Active Page Item Developer - Reference Manual

By Kris Coppieters

1. Introduction

Active Page Item Developer is a toolset for Adobe® InDesign® CS, CS2, CS3 and CS4 which assist the development of JavaScript-based solutions within InDesign.

If you're only interested in compiling stand-alone scripts, you can skip most of this reference manual – but you want to look at the chapter
'6. Licensing your code'

The first major feature is that it extends the InDesign JavaScript programming model with event-driven programming.

A second major feature is that it helps you in protecting your scripts from inspection and pirating. If you so desire, you can 'lock' the results of your hard work and only allow them to be run for a predetermined demo trial period, or only when properly licensed.

This manual was last updated when 1.0.47 was the current version of the *APID ToolAssistant* plug-in.

2. System Requirements

Active Page Item Developer is available for Mac OS X and Windows, and for InDesign CS, CS2, CS3 and CS4. It has been tested against InDesign CS 3.0.1 or higher, against InDesign CS2 4.0.5 or higher, against InDesign CS3 5.0.4 or higher, and against InDesign CS4 6.0.3 or higher.

3. Properties and methods added to the ExtendScript DOM by Active Page Items

Remark: In this text, the words 'script tag', 'script label' and 'label' are used interchangeably - they refer to the same thing.

Active Page Item Developer extends the lists of properties and methods for the *PageItem*, *Application*, and *Document* object classes in the InDesign DOM.

The information below (properties and methods) should be seen as an extension to the information provided in the InDesign JavaScript/VBScript/AppleScript documentation available from Adobe Systems.

Active Page Items is primarily meant for use with ExtendScript, but many of the methods and properties listed below are also usable and accessible from VBScript and AppleScript.

3.1. Additional PageItem, Group, Guide and Story Properties

Property	Type	Access	Description
componentId	String	write-once	This is a special string that allows you to 'lock'

			the script info attached to a page item or inside an .spln file against inspection. Most Active Page Item properties become inaccessible and encrypted, unless accessed from a handler attached to another page item that shares a compatible `componentId` string. See the sections on ‘Predefined keys’ and ‘Component Ids and Licensing’ for more information. REMARK: You cannot debug the code of a compiled script if the `componentId` of a page item or scripted plug-in is not the empty string. In InDesign CS and CS2, enabling the debugger will disable any scripted plug-ins (this to avoid code inspection through the debugger).
contextMenu	Array of Array (pairs of strings)	r/w	Defines a context menu for this page item; it consists of an array of arrays. This array contains zero or more 4-element arrays. The individual 4-element arrays should each contain 2 strings, a floating point positioning value, and a Boolean. It is allowed to omit the third and fourth elements (effectively reducing the element to a 2- or 3-element array), in which case the omitted floating point value defaults to 10.0 and the omitted Boolean defaults to `false`. The first string is the menu item text that should be displayed in the context menu. The second string is an event code. The third value is a floating point value that helps positioning the menu items. Higher values end up further down the menu, lower values end up more up the menu. The fourth, Boolean item defines whether this context menu should be applied to a text selection or to the frame. Setting the Boolean to `true` makes the context menu item appear when the frame is context-clicked while it is in text selection mode. When the user brings up the context menu and selects it, the attached JavaScript will be called with `theItem.eventCode` set to the event code associated with the menu item in the corresponding pair. Context menu entries with event codes that start with a `subject…` prefix are automatically associated with all observed subjects for this page item. Any other event codes are associated with this page item itself.
dataStore	variable	r/w	This is a ‘free’ data storage zone that allows you to attach some data ‘in’ the page element. Most

			times when a top-level JavaScript is called for event processing, it starts out with a new execution context - any regular JavaScript global variables that are defined disappear once the event is processed. By using this attribute you can store some data away so that it remains persistent between consecutive non-nested event processing calls. This data is saved with the document, so it remains available even when the document is closed and re-opened. Because of issues with the CORBA glue code, this property is not supported on the InDesign Server version of APID – use *setDataStore/getDataStore* instead.
demoDaysLeft	Integer	read-only	If the currently executing JavaScript needs licensing and is being used in demo mode, this property contains the number of demo days left before execution becomes disabled.
displayErrorDialogs	Boolean	r/w	Whether to bring up error dialogs (e.g. for syntax errors) during the execution of the script attached to this page item.
eventCode	String	read-only	The event currently being handled
eventData	String	read-only	Some optional additional data that might be passed along with the event. For predefined events (like *docLoaded*, *subjectSelected*,…) the *eventData* is an object, currently with only a single attribute: - *nestedInScript* tells you whether this event handler is being called while another script is currently active. This allows checking for issues like *docLoaded* handlers being launched by a script that executes a call to *app.open()*.
eventFilter	String	r/w	String specifying which events are captured by this page item. See "Event Filter Expressions". Shown on the *Active Page Item Developer* palette as *Event Filter*.
eventSource	PageItem or ScriptedPlugin	read-only	When the JavaScript is executed, this property contains a reference to the object that caused the event. Typically you'd refer to `theItem.eventSource` to access the object causing the event
isDemo	Boolean	read-only	Returns `true` when the currently executing JavaScript needs licensing and is being used in demo mode.
javaScript	String	r/w	JavaScript attached to the page item. This JavaScript can use the predefined variable `theItem` to refer to the page item whose script is being executed.
nestedInScript	Boolean	Read-only	Returns `true` when the currently executing

			JavaScript handler is called (directly or indirectly) from a regular user script (e.g. stored in the Presets/Scripts directory).
observers	Array of PageItem	read-only	All page items that are currently observing this page item by means of their `subjectScriptTagFilter`. This property looks at the current page item as being a subject to one or more other page items.
subjects	Array of PageItem	read-only	This contains an array with all page items that are part of the selection determined by the current `subjectScriptTagFilter`. This property looks at the current page item as being the observer of one or more other page items.
subjectScriptTagFilter	String	r/w	String specifying the labels of the page items that are to be observed by this page item. On the *Active Page Item Developer* palette this is shown as *List of Subjects*.
tempDataStore	variable	r/w	This is a 'free' data storage zone that allows you to store some data 'in' the page element. Often when a top level JavaScript is called for event processing, it starts out with a new execution context – any regular JavaScript global variables that were defined by your script might disappear once the event is processed. (On the other hand, global variables defined during the processing of nested events tend to persist in the nested event handlers). By using this attribute you can store some data away so that it is persistent between different (non-nested) event processing calls. This data is **not** saved when the document is closed. Because of issues with the CORBA glue code, this property is not supported on the InDesign Server version – use *setDataStore/getDataStore* instead.
triggeredFromScript	Boolean	read-only	Is true if the current event was caused by some scripted operation. However, this does not apply to the *modified-recomposed…* and *subjectModified-recomposed…* events: these are asynchronous, and by the time they are captured by APID, there is no good way for APID to be able to tell what caused them (scripted operation vs. user interaction).
useDebugger	Boolean	r/w	Currently InDesign CS2/CS3 only. When this is set to *true* then the script will set up a debugger session each time it catches an event.

3.2. Additional PageItem, Group, Guide and Story Methods

Method	Returns	Description

<table>
<tr><td><i>callExtension</i></td><td>variable</td><td>Generic extension mechanism that can be used by InDesign SDK/C++ developers. Calls through to a C++ implementation aggregated onto the page item’s boss class. This is an easy way to extend JavaScript with C++ code.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td><i>interfaceID</i></td><td>An integer – the interface ID of the interface we’re calling from JavaScript. This interface needs to be aggregated onto the page item’s boss class.</td></tr>
<tr><td>[<i>parameter1</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter2</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter3</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter4</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter5</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter6</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter7</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter8</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter9</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter10</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter11</i>]</td><td>An optional parameter</td></tr>
</table>
The underlying implementation can return a return value. See the cookbook for a practical example.</td></tr>
<tr><td><i>createProgressBar</i></td><td>-</td><td>Deprecated. Use <i>Application.createProgressBar()</i> instead</td></tr>
<tr><td><i>getDataStore</i></td><td>variable</td><td>Enhanced form of the <code>dataStore</code> and <code>tempDataStore</code> properties – also see <code>setDataStore</code>. This method retrieves data from one of two associative arrays.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td><i>key</i></td><td>String</td></tr>
<tr><td>[<i>useTempDataStore</i>]</td><td>Boolean</td></tr>
</table>
<p>Using an empty key ("") accesses the same data item as the <code>dataStore</code> or <code>tempDataStore</code> properties.</p>
<p>If the optional <code>useTempDataStore</code> parameter is omitted, it defaults to <code>false</code>.</p>
<p>If <code>useTempDataStore</code> is true, this method accesses a set of data items that are not saved with the document.</p>
<p>If <code>useTempDataStore</code> is false, this method accesses another set of data items that are saved along with the document.</p>
<p>Keys that start and end with a dollar sign are reserved for internal use (see further).</p></td></tr>
<tr><td><i>handleScriptEvent</i></td><td>Boolean</td><td>Sends an event code and some optional event data to this page element for processing.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
</table></td></tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>
<table>
<tr><td>*eventCode*</td><td>String</td></tr>
<tr><td>[*eventData*]</td><td>String</td></tr>
</table>
There are restrictions here: it is not allowed to call an event handler that has been compiled/protected by a component id, unless the caller is also compiled/protected itself by a compatible component id.
Compatible component id must have the same compilation password (other data fields inside the component id can be different).
This mechanism protects compiled code against unwanted use by ‘outside’ scripts: unless the calling script itself has been protected as well with a compatible component id, no other script can call directly into an event handler attached to a page item.
See the sections on ‘Predefine keys’ and ‘Component Ids and Licensing’ for more information.
</td>
</tr>
<tr>
<td>*labeledPageItems*</td>
<td>Array</td>
<td>
Contains all page elements selected through their label by a certain wildcard expression
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*wildcardExpression*</td><td>String</td></tr>
<tr><td>[*where*]</td><td>Integer</td></tr>
</table>
The optional `where` parameter is a combination of zero or more bitflag-values – you can add together some of the values below to combine them:
<table>
<tr><td>0</td><td>same document (default)</td></tr>
<tr><td>1</td><td>same spread as this page item</td></tr>
<tr><td>2</td><td>same page as this page item</td></tr>
<tr><td>4</td><td>same layer as this page item</td></tr>
<tr><td>8</td><td>same parent as this page item</td></tr>
<tr><td>16</td><td>children of this page item</td></tr>
</table>
</td>
</tr>
<tr>
<td>*registerMenuItem*</td>
<td>-</td>
<td>Deprecated – use *Application.registerMenuItem()* instead</td>
</tr>
<tr>
<td>*saveToPlugin*</td>
<td>-</td>
<td>
Saves this page item’s Active Page Item attributes into a stand-alone *.spln* file, which can be stored in the Plug-Ins folder.
When a document is opened, this *.spln* file is automatically loaded and converted into a ‘pseudo-page-item’ which reacts to events the same way as the original page item: it observes page items, and reacts to events.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*pluginName*</td><td>String. Scripted plug-in files whose name starts with ‘@’ get special treatment.</td></tr>
<tr><td>[*compress*]</td><td>Boolean</td></tr>
</table>
The optional *compress* parameter determines whether the scripted plugin should be reduced in size by removing any redundant text (comments, extra spaces and tabs,…) from the JavaScript source code prior to saving.
Later on, when this plug-in is loaded by a document, it will carry a
</td>
</tr>
</table>

