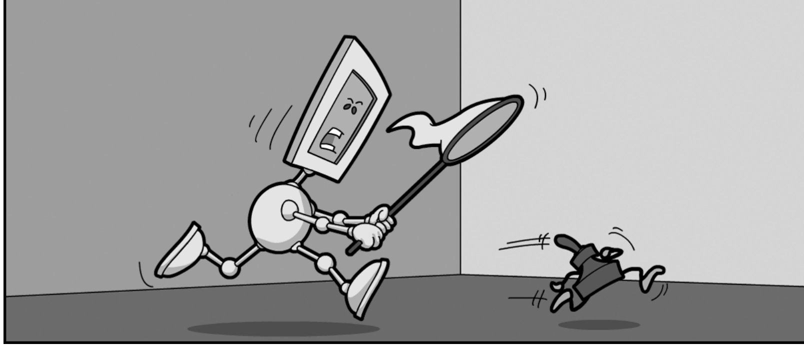
4



Tablet PC Platform SDK: Tablet Input

The previous chapter illustrated how the Tablet PC platform is divided into three logical pieces: tablet input, ink data management, and ink recognition. This chapter will take a close look at the tablet input piece of the platform from an architectural as well as a programmatic standpoint.

We'll begin by learning about how physical pen strokes become useful input to both tablet-aware and tablet-unaware applications. Then we'll see how to use the Tablet PC Platform SDK to enumerate, introspect, and—most important—capture input from tablet devices. At the end of the chapter, some best practice methods to help yield the best Tablet PC user experience will be discussed.

Sample Applications

Various sample applications with source code are provided to demonstrate the practical application of concepts as they are presented. Each application was designed to focus only on what's necessary to illustrate relevant material—other aspects of the applications were intentionally kept minimal for clarity's sake. Considerable effort has been put into testing the applications and making sure that they follow correct usage of the Tablet PC Platform SDK, though in the spirit of software design it's always possible for a bug to creep in. You certainly don't have to be prepared to call the fire department if you copy and paste code from this book into your own application, but please don't call "Mr. T's House of SmackDown!" if an unexpected "feature" crops up during a demo of your app to your boss.

Capturing Input from the Pen

Before we cover how to write applications that receive pen input from the tablet digitizer, let's take a look at the software that enables us to harness the power of the pen.

Note This section can be considered optional reading because one doesn't really need to know the internals of tablet input to get inking in an application. You can therefore skip ahead to the section titled "Platform SDK Support for Tablet Input" if you're not interested in how things work, though we do hope you stick around here because we think it's interesting stuff.

The Tablet PC's tablet input subsystem refers to the logical pieces of software that transfer and transform the data generated from a tablet digitizer into the input an application can make use of. Exactly what kind of input is it that an application can make use of? Chapter 2 gives us a good idea of the experience we want to provide for a Tablet PC user, so let's see whether we can come up with some requirements here that might help us better understand the functions of the input subsystem.

Requirement #1—Mouse Emulation

As you know by now, all tablet-unaware Microsoft Windows XP–compatible applications are fully supported under Windows XP Tablet PC Edition—the pen behaves like a mouse in these cases. It also happens that mouse input is valuable to tablet-aware applications—standard Windows behaviors, controls, and the like are already driven by the mouse, so it's convenient to be able to leverage this functionality for the pen. And of course, it's likely that a physical mouse device will be used to drive the application because many Tablet PCs will have integrated mouse hardware such as a touch pad or mini-trackball.

Note We predict that someday all mouse input handling in Windows will also become tablet input–enabled. This will pave the way for pen-specific behaviors in all areas of the Windows user interface.

Because most applications rely heavily on mouse input, our first requirement of the input subsystem is that it be able to map pen input to mouse input—a process we'll refer to as *mouse emulation*. Both left-mouse and right-mouse buttons should be supported for compatibility.

Requirement #2—Digital Ink

Tablet-enabled applications often accept user input in the form of digital ink, which can be added to a document or recognized into a command. Capturing ink from the pen is one of the most important aspects of realizing an electronic paper paradigm. As such, digital ink must mirror physical ink as closely as possible to provide the most natural and unobtrusive end-user experience. From a user's perspective, ink should "just work" on a Tablet PC and be practically indistinguishable in behavior from physical ink. We can therefore impose the following sub-requirements on a system that's used to capture digital ink:

- **Performance** Ink should appear to be flowing directly out of the tip of the pen in real time and never lagging behind. This requires that the time between sampling the position of the pen and rendering ink on screen be imperceptible to an average user.
- **Accuracy** Ink should follow the exact path of the pen as the pen moves. This requires that the frequency and resolution at which data is captured should result in the digital ink appearing to be smooth in shape to an average user. And as you'll see later, using data captured at higher frequency and resolution also helps improve handwriting and gesture recognition results.
- **Robust data capture** Ink should reflect as much of the physical handling that the pen is subjected to as possible. This requires that not only the pen position be sampled, but that support for sampling of pen tip pressure, the angle between the pen and tablet surface, the rotation of the pen body, and the like should also be provided.

In the initial version of the Tablet PC Platform, normal pressure is the only property besides *X* and *Y* position that the rendering of ink can reflect. However, future versions of the Tablet PC Platform will likely take more properties into account.

It is entirely possible for you to custom draw ink if you wish to have more properties taken into account, as we'll learn in Chapter 5. In the majority of cases, we think *x*, *y*, and pressure yield incredible results.

Requirement #3—Pen-Based Actions

Pen-specific actions like press-and-hold, using the top-of-pen eraser, and pressing pen buttons can become powerful means to streamline the user model of a tablet-aware application. In addition, employing pen input for means other than digital ink is useful because pen-specific properties can enhance existing user interface behaviors. For example, pen rotation could be used to rotate a selection while it's being dragged, or pen pressure could be used to determine how large an erasing area should be used for an eraser tool. Distinguishing between the user tapping the pen and the user dragging it is therefore important functionality.

Note Pen-specific actions should not be confused with ink-based gestures such as scratchout, up arrow, and "curlique". Pen actions—which are formally known as *system gestures*—don't use ink and don't define any application behavior. They are on a par with mouse actions such as click and drag.

Although the tablet hardware's device driver can easily report top-of-pen use (called *pen inversion*) and pen button presses, detecting press-and-hold and associated actions such as tap and drag is a nontrivial algorithm that arguably falls outside the driver's scope; this algorithm is best centralized so that any tablet device can use it.

Summing Up the Requirements

We have now specified that the input system must be able to transform raw pen movement into mouse input (supporting left-mouse and right-mouse button emulation), provide realistic digital ink, and detect higher-level pen-based actions (press-and-hold, pen inversion, pen button presses, and tap versus drag). The Tablet PC Platform's tablet input subsystem (which we'll start referring to as "the TIS" for brevity) provides all this and more.

Anatomy of the Tablet PC's Tablet Input Subsystem

An architectural view of the TIS is outlined in Figures 4-1 and 4-2. There are two main configurations under which the subsystem runs: the first is with Windows XP Tablet PC Edition, and the second is with a Windows 2000–based or Windows XP–based operating system with the Tablet PC runtime libraries installed.

We'll be focusing primarily on the subsystem as it runs on Windows XP Tablet PC Edition because it's a superset of the Tablet PC runtime. But don't worry; any differences between the two configurations will be pointed out as they arise.

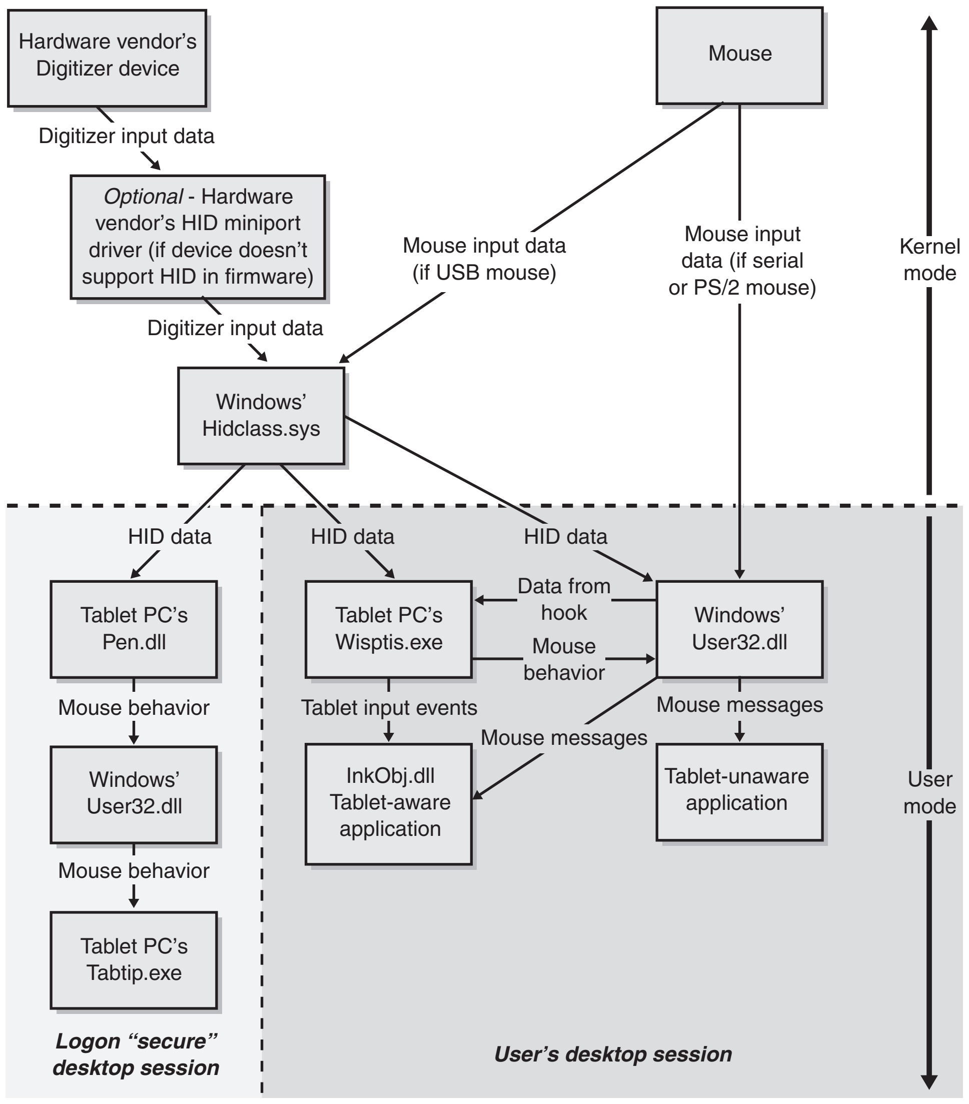


Figure 4-1 The Tablet PC tablet input subsystem architecture, shown running on Windows XP Tablet PC Edition

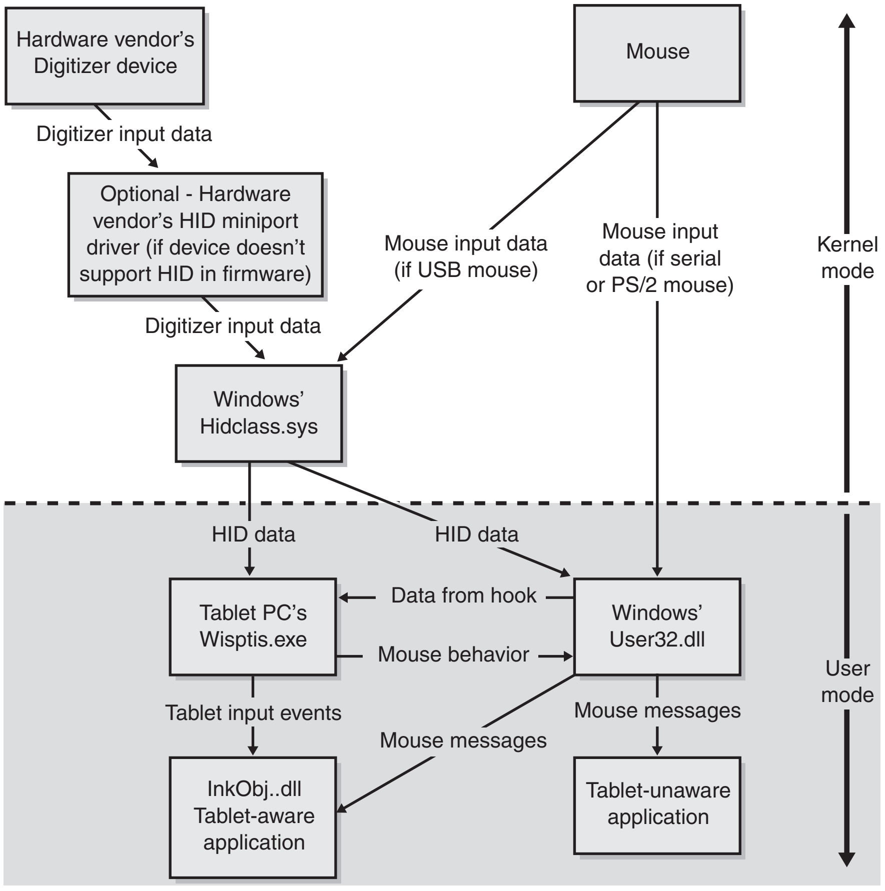

Hardware vendor's Digitizer device
Digitizer input data
Optional - Hardware vendor's HID miniport driver (if device doesn't support HID in firmware)
Digitizer input data
Mouse
Mouse input data (if USB mouse)
Mouse input data (if serial or PS/2 mouse)
Kernel mode
Windows' Hidclass.sys
HID data
HID data
Tablet PC's Wisptis.exe
Data from hook
Mouse behavior
Windows' User32.dll
User mode
Tablet input events
Mouse messages
Mouse messages
InkObj..dll Tablet-aware application
Tablet-unaware application

Figure 4-2 The Tablet PC tablet input subsystem architecture, shown running as the Tablet PC runtime libraries on a Windows-based operating system

Tablet Hardware

In Chapter 1, the essence of a Tablet PC was identified as a portable PC combined with a digitizer integrated with the screen, driven by Windows XP Tablet PC Edition. A digitizer (sometimes ambiguously referred to as just a *tablet*) in the Tablet PC Platform's view is a device that provides user input to a computer via a pointer on a flat rectangular surface. The device is able to sample the *X* and *Y* position of the pointer at regular time intervals and determine whether the pointer is *active*—typically, this means whether the pointer is touching the tablet's surface. In most cases, the pointer is a pen stylus, although the Tablet PC Platform also recognizes a mouse as a tablet device. Indeed, a mouse is an example of a tablet device—it reports *X* and *Y* position as well as whether the pointer is active.

Note Placing the pointer tip on the digitizer surface is called making the pointer *active*. A synonym for active is *down*.

Chock-full of HID-y Goodness

Before the days of Tablet PC, a low-level standard already existed that directly facilitated getting input from a tablet device. Known as HID (short for Human Interface Device), the specification was initially developed to standardize communication to USB hardware such as keyboards, mice, joysticks, tablets, and just about any other device we can use to generate input for using a computer. The tablet input subsystem exclusively leverages HID devices for tablet input, and as such, the digitizers on Tablet PCs (even ones integrated with the video display) are either USB-based and HID-compliant via firmware or come supplied with a miniport driver emulating a HID interface.

Note At the time of this writing, the full HID specification can be found at the USB Implementers Forum Web site, at *http://www.usb.org/developers/hidpage.html*, under the heading "Device Class Definition for Human Interface Devices (HID)."

WinTab

Some alert readers may be aware of another standard tablet input API called WinTab (*http://www.pointing.com/WINTAB.HTM*). It was discovered during the development of the TIS that using WinTab would prove problematic—WinTab's API design puts all responsibility for functional conformance on the shoulders of the driver's implementers (third parties). Some already released WinTab drivers' behavior deviated slightly from the spec, so it became difficult or impossible for the TIS to support WinTab in a generic way. Contrast this with the HID model in which the device or device driver has to specify only supported functionality (named *usages*) and Windows takes care of the API nuances. It was therefore decided that HID devices were the better way to go.

The HID driver, like most other Windows XP drivers, runs at the kernel level, constantly acquiring data from the tablet device and packaging it into HID format if it needs to. This allows the TIS to read input data from the device in a generic fashion.

The Center of the TIS Universe: Wisptis.exe

Arguably the most interesting piece in both Figures 4-1 and 4-2 is a process called Wisptis.exe because it's pretty much the heart and soul of the TIS. It acts like a hub between the HID driver of tablet hardware and applications, and it turns out to be responsible for realizing most of the requirements of the input system we defined earlier in the chapter. The Wisptis process performs the retrieval of input from the tablet devices, mouse emulation, detection of pen-based actions, and the dispatching of events to tablet-aware applications.

Why Wisptis.exe?

