

# Tablet SDK Part I: Ink Collection

---

Dana Tenneson  
Eurographics Sketch-Based  
Interfaces and Modeling 2007

This presentation is part of a Tablet PC development tutorial taking place at Eurographics Sketch-Based Interfaces and Modeling 2007.

Prerequisites:

Setting up a Tablet PC development environment.

## Overview

---

- Demo program highlights useful ink collection techniques
- We will be recreating demo program
  - Step 1: InkCollector and Demo Boiler Plate
  - Step 2: Ink Drawing Attributes
  - Step 3: Ink Events
  - Step 4: Ink Data (Packet Properties)
  - Step 5: Ink Resources

This section will focus on creating a demo program which explores four aspects of ink collection which I have found useful in my research: Drawing Attributes, Ink Events, Ink Packet Properties, and Ink in Resources. The demo program itself is not particularly useful, but creating it will allow the user to explore each of these topics.

## Demo Boiler Plate

□ All demo code available on server



- Form
  - Text: Ink Data
- TextBox
  - Dock: Right
  - Multiline: True
- ToolStripContainer
  - Dock Fill in Form
- ToolStripContainer.ContentPanel
  - BackColor: MistyRose
- ToolStrip

Tablet SDK Part 1: Ink Collection

3

The boiler plate for our demo is a simple Windows Forms application with an ink collector, a text box, and a tool strip. In general, instructions will be given on each slide as to how to create the next piece of functionality. In some cases, we will be incorporating pre-made code to reduce the time it takes to complete the demo.

To begin, start a new C# Windows Forms application. Expand out the form to a roughly rectangular size and drag a text box over from the toolbox. If you place the text box roughly on the right of the form and change two of its properties (Dock: Right, Multiline: True), your text box will fill the right hand side of the form. We will be using this text box for text output about tablet capabilities and ink data.

Dragging a ToolStripContainer from the toolbox and selecting “Dock Fill in Form” will give us the main inking area and a place to add in a tool strip at the bottom.

You can also set the form’s Text property to set the title at the top and the tool strip container’s content panel back color property to customize the look of your program. I’m fond of MistyRose as a good inking background color, but feel free to pick your own color.

# Ink Collector

---

- Adding ink is fairly painless
  - Just attach an InkCollector to your control
- Other ink options
  - InkOverlay
  - InkEdit
  - InkPicture

Adding ink capabilities to a windows form control is fairly painless. You need only create the InkCollector control and attach it to the visual element you wish to ink upon.

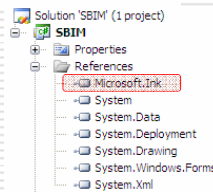
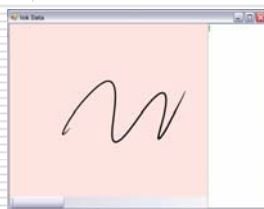
The tablet SDK provides a few different choices for ink controls. The InkCollector is the most generic and is what we will be using in this demo program. The InkOverlay is intended for situations where the ink overlays the control for annotation purposes, but is fairly generic as well. The InkEdit and InkPicture controls are much more special purpose – intended for entering written text and inking on pictures.

# Ink Collection Setup

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

namespace SBIM
{
    public partial class Form2 : Form
    {
        InkCollector collector;

        public Form2()
        {
            InitializeComponent();
            // Create an ink collector
            collector = new InkCollector(toolStripContentPanel);
            // Attach the ink collector to the content panel
            collector.Attach(toolStripContentPanel);
            // Enable the ink collector
            collector.Enabled = true;
        }
    }
}
```



- Microsoft.Ink Reference
- using Microsoft.Ink
- InkCollector
  - AttachedControl: Our ToolStripContainer Content Panel
  - Enabled: True
    - Resets collector
    - Comes after various collector setup commands

Tablet SDK Part 1: Ink Collection

To add an ink collector, add the Microsoft.Ink reference from the Tablet SDK to your project references and add “using Microsoft.Ink;” to your class. Then you can create an ink collector and attach it to our tool strip container’s content panel and enable the collector.

At this point you can ink on the pink content panel and our demo boiler plate is complete.

I should also note here that the InkCollector object is a special case in that it needs to be disposed of manually rather than left to the garbage collector. In c# this can be done in your designer’s .cs file near the bottom of the overridden Dispose method. It’s good coding practice to do this, but it won’t be a problem if your program only creates one InkCollector in its lifetime.

## Ink Drawing Attributes

- Color
- Width & Height
- Pen Tip
- Anti Aliasing
- Raster Operation
- Curve Fitting
- Pressure Use
- Transparency



Anti Aliasing Off



Rectangle Pen Tip



Ignore Pressure On

The appearance of ink drawn in an ink collector is determined by the `DefaultDrawingAttributes` of the ink collector at the time the ink is drawn. Common attributes to change are the color and width of the pen and we will be covering those two in this demo. There are several other properties which can also be explored such as using a rectangular pen tip, ignoring pen pressure, and turning off anti aliasing. For fun, I also recommend experimenting with the raster operation to create neat art effects.

# Adding a Width Text Field

- ❑ Add text field to tool strip
- ❑ Set text
- ❑ Add TextChanged Event
- ❑ In Event
  - Parse text as int
  - Set InkCollector's DefaultDrawingAttributes.Width Property
  - Catch parsing exceptions for good measure



Tablet SDK Part 1: Ink Collection

