

Ink-Enabling Legacy Thin-client Intranet Applications

by Leszynski Group, Inc.

This paper was originally published on TabletPCDeveloper.com.

Abstract

Ink is a brilliant new user-interface paradigm, and Microsoft has given developers a plethora of tools and components for building rich applications to leverage it. But it's only natural to wonder how this new user-interface (UI) can be made to play nicely with the last great UI paradigm: thin-client applications.

In fact, it's a good bet that there are hundreds (maybe thousands) of existing Web-based applications which could be greatly improved if they could somehow take advantage of the pen-based UI and portable form factor of the Tablet PC. Exploring ways to make that happen is the subject of this paper. The content of this document is technical — readers not interested in the gory details are invited to skip to the bottom, which outlines a roadmap for dealing with legacy Web applications.

The version 1.0 documentation for the Microsoft Tablet PC SDK contained a topic entitled "Adding Ink Controls to a Web Page" [1,2], but that topic did little more than document the CLSIDs for the two ActiveX controls that ship with the SDK: InkEdit and InkPicture. So it seems we're on our own, here. Since we're starting from first principles, let us consider using the nicest, newest technology available to us: .NET.

.NET and the Managed Ink Controls

While you may have already used C# or VB .NET to create client-side Web controls derived from `System.Windows.Forms.Control`, you may be shocked to learn that, for security reasons, it's not really feasible to use the managed ink controls (as found in `Microsoft.Ink.dll`) on a Web page.

For the uninitiated, here is an example of the strange new syntax for the ActiveX `<OBJECT>` tag (supported by Internet Explorer 5.01 and later) overriding the `classid` attribute to support .NET controls (see [3] for a thorough exploration of this new feature of IE):

```
<p>This is an instance of Leszynski.WWWCheckBox  
(which extends System.Windows.Forms.CheckBox):</p>  
<object id="WWWCheckBox1" width="300"  
  classid="http:WWWCheckBox.dll#Leszynski.WWWCheckBox">  
  <param name="BackColor" value="#D0D0A0">  
  <param name="Text" value="This checkbox is brown.">  
</object>
```

At the time of this writing, the `Microsoft.Ink.dll` assembly, although strongly-named and pre-installed into the GAC (the Global Assembly Cache) just like `System.Windows.Forms.dll`, does not contain the all-important `AllowPartiallyTrustedCallers` attribute. This attribute, known as APTCA for

short, is an opt-in policy similar in nature to ActiveX's "safe for scripting" component category (and associated `IObjectSafety` interface). With the APTCA, the publisher of the assembly is guaranteeing that the types hosted within have proven safe against attacks from malicious code (think: buffer overrun attacks, luring attacks, etc). Without the APTCA, only code running with the FullTrust permission set (eg: code obtained from the My Computer zone) will be able to load the `Microsoft.Ink.dll` assembly from the GAC.

So, if you're willing to grant FullTrust to all code originating from a particular site (or zone), then this technique should work well. This solution comes with the hope that someday in the future, Microsoft will enable the APTCA, so that something less than FullTrust will be required to load and use the managed Ink API. (And we should all applaud Microsoft for holding off on the APTCA, until they're confident in its safety — even if it makes our lives a little more difficult, in the meantime.)

Of course, in reality things are a bit more complicated. Realize that there is no (easy) way to author custom controls that derive from the managed ink classes, or otherwise require their presence at run-time -- if you care about supporting both Tablet and non-Tablet browser platforms, you'll need to use Reflection to load the Ink API assembly dynamically. And Reflection is a privileged operation in itself (one not provided for by the default Local Intranet zone policy).

A full discussion of the APTCA and Code Access Security in .NET is a bit beyond the scope of this paper (read [4] to explore the topic further). The moral of the story is this: the security requirements of your customers are going to drive the decision on whether you can use .NET for collecting ink on a web page. So, in the general case, we're stuck in the unmanaged world, for now. Fortunately, the Tablet PC SDK includes a couple of ActiveX controls to help us along: the aforementioned InkEdit and InkPicture.

ActiveX and the Unmanaged Ink Controls

Of the two aforementioned controls, InkEdit is arguably the more useful. After all, if you're interested in collecting ink within an *existing* thin client application, chances are you simply want to "reco" the ink as text and post the text back to the server — effectively using ink as an alternative input mechanism to the keyboard. The case for using InkPicture to collect (and post back to the server?) a vector of full-fidelity ink strokes is a bit harder to envision, but the same techniques apply to both.

