y del borde superior del plano
const int _planeTop = 4;
// Coordenada X del borde derecho del plano
const int _planeRight = _planeLeft + (_planeWidth);
// Coordenada Y del borde inferior del plano
const int _planeBottom = _planeTop + (_planeHeight);
// "Resolucion" de la gráfica
const int _resolution = 720;
// "Amplitud" de la funcion trigonométrica
const int _curveAmp = 30;
// COLORES DE TINTA
// Brocha solida
SolidBrush _blackBrush;
// Color de linea para las funciones
Pen _lightGreenPen;
// Color de linea para los ejes del plano
Pen _yellowPen;
```

Podemos observar que definimos bastantes constantes y cada una tiene un rol en el correcto funcionamiento de la aplicación, las primeras dos definen las dimensiones, **_planeX0** y **_planeYO** definen la coordenada (0,0) relativa a las coordenadas reales del área de la Pocket PC que dejamos para ese fin, las siguientes 4 variables nos servirán para saber las coordenadas de cada uno de los 4 fronteras del plano, después **_resolution** servirá para definir el numero de puntos con que contara la curva de la gráfica, en este caso usaremos 720 puntos que son suficientes para ver una curva suave y finalmente **_curveAmp** se usará para controlar la amplitud de las funciones. Como podemos ver aquí también definimos 1 objeto de tipo **SolidBrush** con el que podemos rellenar con algún color un gráfico que hayamos definido, en este caso será para pintar de negro el plano y también tenemos 2 objetos tipo **Pen** con el que dibujaremos las funciones y los ejes respectivamente.

A continuación en el Listado 2 definimos el constructor donde simplemente inicializamos los objetos **SolidBrush** y **Pen** que declaramos anteriormente como podemos ver sus constructores esperan una constante de tipo **Color** que define varios colores predefinidos que podemos usar, la clase **Pen** además espera un valor entero que representa el grosor de línea que usaremos además también listamos las propiedades para poder obtener los valores de las constantes definidas.

Listado 2. Constructor y Propiedades de la clase GraphicsVars

```
public GraphicsVars()
{
    // COLORES DE TINTA
    _blackBrush = new SolidBrush(Color.Black);
    _lightGreenPen = new Pen(Color.LightGreen, 1);
    _yellowPen = new Pen(Color.Yellow, 1);
}
public int PlaneWidth
{
    get { return _planeWidth; }
}
public int PlaneHeight
{
    get { return _planeHeight; }
}
public int PlaneX0
```

```
        {
            get { return _planeX0; }
        }
        public int PlaneY0
        {
            get { return _planeY0; }
        }
        public int PlaneTop
        {
            get { return _planeTop; }
        }
        public int PlaneLeft
        {
            get { return _planeLeft; }
        }
        public int PlaneRight
        {
            get { return _planeRight; }
        }
        public int PlaneBottom
        {
            get { return _planeBottom; }
        }
        public int Resolution
        {
            get { return _resolution; }
        }
        public int CurveAmp
        {
            get { return _curveAmp; }
        }
        public SolidBrush BlackBrush
        {
            get { return _blackBrush; }
        }
        public Pen LightGreenPen
        {
            get { return _lightGreenPen; }
        }
        public Pen YellowPen
        {
            get { return _yellowPen; }
        }
```

Ya tenemos definida la clase q nos ayudara a definir los gráficos de nuestra aplicación, falta definir una pequeña estructura que nos ayudara a procesar operaciones matemáticas como convertir grados a radianes, en realidad en este ejemplo solo servirá para eso pero se podrían agregar otras funciones para aplicaciones mas complejas, el Listado 3 muestra el código.

Listado 3. Estructura de procesamiento matemático MathProcessing

```
    struct MathProcessing
    {
```

```
        public static double gradesToRadians(double grades)
        {
            return (grades * Math.PI) / 180;
        }
    }
```

Ya tenemos todo listo para empezar a programar el comportamiento de nuestra aplicación, así que en el **Solution Explorer** selecciona el archivo de la forma en la q hiciste el **GUI** anteriormente (**MainForm.cs**) y con click derecho selecciona la opción **View Code** para ver el código de la Forma.

Podemos ahora ver el código generado automáticamente por **Visual Studio**, lo primero que haremos será agregar dos campos a la clase **MainForm** como se muestra en el Listado 4. Primero creamos un objeto de tipo **GraphicsVars** llamada **gv** y a continuación una bandera **boolean** que servirá para indicarle al programa cuando debe limpiar el plano para volver a dibujar. También definimos la propiedad **ClientSize** que son los limites del control Form, menos los elementos no cliente como las barras de desplazamiento, bordes, barra de titulo y menus, ésta área se define para funciones con fines como el dibujado en la superficie del control. Esta área la definimos con la clase **Size** que define un área por su largo y ancho. Esta definición se debe hacer inmediatamente después de que se inicializan los controles de la forma con **InitializeComponent()** para evitar conflictos.

Listado 4. La clase MainForm

