ePadLink®

ePad-vision SDK for Chrome & Firefox Integration Guide

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1.0 – Introduction

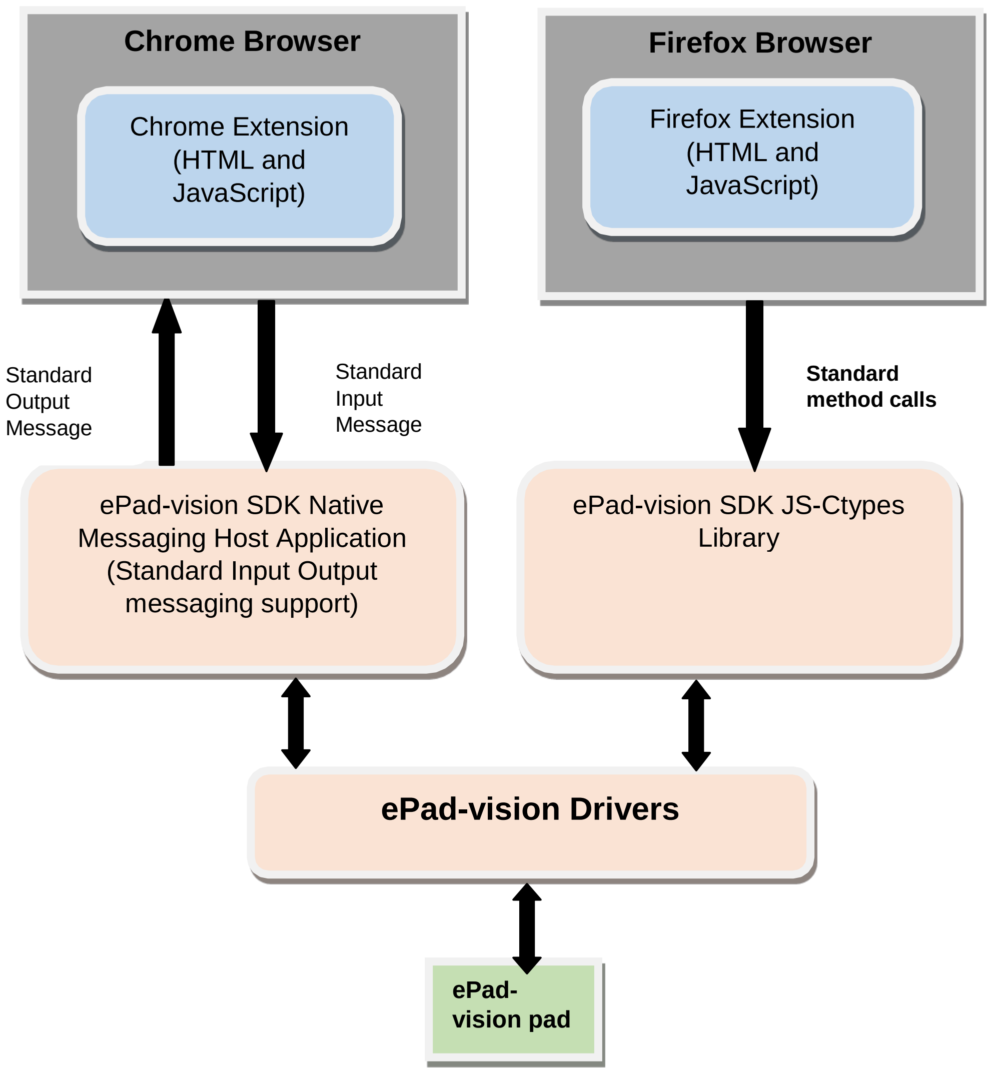
The ePad-vision SDK for Chrome & Firefox offers a mechanism and platform for developers and integrators to capture handwritten signatures securely using ePadLink ePad-vision signature pads for web applications running in the Chrome and Firefox browsers. In addition to signature capture capabilities, the SDK offers the ability to create custom screens with widgets (text, check box, radio button, and push button).

This SDK package does not include the ePad-vision driver, please go to epadsupport.com to download the UI11.6 or later universal installer for Windows or ePad-2.3 driver for Linux system.

2.0 – Overview and Architecture

The diagram below shows the high-level overview of the solution with critical components involved.

Host applications running in Chrome and Firefox browsers interact with the SDK 32-bit native components via extensions. The SDK interacts with the ePad-vision device via standard ePad-vision 32-bit drivers.


Chrome Browser
Chrome Extension (HTML and JavaScript)
Firefox Browser
Firefox Extension (HTML and JavaScript)
Standard Output Message
Standard Input Message
Standard method calls
ePad-vision SDK Native Messaging Host Application (Standard Input Output messaging support)
ePad-vision SDK JS-Ctypes Library
ePad-vision Drivers
ePad-vision pad

2.1 – Chrome Browser

Chrome supports access to local devices (signature pads) using native applications (Native Messaging Hosts) and standard extensions. Web pages can interact with native applications via extensions.

Native Messaging Host Application

The Native Messaging Host is the core component of the SDK and acts as a bridge between the browser and the actual signature capture device. It will have all the input and output interfaces implemented as Standard Input and Output streams as required by the Google Native Messaging API framework. The Native Messaging Host Application processes the input text message from the Chrome browser and executes the request asynchronously, and when the task is complete sends back the status or output data (signature points and widget events) as a native output text message. It will host all the functions for capture of signatures and creation of custom screens on the ePad-vision device.

Chrome runs this application in a separate process and launches it through Connect APIs and sends a notification back to the Chrome Extension when the application is ended by the user.