<table>
<tr><td></td><td></td><td>label that is based on the filename used: the *.spln* extension is stripped off and the filename is converted to all lowercase. The resulting string becomes the label used for the instantiated ScriptedPlugin object, which is owned by the Document object that is being opened.
If the saved file has a name that starts with ‘@’ it is treated slightly differently.
Normally, when a user removes or updates an *.spln* file, any existing associated data that was stored in the dataStore of the corresponding *ScriptedPlugin* object is discarded.
However, if the saved plugin file’s name starts with a ‘@’, then the dataStore will survive updating and removal of the *.spln* file.</td></tr>
<tr><td>*setDataStore*</td><td>-</td><td>Enhanced form of the `dataStore` and `tempDataStore` properties - also see `getDataStore`. This method stores data in one of two associative arrays.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*key*</td><td>String</td></tr>
<tr><td>*data*</td><td>Variable</td></tr>
<tr><td>[*useTempDataStore*]</td><td>Boolean</td></tr>
</table>
Using an empty key (“”) accesses the same data item as the `dataStore` or `tempDataStore` properties.
If the optional `useTempDataStore` parameter is ommitted, it defaults to *false*.
If `useTempDataStore` is true, this method accesses a set of data items that are not saved with the document.
If `useTempDataStore` is false, this method accesses another set of data items that are saved along with the document.
Keys that start and end with a dollar sign are reserved for internal use (see ‘Predefined Keys’ a bit further).</td></tr>
<tr><td>*setProgress*</td><td>Boolean</td><td>Deprecated. Use *Application.setProgress()* instead</td></tr>
</table>

3.3. Additional Document methods

<table>
<tr><td>Method</td><td>Returns</td><td>Description</td></tr>
<tr><td>*labeledPageItems*</td><td>Array</td><td>Contains all page elements selected through their label by a certain wildcard expression
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*wildcardExpression*</td><td>String</td></tr>
</table></td></tr>
<tr><td>*loadedScriptedPlugins*</td><td>Array</td><td>Finds all scripted plug-ins instantiated into this document. The single parameter is a wildcard expression – use a single star to get all available scripted plug-ins.
Scripted plug-in labels are determined by the filename of the scripted plug-in: the .spln extension is stripped off, and the name is converted to all lower case.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td></td><td><table><tr><td><i>labelWildcardExpression</i></td><td>A string with a wildcard expression for matching against the scripted plug-in labels.</td></tr></table></td></tr>
<tr><td><i>multiPropertyAssign</i></td><td>Integer</td><td><table><tr><td>Parameter</td><td>Type</td></tr><tr><td><i>assignmentList</i></td><td>Array of 3-element subarrays.</td></tr></table>
<p>Each subarray has three entries: an object reference (Object), an attribute name (String) and a value to be assigned (anything).</p>
<p>All assignments will be carried out in a single undoable operation, which might lead to speed improvements in some situations.</p>
<p>The return value is zero if all assignments were successful, and non-zero if one or more assignments failed.</p></td></tr>
</table>

3.4. Additional Application methods

<table>
<tr><td>Method</td><td>Returns</td><td>Description</td></tr>
<tr><td><i>callExtension</i></td><td>variable</td><td><p>Generic extension mechanism that can be used by InDesign SDK/C++ developers. Calls through to a C++ implementation aggregated onto the <code>kAppBoss</code> boss class. This is an easy way to extend JavaScript with C++ code.</p>
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td><i>interfaceID</i></td><td>An integer – the interface ID of the interface we’re calling from JavaScript. This interface needs to be aggregated onto the <code>kAppBoss</code> boss class.</td></tr>
<tr><td>[<i>parameter1</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter2</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter3</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter4</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter5</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter6</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter7</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter8</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter9</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter10</i>]</td><td>An optional parameter</td></tr>
<tr><td>[<i>parameter11</i>]</td><td>An optional parameter</td></tr>
</table>
<p>The underlying implementation can return a return value. See the cookbook for a practical example.</p></td></tr>
<tr><td><i>callExtension(0x90b6c,…</i></td><td>variable</td><td><p>Active Page Items also contains a built-in ‘extension’, based on the previous extension scheme. It allows various enhancements built in to Active Page Items to be used from ExtendScript.</p>
<p>To call this built-in extension, the <i>interfaceID</i> parameter must be equal to 0x90b6C (also known as</p></td></tr>
</table>

`IID_IACTIVEPAGEITEMSCRIPTUTILITIESEXTENSION`).

When using this built-in extension, *parameter1* is an integer ‘opcode’ which determines the desired functionality of the call; the other parameters are or are not used depending on the opcode.

The following opcodes have been defined in Active Page Items 1.0.47:

Opcode name	Opcode value
kOpCode_SetApplicationModalLock	10001
kOpCode_ClearApplicationModalLock	10002
kOpcode_IsApplicationModalLocked	10003
-reserved-	10004
-reserved-	10005
kOpCode_DiscardUndo	10006
kOpCode_RunScriptInEngine	10007
kOpcode_GetDocGUID	10008
kOpcode_FindDocGUID	10009
kOpCode_UIColorRGB	10010
kOpcode_SerialNumber	10011
kOpcode_FrontDocGUID	10012
kOpcode_ActiveDocGUID	10013
kOpcode_OwningDocGUID	10014
kOpcode_SPLNFile	10015
kOpcode_MoveInto	10016
kOpcode_Get_theItem	10017
kOpcode_OwningDoc	10018
kOpcode_OwningPage	10019
kOpcode_OwningSpread	10020
kOpcode_IsValidID	10021
kOpCode_SystemID	10022

• The first two opcodes allow you to lock and unlock InDesign’s user interface into a modal mode.

This can be handy while executing an external application – by locking InDesign you can be sure the user won’t be able to do anything until you unlock it again. Also check the `app.launchWith` method in this regard.

• The ‘discard undo’ opcode allows you to clear the undo stack for an InDesign document. You should pass a reference to the document as *`parameter1`*.

‘Regularly’ calling this function in an intensive ExtendScript can give you a significant speed boost – e.g. calling this after about every 50 document changes seems to be a good amount.

- The ‘run script in engine’ opcode only works in InDesign CS3, and is a replacement for the *app.doScript* which also allows you to specifiy an engine for the script to be executed. This can also be achieved via *app.doScript*, but this form is a bit easier to use.

The syntax is

```
success =
  app.callExtension(
    0x90B6C,10007,
    script,
    engineName,
    displayDialogs,
    useDebugger);
```

The last three parameters are optional. *engineName* is a string with the engine name to use; *displayDialogs* and *useDebugger* are booleans.

- The ‘serial number’ opcode gives you a way to access the serial number of InDesign – this can come in handy when implementing various protection schemes, or when sorting out installation issues.

```
serialNumber = app.callExtension(0x90B6C,10011);
```

From CS4 onwards, the serial number is hashed – it has no longer an obvious relation to the InDesign activation code, as was the case in CS3 and earlier.

- GUIDs

The ‘get doc GUID’ function is used as follows:

```
guid =
  app.callExtension(0x90B6C,10008,document);
```

This function returns a GUID (Globally Unique Identifier) for a document. This is a string of the form “{nnnnnnnn-nnnn-nnnn-nnnnnnnnnnnnnnnn}” where all 'n' stand for a hexadecimal digit.

These GUID allow you to work with same-name documents that are concurrently open without getting tangled in the confusion that occurs when using *resolve* calls.

As long as a document is not moved from its original location on disk, it will keep the same GUID. Two documents with the same name but different paths will have different GUID.

Opening, closing, (re)saving to the same location will NOT change the GUID.

Moving a document to another location will change the GUID.

The ‘find open doc by GUID’ function is used as follows:

```
doc = app.callExtension(0x90B6C,10009,guid);
```

This function retrieves a reference to an already open document based on its GUID string.

These GUID strings are unique and persistent and allow persistent references to same-name documents without *resolve*-like issues (but the documents are meant to be 'unmovable' – i.e.

they cannot be renamed or moved to another folder).

Once the document is open, it can be retrieved via its GUID.