```
public partial class MainForm : Form
{
    // Inicializamos variables graficas
    GraphicsVars gv = new GraphicsVars();
    // Bandera de limpiado
    bool clearFlag = false;
    public MainForm()
    {
        InitializeComponent();
        this.ClientSize =
              new System.Drawing.Size(gv.PlaneWidth, gv.PlaneHeight);
    }
  }
```

Ya que tenemos inicializados los componentes todo el dibujado de nuestros objetos se hacen mediante el método **OnPaint()** de nuestra Forma, se define aquí porque queremos que cada vez que **MainForm** se vuelva a dibujar por eventos tales como la minimización y maximización de la ventana, también se vuelvan a dibujar nuestros gráficos, si éste código lo definiéramos por ejemplo en el evento **Load()** solo se dibujarían una sola vez así que **OnPaint()** es el candidato ideal para esta tarea. El Listado 5 muestra la primera parte de el código que definiremos.

Listado 5. OnPaint() y su argumento PaintEventArgs

```
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics dc = e.Graphics;
        }
```

En esta primera redefinimos el método **OnPaint()** de la clase **System.Windows.Forms.Control** y necesitamos que reciba un objeto **PaintEventArgs** que provee información del evento **Paint** que ocurre cuando el objeto se redibuja en este caso la Forma.
En el cuerpo de la función declaramos un objeto tipo **Graphics** de **System.Drawing.Graphics**. Está clase encapsula una superficie de dibujado para GDI+ por eso es q le asignamos la referencia del **Graphics** de **MainForm** por medio del evento que pasamos a la función.
Ahora empezaremos por dibujar el plano inicial sobre el que dibujaremos las funciones de la aplicación Dado que este método se llama cada vez que es necesario redibujar la Forma ahora vemos claro el uso de la variable auxiliar **clearFlag** para que solo pinte el plano vació una vez y cuando el usuario así lo quiera que será cuando presione el botón de Limpiar. En el Listado 6 tenemos el evento **Click** del botón **btLimpiar** que es como yo nombre a éste botón, recuerda que para programar este evento solo necesitas hacer doble click al botón en el **Designer** y automáticamente generará la función **Delegate** del evento.

Listado 6. Evento Click del botón btLimpiar

```
private void btLimpiar_Click(object sender, EventArgs e)
{
    clearFlag = true;
    this.Refresh();
}
```

Con este código decimos que establezca la bandera **clearFlag** a true, así la próxima vez que se llame a **OnPaint()** la aplicación sabrá que solo tiene q dibujar el plano vacío como veremos en el siguiente listado. La función **Refresh()** obliga a llamar **OnPaint()** porque hace que se redibuje la Forma. En el Listado 7 tenemos la segunda parte del metodo **OnPaint()**

Listado 7. OnPaint() y clearFlag==true