```
InitializeComponent();

// Create the ink collector
c = new InkCollector();
// Attach the ink collector to the right panel
c.AttachedControl = this.ContainerControl.ContentPanel;
// Turn on the ink collector. This must be done after setting packet data
c.Enabled = true;

// Set the text of the text box
this.toolStripTextBox.Text = "" + c.DefaultDrawingAttributes.Width;
// Add an event to the text box
this.toolStripTextBox.TextChanged += new EventHandler(toolStripTextBox_TextChanged);

// Respond
// Place upon the "text changes" in our width text box
private void toolStripTextBox_TextChanged(object sender, EventArgs e)
{
    // Parse the text into an int and set the width
    try
    {
        int width = int.Parse(toolStripTextBox.Text);
        c.DefaultDrawingAttributes.Width = width;
    }
    catch (FormatException)
    {
        // Set the text of the text box
        this.toolStripTextBox.Text = "" + c.DefaultDrawingAttributes.Width;
    }
}
```

The first attribute we will be modifying is the width of the pen tip. To do this, we will be adding a text box which the user can use to set the pen width.

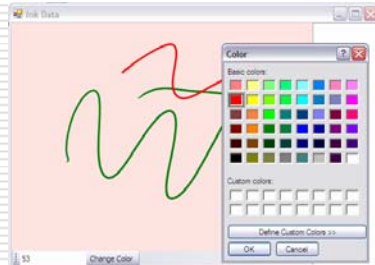
Add a text box to your tool strip via the tool strip's pull down menu. In your class constructor, set the text to be the current width of the pen (c.DefaultDrawingAttributes.Width). We also want the pen width to update when the user changes the value of the text box. So add a TextChanged event to our text box.

While there are more user-friendly ways to error check the data entered, for this demo, we're going to simply try to parse the text into an int and if this throws an exception, reset the text box text. We can interpret the text with the int.Parse method and use that output to set the Width attribute.

At this point the user can draw different sizes of text into the ink collector.

## Adding a Color Dialog

- ❑ Add button to tool strip
  - DisplayStyle: Text
  - Text: Change Color
- ❑ Click Event:
  - Copy from provided source
  - ❑ Create Color Dialog
  - ❑ Set Dialog color to DefaultDrawingAttributes.Color
  - ❑ ShowDialog()
  - ❑ On OK, set DefaultDrawingAttributes.Color



Allowing the user to change the color of the ink is very similar to allowing the user to change the width of the pen. In this case, we are adding a button to the tool strip which will launch a color dialog for selecting the color. By default, buttons in tool strips show up as small icons, so you'll need to set the button's `DisplayStyle` and `Text` properties to yield an intelligible button.

Double click the button in the designer to create a `Click` event for the button and to get to the code. For the sake of time, I'm going to recommend copying the source code from the `ColorChange.rtf` file. This code creates a color dialog with the current ink color set and presents it to the user. If the user clicks ok, the selected color from the dialog is then applied to the ink collector.

Changing other drawing attributes is just as simple as these two.

# NewPackets Event

```
public Form2()
{
    InitializeComponent();

    // Create an ink collector
    c = new InkCollector();
    // Attach it to the content panel for the tool strip area
    c.AttachedControl = this.toolStripContainer1.ContentPanel;

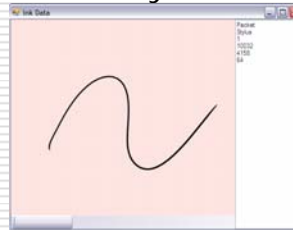
    // Ink for mouse
    c.NewPackets += new InkCollectorNewPacketsEventArgs(NewPackets);