Chrome Extension/Web Page

Extensions are the HTML, JavaScript, and CSS based applications. The extensions use JavaScript based Native Messaging APIs to launch and communicate with the Topaz Native Messaging Host application for signature capture and other relevant features. The extension listens for the output messages from the Native Messaging Host Application and processes them accordingly. Native Messaging has a Connect API to launch the application and a Disconnect event to let the web page know about termination of the native host application. Using Connect and Disconnect, the life cycle of the native host application can be controlled.

2.2 – Firefox Browser

Firefox supports access to local devices (signature pads) using JS-Ctypes framework. The JS-Ctypes framework allows Firefox extensions to invoke native C/C++ libraries.

JS-Ctypes Library

The JS-Ctype library is the core component of the SDK and acts as a bridge between the browser and the actual signature capture device. It will have all the methods implemented as required by the Firefox JS-Ctypes API framework. The library takes all the input data as JSON string messages (same as Chrome NMH input messages) from the Firefox browser and executes the request synchronously, and when the task is complete sends back the status method return parameter.

Firefox Extension/Web Page

Extensions are the HTML, JavaScript, and CSS based applications. The extensions use JavaScript APIs to launch and communicate with the Topaz JS-Ctypes library for signature capture and other relevant features. The extension polls for the signature and widget events from the library and processes them accordingly.

The host application will host all the functions for capture of signatures and creation of custom screens on the ePad-vision device. Firefox runs this application in the same process.

3.0– Key Features

The ePad-vision SDK supports the following features:

- Open and close connection with the ePad-vision device.
- Creation of custom screens.
- Create widgets on screen (Text, Radio button, Check button, and Push buttons).
- Report signature points and widget events (like widget clicked, state changed).
- Creation of PIN pad for password and PIN capture.

4.0 – Operating Systems and Browsers Supported

The ePad-vision SDK can be integrated into client web applications running on

Operating Systems:
- Windows: Windows 7 and up, Windows 8 and up tablets
- Linux: Ubuntu 12, Fedora 20 and above

Browsers:
- Chrome browsers version 29 and above (both 32 bit and 64 bit).
- Firefox browser 30 and above (32 bit only)

The samples have been thoroughly tested in the version mentioned above of Chrome and Firefox browsers. Hence, it is recommended that you install the latest version of the Chrome and Firefox browser.

Device Drivers
- Windows: Universal Installer 11.6 or later 32 bit or 64 bit based on browser type (32 or 64 bit)
- Linux: ePad 2.3 32bit version.

5.0 – Installation of ePad-vision SDK

5.1– Chrome

Download the Chrome signature capture SDK for Windows at:
www.epadsupport.com/getlatest/ePad-visionChromeSDK.msi.

Download the Chrome signature capture SDK for Linux at:
www.epadsupport.com/getlatest/ePadvision_SDKPkg_Linux_ch.gz.

- For Windows, run the ePad-visionChromeSDK.msi file to install the SDK.

  The default install directory will be at c:\ePadLink_SDK\ePad-visionChromeSDK where will be referenced as <SDK location> within this document.

- For Linux, download the ePadvision_SDKPkg_Linux_ch.gz file. Use “tar xzvf ePadvision_SDKPkg_Linux_ch.gz” from a terminal. It will uncompress and untar the files in the zip file to the current folder. Use “sudo sh install_chrome.sh” to place the SDK under /usr/share/ePad/Chrome_extension where will be referenced as <SDK location> within this document.

5.2– Firefox

Download the Firefox signature capture SDK for Windows at:
www.epadsupport.com/getlatest/ePad-visionFirefoxSDK.msi.

Download the Firefox signature capture SDK for Linux at:
www.epadsupport.com/getlatest/ePadvision_SDKPkg_Linux_ff.gz.

- For Windows, run the ePad-visionFirefoxSDK.msi file to install the SDK.

  The default install directory will be at C:\ePadLink_SDK\ePad-visionFirefoxSDK where will be referenced as <SDK location> within this document

- For Linux, download the ePadvision_SDKPkg_Linux_ff.gz file. Use “tar xzvf ePadvision_SDKPkg_Linux_ff.gz” from a terminal. It will uncompress and untar the files in the zip file to the current folder. Use “sudo sh install_firefox.sh” to place the SDK under /usr/share/ePad/Firefox_extension where will be referenced as <SDK location> within this document.

6.0 – Components of the ePad-vision SDK

Extensions provided in the SDK are provided as a sample implementation and they can be used as a reference for developing extensions or can be used as is.

6.1 – Chrome

The major component of the ePad-vision Chrome SDK is the Native Messaging Executable.

For Windows:

- X86/Chrome.ePad-visionSDK.exe for 32 bit Chrome running on both 32 and 64 bit OS.
- X64/Chrome.ePad-visionSDK.exe for 64 bit Chrome Running on Windows 64 bit OS.

For Linux:

- ePadvisionNMHost.exe

6.2 – Firefox

The major component of the ePad-vision Firefox SDK is the Native library. The PinPadWork, an exe, is for the Linux platform only, will be executed by the libFireFoxExtension.so.

For Windows

- FireFoxExtension.dll

For Linux

- libFireFoxExtension.so
- PinPadWork, an executable that will be launched by the libFireFoxExtension.so

Note: Microsoft component .Net Framework 3.5 should be available in the end user machine for the SDK to work properly in the Chrome and Firefox Browser.