```
if (clearFlag == true)
{
    // Plano
    dc.FillRectangle(   gv.BlackBrush,
                        gv.PlaneLeft,
                        gv.PlaneTop,
                        gv.PlaneWidth,
                        gv.PlaneHeight);
    // eje x
    dc.DrawLine(    gv.YellowPen,
                    gv.PlaneLeft,
                    gv.PlaneY0,
                    gv.PlaneRight,
                    gv.PlaneY0);
    // eje y
    dc.DrawLine(    gv.YellowPen,
                    gv.PlaneX0,
                    gv.PlaneTop,
                    gv.PlaneX0,
                    gv.PlaneBottom);
    return;
}
```

Como podemos ver en el listado tenemos un **if** que verifica el valor de **clearFlag**, si es **true** limpia el plano si no continua con lo demás que es el dibujado de las funciones como veremos más

adelante, y cuando termina de dibujar el plano sale de la función con **return** esto para que no se dibuje una función si el usuario no lo ha indicado.
Tenemos tres instrucciones en el cuerpo del **if**, la primera dibuja un rectángulo con relleno con el método de la clase **Graphics FillRectangle** a través de nuestro objeto **dc** que es de tipo **Graphics**. Los argumentos de esta función son en primer lugar un objeto de tipo **Brush** que es usado para dibujar **Shapes** con relleno, así que le pasamos nuestro **BlackBrush** definido en nuestro objeto **GraphicsVars gv** así dibujaremos un rectángulo negro, los siguientes 4 argumentos son las coordenadas de los bordes del plano que también ya habíamos definido.
A continuación dibujamos los ejes con las funciones **DrawLine**, la diferencia con **FillRectangle** es que toma como primer argumento un objeto **Pen** que igual define un color pero para **Shapes** que no son rellenados con un color como el rectángulo anterior, los siguientes 4 argumentos definen las coordenadas x,y de el punto inicial y final de la recta relativas al punto superior izquierdo, q en este caso ya definimos en nuestra clase **GraphicsVars**.
Ya que tenemos el código para limpiar el plano o más bien redibujarlo pero que da ese efecto, ya podemos continuar con dibujar las funciones. Esta parte es mucho más compleja así que iremos paso a paso. En el Listado 8 definimos la condición para dibujar el plano cuando se trata de dibujarlo también con las funciones, esta condición permite que no redibujemos cuando no hay necesidad en realidad.

Listado 8. Dibujando no desperdiciando recursos

```
// Checamos clipping para ver si es necesario redibujar
if (e.ClipRectangle.Top < 77 &&
    e.ClipRectangle.Left < 232 &&
    e.ClipRectangle.Bottom > 2 &&
    e.ClipRectangle.Right > 2
    )
{
    // Plano
    dc.FillRectangle(   gv.BlackBrush,
                        gv.PlaneLeft,
                        gv.PlaneTop,
                        gv.PlaneWidth,
                        gv.PlaneHeight);
    // x axis
    dc.DrawLine(    gv.YellowPen,
                    gv.PlaneLeft,
                    gv.PlaneY0,
                    gv.PlaneRight,
                    gv.PlaneY0);
    // y axis
    dc.DrawLine(    gv.YellowPen,
                    gv.PlaneX0,
                    gv.PlaneTop,
                    gv.PlaneX0,
                    gv.PlaneBottom);
// abajo esta el resto del bloque IF...
```

Este código se debe encontrar inmediatamente después del código del listado anterior, podemos ver que usamos una condición algo extensa para permitir a la aplicación dibujar, la propiedad **ClipRectangle** de **Graphics** devuelve un objeto **Rectangle** que define las dimensiones del área de dibujo que se va a redibujar a causa de un evento **Paint**. Con esto podemos saber si el área a redibujar contiene alguna porción del plano, así si el área a redibujar no cubre alguna parte del plano podemos omitir redibujarlo, con esto ahorramos tiempo de procesador no haciendo

operaciones innecesarias. Lo que sigue es el mismo código para dibujar el plano, así que continuemos con el dibujado de las funciones dependiendo de la opción de los **RadioButton** que definimos. El Listado 9 muestra la programación del evento **Click** del botón para graficar.

Listado 9. Evento Click del botón btGraficar

```
private void btGraficar_Click(object sender, EventArgs e)
{
    clearFlag = false;
    this.Refresh();
}
```

En este evento lo único que hacemos es regresar la bandera a **false** para que al momento de dibujar no solo dibuje el plano sino la función seleccionada. En el Listado 10 tenemos la continuación del Listado 8.

Listado 10. El código para graficar las funciones

```
            // Puntos de grafica
            Point[] points = new Point[gv.Resolution];
            // Calculo de puntos
            double result = 0;
            for (double x = -360, offset = gv.PlaneLeft;
                 x < 360; x++, offset+=((double)gv.PlaneRight/720))
            {
                if(rbSeno.Checked == true)
                    result = System.Math.Sin(
                             MathProcessing.gradesToRadians(x));
                else if(rbCoseno.Checked == true)
                    result = System.Math.Cos(
                             MathProcessing.gradesToRadians(x));
                else if (rbTangente.Checked == true)
                    result = System.Math.Tan(
                             MathProcessing.gradesToRadians(x));