Unfortunately, before we can start dreaming up fanciful use-cases for hosting InkPicture objects on a Web page, we must fight with security again. Neither InkEdit nor InkPicture come with an Authenticode signature, nor are they installed into the "safe for scripting" component category. They also lack membership in the "safely initializable" component category — which is just as well because neither control implements the `IPersistPropertyBag` interface either.

For the uninitiated, `IPersistPropertyBag` is the mechanism by which Internet Explorer initializes an ActiveX control based on the `<PARAM>` tags embedded in the HTML:

```
<object classid="clsid:232E456A-87C3-11D1-8BE3-0000F8754DA1"
id="Calendar1">
  <param name="ForeColor" value="0">
  <param name="BackColor" value="16777215">
  <param name="BorderStyle" value="0">
  <param name="Appearance" value="1">
</object>
```

So, even if we can manage to get the controls up and running on a Web page, without `<script>` or `<param>` tags, we won't be able to do much with them!

Fortunately (or not, depending on your point of view) security in the ActiveX world is not nearly as prohibitive as the hard-line enforcement of Code Access Security in .NET. Once installed and running, an unmanaged ActiveX control can do anything it wishes. This means it's possible to develop your own ActiveX controls (call them, say, `InkEditWrapper` and `InkPictureWrapper`) which simply wrap the Microsoft ink controls — or degrade gracefully in their absence. Exploring this route, let us take a moment to enumerate the technical requirements for these hypothetical "ink-wrapper" controls.

Requirements for the Wrapper Controls:

1. Implement `IDispatch` (to support scripting), and enlist in the "safe for scripting" component category.
2. Implement `IPersistPropertyBag` (to support declarative initialization, via `<PARAM>` tags), and enlist in the "safely initializable" component category.
3. Be safe: Take Microsoft at their word, and don't expose any of the underlying controls' interfaces directly, without strict and thorough parameter validation.
4. Degrade gracefully in the absence of the ink controls (to support non-Tablet PC clients).

Point number four poses a bit of a problem — HTTP-savvy readers might be thinking they can get around this by simply redirecting Tablet and non-Tablet clients to different pages (or emitting different DHTML code, etc.) based on the platform id in the HTTP header's User-Agent string. But unfortunately, at the time of this writing (build 2600.xpsp1) the User-Agent string sent by the Tablet PC is exactly the same as the one sent by a Windows XP machine: Tablet PCs are not distinguishable from an ordinary Windows XP machine.

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
.NET CLR 1.0.3705)
```

But perhaps it's point number three that poses the biggest challenge — properly hiding all of the existing interfaces will require writing wrapper implementations for hundreds of methods on the Microsoft controls in a safe, checked manner. Even determining what those interfaces *are* could prove to be a Herculean task, without access to the source code (unlike .NET and Java, COM offers no mechanism for "reflection"). If we relax this requirement, we can simply aggregate the Microsoft control, and all its interfaces, with our own implementation of `IPersistPropertyBag` or `IPersistPropertyBag2` — a generic implementation that forwards the Load and Save methods on to the corresponding `[propput]` and `[propget]` methods on the `IDispatch` interface.

It should be noted that, in theory, this technique of exposing the Microsoft controls openly via "blind aggregation" is almost as big a breach of security as granting FullTrust permission to all code originating in the Local Intranet zone. Any evil-doer that knows the GUID for your signed/safe wrapper control could construct a Web page that exploits a hole in one of the aggregated Microsoft interfaces! (It is for this reason we've blacked-out the GUIDs in the accompanying sample controls -- you'll have to generate your own.)

Of course in practice, finding and exploiting the holes in the Microsoft interfaces would likely be extremely difficult (if there even are any). And this technique at least has the advantage of not requiring modifications to the default security policy on all

of client machines in your organization. If it makes you feel any easier, the official MSDN documentation for the Tablet SDK (version 1.0) includes a topic entitled "Adding Ink Controls to a Web Page" [2], which might be considered an implicit endorsement of these controls' trustworthiness and general suitability for use on a Web page.

Implementing and Testing the Wrapper Controls

Listing 1a shows the ATL header file for an ActiveX wrapper control which simply aggregates the InkEdit control and inherits our generic ATL implementation of IPersistPropertyBag2 (shown in Listing 1b). In Listing 1a, you can see the only "smarts" in the wrapper — detecting the absence of the InkEdit control, and gracefully failing-over to another control in that case. In our example we've chosen to use RichEdit as a failover for non-Tablet clients, but of course you're free to substitute any control you like. For non-trivial applications it's likely something you'll want to develop yourself, to expose a scripting interface that matches InkEdit's more closely.

Listing 1a

```
#pragma once
#include "resource.h"