The ‘front doc GUID’ gives you the GUID of the front document – even if it is not the active document.

```
frontDocGuid =
  app.callExtension(0x90B6C,10012);
```

Similarly, the ‘active doc GUID’ gives you the GUID of the currently active document – even if it is not the front-most document.

```
activeDocGuid =
  app.callExtension(0x90B6C,10013);
```

The ‘owning doc GUID’ gives you the GUID for the document that owns a particular page item – you hand the function a page item, and it’ll tell you what the document’s GUID is.

```
owningDocGuid =
  app.callExtension(0x90B6C,10014,pageItem);
```

• The ‘SPLN file’ opcode gives you access to the file path of the currently executing SPLN file (e.g. so you can find associated resources that are stored in the same folder):

```
splnFile =
  app.callExtension(0x90B6C,10015);
```

• The ‘Move Into’ opcode allows you to easily move a page item into another one without needing to use cut and paste. Especially handy with server versions of Active Page Items.

```
app.callExtension(
  0x90B6C,10016,
  newParent,movingChild);
```

• The ‘UI Color’ opcode gives you access to the RGB values behind a particular UIColor:

```
uiColor = UIColor.cuteTeal;
rgbValues =
  app.callExtension(0x90B6C,10010,uiColor);
```

• The *`kOpcode_Get_theItem`* gives you access to the value of *`theItem`* as predefined at the beginning of each event handler call - even if *`theItem`* is overwritten, using this opcode allows you to retrieve its original value.

```
theItemAgain = app.callExtension(0x90B6C,10017);
```

• The *`kOpcode_Owning…`* opcodes navigate the page item hierarchy. The spread and page versions are very simple-minded - they cannot handle any kind of nesting; we intend to improve these in 1.0.48.

```
theDoc =
app.callExtension(0x90B6C,10018,thePageItem);
```

• The *`kOpcode_IsValidID`* allows you to check whether a page item id refers to a valid page item without need to force an exception to throw.

```
usableId =
app.callExtension(0x90B6C,10021,theDoc,theId);
```

•The *`kOpCode_SystemID`* returns the system ID used by the

<table>
<tr><td></td><td></td><td>licensing system – this can be handy for generating license files.

`theSysID = app.callExtension(0x90B6C,10022);`</td></tr>
<tr><td>*createProgressBar*</td><td>-</td><td>Brings up a progress bar window. This window stays on the screen until the current ExtendScript event handler terminates or until the *setProgress* is called with a value beyond the value of the *end* parameter.

When using this method from a regular (non-event handler) script, the only way to make the bar disappear is to call *setProgress* with a value beyond the end value.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*title*</td><td>String</td></tr>
<tr><td>[*start*]</td><td>Integer, default 0</td></tr>
<tr><td>[*end*]</td><td>Integer, default 100</td></tr>
<tr><td>[*showCancelButton*]</td><td>Boolean, default false</td></tr>
</table>
The first parameter is required, the three others are optional.

To move the progress bar you need to regularly call the *setProgress* method. *SetProgress* also returns a Boolean which corresponds to the ‘Cancel’. You can check its value and interrupt a lengthy process if so desired.</td></tr>
<tr><td>*getDataStore*</td><td>variable</td><td>Also see *`setDataStore`*. This method retrieves data from one of two associative arrays.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*key*</td><td>String</td></tr>
<tr><td>[*useTempDataStore*]</td><td>Boolean</td></tr>
</table>
If the optional *`useTempDataStore`* parameter is omitted, it defaults to *`false`*.

If *`useTempDataStore`* is true, this method accesses a set of data items that are not saved with the application preferences.

If *`useTempDataStore`* is false, this method accesses another set of data items that are saved along with the application preferences.

Preferrably you should use a key that starts with a reversed domain name to avoid conflicts – e.g. Rorohiko would use keys similar to ‘com.rorohiko.someotherdata’ to avoid key clashes.</td></tr>
<tr><td>*launchWith*</td><td>-</td><td>Launches an external program while passing it a particular file to open.

If you simply want to open an application without associated document, you can optionally pass the path to the application via the first parameter, the document path, and leave the application path undefined.
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td>*documentPath*</td><td>String</td></tr>
<tr><td>[*applicationPath*]</td><td>String</td></tr>
<tr><td>[*modeFlags*]</td><td>Integer</td></tr>
</table>
</td></tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>
<p>For the two paths you’ll typically will want to pass the <code>fsName</code> attribute of an InDesign <code>File</code> object.</p>
<p><em>modeFlags</em> is an integer which is calculated by adding zero or more of the following flag values:</p>
<table>
<tr><th>Flag</th><th>Value</th></tr>
<tr><td><em>asynchronous</em></td><td>1</td></tr>
<tr><td><em>background</em></td><td>2</td></tr>
<tr><td><em>modal</em></td><td>4</td></tr>
</table>
<p>Asynchronous launching will return control to ExtendScript while the application is still being launched.</p>
<p>Background will keep the launched application in the background.</p>
<p>Modal is a special feature: it will lock InDesign’s user interface into an artificial ‘modal mode’ until the launched application quits.</p>
<p>This assists in applications where an external application is used to develop dialogs that act almost the same as built-in InDesign dialogs. An example is provided with the documentation.</p>
</td>
</tr>
<tr>
<td><em>registerMenuItem</em></td>
<td>-</td>
<td>
<p>Registers a global menu item into the <b>API</b> menu on the menu bar. Once registered, a menu item will remain there until InDesign quits or until it is removed by a corresponding call to the <em>unregisterMenuItem</em> method.</p>
<table>
<tr><th>Parameter</th><th>Type</th></tr>
<tr><td><em>eventCode</em></td><td>String</td></tr>
<tr><td><em>menuName</em></td><td>String</td></tr>
</table>
<p>The <em>eventCode</em> is treated differently depending on its prefix. There are two possible prefixes: <em>subject-</em> and <em>menu-</em></p>
<p><b>Simple context menu</b>: If the <em>eventCode</em> does not start with either <em>subject-</em> or <em>menu-</em> then the menu will be enabled when one or more selected page items match the registered <em>eventCode</em> with their <em>eventFilter</em> mask.</p>
<p>If none of the currently selected page items accepts the event code, then the menu is disabled. Selecting the menu item will cause the <em>eventCode</em> to be sent to the selected page items.</p>
<p><b>One observer for multiple items</b>: If the <em>eventCode</em> starts with <em>subject-</em> then the menu item will be enabled when one or more selected page items have one or more observers whose <em>eventFilter</em> mask matches the registered <em>eventCode</em>. Selecting the menu will cause the <em>eventCode</em> to be sent to the observers of the selected page items.</p>
<p><b>Menu bar menus:</b> If the eventCode starts with <em>menu-</em> then the menu item will be enabled when one or more objects match the <em>eventCode</em> with their <em>eventFilter</em> mask.</p>
<p>This is independent of whether these objects are selected or not. The mere presence of the object with a proper <em>eventFilter</em> causes the menu item to be enabled.</p>
<p>Typically you will have a scripted plug-in which has an</p>
</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td><p><em>eventFilter</em> that matches this <em>menuSomething</em> event code</p>
<p>These menu bar menus also work together with the <em>enableMenus</em> event: each observer which accepts an event code <em>menu…</em> can optionally also accept an <em>enableMenus</em> event code.</p></td>
</tr>
<tr>
<td><em>setDataStore</em></td>
<td>-</td>
<td><p>Also see <code>getDataStore</code>. This method stores data in one of two associative arrays.</p>
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td><em>key</em></td><td>String</td></tr>
<tr><td><em>data</em></td><td>Variable</td></tr>
<tr><td>[<em>useTempDataStore</em>]</td><td>Boolean</td></tr>
</table>
<p>If the optional <code>useTempDataStore</code> parameter is □oolean, it defaults to <em>false</em>.</p>
<p>If <code>useTempDataStore</code> is true, this method accesses a set of data items that are not saved with the application preferences.</p>
<p>If <code>useTempDataStore</code> is false, this method accesses another set of data items that are saved along with the application preferences.</p>
<p>Preferrably you should use a key that starts with a reversed domain name to avoid conflicts – e.g. Rorohiko would use keys similar to ‘com.rorohiko.someotherdata’ to avoid key clashes.</p></td>
</tr>
<tr>
<td><em>setProgress</em></td>
<td>Boolean</td>
<td><p>Also see <em>createProgressBar</em>. <em>setProgress</em> accepts a single parameter, which can be an Integer or a String.</p>
<p>When the parameter is an Integer value between the <em>start</em> and <em>end</em> values that were passed to <em>createProgressBar</em>, it will move the progress bar.</p>
<p>If the parameter is an Integer with a value greater than the end value, the progress bar window will be closed. A simple way to manage the progress bar window is to call <em>setProgress</em> with values from start to end+1 – the last call to <em>setProgress</em> will close the window.</p>
<p>If the parameter is a String, it will change the progress message.</p>
<p><em>setProgress</em> returns a Boolean which will be <em>true</em> when the user has clicked the optional <em>Cancel</em> button on the progress bar window.</p>
<p>The idea is that you regularly call <em>setProgress</em> to move the progress bar forward, and each time check the return value. If the return value is <em>true</em>, the user wants to cancel – so you should exit your processing loop as soon as possible.</p></td>
</tr>
<tr>
<td><em>unregisterMenuItem</em></td>
<td>-</td>
<td><p>Remove a global menu item from the <strong>API</strong> menu.</p>
<table>
<tr><td>Parameter</td><td>Type</td></tr>
<tr><td><em>eventCode</em></td><td>String</td></tr>
</table></td>
</tr>
</table>

4. Predefined event codes

It is perfectly acceptable to ‘invent’ most any `eventCode` you want – when sending events from one page item you can choose pretty much any descriptive string as the event code – for example, when defining the event code attached to a context menu.