                if (Double.IsNegativeInfinity(result) ||
                    (gv.PlaneBottom/2 + (gv.CurveAmp * result)) < gv.PlaneTop)
                    points[(int)x + 360] = new Point(
                                Convert.ToInt32(offset),
                                Convert.ToInt32(gv.PlaneY0));
                else if (Double.IsPositiveInfinity(result) ||
                    (gv.PlaneBottom/2 + (gv.CurveAmp * result)) > gv.PlaneBottom)
                    points[(int)x + 360] = new Point(
                                Convert.ToInt32(offset),
                                Convert.ToInt32(gv.PlaneY0));
                else
                    points[(int)x+360] = new Point(
                                Convert.ToInt32(offset),
                                Convert.ToInt32(gv.PlaneY0 + (gv.CurveAmp * result)));
            }
            // Despliegue de grafica
            dc.DrawLines(gv.LightGreenPen, points);
        }// Fin de if()
    }// Fin de OnPaint()
```

Primeramente declaramos un arreglo de objetos de tipo **Point** este arreglo nos servirá para definir el número de puntos del que consistirá la gráfica, se puede aumentar para que se vea mas suave la curva o disminuir según sea necesario por las limitaciones del hardware.
Inmediatamente después se entra a un ciclo que calculará la posición de cada punto en base al ángulo **x** que va desde -360° a 360° esto lo hacemos haciendo uso de las funciones **Math.Cos, Math.Sin y Math.Tan** que reciben como parámetro el ángulo en radianes que calculamos a su vez

con la función que hicimos de la clase **MathProcessing** el programa sabe que función calcular verificando la propiedad **Checked** de los radio botones.
Para evitar valores infinitos causados por el cálculo de algunos valores de la tangente de una función usamos el método **IsNegativeInfinity** e **IsPositiveInifinity** de la clase **Double**, y para evitar valores fuera de la gráfica solo verificamos el valor de **result** como se muestra en los dos bloques **if** que lo unico que hacen en caso de encontrarse las situaciones ya mencionadas es establecer la coordenada **Y** del punto actual en el centro de la gráfica para eliminarlo prácticamente. En caso de que los puntos calculados se encuentren dentro de la gráfica entrando en el bloque **else** solo convertimos obtenido a uno relativo al plano de la gráfica.
Al final del ciclo solo llamamos a la función **DrawLines** para dibujar los puntos obtenidos, está función toma el objeto **Pen** y el arreglo **points** como argumentos.

Tercer paso
Corriendo la aplicación

Para correr la aplicación se puede hacer mediante el emulador integrado de Visual Studio o directamente en un dispositivo móvil, aquí presento las dos formas de hacerlo.

Usando el emulador

El emulador de Visual Studio es una útil herramienta para hacer pruebas de la aplicación sin necesidad de transferirla al dispositivo móvil, así ahorramos tiempo en las pruebas.
Para hacer esto debemos primero construir la aplicación con **Build Solution** como cualquier otro proyecto, después de construida la aplicación podemos usar la opción **Deploy Solution** la cual nos presentara una ventana para decidir si queremos enviar nuestra aplicación al emulador de nuestra PC o directamente enviarlo a nuestro dispositivo móvil a través de **Active Sync.** En la **Figura 6** se muestra la aplicación corriendo en el emulador y en la Pocket PC.

Figura 6. Aplicación emulada

Figura 7. Aplicación en ejecución

PARTE 2: Comunicación Bluetooth

Introducción teórica

Tecnología Bluetooth

Bluetooth es el nombre común de la especificación industrial **IEEE 802.15.1**, que define un estándar global de comunicación inalámbrica que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia segura, globalmente y sin licencia de corto rango.
Para este tutorial usaremos uno de los servicios que ofrece esta tecnología, éste es el Puerto Serie Virtual, con el que transmitiremos mensajes de nuestra PC a una Pocket PC y también veremos como puede hacerse entre dos Pocket PC que es de igual manera sencillo. Podriamos usar también el servicio de Red Bluetooth pero en este tutorial solo se harán comunicaciones seriales.

Adaptadores Bluetooth

En este tutorial asumo que tu PC no cuenta con tecnología Bluetooth, por lo que es necesario tener un adaptador que le de esa funcionalidad, en este caso lo más común es usar un dispositivo Adaptador USB-Bluetooth debido a su practicidad, aunque también existen tarjetas PCI que dan la funcionalidad de Bluetooth a la PC pero perderíamos portabilidad de esta forma, para este tutorial usare un adaptador usb-bluetooth marca **Encore Eelectronics modelo ENUBT-C1E** que no son muy caros y son de buena calidad e incluyen el software BlueSoleil para administrar los servicios y las conexiones de Bluetooth del dispositivo. Por experiencia personal no recomiendo usar adaptadores genéricos o de marcas no reconocidas porque pueden resultar de muy mala calidad.

Dispositivo Adaptador USB-Bluetooth usado para este tutorial

Primer paso
Aplicación de PC para el chat

¿Qué es lo que se va a hacer?

Antes de empezar a analizar el código es importante saber que para realizar comunicaciones mediante el puerto serie virtual tendremos que usar dos puertos, uno de entrada de datos y otro de salida de datos, para la aplicación de escritorio deberás consultar en el software incluido con tu adaptador que puertos de entrada y salida maneja tu dispositivo para prestar el servicio de Puerto Serie Virtual para así poder hacer la comunicación. A través del puerto de salida se enviaran mensajes y por el de entrada se recibirán.
.NET Framework 2.0 ya tiene incluido clases para trabajar con puertos seriales a través del namespace **System.IO.Ports** además que esta vez tendremos que usar hilos para poder atender asincronamente los mensajes del otro dispositivo, esto será para ambos casos la aplicación para PC y para Pocket PC por lo que usaremos también **System.Threading**

Aplicación de Chat Serial para Escritorio

Definiendo la interfaz de usuario

Para la interfaz utilizaremos dos **ComboBox** para elegir nuestros puertos de entrada y salida, justo abajo de éstos tendremos dos **TextBox** de solo lectura para mostrar información del puerto, un **TextBox** para poner nuestro nombre en el chat, un botón para conectarnos al dispositivo remoto y dos **TextBox** adicionales para enviar y recibir mensajes. La interfaz se debe ver como la **Figura 1.**

Figura 1.

Programación de la aplicación

Primero tenemos que indicar al compilador que usaremos los namespaces mencionados