#include "InkControlWrappers.h"
#include "IPersistPropertyBag2Impl.h"

struct __declspec(uuid("E5CA59F5-57C4-4DD8-9BD6-1DEEEDD27AF4"))
InkEdit;
struct __declspec(uuid("3B7C8860-D78F-101B-B9B5-04021C009402"))
RichEdit;

// CInkEditWrapper
//
// Wraps the Microsoft InkEdit ActiveX control, via "blind
// aggregation".
// On downlevel (non-Ink) platforms, we fall back to wrapping an
// instance
// of the RichEdit control. We're also providing an implementation of
// IPersistPropertyBag2, which delegates to IDispatch, so that the
// control can be initialized with <PARAM> tags.
//
class ATL_NO_VTABLE CInkEditWrapper :
public CComObjectRootEx<CComMultiThreadModel>,
public CComCoClass<CInkEditWrapper, &CLSID_InkEditWrapper>,
public IPersistPropertyBag2DelegatingImpl<CInkEditWrapper>
{
    CComPtr<IUnknown> m_aggregateSP;

public:
    CInkEditWrapper()
    { }

    DECLARE_REGISTRY_RESOURCEID(IDR_INKEDITWRAPPER)

    BEGIN_COM_MAP(CInkEditWrapper)
        COM_INTERFACE_ENTRY(IUnknown)
        COM_INTERFACE_ENTRY(IPersistPropertyBag2)
        COM_INTERFACE_ENTRY_AGGREGATE_BLIND(m_aggregateSP.p)
```

```

END_COM_MAP()

DECLARE_PROTECT_FINAL_CONSTRUCT()
DECLARE_GET_CONTROLLING_UNKNOWN()

HRESULT FinalConstruct()
{
// Try to instantiate the InkEdit control -- this will fail on non-Ink
platforms
    HRESULT hr = m_aggregateSP.CoCreateInstance( __uuidof(InkEdit),
GetControllingUnknown());

// Fallback to some other compatible but non-Ink UI.  We could be
really
// creative here -- it would be best to fallback to a control that
exposes
// the same interface as InkEdit.  Here we use RichEdit, as an example.
if (FAILED(hr))
    hr = m_aggregateSP.CoCreateInstance( __uuidof(RichEdit),
GetControllingUnknown());

    ATLASSERT(SUCCEEDED(hr) && L"Failed to create both InkEdit and
RichEdit!");
return hr;
}

void FinalRelease()
{
    m_aggregateSP.Release();
}
};

OBJECT_ENTRY_AUTO( __uuidof(InkEditWrapper), CInkEditWrapper)

```

Listing 1b

```

#pragma once

// IPersistPropertyBag2DelegatingImpl
//
// Allows an ActiveX control that doesn't implement IPersistPropertyBag
to
// be initialized from a set of name/value pairs, by forwarding the
Load
// method on to a series of IDispatch::Invoke(propput) calls.
//
template <class T>
class ATL_NO_VTABLE IPersistPropertyBag2DelegatingImpl :
public IPersistPropertyBag2
{
public:

// IPersist methods
//
    STDMETHODIMP GetClassID(CLSID* clsidP)
    {
if ( !clsidP)
return E_FAIL;

// Ask derived class for his clsid

```

```

        *clsidP = T::GetObjectCLSID();
return 0;//ok
    }

// IPersistPropertyBag2 methods
//
    STDMETHODIMP InitNew()
    {
// NOP -- derived classes can override, if desired
return 0;//ok
    }

    STDMETHODIMP IsDirty()
    {
//todo: Implement this in the future, if needed
return S_FALSE;//nothing to save
    }

    STDMETHODIMP Save( IPropertyBag2* bagP, BOOL fClearDirty, BOOL
fSaveAllProperties)
    {
//todo: Implement this in the future, if needed
return 0;//ok
    }

    STDMETHODIMP Load( IPropertyBag2* bagP, IErrorLog* errlogP)
    {
        T* tP = static_cast<T*>(this);

if ( !bagP)
return E_INVALIDARG;

        HRESULT hr = 0;//ok

// Get a ref on our own IDispatch interface
        CComDispatchDriver dd = tP->GetUnknown();

// Enumerate properties in bag
        ULONG numProps = 0;
        bagP->CountProperties( &numProps);

// For each property...
for ( unsigned int n=0; n < numProps; ++n)
    {
// Get the name
        PROPBAG2 pb2 = {0};
        ULONG nn = 1;
        hr = bagP->GetPropertyInfo( n, nn, &pb2, &nn);
        ATLASSERT( SUCCEEDED(hr));

// Get the value
        CComVariant propVal;
        bagP->Read( 1, &pb2, errlogP, &propVal, &hr);
        ATLASSERT( SUCCEEDED(hr));