7.0 – Chrome Integration

The Native Messaging Host (NMH) is the critical component of the ePad-vision SDK and Chrome allows access to NMH in web pages via extensions. Integration into a web page requires a background extension that acts as a bridge between the web page and the native messaging host application. The following sections explain installation of NMH, writing extensions for NMH access, installation of extension, and API access:

7.1 – Creating an Extension

The first step of the extension is to write the manifest file. Below is the sample manifest.json file of the ChromeExtension of the SDK.

```
{
  "name": "ePad-vision SDK Chrome Sample Extension",
  "version": "1.0",
  "manifest_version": 2,
  "background": {
      "scripts": ["background.js"],
      "persistent": false
  },
  "content_scripts": [{
      "matches": ["http://*/*", “<all_urls>"],
      "js": ["content.js"]
  }],
  "permissions" : ["nativeMessaging","activeTab","tabs","<all_urls>"]

}
```

Parameter	Description
Name	Name of the extension
Version	Version of the extension
Manifest_version	Version of the manifest.
Background	This section specifies the information about the JS file which has the reference of the extension background JS file. This file will have all the code for communication with Chrome Native Messaging Host. Persistent specifies if the extension background js is an event page or not. Specifying false will make the JavaScript page load when the specific event happens. We suggest you to keep this the same.
Content_Scripts	This section has information related to the script file which gets injected into the web page. Content script code registers for an event which web page raises, and when the event is raised, it invokes the background script for NMH communication. The matches attribute specifies the application URLs into which the content script should be injected. Js: content java script file.
Permissions	List of permissions the extension requires for communicating with the device.

Once the manifest file creation is completed, the next step is to create the files referenced in the manifest file:

- Background.js JavaScript file which communicates with NMH for capturing signature and interaction with the ePad-vision device.
- content.js – Content script file, gets injected to web pages matching URLs mentioned in manifest file.

If your implementation of the extension differs from that used in the sample above, you should create your own files (.js, .htm, etc. as appropriate) that provide the details of the files named in the manifest file.

7.2 – Extension Installation

- Open the Chrome browser, type and navigate to the URL chrome://extensions/ or else go to Settings → Extensions.
- Make sure Developer Mode is selected and then click Load unpacked extension button.
- Browse to Extension folder (<SDK location>\Sample\ChromeExtension) where the extension's manifest.json file is located and click OK.
- Confirm and accept the permissions dialog displayed.
- Once the installation is complete, take a note of the ID of the extension. This should be copied into the NMH manifest file.

7.3 – Native Messaging Host Installation

The first step of the installation is to register the native messaging host on the client/developer desktop. For registering a native messaging host, the application must install a manifest file that defines the native messaging host configuration. Below is the default com.topaz.epadvision.win.json, distributed along with the SDK and it can be used. Some of the parameters of the file will be changed based on the target machine.

```
{
  "name": "com.topaz.epadvision.win",
  "description": "Chrome ePad-vision SDK Native Messaging Host",
  "path":"C:\\ePad-visionSDK\\SDK\\Chrome\\Chrome.ePad-visionSDK\\Chrome.ePad-visionSDK.exe",
  "type": "stdio",
  "allowed_origins": [
    "chrome-extension://mcjlhkcioghfmodbaapbhgagdcgehmfg/"
  ]
}
```

Parameter	Description
Name	Name of the native messaging host. Clients pass this string to the Chrome NMH connect call and launch the host. Keep it the same as above.
Description	Short application description. Keep it the same as above.
Path	Path to the native messaging host binary *Chrome.ePad-visionSDK.exe*. This should be changed based on location where the SDK is installed/copied. If the executable and the manifest file are in same location, the name itself will do. If absolute or relative paths are specified, please make sure to enter '\\' as the folder separator. Adjust the paths as needed for Windows and Linux operating systems.
Type	Type of the interface used to communicate with the native messaging host. Currently there is only one possible value for this parameter: stdio. It indicates that Chrome should use stdin and stdout to communicate with the host.
allowed_origins	List of extensions that should have access to the native messaging host. Normally specifies the extensions from which the host can be accessed. Supports specifying multiple extension IDs. The default manifest file shipped with the SDK contains the IDs of the sample extensions that are shipped with the SDK. These should be changed appropriately if custom extensions are developed. Extension ID can be grabbed from Chrome://Extensions page once the extension is deployed. Use the extension ID copied in Extension installation section.

Once the editing of the manifest file is completed, the next step would be to register the host application with the Chrome Browser.

Windows

For 32 bit and 64 bit operating systems running respective Chrome browsers (32 and 64 bit versions) the following registry key should be created. The manifest file can be located anywhere in the file system.

Create a registry key
HKEY_LOCAL_MACHINE\SOFTWARE\Google\Chrome\NativeMessagingHosts\ com.topaz.epadvision.win

For 64 bit operating systems running 32 bit Chrome the following key should be created in the registry.
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Google\Chrome\NativeMessagingHosts\ com.topaz.epadvision.win

Once the key is created, set the default value of that key to the full path of the manifest file.

Now the host application is ready for access from the specified extensions.

For 32 bit Chrome it is recommended to enter the X86 version of manifest file path located under:
<SDK_location>\SDK\ Chrome-NativeMessagingHost\Windows\X86\manifest.json

For 64 bit Chrome it is recommended to enter the X64 version of manifest file path located under:
<SDK_location>\SDK\ Chrome-NativeMessagingHost\Windows\X64\manifest.json

Linux

Below is the Linux version of com.topaz.epadvision.win.json, distributed along with the SDK and it can be used. Some of the parameters of the file will be changed based on the target machine.

```
{
  "name": "com.topaz.epadvision.win",
  "description": "Chrome ePad-vision SDK Native Messaging Host",
“path":"/usr/share/ePad/Chrome_extension/SDK/Chrome_NativeMessagingHost/ePadvisionNMHost.exe",
"type": "stdio",
  "allowed_origins": [
    "chrome-extension://mcjlhkcioghfmodbaapbhgagdcgehmfg/"
  ]
}
```

The path field is set to the location where the ePadvisionNMHost.exe located.

7.4 – Integrating the Native Messaging Host into Applications

The native messaging host can be integrated directly into an extension or web page. Integration into a web page requires a background extension that acts as a bridge between the web page and the native messaging host application.