```
using System.IO.Ports;
```

```
using System.Threading;
```

Y dentro de nuestro namespace declarar el **Delegate** que nos servira para actualizar el cuadro de mensajes en el hilo que atiende los mensajes remotos de entrada, solo recibe como parametro un **string** con el mensaje

```
    public delegate void ReadDelegate(string message);
```

Dentro de la clase de nuestro **Form** definimos nuestros objetos **SerialPort**, una bandera **bool** para control del hilo, un **StringComparer** para saber cuando teclea **salir** el usuario para terminar la aplicación y nuestro hilo para atender mensajes como se muestra en el **Listado 1**

Listado 1. Declaraciones

```
        // Variable auxiliar para indicar al hilo que atiende mensajes de
        // entrada que debe continuar recibiendo mensajes
        static bool _continue;
        // Objetos de tipo SerialPort para comunicación serial
        static SerialPort _serialPortIn;
        static SerialPort _serialPortOut;
        // StringComparer que usaremos para verificar que se envio
"salir"
        StringComparer stringComparer = StringComparer.OrdinalIgnoreCase;
        // Hilo para atender mensajes de entrada del dispositivo remoto
        Thread readThread;
```

Así pasamos al codigo mostrado en el **Listado 2** cuando se carga nuestro **Form** donde reservamos memoria para nuestros **SerialPort** y agregamos a nuestros **ComboBox** la información de los puertos disponibles en el sistema con la función estática **SerialPort.GetPortNames()**, adicionalmente se pueden obtener otras configuraciones con funciones similares que se muestran en el codigo comentado pero para el fin de este ejemplo solo necesitamos usar la configuración por default de los puertos.

Listado 2. Form_Load()

```
        private void Form1_Load(object sender, EventArgs e)
        {
            // Crear una nuevo objeto SerialPort con configuración por
default
            _serialPortIn = new SerialPort();
            _serialPortOut = new SerialPort();
            // Obtener puertos seriales I/O disponibles en el sistema
local
            foreach (string s in SerialPort.GetPortNames())
                cbPortIn.Items.Add(s);
            foreach (string s in SerialPort.GetPortNames())
                cbPortOut.Items.Add(s);
            /**** PARA CONFIGURACION ADICIONAL
            // Obtener paridades disponibles
            foreach (string s in Enum.GetNames(typeof(Parity)))
                cbParity.Items.Add(s);
            // Obtener bits de parada
            foreach( string s in Enum.GetNames(typeof(StopBits)))
                cbStopBits.Items.Add(s);
```