    // Turn on the collector -- do this after setting packet data.
    c.Enabled = true;
}

void NewPackets(object sender, InkCollectorNewPacketsEventArgs e)
{
    // Fill our text box with a report of the packet data.
    textBox1.Text += "Packet: " + e.PacketCount + "\n";
    textBox1.Text += "Cursor: " + e.Cursor.X + " " + e.Cursor.Y + "\n";
    textBox1.Text += "Device: " + e.Device + "\n";
    textBox1.Text += "PacketData: " + e.PacketData + "\n";
    textBox1.Text += "-----\n";
}

```

- Fires when the stylus is pressed to the tablet
- Similar to MouseDrag, but higher resolution
- Works with mouse as well as stylus



Tablet SDK Part 1: Ink Collection

9

We now move on to ink-based events. The NewPackets event is very similar to a MouseDrag event in that it occurs when the pen is in contact with the stylus and ink is being collected. However, the NewPackets event occurs at a much higher resolution than MouseDrag. This resolution is higher in both event frequency and coordinate accuracy. Ink space allows for much finer detail than pixel space. Because of this increase in resolution, it is very important that any code in the NewPackets event returns quickly. We will be breaking this rule in this demo by having each event print its data out to the text field. However, in your programs, I very much recommend not putting any slow operations like text output in this event. If necessary, spawn another thread to handle the slow code.

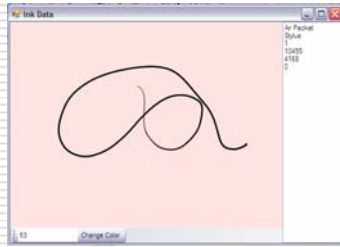
Our Packets event prints out the entire contents of the InkCollectorNewPacketsEventArgs object. We print the cursor making the ink (such as the mouse, stylus or eraser), the packet count, and each entry in the packet data. Later we will be covering the nature of each of these packet data entries.

# NewInAirPackets Event

```
// Ask for events
a.NewPackets == new TabletInkCollectorPacketsHandler(a.NewPackets);
c.NewInAirPackets == new TabletInkCollectorPacketsHandler(a.NewInAirPackets);

// Turn on the collector == do this after setting packet data.
c.Enabled = true;
}

void NewInAirPackets(object sender, TabletInkCollectorPacketsEventArgs e)
{
    // Print out each line with a report of the packet data
    packetData += "Tip Air Packet: " + e.Packets[0].TipAirPacketData + "\n";
    packetData += "e.Cursor = " + e.Cursor + " (X: " + e.Cursor.X + ", Y: " + e.Cursor.Y + ");\n";
    packetData += "e.PacketCount = " + e.PacketCount + "\n";
    packetData += "e.PacketData\n";
    packetData += "e.PacketData\n";
    packetData += "e.PacketData\n";
}
```



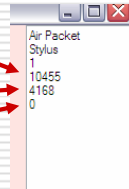
- ❑ Fires when stylus hovers over tablet
  - Equivalent to MouseMove
- ❑ Like NewPackets, must be fast
  - This text output doesn't cut it.

Tablet SDK Part 1: Ink Collection

Similarly, the NewInAirPackets event covers event made when the stylus hovers over the ink collector without pressing down. It is therefore very similar to a MouseMove event. The only difference in the event code here is that we indicate that the packet is an “in air packet” Now we can collect ink data whenever the stylus is moved over the ink collector whether it is currently inking or not.

## Default Packet Data

- [0] X (ink space)
- [1] Y (ink space)
- [2] Pressure
- That's it
  - Very efficient for high-speed inking



You may have guessed by now what the different pieces of data we are receiving from our ink events are. The first one is the X coordinate in ink space. The second is the Y coordinate in ink space. The third is the pressure (0-255) that is being applied to the stylus. When we are getting “in air” packets, the pressure is set to 0.

This is all of the data returned by default and the small size of the data packet is very efficient for allowing a high throughput of ink packets. As we expand the data packets, the speed will decrease.

## PacketProperty Class

---

- Defines ink data types that can be returned

AltitudeOrientation	TimerTick
AzimuthOrientation	TwistOrientation
ButtonPressure	X
NormalPressure	XTiltOrientation
PacketStatus	Y
PitchRotation	YawRotation
RollRotation	YTiltOrientation
SerialNumber	Z
TangentPressure	

Requesting additional packet data is fairly straightforward ... once you know about the packet property class. PacketProperty enumerates all the different kinds of ink data we can possibly receive from the stylus. PacketProperty treats the stylus as a fully 3D input device with placeholders for data such as tilt orientation, yaw, pitch, and roll. In reality, not every tablet supports all of these properties. In fact, the number supported is usually very small.

## Supported Packet Properties

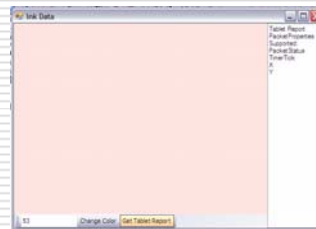
- ❑ MSDN has demo code on detecting supported properties
- ❑ Note that pressure isn't on the list
  - This is for properties supported by **all** tablets – including your mouse.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace InkData
{
    public partial class InkData : Window
    {
        public InkData()
        {
            InitializeComponent();
        }

        private void GetTabletReport_Click(object sender, RoutedEventArgs e)
        {
            TabletReport.Report();
        }
    }
}
```

```
Tablet Report
PacketProperties
Supported:
PacketStatus
TimerTick
X
Y
```