All the APIs required for launching the host, sending messages to the host, and receiving messages from the host are available in Chrome JavaScript.

Note: The Chrome Native Messaging related JavaScript APIs cannot be accessed directly from a web page. For security reasons, Chrome allows only an extension to access the native messaging hosts.

7.5 – Launching the Native Messaging Host

Within the extension JavaScript, the following code can be used to launch the native messaging host application.

```
    var hostName = "com.topaz.epadvision.win";
    port = chrome.runtime.connectNative(hostName);
   // Register the listener to handle output messages from native messaging host.
   port.onMessage.addListener(on
NativeMessage);
   // Register the listener to handle native messaging host disconnect or closed.
  port.onDisconnect.addListener(onDisconnected);
```

7.6 – ePad-vision Device Access

The ePad-vision SDK for Chrome is implemented as a Native Messaging Host. As the Google Native Messaging API framework mandates using Standard Input and Output streams for communication between the Chrome and the native application, only text data can be exchanged between the applications. The Input messages trigger communication with device, and the input message itself contains all the required data as payload.

The Output message payload contains the status of the transaction and also signature points, widget events, etc.

The Native Messaging framework mandates that the input and output messages should be in JSON format. Native messaging allows you to send multiple attributes as part of the JSON string/message. The format of the JSON message is:
{text: value1, text1: value2}

where ‘text’ and ‘text1’ are the names of the JSON parameters.

The value1and value2 entries are the values in JSON format. These will be used by the native host application to interpret the signature capture input and process it accordingly. There are no practical restrictions on the number of parameters, names of the parameters, or the length of the data any parameter can contain.

Sending input messages to the Native Messaging host.

Chrome API postMessage should be used to send input messages to the Native Messaging Host

```
    message = { "command": 1, "inking": false, "inkRegionx": 0, "inkRegiony": 0, "inkRegionWidth": 0,
"inkRegionHeight": 0, "widgetLayout": confirmXml };
    port.postMessage(message);
```

Receiving output messages

```
   // Register the listener to handle output messages from native messaging host.
   port.onMessage.addListener(onNativeMessage);
// Receiving response from Native Host
(onNativeMessage Event) function
onNativeMessage(message) {
    console.log("onNativeMessage : " + message);
// Write logic to handle the messages received
```

Once an Input message is sent, the Native Messaging Host processes the request and sends back the status asynchronously to the extension by firing the callback function registered.

8.0 – Firefox Integration

JS-Ctypes native library (FireFoxExtension.dll or libFireFoxExtension.so) is the critical core component of the ePad-vision SDK and Firefox allows access to the native library in web pages via extensions.

Integration into a web page requires a background extension that acts as a bridge between the web page and the native messaging host application.

Normally the native library can be packaged and installed along with the extension.

8.1 – Creating an Extension

A Firefox Extension is normally implemented as a JavaScript file and coupled with a few other supporting files qualifies to be an extension for Firefox

Major Components of Extension

- chrome.manifest – Registers the extension with engine.
- install.rdf – Information about the extension
- content/FireFoxJSON.js – core JavaScript file

Refer to SDK's Sample/FirefoxExtension folder for further info on structure and contents of the extension.

Here is a typical directory structure of the extension:

- install.rdf
- chrome.manifest
- \locale\en-us\translations.dtd
- \content\FireFoxJSON.js
- \content\FireFoxJSON.xul
- \default\preferences\prefs.js

- **Install.rdf :**

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:em="http://www.mozilla.org/2004/em-rdf#">
 <Description about="urn:mozilla:install-manifest">
                         <em:id>support@topazsystems.com</em:id>
                         <em:name>Firefox Signature Capture</em:name>
                         <em:version>1.0</em:version>
                         <em:type>2</em:type>
                         <em:creator>Topaz Systems</em:creator>
                         <em:description>Firefox Signature Capture</em:description>
                         <em:homepageURL>http://www.topazsystems.com/</em:homepageURL>
                         <em:unpack>true</em:unpack>
                         <em:targetApplication>
                                          <Description>
                                                  <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
                                                  <em:minVersion>1.5</em:minVersion>
                                                  <em:maxVersion>4.0.*</em:maxVersion>
                                          </Description>
                         </em:targetApplication>
             </Description>
</RDF>
```

Parameter	Description
em:id	Email Id of the developer
em:name	Extension name.
em:version	Extension version.
em:type	"2" declares that it is installing as an extension.
em:creator	Extension creator
em:description	Extension description.
em:homePageURL	URL of the company providing extension.
em:unpack	Always true.
Firefox application ID	**{ec8030f7-c20a-464f-9b0e-13a3a9e97384}**
em:minVersion	Minimum version of the Firefox that supports this extension, e.g. 1.5
em:maxVersion	Maximum or recent version of Firefox that supports this extension, e.g. 4.0

- **Chrome.manifest**
  Meta information about the extension

```
content FirefoxExtension content/

overlay chrome://browser/content/browser.xul   chrome://FirefoxExtension/content/FireFoxJSON.xul

locale FirefoxExtension                   en-US              locale/en-US/

skin FirefoxExtension classic/1.0 skin/
```

- **Extension.xul**
  Has information about the core js of the Extension

```
<?xml version="1.0"?>

<!DOCTYPE JSON SYSTEM "chrome://JSON/locale/overlay.dtd">

<overlay id="FireFoxJSON" xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

<script src="FireFoxJSON.js"/>

 </overlay>
```

8.2 – Extension Installation

Extensions are packaged and distributed in ZIP files with the XPI file extension.

Zip all the files in the folder.

Add .xpi extension for the zipped folder.

1. Open the Mozilla Firefox browser and navigate to Tools > Add-ons > Extension tab.
2. Click on “Tools for all add-ons” and select “install Add-on from file…” option.
3. Browse for .xpi file or .zip file and click on Install Now button.
4. After installation, restart the browser.

8.3 – Invocation of Native Library from Extension

The core component of the ePad-vision SDK is developed as a native library and Firefox uses JS- Ctypes framework for invocation and marshalling and un-marshalling data between the JavaScript and native library.

The first step is to import the JS-Ctypes and other required JavaScript modules (jsm) files. Then define the path of the native library. As the library is installed along with the extension, it can be referenced (relative to the extension) as shown below.

Load the library.