Wisptis.exe refers to "WISP TIS." WISP (Windows Ink Services for Pen) was the former name for the Tablet PC Platform, and TIS—well, hopefully at this point in the chapter you know what that means.

Note Multiple tablet devices may be installed and used at the same time, a feature fully supported by the TIS. This is actually quite a common occurrence because, as we now know, the mouse is considered a tablet device, so if your Tablet PC has an integrated touchpad or is ever docked in a desktop scenario, or if you've attached an external tablet to your desktop machine, from the TIS perspective there will be at least two tablet devices installed.

Getting Input from the Driver

Input is received from the digitizer via the HID driver and from the mouse via a low-level mouse hook. Data received from the digitizer is used for performing mouse emulation and detecting pen-based actions.

It's interesting to note that even though the mouse is viewed as a tablet device and a USB mouse uses a HID driver to communicate with Windows, the

TIS does *not* get mouse data from the HID driver. Why? There are a couple of reasons. First, the mouse might not be USB-based (serial and PS/2 mice are still common) and a generic solution to reading mouse input is desirable; second, User32.dll opens the mouse driver exclusively, making it impossible for the TIS to get at the mouse directly. That's why the TIS gets the mouse input data from User32.dll via a low-level hook.

You might be thinking that a cleaner architectural model would be to integrate tablet support directly into User32.dll, and we'd tend to agree with you. However, a design goal of the TIS (and a practical one at that) was to merely augment the existing OS as much as possible, rather than "invade" it with new code. Additionally, we'll soon see that User32's existing message-based input architecture doesn't lend itself too well to tablet input.

Performing Mouse Emulation

The mouse cursor is controlled by the pen through one of two sets of mappings. The first set is used when the press-and-hold option (see Chapter 2) is disabled. Table 4-1 lists which pen actions will result in what mouse actions.

Table 4-1 A Simple Mapping of Pen Actions to Mouse Actions to Perform Mouse Emulation

User Pen Action	Resulting Mouse Action
In-air pen movement	Mouse movement with no buttons pressed, typically matching the pen tip location but using a hover filter
Pen touches digitizer with no barrel buttons pressed	Mouse left button pressed
Pen touches digitizer with barrel button pressed	Mouse right button pressed
Pen moves across digitizer's surface	Mouse movement with left or right button pressed, depends on which mouse button was determined to be pressed upon pen's contact with digitizer; movement matches pen tip location exactly
Pen lifted from digitizer	Mouse left or right button released, depends on which mouse button was determined to be pressed upon pen's contact with digitizer

This works pretty well in practice, with one small exception. Getting ToolTips to pop up can be pretty tough because the mouse cursor has to be completely still when hovering over, say, a toolbar button. Most people have a slight sway to the pen when they try to hold it still, so getting the mouse cursor to be motionless will typically require too much effort. Thus, a requirement for a *hover filter* is imposed, whose purpose is to ignore small changes in pen movement while the pen is in the air. Chapter 2 covers the hovering problem in more detail.

The second set of mappings is used when the press-and-hold option is enabled; observe that Table 4-2 is a little more complex than Table 4-1.

Table 4-2 The Mapping of Higher-Level Pen Actions to Mouse Actions to Perform Mouse Emulation when Press-and-Hold Is Active

Pen Action	Mouse Action
In-air pen movement	Mouse movement with no buttons pressed, typically matching the pen tip location but using a hover filter
Pen taps the digitizer	Mouse left button pressed, mouse left button released
Pen held down on the digitizer within press-and-hold time window and then pen lifted	Mouse right button pressed, mouse right button released
Pen held down on the digitizer and pen starts to move	Mouse left button pressed
Pen held down on the digitizer within press-and-hold time window and pen starts to move	Mouse right button pressed
Pen held down on the digitizer beyond press-and-hold time-out	Mouse left button pressed
Pen moves across digitizer's surface while held down	Mouse movement with left or right button pressed, depends on which mouse button was determined to be pressed upon pen's contact with digitizer; movement matches pen tip location exactly
Pen lifted from digitizer	Mouse left or right button released, depends on which mouse button was determined to be pressed upon pen's contact with digitizer

Notice how we need to trigger mouse actions on pen actions such as press-and-hold, tap, and drag. This is because when the pen initially touches the digitizer, the fact that press-and-hold is enabled means we don't yet know whether the user wants to perform a left button or a right button operation. You might think that the left mouse button can always get pressed on pen down, and when a press-and-hold occurs the left button would get released immediately, followed by a right button down, but that won't work well at all. Consider the following example: using everyone's favorite accessory, Notepad, select some text and right click it with the mouse. Notice how the selection stays intact and a context menu appears. If we use the "pen touching the digitizer always causes a left button down" method, whenever the user performs a press-and-hold it will cause a mouse left click followed by a right button down. Try doing those actions with the selection in Notepad. You'll see that the selection gets dismissed on the left click, and the right click displays the context menu with different items enabled—definitely not desirable behavior.

Detecting press-and-hold, tap versus drag, and hover filtering brings us to the next bit of functionality Wisptis.exe provides: detection of pen-based actions.

Detecting Pen-Based Actions

One of the things that makes a Tablet PC so appealing is its emphasis on *natural computing*—that is, taking human physiology into account for its input model. It's more natural for most people to use a pen than a mouse, as was discussed in Chapter 2.

Contrasting the pen with the mouse brings up an interesting difference: pens tend to be much "noisier" with their data. That's not really the correct term, as it implies low accuracy of data capture, but what's being referred to is how a pen generates much more subtle variance in movement than a mouse does because of the human factor. A mouse is normally at rest and gets interrupted from that state when we move it, whereas a pen that's held is at the mercy of our central nervous system's accuracy. To improve the Tablet PC user experience, therefore, it's a good idea for certain pen input to be filtered a little.

Note Consider this author's observation: the difficulty of making ToolTips appear is exponentially proportional to the amount of caffeine ingested.

Figure 4-3 is a state diagram of how the detection of pen-based actions occurs. The boxes represent events that should be both mapped to mouse input and sent to tablet-aware applications.

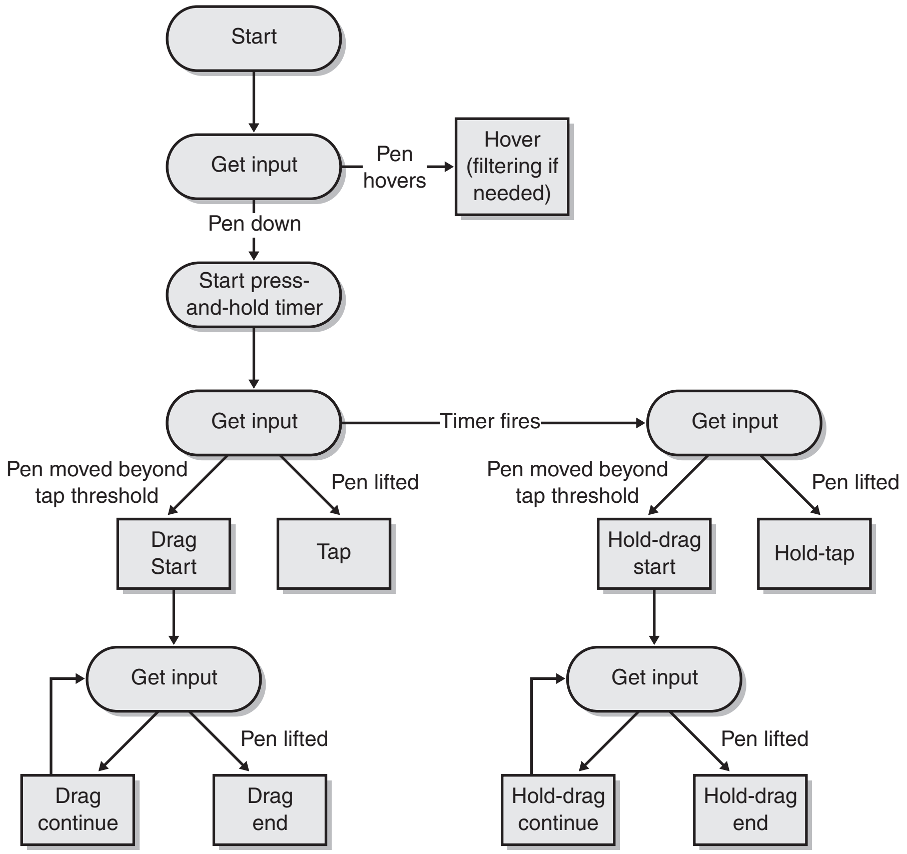


Figure 4-3 A high-level state diagram illustrating how pen-based actions are detected

The precise algorithms for determining pen hovering and tap versus drag won't be covered here because they're a little out of this book's scope.

Dispatching Events

It's not helpful if Wisptis.exe can determine all sorts of useful pen and mouse input but can't tell anybody about it. Luckily for us, the last key function Wisptis.exe performs is client notification of input events for both mouse and tablet input.

Mouse input is easy enough to send to applications because User32.dll provides a convenient function named *SendInput* to automate mouse action. Tablet input, on the other hand, needs a more efficient mechanism because there's typically so much data to send. Remember that in order to have great ink, the frequency of data sampling must be high and pen handling such as pressure and tilt should be captured if possible. Both of these variables raise the required data throughput of tablet input data across processes substantially. The Windows mouse message architecture would not be able to efficiently handle the data throughput requirements of tablet input because of both the high amount of data per message and its sampling frequency. Realistically behaving ink needs to be responsive and accurate; therefore, another mechanism must be used.

Wisptis.exe communicates tablet input events to a tablet-aware application using RPC (remote procedure) calls and a shared-memory queue. A DLL running in the app's process space, InkObj.dll, receives notifications from Wisptis.exe that events are occurring, reads them from the shared-memory queue, and dispatches them to the appropriate handler in the tablet-aware app. Figure 4-4 illustrates the communication of data between Wisptis.exe and a tablet-aware application.

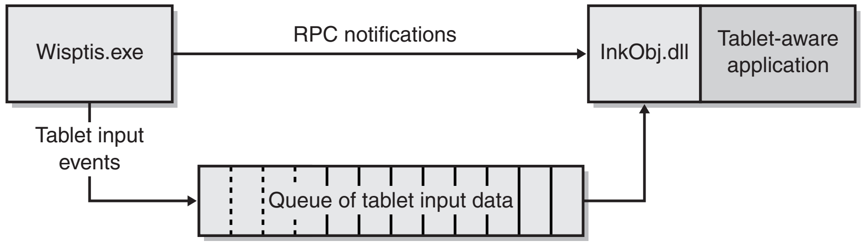


Figure 4-4 Wisptis.exe communicates tablet input data to a tablet-aware application through a queue to avoid losing any data.

Although tablet-unaware applications receive mouse events, it's interesting to note that tablet-aware applications receive *both* mouse events and tablet input events. The reason for this is simple: backward compatibility. Many existing Windows technologies rely only on mouse messages—OLE drag and drop, windowless controls, and even setting the mouse cursor properly all require mouse input. We'll cover some interesting side effects of receiving two sets of events later in the chapter.

Making Sense of It All

Wisptis.exe sure does a lot of stuff, doesn't it? To help illustrate what's going on, the following pseudocode highlights the main functionality of Wisptis.exe.

```
// WISPTIS.EXE Pseudocode
while (true)
{
    foreach (tablet in globalTabletList)
    {
        // First, get raw pen input (down, move, up)
```

(continued)

```
            input = GetInputDataFromHID(tablet);
            if (no input retrieved)
                continue;

            // See if the input is for a tablet-aware app. If
            // it is, dispatch it to the app's instance of InkObj.dll.
            app = GetTargetedApplication(input);
            if (app is tablet-aware)
                DispatchInputEventToQueue(app, input);

            // Now see if a higher-level action can be detected
            penAction = GetPenBasedAction(input, pressAndHoldMode);
            if (no penAction detected yet)
                continue;

            // See if action is for a tablet-aware app. If
            // it is, dispatch it to the app's instance of InkObj.dll.
            if (app is tablet-aware)
                DispatchInputEventToQueue(app, penAction);

            // Map pen action to mouse action. User32.dll will do
            // the "hard stuff" as far as targeting windows, posting
            // messages to queues, etc.
            mouseAction = MapPenActionToMouseAction(penAction);
            DispatchMouseEvents(mouseAction);
        }
    }
```

In essence, the data that is retrieved from the HID driver is used to determine whether the targeted application is tablet aware. If the application is tablet aware, the data is dispatched to it. The data is then processed by some code to detect pen-based actions, and if an action is detected the pen action is then dispatched to the targeted tablet-aware application, if there is one. Finally, the corresponding mouse behavior is performed.

Winlogon Desktop Support

The following applies only to Windows XP Tablet PC Edition. Notice how the diagram in Figure 4-1 is divided into left and right portions—this logical division represents the *Winlogon "secure" desktop* and the *Application desktop*. The Winlogon desktop is the one you see at the Windows XP login screen; it operates at a high security level. When the Winlogon desktop is active (for example, the user is logging in or is unlocking his or her machine), the Tablet PC Input Panel (TIP) can be used to enter credentials. Because the Wisptis.exe process executes only in a user's session, mouse emulation is then needed to be able to use the

TIP. A small DLL file, named Tpgwlnot.dll, executes in the Winlogon.exe process and performs that mouse emulation, though only with basic functionality—no pen-based actions are detected. Tpgwlnot.dll also launches Wisptis.exe when a user first logs on and restarts it if needed (for example, Wisptis.exe terminates because the user ends the process or an exception occurs).

What About Ink?

So far, everything we've covered in the Tablet PC's TIS has met the requirements of the input system we defined, except for one thing: real-time ink! Some of the pieces are there—such as good data capture and throughput—but rendering and storing the strokes in memory aren't. The Tablet PC Platform supports this, though not as part of the TIS. Instead, that functionality lies in the domain of ink data management, mostly the subject of the next two chapters.

Now that we conceptually understand what the tablet input subsystem does, let's actually use it to do something, shall we? It's time to write some code!

Platform SDK Support for Tablet Input

There are two key classes in the Tablet PC managed API that facilitate tablet input—the *InkCollector* class and the *InkOverlay* class. You may also recall that the Tablet PC Platform SDK provides some controls that perform tablet input as well; they will be the subject of Chapter 8. For the time being, we'll focus on tablet input at the class level.

Getting Ink from a Tablet

Real-time inking is arguably the most desirable functionality in a Tablet PC application. After all, that's one of the key differentiators between a Tablet PC and a traditional PC, and it turns out to be one of the most nontrivial to implement. The designers of the Tablet PC Platform realize this, and they have turned a non-trivial task into a trivial one by packaging real-time inking functionality into the *InkCollector* and *InkOverlay* classes.

Say Hello to the *InkCollector*

We'll start off by looking at *InkCollector*—a class whose primary purpose is to provide real-time ink input to an application. *InkCollector* objects use a Windows Forms–based window as an *ink canvas*—a rectangular region in which pen input will be captured. This window is commonly referred to as the *InkCollector*'s *host window*.

The *InkCollector* class can provide an application with useful events such as system gesture detection and ink gesture recognition if desired. It also remembers the ink that the user has drawn, so repaints of the host window preserve any ink that was previously drawn. The bonus here is that the *InkCollector* class is extremely easy to use, as you'll see in this first sample application.

Note Recall that only Windows XP Tablet PC Edition ships "out of the box" with ink recognition capability.

Sample Application: "HelloInkCollector"

Let's dive right into learning about using *InkCollector* by looking at some code. This sample shows the most straightforward use of the *InkCollector* class in an application: a form is created and an *InkCollector* instance is attached to the window. Digital ink can then be drawn on the form, as shown in Figure 4-5, using the tablet hardware installed in the system, including the mouse.

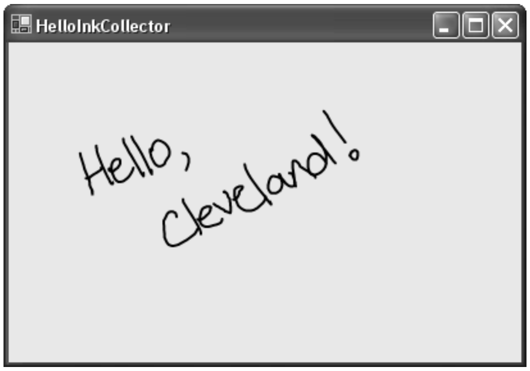


Figure 4-5 Greetings from the HelloInkCollector sample application

Perhaps what's most surprising about the HelloInkCollector application is that the key functionality is only two lines of code! Check it out:

HelloInkCollector.cs

```
////////////////////////////////////////////////////////////////////
//
// HelloInkCollector.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program demonstrates the simplest usage of the InkCollector
// class.
//
////////////////////////////////////////////////////////////////////

using System;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
{
    private InkCollector inkCollector;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }

    public frmMain()
    {
        // Set up the form which will be the host window for an
        // InkCollector instance
        ClientSize = new Size(400, 250);
        Text = "HelloInkCollector";

        // Create a new InkCollector, using the form for the host
        // window
        inkCollector = new InkCollector(Handle);

        // We're now set to go, so turn on ink collection
        inkCollector.Enabled = true;
    }
}
```

You'll notice how the Visual Studio .NET forms designer was not used to create the user interface for the application—this was done purposefully. All the sample applications are like this, for two reasons: it simplifies things for those who wish to manually type in the code, and it keeps the code succinct (hopefully) in its meaning.