These event codes end up in `theItem.eventCode` – the `eventCode` property of the page item whose JavaScript is being executed.

Some event codes are treated differently based on their prefix.

4.1. *menu…* event codes

Event codes starting with *menu…* are reserved for handling menu bar menus.

You can invent any event code you want for this, as long as it has the *menu…* prefix.

4.2. *external…* event codes

Event codes starting with *external…* are not blocked after the plug-in has been compiled.

Normally, once a scripted plug-in is compiled, it is 'closed off' to the outside world ('outside world' meaning: other, non-compiled scripts and compiled scripts with incompatible component ids). Any calls to *handleScriptEvent* from the outside are ignored.

However, an exception is made for any event codes that start with the prefix *external…*

By selectively implementing some user defined event codes with this prefix, a developer can open up a scripted plug-in to other, third party scripts (these can be JavaScript, AppleScript or VBScript).

4.3. *subject…* event codes

When implementing context menus using a centralized controller object (see example in cookbook), you should make sure to prefix all your event codes with the *subject…* prefix.

This prefix tells *APID* that it needs to attach these menu options to any of the observed subjects of the controller, and not to the controller itself.

Without the *subject…* prefix, all these context menu options would only appear on the page item that is wrapped around the controller script.

With the *subject…* prefix, *APID* will instead attach the context menu to all subjects of the controller (as defined via the controller's *List of Subjects* field on the *Active Page Item Developer* palette).

4.4. Predefined event codes

The predefined event codes listed below are sent automatically by the *Active Page Item Developer* plug-in as the result of various external events.

Future versions of the *Active Page Item Developer* plug-in will most probably provide additional event codes.

eventCode	Description

created	Sent when a page item is just created. One way to get this event in practice is to copy-paste an existing page item that has a script attached to it – when the copy is pasted, the copied element's script will receive a `created` event and can react to it.
deselected	Sent when this page item has been deselected. Also see the `subjectDeselected` event code.
docClose	Sent when the document is about to be closed.
docDeselected	Sent when the document is losing front focus
docLoaded	Sent just after the document layout window opens.
docSave	Sent when the document is about to be saved.
docSelected	Sent when the document was brought to the foreground
docPrint	Sent when the document is about to be printed. You can set `theItem.tempDataStore` to `true` to abort the print operation.
docPrintConfigure	Sent when the document print setup dialog is about to be shown. You can set `theItem.tempDataStore` to `true` to suppress the dialog.
docPrintConfigured	Sent after the user has finished with the print setup dialog. You can set `theItem.tempDataStore` to `true` to abort the print operation. `theItem.eventData` is a string which reflects whether the print was completed ("success") or failed ("fail").
docPrinted	Sent after the document was printed.
enableMenus	Sent to an observer which also has a *menu…* event code when the corresponding menu is about to be displayed on the menu bar. The event data (`theItem.eventData`) will contain the menu item in question, and this method should return an integer value in `theItem.tempDataStore` If `theItem.tempDataStore` is set to non-zero on exit, then the menu item will be enabled. The value returned is a combination of flags: 0 = kDisabled_Unselected 1 = kEnabled 2 = kSelected (checked) 4 = kMultiSelected (dash) 8 = kUnderline (not on Windows) kMultiSelected overrides kSelected. Don't attempt to use the *return* statement – it won't work.
fileChanged	Sent after the observed external file has changed. Also see the reserved keys for `getDataStore/setDataStore: $FILEPATH$, $LASTMODIFIED$`
idle	Sent when the application is idle, about once per second. This allows you to implement background processing tasks.
loadContextMenu	Sent when to a page item when the context menu for that selected item needs to be reloaded.
modified	Sent when this page item is being modified. Also see the

	`subjectModified` event code.
modified-objectstyle	Sent when the page item is assigned a new object style
modified-recomposed-overset	Sent when the InDesign recomposition engine has finished recomposing the text, and there was some overset text.
modified-recomposed-nooverset	Sent when the InDesign recomposition engine has finished recomposing the text, and there was no overset text.
modified-text	Sent when the active page item is a text frame, and the text contained inside has been reflown (e.g. because text was added or modified, or the size or shape of the box has changed). There is a default 3-second lag during which no user activity is seen before the event is sent – this to avoid a rapid fire of `modified-text` events while the user is tapping and editing away. This event is now deprecated – the `modified-recomposed-overset`, `modified-recomposed-nooverset` and `modified-usertyping` events should give better control.
modified-textformat	Sent when any text has been assigned a different format.
modified-usertyping	Sent when the user has been editing the story associated with the text frame.
parentModified-recomposed-overset	Sent when the InDesign recomposition engine has finished recomposing the text, and there was some overset text in this item's parent. This is useful for anchored frames – it allows them to react to recomposition of the story they are embedded in.
parentModified-recomposed-nooverset	Sent when the InDesign recomposition engine has finished recomposing the text, and there was no overset text in this item's parent. This is useful for anchored frames – it allows them to react to recomposition of the story they are embedded in.
parentModified-text	Sent when the active page item parent is a text frame, and the text contained inside has been reflown (e.g. because text was added or modified, or the size or shape of the box has changed). There is a default 3-second lag during which no user activity is seen before the event is sent – this to avoid a rapid fire of `modified-text` events while the user is tapping and editing away. This event is deprecated – the `parentModified-recomposed-overset`, `parentModified-recomposed-nooverset` and `parentModified-usertyping` events should give better control.
run	Sent when the user selects the Run Script menu item in the Active Page Item Developer palette menu.
scriptTagChanged	Sent after this page item's script tag was changed.
selected	Sent when this page item is being selected. Also see the `subjectSelected` event code.
subjectCreated	Sent when one of the observed page items was just created. In many cases, you will want to set the *List of Subjects* (a.k.a. the `subjectScriptTagList` property) to a single * wildcard expression to capture any new page item being created. This because most newly created page items have an empty script tag (unless they are the result of a copy-paste operation). Also see the `created` event code.

subjectDelete	Sent when one of the observed page items is about to be deleted. Also see the `delete` event code.
subjectDeselected	Sent when an observed page item has been deselected. Also see the `deselected` event code.
subjectFileChanged	Sent when one of the observed page items has noticed a changed in the attached external file which it itself is observing. Also see the `fileChanged` event code.
subjectLoadContextMenu	Send when the context menu(s) for a number of selected page items must be calculated. You probably need to analyze `app.selection` to decide what context menu array to return.
subjectModified	Sent when one of the observed page items has been modified. Also see the `modified` event code.
subjectModified-objectstyle	Sent when one of the observed page items has been assigned a new object style. Also see the `modified-objectstyle` event code.
subjectModified-recomposed-overset	Sent when the InDesign recomposition engine has finished recomposing the text in one of the observed page items, and there was some overset text. Also see the `modified-recomposed-overset` event code.
subjectModified-recomposed-nooverset	Sent when the InDesign recomposition engine has finished recomposing the text in one of the observed page items, and there was no overset text. Also see the `modified-recomposed-nooverset` event code.
subjectModified-text	Sent when one of the observed page items text contents have been reflown due to some change (e.g. text added or modified, size of box changed,…). Also see the `modified-text` event code.
subjectModified-textformat	Sent when one of the observed page items text content has been assigned a different format. Also see the `modified-textformat` event code.
subjectModified-usertyping	Sent when the user has been editing the story associated with the observed text frame. Also see the `modified-usertyping` event code.
subjectParentModified-recomposed-overset	Sent when the InDesign recomposition engine has finished recomposing the text, and there was some overset text in the observed item's parent. This is useful for anchored frames – it allows them to react to recomposition of the story they are embedded in.
subjectParentModified-recomposed- nooverset	Sent when the InDesign recomposition engine has finished recomposing the text, and there was no overset text in the observed item's parent. This is useful for anchored frames – it allows them to react to recomposition of the story they are embedded in.
subjectParentModified-text	Sent when the observed item's parent is a text frame, and the text contained inside has been reflown (e.g. because text was added or modified, or the size or shape of the box has changed). There is a default 3-second lag during which no user activity is seen before the event is sent – this to avoid a rapid fire of `modified-text` events while the user is tapping and editing away. This event is deprecated – the `subjectParentModified-recomposed-overset`, `subjectParentModified-recomposed-nooverset` and `subjectParentModified-usertyping` events

	should give better control.
subjectScriptTagChanged	Sent when the `label` property of one of the observed page items was modified. Also see the *`scriptTagChanged`* event code.

5. Predefined keys

When using the *`getDataStore/setDataStore`* methods, a few subsets of all possible keys get special treatment.

5.1. External keys

Any key string that starts with the prefix 'external…' is considered a 'public' key. Normally, when a script is compiled by setting its componentId to a non-empty string, it becomes inaccessible to 'outside' scripts: its script code and data stores are closed for external access. However, data store elements whose key starts with the 'external' prefix remain accessible to 'outside parties'.

5.2. Reserved keys

Any key string that starts and ends with a dollar sign is reserved for use by the plug-in.

The following such key strings are already defined and in use:

`$ADORNMENT<adornmentID>$` (where *`<adornmentID>`* should be replaced by a unique string – preferably a reversed domain name, with possibly an underscore prefix). This allows you to add adornments to page items – these can either be temporary or permanent – depending on whether you use the *tempDataStore* or the *dataStore*. For more info see further - **9.8 Adornments**

`$DELAY_<eventcode>$` (where *`<eventcode>`* should be replaced by one of the APID event codes): defines a delay for various events, expressed in milliseconds. By default, the only event code that has a delay is *`modified-text,`* which has a default delay of 3000 milliseconds. The corresponding reserved key is *`$DELAY_modified-text$`* (with an underscore after *`DELAY`*, and a dash between *`modified`* and *`text`*).
This mechanism can be used for all APID-generated events, so if you'd execute *`theItem.setDataStore("$DELAY_selected$",2000);`* and then the user would select/deselect a page item many times in quick succession (less than 2 seconds between successive selections), you'll only get a single `selected` event.