Tablet SDK Part 1: Ink Collection

13

We can ask the tablet whether or not it supports each packet property with a little piece of support code from the MSDN. Create a button to generate a “Tablet Report” in the tool strip and in that button’s click event, include the event code found in TabletReport.rtf You’ll also find the TabletReport method in the same file.

When we click the button in the application, we find that the supported properties are PacketStatus, TimerTick, X, and Y. This is a much reduced list from the entirety of the PacketProperty class. You may also note that Pressure isn’t on the list, which is strange since we were receiving that data before without problem.

The catch here is that this particular MSDN example checks for properties supported by **all** tablets on the system – including the mouse. The mouse, of course, has no concept of pressure.

## Actually Supported Packet Properties

- Asking for supported properties of each tablet yields better results
  - You **can** get NormalPressure
  - However, have yet to see X and Y TiltOrientation return non 0 result

```
Tablet: \\.\DISPLAY1
PacketProperties
Supported:
PacketStatus
TimerTick
X
Y

Tablet: ISD V4
PacketProperties
Supported:
NormalPressure
PacketStatus
TimerTick
X
XTiltOrientation
Y
YTiltOrientation
```

By changing the TabletReport definition line to the one commented out in the provided code and switching over to the commented code in our button click event, we can instead generate an individual report on the packet properties supported by each tablet on the system. Here we find that the internal tablet can provide pressure data as well as tilt orientation data. This list of supported data is common for the toshiba and hp tablets I've worked with, but it is my understanding that there is some difference between the properties supported by different tablet manufacturers.

Yet again, there is a problem with the supported packet properties list. Unfortunately, it has been my experience that the tilt orientation properties always report a tilt of 0 on the tablets I've worked with. So, yet again there is a problem with the getting an accurate idea of which properties are supported on a given tablet.

## Requesting More Data

- ❑ Accomplished through DesiredPacketDescription
- ❑ You always get X and Y (even if not requested)
- ❑ Packet Status always comes third
- ❑ Set the DesiredPacketDescription before enabling InkCollector

```
// Get More Ink Data -- You always get X & Y and packet status comes third if specified
// Note how slow this gets and that you don't actually get the data.
Guid[] packetDataFormat = new Guid[7];
packetDataFormat[0] = PacketProperty.X;
packetDataFormat[1] = PacketProperty.Y;
packetDataFormat[2] = PacketProperty.XTiltOrientation;
packetDataFormat[3] = PacketProperty.YTiltOrientation;
packetDataFormat[4] = PacketProperty.PacketStatus;
packetDataFormat[5] = PacketProperty.TimerTick;
packetDataFormat[6] = PacketProperty.NormalPressure;
c.DesiredPacketDescription = packetDataFormat;
```

Packet Stylus
1
6374
3304
1
0
0
2703
166

Tablet SDK Part 1: Ink Collection

15

Now that we know which types of data can be requested, the ink collector's DesiredPacketDescription can be used to specify the data our application wants. In our class constructor, create a Guid array to contain the Guid's of the PacketProperties wanted and assign this array to the ink collector. It is important to note that setting the DesiredPacketDescription cannot be done when the ink collector is enabled, so place this code before the line which enables the ink collector.