After creating a form, the HelloInkCollector application includes the form's handle property in the *InkCollector* constructor—this tells *InkCollector* we want the form to be the host window:

```
// Create a new InkCollector, using the form for the host
// window
inkCollector = new InkCollector(Handle);
```

Once the *InkCollector* object has been created, inking functionality can be activated by setting the *Enabled* property to *true*.

```
// We're now set to go, so turn on ink collection
inkCollector.Enabled = true;
```

At this point, the user is free to ink on the form using any installed tablet device. When the form is invalidated the ink will repaint automatically, and if you try to draw ink off the edge of the form, the ink will be clipped to the form's boundaries. Not bad for a couple of lines of code!

Note The HelloInkCollector sample uses an entire form's client area for the ink canvas. If a smaller area in the form is desired, there are three ways to accomplish this: the first method is to use a child window on the form as the host window (which is what the rest of the samples in this chapter do), the second method is to specify to *InkCollector* an input rectangle within the host window via the *SetWindowInputRectangle* API, and the third is to set the *InkCollector's* Margin X and Margin Y properties.

Now that we have a basic application that provides inking functionality up and running, let's see how easy it is to get some editing functionality running.

When Ink Is Not Enough

The *InkCollector* class is great at providing real-time ink, but oftentimes you'll want to give your users the ability to select, manipulate, and erase the ink they've drawn. *InkCollector* doesn't have any support for this, but it's definitely possible to augment *InkCollector* and write all that functionality yourself. However, that would be a rather time-consuming task, and quite a wasteful one—

especially if only standard ink interaction behavior was desired! Tablet PC developers everywhere would be reinventing the wheel, which isn't exactly an indicator of a great software platform. Fortunately, the Tablet PC Platform SDK provides a class named *InkOverlay* that implements common ink-interaction behaviors—it supports selecting, moving, resizing, and erasing ink, as well as all the real-time inking capability that *InkCollector* has.

Note *InkOverlay* is a proper superset of the *InkCollector*—an instance of *InkCollector* can be replaced by an instance of *InkOverlay* and it will always function identically.

The Ink Controls: InkPicture and InkEdit

In addition to the *InkCollector* and *InkOverlay* classes, the Tablet PC Platform provides two controls that are capable of accepting input: InkPicture and InkEdit. They are both Windows Forms controls and are designed to make forms-based ink capture easier. We'll discuss them in detail in Chapter 8.

InkOverlay has a property named *EditingMode* that indicates the input behavior (or input *mode*) that should be currently active. The property is of the type *InkOverlayEditingMode*. Table 4-3 lists its members and the resulting behaviors.

Table 4-3 The Members of *InkOverlayEditingMode* and Their Meanings

Member	Editing Behavior
Ink	Real-time inking mode—ink is drawn wherever the pen touches in the input area. *InkOverlay* will act just like *InkCollector*.
Select	Selection mode—tapping or lassoing ink selects it, and tapping on white space dismisses the selection. The selection can be moved or resized.
Delete	Eraser mode—ink is erased whenever encountered by the pen. The erase granularity is either at the stroke level or the point level, determined by *InkOverlay*'s *EraseMode* property.

Sample Application: HelloInkOverlay

Demonstrating most of the extra functionality that *InkOverlay* has over *Ink-Collector* is quite easy. This next sample application is similar to HelloInkCollector except it uses a panel control as the host window, adds a ComboBox to change the *EditingMode*, and adds a push button to change ink color. You could also use a panel as the host window and include the ability to change ink color in HelloInkCollector because inking functionality is identical between *InkCollector* and *InkOverlay*. For the first sample to be as brief as possible we opted not to include them. Figure 4-6 shows what HelloInkOverlay looks like in action.

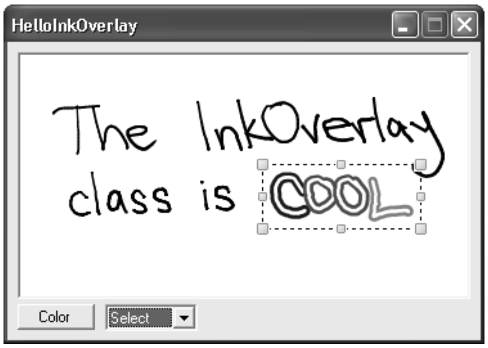


Figure 4-6 The *InkOverlay* class provides everything *InkCollector* does and also has selection and erasing abilities.

HelloInkOverlay.cs

```
/////////////////////////////////////////////////////////////////////
//
// HelloInkOverlay.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program demonstrates basic usage of the InkOverlay class.
//
/////////////////////////////////////////////////////////////////////

using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
```

```
{
    private Panel       pnlInput;
    private Button      btnColor;
    private ComboBox    cbxEditMode;
    private InkOverlay  inkOverlay;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }

    // Main form setup
    public frmMain()
    {
        SuspendLayout();

        // Create and place all of our controls
        pnlInput = new Panel();
        pnlInput.BackColor = Color.White;
        pnlInput.BorderStyle = BorderStyle.Fixed3D;
        pnlInput.Location = new Point(8, 8);
        pnlInput.Size = new Size(352, 192);

        btnColor = new Button();
        btnColor.Location = new Point(8, 204);
        btnColor.Size = new Size(60, 20);
        btnColor.Text = "Color";
        btnColor.Click += new System.EventHandler(btnColor_Click);

        cbxEditMode = new ComboBox();
        cbxEditMode.DropDownStyle = ComboBoxStyle.DropDownList;
        cbxEditMode.Location = new Point(76, 204);
        cbxEditMode.Size = new Size(72, 20);
        cbxEditMode.SelectedIndexChanged +=
            new System.EventHandler(cbxEditMode_SelIndexChg);

        // Configure the form itself
        ClientSize = new Size(368, 236);
        Controls.AddRange(new Control[] { pnlInput,
                                          btnColor,
                                          cbxEditMode});
        FormBorderStyle = FormBorderStyle.FixedDialog;
        MaximizeBox = false;
        Text = "HelloInkOverlay";

        ResumeLayout(false);
```

(continued)

HelloInkOverlay.cs *(continued)*

```
        // Fill up the editing mode combobox
        foreach (InkOverlayEditingMode m in
            InkOverlayEditingMode.GetValues(
            typeof(InkOverlayEditingMode)))
        {
            cbxEditMode.Items.Add(m);
        }

        // Create a new InkOverlay, using pnlInput for the
        // collection area
        inkOverlay = new InkOverlay(pnlInput.Handle);

        // Set eraser mode to be point-level rather than stroke-level
        //inkOverlay.EraserMode = InkOverlayEraserMode.PointErase;
        //inkOverlay.EraserWidth = 200;

        // Select the current editing mode in the combobox
        cbxEditMode.SelectedItem = inkOverlay.EditingMode;

        // We're now set to go, so turn on tablet input
        inkOverlay.Enabled = true;
    }

    // Handle the click of the color button
    private void btnColor_Click(object sender, System.EventArgs e)
    {
        // Create and display the common color dialog, using the
        // current ink color as its initial selection
        ColorDialog dlgColor = new ColorDialog();
        dlgColor.Color = inkOverlay.DefaultDrawingAttributes.Color;
        if (dlgColor.ShowDialog(this) == DialogResult.OK)
        {
            // Set the current ink color to the selection chosen in
            // the dialog
            inkOverlay.DefaultDrawingAttributes.Color = dlgColor.Color;
        }
    }

    // Handle the selection change of the editing mode combobox
    private void cbxEditMode_SelIndexChg(object sender,
        System.EventArgs e)
    {
        // Set the current editing mode to the selection chosen
        // in the combobox
        inkOverlay.EditingMode =
            (InkOverlayEditingMode)cbxEditMode.SelectedItem;
    }
}
```

That's a fair bit longer of a listing than HelloInkCollector, isn't it? There isn't much more Tablet Input API usage, though—you'll notice that most of the extra code deals with the child controls on the form. Let's take a closer look at the interesting parts of the sample.

After creating the child controls, placing them on the form, and filling up the ComboBox using C#'s awesome reflective abilities, an *InkOverlay* object is created, specifying the panel control as the host window:

```
// Create a new InkOverlay, using pnlInput for the
// collection area
inkOverlay = new InkOverlay(pnlInput.Handle);
```

That's a bit different from using the entire form as the host window, but *InkCollector* and *InkOverlay* can handle this situation nicely (pardon the pun). Ink will be clipped to the edge of the control, and the user won't be able to start inking outside the control's boundaries. By using the 3-D border effect and white background on the panel control we get a nice visual representation of where the user can and cannot ink.

Next the selection in the ComboBox is updated using the *EditingMode* property of *InkOverlay*, and then tablet input is enabled.

```
 // Select the current editing mode in the combobox
cbxEditMode.SelectedItem = inkOverlay.EditingMode;

// We're now set to go, so turn on tablet input
inkOverlay.Enabled = true;
```

This code snippet changes the color of the ink using the common color dialog:

```
    // Handle the click of the color button
    private void btnColor_Click(object sender, System.EventArgs e)
    {
        // Create and display the common color dialog, using the
        // current ink color as its initial selection
        ColorDialog dlgColor = new ColorDialog();
        dlgColor.Color = inkOverlay.DefaultDrawingAttributes.Color;
        if (dlgColor.ShowDialog(this) == DialogResult.OK)
        {
            // Set the current ink color to the selection chosen in
            // the dialog
            inkOverlay.DefaultDrawingAttributes.Color = dlgColor.Color;
        }
    }
```

InkCollector and *InkOverlay* objects keep a set of ink rendering properties around named *default drawing attributes*. These are characteristics such as color, thickness, and pen tip style that are encapsulated by a class named *DrawingAttributes*. *InkCollector* uses a property of type *DrawingAttributes* to maintain the default drawing attributes, named *DefaultDrawingAttributes*.

Subsequent strokes created in the *InkCollector* will take on the new color set from the dialog. More detailed coverage of drawing attributes and ink rendering will be covered in the next chapter.

Lastly, when the *EditingMode* ComboBox selection is changed, the editing mode of the *InkOverlay* instance is updated.

```
// Handle the selection change of the editing mode combobox
private void cbxEditMode_SelIndexChg(object sender,
    System.EventArgs e)
{
    // Set the current editing mode to the selection chosen
    // in the combobox
    inkOverlay.EditingMode =
        (InkOverlayEditingMode)cbxEditMode.SelectedItem;
}
```

When you run the HelloInkOverlay application, it will quickly become apparent just how much functionality *InkOverlay* has. You can draw ink, select the ink, move it, resize it, and erase it—all from a small program.

Changing the Eraser Mode

The *Delete* mode can be either stroke-based or point-based, referring to the granularity of ink that is removed when the stroke is touched. Stroke-based erasure will delete the entire stroke when it's hit, and point-based erasure chops out ink from a stroke when it's hit (much like a real eraser does). The property *EraserMode* in the *InkOverlay* class indicates which form of erasing should be performed. It is of type *InkOverlayEraserMode*, which is an enumeration with two members: *StrokeErase* and *PointErase*. The default value of the *EraserMode* property is *InkOverlayEraserMode.StrokeErase*. Point-based erase has an eraser size—essentially the amount of ink to erase from within a stroke—that is specified by the *EraserWidth* property on an *InkOverlay* object.

Try uncommenting the code just after the *InkOverlay* object is created to play around with the point-level erase functionality:

```
// Create a new InkOverlay, using pnlInput for the
// collection area
inkOverlay = new InkOverlay(pnlInput.Handle);

// Set eraser mode to be point-level rather than stroke-level
inkOverlay.EraserMode = InkOverlayEraserMode.PointErase;
inkOverlay.EraserWidth = 200;
```

The eraser width is specified in 100ths of a millimeter, otherwise known as *HIMETRIC units*—the coordinate measurement used for all ink in the Tablet PC Platform.

The *InkControl* Class in the *BuildingTabletApps* Library

Included on the CD-ROM of this book is the *BuildingTabletApps* library, containing numerous helper classes and functions you are free to leverage in your own applications. The functionality of HelloInkOverlay is encapsulated in the *InkControl* class, used in upcoming chapters' sample applications to provide a "quick and dirty" editing UI, avoiding the replication of HelloInkOverlay's code in every case.

InkOverlay's Attach Mode

Another advantage the *InkOverlay* class has over *InkCollector* is the ability to attach to the host window in two ways. By default, *InkCollector* and *InkOverlay* objects will use the actual host window as the canvas to collect and draw ink on. Depending on the behavior of the host window, though, it is possible for redraw problems to occur.

For example, a host window might draw on itself when an event other than paint occurs, which could result in ink being obscured until the next paint event occurs. Another example is if a control is specified as the host window when the host window belongs to another process (perhaps an OCX is used that is implemented in a separate .exe). In this case, ink collection will fail because the *InkCollector* and *InkOverlay* require the host window to belong to the same process as they do.

To solve these problems, the *InkOverlay* class can use a window of its own to collect and render ink. Setting the *AttachMode* property of an *InkOverlay* to *InkOverlayAttachMode.InFront* results in a transparent window being used instead of the host window. The default value of the *AttachMode* property is *InkOverlayAttachMode.Behind*, the other value in the *InkOverlayAttachMode* enumeration.

The *InkOverlay* class is a great way for your application to get common ink behavior. However, it can't fully provide the ink experience that the Tablet PC can in an application like Windows Journal. A brief summary of the functionality that the *InkOverlay* class *doesn't* provide you is listed here:

- Using the top-of-pen as an eraser
- Press-and-hold (or right-click and right-drag) in ink mode to modelessly switch to select mode
- An insert/remove space mode

- Showing selection feedback in real time (for instance, as the lasso is being drawn, ink becomes selected or deselected immediately as it is enclosed or excluded by the lasso)
- Using a scratchout gesture to delete strokes

Luckily, that's a pretty short list. And these deficiencies are addressed by sample applications in this book. The first two items are covered in this chapter; the next two are covered in the next chapter, and the last is covered in Chapter 7.

Note It's not really fair to do a full-out feature comparison of *InkOverlay* and Windows Journal because Journal was written as an end-to-end application. However, there is a quantifiable set of features that defines an inking experience, and it's that set that is being used to compare the two pieces of software.

Note If the *InkOverlay* class is a superset of *InkCollector* with commonly used functionality, you might ask why *InkCollector* even exists. That's a good question! The most reasonable answer we can come up with is this: *InkCollector* is useful if you want to customize tablet input behavior and when little or none of *InkOverlay's* functionality is desired—that makes a cleaner basis to start from. Otherwise, you might as well always use *InkOverlay* and get its extra functionality for free.

Now that we've seen the surface of tablet input functionality that the Tablet PC Platform provides, let's move on to studying tablet input events of *InkCollector* and *InkOverlay*. To keep things simple, we'll return to using *InkCollector* as the subject for tablet input capture; later on the extra events *InkOverlay* has will be discussed.

InkCollector Events

The default behavior of *InkCollector* and *InkOverlay* is cool—but what if you wanted to extend or alter that behavior, or perform certain custom actions for your own application's needs? For example, you might want your application to

- Be notified whenever an ink stroke is drawn so that the stroke can be serialized and sent over a network connection to another machine, perhaps as part of a collaborative whiteboard application.
- Be notified when a press-and-hold system gesture occurs so that an object can be selected.
- Prevent inking entirely but still receive "raw" tablet input events so that direct-manipulation editing operations can be performed.

Note Applying the term *raw* to tablet input refers to the simplest form of events that occur when a pen interacts with a digitizer: hover, pen down, pen move, and pen up.

InkCollector and *InkOverlay* expose an extensive set of event notifications that can be used to trigger other functionality or alter default behavior. These events can be grouped into various categories of notifications to better understand their purpose.

Ink Stroke Events

This first class of events occurs as a result of digital ink being created. An ink stroke can cause either the *Stroke* event or *Gesture* event to fire when it's created—by default, *InkCollector* and *InkOverlay* do not try to recognize strokes as gestures, so the *Stroke* event always is fired when a stroke is created. *InkCollector* and *InkOverlay* have a property named *CollectionMode* (of type *CollectionMode*) that indicates how gesture recognition should take place—collect ink only and not recognize gestures (the default value of *InkOnly*), collect ink and recognize ink as gestures if possible (*InkAndGesture*), or recognize ink as gestures only (*GestureOnly*). The *Stroke* and *Gesture* events are shown in Table 4-4.