`$DEMO_TIMED_OUT_MESSAGE$`: Contains a message that is displayed when you have used the optional licensing restrictions when compiling your plug-in. This message is used when your plug-in is in 'lapsed-demo'-mode and the user attempts to first access your plug-in during an InDesign session. A 'demo version' dialog will appear showing this message.

In both these message strings you can use two 'placeholders' - ^1 and ^2. The substring ^1 will be replaced by your plug-in's component name, and the substring ^2 will be replaced by the number of remaining demo days before the demo times out.

`$FILEPATH$`: File path of the file being observed through use of the *`fileChanged`* event code. Uses the *`$LASTMODIFIED$`* entry to keep track of when the file was last modified – if the last modified date on the file changes, it will fire a *`fileChanged`* event.

`$GROUP_EVENTS$`: by default contains *`false`*. Can be set to *`true`*, after which events received by this particular page item will be grouped for efficiency whenever possible.

If multiple similar events need to be sent to this same target item, they will be grouped, and the *`theItem.eventSource`* can either be a single event source, or it can be an array of multiple grouped event sources.

`$GROUP_EVENTS$` also affects the way the *`enableMenus`* event is handled.

If *`$GROUP_EVENTS$`* is set to *`false`*, a separate *`enableMenus`* event will be sent for each individual menu item in the current menu group.

If *`$GROUP_EVENTS$`* is set to *`true`*, the event data will be an array containing multiple menu event codes, and an array of booleans must be returned – one for each menu event code in the array that was received via the event data.

`$GROUP_EVENTS$` can greatly reduce overheads, for example when processing *`subjectSelected`* when the user does a 'Select All': instead of causing many events, only a single grouped event will be fired.

`$LASTMODIFIED$`: Used to keep track of the last modification of the file being observed by the *`fileChanged`* event. Also see *`$FILEPATH$`*.

`$LICENSE_MESSAGE$`: Contains a message that is displayed when you have used the optional licensing restrictions when compiling your plug-in. This message is used when your plug-in is in demo-mode and the user first accesses your plug-in during an InDesign session. A 'demo version' dialog will appear showing this message.

`$LICENSE_URL$`: Contains a URL to be used when you have implemented the optional licensing restrictions when compiling your plug-in. This URL is accessed when your plug-in is in demo-mode and the user clicks one of the 'Get License…' buttons on the APID about screen or 'demo version' dialog.

In this URL you can use a number of 'placeholders' - ^1, ^2. ^3, and ^4. The substring ^1 will be replaced by the serial number of the copy of InDesign that is currently running, and the substring ^2 will be replaced by the name of your scripted plug-in. Substring ^3 will be replaced by a unique system identifier for the computer requesting the license. String 3 is formatted as xx:xx:xx:xx:xx:xx with each 'x' a hexadecimal digit. Substring ^4 is replaced by a letter R, D or E – reflecting the level of API license that the computer has installed. R = free APIR license, D = licensed for APID, E = licensed for APIE.

`$PLUGINLEVEL$:` The current 'level' of *APID ToolAssistant* installed (returns a letter 'R', 'D' or 'E'). 'R' = unlicensed *APID ToolAssistant* or *Active Page Item Runtime*, 'D' = licensed *APID ToolAssistant*, 'E' = *Active Page Item Enterprise.*

`$PLUGINNAME$:` Name of the installed *APID ToolAssistant* (or compatible).

`$SPLNVERSION$`: can be used to store a version number for the compiled .spln file. The *APIDTemplate.indd* provided in the *APIDToolkit* release has code that demonstrates how to use this. The end-result is that the API *About Window* will display a version number for any compiled .spln files that use this feature.

`$VERSION$`: Contains the string form of the version number of the *APID ToolAssistant* plug-in; accessing `theItem.getDataStore("$VERSION$")` gives you a handy means of making your script more resistant against changes in the functionality of the *APID ToolAssistant* plug-in.

5.3. What does a componentId do?

Before diving too deep into component id strings: make sure you try out the *APIDTemplate.indd* document that is provided with APID – it automates a great deal of the complexities surrounding component ids. Check the cookbook for a practical example.

Component id strings are page item properties that are used to 'lock' your JavaScripts and make sure they cannot be visually or programmatically inspected and are not stored in a readable format.

Also, compiled code cannot be inspected through the script debugger. Enabling the debugger will disable any compiled scripts whose component id is not the empty string. If you want to debug a compiled script, you must use an empty component id during the debug phase.

Component ids also allow you to create time-limited demo versions of your scripts that stop working after a certain date or after a certain number of days have passed.

Demo versions of your scripts can be converted into fully enabled versions by sending your end-user a small 'license file' which contains encrypted data that enables a particular copy of InDesign (identified by the first 20 characters of its 24-character activation code and optionally a 17-character system id) to run a particular component id.

Once a `componentId` has been assigned to a page item, its *Active Page Item Developer* data become inaccessible for inspection.

The *Active Page Item Developer* data is only available to scripts attached to those page items with a compatible component id – compatible component ids have the same password in the second subfield of their name field.

Page items with incompatible or empty component ids cannot access the locked properties and methods (except for event codes and data store keys that start with the prefix 'external…')

This allows the creation of a 'swarm' of page items that can access each other's *Active Page Item Developer* data, but are inaccessible for anything outside the swarm.

When you want to distribute your scripts to third parties, you can go about it several ways.

First, you could attach a number of scripts to a number of page items in a (template) document, and distribute the template document. In that case, you'd have to assign the same *`componentId`* to each of the elements that has a script assigned to it and is part of your scripted solution.

The better solution is to store your scripts in a single page item, and make the page item into a 'controller', which observes other page items.

Now you only have to protect a single page item by setting its component id.

An additional benefit is that you can 'compile' this single page item into a so-called 'scripted plug-in'. (It is not a 'real' compilation - all that happens is that the encrypted *Active Page Item Developer* data is saved into the scripted plug-in file. Behind the screens it is still the same JavaScript. Optionally, comments and unnecessary white space can be removed during compilation).

A scripted plug-in will start its life simply as some box on a spread with some script attached to it.

The box is made into an observer, and can be tested. Once it all works correctly, it can be made to save its 'Active Page Item' contents into a stand-alone *.spln* file which is stored in the plug-ins folder by calling its *`saveToPlugin`* method.

When a document is opened or a new document is created, the *.spln* file is loaded and made into a 'page-item-like' entity that observes and reacts to events just like normal active page items do.

6. Licensing your code

6.1. ComponentId structure

A component id is a structured string. You compose it from the following comma-separated fields:

`name,minVersion,endYear,endMonth,endDay,numactual,numdemo`

name itself is composed of the following three or four semicolon-separated subfields:

`softwareName;password;copyright[;Free]`

None of these fields should contain commas or semicolons. Preferrably, the password should also not contain any slash characters.

softwareName is a descriptive name that describes the software.

password is a string that locks out external access by 'outside scripts'.

copyright is a string that could include your (company) name and other copyright information.

The optional fourth field is the word *Free* (with uppercase 'F'). If this word is present, the software will not need a license file to run. It is still locked against inspection, but anybody who has a licensed *APID ToolAssistant* plug-in installed can make use of your script without need for a license file.

minVersion is a version number in the form n.nnn or n.n.nn, which reflects on the version number of the Active Page Item plug-in: the script will only be active if the version number of the Active Page Item is at least this version number. Also see the predefined key *$VERSION$* for the *getDataStore/setDataStore* methods for an alternate version-checking approach.

endYear, endMonth, endDay allow the creation of time-limited demo-solutions: your encrypted script will cease to be active after this date (unless a license file is installed).

numActual and *numDemo* are an alternate form of time-limitation: they allow the demo-solution to be used for *numDemo* consecutive days, or for *numActual* days of actual use.

All three forms of time limitation can be used concurrently - whichever times out first determines when the script stops working.

6.2. Beg Window and About Window

There are a few other relevant bits of data with regards to the licensing system.

For stand-alone scripts, there are three text fields that can be configured via the InDesignScriptCompiler.

For scripted plug-ins, there are three keys *$LICENSE_URL$*, *$LICENSE_MESSAGE$*, *$DEMO_TIMED_OUT_MESSAGE$* which can be used with the *getDataStore/setDataStore* methods on the object that corresponds to your scripted plug-in. You would normally initialize these prior to compilation. Check the *APIDTemplate.indd* document for example code.

$LICENSE_URL$ or the contents of the *License URL* field in the *InDesignScriptCompiler*:
this URL is accessed when your script or plug-in is in demo-mode and the user clicks one of the ‘Get License…’ buttons on the APID *About…* screen or the ‘demo version’ dialog (also known as ‘the beg window’). You would normally set this to point to a URL on your web site.

In this URL you can use four ‘placeholders’ - ^1, ^2, ^3, and ^4. The string ^1 will be replaced by the serial number of the copy of InDesign that is currently running, and the string ^2 will be replaced by the name of your scripted plug-in (as extracted from the componentId). Substring ^3 will be replaced by a unique system identifier for the computer requesting the license. String 3 is formatted as xx:xx:xx:xx:xx:xx with each ‘x’ a hexadecimal digit. Substring ^4 is replaced by a letter R, D or E – reflecting the level of API license that the computer has installed. R = free APIR license, D = licensed for APID, E = licensed for APIE.

$LICENSE_MESSAGE$ or the contents of the *Licensing Message* field in the *InDesignScriptCompiler*: this message is displayed when your plug-in or script is in demo-mode and the user first accesses your plug-in during an InDesign session.