```
Components.utils.import("resource://gre/modules/ctypes.jsm");
Components.utils.import("resource://gre/modules/Services.jsm");
//Components.classes["@mozilla.org/net/osfileconstantsservice;1"].getService(Components.interfaces.nsIOS
Fi leConstantsService).init();
const consoleJSM
=Components.utils.import("resource://gre/modules/devtools/Console.jsm",{}); var profileDir
=
Components.classes["@mozilla.org/file/directory_service;1"].getService(Components.interfaces.nsIProperties
). get("ProfD", Components.interfaces.nsIFile).path ;
var mylibPath_win = profileDir +
"/extensions/support@topazsystems.com/FireFoxExtension.dll"; var mylibPath_linux =
profileDir + "/extensions/support@topazsystems.com/libFireFoxExtension.so"; var
mylib=null;
//compare the OS and then load the
respective library console.log("appversion="
+ navigator.appVersion);
   if (
     navigator.appVersion.indexOf("Win"
     )!=-1) {
     mylib=ctypes.open(mylibPath_win);
   } else if ( navigator.appVersion.indexOf("X11")!=-1 ||
     navigator.appVersion.indexOf("Linux")!=-1) { mylib=ctypes.open(mylibPath_linux)
   }
```

Declare the functions of the native library for use within JavaScript

```
var send=null;
var
signatureFunc
=null; var
widgetFunc=nu
ll;
//Declare the main function for interaction with the device.
send = mylib.declare("JsonDeserialize",ctypes.default_abi,ctypes.char.ptr, ctypes.char.ptr);
//Declare Signature data polling function
signatureFunc =
 mylib.declare("SignatureData",ctypes.default_abi,ctypes.char.ptr); widgetFunc =
 mylib.declare("WidgetData",ctypes.default_abi,ctypes.char.ptr);
```

Once the functions are declared, you can invoke these functions.

8.4 – Invoking the Extension from Web Page

Normally Firefox extension gets loaded when the Firefox browser gets started.

Extension register for a custom HTML DOM event which the web page raises when signature needs to be captured. Extension implements the code for ePad-vision device access within this custom event listener.

```
document.addEventListener("TopazMozillaSignStartEvent", TopazMozillaSignStart, false, true);
```

The web page uses the following code to raise the custom DOM event and start signature capture.

```
// Create a dummy HTML Element
var element = document.createElement("JSONParseElement");
//Append the element to the document
object
document.documentElement.appendChild(
element); function
TopazMozillaSignStart()
{
    var evt =
    document.createEvent("Events");
    evt.initEvent("TopazMozillaSignStartEv
    ent", true, false);
    element.dispatchEvent(evt);
}
// Raise the custom event when user clicks on button e.g. sign button/link
document.getElementById('sign').addEventListener('click', TopazMozillaSignStart);
```

8.5 – ePad-vision Device Access

ePad-vision Firefox native libraries **JsonDeserialize** function is invoked for accessing different features of the SDK. The same function should be invoked with different input messages for accessing various features like OpenConnection, CloseConnection, CreateScreen (without Signature capture), CreateScreen (with Signature capture), PinPad screen and Refresh screen.

The function takes the input data as a JSON message. Though regular strings and other regular data types can be used, JSON format is used to make the Firefox implementation consistent with Chrome APIs.

The format of the JSON message is
{text: value1, text1: value2}

where ‘text’ and ‘text1’ are the names of the JSON parameters.

The value1 and value2 entries are the values in JSON format. These will be processed by the library and interact with the ePad-vision device for signature capture and screen creation.

```
var cmdOpenConn = {"command": 1 , "inking": false , "inkRegionx": 0 , "inkRegiony": 0, "inkRegionWidth": 0 ,
"inkRegionHeight":0, "widgetLayout": ''};
var
cmdOpenConnObj=JSON.stringify(cmdOpenCo
```

The return value from the JsonDeserialize function is again a JSON message which is the output message described in subsequent ePad-vision SDK API section. In case of Chrome the output message is sent as a callback to the registered function asynchronously.

9.0 – ePad-vision SDK API

Following section described the interface call in detail.

9.1 – Open Connection

Opens the connection with the device.

Input Message

```
{"command": 1 , "inking": false , "inkRegionx": 0 , "inkRegiony": 0, "inkRegionWidth": 0 ,
"inkRegionHeight": 0, "widgetLayout": "" }
```

The only parameter required is command number and all other parameters can be defaulted as shown below.

Parameter	Description
Command	Number identifying the command. 1 for open connection.
Inking	False
inkRegionx	0
inkRegiony	0
inkRegionWidth	0
inkRegionHeight	0
widgetLayout	Empty String

Output Message

```
{"command":1,"status":False,"message":"Could not find a signature pad."}
```

Parameter	Description
Command	Number identifying the command. 1 for open connection.
Status	True if connection is successful, false otherwise.
Message	Empty in case connection is successfully opened, message in case connection fails.

9.2 – Close Connection

Closes the connection with the device Input Message