Table 4-4 *Stroke* and *Gesture* Events

Event Name	Event Arguments Class	Description
Stroke	*InkCollectorStrokeEventArgs*	An ink stroke was just created.
Gesture	*InkCollectorGestureEventArgs*	An ink stroke was just created and was recognized as a gesture.

When either the *Stroke* or *Gesture* event fires, the corresponding *EventArgs*-based object has a property named *Cancel* that allows the ink stroke to be thrown away or added to the *InkCollector* or *InkOverlay*'s *Ink* object. By default, the *Stroke* event has this property set to *false* (to mean always save the stroke unless code in the event handler says otherwise), as does the *Gesture* event (to mean always throw the stroke away and fire the *stroke* event unless code in the event handler says otherwise).

Pen Movement Events

The next category of events occurs as a result of discrete physical actions with the cursor. A *cursor* in the Tablet PC Platform sense simply refers to a pen or a mouse. Take a look at the pen movement events in Table 4-5. You can see how the names of these events map easily to their descriptions.

Table 4-5 The Pen Movement Events

Event Name	Event Arguments Class	Description
CursorInRange	*InkCollectorCursorInRangeEventArgs*	The cursor has come within proximity of the digitizer device or hovered into the ink canvas's space.
NewInAirPackets	*InkCollectorNewInAirPacketsEventArgs*	An update of the cursor state when it is hovering.
CursorButtonDown	*InkCollectorCursorButtonDownEventArgs*	A button on the cursor has been pressed.
CursorDown	*InkCollectorCursorDownEventArgs*	The cursor tip has touched the surface of the digitizer.
NewPackets	*InkCollectorNewPacketsEventArgs*	An update of the cursor state when it is on the digitizer's surface.
SystemGesture	*InkCollectorSystemGestureEventArgs*	A system gesture (pen-based action) has occurred.

Table 4-5 The Pen Movement Events *(continued)*

Event Name	Event Arguments Class	Description
CursorButtonUp	*InkCollectorCursorButtonUpEventArgs*	A button on the cursor has been released.
CursorOutOfRange	*InkCollectorCursorOutOfRangeEventArgs*	The cursor has left the proximity of the digitizer or hovered out of the ink canvas's space.

The *CursorInRange* and *CursorOutOfRange* events indicate the cursor is coming in or out of physical range with the ink canvas area—this can mean either horizontally (within the x and y plane) or vertically (if the tablet hardware supports this). The *CursorDown*, *CursorButtonDown*, and *CursorButtonUp* events refer to the cursor tip going down or pen buttons being pressed or released.

The *NewPackets* and *NewInAirPackets* events signal that the current cursor state has been updated. They will be further discussed later in this chapter.

The *SystemGesture* event is one of the most useful events in this list because it refers to the fact that a system gesture (referred to as a pen-based action earlier in the chapter) has been recognized. The *InkCollectorSystemGestureEventArgs* object given to the event handler specifies which system gesture was recognized through its *Id* property—a value in the *SystemGesture* enumeration. System gestures are useful when implementing your own editing behaviors.

Note Members of the *SystemGesture* enumeration include *Tap*, *Drag*, *RightTap*, *RightDrag*, and *DoubleTap*.

Mouse Trigger Events

Mouse events are typically sent alongside tablet input events. The mouse trigger events of the *InkCollector* class, described in Table 4-6, are used to prevent those mouse events from being fired.

Table 4-6 Mouse Trigger Events

Event Name	Event Arguments Class	Description
DoubleClick	*System.ComponentModel.CancelEventArgs*	A DoubleClick event is about to be fired.
MouseDown	*CancelMouseEventArgs*	A MouseDown event is about to be fired.
MouseMove	*CancelMouseEventArgs*	A MouseMove event is about to be fired.
MouseUp	*CancelMouseEventArgs*	A MouseUp event is about to be fired.
MouseWheel	*CancelMouseEventArgs*	A MouseWheel event is about to be fired.

Each event's *EventArg*-based parameter has a *Cancel* property that is initially set to *false*. If the event handler sets the value to *true*, the corresponding mouse event will not fire.

Tablet Hardware Events

The class of events pertaining to tablet hardware occurs when a tablet device is either added or removed from the system. These events are listed in Table 4-7.

Table 4-7 Tablet Hardware Events

Event Name	Event Arguments Class	Description
TabletAdded	*InkCollectorTabletAddedEventArgs*	A new digitizer device has been added to the system.
TabletRemoved	*InkCollectorTabletRemovedEventArgs*	A digitizer device has been removed.

Note The *InkCollectorTabletRemovedEventArgs* class's property *TabletId* is the index into the *Tablets* collection of the *Tablet* object being removed. The *Tablets* collection is introduced in the upcoming section, "Getting Introspective."

Rendering Events (*InkOverlay* Only)

The *InkOverlay* class provides two events related to rendering—the *Painting* event, which indicates that the *InkOverlay* object is about to draw itself, and the *Painted* event, which indicates that drawing is complete. This is shown in Table 4-8.

Table 4-8 *InkOverlay* Rendering Events

Event Name	Event Arguments Class	Description
Painting	*InkOverlayPaintingEventArgs*	The *InkOverlay* is about to paint itself.
Painted	*System.Windows.Forms.PaintEventArgs*	The *InkOverlay* is finished painting itself.

The *Painting* event proves useful if you'd ever want to alter any properties of the *Graphic* object being drawn to, adjust the clipping rectangle, or cancel rendering from happening altogether. The *Painted* event allows you to augment the rendering of the *InkOverlay* with any drawing of your own—for example, when implementing some tagging functionality an application would draw its tag icons in an event handler for the *Painted* event.

Ink Editing Events (*InkOverlay* Only)

The events in this ink editing category, described in Table 4-9, are fairly interesting because they can be used to somewhat alter the *InkOverlay*'s behavior.

Table 4-9 Ink Editing Events

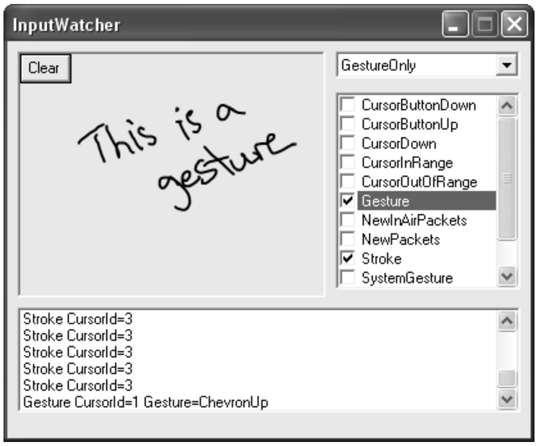
Event Name	Event Arguments Class	Description
SelectionChanging	*InkOverlaySelectionChangingEventArgs*	The selection is about to change.
SelectionChanged	*System.EventArgs*	The selection has changed.
SelectionMoving	*InkOverlaySelectionMovingEventArgs*	The selection is in the process of moving.
SelectionMoved	*InkOverlaySelectionMovedEventArgs*	The selection has been moved.
SelectionResizing	*InkOverlaySelectionResizingEventArgs*	The selection is in the process of being resized.
SelectionResized	*InkOverlaySelectionResizedEventArgs*	The selection has been resized.
StrokesDeleting	*InkOverlayStrokesDeletingEventArgs*	One or more strokes is about to be deleted.
StrokesDeleted	*System.EventArgs*	One or more strokes has been deleted.

The events with the suffix "ing" permit their impending behavior to be changed (or even canceled) by setting relevant data in the *EventArgs*-based object given to an event handler. *SelectionChanging* event's *EventArgs* object makes available for inspection and modification the collection of strokes that is to become selected, *SelectionMoving* makes available for inspection and modification the rectangle of the in-progress move location, *SelectionResizing* makes available for inspection and modification the rectangle of the in-progress resize amount, and *StrokesDeleting* makes available for inspection and modification the collection of strokes to be deleted.

Exposing data such as this enables an application to implement functionality such as read-only ink, unselectable ink, or even remotely automated user interface interaction.

Sample Application: InputWatcher

After all this talking about events, it would be great to get a better idea of exactly what *InkCollector* events get fired, when, and in what order. This next sample application lets you see just that—the events from an *InkCollector* object are monitored. The sample allows you to turn on those events you want to see logged, and when events fire their results are logged to a window. You can also change the collection mode of the *InkCollector* to observe the effect it has. The application is shown in Figure 4-7.



GestureOnly

- [ ] CursorButtonDown
- [ ] CursorButtonUp
- [ ] CursorDown
- [ ] CursorInRange
- [ ] CursorOutOfRange
- [x] Gesture
- [ ] NewInAirPackets
- [ ] NewPackets
- [x] Stroke
- [ ] SystemGesture

```
Stroke CursorId=3
Stroke CursorId=3
Stroke CursorId=3
Stroke CursorId=3
Stroke CursorId=3
Gesture CursorId=1 Gesture=ChevronUp
```

Figure 4-7 InputWatcher logs events from *InkCollector* to an output window.

The source for this sample is quite lengthy, but you might find it's well worth playing around with it in Visual Studio .NET to get a better feel for the various properties on the *EventArgs*-based objects. So here is the source listing in its entirety:

InputWatcher.cs

```
////////////////////////////////////////////////////////////////////
//
// InputWatcher.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program demonstrates how and when events are dispatched for
// the InkCollector class.
//
////////////////////////////////////////////////////////////////////

using System;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
{
    private Panel           pnlInput;
    private ComboBox        cbxMode;
    private CheckedListBox  clbEvents;
    private ListBox         lbOutput;
    private Button          btnClear;
    private InkCollector    inkCollector;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }

    // Main form setup
    public frmMain()
    {
        SuspendLayout();

        // Create and place all of our controls
        pnlInput = new Panel();
        pnlInput.BorderStyle = BorderStyle.Fixed3D;
        pnlInput.Location = new Point(8, 8);
        pnlInput.Size = new Size(240, 192);

        btnClear = new Button();
```

(continued)

InputWatcher.cs *(continued)*

```
        btnClear.Size = new Size(40, 23);
        btnClear.Text = "Clear";
        btnClear.Click += new System.EventHandler(btnClear_Click);

        pnlInput.SuspendLayout();
        pnlInput.Controls.AddRange(new Control[] {btnClear});
        pnlInput.ResumeLayout(false);

        cbxMode = new ComboBox();
        cbxMode.DropDownStyle = ComboBoxStyle.DropDownList;
        cbxMode.Location = new Point(256, 8);
        cbxMode.Size = new Size(144, 21);
        cbxMode.SelectedIndexChanged +=
            new System.EventHandler(cbxMode_SelIndexChg);

        clbEvents = new CheckedListBox();
        clbEvents.CheckOnClick = true;
        clbEvents.Location = new Point(256, 40);
        clbEvents.Size = new Size(144, 154);
        clbEvents.ThreeDCheckBoxes = true;
        clbEvents.ItemCheck +=
            new ItemCheckEventHandler(clbEvents_ItemCheck);

        lbOutput = new ListBox();
        lbOutput.Location = new Point(8, 208);
        lbOutput.ScrollAlwaysVisible = true;
        lbOutput.Size = new Size(392, 94);
        lbOutput.Sorted = false;

        // Configure the form itself
        ClientSize = new Size(408, 310);
        Controls.AddRange(new Control[] { pnlInput,
                                          cbxMode,
                                          clbEvents,
                                          lbOutput});
        FormBorderStyle = FormBorderStyle.FixedDialog;
        MaximizeBox = false;
        Text = "InputWatcher";

        ResumeLayout(false);

        // Fill up the collection mode ComboBox
        foreach (CollectionMode c in
            CollectionMode.GetValues(typeof(CollectionMode)))
        {
            cbxMode.Items.Add(c);
        }
        // Fill up the events ListBox
        clbEvents.Items.Add("CursorButtonDown");
        clbEvents.Items.Add("CursorButtonUp");
        clbEvents.Items.Add("CursorDown");
        clbEvents.Items.Add("CursorInRange");
        clbEvents.Items.Add("CursorOutOfRange");
        clbEvents.Items.Add("DoubleClick");
        clbEvents.Items.Add("Gesture");
        clbEvents.Items.Add("MouseDown");
        clbEvents.Items.Add("MouseMove");
        clbEvents.Items.Add("MouseUp");
        clbEvents.Items.Add("MouseWheel");
        clbEvents.Items.Add("NewInAirPackets");
        clbEvents.Items.Add("NewPackets");
        clbEvents.Items.Add("Stroke");
        clbEvents.Items.Add("SystemGesture");
        clbEvents.Items.Add("TabletAdded");
        clbEvents.Items.Add("TabletRemoved");

        // Create a new InkCollector, using pnlInput for the
        // collection area
        inkCollector = new InkCollector(pnlInput.Handle);

        // Set the selection in the collection mode ComboBox to
        // the current collection mode in inkCollector
        cbxMode.SelectedItem = inkCollector.CollectionMode;

        // We're now set to go, so turn on tablet input
        inkCollector.Enabled = true;
    }

    // Events checked-ListBox item checked handler
    private void clbEvents_ItemCheck(object sender,
        ItemCheckEventArgs e)
    {
        if (e.NewValue == CheckState.Checked)
        {
            // Add the desired event handler to inkCollector
            switch (e.Index)
            {
                case 0:
                    inkCollector.CursorButtonDown +=
                        new InkCollectorCursorButtonDownEventHandler(
                        inkCollector_CursorButtonDown);
                    break;

                case 1:
```

(continued)

InputWatcher.cs *(continued)*

```
                    inkCollector.CursorButtonUp +=
                        new InkCollectorCursorButtonUpEventHandler(
                        inkCollector_CursorButtonUp);
                    break;

                case 2:
                    inkCollector.CursorDown +=
                        new InkCollectorCursorDownEventHandler(
                        inkCollector_CursorDown);
                    break;

                case 3:
                    inkCollector.CursorInRange +=
                        new InkCollectorCursorInRangeEventHandler(
                        inkCollector_CursorInRange);
                    break;

                case 4:
                    inkCollector.CursorOutOfRange +=
                        new InkCollectorCursorOutOfRangeEventHandler(
                        inkCollector_CursorOutOfRange);
                    break;

                case 5:
                    inkCollector.DoubleClick +=
                        new InkCollectorDoubleClickEventHandler(
                        inkCollector_DoubleClick);
                    break;

                case 6:
                    inkCollector.Gesture +=
                        new InkCollectorGestureEventHandler(
                        inkCollector_Gesture);
                    break;

                case 7:
                    inkCollector.MouseDown +=
                        new InkCollectorMouseDownEventHandler(
                        inkCollector_MouseDown);
                    break;

                case 8:
                    inkCollector.MouseMove +=
                        new InkCollectorMouseMoveEventHandler(
                        inkCollector_MouseMove);
                    break;

                case 9:
```

```
                inkCollector.MouseUp +=
                    new InkCollectorMouseUpEventHandler(
                    inkCollector_MouseUp);
                break;

            case 10:
                inkCollector.MouseWheel +=
                    new InkCollectorMouseWheelEventHandler(
                    inkCollector_MouseWheel);
                break;

            case 11:
                inkCollector.NewInAirPackets +=
                    new InkCollectorNewInAirPacketsEventHandler(
                    inkCollector_NewInAirPackets);
                break;

            case 12:
                inkCollector.NewPackets +=
                    new InkCollectorNewPacketsEventHandler(
                    inkCollector_NewPackets);
                break;

            case 13:
                inkCollector.Stroke +=
                    new InkCollectorStrokeEventHandler(
                    inkCollector_Stroke);
                break;

            case 14:
                inkCollector.SystemGesture +=
                    new InkCollectorSystemGestureEventHandler(
                    inkCollector_SystemGesture);
                break;

            case 15:
                inkCollector.TabletAdded +=
                    new InkCollectorTabletAddedEventHandler(
                    inkCollector_TabletAdded);
                break;

            case 16:
                inkCollector.TabletRemoved +=
                    new InkCollectorTabletRemovedEventHandler(
                    inkCollector_TabletRemoved);
                break;
        }
```

(continued)