```
            // Obtener controles de flujo
            foreach (string s in Enum.GetNames(typeof(Handshake)))
                cbHandshake.Items.Add(s);
             */
        }
```

Cuando seleccionemos el boton para conectarnos tenemos que asignar la propiedad **PortName** a nuestros puertos creados de los **ComboBox**, actualizamos también la información del puerto seleccionado en nuestros **TextBox** de solo lectura, establecemos un tiempo de espera en milisegundos de espera para recibir y escribir datos en el puerto que será de 500 milisegundos. Con el método **Open()** de los puertos indicamos que los abrimos para lectura o escritura, después inicializamos el hilo para recibir mensajes en el puerto de entrada como se muestra en el **Listado 3.**

Listado 3. Click del boton para iniciar el servicio

```
        // Configurar puerto
        private void btConectar_Click(object sender, EventArgs e)
        {
            // Le decimos a nuestros objetos de puerto el nombre
            // del puerto a utilizar
            _serialPortIn.PortName = SetPortNameIn(
                  _serialPortIn.PortName);
            _serialPortOut.PortName = SetPortNameOut(
                  _serialPortOut.PortName);
            /* CONFIGURACION ADICIONAL
            _serialPort.BaudRate = SetPortBaudRate(_serialPort.BaudRate);
            _serialPort.Parity = SetPortParity(_serialPort.Parity);
            _serialPort.DataBits = SetPortDataBits(_serialPort.DataBits);
            _serialPort.StopBits = SetPortStopBits(_serialPort.StopBits);
            _serialPort.Handshake =
SetPortHandshake(_serialPort.Handshake);
            */
            // Mostramos informacíón del puerto
            lbPortIn.Text = _serialPortIn.PortName;
            tbPortIn.Text =
                "Bits por segundo:" + _serialPortIn.BaudRate.ToString() +
                    Environment.NewLine +
                "Paridad:" + _serialPortIn.Parity.ToString() +
                    Environment.NewLine +
                "Bits de datos:" + _serialPortIn.DataBits.ToString() +
                    Environment.NewLine +
                "Bits de parada:" + _serialPortIn.StopBits.ToString() +
                    Environment.NewLine +
                "Control de flujo:" + _serialPortIn.Handshake.ToString();
            lbPortOut.Text = _serialPortOut.PortName;
            tbPortOut.Text =
                "Bits por segundo:" + _serialPortOut.BaudRate.ToString()
+
                    Environment.NewLine +
                "Paridad:" + _serialPortOut.Parity.ToString() +
                    Environment.NewLine +
                "Bits de datos:" + _serialPortOut.DataBits.ToString() +
                    Environment.NewLine +
```

```
                "Bits de parada:" + _serialPortOut.StopBits.ToString() +
                    Environment.NewLine +
                "Control de flujo:" +
_serialPortOut.Handshake.ToString();

            // Establecer tiempos de espera de lectura y escritura
            _serialPortIn.ReadTimeout = 500;
            _serialPortOut.WriteTimeout = 500;
            // Abrimos los puertos
            _serialPortIn.Open();
            _serialPortOut.Open();
            // Creamos el hilo de lectura
            readThread = new Thread(Read);
            // Iniciamos hilo de lectura de datos
            readThread.Start();
            // Establecemos la bandera para continuar
            _continue = true;

        }
```

En el siguiente listado se muestra el código de las funciones auxliares **SetPortNameIn y SetPortNameOut** que lo unico que hacen es regresar el texto seleccionado de los **ComboBox** o en su defecto el puerto default

Listado 4. SetPortNameIn y SetPortNameOut