// Delegate to IDispatch (ignoring any errors... misspelled prop names,
etc)
        hr = dd.PutPropertyByName( pb2.pstrName, &propVal);
if (FAILED(hr))

```

```

        ATLTRACE( L"IPersistPropertyBag2::Load -- failed to set
property by name: %s\n", pb2.pstrName);
    }

return 0;//ok
}
};

```

Also noteworthy: Listing 1c is the ATL registrar script which marks the InkEdit control as "safe for scripting" and "safely initializable". In every other way, the wrapper controls are just boilerplate, garden-variety ATL components generated with the "ATL Simple Object" wizard.

Listing 1c

```

HKCR
{
    InkControlWrappers.InkEditWrapper.1 = s 'InkEditWrapper Class'
    {
        CLSID = s '{GGGGGGGG-GGGG-GGGG-GGGG-GGGGGGGGGGG}'
    }
    InkControlWrappers.InkEditWrapper = s 'InkEditWrapper Class'
    {
        CLSID = s '{GGGGGGGG-GGGG-GGGG-GGGG-GGGGGGGGGGG}'
        CurVer = s 'InkControlWrappers.InkEditWrapper.1'
    }
    NoRemove CLSID
    {
        ForceRemove {GGGGGGGG-GGGG-GGGG-GGGG-GGGGGGGGGGG} = s
'InkEditWrapper Class'
        {
            ProgID = s 'InkControlWrappers.InkEditWrapper.1'
            VersionIndependentProgID = s
'InkControlWrappers.InkEditWrapper'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Both'
            }
            'Implemented Categories'
            {
                '{7DD95801-9882-11CF-9FA9-00AA006C42C4}'
                '{7DD95802-9882-11CF-9FA9-00AA006C42C4}'
            }
        }
    }
}
}

```

The following fragment of HTML demonstrates how to include a pair of <OBJECT> tags for your new ActiveX controls. Of course, you'll need to replace the "HHHH" and "GGGG" GUIDs with your own:

```

<object
id="inkeditwrapper1"
classid="clsid:GGGGGGGG-GGGG-GGGG-GGGG-GGGGGGGGGGG">
<!-- todo: replace above with guid of custom inkeditwrapper -->
    <param name="Text" value="This is some text.">
</object>

<object
id="inkpicturewrapper1"
classid="clsid:HHHHHHHH-HHHH-HHHH-HHHH-HHHHHHHHHHHH">

```

```
<!-- todo: replace above with guid of custom inkpicturewrapper -->
  <param name="BackColor" value="red">
</object>
```

If you've been following along up to this point, you may be disappointed to find that it doesn't work — or rather, that the InkEdit control behaves exactly like a RichEdit control on your development machine, even though you have the Tablet SDK installed. Somewhat annoyingly, the unmanaged InkEdit control is designed to behave this way on "down-level" (non-Tablet) platforms, and according to the official documentation for InkEdit [5], there is no way to override this behavior. (In theory, one could probably override this behavior by hooking the Win32 GetSystemMetrics API, to intercept queries for the SM_TABLETPC value, returning 'true' to masquerade as a Tablet. But such extreme measures are beyond the scope of this paper. Interestingly, the unmanaged InkPicture control *does* seem to work correctly on my non-Tablet Windows XP machine, even though the official documentation for InkPicture [6] seems to mimic that of InkEdit.)

Consider it a good excuse to play with a Tablet PC — throw the HTML above on to a Web page somewhere, grab a Tablet, and use it to take your wrapper controls for a test-drive.

Once you've got that up and running, it's time to do something meaningful with the control — like using it to reco some text, and send the results back to the Web server as part of a form-post. The following trivial ASP .NET page is a useful diagnostic tool, to echo the contents of a form-post back to your browser:

```
<%@ Page language="c#" %>
<html>
<head>
  <title>HTTP request echoer</title>
</head>
<body>

<p>Your HTTP request parameters:</p>

<% foreach ( string key in Request.Params.AllKeys) { %>
  <p><b><%= key%>:</b> <%= Request.Params[key] %></p>
<% } %>