InputWatcher.cs *(continued)*

```
        }
        else
        {
            // Remove the desired event handler from inkCollector
            switch (e.Index)
            {
                case 0:
                    inkCollector.CursorButtonDown -=
                        new InkCollectorCursorButtonDownEventHandler(
                        inkCollector_CursorButtonDown);
                    break;

                case 1:
                    inkCollector.CursorButtonUp -=
                        new InkCollectorCursorButtonUpEventHandler(
                        inkCollector_CursorButtonUp);
                    break;

                case 2:
                    inkCollector.CursorDown -=
                        new InkCollectorCursorDownEventHandler(
                        inkCollector_CursorDown);
                    break;

                case 3:
                    inkCollector.CursorInRange -=
                        new InkCollectorCursorInRangeEventHandler(
                        inkCollector_CursorInRange);
                    break;

                case 4:
                    inkCollector.CursorOutOfRange -=
                        new InkCollectorCursorOutOfRangeEventHandler(
                        inkCollector_CursorOutOfRange);
                    break;

                case 5:
                    inkCollector.DoubleClick -=
                        new InkCollectorDoubleClickEventHandler(
                        inkCollector_DoubleClick);
                    break;

                case 6:
                    inkCollector.Gesture -=
                        new InkCollectorGestureEventHandler(
                        inkCollector_Gesture);
                    break;
                case 7:
                    inkCollector.MouseDown -=
                        new InkCollectorMouseDownEventHandler(
                        inkCollector_MouseDown);
                    break;

                case 8:
                    inkCollector.MouseMove -=
                        new InkCollectorMouseMoveEventHandler(
                        inkCollector_MouseMove);
                    break;

                case 9:
                    inkCollector.MouseUp -=
                        new InkCollectorMouseUpEventHandler(
                        inkCollector_MouseUp);
                    break;

                case 10:
                    inkCollector.MouseWheel -=
                        new InkCollectorMouseWheelEventHandler(
                        inkCollector_MouseWheel);
                    break;

                case 11:
                    inkCollector.NewInAirPackets -=
                        new InkCollectorNewInAirPacketsEventHandler(
                        inkCollector_NewInAirPackets);
                    break;

                case 12:
                    inkCollector.NewPackets -=
                        new InkCollectorNewPacketsEventHandler(
                        inkCollector_NewPackets);
                    break;

                case 13:
                    inkCollector.Stroke -=
                        new InkCollectorStrokeEventHandler(
                        inkCollector_Stroke);
                    break;

                case 14:
                    inkCollector.SystemGesture -=
                        new InkCollectorSystemGestureEventHandler(
```

(continued)

InputWatcher.cs *(continued)*

```
                        inkCollector_SystemGesture);
                    break;

                case 15:
                    inkCollector.TabletAdded -=
                        new InkCollectorTabletAddedEventHandler(
                        inkCollector_TabletAdded);
                    break;

                case 16:
                    inkCollector.TabletRemoved -=
                        new InkCollectorTabletRemovedEventHandler(
                        inkCollector_TabletRemoved);
                    break;
            }
        }
    }

    // Collection mode ComboBox selection changed handler
    private void cbxMode_SelIndexChg(object sender,
        System.EventArgs e)
    {
        // Turn off ink collection since we're changing collection mode
        inkCollector.Enabled = false;

        // Set the new mode
        inkCollector.CollectionMode =
            (CollectionMode)cbxMode.SelectedItem;

        // Set up the gestures we're interested in recognizing
        if ((inkCollector.CollectionMode ==
            CollectionMode.InkAndGesture) ||
            (inkCollector.CollectionMode ==
            CollectionMode.GestureOnly))
        {
            inkCollector.SetGestureStatus(
                ApplicationGesture.AllGestures, true);
        }
        else
        {
            inkCollector.SetGestureStatus(
                ApplicationGesture.AllGestures, false);
        }

        // We're done, so turn ink collection back on
        inkCollector.Enabled = true;
    }
    // Clear Button clicked handler
    private void btnClear_Click(object sender, System.EventArgs e)
    {
        // Clear out all strokes
        inkCollector.Ink.DeleteStrokes();
        pnlInput.Invalidate();

        // Clear output window
        lbOutput.Items.Clear();
    }

    // Log a string to the output window
    private void LogToOutput(string strLog)
    {
        lbOutput.Items.Add(strLog);
        lbOutput.TopIndex = lbOutput.Items.Count - 1;
    }

    // Various tablet input event handlers - each logs its relevant
    // EventArgs values
    private void inkCollector_CursorButtonDown(object sender,
        InkCollectorCursorButtonDownEventArgs e)
    {
        LogToOutput(String.Format(
            "CursorButtonDown CursorId={0} BtnName={1} BtnId={2}",
            e.Cursor.Id, e.Button.Name, e.Button.Id));
    }

    private void inkCollector_CursorButtonUp(object sender,
        InkCollectorCursorButtonUpEventArgs e)
    {
        LogToOutput(String.Format(
            "CursorButtonUp CursorId={0} BtnName={1} BtnId={2}",
            e.Cursor.Id, e.Button.Name, e.Button.Id));
    }

    private void inkCollector_CursorDown(object sender,
        InkCollectorCursorDownEventArgs e)
    {
        LogToOutput(String.Format(
            "CursorDown CursorId={0} CursorName={1}",
            e.Cursor.Id, e.Cursor.Name));
    }

    private void inkCollector_CursorInRange(object sender,
        InkCollectorCursorInRangeEventArgs e)
    {
```

(continued)

InputWatcher.cs *(continued)*

```
        LogToOutput(String.Format(
            "CursorInRange CursorId={0} Inverted={1} NewCursor={2}",
            e.Cursor.Id, e.Cursor.Inverted, e.NewCursor));
    }

    private void inkCollector_CursorOutOfRange(object sender,
        InkCollectorCursorOutOfRangeEventArgs e)
    {
        LogToOutput(String.Format(
            "CursorOutOfRange CursorId={0}", e.Cursor.Id));
    }

    private void inkCollector_DoubleClick(object sender,
        CancelEventArgs e)
    {
        LogToOutput(String.Format("DoubleClick"));
    }

    private void inkCollector_Gesture(object sender,
        InkCollectorGestureEventArgs e)
    {
        LogToOutput(String.Format(
            "Gesture CursorId={0} Gesture={1} Confidence={2}",
            e.Cursor.Id, e.Gestures[0].Id, e.Gestures[0].Confidence));
    }

    private void inkCollector_MouseDown(object sender,
        CancelMouseEventArgs e)
    {
        LogToOutput(String.Format("MouseDown"));
    }

    private void inkCollector_MouseMove(object sender,
        CancelMouseEventArgs e)
    {
        LogToOutput(String.Format("MouseMove"));
    }

    private void inkCollector_MouseUp(object sender,
        CancelMouseEventArgs e)
    {
        LogToOutput(String.Format("MouseUp"));
    }

    private void inkCollector_MouseWheel(object sender,
        CancelMouseEventArgs e)
    {
        LogToOutput(String.Format("MouseWheel"));
    }

    private void inkCollector_NewInAirPackets(object sender,
        InkCollectorNewInAirPacketsEventArgs e)
    {
        LogToOutput(String.Format(
            "NewInAirPackets CursorId={0} PacketCount={1}",
            e.Cursor.Id, e.PacketCount));
    }

    private void inkCollector_NewPackets(object sender,
        InkCollectorNewPacketsEventArgs e)
    {
        LogToOutput(String.Format(
            "NewPackets CursorId={0} PacketCount={1}",
            e.Cursor.Id, e.PacketCount));
    }

    private void inkCollector_Stroke(object sender,
        InkCollectorStrokeEventArgs e)
    {
        LogToOutput(String.Format(
            "Stroke CursorId={0} Id={1}", e.Cursor.Id, e.Stroke.Id));
    }

    private void inkCollector_SystemGesture(object sender,
        InkCollectorSystemGestureEventArgs e)
    {
        LogToOutput(String.Format(
            "SystemGesture CursorId={0} Id={1} EventLocation=({2},{3})",
            e.Cursor.Id, e.Id, e.Point.X, e.Point.Y));
    }

    private void inkCollector_TabletAdded(object sender,
        InkCollectorTabletAddedEventArgs e)
    {
        LogToOutput(String.Format(
            "TabletAdded TabletName={0}", e.Tablet.Name));
    }

    private void inkCollector_TabletRemoved(object sender,
        InkCollectorTabletRemovedEventArgs e)
    {
        LogToOutput(String.Format(
            "TabletRemoved TabletId={0}", e.TabletId));
    }
}
```

Whew, that *is* a really long listing! But don't worry, it's not as complex as it appears. Unfortunately, there's a lot of bloat that occurs because we are unable to generically add and remove delegates to the *InkCollector*, and also because we can't generically process events received.

The InputWatcher application starts off in a similar fashion to what we've already seen. The main form creates its child controls: a panel to be used as an *InkCollector*'s host window, a button to clear out any ink in the *InkCollector* and any output in the log window, a ComboBox to specify collection mode, a checked ListBox for choosing events to log, and a ListBox used for crude output logging. The user interface elements are then initialized with relevant data, and an *InkCollector* instance is created using the panel as the host window.

Things start to get interesting in the *clbEvents_ItemCheck* event handler, which adds or removes *InkCollector* event handlers from the *InkCollector* object depending on the CheckBox state of the ListBox item. Each handler is responsible for logging interesting properties of the event to the output window.

The *cbxMode_SelIndexChg* event handler deals with the selection changing in the ComboBox that specifies collection mode. Note that the *InkCollector* is disabled when the collection mode is changed and then re-enabled afterward:

```
// Turn off ink collection since we're changing collection mode
inkCollector.Enabled = false;

// Set the new mode
inkCollector.CollectionMode =
    (CollectionMode)cbxMode.SelectedItem;

// Set up the gestures we're interested in recognizing
if ((inkCollector.CollectionMode ==
    CollectionMode.InkAndGesture) ||
    (inkCollector.CollectionMode ==
    CollectionMode.GestureOnly))
{
    inkCollector.SetGestureStatus(
        ApplicationGesture.AllGestures, true);
}
else
{
    inkCollector.SetGestureStatus(
        ApplicationGesture.AllGestures, false);
}

// We're done, so turn ink collection back on
inkCollector.Enabled = true;
```

Some of the *InkCollector*'s properties and methods require tablet input to be shut off in order for them to be used—a safeguard to prevent simultaneous user input and programmatic usage.

The purpose of the *SetGestureStatus* calls in the preceding code is to provide the gesture recognizer component with the list of gestures we're interested in recognizing. This can improve recognition performance and accuracy, which will be discussed in Chapter 7. By default, the gesture recognizer won't try to recognize any gestures; it will merely return *ApplicationGesture.NoGesture* in the *InkCollectorGestureEventArgs* object's *Gestures* array. To get some recognition results, when a collection mode in which gesture recognition can occur is set, we'll ask for every gesture to be recognized for simplicity.

The last piece of code we'll look at in this sample is the *btnClear_Click* event handler. This method removes all strokes from the *InkCollector* with the following code:

```
// Clear out all strokes
inkCollector.Ink.DeleteStrokes();
pnlInput.Invalidate();
```

The *Ink* object attached to the *InkCollector* provides a method named *DeleteStrokes* we can use to remove all the strokes from it. Once the strokes are removed, we need to refresh the host window to reflect the absence of ink in the *Ink* object.

Analyzing the Events

After running InputWatcher and doing some inking on the ink canvas, you'll quickly notice that the log window shows no output. That's because all the events are turned off when the application starts up—try turning on the *Stroke* and *Gesture* events and drawing some ink. Notice how in *InkOnly* collection mode only *Stroke* events occur. If you have a gesture recognizer installed, set the collection mode to *InkAndGesture* and write some text (for instance, your name). You might see some *Gesture* events firing as well as *Stroke* events, meaning that the gesture recognizer thinks one or more of the strokes you wrote looks like a known gesture. If the collection mode is set to *GestureOnly*, you'll notice that only *Gesture* events will fire—this means that every stroke is being recognized as a gesture, although perhaps an unknown one. Additionally, multiple strokes can form a gesture such as a double circle.

Let's take a look at some sequences of events for common actions, so try this: turn on every single event type and draw a stroke across the ink canvas. Whoa! The output window is deluged with loads of events, mostly of the types *MouseMove, NewInAirPackets,* and *NewPackets*. This actually does make sense because those events represent an update to the current state of the cursor, when it's either on the digitizer's surface or hovering over it. More information on what exactly "the current state of the cursor" means will be discussed in the upcoming section, "Specifying the Tablet Data to Capture—Packet Properties."

To reduce the quantity of events logged to the output window, turn off the *MouseDown*, *MouseMove*, *MouseUp*, and *NewInAirPackets* events and tap the Clear button. Now draw a stroke again—this time the output won't be quite as overwhelming. You should see something similar to the following:

```
CursorButtonDown CursorId=8 BtnName=tip BtnId={GUID value}
CursorDown CursorId=8 CursorName=Pressure Stylus
NewPackets CursorId=8 PacketCount=1
NewPackets CursorId=8 PacketCount=13
NewPackets CursorId=8 PacketCount=5
SystemGesture CursorId=8 EventName=Drag EventLocation=(1391,1797)
NewPackets CursorId=8 PacketCount=11
NewPackets CursorId=8 PacketCount=2
NewPackets CursorId=8 PacketCount=1
NewPackets CursorId=8 PacketCount=2
...lots of NewPackets...
Stroke CursorId=8 Id=2
CursorButtonUp CursorId=8 BtnName=tip BtnId={GUID value}
```

The installed tablet devices each have at least one cursor that is used to perform input. To distinguish between various cursors, they are assigned a unique ID, known as the *CursorId*. Each cursor also has one or more cursor buttons, where a cursor button can be either a tip on a stylus or a physical button on a cursor.

The preceding events show that when the pen touches the digitizer surface, a *CursorButtonDown* event is fired and immediately followed by a *CursorDown* event. Remember that a cursor button can be a tip of a pen, so the tip being touched to the digitizer surface is considered the button being pressed down—hence the *CursorButtonDown* event being fired. The *CursorDown* event refers to the fact that a tip or the primary button on the cursor has been pressed down, signaling the start of an ink stroke.

The various *NewPackets* events following the *CursorDown* mean the cursor's input state is updated—that is, the cursor's location and possibly other data such as pressure amount and tilt has been sampled and packaged into one or more packets and then sent to the *InkCollector* instance. These events are received throughout the duration of the cursor being pressed down.

The *SystemGesture* event indicates that a *Drag* system gesture was detected, though in your case it could be *Tap*, *RightDrag*, or *RightTap*, depending on how the stroke was drawn. Notice how the system gesture event didn't fire immediately after *CursorDown* because Wisptis.exe needs to compute whether a tap or a drag is occurring, as we found out earlier in the chapter. The other *NewPackets* events following *SystemGesture* indicate how data is continuously being sampled for the cursor and sent to the *InkCollector* object.

Finally a *Stroke* event is fired, indicating that a new stroke has been created, and a *CursorButtonUp* event is fired, meaning that the cursor has been lifted or its primary button has been released.

Tip The *CursorButtonDown* and *CursorButtonUp* events always bookend each other, as does *CursorDown* with *Stroke* or *Gesture*.

Now try hovering the pen over the input area and clicking the barrel button if it has one. You should see something similar to this output:

```
CursorButtonDown CursorId=8 BtnName=barrel BtnId={GUID value}
CursorButtonUp CursorId=8 BtnName=barrel BtnId={GUID value}
```

The cursor ID is the same as before, but this time the button name and perhaps the GUID value are different, implying that a different button was detected.

If your pen has an eraser tip, try inverting the pen over the input area and then turning it back over to the writing end—observe:

```
CursorInRange CursorId=9 Inverted=True NewCursor=True
CursorOutOfRange CursorId=9
CursorInRange CursorId=8 Inverted=False NewCursor=False
CursorOutOfRange CursorId=8
```

The *NewCursor* property of the *InkCollectorCursorInRangeEventArgs* class discloses whether the *InkCollector* instance has seen that cursor over the *InkCollector's* lifetime. That information can be useful for your application to initialize some data structures to handle unique properties of the cursor, or even to trigger some user interface prompting the user to set some initial properties for the cursor (for instance, ink color and ink vs. eraser functionality). The collection of cursors encountered by the *InkCollector* is provided by the cursor's property of the *InkCollector* class. The preceding example also shows how the eraser tip is considered a different cursor from the ink tip because it has a different cursor ID and *NewCursor* equals *true*. Note also how the *Inverted* property equals *true*—we'll use that information later when we implement top-of-pen erase functionality. When the pen is righted again, we see that the cursor ID is the original value it was before the eraser tip was used—and we also know the ink tip is being used because the *Inverted* property equals *false*, and the *InkCollector* has seen the cursor before.

Mouse Message Synchronization

Events from the *InkCollector* travel a different path through the Windows OS than mouse messages do. *InkCollector* events come from Wisptis.exe, dispatched to a tablet-aware application by RPC. Mouse events are triggered by Wisptis.exe, but come from User32.dll and are dispatched to an application through the application's message queue. Both processes (Wisptis.exe and the application receiving the event) are naturally executing asynchronously to one another. Because of this, you cannot rely on the ordering of *InkCollector* events *in conjunction with* mouse events. It's perfectly OK to rely on the ordering of just *InkCollector* events and/or just mouse events, *but not both together*.