$DEMO_TIMED_OUT_MESSAGE$ or the contents of the *Timed Out Message* field in the *InDesignScriptCompiler*:
this message is displayed when your plug-in or script is in ‘lapsed-demo’-mode and the user attempts to first access your plug-in or script during an InDesign session.

In both these message strings you can use two special ‘placeholders’ - ^1 and ^2. The string ^1 will be replaced by the name of your plug-in, and the string ^2 will be replaced by the number of remaining demo days before the demo times out.

6.3. License Generator

The component id system works in conjunction with a command-line utility, which allows you to generate license files for your compiled plug-in and compiled scripts.

In order to create a proper license, you need the *APIDLicenseGenerator* command-line utility.

You could have a look at the *APIDTemplate.indd* example document – it automates a few of the steps described below, and provides a copy-paste-able command line template in the generated template files.

Preparing

Start a command line session (use the Terminal application from the Mac OS X `Applications/Utilities` folder, or the *Command Prompt* from the *Start – Programs – Accessories* menu item on Windows).

Navigate to the proper location.

On Windows, if you have more than one drive letter (not just C:, you might need to change drive first – e.g. if the command line utility resides in a folder that is located on drive F, you first enter a command line ‘F:’ (the drive letter, followed by a colon, followed by <Return> or <Enter>).

You can the navigate quickly to the correct folder by typing ‘cd ‘ (cee dee followed by a space) on the command line, and then dropping the folder icon of the folder which contains the command line utility into the command line window. Then press <Return> or <Enter>. This works on Mac OS X and on Windows.

Command line parameters

This utility accepts 5 or more parameters, separated by spaces.

Parameter 1 is the name of the license file (without the .license suffix – the license generator will automatically append the suffix).

Parameter 2 is composed of the name and the password used in the component id (see previous section about the component id structure). The softwareName and password are to be separated by semicolons. This parameter is case-sensitive. It is best to enclose this command-line parameter in quotes.

Parameter 3, 4, 5 are either all equal to -1, or they can be a year, month, day value – this allows you to generate time-limited license files. In most cases, you’ll set these all to -1.

Parameter 6 and beyond is optional. If you omit parameter 6 and beyond, you will generate a ‘global’ license file, which works on all copies of InDesign, irrespective of their serial number.

You will normally include one or more InDesign serial numbers and system ids as parameter 6, 7… When you do, you'll generate a license file that targets those specific serial numbers and system ids.

InDesign serial numbers consist of the first 20 characters of the 24-character InDesign activation code – you don't need a user's full activation code. The 20-character serial number is displayed on the InDesign 'About' screen.

After each serial number listed you can *optionally* add a system id – if you omit the system id, then the license file will work on all computers that share the same InDesign serial number.

System ids are formatted as xx:xx:xx:xx:xx:xx (where each 'x' is a hexadecimal digit).

You need the user's InDesign serial number, as well as the exact name componentId part that was used when compiling the plug-in.

Example

Let's assume you've already fired up a command-line window, and navigated to the correct location.

For the sake of the argument, assume the serial number of your end-user's copy of InDesign is
`12345678901234567890`
and the component id you used for compiling your scripted plugin was
`testPlugin;myPassword;(c) 2005 MyCompany Ltd,1.0.22,-1,-1,-1,20,20`

That means that the command line would look like this on Mac OS X:

```
./APIDLicenseGenerator testFile “testPlugin;myPassword” -1 -1 -1
12345678901234567890
```

and on PC:

```
APIDLicenseGenerator testFile “testPlugin;myPassword” -1 -1 -1
12345678901234567890
```

That will generate a file called *`testFile.license`* that has to be imported by the end-user in order to lift the demo restrictions. This particular license file will work on all computers that share the same InDesign serial number.

If you also have a system id *`AA:BB:CC:11:23:45`* (which you retrieved from the *`$LICENSE_URL$`*-based URL emitted when the user clicked 'Get License…'), you'd use:

```
./APIDLicenseGenerator testFile2 “testPlugin;myPassword” -1 -1 -1
12345678901234567890 AA:BB:CC:11:23:45
```

on Mac OS X, and on PC:

```
APIDLicenseGenerator testFile2 “testPlugin;myPassword” -1 -1 -1
12345678901234567890 AA:BB:CC:11:23:45
```

That will generate a file called *`testFile2.license`*. This license file will work only on the particular computer that was used to request the license, and only with the copy of InDesign with this particular serial number.

6.4. Combined .spln/APID ToolAssistant license.

From APID 1.0.44 onwards, an enhanced licensing system has become available which allows a script developer to provide his customers with special license files for his scripted tools which can optionally embed a license for *APID ToolAssistant* if needed.

This makes it easier for an end-user to get the needed licenses. Instead of needing two license files - a license file from the script developer for his scripted solution, and a second license file from Rorohiko for *APID ToolAssistant* - the end-user can now use a single, combined license file provided to him by the script developer.

In order to use the enhanced licensing scheme, the script developer needs to use an 'approved componentId' for his commercial, APID-based .spln files. For example, if the ‘normal’ component id he’s using were:
`TestAPID;MyPassword;(c) 2008 Rorohiko Ltd.,1.044,-1,-1,-1,3,4`
then the script developer could get this component id approved by Rorohiko.

Approving means that the componentId password is prefixed with a unique approval number and a slash - for example it would become something like:
`TestAPID;12345654654/MyPassword;(c) 2008 Rorohiko Ltd.,1.044,-1,-1,-1,3,4`

The script developer can still change parts of the component id without invalidating the approval number; but four of the fields are locked after approval:
- the component license name (*TestAPID* in the example)
- the copyright string (e.g. *(c) 2008 Rorohiko Ltd.*)
- the two demo-day-counts (e.g. *3* and *4*).

The script developer can change the other component id fields (e.g. the minimum *APID ToolAssistant* version, the end year, month, day, and the password) without invalidating the approval number.

Of course, getting an approved component id is not a must, but has the advantage that it gives the script developer access to this enhanced licensing scheme.

The idea is that Rorohiko approves only reasonable demo times for third party tools - e.g. we would not approve a request for a component id with demo-days set to 1000 days or so.

Now suppose an APID-based developer has an approved component id and he uses it for a commercial tool.

Also suppose an end-user installs a demo of this tool, and the end-user has a timed-out demo version of *APID ToolAssistant* installed (e.g. because he’s tried out the APIDToolkit demo some time earlier).

Here's the advantage of an approved component ID: even though the end-users' *APID ToolAssistant* has timed out, it will still allow any demo of any tool with an approved component ID to work normally for the duration of the .spln demo time.

In this case, the *APID ToolAssistant*-based tool demo would be allowed to work normally, until the end of its own demo period, but the end-user would still not be able to use *APID ToolAssistant* to build another .spln (because it's a lapsed demo, and he's had his chance to try APID Toolkit).

Essentially, an approved component id 'overrides' the lapsed demo status of *APID ToolAssistant*, until the component id's own demo time-out takes effect.

That fixes the issue of getting stuck with a 'dead' *APID ToolAssistant* during a developer demo time-out period.

To make licensing easier, a developer will then be able to request from Rorohiko the creation of a license file for an approved tool that also has an additional *APID ToolAssistant* license 'embedded' in the same file.

Control of this ‘combined’ license file generator remains at Rorohiko.

Some time into the future there will be a protected tool that's accessible via the web where a script developer can log in and enter his component id data, the user's InDesign serial number and system id, and in return he will get a license file which has the *APID ToolAssistant* activation embedded as well as the activation for his own tool.

This tool will be 'web-linkable' so the script developer will be able to transparently embed it into his own web site, and all operations can stay transparent for the end-user.

This tool is not available yet. Until this web-based tool becomes available, combined license files will need to be requested via e-mail – contact APIDlicenses@rorohiko.com

At the end of a pre-agreed time period, the developer will then be invoiced by Rorohiko for the number of *APID ToolAssistant* licenses that have been embedded in such a way.

In this enhanced licensing scheme, the end-user now receives a single license file from the script developer, and this single license file activates both APID and the scripted software.

6.5. Licensing System Limitations

No guarantees are being made that the licensing system for scripted plug-ins and locked page items provided by this plug-in is ‘hacker-resistant’. There is no doubt about it that a determined hacker can crack the licensing scheme.

7. Filtering Script Tags

There are two levels for filtering labels.

The first level is defined by the `subjectScriptTagFilter` property - it determines what page elements are to be observed. Any page item whose label matches at least one of the listed wildcard expressions becomes an observed subject, observed by this page item.

The second level of filtering is optional, and resides inside the `eventFilter` expression: for each selected `eventCode`, there is an optional list of wildcard expressions for script tags, which allows you to further restrict the page items whose events are to be processed.

If the second level of filtering is present, then only page items that pass both the first and second level filtering will be able to send or cause events to be processed by this observer.

8. Event Filter Expressions

An *eventFilter* string is composed of a comma-separated list of zero, one, or more event expressions.

```
eventFilter :=
  [ <eventExpression> [ ‘,’ <eventExpression> ]...]
```

An *eventExpression* is composed of an event code, and optionally a list of label expressions in parenthesis.

These label expressions used to further filter the labels of any page items that were already filtered through the `subjectScriptTagFilter` (i.e. when calculating the list of page items whose events to handle, the plug-in will first take into account the `subjectScriptTagFilter` before ever examining the label expressions in the *eventFilter*.

```
eventExpression :=
  <eventCode>
    [ ‘#’ <engineName> ]
    [ ‘(‘ <labelExpression> [ ‘,’ <labelExpression> ]... ‘)’ ]
```

An *eventCode* is composed of a main event code string, optionally followed by a repetition of dashes and an event code suffixes (also known as sub-event code suffix).

```
eventCode :=
  <mainEventCode> [ ‘-’ <subEventCode> ]…
```

An *engineName* is the optional name of an InDesign scripting engine. In InDesign CS and CS2, these *engineName* are ignored. In InDesign CS3, they are used to determine what script engine should process the event when it is received.