```
{"command": 2 , "inking": false , "inkRegionx": 0 , "inkRegiony": 0, "inkRegionWidth": 0 ,
"inkRegionHeight": 0, "widgetLayout": "" }
```

The only parameter required is the command number, and all other parameters can be defaulted as shown below.

Parameter	**Description**
Command	Number identifying the command. 2 for close connection.
Inking	False
inkRegionx	0
inkRegiony	0
inkRegionWidth	0
inkRegionHeight	0
widgetLayout	Empty String

Output Message

```
{"command":2,"status":False,"message":"Could not find a signature pad."}
```

Parameter	**Description**
Command	Number identifying the command. 2 for close connection.
Status	True if connection is successfully closed, false otherwise.
Message	Empty in case connection is successfully closed message in case connection close fails.

9.3 – Create Screen (No Signature)

Creates a screen on the ePad-vision device. Should be called after opening a connection

Input Message

```
{"command": 3 , "inking": false , "inkRegionx": 0 , "inkRegiony": 0, "inkRegionWidth": 0 ,
"inkRegionHeight":0, "widgetLayout": "XML String containing widget information" }
```

Parameter	Description
Command	Number identifying the command. 3 for screen creation.
Inking	False
inkRegionx	0
inkRegiony	0
inkRegionWidth	0
inkRegionHeight	0
widgetLayout	XML String containing widgets to be created on ePad vision screen.

Here is a sample XML (Sample application of SDK uses this for widgetLayout parameter) for creating a screen with widgets, which has text widgets, push buttons, and radio buttons. Each widget can specify the ID, type, font details, location and size on the device etc.

"<WidgetLayout><Widget ID=\"11\" Type=\"TEXT\" FontName=\"SAN_B_24\" BGColor=\"0000FF\" FGColor=\"ffd700\" Width=\"450\" Height=\"100\" X=\"10\" Y=\"10\" Effect=\"NONE\" Text=\"In this transaction, you have the option to capture your signature electronically or enter your pin or opt-out. Please make your selection below.\" /><Widget ID=\"12\" Type=\"RADIOBUTTON\" FontName=\"SAN_B_16\" BGColor=\"0000FF\" FGColor=\"ffd700\" Width=\"320\" Height=\"34\" X=\"50\" Y=\"110\" Effect=\"NONE\" Text=\"Continue with signing process electronically\" GrpID=\"1\" /><Widget ID=\"13\" Type=\"RADIOBUTTON\" FontName=\"SAN_B_16\"BGColor=\"0000FF\" FGColor=\"ffd700\" Width=\"170\" Height=\"34\" X=\"50\" Y=\"140\" Effect=\"NONE\" Text=\"Entering pin number\" GrpID=\"1\" /><Widget ID=\"14\" Type=\"RADIOBUTTON\" FontName=\"SAN_B_16\" BGColor=\"0000FF\" FGColor=\"ffd700\" Width=\"150\" Height=\"34\" X=\"50\" Y=\"170\" Effect=\"NONE\" Text=\"Opt-out\" GrpID=\"1\" /><Widget ID=\"15\" Type=\"BUTTON\" FontName=\"SAN_B_16\" BGColor=\"00FF00\" FGColor=\"ffd700\" Width=\"50\" Height=\"50\" X=\"200\" Y=\"220\" Effect=\"ThreeD\" Text=\"OK\" GrpID =\"2\" /></WidgetLayout>"

For more information on XML refer to ePad-vision device driver documentation.

Output Message

```
{"command":3, status":false,”message”:” widget creation failed”}
```

Parameter	Description
Command	Number identifying the command. 3 for screen creation.
Status	True if command executes successfully, false otherwise.
Message	Empty in case create screen successful, message in case screen creation fails.

The output message specifies if screen creation is successful or not. User selections or responses (Widget Events) will be sent back as a separate output message (event), which is explained in detailed in sections below.

9.4 – Create Screen (Signature Capture)

Creates a screen on the ePad-vision enabling users to scribble signatures on the device. Should be called after opening a connection. For capturing signatures, inking must be enabled and then the region on the device on which signature will be captured should be specified.

Input Message

```
{"command": 3 , "inking": true , "inkRegionx": 0 , "inkRegiony": 0,
"inkRegionWidth": 480 , "inkRegionHeight":200, "widgetLayout": "XML String
containing widget information" }
```

Parameter	Description
Command	Number identifying the command.3 for screen creation
Inking	True
inkRegionx	0
inkRegiony	0
inkRegionWidth	480 width of the ink capture region from top left
inkRegionHeight	200 height of the ink capture region from top left
widgetLayout	XML String containing widgets to be created on ePad vision screen.

Here is a sample XML (Sample application of SDK uses this for widgetLayout parameter) for creating a screen with button widgets, to accept, clear, and cancel the signatures.

```
"<WidgetLayout><Widget ID=\"2\" Type=\"BUTTON\" FontName=\"SAN_B_16\" BGColor=\"00FFFF\"
FGColor=\"000000\" Width=\"80\" Height=\"34\" X=\"80\" Y=\"220\" Effect=\"ThreeD\"
Text=\"Accept\"/><Widget ID=\"3\" Type=\"BUTTON\" FontName=\"SAN_B_16\" BGColor=\"00FFFF\"
FGColor=\"000000\" Width=\"80\" Height=\"34\" X=\"200\" Y=\"220\" Effect=\"ThreeD\"
Text=\"Clear\"/><Widget ID=\"4\" Type=\"BUTTON\" FontName=\"SAN_B_16\" BGColor=\"00FFFF\"
FGColor=\"000000\" Width=\"80\" Height=\"34\" X=\"320\" Y=\"220\" Effect=\"ThreeD\"
Text=\"Cancel\"/></WidgetLayout>"
```

For more information on XML refer to ePad-vision device driver documentation.

Output Message