InkOverlay Events

The additional events that the *InkOverlay* class fires, that is, *Painting*, *Painted*, *SelectionChanging*, *SelectionChanged*, *SelectionMoving*, *SelectionMoved*, *SelectionResizing*, *SelectionResized*, *StrokesDeleting*, and *StrokesDeleted*, weren't included in the already long sample for brevity, and for the fact that they are straightforward in their behavior. Hence, it's left as an exercise for the reader to add the *InkOverlay* events to InputWatcher and observe when they fire.

What you'll see in the modified InputWatcher application is that the events suffixed with "ing" always precede their corresponding counterpart suffixed with "ed". For example, *SelectionChanging* will always fire before *SelectionChanged*, and *SelectionMoving* will always fire before *SelectionMoved*.

Specifying the Tablet Data to Capture—Packet Properties

Now it's time to take a closer look at the data received by the *NewPackets* and *NewInAirPackets* events. What exactly is a packet, and what information is in one? To explain this, we'll take another quick dive under the covers of the TIS to see what's going on.

Recall that in addition to the location of the cursor, tablet digitizer hardware may provide other data such as pen pressure, tilt angle, and rotation angle to Wisptis.exe via the HID driver. This data can be used to render ink in a more expressive and realistic manner, and/or provide extended user input capabilities as discussed earlier. The various properties available from a digitizer are known as packet properties, referring to how data is communicated from the HID driver to Wisptis.exe in the form of packets—chunks of data with a format known to both the driver and Wisptis.exe. Packet properties are communicated from Wisptis.exe to a tablet-aware application, though it is the application's responsibility to tell Wisptis.exe which properties it wants to receive.

Tip The availability of various packet properties is totally dependent on the hardware manufacturer (and software driver if applicable) of the tablet device. The sample application DeviceWalker found later in this chapter is able to display exactly which properties are supported by your installed hardware.

The *InkCollector* class exposes packet properties via its *DesiredPacket-Description* property. A packet description is defined to be simply a list of packet properties. As such, the *DesiredPacketDescription* property is an array of zero or more *System.Guids*—valid values of which are found as static members in the *Microsoft.Ink.PacketProperty* class. A few of the properties and their descriptions are given in Table 4-10.

Table 4-10 A Partial Listing of *PacketProperty* Members and Their Descriptions

***PacketProperty* Member Name**	**Description**
X	The *X* coordinate of the pointer's location
Y	The *Y* coordinate of the pointer's location
NormalPressure	The pressure measurement of the pointer
PacketStatus	Private Wisptis data
RollRotation	The rotation angle about the long axis of the pointer
PitchRotation	The rotation angle about the normal vector to the digitizer surface of the pointer

Figure 4-8 shows how packets relate to packet properties, packet property values, and the packet description.

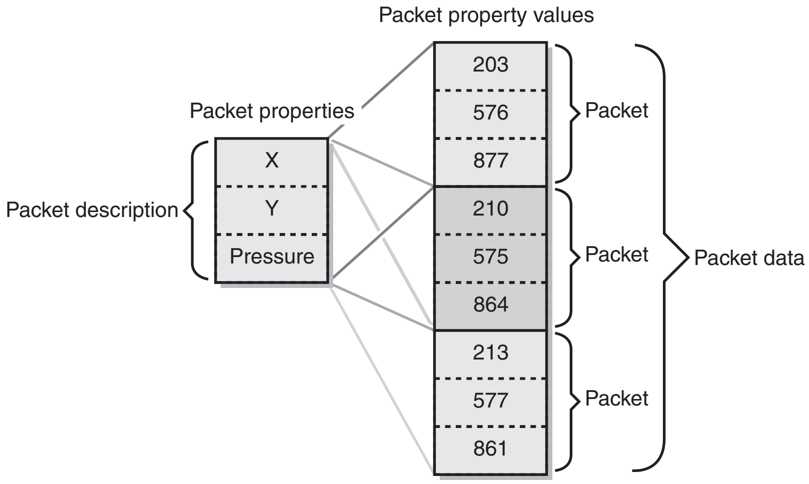


Figure 4-8 The relationship of packets, packet properties, packet property values, and packet description

InkCollector always ensures that *X*, *Y*, and *PacketStatus* are in the packet description it provides to Wisptis.exe, even if *DesiredPacketDescription* is null or doesn't contain those values. In fact, if the *X, Y*, or *PacketStatus* packet property is present in the desired packet description of an *InkCollector* object, that property is ignored. Those packet properties are always prepended by *InkCollector* to the packet description when it's specified to Wisptis.exe to make sure that the location and cursor button state of the pointer is always captured.

If the tablet digitizer you're using supports pressure information, you'll notice that ink drawn in HelloInkCollector will vary in width according to how much pressure was applied. This is because by default the *DesiredPacket-Description* property of an *InkCollector* object is an array containing the *X*, *Y*, and *NormalPressure* packet properties. Figure 4-9 shows what pressurized ink looks like—cool!

If your tablet hardware does support pressure sensitivity, try adding the following line of code in HelloInkCollector just after the *InkCollector* object is created:

```
// We'll ask the InkCollector to not receive any pressure data
// from the devices
inkCollector.DesiredPacketDescription = new Guid [] {
    PacketProperty.X, PacketProperty.Y };
```

When you compile and run the application, you'll notice that ink thickness will not change with the amount of pressure used as strokes are drawn.

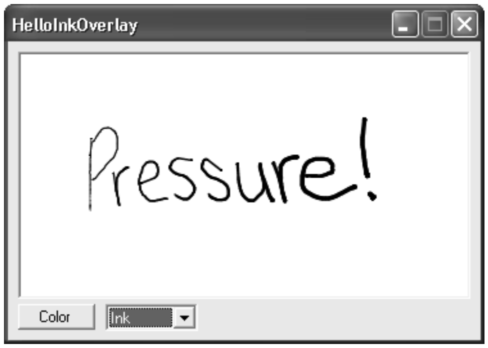


Figure 4-9 The result of pressure support in the HelloInkCollector application

Requesting Packet Properties

If an *InkCollector* object's desired packet description contains packet properties that aren't supported by an installed tablet device, that's OK—they're properties that are *desired*, not required. The *InkCollector* won't yield any data from that tablet for those properties, and it will work fine. To prove this point, notice how ink can be drawn with the mouse in the HelloInkCollector application when the *DesiredPacketProperties* contain *PacketProperty.NormalPressure*—the ink just won't have any pressure data.

We'll see how to determine the list of packet properties that are supported by a tablet device in the next sample application.

The *NewPackets* and *NewInAirPackets* events notify an application that a set of packet property values was received—*NewPackets* indicates the packets were received when the cursor was down, and *NewInAirPackets* indicates the packets were received when the cursor was hovering. They are two separate events instead of one because not all the *EventArgs* data overlaps between them.

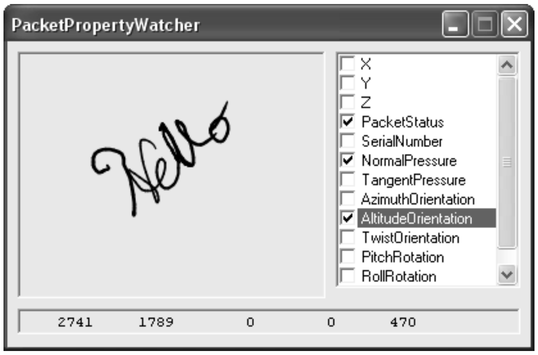
Sample Application: *PacketPropertyWatcher*

To illustrate using packet property values, this sample application will list all supported packet properties for the default tablet device, allow the selection of any number of them, and display the corresponding values as the pointer is used with an instance of *InkCollector*.

Tip Because the Tablet PC Platform supports multiple tablet devices being installed, a default tablet exists to identify which tablet device should be used as the primary one.

If the only tablet device you have installed is a mouse, or if the default tablet device installed is rather meager in capability, chances are that there won't be any packet properties available besides *X* and *Y* coordinates and *PacketStatus*. In this case, this sample application might not clearly demonstrate much functionality, but it is hopefully still useful for reference.

Figure 4-10 shows what the sample looks like in action.



```
2741     1789     0     0     470
```

Figure 4-10 The PacketPropertyWatcher sample application displaying the values of desired packet property types from the default tablet.

Now let's take a look at the source code to PacketPropertyWatcher:

PacketPropertyWatcher.cs

```
////////////////////////////////////////////////////////////////////////////
//
// PacketPropertyWatcher.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program allows the user to choose packet properties to
// collect and then displays their values in real time.
//
////////////////////////////////////////////////////////////////////////////

using System;
using System.Collections;
using System.Drawing;
using System.Reflection;
using System.Text;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
{
    private Panel           pnlInput;
    private CheckedListBox  clbPacketProps;
    private TextBox         txtOutput;
    private InkCollector    inkCollector;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }

    // Main form setup
    public frmMain()
    {
        SuspendLayout();

        // Create and place all of our controls
        pnlInput = new Panel();
        pnlInput.BorderStyle = BorderStyle.Fixed3D;
        pnlInput.Location = new Point(8, 8);
        pnlInput.Size = new Size(240, 192);

        clbPacketProps = new CheckedListBox();
        clbPacketProps.CheckOnClick = true;
        clbPacketProps.Location = new Point(256, 8);
        clbPacketProps.Size = new Size(144, 192);
        clbPacketProps.ThreeDCheckBoxes = true;
        clbPacketProps.ItemCheck += new ItemCheckEventHandler(
            clbPacketProps_ItemCheck);

        txtOutput = new TextBox();
        txtOutput.Font = new Font(FontFamily.GenericMonospace, 8.0f);
        txtOutput.Location = new Point(8, 208);
        txtOutput.ReadOnly = true;
        txtOutput.Size = new Size(392, 24);
        txtOutput.WordWrap = false;
```

(continued)

PacketPropertyWatcher.cs *(continued)*

```
        // Configure the form itself
        ClientSize = new Size(408, 238);
        Controls.AddRange(new Control[] { pnlInput,
                                          clbPacketProps,
                                          txtOutput});
        FormBorderStyle = FormBorderStyle.FixedDialog;
        MaximizeBox = false;
        Text = "PacketPropertyWatcher";

        ResumeLayout(false);

        // Create an InkCollector object
        inkCollector = new InkCollector(pnlInput.Handle);

        // For simplicity, we're only going to have InkCollector
        // use the default tablet since multiple tablets will
        // probably have different sets of supported packet
        // properties - that would make the code/UI more complex
        // and obsfucate what's we're trying to illustrate.
        Tablet t = (new Tablets()).DefaultTablet;
        inkCollector.SetSingleTabletIntegratedMode(t);

        // Fill up ListBox with supported packet properties
        foreach (FieldInfo f in
            typeof(PacketProperty).GetFields())
        {
            // We're only interested in static public members of
            // the class.
            if (f.IsStatic && f.IsPublic)
            {
                Guid g = (Guid)f.GetValue(f);
                if (t.IsPacketPropertySupported(g))
                {
                    clbPacketProps.Items.Add(f.Name);
                }
            }
        }

        // Hook up event handlers to inkCollector
        inkCollector.NewInAirPackets +=
            new InkCollectorNewInAirPacketsEventHandler(
            inkCollector_NewInAirPackets);
        inkCollector.NewPackets +=
            new InkCollectorNewPacketsEventHandler(
            inkCollector_NewPackets);

        // We're now set to go, so turn on ink collection
        inkCollector.Enabled = true;
    }

    // InkCollector stroke event handler
    private void clbPacketProps_ItemCheck(object sender,
        ItemCheckEventArgs e)
    {
        // Get the Guid value of the item being (un)checked
        string n = clbPacketProps.Items[e.Index] as string;
        Guid g = new Guid();
        foreach (FieldInfo f in
            typeof(PacketProperty).GetFields())
        {
            // We're only interested in static public members
            // of the class.
            if (f.IsStatic && f.IsPublic)
            {
                if (f.Name == n)
                {
                    // Found it!
                    g = (Guid)f.GetValue(f);
                    break;
                }
            }
        }

        // Wait until any current input is finished
        while (inkCollector.CollectingInk) {}

        // Turn off ink collection so we can alter the desired
        // packet description
        inkCollector.Enabled = false;

        if (e.NewValue == CheckState.Checked)
        {
            // Add the new packet property to the desired packet
            // properties
            ArrayList propList = new ArrayList(
                inkCollector.DesiredPacketDescription);
            propList.Add(g);
            inkCollector.DesiredPacketDescription =
                (Guid[])propList.ToArray(typeof(Guid));
        }
        else
        {
            // Remove the packet property from the desired
            // packet properties
```

(continued)

PacketPropertyWatcher.cs *(continued)*

```
            ArrayList propList = new ArrayList(
                inkCollector.DesiredPacketDescription);
            propList.Remove(g);
            inkCollector.DesiredPacketDescription =
                (Guid[])propList.ToArray(typeof(Guid));
        }

        // We're done, so turn ink collection back on
        inkCollector.Enabled = true;
    }

    // InkCollector new in-air packets event handler
    private void inkCollector_NewInAirPackets(object sender,
        InkCollectorNewInAirPacketsEventArgs e)
    {
        // Update the output window with the packet data
        UpdateOutput(e.PacketCount, e.PacketData);
    }

    // InkCollector new packets event handler
    private void inkCollector_NewPackets(object sender,
        InkCollectorNewPacketsEventArgs e)
    {
        // Update the output window with the packet data
        UpdateOutput(e.PacketCount, e.PacketData);
    }

    // Updates the UI with new packet values
    private void UpdateOutput(int cPktCount, int [] packetData)
    {
        // This shouldn't ever occur, but let's be safe to avoid a
        // divide-by-zero exception
        if (cPktCount == 0)
        {
            return;
        }

        // Compute the length of one packet
        int nPktLen = packetData.GetLength(0) / cPktCount;

        // Compute the starting index of the last packet
        int nOffset = nPktLen * (cPktCount-1);

        // Get all of the packet property values
        StringBuilder builder = new StringBuilder();
        for (int n = 0; n < nPktLen; n++)
        {
            builder.AppendFormat("{0,8} ",
                packetData[nOffset+n].ToString());
        }

        // Update the EditBox's text
        txtOutput.Text = builder.ToString();
    }
}
```

The main form of the application starts off as usual by creating its child controls: a Panel to be used as an *InkCollector* host window, a CheckedListBox to list all the supported packet properties, and an EditBox to act as a crude UI for displaying the packet property values.

After an *InkCollector* instance is created using *pnlInput* as the host window, it is followed by some rather curious-looking code:

```
Tablet t = (new Tablets()).DefaultTablet;
inkCollector.SetSingleTabletIntegratedMode(t);
```

Recall that by default an *InkCollector* instance receives tablet input from all installed tablet devices. Different tablet devices can (and usually do) have different sets of supported packet properties, so to keep the user interface of this application as simple as possible we'll just deal with the default tablet. Any other installed tablet device could be used, but the default one is typically the most capable. We tell the *InkCollector* which tablet device to accept input from by calling *InkCollector*'s *SetSingleTabletIntegratedMode* method.

Single vs. Multiple Tablet Mode

Most tablet-aware applications will want all installed tablet devices used for input. However, if that's not desirable for whatever reason, the *Ink-Collector* class supports excluding the mouse from input as well as listening to a single tablet device. *InkCollector* constructor overloads or the methods *SetAllTabletsMode* and *SetSingleTabletIntegratedMode* are used, respectively.

The term "integrated" indicates how input on the digitizer device should be mapped to the screen. It loosely refers to the physical relationship of the tablet device to the screen display, though it doesn't enforce that relationship: external tablets are nonintegrated with the display, and tablets layered over the display are integrated. In the integrated case, digitizer input should be mapped to the entire screen area so that the mouse cursor will follow the pen. In the nonintegrated case, digitizer input is mapped to the *InkCollector* object's host window dimensions to typically allow greater precision.

Version 1 of the Tablet PC Platform supports only integrated mapping; sometime in the future, nonintegrated mapping may be added.

The *Tablets* class encapsulates the collection of installed tablet devices in the system—its elements are *Tablet* objects. By simply creating an instance of *Tablets*, the collection is automatically filled and can then be used—or as the preceding code shows, its *DefaultTablet* property can be queried. We'll learn more about the *Tablets* and *Tablet* classes toward the end of the chapter in the section "Getting Introspective."

Next the PacketPropertyWatcher application uses the reflective abilities of C# and the .NET Framework to fill up the CheckedListBox with the packet properties supported by the default tablet device. The *Tablet* class method *IsPacketPropertySupported(Guid g)* determines whether a given tablet device supports a given packet property. As the various packet property GUIDs are iterated over, adding only the supported ones to the CheckedListBox is done as follows:

```
Guid g = (Guid)f.GetValue(f);
if (t.IsPacketPropertySupported(g))
{
   clbPacketProps.Items.Add(f.Name);
}
```

The event handlers to the *InkCollector* events *NewPackets* and *NewInAir-Packets* are added, and the *InkCollector* is enabled so that tablet input can begin.