A *labelExpression* is a wildcard string for matching labels (also known as script tags) – events caused by any item whose label matches this expression will be passed on to the attached JavaScript.

```
labelExpression :=
  <wildcardString>
```

The *mainEventCode* is an alphanumeric string. For your own events (e.g. used with the `handleScriptEvent` method or with the `contextMenu` property) you can use any string you want.

The *APID ToolAssistant* plug-in also generates a number of pre-defined event codes depending on external events taking place.

```
mainEventCode :=
  <alphaNumString>
```

Some events utilize a sub-event code to allow further filtering. For example, two of the internally generated events are *`modified`* and *`modified-text`*.

For all intents and purposes, *`modified-text`* is a *`modified`* event, but further refined. To correctly capture all *`modified`* events, you should use the *`modified*`* wildcard expression, which matches all *`modified`* events.

```
subEventCode :=
  <alphaNumString>
```

alphaNumString and *wildcardString* are basic building blocks used for the other items – *alphaNumString* should not start with a digit.

```
alphaNumString :=
  string of a-z, A-Z, 0-9, $, _
```

In a *wildcardString*, the * means: 0 or more characters; the *?* means: exactly one character.

```
wildcardString :=
  string of a-z, A-Z, 0-9, $, _, * and ?
```

Example Event Filter Expression:

```
subjectSelected(yellow*Box,green*Box),subjectDe*(*Box),subjectModified-text(*Text*)
```

This filters *`subjectSelected`* events from any page item that has a label starting with *yellow* or *green* and ending in *Box* - e.g. *yellowAdBox, greenTextBox, yellowBox,...*

It also captures any event whose code starts with *subjectDe…* for any page item whose label ends in *Box* - e.g. *subjectDeselected* events from *yellowBox, yellowAdBox,....*

Lastly, it detects *`modified-text`* events (*mainEventCode* = *`modified`*, *subEventCode* = *`text`*) originating from any page item whose label contains the word *Text* - e.g. *greenTextBox,...*

9. Being a good script citizen

When developing scripts, it is important to observe some basic rules in order to avoid conflicts between multiple scripts, especially in InDesign CS and CS2.

9.1. Scripts can run unexpectedly

Because Active Page Items allows event-driven programming, you need to take into consideration that some innocuous commands can cause other scripted event handlers to launch and run during the execution of your script.

For example, calling *`app.documents.add()`* will trigger a *`docLoaded`* event for scripts monitoring this event. These scripts will be executed ‘inside’ the *`app.documents.add()`* command.

Any scripted command that can cause an event to be triggered can cause a 'sub-script' to run.

This means that scripts must always be careful not to unnecessarily disturb the environment they are running in – as to not disrupt the operation of the 'outer' script.

9.2. Globals and function names are often shared

The issue discussed in 8.1 has another side effect: if a script command causes an event to fire and a sub-script to run, then the outer script and the inner script share the same global variable space.

That means that global variables should be avoided as much as possible – local variables and function parameters are the best way to store and pass data.

Using unique prefixes

If you really must use a global variable, its name should be as unique as possible – the 'proper' behavior would be to name every global variable with a unique prefix related to the name of your application.

For example, Rorohiko's ChatterGoofy.spln uses a global whose name is gChatterGoofyPlugin.

This reduces the risk of clashes: the odds that the end-user would be running TWO applications that both are named ChatterGoofy is pretty remote.

Function names are another point of potential clashes – running a sub-script which redefines certain functions can will have unwanted side-effects.

In order to further reduce the odds of conflicts, Rorohiko will run a registry of global name prefixes – every script developer can check with us whether a particular prefix is already used, and if not, register a particular prefix with us. Contact pluginsupport@rorohiko.com with your request – things will be pretty manual at first, but eventually we'll switch to an automated system.

Prefixes should be composed of letters and/or digits and start with an uppercase letter (i.e. no underscores etc – letters and digits only).

There is no enforcement of prefixes! Registering a particular prefix simply means you are letting the world know you intend to use global names of the form:

```
<lowercase letter><Prefix><variableName>
```

and function names of the form:

```
<Prefix><functionName>
```

i.e. the prefix has to be prefixed with a single lower case letter before use to name a global or constant.

Typical letters are 'g' for globals, 'k' or 'c' for constants. Examples: *`gChatterGoofyPlugin`*, *`cChatterGoofyWindowWidth`*, *`ChatterGoofy_CalculateNewStory()`*,…

If the world chooses to ignore this, there is little that can be done except starting a dialog with the 'clashing party' – Rorohiko will never enforce adherence to prefixes; it's all purely voluntary.

Choosing less common prefixes will also help in avoiding clashes, even with scripts that do not adhere to the convention.

Embedding the script inside a protective function

Another good way to avoid clashes is to always wrap the complete body of your plug-in code inside a protective function.

As long as you make sure that all variables are declared with the 'var' command, they would act like globals, but they are actually local to the wrapper function.

```
PluginMain();

function PluginMain()
{
  var x;
  function AddOneAndX(y)
  {
    return(y+1+x);
  }

  x = 1;
  alert(AddOneAndX(x));
  alert(x);
}
```

Within the whole body of `PluginMain`, the variable x acts as a global variable – e.g. it is accessible from inside `AddOneAndX`. Also, `AddOneAndX` is not accessible in the global scope.

We are currently considering automating the above approach in Active Page Items – i.e. starting with some future version, there would always be an invisible wrapper around any plug-in code.

That would not completely close the door on abuse of global variables – if the developer forgets to declare the variable name with `var` he would still be accessing a variable in the outermost global scope.

It is possible to go even one step further, and not even define a function like `PluginMain` by using an anonymous function, for example like this:

```
(function(theItem)
{
  var x;
  function AddOneAndX(y)
  {
    return(y+1+x);
  }

  x = 1;
  alert(AddOneAndX(x));
  alert(x);
})(theItem);
```

Use a ‘private’ engine

From InDesign CS3 onwards, you can use a separate, private ExtendScript engine in order to avoid clashing with other people’s scripted material.

Also see section *8. Event Filter Expressions*, and look for the use of engineName.

9.3. Be aware of hidden nesting

Keep in mind that your script might be called recursively – if you choose to use a single controlling script, which handles a variety of events, you will often find that your script is called in a nested way.

For example, while processing *`docLoaded`*, your script might be causing *`modified`* events to occur, causing nesting to occur.

For example, suppose your script starts with:

```
...
gChatterGoofy_EventReceiver = theItem;
...
```

When nesting occurs, you can have all kinds of unexpected behavior with such command. The nested script overrides whatever the content of *`gChatterGoofy_EventReceiver`* was, and upon return, the outer script will probably break because this variable has unexpectedly changed value.

Instead it is much better to write ‘trigger-once’ code for globals – for example:

```
...
var gChatterGoofyPlugin;
if (gChatterGoofyPlugin == undefined)
{
  gChatterGoofyPlugin = RetrieveScriptedPlugin(theItem);
}
...
```

With this approach, even when nesting occurs, the variable *`gChatterGoofyPlugin`* will keep its original value.

9.4. Don’t redefine built-in functions

A user reported a conflict between an existing, stand-alone ExtendScript application and *Active Page Items* – the script failed with a strange error when *Active Page Items* was installed, and worked fine without *Active Page Items*.

It turned out neither the stand-alone script nor *Active Page Items* was at fault, but the two together were clashing.

The stand-alone script contained a custom function named *`resolve`* – which is perfectly legal.

However, *`resolve`* is also the name of a built-in function – one that Active Page Items relies on heavily.

What happened was that due to nesting, Active Page Items was now calling the redefined version of *`resolve`*, with all kinds of dire consequences.

One way to avoid these types of conflicts is to only use function names that start with an upper case letter – all built-in functions start with a lower case letter.

9.5. Be aware that the user interaction level might change.

If your script needs to show a dialog, you should make sure you test and possibly change the current user interaction level – other scripts might have changed this level causing your dialog to be disabled.

The following example function works for both CS, CS2 and CS3.

```
function EnableDialogs()
  {
    var done = false;
    try
    {
      app.userInteractionLevel =
        UserInteractionLevels.interactWithAll;
      done = true;
    }
    catch(err)
    {
    }
    if (! done)
    {
      try
      {
        app.scriptPreferences.userInteractionLevel =
          UserInteractionLevels.interactWithAll;
        done = true;
      }
      catch(err)
      {
      }
    }
    return done;
  }
```

This function is a bit 'rude' – it might be better to save the current interaction level first, and restore it after the dialog has been shown.

9.6. Cannot immediately close or save a document from a handler.

When you attempt to close or save a document from an event handler script, you will notice that this command is not executed straight away.

Active Page Items is not able to close or save a document when a handler is in progress. Instead it will postpone the request until the handler finishes.

For example, the following script will exhibit different behaviors when it is executed as a stand-alone script (via the Scripts palette) compared to when it is executed as a handler (e.g. in the `docLoaded` handler).

```
alert(app.documents.length);
var theTempDoc = app.documents.add();
theTempDoc.label = "xyz";
theTempDoc.close(SaveOptions.no);
alert(app.documents.length);
```

When executed as a stand-alone script, the two alerts will display the same value.

When executed as an event handler, the second alert will show a value one higher than the first alert – *`theTempDoc`* cannot be closed immediately. It will be closed later on, after the handler finishes.