```
{"command":3, status":false,”message”:” Screen creation failed”}
```

Parameter	Description
Command	Number identifying the command. 3 for open connection.
Status	True if command successfully false otherwise.
Message	Empty in case create screen successful, message in case screen creation fails.

9.5 – PinPad Screen

Display PinPad screen on ePad, to capture password or PIN key.

Input Message

```
{"command": 4 , "inking": false , "inkRegionx": 0 , "inkRegiony": 0, "inkRegionWidth": 0 ,
"inkRegionHeight":0, "widgetLayout": "XML String containing pinpad information" }
```

Parameter	Description
Command	Number identifying the command.4 for PinPad screen.
Inking	False
inkRegionx	0
inkRegiony	0
inkRegionWidth	0
inkRegionHeight	0
widgetLayout	XML String containing pinpad widgets to be created on ePad vision screen.

Here is a sample XML (Sample application of SDK uses this for widgetLayout parameter) for creating a screen for pinpad screen.

*"<PinPad><Widget ID=\"NUM_1\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"275\" Y=\"2\" Effect=\"ThreeD\" /><Widget ID=\"NUM_2\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"340\" Y=\"2\" Effect=\"ThreeD\" /><Widget ID=\"NUM_3\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"405\" Y=\"2\" Effect=\"ThreeD\" /><Widget ID=\"NUM_4\"FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"275\" Y=\"74\" Effect=\"ThreeD\" /><Widget ID=\"NUM_5\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"340\" Y=\"74\" Effect=\"ThreeD\" /><Widget ID=\"NUM_6\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"405\" Y=\"74\" Effect=\"ThreeD\" /><Widget ID=\"NUM_7\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"275\" Y=\"146\" Effect=\"ThreeD\" /><Widget ID=\"NUM_8\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"340\" Y=\"146\" Effect=\"ThreeD\" /><Widget ID=\"NUM_9\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"70\" X=\"405\" Y=\"146\" Effect=\"ThreeD\" /><Widget ID=\"NUM_0\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"60\" Height=\"60\" X=\"275\" Y=\"218\" Effect=\"ThreeD\" /><Widget ID=\"FUN_BACKSPACE\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"125\" Height=\"60\" X=\"340\" Y=\"218\" Effect=\"ThreeD\" Text=\"Backspace\"
/><Widget ID=\"CAPTION\" FontName=\"SAN_R_16\" BGColor=\"000000\" FGColor=\"FFFF00\" Width=\"240\" Height=\"40\" X=\"10\" Y=\"5\" Effect=\"None\" Text=\"Use the pad on the right to enter your pin.\" /><Widget ID=\"CONTENT\" FontName=\"SAN_B_20\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"220\" Height=\"30\" X=\"10\" Y=\"60\" Effect=\"ThreeD\" /><Widget ID=\"FUN_SEC\" FontName=\"SAN_R_20\" BGColor=\"9C9C9C\" FGColor=\"FF0000\" Width=\"220\" Height=\"20\" X=\"10\" Y=\"100\" Effect=\"ThreeD\" Text=\"Hide the text\" /><Widget ID=\"FUN_OK\"*

FontName=\"COM_B_24\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"100\" Height=\"60\" X=\"10\" Y=\"150\" Effect=\"ThreeD\" Text=\"OK\"/><Widget ID=\"FUN_CANCEL\" FontName=\"COM_B_24\" BGColor=\"0000FF\" FGColor=\"000000\" Width=\"100\" Height=\"60\" X=\"120\" Y=\"150\" Effect=\"ThreeD\" Text=\"Cancel\"/></PinPad>";

For more information on XML refer to ePad-vision device driver documentation. The default timeout period is 40 seconds.

Output Message

```
{"command":4, status":false,”message”:” Screen creation failed”}
```

Parameter	Description
Command	Number identifying the command. 4 for PinPad screen.
Status	True if command successfully and user input within timeout period, false otherwise.
Message	Password or PinPad string in case operation successful, message in case pinpad operation fails.

Note: PinPad screen will not receive individual widget event though.

9.6 – Refresh Screen

Refreshes the screen on ePad. Normally used to clear signatures.

Input Message

```
{ "command": 5, "inking": false, "inkRegionx": 0, "inkRegiony": 0, "inkRegionWidth": 0,
"inkRegionHeight": 0, "widgetLayout": "" }
```

Parameter	Description
Command	Number identifying the command. 5 for screen refresh
Inking	False
inkRegionx	0
inkRegiony	0
inkRegionWidth	0
inkRegionHeight	0
widgetLayout	Empty String

Output Message

```
{"command":5,"status":false,"message":"Failed to refresh the screen"}
```

Parameter	Description
Command	Number identifying the command. 5 for screen refresh.
Status	True if refresh is successful, false otherwise.
Message	Empty in case refresh is successfully executed, message in case refresh fails.

9.7 – Signature Event Polling (Firefox Only)

All the signature points scribbled on the device should be polled from the native library. Polling should be started right after the signature capture screen is created on the screen.

SignatureEvent polling function should be called at repeated intervals to query the signature data. Refer to the sample extension for reference implementation of the polling.

Input: None

Return Value: Signature point data as a comma separated values. Below is the format

```
X,Y,Pressure,TimeStamp
```

Parameter	Description
X	Point x as number
Y	Point y as number
Pressure	Pressure as number
TimeStamp	TimeStamp as long

9.8 – Widget Event Polling (Firefox Only)

The widget events should be polled from the native library. Polling should be started right after the signature capture screen is created on the device screen.

WidgetEvent polling function should be called at repeated intervals to query the signature data. Refer to the sample extension for reference implementation of the polling.

Input: None

Return Value: Widget details as a comma separated values. Below is the format