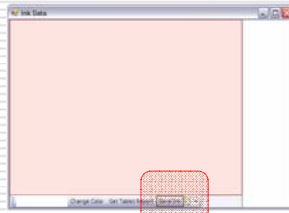
There are several more exceptions to the rule when it comes to setting the DesiredPacketDescription. First of all, you always get X and Y data at the front of your packets whether or not you request the data to be elsewhere in the packet or not in the packet at all. Similarly, if you ask for the PacketStatus, it always fills index [2] of the packet array regardless of where you put it in your packet description. The rest of the requested data appears in your packets in the order requested.

You may be noticing at this point how much slower the InkPacket events are coming.



## Saving Ink as a Resource

- ❑ Add “Save Ink” button to tool strip
- ❑ Use SaveFileDialog to select output file
- ❑ ResXResourceWriter writes resource file
- ❑ Ink.Save() creates byte array



Tablet SDK Part 1: Ink Collection

17

First, we will be serializing ink out to a resource file. Add a “Save Ink” button to our tool strip and grab the code in SaveInk.rtf for the contents of the event. This code first creates a standard SaveFileDialog and sets its filter to .resx files. .resx is the file type for XML-based resources. The user is shown the dialog and if a save file is specified, the ink byte stream is written out with a ResXResourceWriter. Resource files are dictionaries of potentially many key value pairs. In this case, we only need one value which we are giving the key “Ink”. However, many ink and non-ink resources can be put in a single file. The .resx file can then be opened with any text editor such as WordPad or Visual Studio.

## Loading Ink Resources

---

- ❑ Add “Load Ink” button to tool strip
- ❑ Use OpenFileDialog to select input file
- ❑ ResXResourceReader reads resource file
- ❑ Ink.Load() loads byte array
- ❑ Complications
  - Ink must be brand new
  - Can’t replace ink while ink collector is active

Tablet SDK Part 1: Ink Collection

18

Loading ink resources back from file is mostly the same process as saving but in reverse. The `OpenInk.rtf` file contains the event code for a “Load Ink” button on your tool strip. The `OpenFileDialog` is set up with the same filter as the `SaveFileDialog` and is shown to the user. If the user specifies a file, it is opened with a `ResXResourceReader` and searched for our “Ink” key. The value is then loaded into the ink object.

Loading is complicated by the fact that `Ink.Load()` can only be called on a brand new (or at least uninked) `Ink` object which is not active at the time of loading. In the example program, we are creating a new `Ink` object and using it to replace the ink collectors. If you wish to add saved ink into a `InkCollector` already containing ink, you need to create a new `Ink` object for the deserialization and then add the loaded ink `Strokes` to the existing `Ink` control.

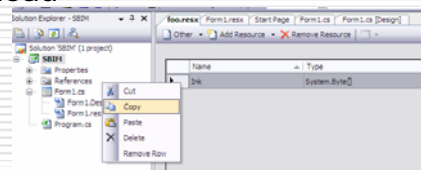
# Embedding Ink Resources

- ❑ Open .resx file with Visual Studio
- ❑ Copy “Ink” dictionary key
- ❑ Open form resx file
- ❑ Paste the key
- ❑ Use a ResourceManager to load resource
  - typeof(<your form>)
  - Load bytes as usual

```
using System.Windows.Forms;  
using System.Resources;  
using System.Collections;  
using Microsoft.Ink;
```

```
namespace SBIM
```

```
{
```



```
// Set the text of the text box  
this.toolStripTextBox1.Text = "" + o.DefaultDrawingAttributes.Width;  
// Add an event to the text box  
this.toolStripTextBox1.TextChanged += new EventHandler(toolStripTextBox1_TextChanged);
```

```
// For load some ink  
ResourceManager rmc = new ResourceManager(typeof(Form1));  
byte[] input = (byte[]) rmc.GetObject("ink");  
o.InkCollection.Add(input);
```

Tablet SDK Part 1: Ink Collection

19

Finally, we can embed these resources directly into our application deployment. There are a few different ways of doing this, but one straightforward way is to add the ink resource to the resx file created to accompany the form or control using the ink collector. To add a saved ink resource to our demo's form, open the .resx file in visual studio and copy the “Ink” dictionary key. Then open the .resx file for your form (found indented and beneath your form's .cs file in the Solution Explorer). Paste the copied key into the form's .resx file and your ink is now embedded in the project. To load the ink internally, you can use a ResourceManager object to retrieve your byte array. In the demo, we can have our program start up with ink already on display by adding this code to the class constructor. If you have not done so already, you'll need to add using System.Resources and System.Collections to your class.

We're done. You now have a demo program exploring arcane and not so arcane aspects of TabletPC ink collection which will help you on your way to our next topic of Ink Analysis.