It is allowed to call *`theTempDoc.close()`* multiple times – any additional calls will be ignored. However be careful not to introduce endless loops – for example:

```
var theTempDoc = app.documents.add();
theTempDoc.label = “xyz”;
while (app.documents.length > 0)
  app.documents.item(0).close;
```

In a stand-alone script, the above example would work correctly, but when this construct is used in a handler, *`app.documents.length`* will remain positive (because the document cannot be closed yet), and the script will loop forever.

9.7. The script engine version is global

For compatibility reasons, the InDesign CS3 script interpreter can be switched back to be compatible with InDesign CS2’s ExtendScript interpreter.

When doing so, keep in mind that this affects all script engines in InDesign CS3 – so you should put this back to whatever value you found it to be at first, after your script finishes executing.

9.8. Adornments

ExtendScript syntax is shown below, but VBScript and AppleScript syntax are analogous - see samples further down.

```
theItem.setDataStore(
  "$ADORNMENT<unique_key>$",
  <adornment>,
  useTempDataStore);
```

The last parameter (*`useTempDataStore`*) is optional, and defaults to 'false'.

`<unique_key>` should be a unique string, preferably based on your reversed domain name, to avoid clashes with other APID developers.

Adornments are a shared resource, and more than one adornment-using script might be installed. By using a unique key, clashes between multiple adornments are avoided; they will be shown concurrently.

`<adornment>` is either a string or an array. If it's a string, a simple label adornment is shown on top of the page item.

If it's an array then it contains the following elements, which are all optional except for the first element:

```
[ <label>, <pngfile>, <side>, <fillcolor>, <textcolor>, <strokecolor>,
<direction> ]
```

Only the first element *`(<label>`*) is necessary.

`<label>` can be *`null`* or a string

`<pngfile>` can be a (pathless) file name or a *`File`* object.

`<side>` is 1, 2, 3 or 4 (1=top, 2=left, 3=bottom, 4=right)

the three `<...color>` entries each are 3-element arrays with floating point RGB values from 0 to 1 (e.g. `[1.0, 0, 0]` is red)

`<direction>` is 0 (left-to-right) or 1 (right-to-left)

`<pngfile>` can be a `File` object - but it's not recommended. Using a `File` object makes the adornment dependent on the presence of a PNG file on some absolute path.

Transporting a document to another computer would nearly certainly invalidate that path, and the adornment would not display any more.

Instead, the preferred way is to use a simple file name (without path). The corresponding PNG file can then be installed anywhere below the InDesign application folder - APID will find it and pick it up.

If you're using .spln-based deployment, you would simply store your PNG files together with your .spln file in some subfolder of the Plug-Ins folder.

If you're using normal InDesign scripts, you would probably create a subfolder somewhere inside the InDesign *Scripts* folder, and store your script and your PNG files in there.

Because there's no way you can enforce or guarantee that end-users will install things the way you prefer, APID is made oblivious to the *Plug-Ins* or application folder structure - .spln files can be anywhere below the *Plug-Ins* folder, PNG files can be anywhere below the application folder,... and things will still be picked up and linked up correctly.

A drawback of such a lenient matching system is that PNG file names must be globally unique - so for widely distributed APID-based scripts or .spln files, you should use a domain name or GUID-based scheme to make sure your PNG files have a unique file name.

For example, we would use `com.rorohiko.textexporter.ignoredframe.png` as a file name, and we'd be fairly sure no-one else would have a same-name PNG file installed anywhere under the InDesign application folder.

Adornment entries are picked up by APID both from the `dataStore` and the `tempDataStore` - any entries that have a key that starts and ends with a dollar sign, and begin with `$ADORNMENT` are considered adornment entries.

`tempDataStore` entries are purposely non-persistent - this would allow you to have adornments that 'disappear' when the document is opened on a computer where your .spln is not present.

The idea is that on processing of the `docLoaded` event you would re-create the necessary `tempDataStore` entries to make your adornments display.

Then, on a computer where your spln is missing, the entries won't be re-created, and the adornments won't show up - which is what you'd typically want.

`dataStore` entries, on the other hand, are persistent - so you can create adornments that survive even if your script or .spln file is inactive or missing (as long as the APIDToolAssistant is still alive and the needed PNG files are still installed).

See

http://www.rorohiko.com/wordpress/?p=4

for some more info.

ExtendScript sample code:

```
// Add a permanent little label "Hello" in the top left
// corner of a page item 'theItem'
// Label remains after document is closed and reopened
theItem.setDataStore(
  "$ADORNMENT_com.rorohiko.kris.test1$",
  "Hello"); // uses dataStore

// Add a temporary little label "Hello" in the top left
// corner of a page item 'theItem'
// Label disappears after document is closed and reopened
theItem.setDataStore(
  "$ADORNMENT_com.rorohiko.kris.test1$",
  "Hello",
  1); // uses tempDataStore

// The two samples above use a shorthand syntax.
// The next two samples do exactly the same but
// now using the 'full' syntax.
// Add a permanent little label "Hello" in the top left
// corner of a page item 'theItem'
// Label remains after document is closed and reopened
// Uses a three-element array: label content,
// PNG file name (set to null),
// what side (1 = top)
theItem.setDataStore(
  "$ADORNMENT_com.rorohiko.kris.test1$",
  ["Hello",null,1]); // uses dataStore

// Add a temporary little label "Hello" in the top left
// corner of a page item 'theItem'
// Label disappears after document is closed and
// reopened
theItem.setDataStore(
  "$ADORNMENT_com.rorohiko.kris.test1$",
  ["Hello",null,1],
  1); // uses tempDataStore

// Next two samples show the labels at the left (side 2)
// and bottom (side 3) of the page item, and use a PNG
// element instead of a string
theItem.setDataStore(
  "$ADORNMENT_com.rorohiko.kris.test1$",
  [null,"myIcon.png",2]); // uses dataStore

theItem.setDataStore(
  "$ADORNMENT_com.rorohiko.kris.test1$",
  ["Hello","myBackground.png",3],
  1); // uses tempDataStore
```

AppleScript sample code:

```
tell application "Adobe InDesign CS3"
 set theItem to the first item of the selection
 tell theItem
   -- Add a permanent little label "Hello" in the
   -- top left corner of a page item 'theItem'
   -- Label remains after document is closed and reopened
   set data store key "$ADORNMENT_com.rorohiko.kris.test1$" value "Hello"

   -- Add a temporary little label "Hello" in the
   -- top left corner of a page item 'theItem'
   -- Label disappears after document is closed and reopened
   set data store key "$ADORNMENT_com.rorohiko.kris.test1$" value "Hello"
with use temp data store param

   -- The two samples above use a shorthand syntax. The next two
   -- samples do exactly the same but now using the 'full' syntax
   -- Add a permanent little label "Hello" in the top left corner
   -- of a page item 'theItem'
   -- Label remains after document is closed and reopened
   -- Uses a three-element array: label content, PNG file name
   -- (set to null), what side (1 = top)
   set data store key "$ADORNMENT_com.rorohiko.kris.test1$" value {
"Hello", null, 1 }

   -- Add a temporary little label "Hello" in the top left corner
   -- of a page item 'theItem'
   -- Label disappears after document is closed and reopened
   set data store key "$ADORNMENT_com.rorohiko.kris.test1$" value {
"Hello", null, 1 } with use temp data store param

   -- Next two samples show the labels at the left (side 2) and
   -- bottom (side 3) of the page item,
   -- and use a PNG element instead of a string
   set data store key "$ADORNMENT_com.rorohiko.kris.test1$" value { null,
"myIcon.png", 2 }

   -- Next two samples show the labels at the left (side 2) and
   --  bottom (side 3) of the page item,
   -- and use a PNG element instead of a string
   set data store key "$ADORNMENT_com.rorohiko.kris.test1$" value {
"Hello", "myBackground.png", 3 } with use temp data store param

 end tell

end tell
```

VBScript sample code:

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set theItem = myInDesign.Selection.Item(1)

Dim params(3)

' Add a permanent little label "Hello" in the top left
' corner of a page item 'theItem'
' Label remains after document is closed and reopened
theItem.SetDataStore "$ADORNMENT_com.rorohiko.kris.test1$", "Hello" ' uses
dataStore

' Add a temporary little label "Hello" in the top left
' corner of a page item 'theItem'
' Label disappears after document is closed and reopened
theItem.SetDataStore "$ADORNMENT_com.rorohiko.kris.test1$", "Hello", 1 '
uses tempDataStore

' The two samples above use a shorthand syntax. The next
' two samples do exactly the same but
' now using the 'full' syntax
' Add a permanent little label "Hello" in the top left
' corner of a page item 'theItem'
' Label remains after document is closed and reopened
' Uses a three-element array: label content,
' PNG file name (set to null), what side (1 = top)
params(0) = "Hello"
params(1) = null
params(2) = 1
theItem.SetDataStore "$ADORNMENT_com.rorohiko.kris.test1$", params ' uses
dataStore

' Add a temporary little label "Hello" in the top left
' corner of a page item 'theItem'
' Label disappears after document is closed and reopened
params(0) = "Hello"
params(1) = null
params(2) = 1
theItem.SetDataStore "$ADORNMENT_com.rorohiko.kris.test1$", params, 1 '
uses tempDataStore

' Next two samples show the labels at the left (side 2)
' and bottom (side 3) of the page item,
' and use a PNG element instead of a string
params(0) = null
params(1) = "myIcon.png"
params(2) = 2
theItem.SetDataStore "$ADORNMENT_com.rorohiko.kris.test1$", params ' uses
dataStore

' Next two samples show the labels at the left (side 2)
' and bottom (side 3) of the page item,
' and use a PNG element instead of a string
params(0) = "Hello"
params(1) = "myBackground.png"
params(2) = 2
theItem.SetDataStore "$ADORNMENT_com.rorohiko.kris.test1$", params, 1 '
uses tempDataStore
```