When an item is checked or unchecked in the CheckedListBox, the *clbPacketProps_ItemCheck* event handler adds or removes the corresponding packet property from the *InkCollector*'s *DesiredPacketDescription*. The first part of the function computes the GUID of the packet property because items in the CheckedListBox are stored as strings, and the second part does the adding or removing:

```
// Wait until any current input is finished
while (inkCollector.CollectingInk) {}

// Turn off ink collection so we can alter the desired
// packet description
inkCollector.Enabled = false;

if (e.NewValue == CheckState.Checked)
{
    // Add the new packet property to the desired packet
    // properties
    ArrayList propList = new ArrayList(
        inkCollector.DesiredPacketDescription);
    propList.Add(g);
    inkCollector.DesiredPacketDescription =
        (Guid[])propList.ToArray(typeof(Guid));
}
else
{
    // Remove the packet property from the desired
    // packet properties
    ArrayList propList = new ArrayList(
        inkCollector.DesiredPacketDescription);
    propList.Remove(g);
    inkCollector.DesiredPacketDescription =
        (Guid[])propList.ToArray(typeof(Guid));
}

// We're done, so turn ink collection back on
inkCollector.Enabled = true;
```

You might be wondering what's going on with that first *while* statement. Well, it turns out that to change the *DesiredPacketDescription* property of an *InkCollector* it must be in the disabled state. When changing the enabled state of an *InkCollector* object, no current ink collection can be occurring or else an exception will be thrown. *InkCollector*'s property *CollectingInk* is polled to cause the program flow to temporarily halt if any in-progress inking is occurring. You might now be thinking that in order for the checked ListBox's *itemChecked* event to fire, the *InkCollector* couldn't be in the middle of inking because the pointer was used to select the item, right? Yes, physically that's what might have happened, but recall the asynchronous execution of Windows messages and tablet input, and how either form of event could occur in any order. If the user is quick enough, he or she could stop inking and immediately choose an item in the ListBox, causing a Windows message to fire indicating an item has been checked or unchecked in the ListBox. This would cause *InkCollector*'s *Enabled* property to be set to *false*, but in the meantime it's possible that the *InkCollector* is *still processing tablet input* from the ink stroke. That would cause the *Enabled* property to throw an exception—an undesirable behavior.

The simple rule here is this: whenever you're going to disable an *InkCollector* object, make sure you wait until any inking is completed to avoid any problems.

Once the *InkCollector* is disabled, the desired packet description is modified to either add or remove the chosen packet property GUID from it. Then the *InkCollector* is re-enabled so that tablet input can resume.

The *inkCollector_NewInAirPackets* and *inkCollector_NewPackets* event handlers call the *UpdateOutput* method to get the packet data displayed in the output window. Packets arrive as an array of integers along with the count of the number of packets contained in the array. Because multiple packets can arrive in one event, *UpdateOutput* displays only the data in the last packet—there's no use in trying to display the others as they'd just quickly flicker by in the output window. You can see how this is handled in the following code:

```
// Compute the length of one packet
int nPktLen = packetData.GetLength(0) / cPktCount;

// Compute the starting index of the last packet
int nOffset = nPktLen * (cPktCount-1);

// Get all of the packet property values
StringBuilder builder = new StringBuilder();
for (int n = 0; n < nPktLen; n++)
{
    builder.AppendFormat("{0,8} ",
        packetData[nOffset+n].ToString());
}

// Update the edit box's text
txtOutput.Text = builder.ToString();
```

The method first computes the length of one packet in the array, easily accomplished by dividing the number of packets into the total array length. An offset to the last packet in the array is computed, and then a for-loop concatenates each packet value to a string. The resulting string is then displayed in the output window.

If we wanted to actually make sense of the various members in the packet, we'd need the packet description for the tablet device. This is easily obtained in a *NewPackets* event handler because a *Stroke* object is available in the *EventArgs* that has a specific *PacketDescription* property. For a *NewInAirPackets* event handler, the packet description would have to be manually computed, and this is accomplished by obtaining the *DesiredPacketDescription* from the *InkCollector*, seeing if each GUID in the packet description is supported by the tablet device using the method *Tablet.IsPacketPropertySupported*, and then constructing an array of supported properties, always prepending the GUIDs for *X* and *Y*.

Notice that in the PacketPropertyWatcher application if you turn off the *X* and *Y* properties their values will still be displayed in the output window, proving that those properties are always collected.

Ink Coordinates

You may have noticed that the *X* and *Y* values in the packet may seem rather large in magnitude—too big to be screen pixels. Recall that tablet input not only needs to be captured quickly, but its resolution must be significantly high enough to give a great inking experience and yield high recognition accuracy.

The *X* and *Y* values in a packet are in ink coordinate space—otherwise known as HIMETRIC units. We'll learn more about them in the next chapter, along with how to convert *X* and *Y* values into screen pixels (for example, to implement your own editing behavior or a custom ink type, perhaps).

Extending InkOverlay Behaviors

The functionality in the *InkOverlay* is cool, and it provides us with some pretty complex functionality for free. As we saw earlier, though, if you contrast *Ink-Overlay* to the inking experience Windows Journal yields, you'll see that some functionality is missing. That functionality is what your application's users may request once they begin using a Tablet PC with your application. Recall the list of missing functionality we enumerated earlier:

- Using the top-of-pen as an eraser
- Press-and-hold (or right-click and right-drag) in ink mode to modelessly switch to select mode
- An insert/remove space mode
- Showing selection feedback in real time (for example, as the lasso is being drawn, ink becomes selected or deselected immediately as it is enclosed or excluded by the lasso)
- Using a scratchout gesture to delete strokes

This section addresses the first two items listed: lack of top-of-pen erase and modeless switching to select mode. The samples here illustrate possible solutions to rolling your own functionality into these areas.

Sample Application: TopOfPenErase

Top-of-pen erase is an extremely handy shortcut to access ink erasing functionality, not to mention an easy one to understand. The *InkOverlay* class provides an explicit editing mode for erasing ink (`EditingMode == DeleteMode`), and we saw earlier that eraser tip usage of a pen is detected through the *Cursor* class's property *Inverted*. If we can put these two pieces of functionality together, it seems that we're most of the way there in implementing top-of-pen erase.

The task at hand here then is to switch an *InkOverlay* into *DeleteMode* when the cursor becomes inverted and switch back to the previous mode when the pen returns to being right side up. Sounds easy enough, doesn't it? Let's enhance the HelloInkOverlay sample presented earlier so that it provides top-of-pen erasing:

TopOfPenErase.cs

```
////////////////////////////////////////////////////////////////////
//
// TopOfPenErase.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program demonstrates usage of the InkOverlay class and how
// to respond to system gestures in order to implement top-of-pen
// erase functionality.
//
////////////////////////////////////////////////////////////////////

using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
{
    private Panel                   pnlInput;
    private Button                  btnColor;
    private ComboBox                cbxEditMode;
    private InkOverlay              inkOverlay;
    private InkOverlayEditingMode   modeSaved;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }

    // Main form setup
    public frmMain()
    {
        SuspendLayout();
        // Create and place all of our controls
        pnlInput = new Panel();
        pnlInput.BorderStyle = BorderStyle.Fixed3D;
        pnlInput.Location = new Point(8, 8);
        pnlInput.Size = new Size(352, 192);

        btnColor = new Button();
        btnColor.Location = new Point(8, 204);
        btnColor.Size = new Size(60, 20);
        btnColor.Text = "Color";
        btnColor.Click += new System.EventHandler(btnColor_Click);

        cbxEditMode = new ComboBox();
        cbxEditMode.DropDownStyle = ComboBoxStyle.DropDownList;
        cbxEditMode.Location = new Point(76, 204);
        cbxEditMode.Size = new Size(72, 20);
        cbxEditMode.SelectedIndexChanged +=
            new System.EventHandler(cbxEditMode_SelIndexChg);

        // Configure the form itself
        ClientSize = new Size(368, 232);
        Controls.AddRange(new Control[] { pnlInput,
                                          btnColor,
                                          cbxEditMode});
        FormBorderStyle = FormBorderStyle.FixedDialog;
        MaximizeBox = false;
        Text = "TopOfPenErase";

        ResumeLayout(false);

        // Fill up the editing mode combobox
        foreach (InkOverlayEditingMode m in
            InkOverlayEditingMode.GetValues(
            typeof(InkOverlayEditingMode)))
        {
            cbxEditMode.Items.Add(m);
        }

        // Create a new InkOverlay, using pnlInput for the collection
        // area
        inkOverlay = new InkOverlay(pnlInput.Handle);

        // Select the current editing mode in the combobox
        cbxEditMode.SelectedItem = inkOverlay.EditingMode;
```

(continued)

TopOfPenErase.cs *(continued)*

```
        // Install handler for cursor in range so we can detect when
        // the eraser tip or ink tip is used.
        inkOverlay.CursorInRange +=
            new InkCollectorCursorInRangeEventHandler(
            inkOverlay_CursorInRange);

        // Initialize the saved editing mode value
        modeSaved = inkOverlay.EditingMode;

        // We're now set to go, so turn on tablet input
        inkOverlay.Enabled = true;
    }

    // Handle the click of the color button
    private void btnColor_Click(object sender, System.EventArgs e)
    {
        // Create and display the common color dialog, using the
        // current ink color as its initial selection
        ColorDialog dlgColor = new ColorDialog();
        dlgColor.Color = inkOverlay.DefaultDrawingAttributes.Color;
        if (dlgColor.ShowDialog(this) == DialogResult.OK)
        {
            // Set the current ink color to the selection chosen in
            // the dialog
            inkOverlay.DefaultDrawingAttributes.Color = dlgColor.Color;
        }
    }

    // Handle the selection change of the editing mode combobox
    private void cbxEditMode_SelIndexChg(object sender,
        System.EventArgs e)
    {
        // Set the current editing mode to the selection chosen in the
        // combobox
        inkOverlay.EditingMode =
            (InkOverlayEditingMode)cbxEditMode.SelectedItem;

        // Save current editing mode in case it gets restored later
        modeSaved = inkOverlay.EditingMode;
    }

    // Handle cursor in range events from inkOverlay
    private void inkOverlay_CursorInRange(object sender,
        InkCollectorCursorInRangeEventArgs e)
    {
        if (e.Cursor.Inverted)
        {
            // Eraser tip is being used, so switch to delete mode if
            // we need to

            if (inkOverlay.EditingMode != InkOverlayEditingMode.Delete)
            {
                // Save current editing mode so we can restore it later
                modeSaved = inkOverlay.EditingMode;

                // Switch to delete mode
                inkOverlay.EditingMode = InkOverlayEditingMode.Delete;
            }
        }
        else
        {
            // Ink tip is being used, so restore previous mode if we
            // need to

            if (inkOverlay.EditingMode == InkOverlayEditingMode.Delete &&
                modeSaved != InkOverlayEditingMode.Delete)
            {
                // Restore the previous editing mode
                inkOverlay.EditingMode = modeSaved;
            }
        }
    }
}
```

The example differs only slightly from the original—a member variable *modeSaved* and an event handler for *CursorInRange* are added. The *inkOverlay_CursorInRange* event handler switches to *DeleteMode* when the pen is inverted and reverts to the previous mode when it is righted again. The *modeSaved* member stores the current *InkOverlayEditingMode* before the switch to *DeleteMode* occurs so that we know which mode to return to when the ink tip gets detected.

Notice how the *modeSaved* member is also updated in *cbxEditMode_SelIndexChg*. This is done to avoid a bug—the accidental reverting to *modeSaved*'s value if the *CursorInRange* event fires when the eraser tip isn't used (for instance, the cursor goes outside the input area and then returns). To see the problem, try this: comment out the line that updates the value and run the application. Switch to select mode, move the pen outside the input area and then back into it—you'll see that ink mode gets switched to instead of select mode.

Sample Application: ModelessSwitch

Explicit mode switching, particularly between inking and editing, is cumbersome, inefficient, and annoying. Using right-tap and right-drag (via press-and-hold, a pen barrel button, or the right mouse button) to access selection and editing behavior is convenient, not to mention powerful.

This next sample application shows an implementation of modeless switching between ink and input modes via right-tap and right-drag. In Chapter 6, at which point we'll better understand rendering and hit-testing ink, we'll utilize some homemade lasso selection capability (with real-time selection feedback!) so that we can complete this application's functionality.

The application is based on the HelloInkCollector sample with one initial change—an *InkOverlay* is used instead of an *InkCollector* to get selection functionality.

ModelessSwitch.cs

```
////////////////////////////////////////////////////////////////////
//
// ModelessSwitch.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program demonstrates the basis for how to do an editing mode
// switch in a modeless fashion using the InkOverlay class.
//
////////////////////////////////////////////////////////////////////

using System;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
{
    private InkOverlay inkOverlay;

    private Stroke  strkCurr = null;
    private Stroke  strkCancel = null;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }
```

```
// Main form setup
public frmMain()
{
    // Set up the form which will be the host window for an
    // InkCollector instance
    ClientSize = new Size(472, 240);
    Text = "ModelessSwitch";

    // Create a new InkOverlay, using the form for the collection
    // area
    inkOverlay = new InkOverlay(this.Handle);

    // Set up event handlers for inkOverlay
    inkOverlay.CursorDown +=
        new InkCollectorCursorDownEventHandler(
        inkOverlay_CursorDown);
    inkOverlay.Stroke +=
        new InkCollectorStrokeEventHandler(
        inkOverlay_Stroke);
    inkOverlay.SystemGesture +=
        new InkCollectorSystemGestureEventHandler(
        inkOverlay_SystemGesture);

    // We're now set to go, so turn on tablet input
    inkOverlay.Enabled = true;
}

// Handle a cursor down event from inkOverlay
private void inkOverlay_CursorDown(object sender,
    InkCollectorCursorDownEventArgs e)
{
    // Remember the current stroke being created
    strkCurr = e.Stroke;
}

// Handle a stroke event from inkOverlay
private void inkOverlay_Stroke(object sender,
    InkCollectorStrokeEventArgs e)
{
    // Throw away the stroke if a right tap or right drag was
    // detected during it's creation
    if (strkCancel != null && (e.Stroke.Id == strkCancel.Id))
    {
        e.Cancel = true;
        strkCancel = null;

        // Turn dynamic stroke rendering back on
```

(continued)

ModelessSwitch.cs *(continued)*

```
                inkOverlay.DynamicRendering = true;
            }

            // Reset current stroke value for the next stroke created
            strkCurr = null;
        }

        // Handle a system gesture event from inkOverlay
        private void inkOverlay_SystemGesture(object sender,
            InkCollectorSystemGestureEventArgs e)
        {
            if (e.Id == SystemGesture.RightTap)
            {
                // Right tap means throw out the stroke, and show a context
                // menu
                strkCancel = strkCurr;
                Invalidate();

                //LATER: hit test the item at e.Point and select it
            }
            else if (e.Id == SystemGesture.RightDrag)
            {
                // Right drag means throw out the stroke, and start up the
                // selection lasso
                strkCancel = strkCurr;
                // Turn off dynamic rendering and set the in-progress ink
                // stroke to be invisible - that way when we invalidate the
                // form no ink will be drawn for the stroke
                inkOverlay.DynamicRendering = false;
                strkCancel.DrawingAttributes.Transparency = 255;
                Invalidate();

                //LATER: start up selection lasso, showing realitme
                // feedback of ink selection
            }
        }
    }
```

You can see there's a fair amount of logic over and above HelloInkCollector to support modeless switching. Let's take a look at what this extra code does.

The implicit mode switch to select mode is triggered by a right-tap or right-drag, which the *SystemGesture* event will notify us of. When either system gesture fires, we want to throw away the ink stroke that was created; otherwise, ugly ink blobs will be added into our *Ink* object.

The system gestures *RightTap* and *RightDrag* are responded to in the *InkOverlay_SystemGesture* event handler. In the *RightTap* case, a reference to the stroke is saved so that it can be later thrown away in the imminent *Stroke* event (by setting *Cancel* to *true* in the *InkCollectorStrokeEventArgs*). For *Right-Drag*, not only is a reference to the stroke saved for later disposal, but also dynamic ink rendering is turned off so that the ink stroke doesn't continue to be drawn. Also, the stroke is made completely invisible by having its drawing attributes' *Transparency* property set to 255 (the maximum value). These both give the user the visual impression that the stroke disappears during creation.

Getting Introspective

Sitting alone in front of a crackling fire, sipping a glass of cabernet, listening to William Shatner's rendition of "Mr. Tambourine Man"… life, the universe—what does it all *mean*? OK, that's actually not what's being referred to here. Rather, we're going to talk about how your tablet-aware application might find it useful to know more about the hardware environment it's running in—perhaps find out how many tablet devices are installed, what their capabilities are, and so on.

This section covers the Platform SDK support to introspect the system, in order to garner all the information you'd ever need about the installed tablet devices.