```
        // Funcion que regresa el puerto de entrada seleccionado
        public string SetPortNameIn(string defaultPortName)
        {
            string portName;
            if (cbPortIn.SelectedItem.ToString() == "")
            {
                portName = defaultPortName;
            }
            else
            {
                portName = cbPortIn.SelectedItem.ToString();
            }
            return portName;
        }
        // Funcion que regresa el puerto de salida seleccionado
        public string SetPortNameOut(string defaultPortName)
        {
            string portName;
            if (cbPortOut.SelectedItem.ToString() == "")
            {
                portName = defaultPortName;
            }
            else
            {
                portName = cbPortOut.SelectedItem.ToString();
            }
            return portName;
        }
```

Y el código para la función que usa el hilo para recibir mensajes donde utilizamos la función **ReadLine()** de nuestro puerto de entrada para leer una cadena de texto que este almacenada en su buffer y usamos **Invoke** con una instancia del **Delegate** que definimos al principio y de parámetro el mensaje que recibimos para que actualice correctamente el cuadro de mensajes

Listado 5. Funcion Read()

```
        // Funcion para procesar mensajes
        private void Read()
        {
            while (_continue)
            {
                try
                {
                    // ReadLine lee una linea de texto del puerto
                    string message = _serialPortIn.ReadLine();
                    // Usamos Invoke para llamar a la función que
actualiza
                    // el textbox de mensajes
                    Invoke(
                        new ReadDelegate(ChangeChatText),
                        new object[] { message });
                }
                catch (TimeoutException) { }
            }
        }
```

Finalmente definimos el evento **Click** de nuestro boton para enviar mensajes donde usamos el método **WriteLine()** de nuestro puerto de salida para enviar datos al puerto remoto de entrada

Listado 6. Evento Click para enviar mensajes

```
        // Enviar texto
        private void btEnviar_Click(object sender, EventArgs e)
        {
            // Si tecleamos salir terminamos el hilo y quitamos la
aplicación
            if (stringComparer.Equals("salir", tbEnviar.Text))
            {
                _continue = false;
                this.Dispose();
            }
            // Si no escribimos al puerto
            else
            {
                // WriteLine envia una cadena a la salida del puerto
                _serialPortOut.WriteLine(String.Format(
                    "<{0}>: {1}",
                    tbName.Text,
                    tbEnviar.Text));
                // Actualizamos el textbox con lo que escribimos
                tbChat.Text += String.Format(
                    System.Environment.NewLine + "<{0}>: {1}",
```