</body>
</html>
```

Other Approaches

Hopefully, before the next release of the Tablet PC platform, Microsoft will have a better story for using ink on a Web page. Remember, the approach taken by this paper is really only appropriate for adding rudimentary ink support to an existing, legacy thin-client application. For new development, one should seriously consider rich-client technology, built with Windows Forms and the managed Ink API, and deployed via HTTP. (See [7] for a popular example of WinForms-over-HTTP technology.)

The code in Listing 1 may not seem like very much — but one should not take the security implications lightly. And remember, you'll still have to revisit every control on every form/page in all your intranet applications, and wire them up to use the new InkEditWrapper `<object>` in place of the ordinary text `<input>` fields, and such. It's worthwhile to consider some other alternatives.

Unfortunately, most all of the other alternatives require making at least *some* change to client machines' security settings, or pre-installing some software onto the client machines. But if you're willing and able to go down that road, then a vast array of possibilities emerge.

For example, one could develop a browser-helper object (a BHO) to facilitate ink-input into all Web pages (even Internet pages). A BHO is a COM object that integrates with Internet Explorer, to provide an enhanced user-experience of some sort. Such a mechanism might look and feel similar to TIP (the Tablet PC Input Panel), but it could be tailored for use within IE — perhaps even tailored for use with each individual type of HTML form control. In fact, at Leszynski Group we've developed just such a BHO for some of our customers, and it's proven quite popular.

Another solution, which may be the easiest, is to configure your customers' Tablets to connect through a custom HTTP proxy, which adds a field to callers' HTTP request headers in order to more easily identify them as Tablet PCs. Then configure your Web server to redirect requests based on the presence or absence of that field. (If you're feeling fancy, a custom ISAPI filter could do a truly seamless job of this.)

Throughout this paper, we've touched on a number of possibilities for ink-enabling a legacy thin-client Web application. The following is a general list of questions to consider, as the first step in a roadmap for updating your legacy Web apps to support the Tablet PC platform.

If you don't care about support for non-Tablet client platforms, and don't mind modifying clients' security settings:

Then congratulations! You're ready for the managed world. (Or rather, the managed world is ready for you.) You can simply push an MSI file on to an intranet Web server which does nothing more than give the FullTrust permission to all code originating from that server.

Then rewire your legacy Web apps to make use of managed ink controls (viz: custom .NET controls that derive from Microsoft.Ink.InkEdit or encapsulate functionality from Microsoft.Ink.InkCollector, etc) wherever appropriate.

If you need to support non-Tablet clients, but don't mind pre-installing software onto clients' machines, and don't have any seriously ink-specific new features to add to your legacy Web applications:

Then you might still be in luck — you might be able to get by without touching your existing Web apps' code at all, and develop a browser-helper object (BHO) to be pre-installed onto all your Tablet PC client machines (eg: via publishing the MSI file on IntelliMirror, or an intranet Web server).

If you need to support non-Tablet clients, but don't mind tweaking the configuration on your Tablet clients:

Perhaps you can work around the problem of the Tablet's lackluster User-Agent string by setting up a custom HTTP proxy, as described above. Or, perhaps

even more simply, by choosing to name all of your Tablet PC machines in an easily identifiable way. Any of these approaches should then allow your Web service to redirect Tablet and non-Tablet clients, to an appropriate set of pages/controls.

Any such solution should suffice, as a stop-gap measure until Microsoft gets around to modifying the User-Agent string — hopefully in the next release of the Tablet OS.

If you need to support *both* Tablet and non-Tablet clients, and *cannot* afford to make changes to their security policies or pre-install custom software:

Then and only then, consider using the approach outlined in this paper: wrapping the unmanaged Microsoft ink controls with signed, scriptable, initializable ActiveX controls, which fail or degrade gracefully on non-Tablet platforms.

Bear in mind, you can author your wrapper controls in VB6 instead of ATL; you can call down into the .NET ink controls instead of wrapping the unmanaged controls; or you may have something even more creative in mind — the ATL code presented in this paper is just the simplest possible example of what can be done.

References

- [1] *Adding Ink Controls to a Web Page* (on msdn.microsoft.com — now 404?)
- [2] *Adding Ink Controls to a Web Page* (as shipped in tpcsdk10.chm)
- [3] *DHTML and .NET Host Secure, Lightweight Client-Side Controls in Microsoft Internet Explorer* (Jay Allen, MSDN Magazine, January 2002)
- [4] *.NET Framework Assemblies and the AllowPartiallyTrustedCallers Attribute* (on msdn.microsoft.com)
- [5] *Using InkEdit on Earlier Versions of Windows* (on msdn.microsoft.com)
- [6] *Using InkPicture on Earlier Versions of Windows* (on msdn.microsoft.com)
- [7] *The Game of Wahoo* (a rich-client, HTTP-downloadable WinForms app on www.sellsbrothers.com)

About Leszynski Group Inc.

Leszynski Group develops line-of-business applications and retail software built around databases, XML, and digital ink using the Microsoft .NET architecture. Leszynski Group was the first solution provider to develop and ship ink-based applications for the Tablet PC. For more information and articles, see <http://www.leszynski.com>.