Tablets Collection

We briefly saw earlier that the *Tablets* class encapsulates the collection of installed tablet devices in the system. The collection's elements are *Tablet* objects, representing a single installed tablet device. You obtain an instance of *Tablets* simply by allocating it—Tablet PC Platform will automatically fill in the contents.

Besides having the normal collection properties and methods, the Tablets class also has a *DefaultTablet* property that identifies the primary tablet device installed in the system.

Tablet Class

Tablet devices installed in the system have various attributes, such as a name, a coordinate system, a list of all the packet properties it can report, and other capabilities such as being able to uniquely identify pens.

The *Tablet* class encapsulates the properties of an installed tablet device. Let's take a look at each of these properties in Table 4-11:

Table 4-11 The Properties of the *Tablet* Class

Property Name	Type	Description
HardwareCapabilities	*TabletHardwareCapabilities*	A bitfield of various device capabilities, defined in the *TabletHardwareCapabilities* enumeration
MaximumInputRectangle	*System.Drawing.Rectangle*	The coordinate space of the entire surface of the tablet device
Name	*String*	A human-readable form of the tablet's name
PlugAndPlayId	*String*	The device name reported to the system by the device

The *HardwareCapabilities* property is a bitfield of values found in the *TabletHardwareCapabilities* enumeration. The various capabilities of tablet hardware that the Tablet PC Platform currently enumerates are as listed in Table 4-12:

Table 4-12 The Members of the *TabletHardwareCapabilities* Enumeration

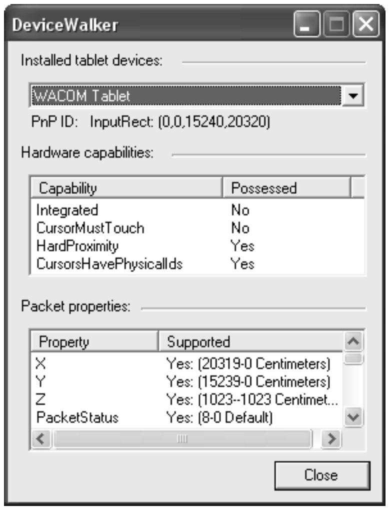
Hardware Capability	Description
CursorMustTouch	The pen must be touching the surface for its position to be sampled.
CursorsHavePhysicalIds	The tablet is able to distinguish between pens used with the device.
HardProximity	The pen's position can be reported while it's in the air but in close proximity to the device.
Integrated	The tablet is integrated with the display.

The *Tablet* class's *IsPacketPropertySupported* method is a useful function that tells you whether the tablet supports or provides a given packet property type. We saw use of this function earlier in the PacketPropertyWatcher sample so that the ListBox could be filled up only with packet properties that the default tablet supported.

The *GetPropertyMetrics* method returns an instance of *TabletPropertyMetrics* given a packet property GUID. *TabletPropertyMetrics* is used to provide the units, resolution, and minimum and maximum values of a packet property type. This is useful information if you ever want to parse packet data beyond just *X* and *Y* values—for example, if you wanted pen pressure or rotation to perform some special behavior, the *TabletPropertyMetrics* of the packet property should be used so that you know how to interpret those packet data values.

Sample Application: DeviceWalker

This final sample application displays all installed tablet devices, their capabilities, and supported packet properties, as shown in Figure 4-11.



DeviceWalker

Installed tablet devices:

WACOM Tablet

PnP ID: InputRect: (0,0,15240,20320)

Hardware capabilities:

Capability	Possessed
Integrated	No
CursorMustTouch	No
HardProximity	Yes
CursorsHavePhysicalIds	Yes

Packet properties:

Property	Supported
X	Yes: (20319-0 Centimeters)
Y	Yes: (15239-0 Centimeters)
Z	Yes: (1023--1023 Centimet...
PacketStatus	Yes: (8-0 Default)

Close

Figure 4-11 The DeviceWalker sample application shows the capabilities of all the tablet devices installed in the system.

DeviceWalker.cs

```
////////////////////////////////////////////////////////////////////
//
// DeviceWalker.cs
//
// (c) 2002 Microsoft Press
// by Rob Jarrett
//
// This program demonstrates introspection of installed Tablet
// devices.
//
////////////////////////////////////////////////////////////////////

using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using Microsoft.Ink;

public class frmMain : Form
{
    private Label       lblTablets;
```

(continued)

DeviceWalker.cs *(continued)*

```
    private Label       lblLine1;
    private ComboBox    cbTablets;
    private Label       lblExtraInfo;
    private Label       lblHardwareCaps;
    private Label       lblLine2;
    private ListView    lvHardwareCaps;
    private Label       lblPacketProps;
    private Label       lblLine3;
    private ListView    lvPacketProps;
    private Button      btnClose;

    // Entry point of the program
    [STAThread]
    static void Main()
    {
        Application.Run(new frmMain());
    }

    // Main form setup
    public frmMain()
    {
        SuspendLayout();

        // Create and place all of our controls
        lblTablets = new Label();
        lblTablets.Location = new Point(8, 8);
        lblTablets.Size = new Size(128, 16);
        lblTablets.Text = "Installed tablet devices:";

        lblLine1 = new Label();
        lblLine1.BorderStyle = BorderStyle.Fixed3D;
        lblLine1.Location = new Point(136, 16);
        lblLine1.Size = new Size(144, 2);

        cbTablets = new ComboBox();
        cbTablets.DropDownStyle = ComboBoxStyle.DropDownList;
        cbTablets.Location = new Point(16, 32);
        cbTablets.Size = new Size(264, 21);
        cbTablets.SelectedIndexChanged +=
            new System.EventHandler(cbTablets_SelIndexChg);

        lblExtraInfo = new Label();
        lblExtraInfo.Location = new Point(16, 56);
        lblExtraInfo.Size = new Size(264, 16);

        lblHardwareCaps = new Label();
        lblHardwareCaps.Location = new Point(8, 80);
        lblHardwareCaps.Size = new Size(120, 16);
        lblHardwareCaps.Text = "Hardware capabilities:";

        lblLine2 = new Label();
        lblLine2.BorderStyle = BorderStyle.Fixed3D;
        lblLine2.Location = new Point(128, 88);
        lblLine2.Size = new Size(152, 3);

        lvHardwareCaps = new ListView();
        lvHardwareCaps.FullRowSelect = true;
        lvHardwareCaps.Location = new Point(16, 104);
        lvHardwareCaps.MultiSelect = false;
        lvHardwareCaps.Size = new Size(264, 80);
        lvHardwareCaps.View = View.Details;

        lblPacketProps = new Label();
        lblPacketProps.Location = new Point(8, 200);
        lblPacketProps.Size = new Size(96, 16);
        lblPacketProps.Text = "Packet properties:";

        lblLine3 = new Label();
        lblLine3.BorderStyle = BorderStyle.Fixed3D;
        lblLine3.Location = new Point(104, 208);
        lblLine3.Size = new Size(176, 3);

        lvPacketProps = new ListView();
        lvPacketProps.FullRowSelect = true;
        lvPacketProps.Location = new Point(16, 224);
        lvPacketProps.MultiSelect = false;
        lvPacketProps.Size = new Size(264, 97);
        lvPacketProps.View = View.Details;

        btnClose = new Button();
        btnClose.DialogResult = DialogResult.OK;
        btnClose.Location = new Point(208, 328);
        btnClose.Text = "Close";
        btnClose.Click +=
            new System.EventHandler(btnClose_Click);

        // Configure the form itself
        AcceptButton = btnClose;
        CancelButton = btnClose;
        ClientSize = new Size(292, 360);
        Controls.AddRange(new Control[] { lblTablets,
                                            lblLine1,
                                            cbTablets,
                                            lblExtraInfo,
                                            lblHardwareCaps,
                                            lblLine2,
```

(continued)

DeviceWalker.cs *(continued)*

```
                                    lvHardwareCaps,
                                    lblPacketProps,
                                    lblLine3,
                                    lvPacketProps,
                                    btnClose });
        FormBorderStyle = FormBorderStyle.FixedDialog;
        MaximizeBox = false;
        Text = "DeviceWalker";

        ResumeLayout(false);

        // Fill the combobox with the currently installed tablet
        // devices
        Tablets tablets = new Tablets();
        foreach (Tablet t in tablets)
        {
            cbTablets.Items.Add(t);
        }

        // Trigger a UI update to fill in the rest of the properties
        cbTablets.SelectedIndex = 0;
    }

    // Tablet device combobox selection changed handler
    private void cbTablets_SelIndexChg(object sender,
        System.EventArgs e)
    {
        // Turn off listview invalidatation for performance
        lvHardwareCaps.BeginUpdate();
        lvPacketProps.BeginUpdate();

        // Remove all items from the listviews
        lvHardwareCaps.Clear();
        lvPacketProps.Clear();

        // Set up their columns
        lvHardwareCaps.Columns.Add("Capability", 150,
            HorizontalAlignment.Left);
        lvHardwareCaps.Columns.Add("Possessed", 100,
            HorizontalAlignment.Left);
        lvPacketProps.Columns.Add("Property", 100,
            HorizontalAlignment.Left);
        lvPacketProps.Columns.Add("Supported", 150,
            HorizontalAlignment.Left);

        // Get the tablet device to introspect
        Tablet t = cbTablets.SelectedItem as Tablet;
        if (t != null)
        {
            // Fill in "extra" info about the tablet
            lblExtraInfo.Text = String.Format(
                "PnP ID: {0}  InputRect: ({1},{2},{3},{4})",
                t.PlugAndPlayId,
                t.MaximumInputRectangle.Left,
                t.MaximumInputRectangle.Top,
                t.MaximumInputRectangle.Bottom,
                t.MaximumInputRectangle.Right);

            // Fill in hardware capabilities by walking through each
            // value in the TabletHardwareCapabilities enum and seeing
            // if the device supports it
            foreach (TabletHardwareCapabilities c in
                TabletHardwareCapabilities.GetValues(
                typeof(TabletHardwareCapabilities)))
            {
                ListViewItem item = new ListViewItem();
                item.Text = c.ToString();
                if ((t.HardwareCapabilities & c) == c)
                {
                    item.SubItems.Add("Yes");
                }
                else
                {
                    item.SubItems.Add("No");
                }
                lvHardwareCaps.Items.Add(item);
            }

            // Fill in packet properties by walking through each value
            // in the PacketProperty class and seeing if the device
            // supports it
            foreach (FieldInfo f in
                typeof(PacketProperty).GetFields())
            {
                // We're only interested in static public members of
                // the class
                if (f.IsStatic && f.IsPublic)
                {
                    ListViewItem item = new ListViewItem();
                    item.Text = f.Name;

                    Guid g = (Guid)f.GetValue(f);
                    if (t.IsPacketPropertySupported(g))
                    {
                        TabletPropertyMetrics tm =
                            t.GetPropertyMetrics(g);
```

(continued)

DeviceWalker.cs *(continued)*

```
                        item.SubItems.Add(
                            String.Format(
                            "Yes: ({0}—{1} {2})",
                            tm.Minimum.ToString(),
                            tm.Maximum.ToString(),
                            tm.Units.ToString()));
                    }
                    else
                    {
                        item.SubItems.Add("No");
                    }
                    lvPacketProps.Items.Add(item);
                }
            }
        }

        // Turn on listview invalidation now that we're done
        lvHardwareCaps.EndUpdate();
        lvPacketProps.EndUpdate();
    }

    // Close Button clicked handler
    private void btnClose_Click(object sender, System.EventArgs e)
    {
        Application.Exit();
    }
}
```

The *cbTablets_SelIndexChg* method does the lion's share of the work in this application—it fills in the UI with the hardware capabilities and supported packet properties of the currently selected tablet device. Again we take advantage of C#'s reflection abilities to enumerate various members of enumerations and classes.

Common Properties on *InkCollector* and *InkOverlay*

In an effort to bring together all that we've learned thus far, we've put together a mini-review of the *InkCollector* and *InkOverlay* classes by listing the commonly used properties, methods, and events in them. Tables 4-13 and 4-14 aren't meant to be an exhaustive reference, merely an effort to summarize the information that has been covered in the chapter.

Table 4-13 *InkCollector* and *InkOverlay* Mini-Reference

Property	Type	Description	Input Can Be Enabled
AutoRedraw	*Bool* (read-write)	Whether to redraw currently captured ink when the host window gets invalidated	Read: Yes Write: Yes
CollectingInk	*Bool* (read-only)	Reports if the *InkCollector* is currently collecting an ink stroke	Read: Yes
CollectionMode	*CollectionMode* (read-write)	Whether the *InkCollector* should recognize ink gestures	Read: Yes Write: No
DefaultDrawingAttributes	*DrawingAttributes* (read-write)	Specifies the drawing attributes to be used when creating new ink strokes	Read: Yes Write: Yes
DesiredPacketDescription	*Guid[]* (read-write)	Specifies which tablet input properties to collect	Read: Yes Write: No
DynamicRendering	*Bool* (read-write)	Whether in-progress ink strokes should be drawn	Read: Yes Write: Yes
Enabled	*Bool* (read-write)	Turns tablet input data capture on and off	Read: Yes Write: N/A
Handle	*IntPtr32* (read-write)	The *InkCollector*'s host window's handle	Read: Yes Write: No
Ink	*Ink* (read-write)	The object used to store collected ink strokes	Read: Yes Write: No

Table 4-14 *InkOverlay* Mini-Reference

Property	Type	Description	Input Can Be Enabled
EditingMode	*InkOverlayEditingMode* (read-write)	The current editing mode used for interaction	Read: Yes Write: Yes
EraserMode	*InkOverlayEraserMode* (read-write)	The type of erasing used when in DeleteMode	Read: Yes Write: Yes
EraserWidth	*int* (read-write)	The eraser width and height for point-level erase	Read: Yes Write: Yes

Best Practices for *InkCollector* and *InkOverlay*

Now that you're armed with a boatload of knowledge about tablet input, it's worth covering a few points of interest in properly using certain facilities. To sum up the core philosophy behind getting the most out of tablet input, "conservation is key."

Let's look at some things you should keep in mind when using either the *InkCollector* or *InkOverlay* classes.

NewPackets and *NewInAirPackets* Events

Remember that tablet events occur a lot more frequently than mouse events do. Performance is crucial when you use the *NewPackets* and *NewInAirPackets* event handlers because they're called so often. Ink performance can suffer dramatically if too much time is taken executing code in them—so if you can, don't even use those handlers in your application. You might be able to use the mouse events instead.

If you do need to use *NewPackets* and *NewInAirPackets*, the code in the handlers should be efficient; try to avoid potentially slow operations such as rendering or network access.

Choosing Desired Packet Properties

Packet property types are cool, but try to be conservative when requesting them. Recall that Wisptis.exe packs the property values into packets that are sent to a tablet-aware application via a packet queue. Because each packet property adds an integer value to each packet, requesting many packet properties can cause the packet size to be large. This can slow down the communication between Wisptis.exe and your application, resulting in poor inking performance.

Gesture Recognition

A similar philosophy to packet properties should be taken with gesture recognition. Gestures can take a long time to recognize, so the set of gestures your application uses should be kept to a minimum. This is not to say gestures are bad because they're definitely not! It's just that executing code that recognizes gestures your application isn't interested in will only slow down getting the results of gestures your application *is* interested in. Therefore, use the *InkCollector*'s *SetGestureStatus* method to specify only the gestures you're interested in being recognized.

Mouse Events

Recall that mouse events occur asynchronously to tablet input events. This can have a negative effect on user interface behavior if you're not careful. If your application changes the user interface state as a result of a tablet event (for example, showing a context menu, bringing up a dialog, or hiding a form), the corresponding mouse event may occur either before or after that user interface change. That may cause some strange behavior that can be tough to debug.

Consider this example—in the *SystemGesture* event handler, an application chooses to show a context menu as the result of a *RightTap*. Sometimes the menu will randomly immediately disappear, not allowing the user to choose any item in it. What's happening? The mouse events that Wisptis.exe is generating are getting processed *after* the tablet input event is, and that causes User32.dll to think that the user has clicked the right-mouse button, causing the menu to be dismissed. Yuck!

The general rule of thumb here is: perform user interface changes only for those mouse events that have an effect in mouse event handlers.

Summary

This chapter covered a lot of material—hopefully everything you'd ever want to know about tablet input.

We started off looking at the requirements and the architecture of the tablet input subsystem in Windows XP Tablet PC Edition and under a Windows-based OS running Tablet PC runtime libraries. We were then introduced to the *InkCollector* and *InkOverlay* classes and how they facilitate any occurring tablet input. We learned about the capabilities of *InkCollector* and *InkOverlay* and how to leverage them in a hands-on sense through various sample applications. And finally we presented some real-world knowledge from folks who have used the stuff.

Now that you know all about collecting ink, the next two chapters will show you how to manipulate it in all sorts of fun ways.