```
                tbName.Text,
                tbEnviar.Text);
        }
    }
```

Eso es todo en cuanto a la aplicación de escritorio para enviar y recibir mensajes, ahora veremos la aplicación para la Pocket PC que es prácticamente idéntica, la única diferencia es que no mostramos la información del puerto. La interfaz se muestra en la **Figura 2**

Figura 2

Aplicación de Chat Serial para Dispositivos Móviles

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.IO.Ports;
using System.Threading;

namespace MobileSerialChat
{
    public delegate void ReadDelegate(string message);
    public partial class Form1 : Form
    {
        static bool _continue;
        static SerialPort _serialPortIn;
        static SerialPort _serialPortOut;

        StringComparer stringComparer = StringComparer.OrdinalIgnoreCase;
        Thread readThread;

        public Form1()
        {
```

```
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Crear una nuevo objeto SerialPort con configuración
default
            _serialPortIn = new SerialPort();
            _serialPortOut = new SerialPort();
            // Obtener puertos disponibles
            foreach (string s in SerialPort.GetPortNames())
                cbPortIn.Items.Add(s);
            foreach (string s in SerialPort.GetPortNames())
                cbPortOut.Items.Add(s);
            /*
            // Obtener paridades disponibles
            foreach (string s in Enum.GetNames(typeof(Parity)))
                cbParity.Items.Add(s);
            // Obtener bits de parada
            foreach( string s in Enum.GetNames(typeof(StopBits)))
                cbStopBits.Items.Add(s);
            // Obtener controles de flujo
            foreach (string s in Enum.GetNames(typeof(Handshake)))
                cbHandshake.Items.Add(s);
             */

        }
        private void btConectar_Click(object sender, EventArgs e)
        {
            // Le damos una configuración al puerto
            _serialPortIn.PortName =
SetPortNameIn(_serialPortIn.PortName);
            _serialPortOut.PortName =
SetPortNameOut(_serialPortOut.PortName);
            /*
            _serialPort.BaudRate = SetPortBaudRate(_serialPort.BaudRate);
            _serialPort.Parity = SetPortParity(_serialPort.Parity);
            _serialPort.DataBits = SetPortDataBits(_serialPort.DataBits);
            _serialPort.StopBits = SetPortStopBits(_serialPort.StopBits);
            _serialPort.Handshake =
SetPortHandshake(_serialPort.Handshake);
            */
            lbPortIn.Text = _serialPortIn.PortName;
            /*
            tbPortIn.Text =
                "Bits por segundo:" + _serialPortIn.BaudRate.ToString() +
                "\nParidad:" + _serialPortIn.Parity.ToString() +
                "\nBits de datos:" + _serialPortIn.DataBits.ToString() +
                "\nBits de parada:" + _serialPortIn.StopBits.ToString() +
                "\nControl de flujo:" +
_serialPortIn.Handshake.ToString();
             */
            lbPortOut.Text = _serialPortOut.PortName;
            /*
```

```
            tbPortOut.Text =
                "Bits por segundo:" + _serialPortOut.BaudRate.ToString()
+
                "\nParidad:" + _serialPortOut.Parity.ToString() +
                "\nBits de datos:" + _serialPortOut.DataBits.ToString() +
                "\nBits de parada:" + _serialPortOut.StopBits.ToString()
+
                "\nControl de flujo:" +
_serialPortOut.Handshake.ToString();
             */

            // Establecer tiempos de espera
            _serialPortIn.ReadTimeout = 500;
            _serialPortOut.WriteTimeout = 500;
            // Abrimos el puerto
            _serialPortIn.Open();
            _serialPortOut.Open();
            readThread = new Thread(Read);
            // Iniciamos hilo de lectura de datos
            readThread.Start();
            _continue = true;
        }

        private void btEnviar_Click(object sender, EventArgs e)
        {
            if (stringComparer.Equals("salir", tbEnviar.Text))
            {
                _continue = false;
                this.Dispose();
            }
            else
            {
                _serialPortOut.WriteLine(String.Format("<{0}>: {1}",
tbName.Text, tbEnviar.Text));
                tbChat.Text += String.Format("\n<{0}>: {1}", tbName.Text,
tbEnviar.Text);
            }
        }

        private void Read()
        {
            while (_continue)
            {
                try
                {
                    string message = "\n" + _serialPortIn.ReadLine();
                    Invoke(new ReadDelegate(ChangeChatText), new object[]
{ message });
                }
                catch (TimeoutException) { }
            }
        }
        private void ChangeChatText(string message)
        {
```

```
        tbChat.Text += message;
    }

    public string SetPortNameIn(string defaultPortName)
    {
        string portName;
        if (cbPortIn.SelectedItem.ToString() == "")
        {
            portName = defaultPortName;
        }
        else
        {
            portName = cbPortIn.SelectedItem.ToString();
        }
        return portName;
    }
    public string SetPortNameOut(string defaultPortName)
    {
        string portName;
        if (cbPortOut.SelectedItem.ToString() == "")
        {
            portName = defaultPortName;
        }
        else
        {
            portName = cbPortOut.SelectedItem.ToString();
        }
        return portName;
    }
```

Estableciendo conexiones con los servicios Bluetooth

Hay que tener en cuenta que tanto en nuestra Pocket PC como en la PC deben estar activados los servicios de Puerto Serie Virtual y haber establecido una conexión entre los dos gestores de servicios de Bluetooth para que la aplicación pueda saber a donde va a enviar los mensajes y desde donde los recibirá también.

Esto se puede lograr a través de nuestro software en la PC como puede ser el BlueSoleil que muy probablemente incluya nuestro adaptador y en el caso de la Pocket PC el fabricante debe haber incluido un software para la gestión de conexiones Bluetooth así que habrá que ver la documentación de ambos dispositivos Bluetooth para saber como conectar nuestro servicio de Puerto Serie del dispositivo móvil a la PC y viceversa.

Una vez establecida la conexión con los servicios de Puerto Serie podemos correr las dos aplicaciones y probar nuestro chat.

Probando nuestra aplicación con dos Pocket PC

Para hacer esto lo unico que tenemos que hacer es copiar el mismo ejecutable de la aplicación móvil del chat a las dos Pocket PC que deseemos y establecer las conexiones de Bluetooth, el modo de operación no cambia.

Hay que tener en cuenta que si queremos crear un chat de mas de dos terminales tendremos que implementar un mecanismo en la aplicación para identificar de donde provienen los mensajes recibidos en los dispositivos ya que no podemos agregar puertos seriales dinamicamente.

Figura 3. Aplicación de Bluetooth corriendo