```
Widgetid,widgetype,widgeteventcode
```

Parameter	Description
Widgetid	ID of the widget as a number (same as specified in the widget layout XML while creating the screen).
Widgettype	Widgettype number (button, radio button, check box etc.) Value – Member Name – Description 1 – Text – Text Widget 2 – Button – Push Button Widget 3 – Checkbox – Checkbox Widget 4 – Radiobutton – Radio Button Widget
Widgeteventcode	Reported widget event code as a number Value – Member Name – Description 1 – CREATE_SUCCEED – The widget has been created. 2 – DELETE_DONE – The widget has been deleted. 3 – CHECKED – The widget has been checked (checkbox, radio button). 4 – UNCHECKED – The widget has been unchecked (checkbox, radio button). 5 – CLICKED – The button has been clicked. 8 – ENABLED – The widget has been enabled. 9 – DISABLED – The widget has been disabled. 10 – SHOW – The widget has been displayed. 11 – HIDE – The widget has been hidden. 12 – TEXT_DONE – The widget text has been changed. 150 – INVALID_ID – The received widget ID is invalid. 151 – INVALID_TYPE – Invalid widget type 152 – INVALID REQUEST – Invalid request (for example, check/uncheck text widget) 154 – CREATE_FAILED – Failed to create widget

9.9 – Signature Event (Chrome Only)

All the signature points scribbled on the device are reported back as an output message. Here is the format of the message.

No input message, signature event data is sent as an output message.

Output Message

```
{"command":101," xAxis": 100,"yAxis":100, "pressure ":10000 , "timeStamp ":111111111 }
```

Parameter	Description
Command	Number identifying the command. 101 for signature event
xAxis	Point x as number
yAxis	Point y as number
Pressure	Pressure as number
Timestamp	Timestamp as long.

9.10 – Widget Event (Chrome Only)

Widget events are reported as output message.

No input message, event data is sent as an output message.

Output Message

```
{"command":102," widgetID":1," widgetType":2, "eventCode":1 }
```

Parameter	Description
Command	Number identifying the command. 102 for widget event
Widgetid	Event source widget ID as number
Widgettype	Type of widget as number (button, radio button or check box) etc.
Eventcode	Event type as number (widget created, deleted, clicked or status changed) etc.

Detail description for widgettype and EventCode see section 9.8

10.0 – Signature Drawing

The signature points reported from the device can be drawn on an HTML 5 Canvas object. Refer to the sample extension for drawing the signature points and scaling of the signature.

11.0 – Instructions to Run Sample Application

Before running the sample web page from the SDK/Sample folder one has to install the Chrome Native Messaging Host and Chrome Extension for the Chrome browser and the Firefox Extension for Firefox Follow the quick instructions to install these.

11.1 – Chrome Extension Installation and Configuration

Windows and Linux

1. Open the Chrome browser, type and navigate to the URL chrome://extensions/ or click “Settings → Extensions”.
2. Click on LoadUnpackedExtension and choose Extension folder (<SDK location>\Sample\ChromeExtension). Confirm the installation by clicking on the OK button in browse dialog.
3. Confirm and accept the permission dialog displayed.
4. Make a note of the extension ID displayed in the extension window.

11.2 – Chrome Native Host Installation and Configuration

Windows

1. Navigate to the Sample/SDK/Chrome-NativeMessagingHost/Windows/X86 or Sample/SDK/Chrome-NativeMessagingHost/Windows/X64 folder of the SDK based on whether the Chrome is 32 bit or 64 bit
2. Edit the manifest.json file in any text editor (<SDK location>\SDK\Chrome-NativeMessagingHost \Windows/X86 or X64) and update the Path attribute value pointing to the physical path of the Chrome.ePad-visionSDK.exe file (same as this manifest file <SDK location>\SDK \ Chrome-NativeMessagingHost \ Windows\ Chrome.ePad-visionSDK.exe). Please make sure to enter '\\' -- double back slash as folder and file separator. As the manifest and executable are in the same location then there is no need to modify this entry.
3. Update the chrome-extension value with the ID captured in step 4 of the Extension installation. \\extension-ID\
4. Open Registry Editor (regedit.exe) to install the native messaging host with Chrome. Navigate to the following key HKEY_LOCAL_MACHINE\SOFTWARE\Google\Chrome\ (create the keys, if any of them are missing).

   For 64 bit machines running 32 bit Chrome these should be under HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Google\Chrome (if any of these are missing, you can create these keys).
5. Create a new key named 'NativeMessagingHosts'.
6. Again, create a new key named com.topaz.epadvision.win
7. In the default value enter the name of the manifest.json file of the Native Messaging host (typically <SDK location>\SDK \ Chrome-NativeMessagingHost \Windows\X86 or X64\manifest.json)

Linux

1. Navigate to the /etc/opt/chrome/native-messaging-hosts folder of the system.
2. Edit the com.topaz.epadvision.win.json file in any text *editor* and update the Path attribute value pointing to the physical path of the ePadvisionNMHost.exe file (same as this com.topaz.epadvision.win.json file <SDK location>/SDK/Chrome-NativeMessagingHost/ Linux/ePadvisionNMHost.exe).
3. Update the chrome-extension value with the ID captured in step 4 of the Extension installation. \\extension-ID\

11.3 – Firefox Extension Installation

Windows and Linux

- Open the Mozilla Firefox browser and navigate to Tools > Add-ons > Extension tab.
- Click on “Tools for all add-ons” and select “install Add-on from file…” option.
- Browse to FirefoxExtension.zip file (<SDK location>\Sample\FirefoxExtension) and click on Install Now button.
- After installation, restart the browser.

11.4 – Running Web Page Sample

Navigate to the <SDK location>\Sample folder and open the main.html page, then press the Start button to run the program.

11.5 – Sample Source Code

Source code for the Sample Chrome Extension, sample Firefox extension, and sample web page are located in the <SDK Location>\Sample folder.